Evaluation and Optimisation of Less-than-Best-Effort TCP Congestion Control Mechanisms

Kevin Ong

This thesis is presented for the degree of Doctor of Philosophy

Murdoch
UNIVERSITY

Discipline of Information Technology, Mathematics, and Statistics
College of Science, Health, Engineering and Education
Murdoch University

Declaration

I declare that this thesis is my own account of my research and contains as its main content work which has not previously been submitted for a degree at any tertiary education institution.

Perth, May 22, 2020

Kevin Ong

Abstract

Increasing use of online software installation, updates, and backup services, as well as the popularity of user-generated content, has increased the demand for bandwidth in recent years. Traffic generated by these applications — when receiving a 'fair-share' of the available bandwidth — can impact the responsiveness of delay-sensitive applications. Less-than-Best-Effort TCP congestion control mechanisms aim to allow lower-priority applications to utilise excess bandwidth with minimum impact to regular TCP carrying delay-sensitive traffic. However, no previous study has evaluated the performance of a large number of this class of congestion control mechanisms. This thesis quantifies the performance of existing Less-than-Best-Effort TCP congestion control mechanisms, and proposes a new mechanism to improve the performance of these mechanisms with high path delay.

This study first evaluated the performance of seven Less-than-Best-Effort congestion control mechanisms in realistic scenarios under a range of network conditions in a Linux testbed incorporating wired Ethernet and 802.11n wireless links. The seven mechanisms evaluated were: Apple LEDBAT, CAIA Delay-Gradient (CDG), RFC6817 LEDBAT, Low Priority, Nice, Westwood-LP, and Vegas. Of these mechanisms, only four had existing implementations for modern operating systems. The remaining three mechanisms — Apple LEDBAT, Nice, and Westwood-LP — were implemented based on published descriptions and available code fragments to facilitate this evaluation.

The results of the evaluation suggest that Less-than-Best-Effort congestion control mechanisms can be divided into two categories: regular TCP-like mechanisms, and low-impact mechanisms. Of the low-impact mechanisms, two mechanisms were identified as having desirable performance characteristics: Nice and CDG. Nice provides background throughput comparable to regular TCP while maintaining low queuing delay in low path delay settings. CDG has the least impact on regular TCP traffic, at the expense of reduced throughput. In high path-delay settings, these reductions to throughput experienced by CDG are exacerbated, while Nice has a greater impact on regular TCP traffic.

To address the very low throughput of existing Less-than-Best-Effort congestion control mechanisms in high path-delay settings, a new Less-than-Best-Effort TCP congestion control algorithm was developed and implemented: Yield TCP. Yield utilises elements of a Proportional-Integral controller to better interpret and respond to changes in queuing delay to achieve this goal while also reducing the impact on regular TCP traffic over TCP-like mechanisms. Source code for the implementation of Yield developed for this research has also been made available.

The results of evaluating Yield indicate that it successfully addresses the low throughput of low-impact Less-than-Best-Effort mechanisms in high delay settings, while also reducing the impact on foreground traffic compared to regular TCP-like congestion control mechanisms. Yield also performs similarly to Nice in low delay settings, while also achieving greater intra-protocol fairness than Nice across all settings. These results indicate that Yield addresses the weaknesses of Nice and CDG, and is a promising alternative to existing Less-than-Best-Effort congestion control algorithms.

Contents

List of Tables

Publications

This paper was used as a preliminary test for many elements of the methodology described in Chapter 3.

> K. Ong, D. Murray, and T. McGill, “Large-Sample Comparison of TCP Congestion Control Mechanisms over Wireless Networks," in 30th IEEE International Conference on Advanced Information Networking and Applications Workshops, Crans-Montana, Switzerland, 2016.

The following two papers present the results of experiments used in Chapter 4. These papers also served to adapt the methodology used for regular TCP congestion control to suit the evaluation of Less-than-Best-Effort TCP. Several experiments were subsequently refined and repeated.

> K. Ong, S. Zander, D. Murray, T. McGill, “Experimental evaluation of Less-than-Best-Effort TCP congestion control mechanisms," in 42nd IEEE Conference on Local Computer Networks (LCN), Singapore, 2017.
>
> K. Ong, S. Zander, D. Murray, and T. McGill, “Experimental Evaluation of Less-than-Best-Effort TCP over 802.11 Wireless Networks," in 23rd Asia-Pacific Conference on Communications (APCC), Perth, Australia, 2017.

Acknowledgements

Thanks to Dave and Sebastian, for helping shape the direction of the project and for their technical advice. And to Tanya, who helped me understand the research process and guided me through the myriad of challenges that arose throughout my candidature.

To Mike, Terry, Mark, James and the rest of my colleagues and peers who kept me motivated to press on and provided sounding boards throughout the process. To my friends and family, for supporting me through this lengthy endeavour. To Oscar, for lifting my spirits at the end of the many long days on campus.

To Graeme, Paul, and the rest of the team at VCI who gave me a glimpse of life beyond academia. And Spencer, for helping me develop the skills I'll need to get there.

Introduction 1

1.1 Overview

The volume of network traffic generated by remote file backups and software updates can have negative impacts on the responsiveness of competing network-based transactions. Historically, these activities were scheduled to occur at times of low network usage. However, use of file synchronisation and sharing services, video streaming, and the shift towards online software distribution cause large volumes of network traffic to be generated during peak usage times [1].

Simultaneously, user-generated audio and video content has become increasingly popular due to platforms such as YouTube, Twitch, and Ustream enabling users to easily distribute content to large audiences [2]. The sharing of high-quality photos to social networks has also become more prevalent. These applications have resulted in increased competition for download and upload bandwidth [1], which can also impact the responsiveness of web browsing and interactive applications.

Traffic generated by these non-critical applications may present a compelling use-case for Less-than-Best-Effort (LBE) TCP congestion control mechanisms, which aim to permit large transfers to take place without impacting interactive applications by utilising excess network bandwidth [3], [4]. Several studies have proposed different schemes for LBE congestion control [5]–[9].

Despite the existence of these proposed LBE congestion control schemes, only one study has evaluated the performance of a wide range of mechanisms [4] to date. As such, the performance of these mechanisms is not well understood and this study seeks to address this issue.

The remainder of this chapter is structured as follows. Section 1.2 outlines the purpose and basic operation of TCP. Section 1.3 describes the principles behind the operation of TCP congestion control, while Section 1.4 briefly describes the differences between LBE congestion control and regular TCP. Section 1.5 describes gaps in knowledge in the domain of LBE congestion control. Section 1.6 presents the objectives of this study, while Section 1.7 provides an outline of the research. Finally, Section 1.8 describes the structure of this dissertation.

1.2 Transmission Control Protocol

Transmission Control Protocol (TCP) specifies an end-to-end communications protocol designed to ensure the reliable and ordered delivery of data [10]. Due to this reliability, TCP is employed in a wide range of applications such as web browsing, email, file transfer, and remote management. TCP is used in a significant portion of Internet-based transactions. TCP provides reliability through the use of acknowledgements (ACKs) for all transmitted data.

Each byte transmitted using TCP is assigned a sequence number relative to the number of bytes transmitted since the session began [10], [11]. These sequence numbers are used by the sender to monitor the status of all transmitted data, and determine which bytes have been acknowledged. Any data not acknowledged within the Retransmission Timeout (RTO) period must be retransmitted by the sender [11]. This process continues until an acknowledgement is received for these bytes (or the maximum number of retransmission attempts is reached).

Extensions to the original version of TCP allow for the use of Duplicate Acknowledgements (DUPACKs) to trigger the retransmission of a TCP segment through a technique known as 'fast retransmission' [11]. DUPACKs can also be used in conjunction with the Selective Acknowledgement (SACK) extension to request the immediate retransmission of multiple missing segments.

1.3 TCP Congestion Control

Following the Internet congestion collapse in 1986, Jacobson [12] proposed an algorithm to limit the rate at which TCP could transmit data over the network. This algorithm was intended to serve two purposes: preventing TCP senders from introducing congestion to the path, as well as alleviating the need to drop packets in response to congestion. TCP congestion control fulfils these purposes by specifying the number of bytes that can be transmitted before an acknowledgement is required: this is referred to as the congestion window [11], [12].

Under TCP congestion control, a sender that reaches number of bytes specified by the congestion window must cease transmission until another acknowledgement arrives [11]. To ensure that the available bandwidth is fully utilised, TCP congestion control adjusts the size of the congestion window during the lifetime of a connection [12]. The size of these adjustments is determined by one of two algorithms: slow start, and congestion avoidance [11], [12].

The slow start algorithm is employed by TCP to quickly increase the transmission rate to utilise the bandwidth available on the network [12]. In slow start, the sender begins with a predetermined initial congestion window [11]. As the sender begins transmitting data and receiving acknowledgements, the congestion window is incremented until the first instance of packet loss is observed or the slow start threshold is exceeded.

Once the congestion window size equals or exceeds the slow start threshold, TCP enters a state known as congestion avoidance [11], [12]. In this state, TCP makes smaller adjustments to the congestion window size based on changes to the available bandwidth.

Historically, TCP utilised packet loss as an indication of network congestion and available bandwidth [12]. However, this approach only allows TCP to detect congestion that has already occurred. More recently, congestion control mechanisms have used estimation of queuing delay to proactively determine whether network congestion is present [11].

1.4 Less-than-Best-Effort Congestion Control

LBE congestion control is a sub-category of congestion control mechanisms that seek to minimise the impact of lower priority transactions on regular TCP transfers [13]. These mechanisms should maximise the use of available bandwidth while no competing foreground traffic is present, while quickly conceding bandwidth to traffic managed by more aggressive TCP congestion control schemes such as NewReno and CUBIC.

LBE mechanisms achieve this goal by more aggressively reducing the size of the congestion window in the presence of this foreground traffic. The detection of foreground traffic is based on an estimation of the current delay, utilising the same techniques as delay-based TCP congestion control [3], [5].

Two LBE congestion control mechanisms have attracted research and practical interest: TCP Low Priority (LP) [3], and Low Extra Delay Background Transport (LEDBAT) [14], [15]. LEDBAT has been included in recent versions of Windows

10 and Windows Server 2016 [16], as well as some versions of macOS [17]. Additional mechanisms have been proposed [5], [7]–[9], [18], [19], but few have been implemented for — or integrated into — modern operating systems.

1.5 Research Problem

Results of user satisfaction and marketing studies have found that reduced page loading times increase user engagement [20] and decrease the rate of cart abandonment for eCommerce websites [21]. Conversely, engagement decreases with loading delays of as little as 100 ms [21], with satisfaction decreasing by up to 36.5% when page loading times increase by 2 seconds [20], [21].

Recommendations arising from these studies focus on improving website performance through page and infrastructure optimisations [20], [21], but the results of previous evaluations [5], [22] suggest that similar loading speed improvements could be achieved through the use of LBE mechanisms.

Evaluations of LBE congestion control techniques have typically been carried out as part of the proposal of new mechanisms, and have included a limited range of other mechanisms [5], [7]–[9], [18], [19]. Only one prior study has examined the performance of several existing LBE congestion control techniques but has focused on simulated experiments [4]. As these simulated networks are not representative of the performance of congestion control mechanisms in real-world use [23], [24], the performance of these mechanisms in live hosts needs to be better understood.

1.6 Research Objectives

This research aims to address the limited understanding of LBE congestion control performance by quantifying the performance of existing mechanisms on live hosts under a range of scenarios. In addition to the performance of individual mechanisms, the compromise between minimising disruption to foreground traffic and acceptable throughput is also examined.

In addition, this research aims to develop a new LBE congestion control algorithm informed by the results of evaluating existing LBE mechanisms. This understanding is intended to provide a greater understanding of the limitations of existing mechanisms, allowing areas of potential improvement to be identified.

These aims will be achieved through the fulfilment of the following objectives:

1. Identify and implement a range of recent and promising LBE congestion control algorithms for the Linux kernel.
2. Carry out an evaluation of LBE congestion control mechanisms on emulated networks.
3. Identify limitations in existing LBE congestion control algorithms.
4. Develop and evaluate a new LBE congestion control mechanism optimised to address weaknesses in existing algorithms.

1.7 Research Approach

To achieve the aims of this research, two studies were carried out: an evaluation of existing LBE congestion control algorithms, as well as the development and evaluation of a new LBE algorithm.

The first study of this research evaluated the performance of seven LBE congestion control mechanisms in different scenarios in a Linux testbed incorporating wired Ethernet and 802.11n wireless links. To facilitate this evaluation, a number of previously proposed LBE congestion control mechanisms were implemented for a recent version of the Linux kernel.

The results of the evaluation were used to identify limitations of existing algorithms that could be improved by future LBE congestion control algorithms. A new LBE congestion control algorithm was then developed and implemented based on the findings of the initial evaluation. This algorithm was also evaluated in comparison to the most promising of the existing algorithms.

1.8 Structure

The remainder of this thesis is structured as follows. Chapter 2 presents a review of the existing literature relating to LBE congestion control mechanisms. This chapter examines existing LBE congestion control mechanisms, the methodologies employed to evaluate these mechanisms, and the results of any prior evaluations of these mechanisms.

Chapter 3 describes the methodology developed for testing LBE congestion control mechanisms, as well as providing additional detail regarding the evaluation carried out in this study. Chapter 4 presents the results and a detailed analysis of the performance of existing LBE congestion control mechanisms.

Chapter 5 describes Yield TCP, a new LBE congestion control mechanism developed to address the poor throughput of LBE congestion control in high delay settings. Chapter 5 also presents the results of the evaluation of Yield and compares its performance to existing mechanisms.

Finally, Chapter 6 concludes with a summary of the study's findings, its implications for research and practice, and possible directions for future work.

2 Literature Review

2.1 Overview

While the original TCP Congestion Control algorithm used packet loss as an indication of network congestion, this approach only allows TCP to detect congestion that has already occurred. As such, LBE congestion control mechanisms typically utilise the delay-based approach proposed by Jain [25] to proactively identify impending congestion events.

LBE congestion control is a relatively new sub-category of TCP congestion control mechanisms. However, a number of mechanisms have already been proposed in the literature [5], [7]–[9], [18], [19]. These mechanisms are described in detail in Section 2.2.

The remainder of this chapter is structured as follows. Section 2.2 describes the operation of LBE congestion control and examines the mechanisms proposed in the literature. Section 2.3 discusses methodologies used by prior studies evaluating the performance of TCP congestion control mechanisms, while Section 2.4 makes comparisons between existing mechanisms based on the findings of these studies. Finally, Section 2.5 presents a summary of the literature examined in this chapter.

2.2 Less-than-Best-Effort Congestion Control

While LBE congestion control is a relatively new subcategory of congestion control mechanisms, these algorithms share many similarities to delay-based congestion control [3], [5], [25], [26]. Both classes of mechanisms detect increasing queuing delay to indicate the presence of competing traffic or network congestion, and respond proactively to these indications.

The delay-based approach is based on the estimation of queuing delay, either as round trip time (RTT) or one-way delay (OWD) [3], [25], [26]. This approach is in contrast to mechanisms based on the original TCP congestion control algorithm which must wait until network congestion has occurred before reacting [12].

Delay-based mechanisms have been shown to provide lower throughput than mechanisms that use packet loss as an indicator of network congestion [27], which led to this approach not being favoured for regular TCP traffic where high throughput has been desirable. However, this property is desirable for LBE congestion control algorithms which aim to minimise their impact on regular TCP traffic.

LBE congestion control algorithms also incorporate more conservative rules regarding the growth of the congestion window ($cwnd$), as well as more aggressive behaviours to reduce $cwnd$ to minimise the impact on regular TCP transfers [28]. Table 2.1 presents a summary of key LBE congestion control algorithms. These algorithms are described in greater detail in the subsections that follow.

2.2.1 Vegas

One of the first delay-based congestion control mechanisms, TCP Vegas, was proposed by Brakmo *et al.* [26] in 1994. Although based on Reno, Vegas employs a more proactive approach to congestion avoidance by attempting to predict impending congestion events before they occur and reduce the congestion window size accordingly. Vegas is able to predict congestion events by estimating the expected throughput for the TCP connection, and comparing the estimate against the actual throughput. This approach is supported by three techniques: more proactive responses to duplicate acknowledgements, predictive indications of congestion events, and a more conservative slow start.

Upon receipt of a duplicate acknowledgement, Vegas checks the RTT for the segment for which the duplicate acknowledgement was received [26]. Should the current RTT for the segment exceed the RTO, Vegas will immediately retransmit the segment. Further, segment losses occurring within a single RTT will only trigger a single window reduction. For duplicate acknowledgements where the segment RTT has not exceeded the RTO, Vegas will respond in the same way as Reno.

To ensure that congestion events are detected as early as is feasible, Vegas compares the expected throughput against the actual throughput during each RTT [26]. This estimate is calculated based on the current $cwnd$ and minimum of RTT measurements ($baseRTT$) using the formula:

$$e = \frac{cwnd}{BaseRTT} \tag{2.1}$$

Mechanism	Technique	$cwnd$ **increase**	$cwnd$ **decrease**
4CP	Probability	$cwnd + 1$ per RTT	$max(cwnd - \frac{1}{tarp \cdot cwnd}, mincwnd)$
Low Priority	One-way delay	$cwnd + 1$ per RTT	$cwnd/2$ on OWD over threshold or on packet loss $cwnd = 1$ on subsequent loss during inference phase
LEDBAT	One-way delay	$cwnd + (GAIN \cdot off\_target \cdot newly\_acked + ALLOWED \cdot MSS/cwnd)$	
Nice	Round trip time	$cwnd + 1$ when below target $cwnd$ $cwnd + 1$ per RTT when not enough delay estimates available	$cwnd - 1$ when above target $cwnd$ $cwnd/2$ when congestion events $>0.5 \cdot cwnd$
Westwood+LP	Round trip time	$cwnd + 1$ per RTT	$cwnd = max(bw\_est * rtt\_min, 2)$ when queue length exceeds threshold or on packet loss
ImTCP-bg	Round trip time	$cwnd + 1$ per RTT	$\frac{cwnd - rtt\_min}{rtt\_avg}$
Eclipse	One-way delay	$cwnd + \frac{off\_target \cdot newly\_acked \cdot MSS}{cwnd}$	
Apple LEDBAT	One-way delay	$cwnd + 1$ per RTT	$cwnd - 1/8 \cdot cwnd$ when over delay target
CDG	Delay-gradient	$cwnd + 1$ per RTT	$cwnd/2$ when $n \geq$ backoff probability

Table 2.1.: Summary of Less-than-Best-Effort congestion control algorithms

The difference between these estimates and measurements is used to determine whether the network is in the process of becoming congested [26]. This approach to congestion detection allows Vegas to proactively resize the congestion window, rather than waiting for congestion to occur as with loss-based congestion control mechanisms.

Vegas also includes a modification to the Reno slow start algorithm, under which the congestion window is expanded every second RTT [26]. However, this modification is omitted from current implementations.

TCP Vegas has been included as an alternate congestion control mechanism in the Linux kernel since version 2.6.13, although it is not enabled by default [11]. However, the DD-WRT router firmware has utilised TCP Vegas as its default congestion control mechanism since version 24 Service Pack 2.

2.2.2 Nice

One of the first proposed LBE congestion control mechanisms, Nice [5] is an extension of Vegas that adds a multiplicative decrease in response to increasing delay. This extension results in $cwnd$ being halved when the number of congestion events detected by TCP exceeds a $fraction$ of the current congestion window size (in TCP segments) — set to 50% by default — in any given RTT.

Nice defines a congestion event to be any occurrence where the current RTT exceeds the value of $(1 - threshold) \cdot (rtt\_min + threshold \cdot rtt\_max)$, where rtt_min and rtt_max represent the minimum and maximum RTTs for a single per-RTT congestion avoidance cycle [5]. The value of $threshold$ specifies the level of tolerable congestion and has a default value of 0.2.

In addition, Nice introduces the ability for TCP to reduce the effective size of the congestion window below 1 [5]. This fractional congestion window is implemented by permitting TCP to release new segments only every n RTTs, where n represents the denominator of the congestion window.

2.2.3 Low Priority

Developed by Kuzmanovic and Knightly [3], LP is another early LBE mechanism. LP evaluates the OWD — measured using TCP timestamp fields — experienced by outbound traffic and rapidly reduces the size of $cwnd$ in response to increasing delay. To facilitate these rapid size reductions of $cwnd$, LP implements two new window reduction behaviours: a multiplicative decrease in response to increasing delay, and a reduction of $cwnd$ to a single segment.

The LP multiplicative decrease is similar to that of Nice, in that it is applied in response to increasing delay [3]. However, application of this multiplicative decrease rule places LP in an inference state. The inference state is used to determine whether the increased delay is a result of the presence of competing network traffic. LP remains in the inference state for $3 \cdot (time - prev\_time)$, where $time$ and $prev\_time$ represent the current time and timestamp for the last segment transmitted, respectively.

If a second delay-based congestion event is experienced while in the inference state, LP will then reduce the $cwnd$ to a single segment. LP makes no attempt to increase the window size while in this state. If no additional congestion events are detected before the expiry of the inference state, LP will apply an additive increase rule to $cwnd$.

The original proposal of LP specified that this rule should increase $cwnd$ by $\frac{1}{cwnd}$ each RTT [3], while the current implementation of LP applies NewReno's additive increase rule when not in the inference state [29].

LP has been available as an alternate congestion control mechanism in the Linux kernel since version 2.6.18.

2.2.4 Competitive and Considerate Congestion Control Protocol

Competitive and Considerate Congestion Control Protocol (4CP) differentiated itself from Nice and LP by being designed to ensure that background traffic is able to retain some bandwidth in the presence of competing foreground traffic [30]. 4CP decision-making is based on a moving average of $cwnd$ size, as well as the probability of packet loss occurring.

In operation, 4CP compares the current probability of a packet loss event (p) against a target probability ($tarp$) [30]. The value of $tarp$ can be a fixed target or calculated dynamically during operation. When calculated automatically, $tarp$ is based on the difference between a historical average of $cwnd$ and its current value, using the formula:

$$tarp = tarp + \alpha(f(tarp) - cwnd))/cwnd \tag{2.2}$$

where $f(tarp)$ represents a moving average of the $cwnd$, while α is a smoothing factor applied to historic average.

When multiple duplicate acknowledgements are received, a reduction is applied to $cwnd$. While a virtual $cwnd$ is used to control this change, the reduction can be simplified as the lesser of $mincwnd$ — a predetermined lower boundary for $mincwnd$ — and the following formula:

$$cwnd = max(cwnd - \frac{1}{tarp \cdot cwnd}, mincwnd) \tag{2.3}$$

When no cross traffic is detected, an additive increase of $\frac{1}{cwnd}$ is applied.

An implementation of 4CP was developed for Windows Vista [28], but has not been made publicly available.

2.2.5 Westwood Low Priority

Westwood Low Priority (Westwood-LP) [6], [18] is an LBE congestion control mechanism based on Westwood [31], which adds an Early Window Reduction (EWR) mechanism to reduce the size of $cwnd$ prior to packet loss occurring.

Westwood-LP determines when EWR should be triggered using a virtual queue length, calculated using the formula below, where bw_est and rtt_min represent the Westwood bandwidth estimate and minimum RTT, respectively [6], [18]:

$$q\_len = cwnd - bw\_est \cdot rtt\_min. \tag{2.4}$$

When this queue length exceeds the EWR threshold, $cwnd$ is reset to the value of $bw\_est \cdot rtt\_min/MSS$, where MSS is the maximum allowable segment size [18]. The threshold used to determine when $cwnd$ should be reset is calculated dynamically [6], [18]. Initially, the EWR threshold calculation was based on the formula [18]:

$$ewr\_thresh = M \cdot (1 - \frac{delay\_min}{delay\_avg} + \delta) \tag{2.5}$$

where M represents the maximum allowable queue length (with a default value of 3), while $delay\_min$ and $delay\_avg$ are exponentially weighted moving averages of the minimum and average RTTs at the time EWR is applied, respectively [18]. Finally, δ is a smoothing factor described as being set to $3 \cdot 10^{-6}/(rtt - rtt\_min)$ where rtt represents the most recent delay estimate.

More recently, the EWR threshold has been described as [6]:

$$ewr\_thresh = M \cdot (1 - \frac{rtt}{rtt\_loss}) \cdot (1 - \frac{delay\_min}{delay\_max}). \tag{2.6}$$

The notable additions to the revised formula are the use of rtt and rtt_loss, which represent the current RTT and a moving average of RTT when packet loss events occur [6]. This addition is used by Westwood-LP to estimate and consider when packet loss may occur in applying EWR. Also notable is the change from $delay\_avg$ to $delay\_max$; a moving average of the maximum observed RTT between each EWR.

2.2.6 Inline measurement TCP-Background

Inline measurement TCP-Background (ImTCP-bg) is an extension to the Inline measurement TCP (ImTCP) proposal by Man, Hasegawa, and Murata [32] that aimed to provide better link utilisation than Nice and LP [22].

Similar to Westwood, ImTCP-bg utilises an RTT-based bandwidth estimation approach to detect the presence of congestion and competing foreground traffic [22]. These bandwidth estimates are based on the amount of data transferred during measurement intervals [32], [33]. ImTCP-bg groups transmission into packet streams to facilitate these measurements.

ImTCP-bg uses an exponentially weighted moving average of the bandwidth averages, in addition to the minimum RTT observed for that connection, to determine the maximum allowable size of $cwnd$ [22]. The minimum RTT is also used to identify congestion events. Congestion events are defined as when $\frac{rtt\_avg}{rtt\_min}$ exceeds a specified threshold (δ) set to 1.2 by default. When congestion events are detected, the cwnd is reduced by a factor of $\frac{rtt\_min}{rtt\_avg}$.

An implementation of ImTCP-bg is available for version 4.1 of FreeBSD [34].

2.2.7 Low Extra Delay Background Transport

LEDBAT, originally referred to as μTorrent Transfer Protocol, was first proposed as a TCP-like congestion avoidance scheme over UDP for peer-to-peer file sharing using BitTorrent [14]. This proposal was later translated into a TCP congestion control mechanism and standardised as RFC6817 [15].

Similar to LP, LEDBAT measures OWD to determine the amount of queuing delay experienced by transmitted segments [3], [15]. However, LEDBAT attempts to keep queuing delay below a specified target value, typically set to 100 ms [15]. However, earlier drafts specified a default delay target of 25 ms [35].

LEDBAT differs from existing LBE congestion control mechanisms in that the size of $cwnd$ is controlled based upon the deviation from the delay target [15]. Specifically, $cwnd$ is altered based on:

$$cwnd = cwnd + \frac{GAIN \cdot off\_target \cdot newly\_acked + ALLOWED \cdot MSS}{cwnd} \tag{2.7}$$

where off_target represents the difference between the current delay and delay target, while $newly\_acked$ is the number of segments that have just been acknowledged. $GAIN$ sets the rate at which LEDBAT responds to changes in queuing delay, while $ALLOWED$ specifies the maximum amount by which LEDBAT can increment the $cwnd$. Both constants are set to 1 by default.

While LEDBAT was intended to concede bandwidth in the presence of competing transfers [15], subsequent studies have found that it can be overly aggressive when competing with regular TCP flows [4] as well as inducing additional queuing delay [8]. These issues have led to the development of newer mechanisms such as Eclipse [8] and FLOWER [9] in an attempt to address them.

LEDBAT has been included in Windows 10 as an experimental option since build 14393 and Windows Server 2016 [16]. An implementation of LEDBAT for recent versions of Linux is also available [36], but has yet to be officially integrated into the kernel.

Apple LEDBAT

A further implementation of LEDBAT has been published as part of the Apple XNU kernel [17], but differs from the algorithm specified in RFC6817. This derivative of LEDBAT replaces the proportional increase and decrease algorithm specified in RFC6817 with an additive increase and multiplicative decrease scheme [17].

The use of an additive increase multiplicative decrease scheme was also proposed by Carofiglio *et al.* [37]. However, Apple's implementation applies a $\frac{1}{8}$ reduction to $cwnd$ when over target, rather than the 40% reduction to $cwnd$ proposed in [37]. The default queuing delay target remains at 100 ms, while the maximum allowed increase ($ALLOWED$) is increased from 1 to 8.

This implementation has been made available for the Apple XNU kernel, but is not included in current versions of macOS.

2.2.8 CAIA Delay-Gradient

CAIA Delay-Gradient (CDG) [19] differs from other delay-based congestion control algorithms by examining delay trends, rather than instantaneous queuing delay, to detect congestion. While not originally designed as a LBE congestion control mechanism, CDG has been found to be a viable congestion control mechanism for LBE traffic [38].

To examine trends in delay, CDG considers the change in the minimum and maximum delays between the current and previous RTTs. These measurements are then used to calculate moving averages of the minimum and maximum delays to reduce the impact of short-term fluctuations on the performance of CDG.

The growth and reduction of $cwnd$ is determined by a backoff probability, calculated based on the moving average of the delay gradient and a configurable scaling factor [19]. When a randomly generated number, between 0 and 1, exceeds the backoff probability CDG will halve $cwnd$. For RTTs where $cwnd$ is not reduced, an additive increase is applied and $cwnd$ increases by 1.

In addition to using delay-gradients, CDG uses two techniques to allow traffic it manages to obtain a fair share of bandwidth when competing with loss-based congestion control [19]. The former, referred to as ineffective backoff detection, is used to detect when multiple successive $cwnd$ reductions have failed to reduce delay and prevents CDG from initiating further reductions. Additionally, CDG tracks a second window variable — referred to as the shadow window — which is used to reverse a gradient-based $cwnd$ reduction following a loss-based congestion event.

The Linux implementation of CDG, developed by Jonassen [39], implemented two notable changes from the original proposal (and FreeBSD implementation). This implementation substitutes the traditional slow start growth behaviour for a modified version of Hybrid Slow Start [40], [41]. This modification results in CDG exiting the slow start phase earlier than its FreeBSD counterpart [39], [41].

The Linux implementation also replaces the scaling factor used to influence the probability that a $cwnd$ reduction will be applied with the backoff factor. The backoff factor serves as a modifier for the $cwnd$ reduction probability, but provides greater granularity [39]. However, this change otherwise has no impact on the operation of the algorithm.

CDG has been included as an alternate congestion control mechanism in the FreeBSD kernel since Revision 252951, as well as in the Linux kernel from version 4.2.

2.2.9 Eclipse

Eclipse was developed to address concerns regarding the aggressiveness of LEDBAT, primarily by shifting from the static delay target used by LEDBAT to an adaptive target [8]. Like LP and LEDBAT, Eclipse utilises estimates of OWD to measure queuing delay [3], [8], [15].

The Eclipse adaptive delay target is a fraction of the average queuing delay observed [8]. Specifically, this queuing delay target is calculated as:

$$target = \beta \cdot (s\_max - s\_min) + s\_min \tag{2.8}$$

where s_min and s_max are moving averages of the minimum and maximum queuing delays (which can be overwritten by values lower and higher than the averages, respectively), while β represents the early congestion indication threshold (effectively the percentage of average queuing delay that indicates network congestion) [8].

The moving averages used in determining $target$ replace the concepts of base delay and current delay used by LEDBAT and are updated at regular intervals [8]. These intervals are referred to as ‘adaptation intervals’ with the length of each adaptation interval calculated as the time difference between the observation of the minimum and maximum delay values for the current interval.

The deviation from the queuing delay target is then used to determine the extent to which $cwnd$ should be adjusted based on the following calculation:

$$cwnd = cwnd + \frac{off\_target \cdot newly\_acked \cdot MSS}{cwnd}. \tag{2.9}$$

2.3 Evaluations of Less-than-Best-Effort Congestion Control

Only one prior study has evaluated several existing LBE congestion control mechanisms [4]. Other evaluations of LBE congestion control have been conducted, but typically in support of a newly proposed algorithm [5], [7]–[9], [18], [19]. As such, the scale of these evaluations has typically been limited, particularly in the number of competing algorithms examined.

In addition, previous evaluations of LBE congestion control have relied primarily on simulated experiments, and focused on the examination of congestion window behaviours rather than broader performance implications such as impact on foreground traffic and throughput.

Simulated experiments allow researchers to evaluate modifications to TCP congestion control relatively quickly and under a wide range of scenarios that may not be possible to easily reproduce using emulation or production testing [23]. However, many aspects of the methodologies employed varied significantly — likely influenced by the typical network infrastructure at the time. These variations introduce significant challenges in comparing the findings of each study. These methodologies and their limitations of previous studies are examined in greater detail in the subsections that follow.

2.3.1 Testing Environments

Evaluations of LBE congestion control have primarily utilised simulated networks — using `ns-2` — for data collection [3], [8], [37]. This approach allows researchers to evaluate modifications to TCP congestion control in a highly controlled environment. Use of simulated networks also allows for testing under a wide range of scenarios that would be difficult to reproduce using emulation or real-world testing [23]. As such, simulated experiments have often been favoured when testing a newly developed algorithm [5], [8], [9], [18], [19].

However, the flexibility of network simulators requires a series of assumptions be made with regards to the network topology, and the quantity and timing of network traffic [23]. Each of these factors can lead to results that are not representative of TCP performance over real-world networks such as the Internet [24].

Network simulators also requires separate implementations of the congestion control mechanism [23], [42]. The performance of these simulator-based implementations has previously been identified to differ from their Linux-based counterparts [42]. While TCP-Linux – an extension to ns-2 — can be used to load Linux-based congestion control modules, performance differences remain between the two platforms (albeit reduced) [42]. As such, the results of simulation-based studies should be considered indicative rather than authoritative [23].

Some previous studies have sought to address the limitations of simulated experiments by evaluating LBE congestion control over the Internet [5], [6], [30]. Internet-based testing allows researchers to remove the possibility that assumptions made in designing the network model will introduce biases into the results [23]. This approach also requires the use of live hosts and by extension, real-world implementations of congestion control mechanisms. However, results of Internet-based testing are likely to be inconsistent and difficult to reproduce due to changes to the routing and topology of the Internet [43].

As a result, large numbers of experiments are needed to demonstrate that results of Internet-based experiments can be reproduced. Whether such results are representative of performance over the broader Internet must also be considered. To address this issue, large service providers run these experiments on live systems generating millions of samples [44], [45].

Another alternative used in evaluating LBE congestion control algorithms are emulated testbeds [19], [38]. Similar to simulated networks, these experimental testbeds are used to model network topologies, including those which may be difficult to gain access to in the real-world [23].

While network testbeds carry a similar potential for unrepresentative network topologies as with simulation [23], the use of hosts with real-world operating systems ensures that congestion control modules respond to network conditions as they would on live hosts. These testbeds also allow for greater control over cross-traffic than is possible in Internet-based testing, which leads to higher reproducibility of results and allows for deeper analyses of congestion control algorithm behaviours.

2.3.2 Evaluation Metrics

While a range of metrics have been used in the evaluation of LBE congestion control, three have been more commonly used than the others. These common metrics are the time required to download a file [3], [5], [6], [22], [30], throughput [3], [5], [8], [18], [22], [30], and the behaviour of $cwnd$ using the LBE mechanism [3], [4], [8], [9], [30].

One of the most commonly used metrics in previous evaluations of LBE congestion control has been transaction time for the foreground transfer: the time required for a foreground transfer to complete [3], [6]. This metric has also been referred to as response time [3], [5], [30]. Transaction time has been used to measure the impact of competing LBE transfers on short foreground traffic, like web pages and other small downloads [3], [18]. Through analysis of this impact, researchers can infer how LBE congestion control will interact with regular TCP in scenarios similar to real-world use. However, this metric has been omitted from more recent evaluations of LBE congestion control in favour of others (such as $cwnd$) which permit examination of algorithm micro-behaviours [8], [9].

Despite the primary goal of LBE congestion control being to reduce impact on foreground traffic, these algorithms should permit the use of available throughput in the absence of competing traffic [13]. Several studies have recorded the size of $cwnd$ to evaluate the ability of LBE algorithms to achieve this goal [3], [4], [8], [9], [18], [46]. The size of $cwnd$ is typically measured both while the LBE-managed background traffic is transferring data in isolation, as well as while in competition with foreground traffic. Carofiglio *et al.* [4] and Trang *et al.* [9] also consider the queue length of the bottleneck link. Other studies have also used throughput of the foreground and background traffic on a shorter timescale as a substitute for $cwnd$ to examine algorithm micro-behaviours [5], [8], [18], while Shimonishi *et al.* [6] also used link utilisation — the proportion of available bandwidth utilised — for the same purpose.

Throughput has also been used on a more macroscopic scale to examine the performance impact of using LBE congestion control on background traffic [5], [8], [22]. In these cases, throughput for background traffic was averaged over the course of one or more experiments. Link utilisation has also been used to fulfil a similar purpose [4], [5].

In addition to yielding bandwidth to foreground traffic, LBE congestion control must still be able to fairly share bandwidth among multiple background transfers. As with evaluations of regular TCP congestion control, this property has been measured using Jain's Fairness Index [3], [4], [6]. The fairness index is a scale between 0 and 1.0, where 1.0 represents a totally fair sharing arrangement [47]. While Fairness indices could also be used to measure the ability of LBE congestion control to concede bandwidth to foreground traffic over the course of whole experiments, this approach would be less informative than other metrics like transaction time or throughput and does not appear to have been used in previous studies.

2.3.3 Evaluated Mechanisms

Despite the existence of numerous LBE congestion control mechanisms, only three have been consistently included in previous performance evalations: LP [4], [18], [22], LEDBAT [4], [8], [9], and Nice [4], [22] (albeit to a lesser degree).

Other mechanisms, such as ImTCP-bg, Westwood-LP, 4CP, and CDG do not appear to have been included in any studies evaluating the performance of LBE congestion control since the time of their initial proposals [6], [7], [19], [22]. The performance of more recently proposed mechanisms, such as Eclipse, has also yet to be extensively evaluated.

Additionally, Liu [7] and Armitage [38] only considered the performance of their proposed mechanisms (4CP and CDG, respectively) to that of regular TCP congestion control mechanisms such as Reno and CUBIC.

2.4 Less-than-Best-Effort Mechanism Performance

Performance comparisons between LBE congestion control mechanisms are challenging due to the limited number of comprehensive studies, as well as the differing approaches to evaluating these mechanisms. However, some relative comparisons can be made based on the results of the commonly evaluated LBE algorithms (LP, LEDBAT, and Nice).

In initial evaluations of LP, Kuzmanovic and Knightly [3] demonstrated that it had a minimal effect (−2.7%) on the throughput of competing TCP Reno flows when foreground transfers were active. This finding was partially supported by Tsugawa, Hasegawa, and Murata [22], who found that foreground transfers competing against LP completed in a similar amount of time as in the case where no background traffic was present, although increased download completion times of approximately 20% were observed in some experiments.

The throughput achieved by LP while competing with foreground traffic has been observed to vary, with Tsugawa, Hasegawa, and Murata [22] observing similar throughput to Reno while Shimonishi *et al.* [6] found a decrease in throughput for LP background traffic of approximately 30%. However, LP was able to utilise greater throughput than Reno when excess bandwidth was available [3].

Kuzmanovic and Knightly [3] also found that LP was able to share throughput fairly when competing over periods of greater than 1 second. However, Jain's Fairness dropped as low as approximately 0.85 when competing over shorter periods (0.1 seconds or below). These findings were supported by subsequent evaluations by Callegari *et al.* [27] and Carofiglio *et al.* [4]. Specifically, Callegari *et al.* [27] found similarly poor short-term fairness (0.84) although it improved over long-lived transfers (0.95).

In the study by Kuzmanovic and Knightly [3], LP reduced the time required to complete HTTP transactions by between 65% and 80% compared to that of Reno depending on the size of the transfer. Similar reductions were observed by Tsugawa, Hasegawa, and Murata [22], who found that LP reduced the time required to transfer web pages when compared to Reno, albeit to a lesser extent. However, these benefits were diminished as the number of simultaneous foreground transfers increased.

Like LP, initial testing of Nice demonstrated a negligible impact on competing foreground traffic, with only a small increase in foreground transfer completion time compared to the case where router-based prioritisation was used [5]. This limited impact represented a significant improvement over the use of Reno for background traffic, and was irrespective of the number of competing background transfers. Likewise, Tsugawa, Hasegawa, and Murata [22] found that foreground transfers competing against Nice completed in similar times to when no competing background traffic was present.

The throughput achieved by Nice was, however, found to be substantially lower than when Reno was used for background transfers [5], [22]. In initial testing [5], this reduction of throughput was found to be approximately 50% regardless of the number of competing background transfers. However, Tsugawa, Hasegawa, and Murata [22] found that the throughput penalties incurred by Nice ranged from 30% to 90% depending on the amount of foreground traffic present.

Nice has also been observed to have difficulty in fairly sharing available bandwidth among multiple concurrent transfers [4]. The evaluation by Carofiglio *et al.* [4] indicated that Jain's fairness indices reached as low as 0.8 in some scenarios.

In initial testing by Shimonishi *et al.* [6], Westwood-LP demonstrated reductions in foreground transfer completion times of approximately 50% when compared to Reno. However, this effect was found to be significantly diminished with short foreground transfers. Westwood-LP has not been evaluated by any studies since.

Testing of ImTCP-bg indicated that it had a negligible impact on competing foreground traffic, with foreground transfers completing in similar time as Nice [22]. However, ImTCP-bg achieved throughput more closely aligned with that achieved by LP in the same experiments.

Rossi *et al.* [46] evaluated LEDBAT using a delay target of 25 ms — as specified by early drafts of RFC6817 [35] — and found that it yielded bandwidth to Reno flows relatively quickly. This finding is supported by similar experiments by Carofiglio *et al.* [4]. Further testing using the RFC6817 delay target of 100 ms identified that LEDBAT acted more aggressively to achieve inter-protocol fairness when competing with foreground traffic.

The FreeBSD implementation of CDG has been found to reduce delay by approximately 60% when used for background traffic as compared with NewReno [38]. Testing with the Linux implementation suggested slight improvements to throughput over the FreeBSD variant [39].

While initial testing of Eclipse and FLOWER suggest that they both address the aggressiveness of LEDBAT, thereby reducing the level of self-inflicted delay, limited information is available on the degree to which these mechanisms improve the completion time of foreground flows [8], [9].

2.5 Summary

Although LBE congestion control is a relatively new sub-category of TCP congestion control algorithms, a number of mechanisms have already been proposed in the literature. Of these algorithms, three have been the most studied: LP, Nice, and LEDBAT. Other algorithms have generally only been evaluated in preliminary experiments at the time of their proposal.

Given the lack of research evaluating newer LBE algorithms, limited information is available regarding their performance characteristics. Available performance information often relies on simulated networks and implementations, which may produce different performance results compared to the performance achieved on live hosts and networks. As such, further evaluation of existing algorithms is needed.

This research addresses this limited understanding of the performance of LBE congestion control by evaluating real-world implementations of LBE congestion control algorithms in an experimental testbed. The methodology used in this evaluation is described in the following chapter.

3 Evaluation Methodology

3.1 Overview

To examine the performance characteristics of LBE congestion control mechanisms, as well as their impact on regular TCP traffic, a series of experiments was carried out in an experimental testbed over wired and wireless links. This chapter describes the experimental methodology that was developed and then utilised in the evaluation of LBE congestion control mechanisms.

This evaluation addresses the limitations of prior studies, by evaluating seven LBE congestion control mechanisms based on traffic and network profiles that typical end-users would encounter. While four mechanisms were already available, three previously proposed LBE congestion control mechanisms were implemented for a recent version of the Linux kernel; the implementations of these algorithms is described in Section 3.6.

The remainder of the chapter is structured as follows. Section 3.2 describes the equipment and networks utilised in this study. Section 3.3 describes the three traffic scenarios used to evaluate the congestion control mechanisms, while Section 3.4 characterises the test data used in these scenarios. Section 3.5 describes the metrics by which the LBE congestion control mechanisms were evaluated. Section 3.6 identifies the congestion control mechanisms included in this evaluation, and describes differences between the algorithms as originally proposed and the implementations of these mechanisms developed for this research. Section 3.7 describes the TEACUP experiment automation tool used to automate the data collection for the study, while Section 3.8 describes the tools and methodology used to analyse the results of the experiment. Finally, Section 3.9 presents a summary of the methodology used.

3.2 Experimental Setup

To examine the performance characteristics of LBE congestion control mechanisms, as well as their impact on regular TCP traffic, a series of experiments was carried out in an experimental testbed over wired and wireless links.

The topology for these experiments included three hosts, each running OpenSUSE Leap 42.1 with kernel version 4.4.15. As shown in Figure 3.1, each host PC was connected via a Gigabit Ethernet access link to its local switch. Unless specified, default networking settings were not altered.

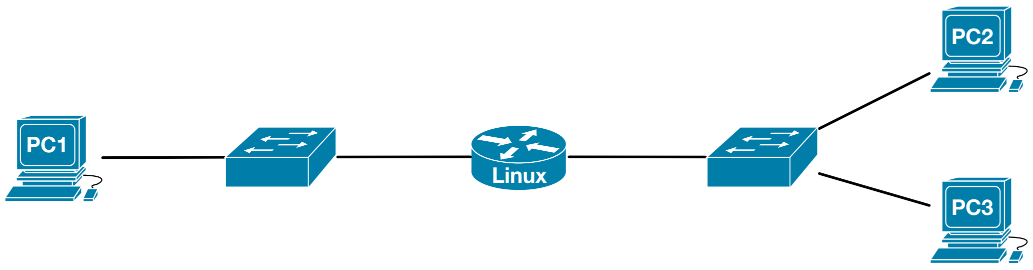


Figure 3.1.: The experimental wired network topology.

In experiments utilising wireless networks, PC1 was connected via 802.11n over 5GHz to an Ubiquiti Networks UniFi UAP AC Pro as shown in Figure 3.2. As in the wired experiments, the other host PCs were connected via a Gigabit Ethernet access link to the local switch.

The access point and PC1 were located in the same room, separated by a distance of approximately 5 m. During the experiments, PC1 recorded typical received signal strength of between -66 dBm and -69 dBm with a negotiated data rate of 300 Mbps.

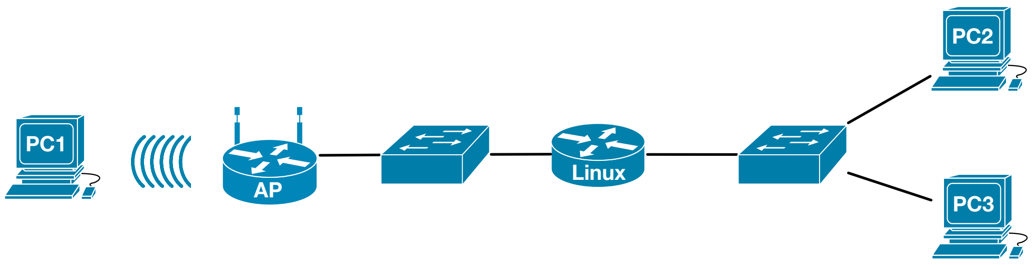


Figure 3.2.: The experimental wireless network topology.

The host PCs operated interchangeably as TCP senders and receivers, dependent on which of the three scenarios described in Section 3.3 was being considered.

A Linux-based router, running OpenSUSE 13.2 with kernel 3.17.4, was used to route packets between hosts. This router also used `netem` to emulate different bottleneck link speeds and delay settings, as described in Section 3.2.1.

The data transferred between these hosts was generated using iPerf and httperf, dependent on the desired type of TCP flow.

3.2.1 Bottleneck Link

To examine potential performance variations amongst LBE congestion control mechanisms, four bottleneck link speeds were chosen based on commonly used Internet connection options: ADSL (G.992.1), ADSL2+ (G.992.5), Fibre, and 100 Mbps Ethernet. To ensure that mechanism performance was not constrained by packet loss, the bottleneck buffer for all link speeds was set to double the bandwidth delay product (BDP) rounded up to the nearest five packets. BDP was calculated based on the downlink speed.

Prior evaluations of LBE congestion control have considered OWD of between 20–32 ms [6], [38]. For consistency with the settings used by Carofiglio *et al.* [4], experiments were conducted with fixed-path OWD of 25 ms.

Further experiments were also conducted with OWD delay values of 50 ms, 100 ms, and 175 ms. These experiments allow for the effect of higher path delay — as might be experienced in inter-continental connections — on the performance of LBE congestion control. The delay values used for these experiments were identified based on latency expected for common inter-continental routes including US-to-Europe, Asia-to-US, Asia-to-Europe [48].

Path delay for all experiments was symmetric, and is subsequently described using the round trip propagation delay for the path (RTT). Table 3.1 lists all link speed, fixed-path delay, and buffer size combinations considered in this study.

3.3 Testing Scenarios

To evaluate the performance characteristics of the LBE congestion control mechanisms, three testing scenarios were defined based on existing literature. Each of these scenarios is representative of real-world uses for LBE congestion control.

Downlink Speed	Uplink Speed	Path Delay	Buffer Size
(Mbps)	(Mbps)	(ms)	(Packets)
8	1	50	70
		100	135
		200	270
		350	470
24	1.5	50	200
		100	400
		200	800
		350	1400
50	20	50	420
		100	835
		200	1670
		350	2920
100	100	25	835
		50	1670
		100	3335
		175	5835

Table 3.1.: Bottleneck link speeds and path delay values.

3.3.1 Competing Downloads

The first scenario examined a long-lived LBE transfer that competed against one or more regular TCP flows. This scenario allows for the investigation of the impact of LBE transfers on the bandwidth available to regular TCP connections, and was also used by Liu [7], as well as Carofiglio *et al.* [4]. In this scenario, PC2 and PC3 acted as TCP senders for the foreground and background traffic, respectively. PC1 was the receiver for all connections.

3.3.2 Simultaneous Upload / Download

Given the increasing popularity of user-generated content and file synchronisation services, the second scenario examines the impact of a large file upload competing against one or more HTTP transfers. This scenario was also considered by Kuzmanovic and Knightly [3]. Similar to the previous scenario, PC2 was the TCP sender for foreground traffic directed to PC1. However, PC1 acted as the sender for the background traffic, which was received by PC3.

3.3.3 Fairness

The final scenario examined the ability of LBE congestion control mechanisms to fairly share available bandwidth without the presence of regular TCP flows. For these experiments, PC2 and PC3 initiated long-lived background transfers to PC1. The number of transfers was split evenly between the two senders.

3.4 Test Data

Given the differing objectives of LBE congestion control to those of regular TCP, two different classes of traffic were generated for this experiment: foreground traffic and background traffic. The data transferred between the testbed hosts was generated using httperf and iPerf dependent on the type of traffic required.

3.4.1 Foreground Traffic

To effectively evaluate the behaviours of congestion control mechanisms, two forms of foreground traffic were used in this evaluation. Prior studies evaluating LBE congestion control have typically used small transfers to emulate web traffic [6], [7]. As such, transfers of 2469 KiB were used to emulate foreground traffic. This volume of data corresponded to the mean data transferred by websites as of November 2016 [49]. These transfers were initiated using httperf.

Additionally, longer foreground transfers were used in a second set of experiments to evaluate the ability of LBE congestion control to share bandwidth with foreground traffic over a longer period of time. This traffic was generated using iPerf for the duration of each experiment. Similar long-lived foreground traffic has also been used in previous evaluations of LBE congestion control [9], [18], [19].

To examine the impact of using multiple transfers, the foreground traffic was split over 1, 4, and 8 TCP connections. For short foreground transfers, the total transfer was split equally across all connections. All foreground transfers were initiated simultaneously.

3.4.2 Background Traffic

As the expected use of LBE congestion control is for large file transfers, it was determined that a finite size for background file transfers was unnecessary. Background traffic was generated using iPerf for the duration of each experiment.

For all tests where foreground traffic was present, only a single background transfer was initiated. Intra-protocol fairness was examined using 2, 4, and 8 concurrent background transfers with no competing foreground traffic.

3.5 Metrics

While previous research evaluating the effectiveness of TCP congestion control mechanisms has utilised a wide range of performance metrics, four primary metrics were selected for this study: foreground transaction time, background throughput, latency, and intra-protocol fairness. These metrics are described in the following sections.

Values were calculated using a Python script developed which utilises the Wireshark packet dissection engine. The data analysis for this study is described further in Section 3.8.

3.5.1 Foreground Transaction Time

The impact of competing LBE transfers on foreground traffic was measured based on the time required for the foreground transfer to be completed. This metric has been used in previous studies, referred to as 'transaction time' [3], 'response time' [5], [7], or 'completion time' [5], [18]. This metric is referred to as 'foreground transaction time' in this thesis to avoid confusion with background transfers.

Foreground transaction times were calculated based on the times the first and last packets of TCP connections were received by the TCP receiver.

3.5.2 Throughput

Throughput describes the rate at which a network device is able to send data end-to-end over a computer network [50]. Prior studies evaluating congestion control mechanisms used throughput as a means for comparison between mechanisms [9], [27], [33].

Throughput was calculated based on the total size of all IP packets received by the TCP receiver divided by the duration of the transfer. Due to the brevity of the foreground TCP connections, throughput is only reported for the background LBE transfers.

3.5.3 Latency

Like throughput, latency or delay can be quantified through multiple units of measurement [51]. Most commonly, delay is measured as a function of RTT [19], or queue length [9].

For this study, latency is calculated based on the Wireshark RTTs for TCP segments from the sender-side packet traces. These RTTs are based on the time difference between the transmission of the original packet and the time at which the acknowledgement was received. Any RTTs calculated for retransmitted segments were excluded.

3.5.4 Fairness

Intra-protocol fairness (sometimes referred to as 'flow rate fairness', or 'RTT fairness') describes the impact of a TCP transfer on the transmission capacity of other TCP transfers operating over the same communications medium [23], [52]. A standard measure of fairness is Jain's Fairness Index [37].

The calculation of Jain's Fairness Index results in a value between 0 and 1, where 1 represents a totally fair sharing arrangement [47]. Jain's Fairness Index is calculated using the formula:

$$fairness = \frac{(\sum T_i/O)^2}{n \cdot \sum (T_i/O)^2} \tag{3.1}$$

where T_i refers to the calculated throughput (in KiB/s) for each TCP connection, while O is the portion of the available bandwidth that should be achieved by the TCP connection in a fair scenario. This share is based on the assumption that each TCP connection should receive an equal share of the overall throughput.

3.6 Congestion Control Mechanisms

Seven LBE congestion control mechanisms were selected from existing literature to be included in this study: Vegas, LP, RFC6817 LEDBAT, Nice, Westwood-LP, Apple LEDBAT, and CDG. Of these mechanisms, Nice, Westwood-LP and Apple LEDBAT were implemented for this research. These implementations are available on Github [53].

The selected mechanisms had been implemented for the Linux kernel or were determined to be feasible to implement based on existing descriptions, code snippets, and other existing TCP congestion control mechanisms. Mechanisms that would have required extensive implementation (eg. 4CP, Eclipse and FLOWER) were excluded.

The default congestion control mechanisms for Linux and FreeBSD — CUBIC and NewReno — were also included in this study, both to provide a comparison against regular TCP as well as for foreground traffic.

Where mechanisms had already been integrated into the Linux kernel — including Vegas, LP, and CDG — the existing kernel modules were loaded as needed. The implementation of RFC6817 LEDBAT developed by [36] was compiled and used in a similar manner. The following sections describe the implementations of Nice, Westwood-LP, and Apple LEDBAT developed as part of this research, noting any differences with the original mechanism proposals described in Section 2.2.

3.6.1 Nice

Given that the original Nice proposal was based on the Vegas specification [5], Nice was implemented for this study based on the Vegas implementation included with version 4.4 of the Linux kernel. As such, any deviations from the original Vegas specification remain present in this implementation of Nice. The most notable change is that current versions of Vegas do not adhere to the original proposal that increases $cwnd$ every second RTT during slow start [54].

This new implementation assumes that the trigger value for the multiplicative window reduction should reset every RTT, while the rules for incrementing and decrementing the fractional $cwnd$ (for $cwnd$ values below 1) should have an equivalent effect to changes applied when $cwnd$ is at or above 1.

This implementation also uses the minimum RTT estimate observed ($baseRTT$) to estimate propagation delay, rather than the minimum estimate for the current congestion avoidance cycle ($minRTT$) for the purpose of identifying delay-based congestion events.

3.6.2 Westwood+LP

While Westwood was based on the original proposal by Mascolo *et al.* [31], this algorithm has been superseded by Westwood+ in the Linux kernel [55]. As such, the implementation of Westwood-LP developed for this study is based on Westwood+.

Although Westwood+ itself made only minor modifications to the bandwidth estimation algorithm over the original, this implementation is referred to as Westwood+LP to distinguish it from the version based on the original algorithm.

This implementation sets initial values for the moving averages of minimum and maximum delay — $delay\_min$ and $delay\_max$ — on the first RTT, and assumes that the first EWR event will be triggered based on the instantaneous delay values, rather than weighted averages. The moving average of RTT when packet loss events occur ($delay\_loss$) is only updated when packet loss events occur.

This implementation also assumes that EWR should not be invoked while TCP is in slow start.

3.6.3 Apple LEDBAT

The implementation of Apple LEDBAT developed for this study is based on the implementation of LEDBAT for Linux as described in RFC6817. As such, it uses OWD rather than RTT to estimate delay.

The regular congestion avoidance behaviour for the RFC6817 implementation of LEDBAT was replaced with an additive increase when delay is under target, while subtracting $\frac{1}{8}$ from $cwnd$ when over the delay target.

3.7 Data Collection

Each combination of bottleneck link speed, number of foreground and background TCP connections constitutes a single experiment. For each experiment, 10 runs were carried out per congestion control mechanism. Packet traces of all traffic sent and received by the testbed hosts were captured over a 60 second period for each run.

The data collection process was automated using TEACUP 1.0 [56], which performed the necessary variation of bottleneck link speeds, path-delays, LBE congestion control modules, as well as initiating file transfers and collecting necessary files — such as packet traces — from the testbed hosts.

For experiments utilising 802.11 wireless networks, channel statistics — including the received signal strength indicator (RSSI) and channel — were logged using a Python script that polled the iwconfig utility every 0.5 seconds.

3.8 Data Analysis

The resulting packet captures were automatically processed by a custom-developed Python script, with the results stored using a SQLite database. This Python script utilises the Wireshark packet analysis engine via the `tshark` command-line interface.

The Python script reads from the packet traces generated by each testbed host (PC1–3) and reconciles the sender and receiver traces for each TCP connection based on unique IP address and port number mappings. These reconciled traces are then used to calculate throughput, latency, and fairness as described in Section 3.5. The results of these calculations are written to an SQLite database for further analysis and graphing.

Calculations of summary statistics for latency excluded retransmissions, which were identified based on the Wireshark flags:

- `tcp.analysis.retransmission`
- `tcp.analysis.fast_retransmission`
- `tcp.analysis.out_of_order`

3.9 Summary

This chapter has described the methodology used in the evaluation of LBE congestion control mechanisms for this study. The methodology ensured that LBE congestion control mechanisms were evaluated based on traffic and network profiles typical end-users would encounter.

The evaluation was carried out using an experimental testbed that included Ethernet and 802.11n wireless links. The testbed network included three hosts running OpenSUSE Linux that acted as TCP senders or receivers depending on the scenario being considered.

The LBE congestion control mechanisms were evaluated based on foreground transaction time, background throughput, latency, and intra-protocol fairness. A total of seven LBE congestion control mechanisms were included in the evaluation: Vegas, LP, Nice, Westwood+LP, LEDBAT, Apple LEDBAT, and CDG. Nice, Westwood+LP, and Apple LEDBAT were implemented for this study.

Chapter 4 presents the results from this evaluation. The methodology described in this chapter is also used in the evaluation of Yield, described in Chapter 5.

4 Evaluation Results

4.1 Overview

This chapter presents and discusses the results of an evaluation of existing LBE congestion control mechanisms. This evaluation was conducted to comprehensively examine the performance characteristics of existing mechanisms, and identify areas where further investigation and improvement could be made.

Seven existing LBE congestion control mechanisms were included in this evaluation: Vegas, LP, Nice, Westwood+LP, LEDBAT, Apple LEDBAT, and CDG. The methodology for this evaluation was described in Chapter 3. The results of this evaluation are grouped by the metrics described in Section 3.5.

Each combination of bottleneck link speed, number of foreground and background TCP connections constitutes a single experiment. For each experiment, 10 runs were carried out per congestion control mechanism. Packet traces of all traffic sent and received by the testbed hosts were captured over a 60 second period for each run.

The data collection process was automated using TEACUP 1.0 [56], which performed the variation of bottleneck link speeds, path-delays, LBE congestion control modules, as well as initiating file transfers and collecting necessary files — such as packet traces — from the testbed hosts.

For experiments utilising 802.11 wireless networks, channel statistics — including the received signal strength indicator (RSSI) and channel number — were logged using a Python script that polled the iwconfig utility every 0.5 seconds.

The remainder of the chapter is structured as follows. Section 4.2 examines the impact of background traffic on foreground transaction time and background throughput. Section 4.3 considers the throughput achieved by foreground traffic. Section 4.4 examines the impact of LBE congestion control on queuing delay. Section 4.5 presents the intra-protocol fairness of each of the LBE congestion control mechanisms. Section 4.6 identifies the key findings of this evaluation and discusses their significance. Finally, Section 4.7 presents a summary of the findings.

4.2 Foreground Traffic Impact

The first set of experiments was carried out to examine the efficacy of LBE congestion control in reducing the impact of background traffic on competing foreground transfers managed by NewReno and CUBIC. In this scenario, heterogeneous traffic was downloaded by a single receiver to emulate the expected traffic profile for web traffic. These experiments also considered the case where the direction of the background traffic was reversed. These scenarios are described in greater detail in Section 3.3.

These experiments were used to determine the impact of LBE transfers on the bandwidth available to regular TCP connections, which was measured based on the foreground transaction time — the time required for foreground HTTP transfers to complete — and the background throughput.

4.2.1 NewReno Foreground Traffic

As the default congestion control mechanism for FreeBSD and some versions of Windows, the impact of the LBE mechanisms was first examined in relation to NewReno. Figure 4.1 plots the time required for a single foreground HTTP transfer using NewReno to complete against the throughput achieved by the background TCP transfer over a 60 second period. The bottleneck link for this experiment was 8 Mbps/1 Mbps with 50 ms of fixed-path delay.

LP and LEDBAT provided similar throughput to NewReno background traffic, with each mechanism achieving median throughputs of 923 KiB/s in this experiment. However, both mechanisms increased the median foreground transaction time to 9.2 seconds and 8.7 seconds respectively, as compared with 8.4 seconds for NewReno.

CDG and Vegas achieved the lowest median foreground transaction times of 3.3 seconds. However, CDG also had the lowest throughput of the mechanisms evaluated; 9.4% lower than NewReno. In contrast, Vegas substantially reduced the impact to foreground TCP traffic while providing throughput comparable to NewReno. Nice slightly increased median foreground transaction times (3.5 seconds) compared to CDG and Vegas, while providing a large throughput increase over CDG.

The implementation of Apple LEDBAT developed for this research experienced a similar trade-off to that of CDG, with a relatively low median foreground transaction time (4 seconds) but also a slight improvement to the throughput provided

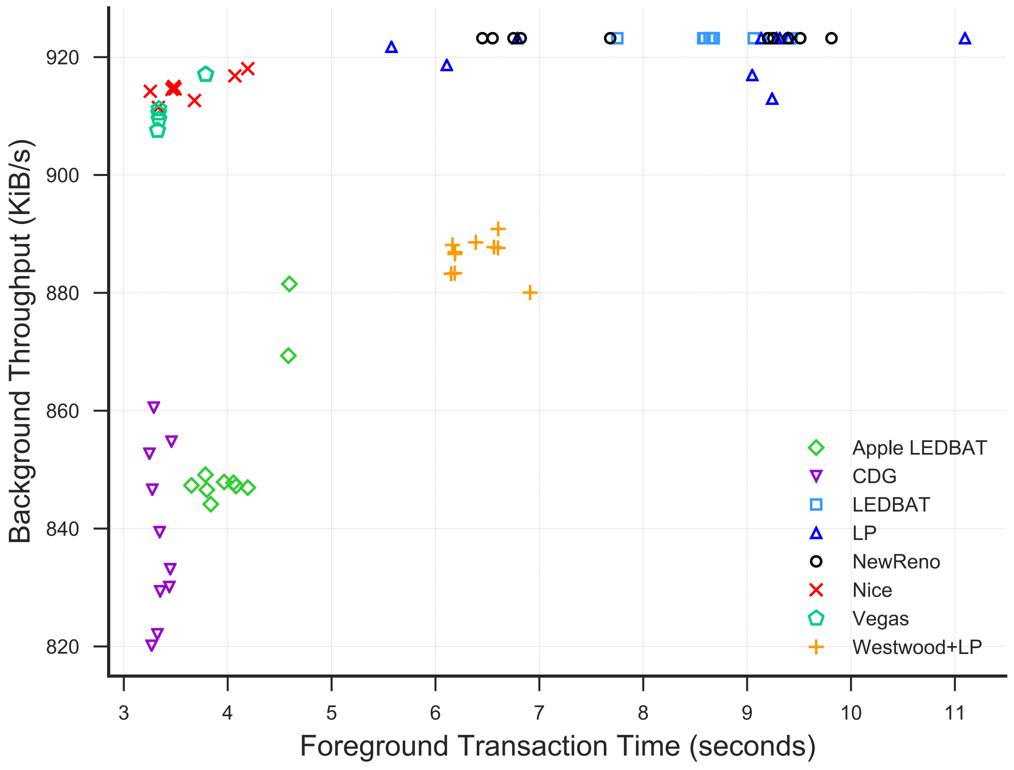


Figure 4.1.: Throughput and foreground transaction time for a download over 8Mbps/1Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

by CDG (848 KiB/s). Similarly, Westwood+LP moderately reduced throughput and foreground transaction times compared to NewReno.

The performance trends observed when the LBE congestion control mechanisms competed against a single HTTP transfer were similar when the same traffic was split across four foreground TCP connections, as shown in Figure 4.2, which plots the mean time required for the four foreground transfers to complete in a given run against the throughput achieved by the background TCP transfer.

As with the previous experiment, LP and LEDBAT performed similarly to NewReno, while Nice and CDG had relatively small impacts on foreground traffic. Consistent with the previous experiment, CDG and Nice demonstrated significant reductions to foreground transaction times with similar reductions in throughput.

However, the relative differences in foreground transaction times between the evaluated mechanisms became smaller as the total data transferred was split across multiple TCP connections. This was most evident in the differences between CDG

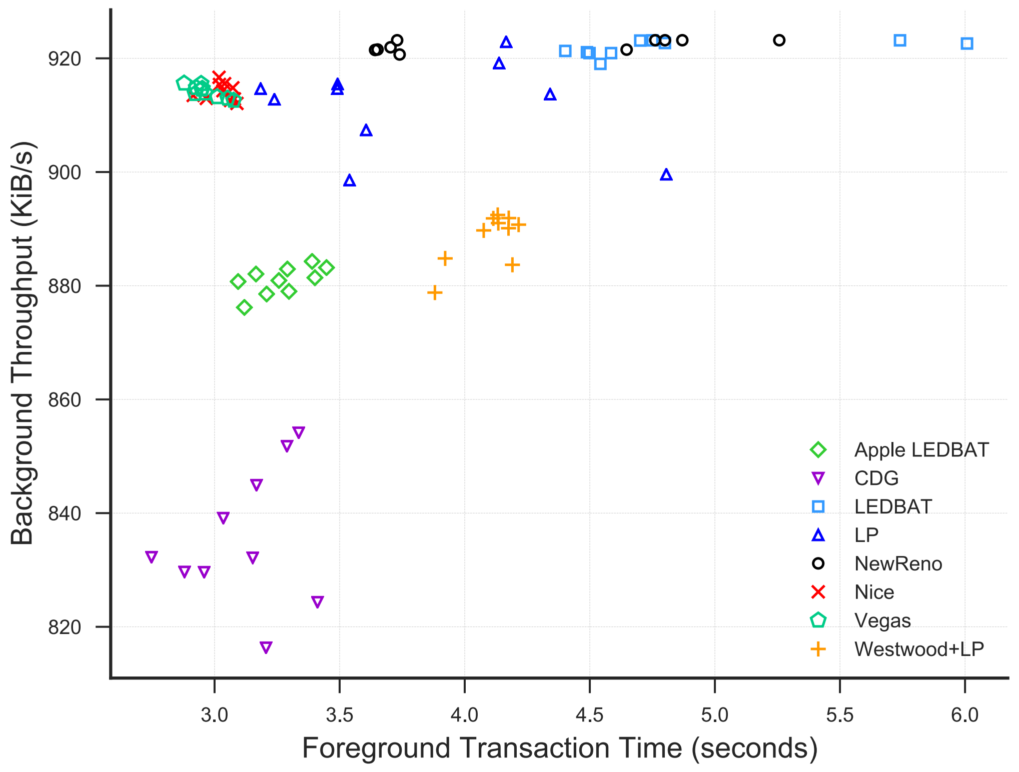


Figure 4.2.: Throughput and mean foreground transaction time for a download over 8Mbps/1Mbps bottleneck link subject to 50 ms fixed-path delay with four foreground TCP transfers using NewReno.

and NewReno, which were reduced from 60.5% to 26.7% for one and four connections, respectively.

Also notable was the performance of Apple LEDBAT, which improved as the number of competing TCP connections was increased. Reductions in background throughput compared to NewReno decreased from 8.2% to 4.5% when competing against one and four foreground transfers, respectively.

These relative performance trends remained consistent when foreground traffic was divided amongst eight TCP connections, although the foreground transaction times for NewReno, LP, and LEDBAT converged on those of Westwood+LP.

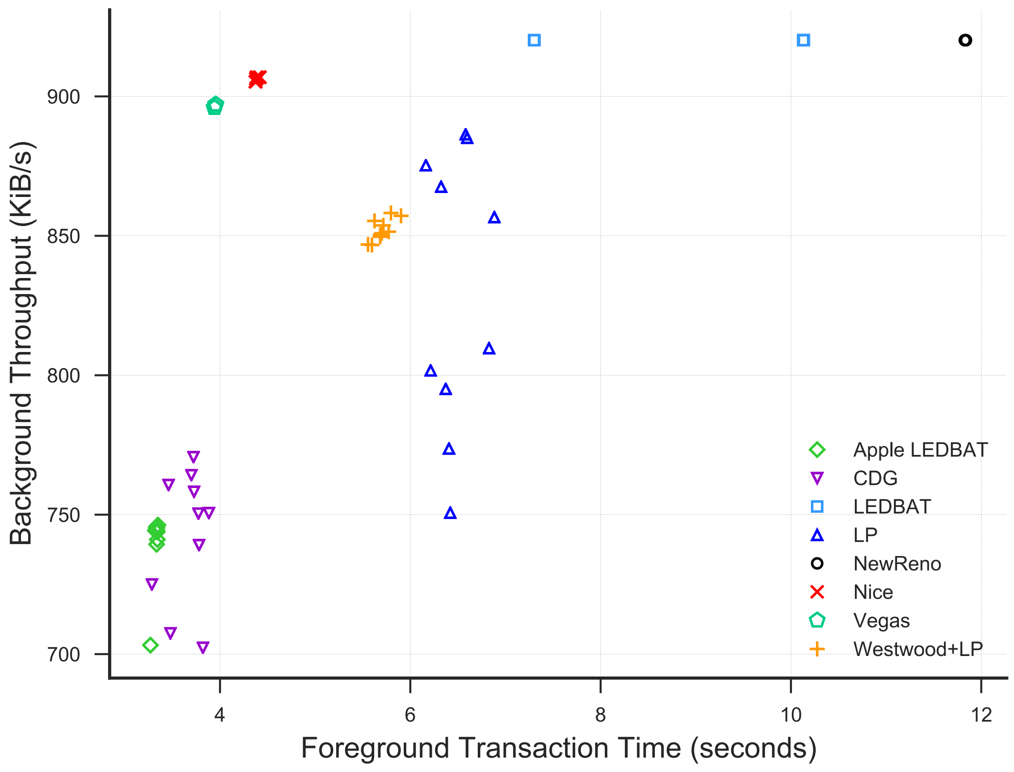


Figure 4.3.: Throughput and foreground transaction time for a download over 8Mbps/1Mbps bottleneck link subject to 100 ms fixed-path delay with a single foreground TCP transfer using NewReno.

4.2.2 Increasing Path Delay

While the performance impact of LBE mechanisms with 50 ms path delay was evaluated to provide consistency with previous experiments, these mechanisms would encounter a much wider range of path delays if used in production networks. As such, experiments were carried out that examined the impact of increasing path delay on the LBE mechanisms.

Fixed-path delay was first increased to 100 ms. Vegas, Nice, LEDBAT, and Westwood+LP performed similarly in this higher delay setting, as shown in Figure 4.3, with throughput and foreground transaction times remaining largely unchanged. However, changes are evident in the performance of LP and Apple LEDBAT.

In the increased delay setting, LP demonstrated a large (46%) reduction in foreground transaction time over NewReno. However, this reduced impact on foreground traffic was accompanied by a 10% reduction in throughput compared to

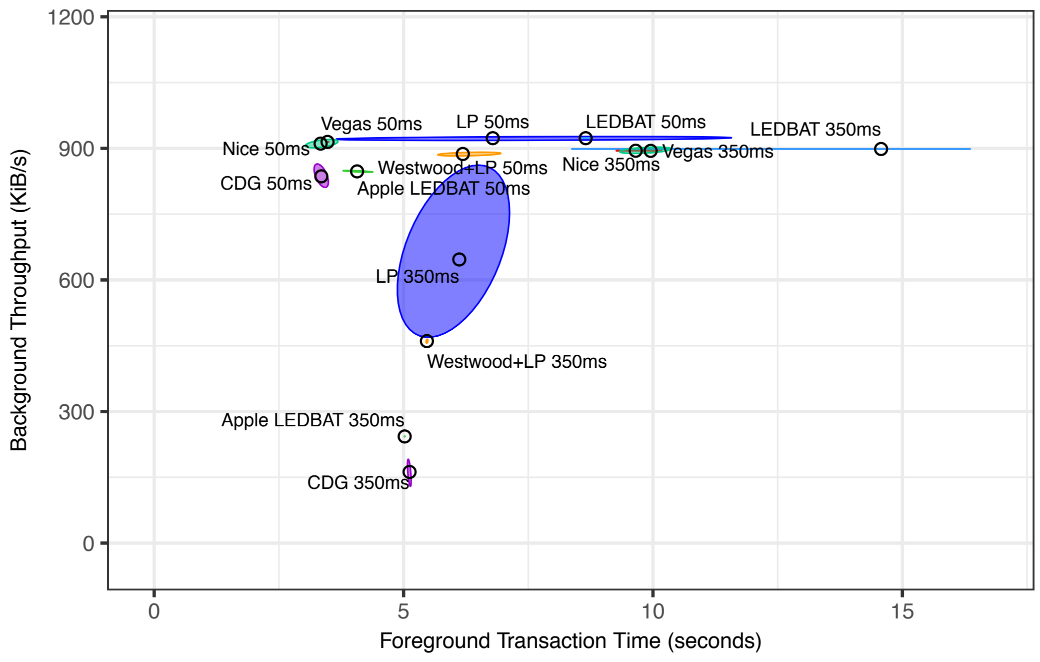


Figure 4.4.: Throughput and foreground transaction time (95% confidence region) for a download over 8 Mbps/1 Mbps bottleneck link subject to 50 ms and 350 ms fixed-path delay with a single foreground TCP transfer using NewReno.

NewReno (previously equivalent). LP also provided the least consistent throughput of the mechanisms evaluated, with a coefficient of variation (CV) — the ratio of standard deviation and mean — almost double that of the next least consistent mechanism, CDG ($CV = 0.06$ compared to $CV = 0.032$).

Apple LEDBAT had similarly reduced impact on foreground traffic, with foreground transaction times lower than CDG (3.3 seconds compared to 3.7 seconds) with a small reduction in throughput. The throughput achieved by Apple LEDBAT and CDG also decreased relative to NewReno, with decreases of 19.1% and 18.4% (compared to 8.2% and 9.4% with 50 ms fixed-path delay), respectively.

Performance trends remained similar when fixed-path delay was further increased (to 200 ms and 350 ms), with the full extent of changes in performance between the low and high delay cases evident in Figure 4.4 which plots the medians and ellipse-like 95% confidence regions for the foreground transaction time and throughput of the LBE congestion control mechanisms for 50 ms and 350 ms fixed-path delay.

The throughput penalties to Apple LEDBAT and CDG were exacerbated in the presence of very high fixed-path delay, with reductions of 72.3% and 81.5%, respectively. Westwood+LP also suffered from substantial throughput reductions at high delays: 26.1% and 47.6% for 200 ms and 350 ms, respectively.

As in the low delay cases, LEDBAT, Vegas, and Nice all demonstrated similar throughput to that of NewReno. However, Vegas and Nice also had an increased impact on foreground traffic, similar to that induced by LEDBAT.

4.2.3 Increasing Bottleneck Link Speed

In addition to a range of path delays, LBE mechanisms would likely be used over a range of link speeds. As such, further experiments were conducted to examine the performance of these mechanisms over high speed links as described in Section 3.2.1.

Most LBE congestion control mechanisms reduced the impact of background traffic on foreground TCP transfers when the bottleneck link speed was increased to 24 Mbps, as shown in Figure 4.5. However, reductions were significantly smaller than at lower speeds with CDG only reducing median foreground transaction time by 0.6 seconds, or 29.3% (compared to 5.1 seconds, or 60.5% over the slower 8 Mbps bottleneck link). The exception to this was LEDBAT, which doubled the median foreground transaction time observed for NewReno for 24 Mbps links.

These reduced differences between mechanisms persisted with increasing link speed (up to 100 Mbps). At these high speeds, the length of the foreground flow used was insufficient to identify performance differences between the LBE congestion control mechanisms. As seen in the results of experiments in which the mechanisms compete against a single foreground flow over a 100 Mbps link, shown in Figure 4.6, only CDG is distinguishable from the other mechanisms evaluated. In these experiments, CDG reduced median foreground transaction time by 29.2%, but also reduced median throughput by 32.2% compared to other mechanisms.

The compression in the differences between LBE mechanisms was due to the brevity of the foreground transfers. As such, additional experiments were conducted in which the size of foreground transfers were scaled up proportional to the link speed. Specifically, these experiments were run with a 100 Mbps bottleneck link and foreground transfers of 30862 KiB.

The results for an experiment conducted under these settings with a single foreground transfer are shown in Figure 4.7. In this experiment, the performance trends for the LBE mechanisms were similar to those observed when using lower

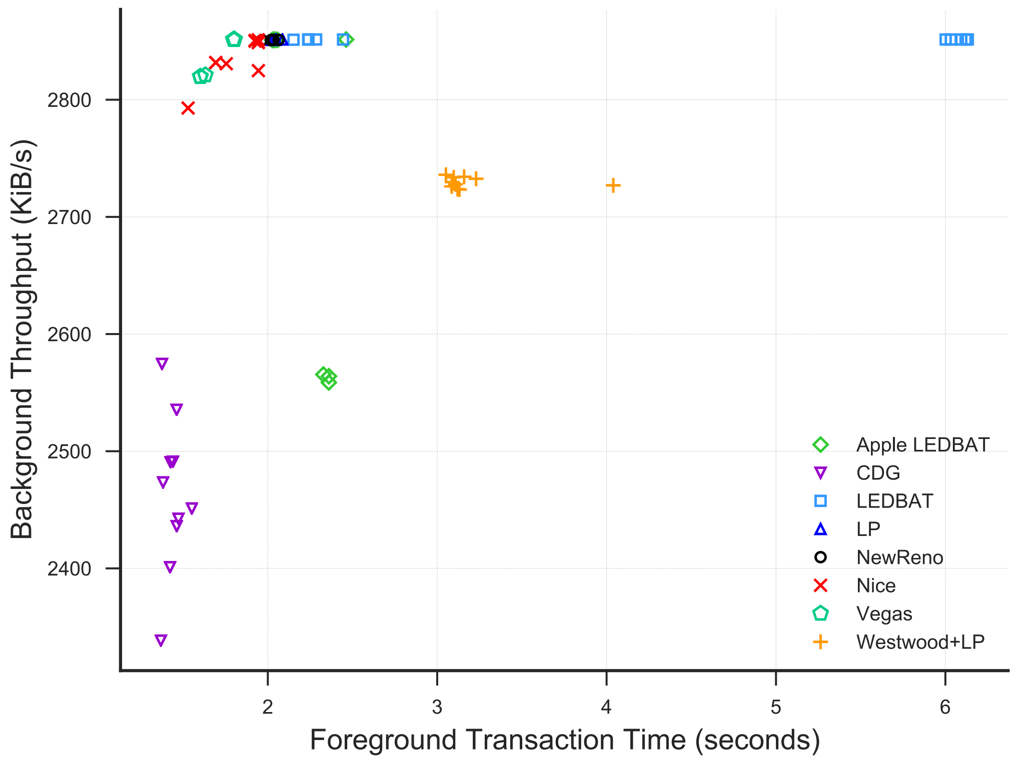


Figure 4.5.: Throughput and foreground transaction time for an 24 Mbps/1.5 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

bottleneck link speeds. However, foreground transaction times were higher than over slower bottleneck links.

In particular, the relative decreases in median foreground transaction time for Nice and Vegas — compared to NewReno — were observed to be approximately one half that demonstrated in experiments using an 8 Mbps link (reductions of 24.3% and 24.1%, compared to 58.8% and 60.5% for Nice and Vegas, respectively). Similarly, Apple LEDBAT only achieved a 13.9% reduction in median foreground transaction time (compared to 52.5% over an 8 Mbps bottleneck link).

While CDG provided a similar improvement to foreground transaction time (improvement of 51.3%, compared to 60.5%), the reduction in background throughput was similar to previous experiments — including that shown in Figure 4.6 — over the high speed link. This 32.3% reduction to background throughput suggests that CDG has difficulty utilising the available bandwidth as link speeds increase.

LP and LEDBAT both achieve decreases in median foreground transaction time — 11.3% and 21%, respectively — instead of the increases observed over the low

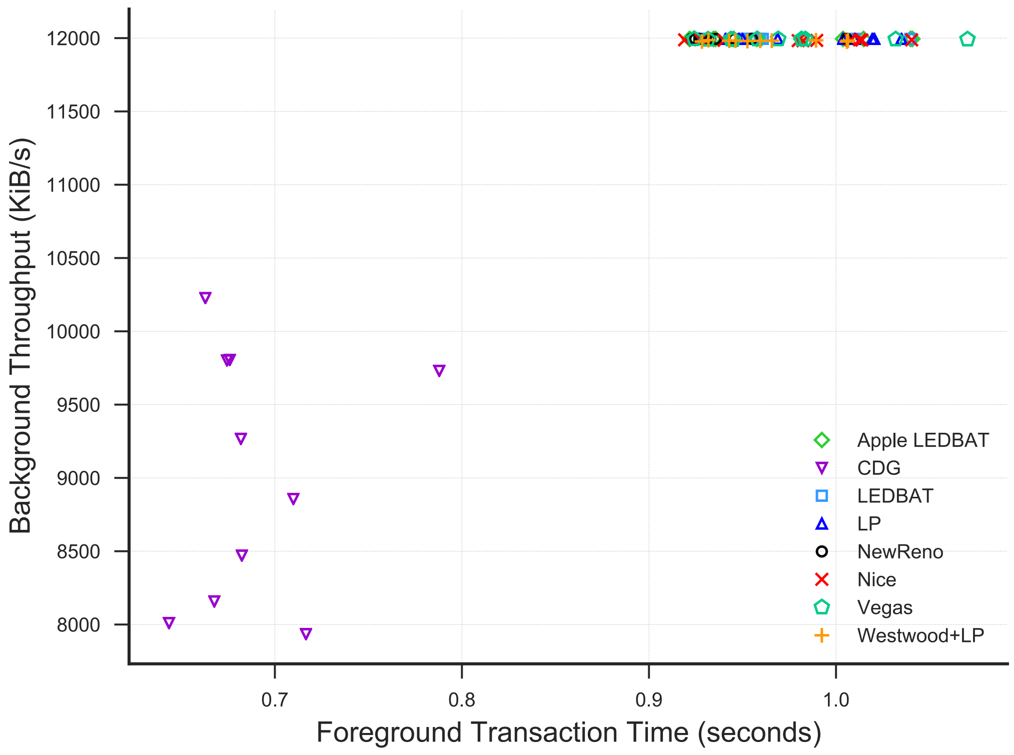


Figure 4.6.: Throughput and foreground transaction time for an 100 Mbps/100 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

speed link. As with slower bottleneck links, LEDBAT achieved this decrease without any impact to background throughput. However, median background throughput was reduced by 11.3% for LP. Westwood+LP also achieved a larger decrease in foreground transaction time (36.7%, compared with 25.5% with an 8 Mbps bottleneck link), but this was accompanied by a 6.6% decrease in median background throughput.

4.2.4 CUBIC Foreground Traffic

CUBIC has replaced NewReno as the default TCP congestion control mechanism used by Linux and recent versions of macOS. As such, LBE mechanisms used over production networks must be able to maintain similarly low impact on CUBIC foreground traffic as when competing against NewReno. To evaluate the impact of the LBE mechanisms on CUBIC-managed foreground traffic, the experiments described in the previous sections were replicated using CUBIC for foreground transfers.

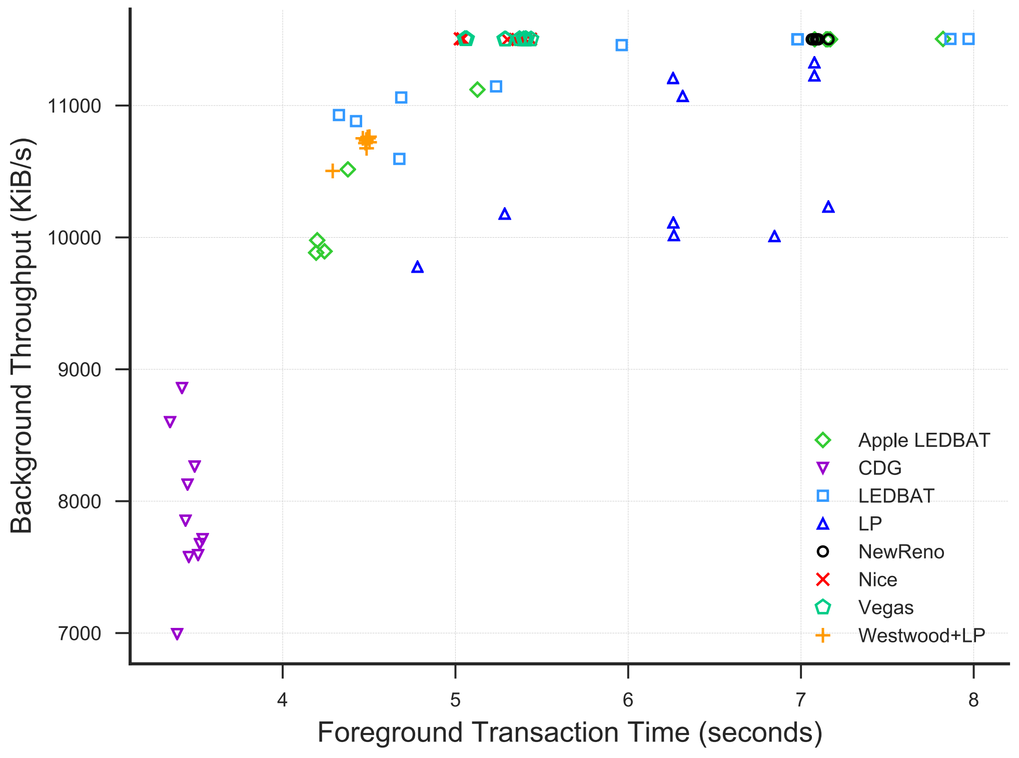


Figure 4.7.: Throughput and foreground transaction time for an 100 Mbps/100 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer of 30862 KiB using NewReno.

The seven LBE mechanisms performed similarly when competing with CUBIC for bandwidth, as depicted in Figure 4.8, with LP, LEDBAT, Nice, and Vegas all achieving throughput comparable to that of CUBIC. However, LP and LEDBAT reduced foreground transaction times when competing against CUBIC (by 15% and 18.5%, respectively).

As when competing against NewReno, CDG, Vegas, Nice, and Apple LEDBAT provided large reductions in foreground transaction time, with reductions of 69.1%, 64.7%, 61.8%, and 60% over CUBIC, respectively. However, CDG and Apple LEDBAT achieved the lowest throughput of the mechanisms.

The performance of the LBE mechanisms was also similar when competing against additional CUBIC-managed foreground connections. Figure 4.9 shows the results from an experiment in which four foreground transfers were used.

As with experiments using NewReno foreground traffic, the differences between the LBE mechanisms became compressed when the number of concurrent foreground transfers was increased. Notably, Apple LEDBAT, CDG, Nice, and Vegas all

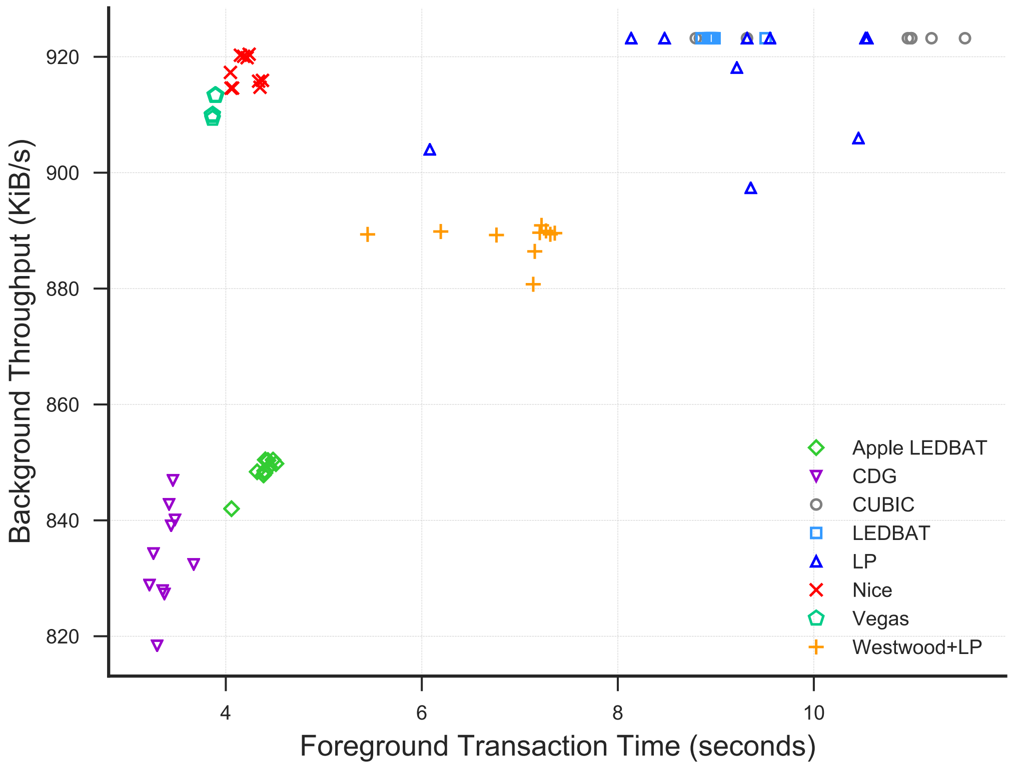


Figure 4.8.: Throughput and foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using CUBIC.

demonstrated very similar foreground transaction times and background throughputs. Performance for LEDBAT was also consistent with competition against a single foreground transfer.

The exception to this homogeneity was LP, which demonstrated a 32.5% reduction to foreground transaction time compared to CUBIC (compared with 15.1% with a single foreground transfer). These relative performance trends remained consistent when foreground traffic was divided amongst eight TCP connections, although the differences between mechanisms was further reduced.

4.2.5 LBE Upload

To examine the potential impact of using LBE mechanisms in scenarios such as file synchronisation and uploading of user-generated content, experiments were conducted in which the direction of the background traffic was reversed. In these

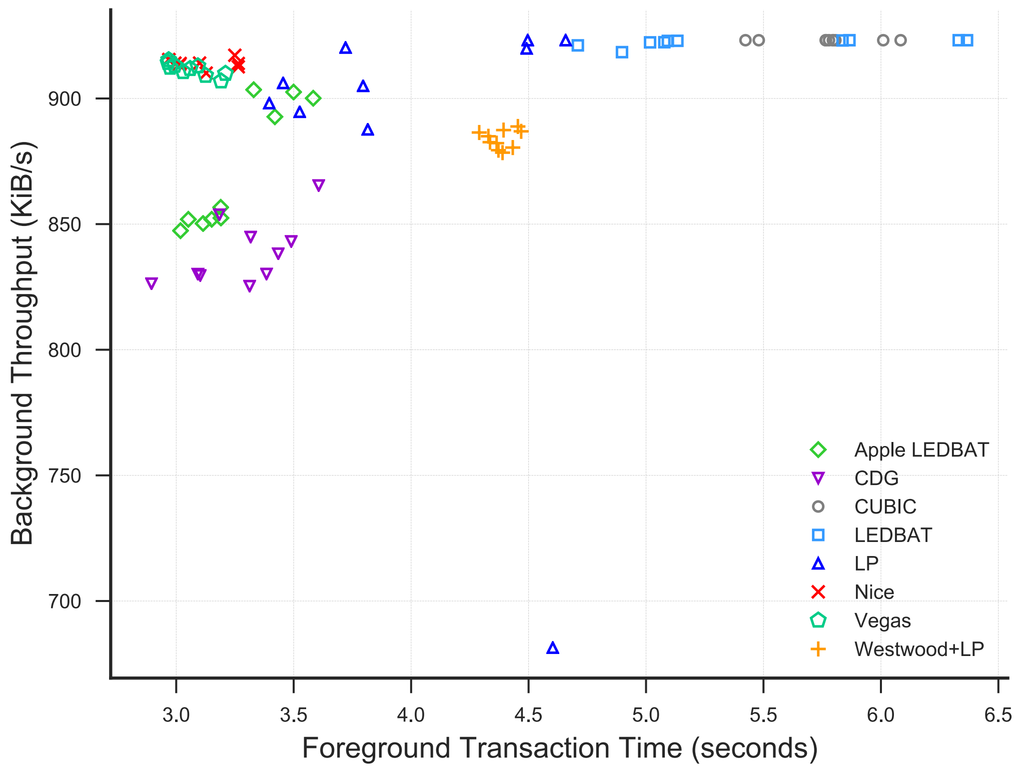


Figure 4.9.: Throughput and mean foreground transaction time for a download over 8Mbps/1Mbps bottleneck link subject to 50 ms fixed-path delay with four foreground TCP transfers using CUBIC.

experiments, a large file upload was managed by LBE congestion control and competed against one or more HTTP downloads.

Due to the very low uplink speeds, all mechanisms achieved similar throughput over the 8 Mbps/1 Mbps and 24 Mbps/1.5 Mbps bottleneck links. However, Figure 4.10 shows that LP and Westwood+LP had the greatest impact on regular TCP traffic. LEDBAT notably has far lower impact on foreground traffic in this scenario, with similar performance to Apple LEDBAT.

As with the competing downloads scenario, performance trends were consistent as path delay increased. The exception to this trend was LEDBAT, which caused larger increases in median foreground transaction times than in previous experiments.

Differences in the performance of the LBE congestion control mechanisms were more apparent when the transfers occurred over higher link speeds, such as with the 50 Mbps bottleneck link, as shown in Figure 4.11. However, the results of these experiments closely resemble those from the competing downloads scenario using a 24 Mbps downlink (due to similarities in link speed for the background

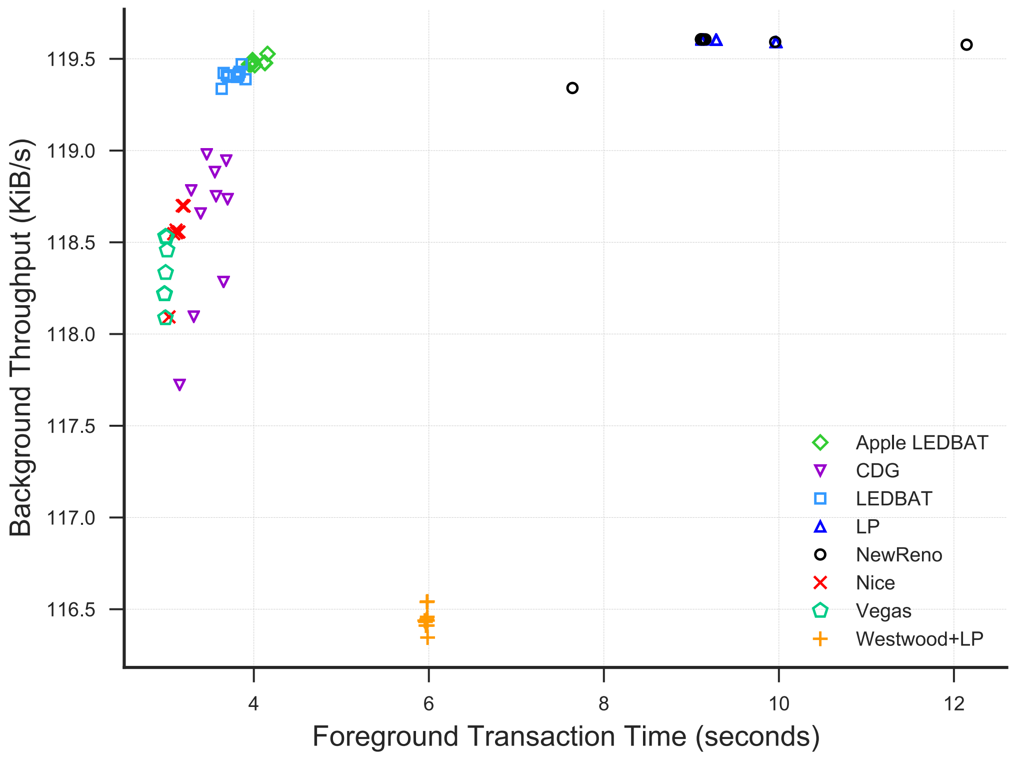


Figure 4.10.: Throughput and foreground transaction time for an upload over a 8 Mbps/1 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

transfers). Consistent with the competing downloads scenario, CDG, Vegas, and Nice provided the lowest foreground transaction times (medians of 0.8, 0.8, and 0.9 seconds compared to 2.1 seconds for NewReno).

Relative throughput for CDG, Vegas, and Nice was also similar to that observed in the competing downloads scenario. LEDBAT achieved background throughput equivalent to that of NewReno, but also had a greater impact on foreground traffic, increasing median foreground transaction time by 10.2%.

As shown in Figure 4.12, all of the mechanisms provided a noticeable improvement to median foreground transaction time compared to CUBIC, from a reduction of 22.8% for LEDBAT to 74.2% for CDG. As in other scenarios, LP was able to achieve these reductions without compromising background throughput, while Nice and LEDBAT demonstrated negligible decreases in background throughput.

As in the competing downloads scenario, all mechanisms except CDG performed very similarly in tests over a 100 Mbps bottleneck link. Performance trends also re-

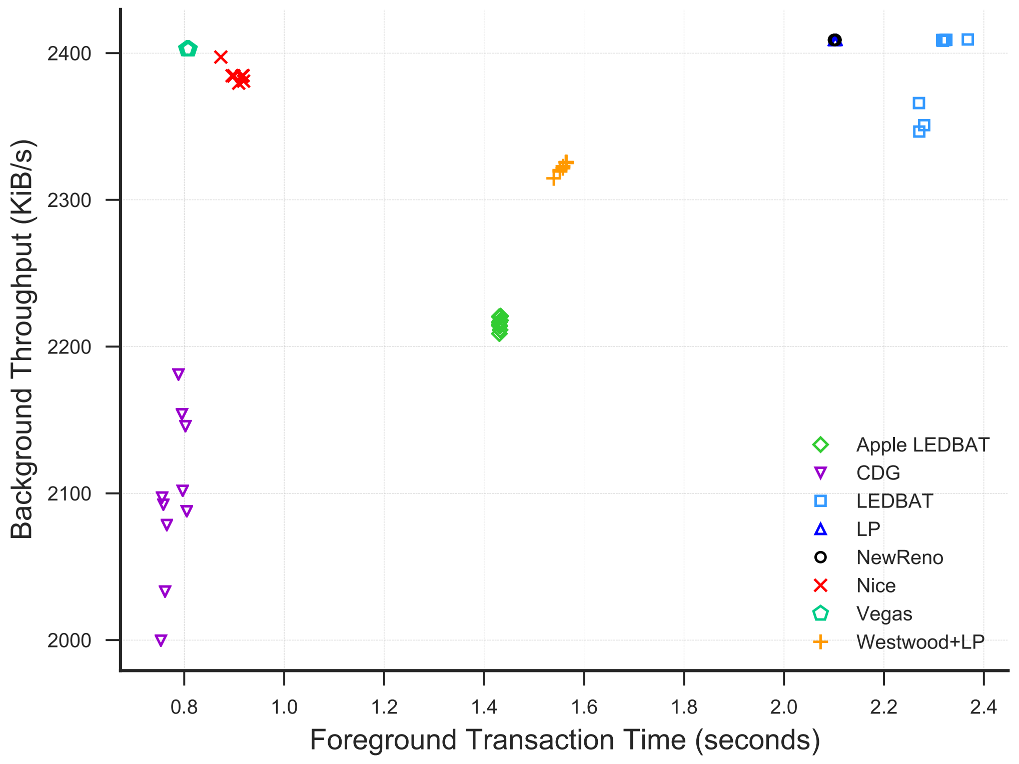


Figure 4.11.: Throughput and foreground transaction time for an upload over a 50 Mbps/20 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

mained consistent when foreground traffic was split across multiple HTTP transfers, as well as when the LBE mechanisms competed with CUBIC.

4.2.6 Wireless Networks

Given the proliferation of WiFi networks, the performance of LBE congestion control mechanisms over an 802.11n wireless link was also investigated. As seen in Figure 4.13, which shows results for experiments in which a single foreground transfer was introduced against a long running background transfer, the performance trends observed in the wireless experiments was largely consistent with the equivalent experiments over wired links (shown in Figure 4.1).

However, some additional variability was observed in the results of experiments that included a wireless link. This observation was evident in comparing the CV values for wired and wireless links. These values were calculated on a per-mechanism

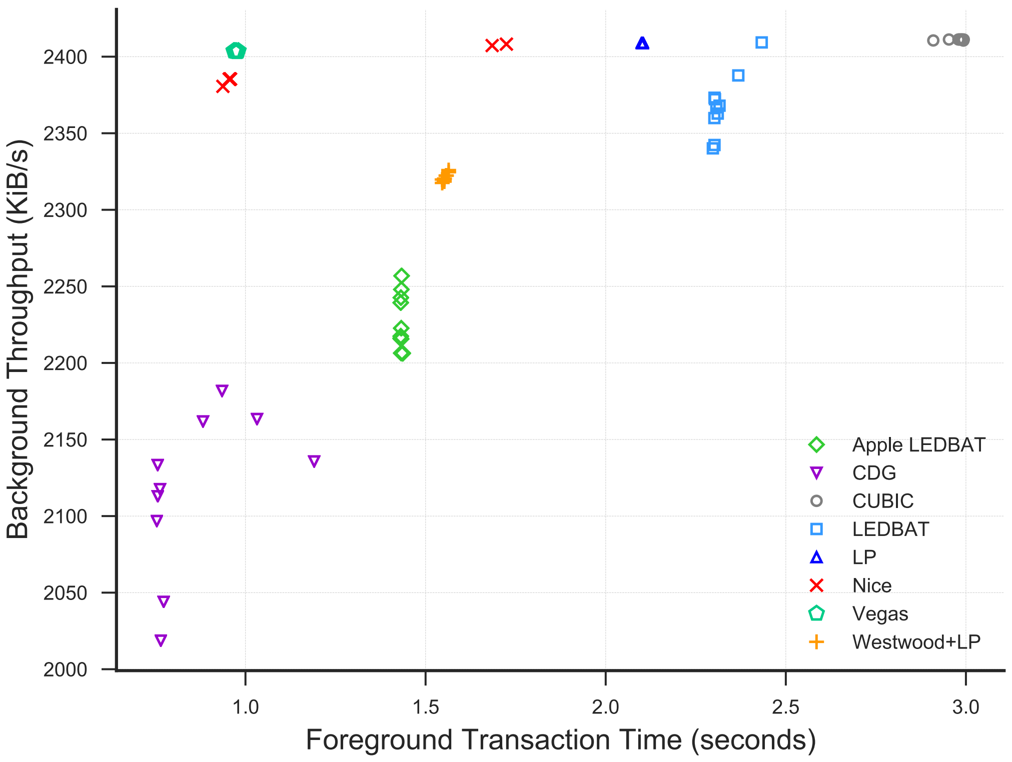


Figure 4.12.: Throughput and foreground transaction time for an upload over a 50 Mbps/20 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using CUBIC.

basis for each experiment, and are shown in Figure 4.14. While the results indicate some additional variability in experiments over wireless networks ($\overline{CV} = 0.08$, compared to $\overline{CV} = 0.05$ over Ethernet), the differences are relatively small.

Another small increase in variability was observed in throughput readings, as shown in Figure 4.15, where the mean CV for wireless networks was found to be 0.08 (compared with $\overline{CV} = 0.05$ over Ethernet). However, the minimal impact of wireless links could primarily be attributed to the ideal channel conditions in the testbed network.

4.3 Foreground Throughput

The experiments described in the previous sections made use of short foreground transfers to emulate the loading of a webpage. However, LBE mechanisms would be expected to encounter scenarios in which regular TCP congestion control was

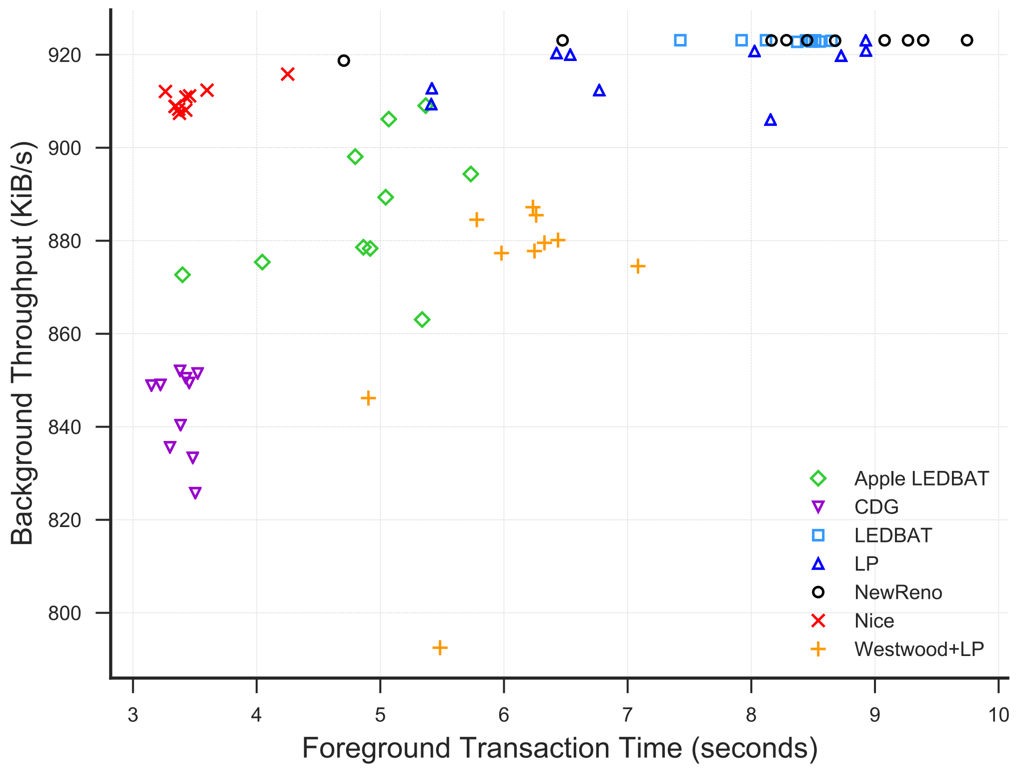
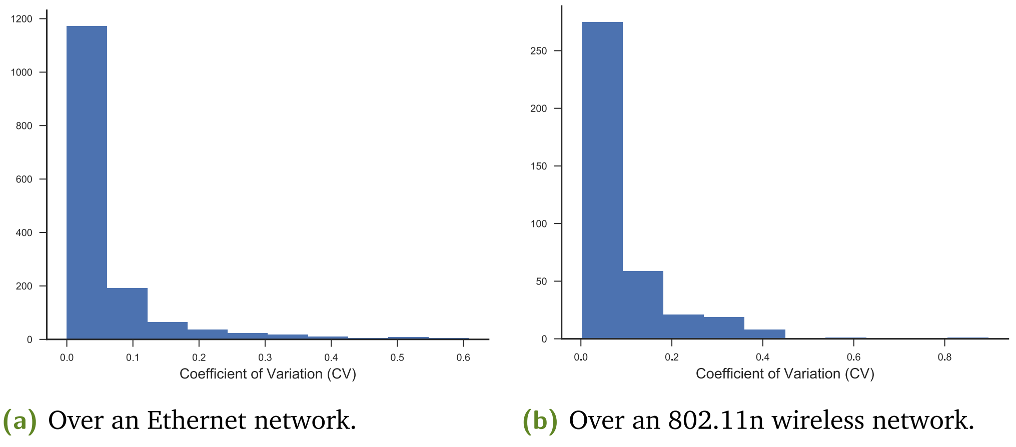


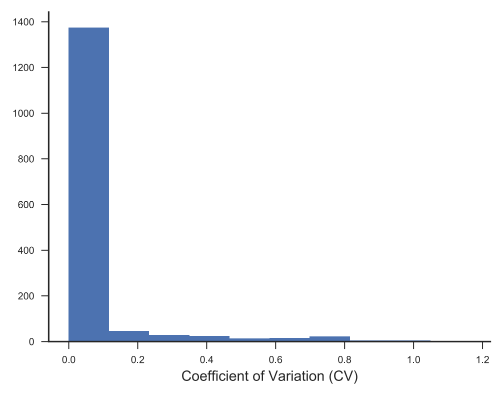
Figure 4.13.: Throughput and foreground transaction time for a download over 8Mbps/1Mbps bottleneck link with a single 802.11n wireless link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.



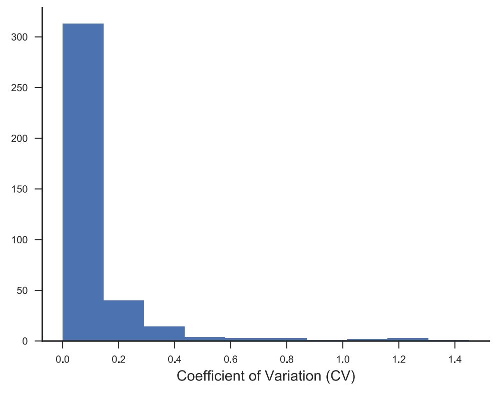
(a) Over an Ethernet network. **(b)** Over an 802.11n wireless network.

Figure 4.14.: Histogram of CV values for foreground transaction time.

employed for larger downloads. As such, an additional set of experiments in which the LBE mechanisms competed against a long-lived foreground TCP transfer was carried out.



(a) Over an Ethernet network.



(b) Over an 802.11n wireless network.

Figure 4.15.: Histogram of CV values for background throughput.

In these experiments, long-lived foreground and background traffic competed for bandwidth over the same bottleneck link. These experiments were also carried out with the direction of the background transfer inverted. The impact of the LBE mechanisms on foreground traffic was measured based on the median throughput achieved in the course of each run.

4.3.1 NewReno Foreground Traffic

The performance of the LBE mechanisms when competing against long-lived foreground traffic was consistent with experiments with short foreground transfers. Figure 4.16 plots the throughput of the NewReno foreground transfers when competing against a single LBE download over an 8 Mbps bottleneck link. LEDBAT remained the most aggressive of the LBE mechanisms, with foreground traffic observed to achieve median throughput of 540 KiB/s. However, LEDBAT provided foreground traffic with a greater share of available bandwidth than when using NewReno for background transfers (median throughput of 480 KiB/s).

As in experiments with shorter foreground transfers, CDG, Nice, and Vegas remained the most friendly of the seven mechanisms evaluated. As in previous experiments, NewReno foreground traffic was able to achieve similar median throughput when competing against Nice and Vegas (877 KiB/s and 886 KiB/s, respectively). However, NewReno foreground traffic achieved slightly less throughput when competing against CDG (median foreground throughput of 846 KiB/s). Apple LEDBAT and Westwood+LP also achieved results consistent with those from experiments

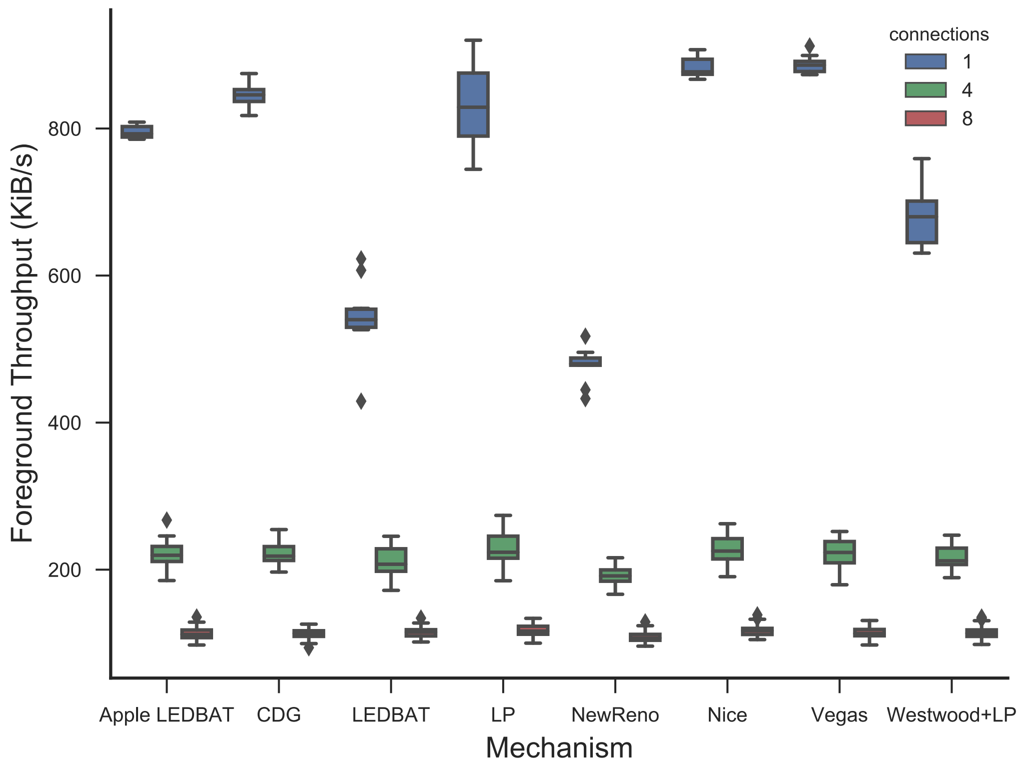


Figure 4.16.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE download over an 8 Mbps/1 Mbps bottleneck link with 50 ms fixed-path delay.

with short foreground transfers (median foreground throughput of 793 KiB/s and 680 KiB/s, respectively).

Interestingly, LP became noticeably friendlier when competing with long foreground flows (median foreground throughput of 829 KiB/s). However, LP was also the least consistent of the LBE mechanisms evaluated, with throughput ranging from 745 KiB/s to 920 KiB/s ($CV = 0.072$).

As observed in experiments with short foreground transfers, such as that shown in Figure 4.2, the differences between the LBE mechanisms were diminished as additional foreground transfers were introduced. When competing with four concurrent NewReno transfers, the median per-connection foreground throughput for the mechanisms ranged from 207 KiB/s to 225 KiB/s (for LEDBAT and Nice, respectively). This trend continued when the number of foreground transfers was again increased (to eight), where the median per-connection foreground throughput ranged from 112 KiB/s to 117 KiB/s (for Apple LEDBAT and Nice, respectively).

4.3.2 Increasing Path Delay

The consistency in results with experiments utilising short foreground transfers was also present at higher delay settings, as shown for 350 ms fixed-path delay using an 8 Mbps link in Figure 4.17. Apple LEDBAT, CDG, LP, and Westwood+LP — which achieved low foreground transaction times in previous experiments — conceded the greatest share of available bandwidth to the foreground traffic. NewReno foreground traffic was also able to achieve median throughput of 906 KiB/s and 902 KiB/s when competing against Apple LEDBAT and CDG, respectively (representing increases in foreground throughput of 61.3% and 60.6% compared to use of NewReno for background transfers).

LEDBAT and Nice were noticeably more aggressive than NewReno in high delay settings, with NewReno foreground traffic achieving 53.8% and 44.3% of the foreground throughput achieved by NewReno background transfers when competing against these mechanisms, respectively. Vegas was also slightly more aggressive than NewReno, with a reduction to foreground throughput of 11.6%.

Differences between the LBE mechanisms were also diminished in this high delay setting as the number of concurrent foreground transfers was increased to four and eight, although some differences between the mechanisms could still be observed. LBE mechanisms that allowed foreground traffic to achieve higher throughput (Apple LEDBAT, CDG, LP, and Westwood+LP) all did so as the number of foreground connections was increased. These mechanisms demonstrated improvements over NewReno of between 49% (for Apple LEDBAT) and 54.6% (for Westwood+LP) when competing against four concurrent transfers, with improvements of 33.4% (for Westwood+LP) to 42.5% (for Apple LEDBAT and LP) for experiments with eight foreground transfers.

Notably, Nice and Vegas became less aggressive when competing with multiple foreground transfers, allowing NewReno foreground transfers to achieve throughput equivalent to that when NewReno was used for background traffic. However, LEDBAT remained the most aggressive of the mechanisms evaluated with reductions in foreground throughput of 28.7% and 15.2% observed when competing against four and eight concurrent foreground transfers, respectively.

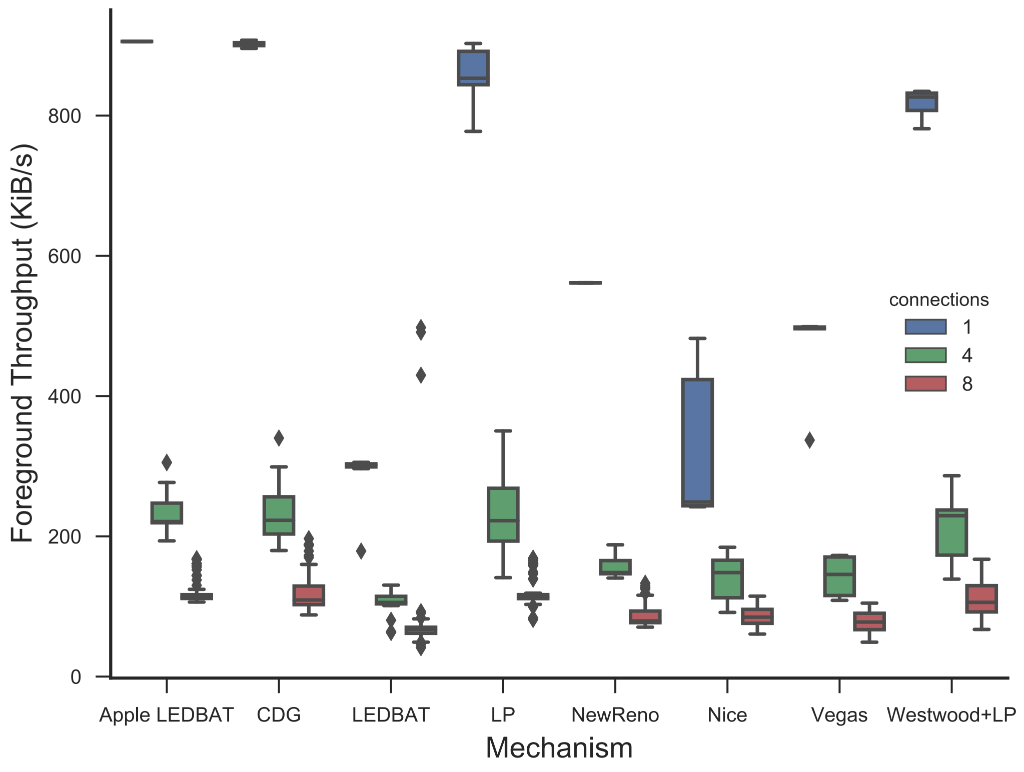


Figure 4.17.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE download over an 8 Mbps/1 Mbps bottleneck link subject to 350 ms fixed-path delay.

4.3.3 Increasing Bottleneck Link Speed

Performance trends remained fairly consistent as bottleneck link speed was increased, although some changes were evident at 50 Mbps and 100 Mbps. Results of experiments over a 100 Mbps bottleneck link, shown in Figure 4.18, indicate that Apple LEDBAT, Nice, and Vegas all became more aggressive than LEDBAT in this setting.

With a 100 Mbps bottleneck link, NewReno foreground traffic achieved median throughput of 7407 KiB/s when competing against LEDBAT. By contrast, foreground transfers competing against Apple LEDBAT, Nice, and Vegas achieved median throughput of 6990 KiB/s, 6890 KiB/s, and 7201 KiB/s (reductions of 5.6%, 6.9%, and 2.7% compared to LEDBAT), respectively. All three mechanisms were still able to concede more bandwidth to foreground transfers than NewReno, with increases in foreground throughput of 26.2%, 24.3%, and 30% over NewReno, respectively.

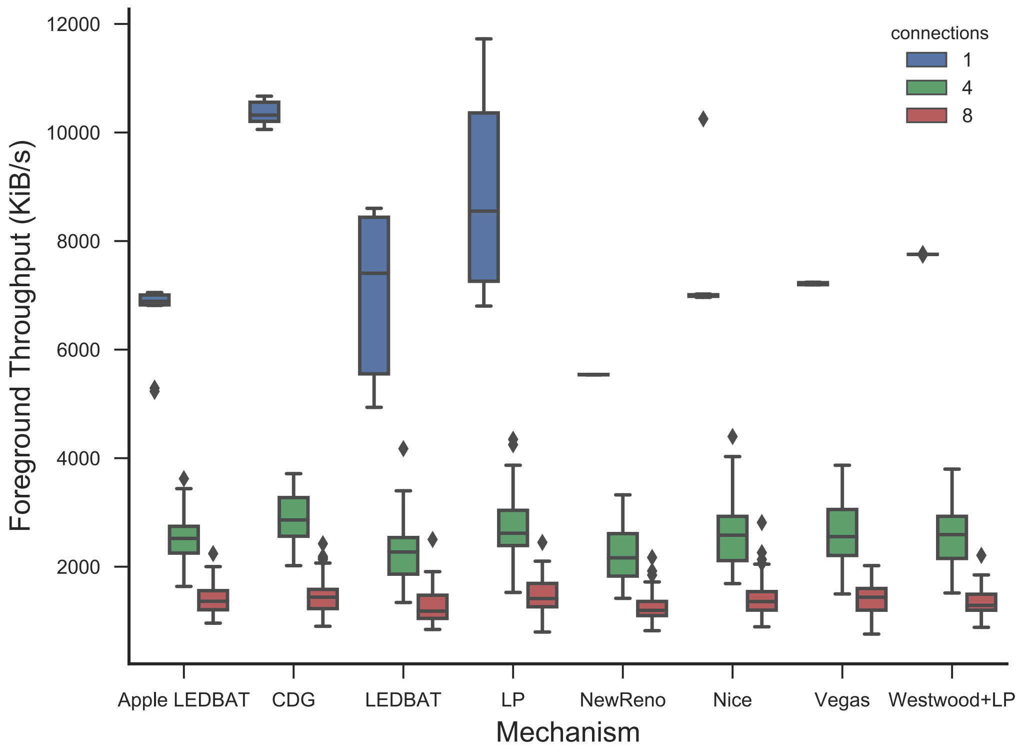


Figure 4.18.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE download over an 100 Mbps/100 Mbps bottleneck link with 50 ms fixed-path delay.

This aggressive behaviour was not evident as additional foreground transfers and path delay were introduced. In these experiments, performance of the LBE mechanisms was similar to that of other experiments with LEDBAT once again being the most aggressive of the LBE mechanisms.

4.3.4 CUBIC Foreground Traffic

The trends observed for NewReno foreground traffic were also apparent when the LBE mechanisms competed against CUBIC foreground traffic, as shown in Figure 4.19. When competing against CUBIC foreground traffic, all mechanisms were able to concede a greater portion of the available bandwidth to CUBIC foreground traffic than was observed when competing against NewReno.

LEDBAT was again observed to be the most aggressive of the LBE mechanisms providing a 31.7% increase in median foreground throughput over CUBIC, with LP allowing foreground traffic to achieve the greatest throughput (2.5x improvement

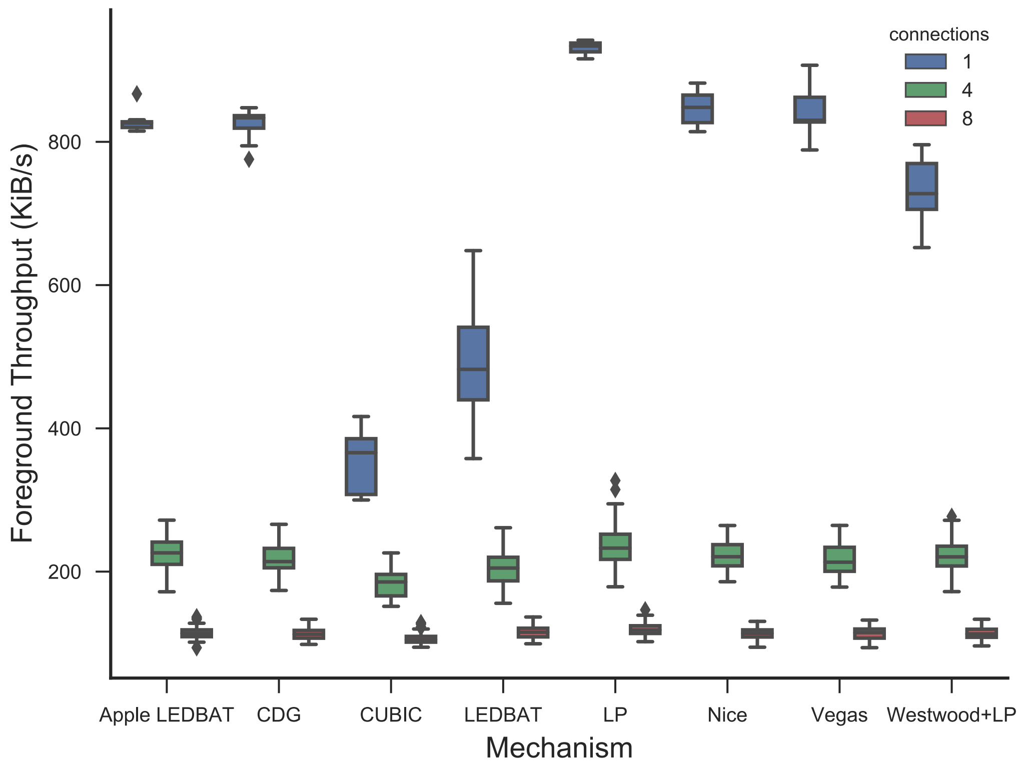


Figure 4.19.: Throughput for one, four, and eight foreground transfers using CUBIC competing against a single LBE download over an 8 Mbps/1 Mbps bottleneck link with 50 ms fixed-path delay.

over CUBIC). The performance of LP was also noticeably more consistent than when competing against NewReno foreground traffic (with $CV = 0.009$ compared with $CV = 0.072$ when competing with NewReno).

The performance of the LBE mechanisms when the number of concurrent foreground transfers, and fixed-path delay, was increased also remained consistent with results of experiments described in the previous sections. Performance was also observed to be consistent when competing against CUBIC over high speed links.

4.3.5 LBE Upload

As when the LBE mechanisms competed against short foreground transfers, and as shown in Figure 4.20, some differences in performance could be observed when the direction of the background transfer was reversed over the 8 Mbps bottleneck link. Westwood+LP, Nice, CDG, and Vegas allowed NewReno foreground traffic to achieve similar throughput as in the competing downloads scenario. LEDBAT became significantly less aggressive in this scenario, with a median increase in foreground throughput of 95.6% over when NewReno was used for background traffic (compared to 12.5% when competing for downstream bandwidth).

LP had a noticeably higher impact on foreground transfer throughput, with only a 9.2% median increase in throughput compared to when NewReno was used for the upload (compared to a 72.7% increase when used for an LBE download). However, this change is consistent with the performance of LP in the short foreground transfer experiments described in Section 4.2.5.

Apple LEDBAT was also somewhat less friendly in absolute terms when the direction of the background transfer wass reversed, with median foreground throughput of 665 KiB/s. However, it still demonstrated a comparable improvement to foreground throughput as in experiments for the competing downloads scenario (median increase of 64.8%, compared to 65.2%). As in previous experiments, these results were reflected when LBE mechanisms competed with CUBIC, and to a lesser degree in performance when competiting against a larger number of foreground transfers.

The results in high delay settings were also consistent with experiments using short foreground transfers, as shown in Figure 4.21. In these experiments, CDG allowed NewReno foreground traffic to achieve the greatest median throughput (917 KiB/s). Nice and Vegas performed similarly, with the next highest median foreground throughputs of 710 KiB/s and 666 KiB/s, respectively. Apple LEDBAT and Westwood+LP were also largely indistinguishable, with median foreground throughputs of 474 KiB/s and 475 KiB/s. LP provided very low foreground throughput equal to that when NewReno was used for background traffic.

These performance trends remained consistent until the LBE mechanisms competed for bandwidth over a 100 Mbps symmetrical bottleneck link, the results of which are shown in Figure 4.22. When competing over the high speed bottleneck link, differences in the performance of the LBE mechanisms could still be observed but were further diminished (even with only a single foreground download). Apple LEDBAT, CDG, and LEDBAT allowed NewReno foreground traffic to achieve the greatest throughput in these experiments. LP, Nice, Vegas, and Westwood+LP were more aggressive, but only marginally; foreground throughput for all LBE mechanisms was over 11 MiB/s. This performance was also mirrored with CUBIC foreground traffic, with differences further diminished between the mechanisms.

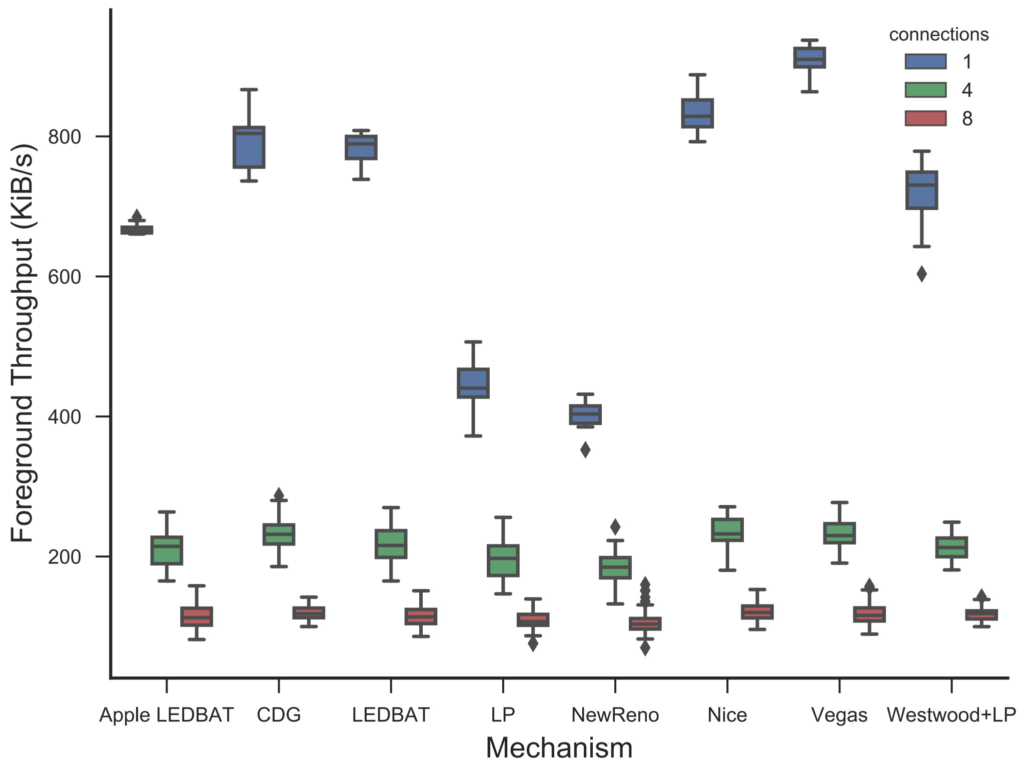


Figure 4.20.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE upload over an 8 Mbps/1 Mbps bottleneck link with 50 ms fixed-path delay.

4.4 Queuing Delay

The effect of LBE congestion control mechanisms on the queuing delay experienced by foreground and background traffic was also investigated. In particular, any increases in the delay experienced by foreground traffic would likely have an effect on the usability of interactive applications, as well as streaming video.

For these experiments, delay was calculated by Wireshark based on the time difference between the transmission of the original packet and the time at which the acknowledgement was received. The impact of LBE mechanisms on delay was only examined in the scenarios where longer foreground transfers were initiated, as well as in the fairness scenario described in Section 3.3.

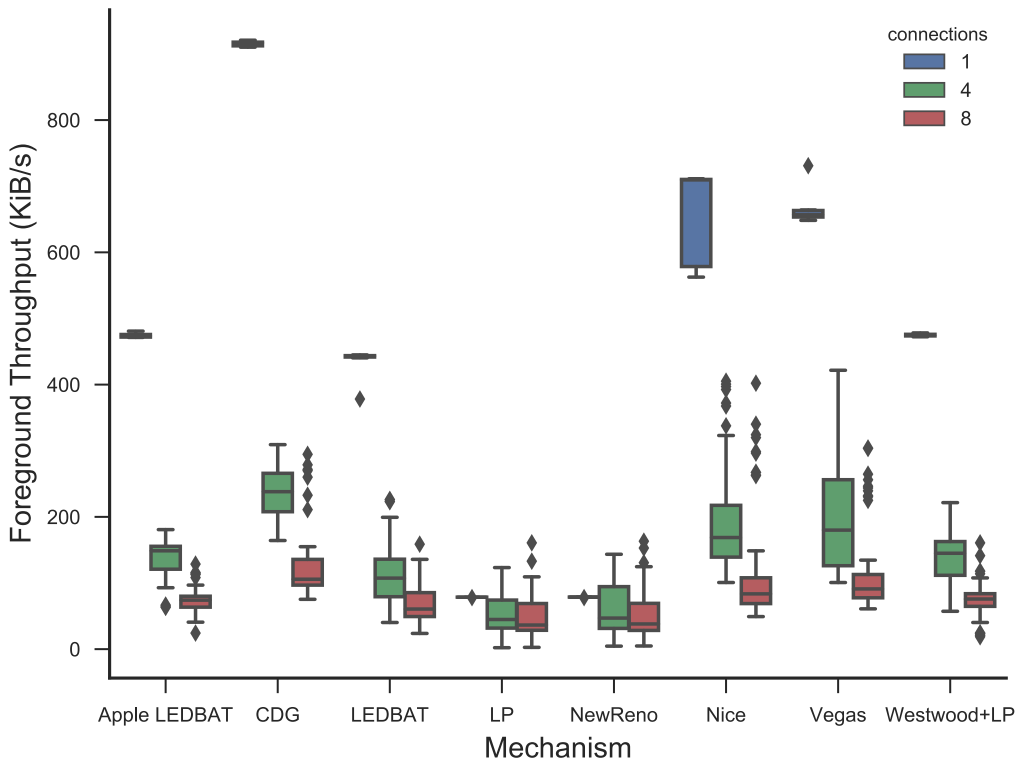


Figure 4.21.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE upload over an 8 Mbps/1 Mbps bottleneck link with 350 ms fixed-path delay.

4.4.1 NewReno Foreground Traffic

In evaluating the impact of the LBE mechanisms on delay, the results of the LBE mechanisms competing with NewReno foreground traffic were first considered. Table 4.1 lists the mean and standard deviation of latency for experiments where a single LBE download competed against one, four, or eight concurrent foreground TCP transfers using NewReno.

For experiments using lower delay settings (≤ 200 ms) LP was the only mechanism to consistently reduce queuing delay compared to NewReno. However, these reductions were only observed with a single foreground transfer. With four or eight concurrent foreground transfers, the LBE mechanisms demonstrated very limited change in delay, likely due to the foreground traffic ensuring that the queue remains full.

LP more consistently reduced queuing delay when path delay was increased to 350 ms, with a mean reduction of 14.4% compared to NewReno. However, both

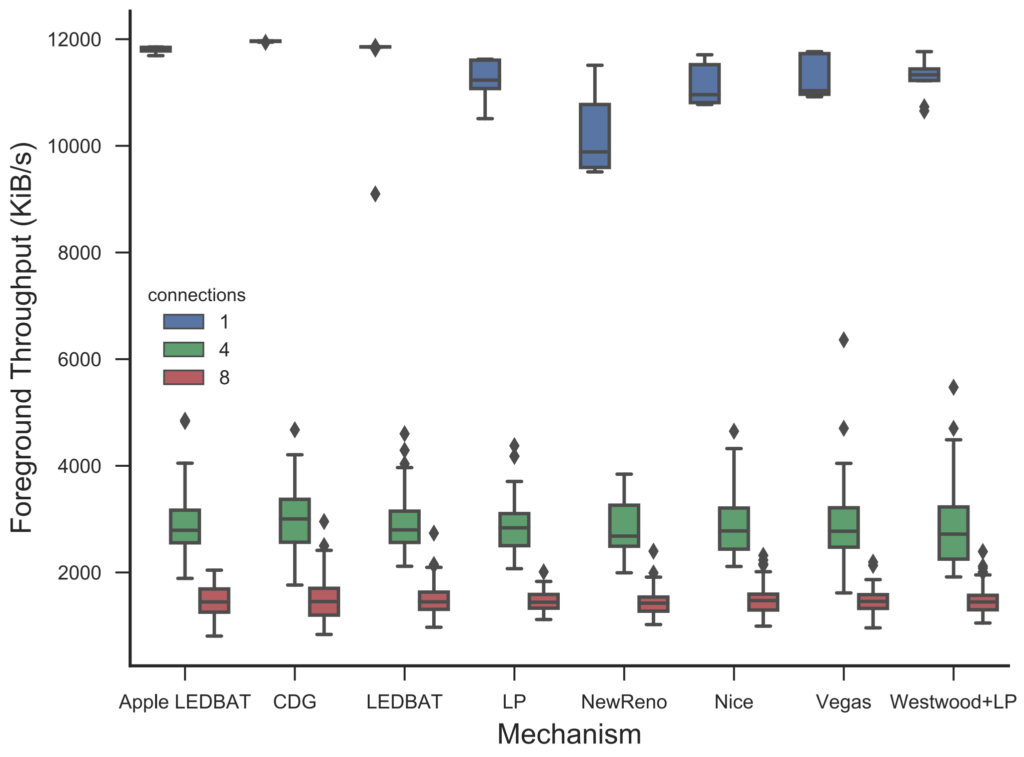


Figure 4.22.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE upload over an 100 Mbps/100 Mbps bottleneck link with 50 ms fixed-path delay.

LEDBAT and Apple LEDBAT demonstrated greater reductions to queuing delay (mean reductions of 21.1% and 17.4%, respectively). In particular, these reductions were much greater when the number of concurrent foreground transfers was increased (24.4%, 14%, and 4.4% for LEDBAT, Apple LEDBAT, and LP with four foreground transfers, respectively).

While Vegas achieved a mean reduction of 8.2% to queuing delay in very high delay settings, Nice was observed to increase delay by 9.6% when competing against a single foreground transfer. CDG and Westwood+LP had little effect on queuing delay in any delay setting (mean reductions of 0.3% and 1.1%, respectively).

	FgConn	1		4		8	
Delay	Mechanism	Mean	Stdev	Mean	Stdev	Mean	Stdev
50 ms	NewReno	125	24.0	135	22.8	144	27.4
	LP	112	27.0	132	25.1	141	27.7
	LEDBAT	132	18.2	135	20.6	142	24.0
	Nice	126	23.7	135	24.0	142	27.2
	Westwood+LP	114	27.6	133	22.4	143	26.0
	Apple LEDBAT	135	25.5	138	23.4	144	27.1
	CDG	126	23.7	136	25.4	144	27.4
	Vegas	127	24.4	137	23.3	143	27.0
100 ms	NewReno	244	51.1	256	43.9	269	45.2
	LP	206	64.7	252	60.3	256	56.0
	LEDBAT	261	34.5	257	39.1	266	42.0
	Nice	250	53.0	261	49.7	267	48.6
	Westwood+LP	234	65.9	261	53.7	269	51.3
	Apple LEDBAT	241	41.7	256	44.6	266	55.4
	CDG	241	58.5	264	54.8	270	51.8
	Vegas	249	50.7	260	49.5	270	48.6
200 ms	NewReno	508	143.6	511	115.0	516	111.8
	LP	396	123.7	503	153.9	518	147.1
	LEDBAT	373	82.2	518	79.5	505	95.2
	Nice	483	119.0	516	113.4	522	109.5
	Westwood+LP	479	164.6	512	153.6	525	136.8
	Apple LEDBAT	417	146.9	501	154.5	510	145.6
	CDG	498	164.2	507	146.6	529	134.9
	Vegas	441	110.2	523	114.0	518	108.6
350 ms	NewReno	900	268.0	922	265.9	905	232.5
	LP	584	174.4	882	341.9	871	331.7
	LEDBAT	624	248.6	697	169.2	830	154.5
	Nice	987	329.1	872	226.2	871	204.2
	Westwood+LP	894	261.2	948	326.8	919	284.3
	Apple LEDBAT	600	144.2	793	338.8	861	333.8
	CDG	869	252.6	882	341.7	906	314.1
	Vegas	779	266.2	858	214.4	866	240.0

Table 4.1.: Median of mean and standard deviation for RTTs (in ms) for a LBE download competing against one, four, and eight concurrent foreground TCP transfers using NewReno for a 8 Mbps/1 Mbps bottleneck link.

4.4.2 CUBIC Foreground Traffic

Trends in queuing delay were generally consistent when competing against CUBIC foreground traffic, with similar performance in lower delay settings. In contrast with throughput and queuing delay, the LBE mechanisms presented smaller reductions to queuing delay when competing against CUBIC foreground traffic compared to when in competition with NewReno. Table 4.2 lists the mean and standard deviation of latency for experiments where a single LBE download competed against one, four, and either concurrent foreground TCP transfers using CUBIC.

As with previous experiments using NewReno, LP and Apple LEDBAT demonstrated the greatest reductions to queuing delay in experiments with 350 ms queuing delay (mean reductions of 14.5% and 11.2%, respectively). However, LEDBAT presented smaller improvements (mean reduction of 6.4%, compared to 21.1% when competing against NewReno).

CDG and Westwood+LP provided small reductions in queuing delay with mean reductions of 5.2% and 5.1%, respectively. Both Vegas and Nice also demonstrated very small — and highly similar — improvements to queuing delay (mean reductions of 2.7% and 3%, respectively).

4.4.3 Self-Induced Delay

The impact of each LBE mechanism when competing against itself was examined. Table 4.3 lists the mean delay and standard deviation for each of the LBE mechanisms where two, four, and eight LBE transfers competed over the same bottleneck link.

CDG was the only mechanism to keep delay close to the propagation delay in most experiments. CDG also produced the most consistent delay results, with the lowest standard deviation of delay across all experimental settings. The exception to this consistency were where eight concurrent CDG transfers were running with 50 ms path delay. However, CDG still achieved the second lowest mean delay of the LBE mechanisms in these experiments.

While Vegas and Nice were similarly able to keep delay relatively close to propagation delay in experiments with 50 ms path delay, both mechanisms were unable to do so in high delay settings. The two mechanisms also continued to perform similarly, with mean reductions compared to NewReno of 38% and 33.5%, respectively.

	FgConn	1		4		8	
Delay	Mechanism	Mean	Stdev	Mean	Stdev	Mean	Stdev
50 ms	CUBIC	142	13.7	147	17.6	152	23.1
	LP	125	22.2	141	21.6	150	21.7
	LEDBAT	136	16.5	145	15.5	150	16.8
	Nice	137	12.6	145	16.9	151	20.3
	Westwood+LP	125	21.2	144	15.7	151	20.0
	Apple LEDBAT	139	15.0	145	15.6	151	21.2
	CDG	134	15.9	145	16.4	152	21.7
	Vegas	138	13.7	146	15.6	151	20.3
100 ms	CUBIC	270	25.4	282	35.5	288	39.7
	LP	247	45.4	264	44.4	274	51.0
	LEDBAT	257	36.5	271	34.4	284	38.8
	Nice	257	27.8	274	32.7	285	39.8
	Westwood+LP	248	35.2	271	30.8	286	38.9
	Apple LEDBAT	250	41.6	275	31.7	286	35.5
	CDG	251	39.8	275	31.8	283	39.4
	Vegas	258	29.2	274	33.3	284	39.6
200 ms	CUBIC	539	76.0	555	81.3	561	100.6
	LP	468	104.7	514	103.2	539	93.3
	LEDBAT	498	91.6	526	101.8	544	98.7
	Nice	511	94.9	531	83.8	550	96.7
	Westwood+LP	517	79.2	523	92.0	551	87.0
	Apple LEDBAT	491	94.9	516	99.8	535	93.9
	CDG	495	84.4	518	98.4	545	89.4
	Vegas	515	88.3	536	87.0	554	89.4
350 ms	CUBIC	952	199.5	973	215.4	991	221.2
	LP	746	219.6	858	256.7	893	253.3
	LEDBAT	891	206.1	898	264.3	941	234.3
	Nice	912	199.7	930	205.4	967	232.6
	Westwood+LP	872	208.3	898	255.0	926	268.6
	Apple LEDBAT	778	215.1	899	244.9	916	264.4
	CDG	838	227.7	901	246.4	923	257.0
	Vegas	914	183.3	947	195.2	946	221.4

Table 4.2.: Median of mean and standard deviation for RTTs (in ms) for a LBE download competing against one, four, and eight concurrent foreground TCP transfers using CUBIC for a 8 Mbps/1 Mbps bottleneck link.

LP kept delay consistent to that inflicted by NewReno in the lowest path delay setting (50 ms), but demonstrated moderate reductions as delay increased. In experiments with path delay of 200 ms or higher, LP achieved a mean reduction of 29.9%. However, LP was also consistently observed to have high standard deviation in the delay experienced by the transfers and a mean CV of 0.44.

Westwood+LP also demonstrated similarly poor consistency in the delay experienced by LBE transfers, with a mean CV of 0.5. However, Westwood+LP consistently achieved greater reductions to the delay experienced than LP (mean reduction of 26.6% compared to NewReno across all experiments).

Apple LEDBAT kept delay consistent with that inflicted by NewReno in experiments with 50 ms path delay. Consistent with the results when LBE mechanisms competed against foreground traffic, Apple LEDBAT demonstrated larger delay decreases in experiments with high path delay (≥ 200 ms).

LEDBAT slightly increased the queuing delay in experiments with low path delay (50 ms), but reduced delay in higher delay settings. However, LEDBAT still consistently inflicted the greatest amount of delay on LBE transfers across all experimental settings.

4.5 Intra-Protocol Fairness

Finally, the ability of LBE mechanisms to fairly share bandwidth amongst multiple homogenous transfers was examined. To do so, experiments were carried out with two, four, and eight concurrent LBE transfers. These experiments are described in Section 3.3.

Figure 4.23 plots the Jain's fairness indices for LBE transfers where the mechanisms were subjected to 50 ms of fixed-path delay grouped by the number of concurrent LBE transfers. CDG and Westwood+LP shared available bandwidth fairly across the same bottleneck link. LEDBAT also shared bandwidth relatively fairly.

Interestingly, LP appeared unable to fairly share bandwidth between multiple LP flows. This relatively poor fairness appeared to be the result of the latecomer flows being starved of bandwidth. Apple LEDBAT, Nice, and Vegas also demonstrated poor intra-protocol fairness.

	BgConn	2		4		8	
Delay	Mechanism	Mean	Stdev	Mean	Stdev	Mean	Stdev
50 ms	NewReno	124	26.3	132	25.7	141	28.6
	LP	121	30.1	126	28.9	135	34.4
	LEDBAT	150	15.8	144	19.3	145	21.0
	Nice	64	13.4	86	19.4	129	18.9
	Westwood+LP	102	39.4	102	39.4	103	40.9
	Apple LEDBAT	127	30.2	135	23.8	142	26.3
	CDG	61	9.0	85	15.4	125	22.4
	Vegas	63	9.3	88	15.8	132	21.8
100 ms	NewReno	242	58.3	251	55.9	264	53.9
	LP	171	108.4	169	83.2	222	77.2
	LEDBAT	220	48.1	221	50.4	241	62.1
	Nice	144	42.4	144	46.3	193	46.2
	Westwood+LP	201	81.2	197	82.4	190	82.0
	Apple LEDBAT	172	56.3	189	63.3	229	62.4
	CDG	105	7.1	111	11.0	133	19.9
	Vegas	125	37.7	144	31.4	185	42.7
200 ms	NewReno	467	144.1	496	138.2	509	130.8
	LP	344	170.9	320	206.0	327	184.4
	LEDBAT	397	150.9	371	151.6	365	147.6
	Nice	342	86.3	276	72.9	350	100.7
	Westwood+LP	362	172.8	373	185.3	357	181.0
	Apple LEDBAT	282	152.5	297	156.9	329	148.4
	CDG	203	6.4	206	10.3	211	15.1
	Vegas	322	106.4	255	43.4	315	75.5
	NewReno	725	287.9	894	288.3	878	284.0
350 ms	LP	561	189.9	574	274.6	571	336.0
	LEDBAT	750	315.8	643	341.0	606	326.2
	Nice	526	104.1	600	103.1	557	99.6
	Westwood+LP	485	384.1	533	368.4	573	358.1
	Apple LEDBAT	494	395.5	495	358.8	501	338.7
	CDG	353	5.6	354	8.7	358	15.0
	Vegas	465	133.5	485	116.7	466	86.7

Table 4.3.: Median of mean and standard deviation for RTTs (in ms) for two, four, and eight concurrent LBE transfers for a 8 Mbps/1 Mbps bottleneck link with 50 ms and 100 ms of fixed-path delay.

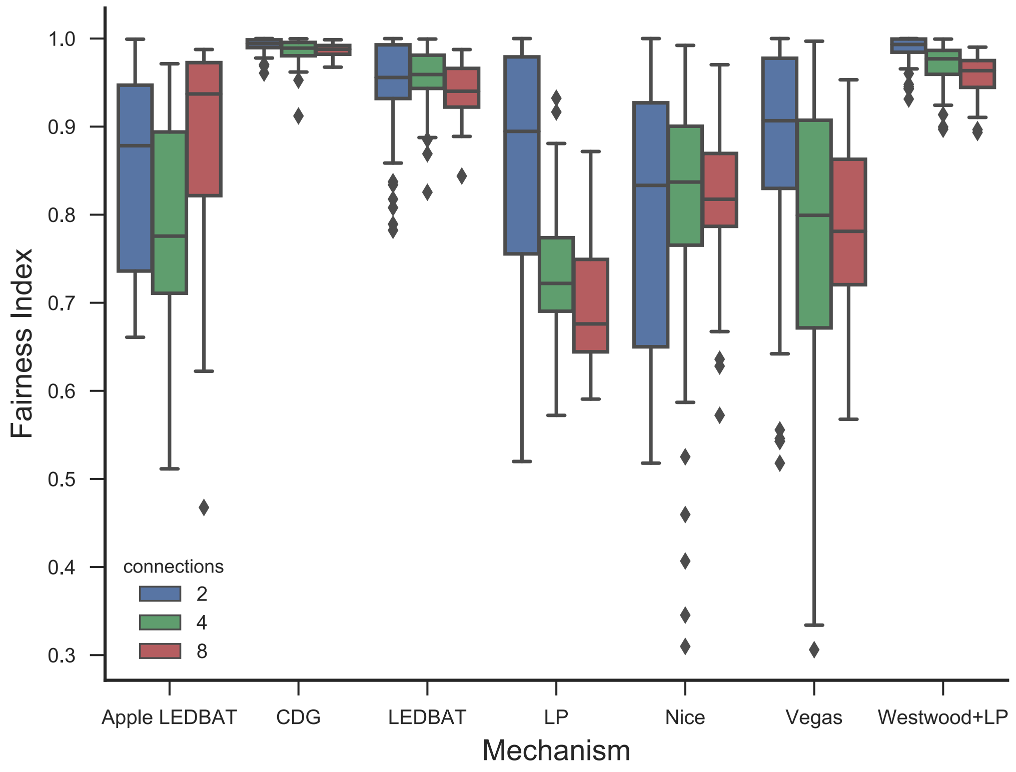


Figure 4.23.: Jain's fairness indices for LBE transfers subjected to 50 ms fixed-path delay grouped by number of concurrent transfers.

Similar results were observed regardless of the number of TCP connections, although all mechanisms became less able to fairly share bandwidth as the number of concurrent LBE transfers increased.

However, with the exception of LP and Vegas, the relative fairness of the LBE mechanisms remained largely unchanged regardless of the number of concurrent connections. The LP latecomer unfairness issue was exacerbated by the increased number of concurrent transfers, leading to poorer fairness than was observed with other mechanisms.

While these performance trends were further exaggerated at higher link speeds, as shown in Figure 4.24, the relative fairness of the LBE mechanisms was unchanged by variation in bottleneck link speed. However, Apple LEDBAT was observed to more fairly share bandwidth, relative to the other LBE mechanisms, over a 100 Mbps bottleneck link.

The impact of fixed-path delay on intra-protocol fairness was also examined. The results, shown in Figure 4.25, indicate that intra-protocol fairness for the LBE mech-

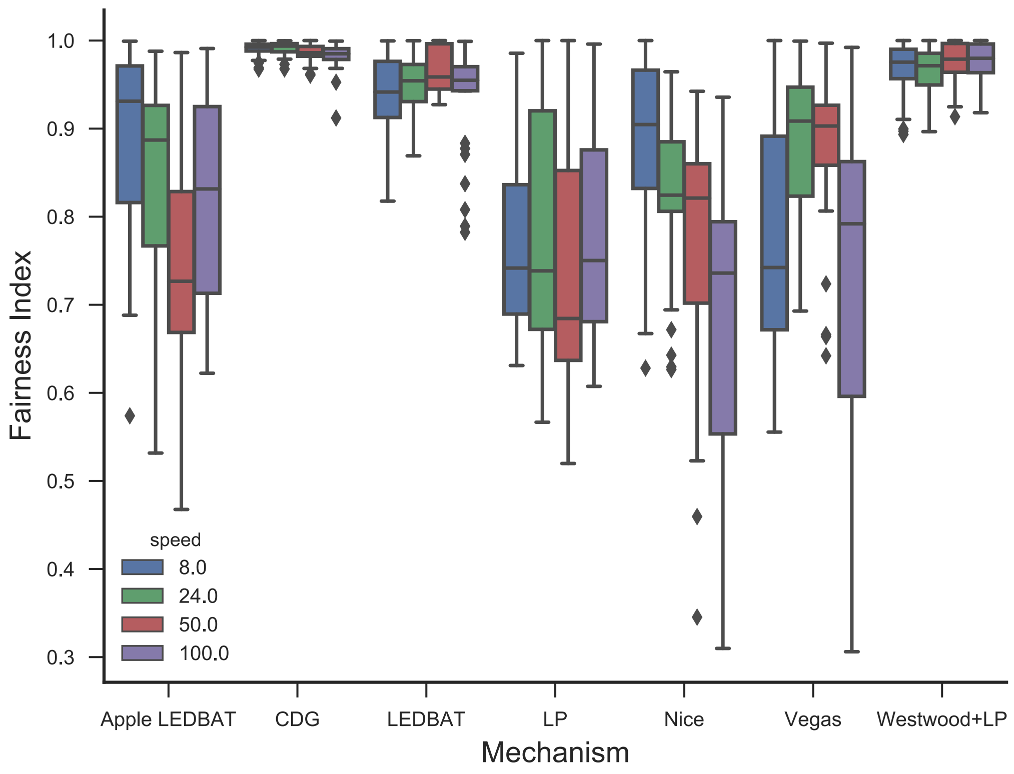


Figure 4.24.: Jain's fairness indices for LBE transfers subjected to 50 ms fixed-path delay grouped by bottleneck link speed.

anisms is reduced as additional delay is introduced. For experiments with 50 ms and 100 ms fixed-path delay, the performance of the mechanisms remained consistent relative to one another. However, CDG became less fair than LEDBAT at 350 ms of fixed-path delay.

4.6 Discussion

The results of this evaluation suggest two primary groupings of LBE mechanism behaviours: regular TCP-like mechanisms, and low-impact mechanisms. The first group includes LP and LEDBAT, while the remaining mechanisms — Nice, Westwood+LP, Apple LEDBAT, CDG, and Vegas — were classed as low-impact. Two sub-groups of low-impact mechanisms were also identified, and are discussed later in this section.

Regular TCP-like mechanisms behave similarly to regular TCP congestion control and provide similar throughput to NewReno and CUBIC. These mechanisms demon-

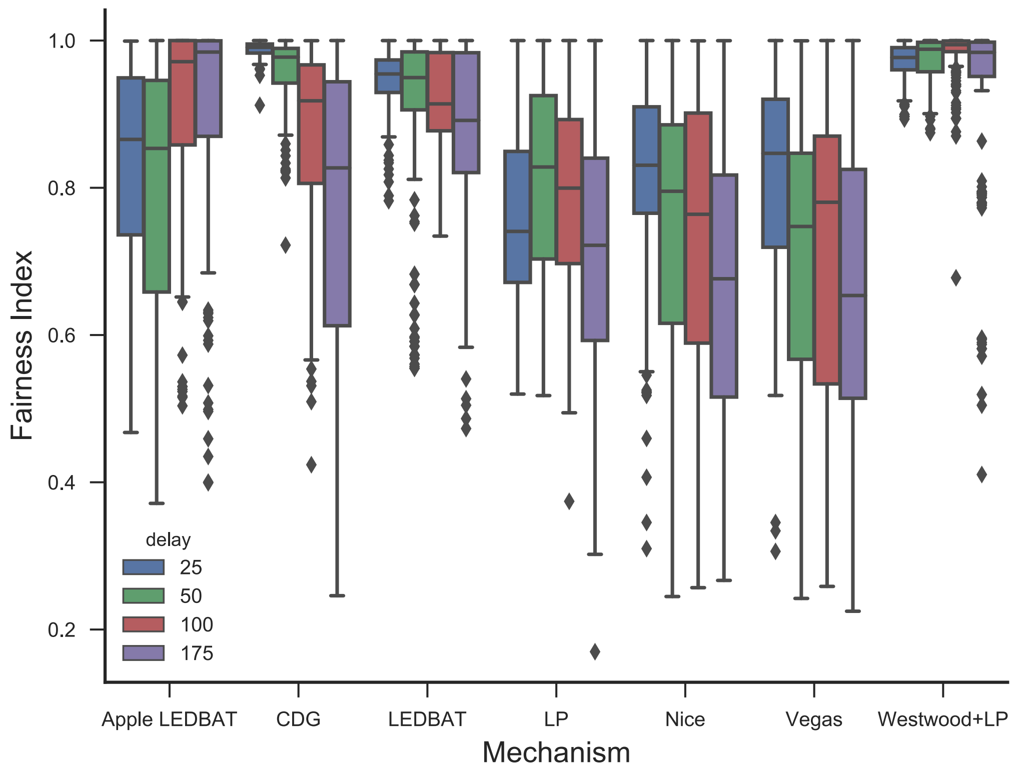


Figure 4.25.: Jain's fairness indices for LBE transfers grouped by fixed-path delay values.

strate no benefit to foreground transaction times when competing against NewReno, with LEDBAT frequently increasing the time required for foreground transfers to complete. Small reductions to foreground transaction times were observed when mechanisms in this group compete against CUBIC. Performance of mechanisms in this first group diverges in high delay settings.

Improvements in foreground transaction time for LP in this study were typically greater than those observed by Tsugawa, Hasegawa, and Murata [22]. However, the impact of LP on foreground traffic was observed to be highly variable. This variation was particularly evident in the results shown in Figure 4.1, where the foreground transaction time for LP ranged from 5.6 seconds to 11.1 seconds. These variations appear to be the result of LP only partially reducing the size of $cwnd$ in response to the foreground transfers; a response more consistent with NewReno.

Although LP demonstrated reductions to the latency for a single foreground transfer, these reductions were not evident as the number of concurrent foreground transfers increased. The impact of LP on latency for background traffic was also relatively small compared to using NewReno.

LP had a reduced impact on foreground traffic in experiments with 350 ms path delay over an 8 Mbps bottleneck link, with mean reduction in foreground transaction time of 37% compared to NewReno. However, these improvements were not evident over high-speed bottleneck links where LP experienced a 40% reduction of mean background throughput.

LEDBAT increased the delay experienced by both foreground and background traffic in many of the experiments conducted. These increases in delay corresponded with the 100 ms delay target used by LEDBAT. This finding, along with the limited reduction in impact to foreground traffic when competing with NewReno and CUBIC, is consistent with previous studies on the impact of LEDBAT on competing TCP flows [13], [37]. Eclipse [8] and FLOWER [9] have attempted to address the aggression of LEDBAT, but are yet to be implemented for the Linux kernel.

The performance of LEDBAT was similar to that of NewReno in high delay settings ($\geq$ 200 ms). However, LEDBAT reduced foreground and background delay when competing against NewReno in the highest delay setting (350 ms).

The low-impact group of LBE mechanisms consistently allowed foreground transfers to complete substantially faster than regular TCP and regular TCP-like LBE mechanisms. This group can be further sub-divided into mechanisms that achieve throughput similar to that of regular TCP (Nice and Vegas) and those that experience noticeable throughput reductions (Westwood+LP, Apple LEDBAT, and CDG). Figure 4.26 summarises these categorisations.

The performance of the low-impact LBE mechanisms further diverges in high delay settings, with the throughput reductions of the latter sub-group (comprised of Westwood+LP, Apple LEDBAT, and CDG) exacerbated as path delay increases while Vegas and Nice become more closely aligned with the more aggressive LBE mechanisms. These changes are summarised in Figure 4.27.

Of the low-impact mechanisms, CDG appears to have the least impact on regular TCP transfers, reducing foreground transaction time at 50 ms path delay by an average of 35.2% and 49.6% compared to NewReno and CUBIC. Consistent with findings by Armitage and Khademi [38], CDG avoided excessive queuing delay with median delay for background traffic remaining close to the fixed propagation delay used in the testbed. Foreground traffic competing against CDG-managed transfers generally experienced similar delay to when only regular TCP congestion control was used.

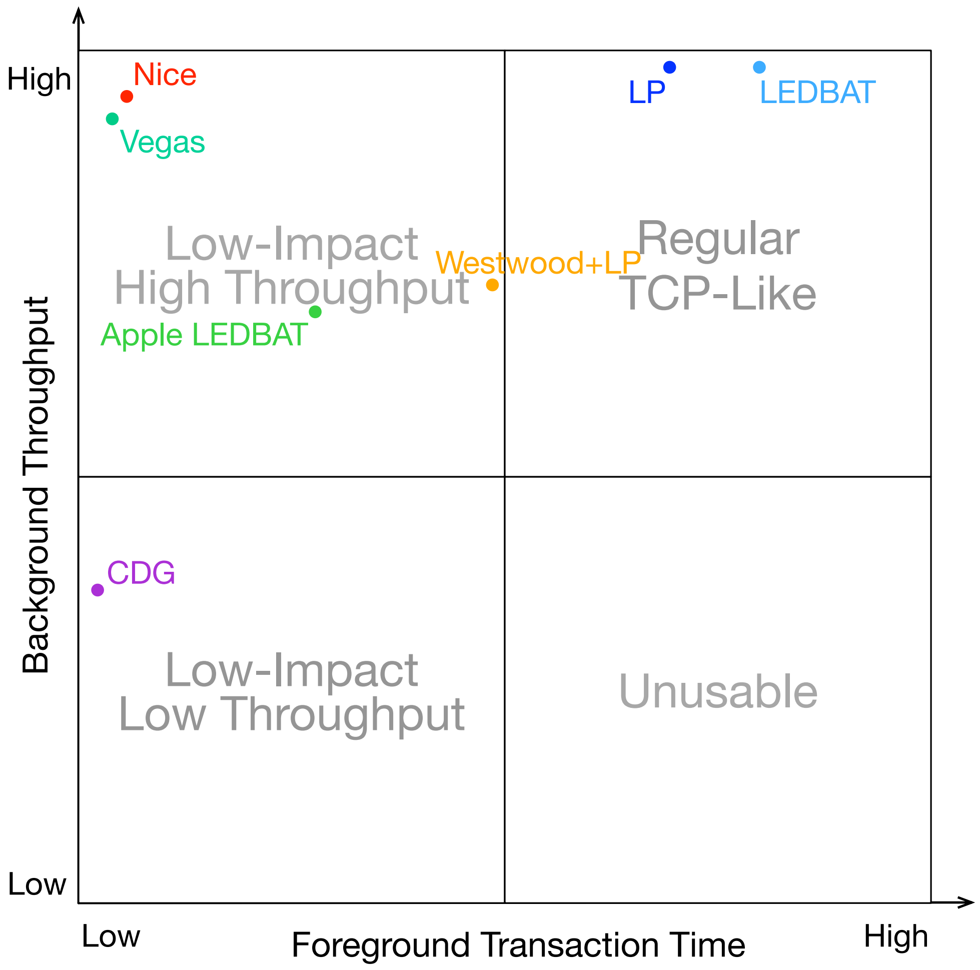


Figure 4.26.: Categorisation of LBE congestion control mechanisms at 50 ms path delay.

CDG's reduced impact on foreground traffic and delay was accompanied by a mean reduction in throughput of 16.1% when competing with foreground traffic for downlink bandwidth. This reduction was observed to increase substantially at very high delay settings, with mean reductions of 77.5% and 67.2% compared to NewReno and CUBIC, respectively. CDG also had a tendency to under-utilise bandwidth when there was no competing traffic and when used over high speed links.

Based on its performance, CDG may be suitable for use in applications where the impact of background traffic must be minimised, propagation delay is expected to be relatively low, and reduced throughput can be tolerated. However, CDG consistently demonstrated some variation in both foreground transaction time and background throughput. This variation was exacerbated by the use of high speed links,

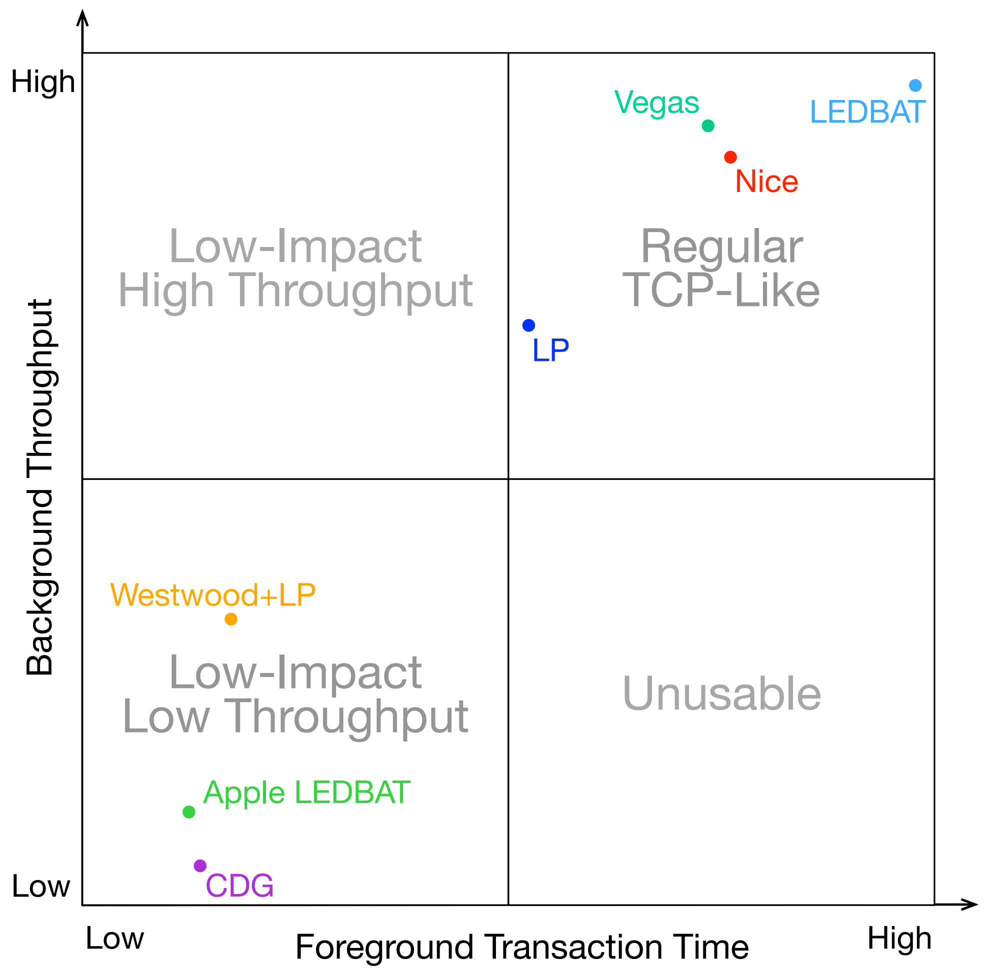


Figure 4.27.: Approximate categorisation of LBE congestion control mechanisms at 350 ms path delay.

and may be attributed to the probabilistic element used to determine whether to reduce $cwnd$ and may warrant further investigation.)

The presence of Apple LEDBAT in the low-impact group suggests that the fixed $\frac{1}{8}$ reduction to $cwnd$ when the delay target is exceeded at least partially addresses the aggression observed with RFC6817 LEDBAT. This modification allowed Apple LEDBAT to apply larger reductions to $cwnd$ than RFC6817 LEDBAT, which was evident in the 9.3% reduction to mean foreground transaction time when competing with NewReno. Apple LEDBAT also achieved a 21% reduction when competing with foreground traffic using CUBIC. These reductions were accompanied by a small (3.52%) reduction in throughput. Apple LEDBAT also slightly reduced the delay experienced by background traffic when competing against NewReno and CUBIC.

Westwood+LP appears to have a moderate impact on foreground traffic while providing slightly lower throughput than Apple LEDBAT in most scenarios. Despite the small mean reduction in foreground transaction time relative to other mechanisms, Westwood+LP still managed to provide reductions over NewReno (up to 38%) and CUBIC (up to 40.8%). The NewReno reduction is broadly consistent with the results presented in the proposal for Westwood-LP [6]. Westwood+LP also reduced background delay when competing against CUBIC (mean reduction of 25%), although it provided no appreciable benefits over NewReno.

As shown in Figure 4.26, Vegas and Nice had similarly low impact on foreground traffic to CDG without the reduction in background throughput associated with other low-impact LBE mechanisms. The performance of these two mechanisms was highly similar, with foreground transaction time and throughput slightly lower for Vegas. This similarity was expected, given that Nice is based on the Vegas algorithm.

In achieving low foreground transaction time, the results for Nice were also partially consistent with prior testing [5], [22]. However, the throughput penalties observed for Nice in previous studies was not evident. This difference may be a result of changes in the underlying Vegas implementation for Linux used as the basis for this version of Nice, discussed in Section 3.6.

Like CDG, Vegas and Nice were able to keep median background delay close to the fixed propagation delay for experiments with 50 ms path delay. However, additional queuing delay was observed for both mechanisms in higher delay settings.

Both Vegas and Nice more closely aligned with the performance of LEDBAT in high delay settings, although both mechanisms still demonstrated reductions in foreground transaction time (albeit significantly smaller than in low delay settings).

Nice and Vegas consistently displayed low median fairness scores (< 0.9) across all experimental settings. Similar fairness issues with Nice were also observed by Carofiglio *et al.* [4], who found that Nice exhibits fairness indices as low as 0.8.

The increased background throughput in low delay settings, along with the relatively small throughput penalty in very high delay settings make either mechanism viable options when a small increase in impact to foreground traffic (relative to CDG) could be tolerated despite the poor fairness results.

4.6.1 Limitations of the Study

For experiments using short foreground transfers (described in Section 4.2) the total transfer size was fixed throughout all experiments. This fixed size resulted in compression in the performance differences between LBE mechanisms over high speed links due to the increased throughput available, introducing challenges in differentiating between the mechanisms. However, the results of additional experiments with foreground transfers scaled up proportional to the link speed were largely consistent with those at lower link speeds. Given the consistency in results, as well as that foreground transfer file size (2469 KiB) was selected based on the average data transferred from websites, further experiments with larger foreground transfers were not conducted.

In this evaluation, module parameters for LBE mechanisms were based on default values either from the existing Linux kernel modules, or from original proposals. This approach resulted in undesirable performance for LEDBAT, which was observed to behave like a regular TCP congestion control at least partially due to the high default delay target. However, it was considered likely that default parameters would be unchanged if LBE congestion control mechanisms were deployed in production systems. As such, default parameters were used.

The results showed some additional variation within the results of each experiment when using 802.11n networks compared to experiments over wired networks. However, this variation was relatively small due to the stable channel conditions and limited contention from nearby devices. While more realistic channel conditions were considered, it was infeasible to implement them using the testbed network. As such, testing with more realistic channel conditions could instead be carried out alongside Internet-based testing using existing wireless networks.

4.6.2 Recommendations

The findings of this evaluation suggest that Nice, Vegas, and CDG are all strong candidates for use in low priority applications, such as file synchronisation and sharing or uploading of user-generated content. Given that Vegas and Nice provide background throughput comparable to regular TCP, while maintaining low queuing delay, they could be considered for applications where minimising impact on other traffic is desirable but not vital. Conversely, CDG may be more suitable for applications where the impact of background traffic must be minimised and reduced background throughput can be tolerated.

Aside from Nice and Vegas, all low-impact LBE mechanisms suffer significant throughput penalties in high delay settings. Nice and Vegas provide small improvements over NewReno at high delays, but only if the number of foreground traffic flows is low. However, the performance of all LBE mechanisms in high delay settings is suboptimal.

4.7 Summary

This chapter has presented and discussed the results of the evaluation of seven LBE congestion control mechanisms in an emulated network. Of these mechanisms, three — Nice, Westwood-LP, and Apple LEDBAT — were implemented for the Linux kernel to facilitate this evaluation. All mechanisms were evaluated in three realistic scenarios to determine their impact on, and performance relative to, regular TCP congestion control mechanisms such as NewReno and CUBIC.

The findings of this evaluation identified two groups of LBE congestion control mechanism behaviours. LP and LEDBAT typically performed similarly to NewReno and CUBIC, and were classified as regular TCP-like mechanisms. The remaining mechanisms — Nice, Westwood+LP, Apple LEDBAT, CDG, and Vegas — had a lower impact on foreground traffic. The findings further suggest that Nice, Vegas, and CDG are strong candidates for use in low priority applications, such as file synchronisation and sharing or uploading of user-generated content. However, the reduced throughput of CDG may not be acceptable in some scenarios.

Given the significant throughput penalties observed for most low-impact LBE mechanisms, as well as the limited improvements offered by Nice and Vegas, the remainder of this study will focus on the development of a new LBE congestion control mechanism designed to improve the throughput achieved by low-impact LBE congestion control mechanisms in high delay settings.

Yield TCP

5

5.1 Overview

The results of the evaluation described in Chapter 4 identified three promising LBE congestion control mechanisms: CDG, Nice, and Vegas. However, these mechanisms demonstrated limitations in some scenarios. Nice and Vegas had a low impact on foreground traffic with no appreciable decrease to throughput in low-delay settings, but increased foreground transaction time significantly when additional path delay was introduced. CDG demonstrated the lowest throughput of the evaluated mechanisms at low delay, and this was exacerbated in high-delay settings.

This chapter describes the development of a new LBE congestion control algorithm designed to address these limitations: Yield TCP. Yield was designed to fulfil the second aim of this research by addressing the throughput penalties associated with low-impact LBE mechanisms in high fixed-path delay scenarios while reducing the impact on foreground traffic over TCP-like mechanisms.

The remainder of this chapter is structured as follows. Section 5.2 describes the motivation and design goals for Yield. Section 5.3 describes the design of Yield, while also detailing how this algorithm addressed the stated goals. Section 5.4 describes the operation of Yield, as well as the implementation of Yield for Linux. Section 5.5 outlines the methodology used to evaluate Yield. Section 5.6 presents the results of the evaluation. Section 5.7 discusses the performance of Yield in relation to existing mechanisms and the stated design goals, as well as potential avenues through which Yield could be improved in future work. Finally, Section 5.8 presents a summary of Yield and its performance.

5.2 Yield TCP Motivation

The results of the LBE congestion control evaluation — described in Chapter 4 — identified that existing mechanisms such as Nice and CDG can provide significant improvements in low-delay settings. In high-delay settings, these mechanisms can be divided into two performance profiles:

1. Low impact on foreground traffic with very low throughput (CDG, Apple LEDBAT, Westwood+LP)

2. Similar impact on foreground traffic and throughput to regular TCP (LEDBAT, LP, Vegas, Nice[1])

Given the lack of clear improvements to foreground transaction times provided by Nice at high delay settings, as well as the poor intra-protocol fairness, an opportunity was identified to develop a new LBE congestion control mechanism that would provide a better balance between these performance metrics.

Yield TCP was therefore designed to improve the throughput achieved by low-impact LBE congestion control mechanisms, while reducing the impact on foreground traffic compared to Nice in high delay settings. The intention was to achieve these improvements without significant performance regression in low-delay settings (when compared with Vegas, Nice, and CDG). Improved fairness was also considered desirable, but not an explicit design goal. However, fairness should not regress when compared to Nice.

5.3 Yield TCP Design

Yield utilises elements of control theory, specifically the Proportional-Integral (PI) controller, to better interpret and respond to changes in queuing delay. This PI controller uses two properties — proportional and integral — to adjust its response to increasing queuing delay [57]. This approach has previously been utilised by the Proportional-Integral Controller Enhanced (PIE) active queue management scheme [58].

[1]Nice and Vegas slightly reduce impact on foreground traffic. However, these reductions were sufficiently small that these mechanisms should be considered part of the regular TCP-like mechanisms.

The basic operation of Yield utilises a binary decision-making process, similar to that of Apple LEDBAT, based on a delay target and an estimation of queuing delay. Yield applies an additive increase to $cwnd$ when under the delay target, while utilising a multiplicative decrease when over target. This multiplicative decrease is based on the proportional characteristic of the PI controller, as the size of decreases will be scaled based on the target and current queuing delay.

Unlike existing algorithms that utilise a fixed multiplicative decrease, such as Apple LEDBAT, Yield uses changes in delay estimates over time to modify the extent to which $cwnd$ is reduced when the delay target is exceeded. In considering changes in delay estimates over time, Dynamic Trend-Based Reduction satisfies the integral characteristic of the PI controller. The technique used to track and consider these changes is described further in Section 5.3.1.

As with other recent LBE congestion control algorithms [8], [9], [15], Yield utilises OWD for delay estimation. Use of OWD estimates prevents Yield from erroneously responding to cross-traffic in the reverse direction [3].

In total, Yield implements three components to achieve the design goals identified in Section 5.2:

1. Dynamic Trend-Based Reduction
2. Adaptive Delay Targeting
3. Cross-Traffic Detection

Each of these components are described in the subsections below.

5.3.1 Dynamic Trend-Based Reduction

The primary point of differentiation between Yield and existing LBE congestion control mechanisms is the use of delay signals to adjust the size of the multiplicative decrease applied when the queuing delay exceeds the delay target. Specifically, Yield considers the change in queuing delay estimates to determine whether to increase or decrease the size of the next $cwnd$ reduction to be applied.

The size of this decrease is controlled by a reduction factor to control the size of the next $cwnd$ adjustment. The reduction factor (r) is similar to that used by Apple LEDBAT and is used to reduce the size of $cwnd$ by:

$$cwnd = cwnd - \frac{1}{2^r} \cdot cwnd \tag{5.1}$$

The values of r are bounded between 0 and 5 to prevent negative or small ineffectual $cwnd$ reductions from being applied. This action is implemented using a right-shift bit operation on the current $cwnd$, represented by:

$$cwnd = cwnd - (cwnd \gg r) \tag{5.2}$$

To ensure that an appropriate reduction is applied, Yield updates the reduction factor upon the receipt of every new acknowledgement. This mechanism is partly based on the principles of the PI controller, and uses the change in delay estimates over previous acknowledgements to increase or decrease the reduction factor.

Yield calculates the change in delay on receipt of each new acknowledgement as:

$$\delta = current\_delay - prev\_delay \tag{5.3}$$

where $current\_delay$ represents the most recent OWD estimate, while $prev\_delay$ is the previous delay estimate. The previous delay estimate can be averaged over multiple historical readings to filter random noise. Use of delay smoothing is discussed further in Section 5.4.2.

A positive δ indicates that delay is increasing, and Yield therefore increases the size of the next reduction to be applied (by decreasing the reduction factor). Conversely, a negative δ indicates queuing delay is reducing and Yield decreases the size of the next reduction.

This multiplicative decrease is applied when current queuing delay exceeds the delay target. Upon application of the multiplicative decrease, the reduction factor is incremented to ensure that Yield does not over-react in the presence of congestion.

5.3.2 Dynamic Delay Targeting

In the evaluation of existing LBE congestion control algorithms, described in Chapter 4, LEDBAT was identified as performing similarly to regular TCP congestion control mechanisms. This behaviour was subsequently identified to be caused by the use of a fixed queuing delay target of 100 ms.

The use of a fixed queuing delay target was also previously identified as a design flaw in LEDBAT by Adhari *et al.* [8], and was addressed in the design of Eclipse through the implementation of a dynamic queuing delay target. Yield utilises a dynamic delay target, based on a modified version of the Eclipse formula. As in Eclipse, the queuing delay target is calculated as:

$$target = \beta \cdot (s\_max - s\_min) + s\_min \tag{5.4}$$

where s_min represents a moving average of the minimum queuing delay, which can be overwritten by values lower than the current s_min. β is the early congestion indication threshold. However, in Yield s_max represents the absolute maximum of queuing delay estimates observed by Yield (rather than the moving average of maximum queuing delay [8]). The delay target effectively represents a percentage of the maximum queuing delay observed, as specified by β.

This change to s_max was made during preliminary testing of Yield, in which the value of s_max was observed to converge with that of s_min when competing foreground flows were introduced. This convergence resulted in a decrease to the queuing delay target, permitting Yield to compete for a fairer share of the available bandwidth (counter to the performance expectations of LBE congestion control mechanisms). The values of s_min and s_max are updated when new OWD estimates are calculated on receipt of a new acknowledgement.

5.3.3 Cross-Traffic Detection

In testing early versions of Yield, it was observed that Yield would fail to fully utilise the available bandwidth following the end of a foreground TCP transfer. This issue was partly caused by applications of overly aggressive reductions to $cwnd$ and led to a small difference in throughput once Yield attempted to recover following a foreground TCP transfer. Such a throughput penalty was considered to be detrimental to the design goal to avoid performance regression compared to Nice.

To address this issue, Yield was modified to infer the presence of cross-traffic based on changes in the delay trend. This mechanism infers whether increases in queuing delay are self-induced or the result of competing traffic and adjusts Yield's response accordingly.

Specifically, Yield compares the change in delay over recent history, as described in Section 5.3.1, to a long-term moving average of historic delay trend readings. To ensure responsiveness, particularly in high delay settings, this average is updated upon the receipt of each new acknowledgement.

Yield identifies the presence of cross-traffic by applying a multiplier (referred to as ct_thresh) to the historic delay trend average, based on:

$$trend > ct\_thresh \cdot delay\_trend \tag{5.5}$$

where $trend$ represents the change in queuing delay for the most recent acknowledgement, and $delay\_trend$ represents the historic trend average. When the trend exceeds this cross-traffic threshold, Yield permits more aggressive reductions to $cwnd$ by modifying the lower boundary of the reduction factor.

As a result of this mechanism, two lower boundaries are applied to the Yield reduction factor depending on whether cross-traffic has been detected in the current RTT (referred to as $maxc\_reduction$ and $max\_reduction$ with and without cross-traffic, respectively). Yield applies these lower boundary checks when reducing the reduction factor, with maximum reduction factor (with cross traffic) only applied when cross-traffic detection has been triggered.

Once cross-traffic is detected, the lower reduction factor boundary persists until the end of the current RTT.

5.4 Yield TCP Operation

When a new acknowledgement is received, Yield updates many of the internal variables that control the next $cwnd$ reduction to be applied. A full listing of these actions is presented in Algorithm 1.

Yield first updates the OWD estimate using the TCP timestamps included in the packet headers. This estimate is then used to update the values of s_min and

s_max as described in Section 5.3.2, as well as to calculate the change in delay since the previous acknowledgement was received.

As described in Section 5.3.3, Yield then infers whether it is currently competing with cross-traffic based on the change in delay trend and determines the upper size limit on the next $cwnd$ reduction. The current trend is also used to increase or decrease the size of the reduction factor, as described in Section 5.3.1. Finally, Yield updates the delay history using the current delay estimate.

Algorithm 1 Yield per-acknowledgement operation

```
$current\_delay \leftarrow time - remote\_time$ {update delay estimate based on TCP timestamps}
if $current\_delay < delay\_smin$ then
  $delay\_smin \leftarrow current\_delay$ {overwrite the smoothed minimum}
else
  $delay\_smin \leftarrow update\_delay(current\_delay)$ {update smoothed minimum}
end if
if $current\_delay > delay\_smax$ then
  $delay\_smax \leftarrow current\_delay$ {overwrite the smoothed maximum}
end if
$trend \leftarrow current\_delay - prev\_delay$ {determine whether delay is increasing or decreasing}
if $trend > trend\_hist * ct\_thresh$ then
  $cross\_traffic \leftarrow 1$ {identify potential cross-traffic}
end if
if $trend > 1$ then
  $reduction\_factor \leftarrow reduction\_factor - 1$ {delay is increasing so a bigger decrease will be needed}
else
  if $trend < 1$ then
    $reduction\_factor \leftarrow reduction\_factor + 1$ {delay is decreasing so a smaller decrease will be needed}
  end if
end if
$prev\_delay \leftarrow (prev\_delay + current\_delay)/2$ {current delay reading becomes last seen}
$delay\_trend \leftarrow trend$ {update delay trend history using current}
```

Algorithm 2 presents the per-RTT operation of Yield. During each RTT, Yield recalculates the delay target using the equation described in Section 5.3.2 and determines whether the current queuing delay has exceeded the target. The $cwnd$ is then increased — using the NewReno additive increase — or decreased as per the process described in Section 5.3.1, as appropriate.

Algorithm 2 Yield per-RTT operation

```
$target \leftarrow beta \cdot delay\_smax - delay\_smin$ {update delay target}
$off\_target \leftarrow target - qdelay$ {calculate deviation from delay target}
**if** $off\_target \geq 0$ **then**
  $cwnd \leftarrow cwnd + 1$ {under target, increase cwnd}
**else**
  $decrement \leftarrow cwnd \gg reduction\_factor$
  $cwnd \leftarrow cwnd - decrement$ {over target, decrease cwnd}
  $reduction\_factor \leftarrow reduction\_factor + 1$
**end if**
```

5.4.1 Yield TCP Implementation

To enable the evaluation of Yield in comparison to existing LBE congestion control mechanisms, a prototype was developed for version 4.4.15 of the Linux kernel. This prototype was implemented using the standard TCP congestion control module interface available in the Linux kernel. Source code for this prototype is listed in Appendix B.

The Yield prototype uses elements of existing congestion control modules where possible. The OWD estimation was incorporated from the RFC6817 LEDBAT implementation from [36], while the core congestion avoidance behaviour utilises a modified version of the Apple LEDBAT implementation developed for the evaluation described in Chapter 4. The implementation of Apple LEDBAT was described in Section 3.6.3.

The dynamic delay target was implemented based on the proposal by Adhari *et al.* [8] and adapted as described in Section 5.3.2. The calculation of the delay target accepts β values in increments of 1%.

5.4.2 Module Parameters

Given the experimental nature of Yield, many of the key decision points are implemented as module parameters which can be modified after the module has been loaded (without the need to re-compile the kernel module). Table 5.1 lists the available module parameters, with more complete descriptions below.

In many cases, the optimal default value was determined through a preliminary series of experiments using the simultaneous downloads scenario, described in Section 3.3, with 25 ms and 175 ms fixed path delay. In these experiments, 10 runs

Name	Parameter	Description	Default
Beta	beta	Percentage of maximum queuing delay for delay target	15
Minimum Reduction Factor	min_reduction	Minimum reduction_factor	5
Maximum Reduction Factor	max_reduction	Maximum reduction factor without cross-traffic	3
Maximum Reduction Factor	maxc_reduction	Maximum reduction factor with cross-traffic	0
Cross-Traffic Threshold	ct_thresh	Multiplier applied to trend history to infer presence of cross-traffic.	2
Delay History Factor	hist_factor	Bit shift to be applied to average of delay history	2
Delay Trend Factor	trend_factor	Averaging factor to be applied to delay history (for cross-traffic detection)	128
Increase Threshold	increase_threshold	Smallest delay increase required to increase reduction factor	1
Decrease Threshold	decrease_threshold	Smallest delay decrease required to decrease reduction factor	-1

Table 5.1.: Module parameters for Yield implementation.

were carried out with each setting. The results for these experiments are presented in the subsections below.

Beta

Beta (β) represents the percentage of maximum queuing delay to be used in the calculation of the delay target. Consistent with the setting used by Eclipse [8], and based on the recommendation by Kuzmanovic and Knightly [59], a default value of 15% is used. Beta can be set in increments of 1%.

Minimum Reduction Factor

The minimum reduction factor specifies the smallest reduction that can be applied to $cwnd$ as a number of bits to be right-shifted. This bounding prevents Yield from applying an ineffectually small reduction. By default, the minimum reduction is set to 5, representing a reduction of $\frac{1}{32}$.

Maximum Reduction Factor (without Cross-Traffic)

The maximum reduction factor specifies the largest reduction that can be applied to $cwnd$ when no cross-traffic has been detected as a number of bits to be right-shifted. This upper boundary prevents Yield from making excessive $cwnd$ reductions when no cross-traffic is present.

Preliminary testing using different maximum reduction factors, results for which are shown in Table 5.2, indicated that the maximum reduction factor had limited impact on throughput and foreground transaction time at low delay settings. However, a default value of 3 was selected due to the noticeable decrease in foreground transaction time in high delay settings. This setting represents a $cwnd$ reduction of $\frac{1}{8}$, consistent with that used by Apple LEDBAT, when no cross traffic is detected.

Delay	50 ms		350 ms	
MaxRed	FTT	BgTput	FTT	BgTput
0	4.1	902	9.3	816
1	4.1	919	9.5	820
2	4.1	905	11.1	834
3	4.2	904	6.9	742

Table 5.2.: Median throughput (in KiB/s) and foreground transaction time (in seconds) for a download over 8 Mbps/1 Mbps bottleneck link with a single foreground TCP transfer using NewReno with different maximum reduction factors.

Delay	50,ms		350,ms	
ct_threshold	FTT	BgTput	FTT	BgTput
1	3.5	902	7.7	724
2	4.2	904	6.9	742
3	4.0	901	7.7	729

Table 5.3.: Median throughput (in KiB/s) and foreground transaction time (in seconds) for a download over 8 Mbps/1 Mbps bottleneck link with a single foreground TCP transfer using NewReno with different cross-traffic detection thresholds.

Maximum Reduction Factor (with Cross-Traffic)

The maximum reduction factor specifies the largest reduction that can be applied to $cwnd$ when Yield has detected the presence of cross-traffic as a number of bits to be right-shifted. The maximum reduction with cross-traffic is 0 by default, representing a complete collapse of $cwnd$ (to 0). However, $cwnd$ will be reset to 2 segments before the end of the RTT.

Cross-Traffic Threshold

As described in Section 5.3.3, the cross-traffic threshold is used by Yield to infer the presence of competing traffic. This multiplier is applied to the change in delay trend. Based on the results of preliminary testing, shown in Table 5.3, the default multiplier is 2, requiring the current change in delay to be double the historic delay trend before Yield acknowledges the presence of cross traffic.

Delay	50 ms		350 ms	
HistFactor	FTT	BgTput	FTT	BgTput
0	3.9	866	7.5	782
1	4.2	892	6.9	699
2	4.2	904	6.9	742
3	3.8	864	7.4	623

Table 5.4.: Median throughput (in KiB/s) and foreground transaction time (in seconds) for a download over 8 Mbps/1 Mbps bottleneck link with a single foreground TCP transfer using NewReno with different smoothing factors for delay history.

Delay History Factor

The delay history factor is the averaging factor applied to the exponentially weighted moving average of previous delay readings, as a number of bits to be right-shifted. Smaller values represent shorter delay history, making Yield more sensitive to changes in delay. A larger averaging factor ensures that older delay estimates are considered, but results in Yield being slower to respond to competing traffic. This moving average is applied as:

$$avg = \left(1 - \frac{1}{2^f}\right) \cdot avg + \frac{1}{2^f} \cdot delay \tag{5.6}$$

where f represents the averaging factor, while $delay$ is the current delay estimate and avg is the previous average. The results of preliminary testing, shown in Table 5.4, indicates that use of no delay smoothing (HistFactor = 0) resulted in lower foreground transaction time, at the expense of throughput in the low delay case. Foreground transaction time was also increased when no historical averaging was used in the high delay setting. As such, the default averaging factor for Yield is 2, representing a $\frac{1}{4}$ exponentially weighted moving average.

Delay Trend Factor

The delay trend factor is the averaging factor applied to the exponentially weighted moving average of previous delay readings, as the denominator of the factor to be applied. Smaller values represent shorter delay history, making Yield more sensitive to changes in delay. A larger averaging factor ensures that older delay estimates are considered, but results in Yield being slower to respond to competing traffic. This moving average is applied as:

Delay		50ms		350ms	
HistFactor	TrendFactor	FTT	BgTput	FTT	BgTput
0	8	3.5	850	6.7	659
	64	3.7	844	7.4	619
	128	3.9	866	7.5	782
	256	3.8	846	7.5	789
2	8	4.0	900	6.9	630
	64	3.4	832	7.5	785
	128	4.2	904	6.9	742
	256	3.4	825	7.5	701

Table 5.5.: Median throughput (in KiB/s) and foreground transaction time (in seconds) for a download over 8 Mbps/1 Mbps bottleneck link with a single foreground TCP transfer using NewReno with different smoothing factors for delay trend.

$$avg = \left(1 - \frac{1}{f}\right) \cdot avg + \frac{1}{f} \cdot delay \tag{5.7}$$

where f represents the averaging factor, while $delay$ is the current delay estimate and avg is the previous average. In preliminary testing, the results for which are shown in Table 5.5, use of short-term trend averaging led to good foreground transaction time and throughput in low delay settings. However, these results were at the expense of throughput in high delay experiments. As improvement in high delay settings was one of the stated goals of Yield, a default trend averaging factor of 128 was selected, representing a $\frac{1}{128}$ moving average.

Increase Threshold

The increase threshold specifies the smallest delay increase (in milliseconds) required to trigger an increase of the size of the next $cwnd$ reduction (by decreasing the reduction factor). The default value is set to 1 ms.

Decrease Threshold

The decrease threshold specifies the smallest delay decrease (in milliseconds) required to trigger an increase of the size of the next $cwnd$ reduction (by decreasing the reduction factor). The default value is set to -1 ms.

5.5 Evaluation Methodology

To examine the performance characteristics of Yield relative to existing LBE congestion control mechanisms, a series of experiments was carried out in an experimental testbed over wired and wireless links. The methodology for these experiments was heavily based on that used in the evaluation of existing LBE congestion control, but only the 8 Mbps and 50 Mbps bottleneck link speeds were included due to the limited differences observed with different speeds.

A summary of the methodology is presented in the subsections below, with full descriptions in Chapter 3.

5.5.1 Experimental Setup

The topology for these experiments included three hosts, each running OpenSUSE Leap 42.1 with kernel version 4.4.15. Each host PC was connected via a Gigabit Ethernet access link to its local switch. This topology is shown in Figure 5.1.

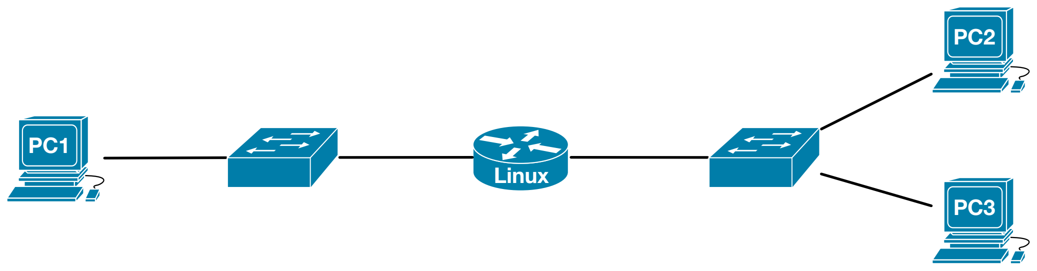


Figure 5.1.: The experimental wired network topology.

In experiments utilising wireless networks, PC1 was connected via 802.11n to an Ubiquiti Networks UniFi UAP AC Pro as shown in Figure 5.2. As in the wired experiments, the other host PCs were connected via a Gigabit Ethernet access link to the local switch. The wireless channel conditions described in Section 3.2 were also used in this evaluation.

A Linux-based router, running OpenSUSE 13.2 with kernel 3.17.4, was used to route packets between hosts and emulate different bottleneck link speeds and delay settings. The host PCs operated interchangeably as TCP senders and receivers, dependent on which of the three scenarios described in Section 5.5.3 was being considered.

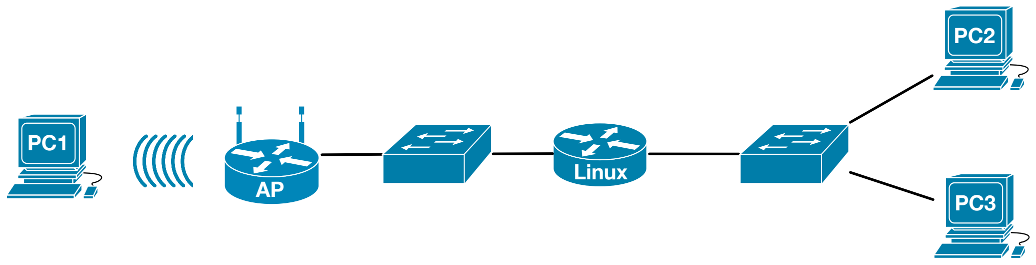


Figure 5.2.: The experimental wireless network topology.

Downlink Speed	Uplink Speed	Path Delay	Buffer Size
(Mbps)	(Mbps)	(ms)	(Packets)
8	1	50	70
		100	135
		200	270
		350	470
50	20	50	420
		100	835
		200	1670
		350	2920

Table 5.6.: Bottleneck link speeds and path delay values.

5.5.2 Bottleneck Link

Due to the limited differences in performance observed when bottleneck link speeds were varied described in the evaluation in Chapter 4, particularly when the amount of foreground data was increased, the evaluation of Yield only included two bottleneck link speeds: 8 Mbps/1 Mbps and 50 Mbps/20 Mbps. The bottleneck buffer was set to double the BDP and was calculated based on the downlink speed.

As with previous experiments, OWD delay values of 25 ms, 50 ms, 100 ms, and 175 ms were used to examine the effect of high fixed-path delay on LBE congestion control. Path delay for all experiments was symmetric, and is subsequently described using the total propagation delay for the path (RTT). Table 5.6 lists all link speed, fixed-path delay, and buffer size combinations considered.

5.5.3 Testing Scenarios

Consistent with the previous evaluation, three testing scenarios were included in this evaluation:

Competing Downloads: a long-lived LBE transfer that competed against one or more regular TCP flows. PC2 and PC3 acted as TCP senders for the foreground traffic and the long LBE transfer, respectively. PC1 was the receiver for all connections.

Simultaneous Upload / Download: a large file upload competing against one or more HTTP transfers. PC2 was the TCP sender for foreground traffic directed to PC1. However, PC1 acted as the sender for the LBE traffic, which was received by PC3.

Fairness: PC2 and PC3 initiated long-lived background transfers to PC1. The number of transfers was split evenly between the two senders.

The former two scenarios were run with both short and long-lived background transfers. These scenarios are described in greater detail in Section 3.3.

5.5.4 Test Data

The data transferred between the testbed hosts was split into two categories: foreground traffic and background traffic.

Foreground traffic was generated using httperf, with transfers of 2469 KiB. Additional experiments were also conducted with long-lived foreground transfers generated using iPerf for the duration of each experiment. These transfers were split equally over 1, 4, and 8 TCP connections.

Background traffic was generated using iPerf for the duration of each experiment. Intra-protocol fairness was examined using 2, 4, and 8 concurrent background transfers with no competing foreground traffic.

5.5.5 Metrics

Consistent with the broader evaluation of LBE congestion control, described in Chapters 3 and 4, Yield was evaluated based on four primary metrics:

Foreground Transaction Time: The time required for the foreground transfer to be completed.

Background Throughput: The average rate at which a network device is able to send data end-to-end over the network over the duration of the transfer. Only reported for the background LBE transfers.

Latency: Time between transmission of a packet and receipt of the associated acknowledgement. Retransmitted segments were excluded from these calculations.

Fairness: Impact of a TCP stream on the transmission capacity of other TCP streams operating over the same communications medium, reported using Jain's Fairness Index [47].

All metrics were calculated using a Python script developed which utilises the Wireshark packet dissection engine. The data analysis approach is described further in Section 3.8.

5.5.6 Included Congestion Control Mechanisms

Of the seven LBE congestion control mechanisms considered in the evaluation described in Chapters 3 and 4, the mechanisms that demonstrated the best performance were selected for comparison with Yield: Nice[2], and CDG.

The default congestion control mechanisms for Linux and FreeBSD — CUBIC and NewReno — were also included, both to provide a comparison against regular TCP as well as for foreground traffic.

[2]Given the similar results observed for Nice and Vegas, only Nice was selected as it was designed specifically as a LBE mechanism.

5.5.7 Data Collection

Each combination of bottleneck link speed, number of foreground and background TCP connections constitutes a single experiment. For each experiment, 10 runs were carried out per congestion control mechanism. Packet traces of all traffic sent and received by the testbed hosts were captured over a 60 second period for each run. The data collection process was automated using TEACUP 1.0 [56].

5.5.8 Data Analysis

The resulting packet captures were automatically processed by a Python script using the Wireshark packet analysis engine. Both sender and receiver-side packet traces are reconciled before being used to calculate throughput, latency, and fairness as described in Section 3.5. Calculations of statistics for latency excluded retransmissions.

5.6 Evaluation Results

Sections 5.6.1 and 5.6.2 examine the impact of Yield on foreground transaction time and throughput, respectively. Section 5.6.3 considers the effect of Yield on delay experienced, both by foreground traffic and when competing against itself. Section 5.6.4 examines the intra-protocol fairness properties of Yield compared to existing LBE mechanisms.

5.6.1 Foreground Traffic Impact

To evaluate Yield's impact on NewReno and CUBIC foreground traffic, this evaluation first examined its relative performance in the scenario where both foreground and background traffic was downloaded by a single receiver using short foreground transfers. Figure 5.3 plots the time required for a single foreground HTTP transfer using NewReno to complete against the throughput achieved by the background TCP transfer over a 60 second period. The bottleneck link for this experiment was 8 Mbps/1 Mbps with 50 ms of fixed-path delay.

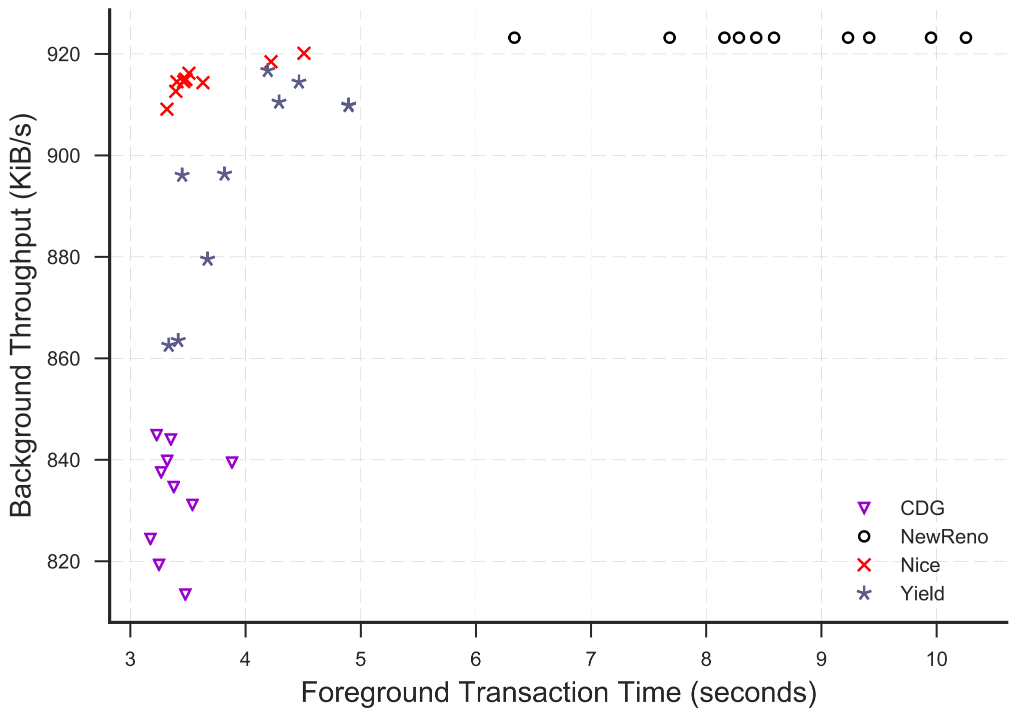


Figure 5.3.: Throughput and foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

As in the previous evaluation, described in Chapter 4, CDG and Nice achieve very low foreground transaction times (medians of 3.3 seconds and 3.5 seconds, respectively) with CDG also achieving relatively low throughput.

Yield appears to be slightly more aggressive than Nice, with a median foreground transaction time of 4 seconds. However, this result represents a reduction of 53% when compared to NewReno. Yield also achieved similar throughput to Nice (median throughput of 903 KiB/s compared with 915 KiB/s, respectively).

The results were similar when foreground traffic was split across four and eight TCP connections, the former of which is shown in Figure 5.4. However, Yield had a lower impact on foreground traffic — equivalent to that of CDG — in these experiments than in the single connection experiment.

Despite this decreased impact on foreground traffic, Yield achieved small improvements to throughput compared to CDG (8.2% and 5.4% for four and eight TCP connections, respectively).

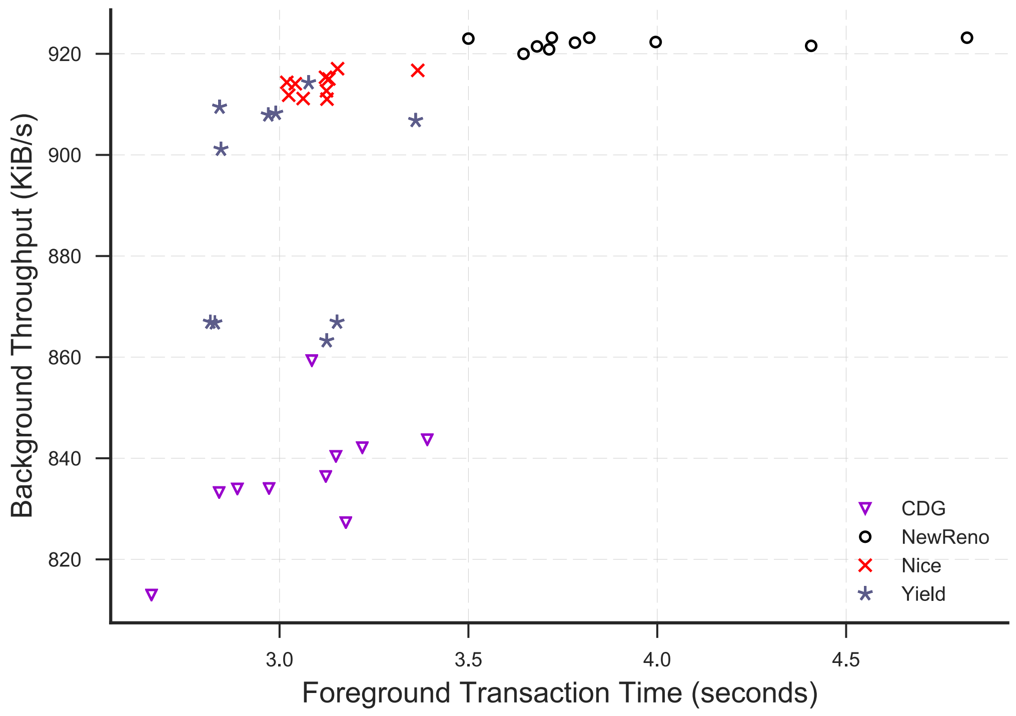


Figure 5.4.: Throughput and mean foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 50 ms fixed-path delay with four foreground TCP transfers using NewReno.

Increasing Path Delay

Next, the fixed-path delay was increased to examine the effect of higher delay on the performance of Yield. In higher delay settings, Yield performed similarly to Nice. As shown in Figure 5.5, Yield achieved a slightly lower median foreground transaction time than Nice (4.3 seconds, compared with 4.4 seconds). While this result was still worse than CDG (3.6 seconds), Yield continued to achieved greater throughput (17.8% improvement).

The impact of Yield on foreground traffic continued to improve as fixed-path delay was further increased (to 200 ms and 350 ms). As shown in Figure 5.6, performance trends for CDG and Nice remained consistent with findings of the previous evaluation (see Section 4.2.2), with CDG having minimal impact but very low throughput and Nice becoming regular TCP-like.

In high delay settings, Yield demonstrated higher foreground transaction times than CDG (5.2 seconds compared to 4.3 seconds at 200 ms, and 7.7 seconds compared to 5.2 seconds at 350 ms). However, the throughput penalty experienced by Yield

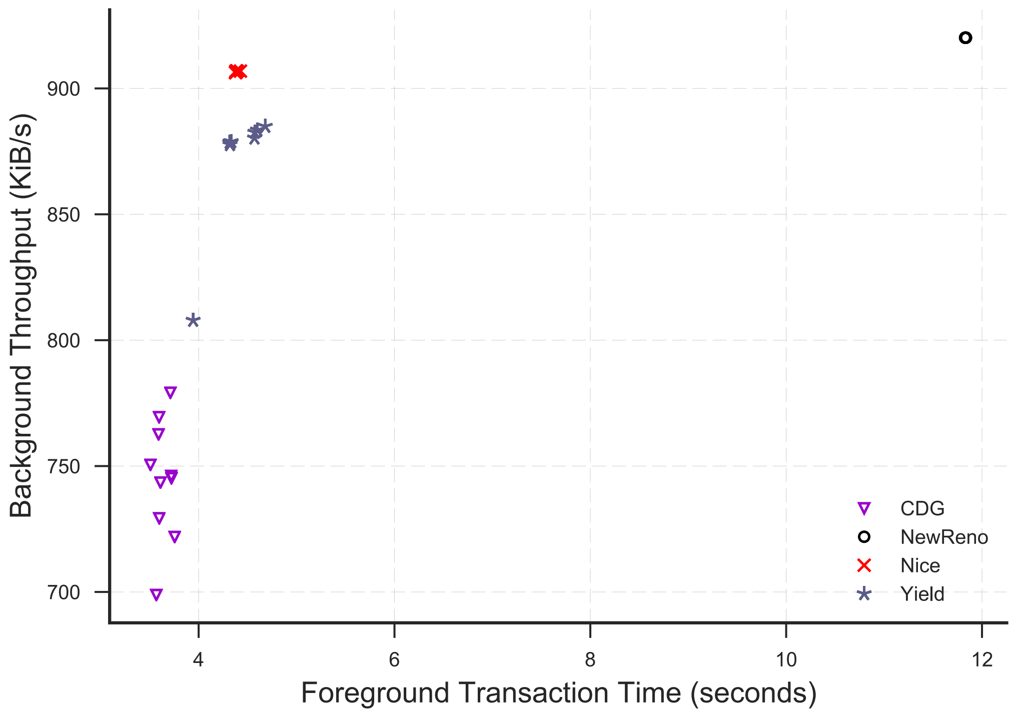


Figure 5.5.: Throughput and foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 100 ms fixed-path delay with a single foreground TCP transfer using NewReno.

when compared to NewReno was much smaller than the reduction experienced by CDG (4.4% compared to 48.7% at 200 ms, and 15.7% compared to 82.1% at 350 ms).

Increasing Bottleneck Link Speed

In addition to operating in high delay settings, the performance of Yield (relative to other LBE mechanisms) over high speed links was evaluated with a 50 Mbps bottleneck link.

As shown in Figure 5.7, and consistent with the results in Section 4.2.3, the differences between LBE mechanisms were diminished when bottleneck link speed was increased without increasing the of the foreground transfer.

At higher link speeds, Yield demonstrated similar impact on foreground traffic to CDG (median foreground transaction time of 1 second, compared with 0.9 seconds). Yield also experienced a smaller decrease in median throughput than CDG,

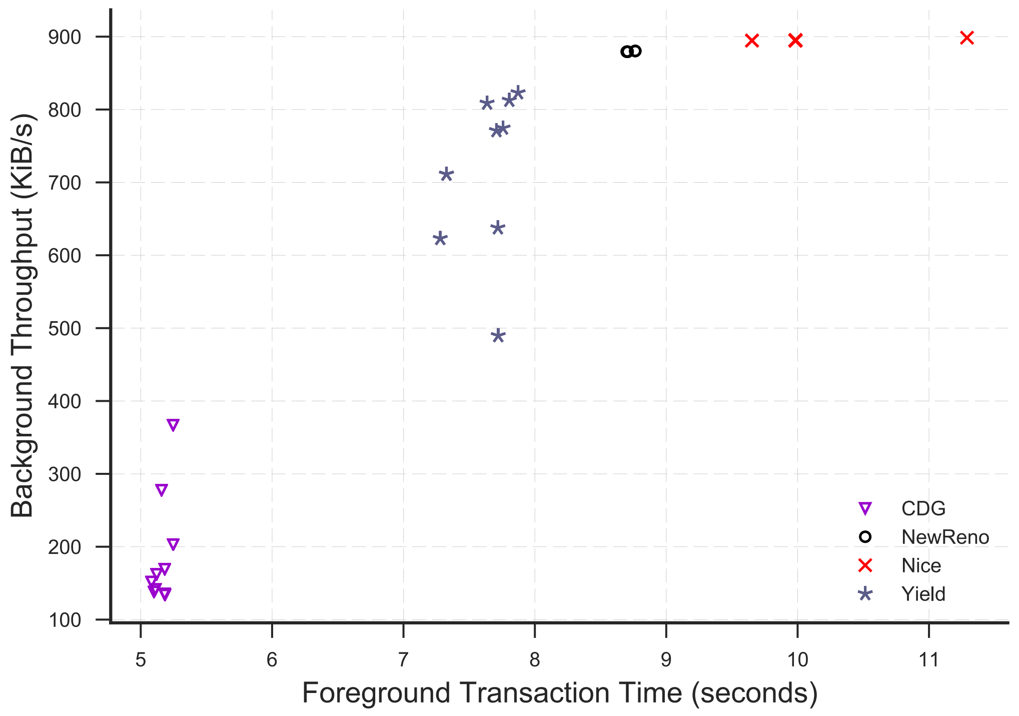


Figure 5.6.: Throughput and foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 350 ms fixed-path delay with a single foreground TCP transfer using NewReno.

with of 8.8% compared to NewReno (down from 18.7% for CDG). A more significant throughput reduction was observed for Yield in one test, which is discussed in Section 5.7.

The performance trends reported in the previous subsections remained similar when the number of foreground TCP connections and fixed-path delay were increased. As shown in Table 5.7, there were no differences in median foreground transaction time for the four mechanisms in the presence of 350 ms fixed-path delay. While Yield performed consistently with NewReno and Nice in this high delay setting, CDG demonstrated comparatively low throughput without any associated benefit in foreground transaction time.

CUBIC Foreground Traffic

Given that CUBIC has replaced NewReno as the default congestion control mechanism in Linux and recent versions of macOS, Yield's ability to co-exist with CUBIC foreground traffic was examined. To evaluate the impact of Yield on CUBIC-

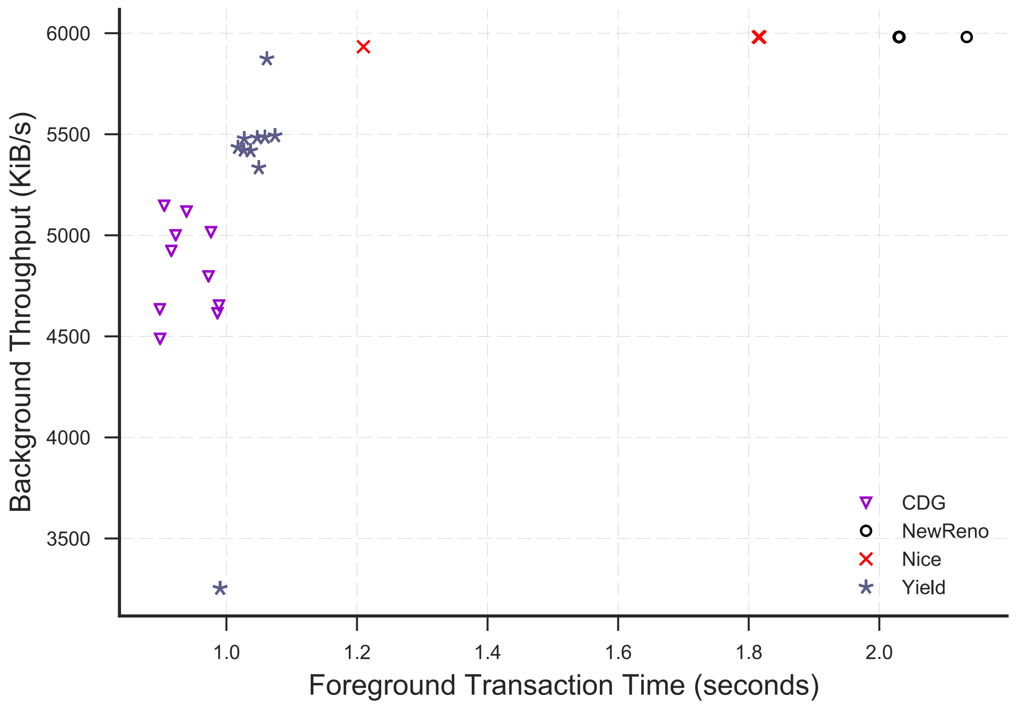


Figure 5.7.: Throughput and foreground transaction time for a download over 50 Mbps/20 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

managed foreground traffic, the experiments described in Section 5.5.3 were repeated using CUBIC for foreground transfers.

The LBE mechanisms, including Yield, performed consistently with previous experiments when competing against foreground traffic managed by CUBIC (as shown in Figure 5.8). Yield was the least friendly of the LBE mechanisms with median foreground transaction time of 5 seconds. However, it typically suffers no additional throughput penalty when compared to Nice. Relative to the other mechanisms, Yield's performance is unchanged when fixed-path delay is increased to 100 ms.

Consistent with experiments using NewReno foreground traffic, Yield had a lower impact on foreground traffic than Nice at higher delay settings (Figure 5.9 shows the results for 350 ms). This improvement was greater than when Yield competed against NewReno, with reductions in median foreground transaction times of 50.7% and 51% for 200 ms and 350 ms, respectively.

The relative reductions in foreground transaction time were smaller when compared to Nice, which conceded bandwidth more readily when competing against

	FgConn	1		4		8	
Delay	Mechanisms	FTT	BgTput	FTT	BgTput	FTT	BgTput
50 ms	NewReno	2.0	5981	1.8	5981	1.5	5981
	Nice	1.8	5981	2.1	5977	1.4	5980
	CDG	0.9	4860	0.8	4885	0.8	4918
	Yield	1.0	5456	0.9	5656	0.8	5832
100 ms	NewReno	2.0	5947	1.5	5947	1.4	5947
	Nice	2.0	5905	1.6	5905	1.4	4905
	CDG	1.3	3019	1.0	3005	0.9	2686
	Yield	2.0	5946	1.5	5946	1.4	5946
200 ms	NewReno	2.3	4591	1.9	4597	1.7	4595
	Nice	2.3	4597	1.9	4597	1.7	4597
	CDG	2.1	988	1.7	877	1.5	1449
	Yield	2.3	4598	1.9	4597	1.7	4441
350 ms	NewReno	3.6	2606	2.9	2606	2.5	2605
	Nice	3.6	2608	2.9	2607	2.5	2608
	CDG	3.6	883	2.9	306	2.5	1627
	Yield	3.6	2615	2.9	2614	2.5	2613

Table 5.7.: Throughput (in KiB/s) and foreground transaction time (in seconds) for a download over a 50 Mbps/20 Mbps bottleneck link with one, four, and eight foreground TCP transfers using NewReno.

CUBIC. In experiments with 200 ms and 350 ms of path delay, Yield provided 4.6% and 10.1% improvements in foreground transaction time over Nice, respectively.

The LBE mechanisms provided smaller improvements compared to CUBIC when foreground traffic was divided amongst four and eight TCP connections. However, the performance of Yield — as well as the other LBE mechanisms — remained consistent with experiments using a single foreground transfer.

Yield also performed well when competing against CUBIC at higher speeds, as shown in Figure 5.10. Yield was noticeably less aggressive than Nice, with a median foreground transaction time of 1.3 seconds (compared to 1.9 seconds for Nice) but slightly more aggressive than CDG (1.1 seconds). However, Yield again demonstrated substantially higher throughput compared to CDG (reductions relative to CUBIC of 3.3% compared to 18% for CDG).

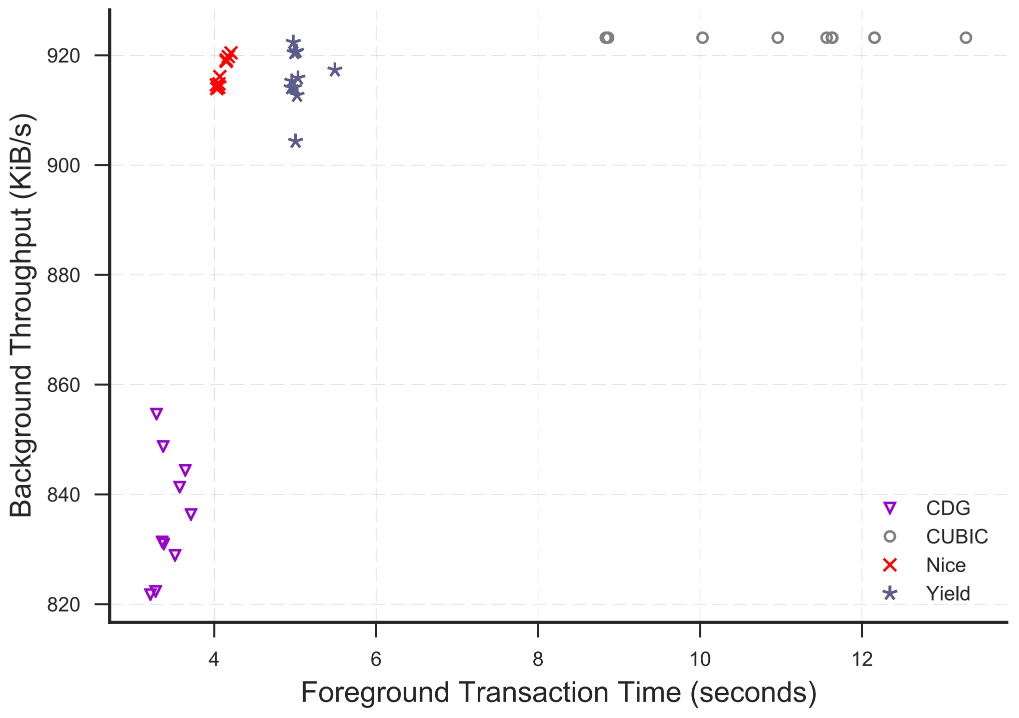


Figure 5.8.: Throughput and foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using CUBIC.

LBE Upload

Experiments were also carried out examining the impact of an LBE upload on foreground HTTP downloads. In these experiments, all LBE mechanisms achieved similar throughput when a background upload competed against a foreground download over an 8 Mbps bottleneck link. However, Figure 5.11 shows that Yield had greater impact on foreground traffic compared to Nice and CDG.

In this scenario, Yield achieved a median foreground transaction time of 6.8 seconds, compared with 3.1 seconds and 3.3 seconds for CDG and Nice, respectively. However, this result still represents a 25.3% improvement compared to NewReno.

This trend remained consistent when the number of foreground TCP connections was increased. However, Yield no longer demonstrated any improvement over — but still performed equivalently to — NewReno as fixed-path delay was increased.

Performance when the LBE mechanisms competed against CUBIC remained relatively consistent, although Yield achieved a mean reduction in foreground transaction time of 43.1% and 26.4% with 100 ms and 200 ms fixed-path delay.

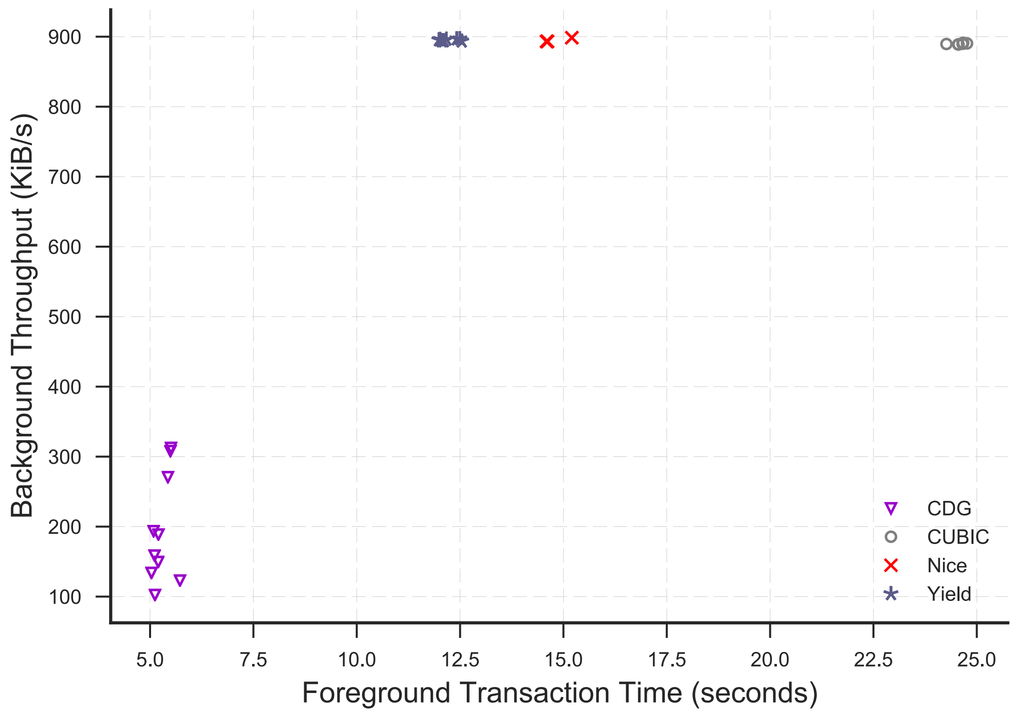


Figure 5.9.: Throughput and foreground transaction time for a download over 8 Mbps/1 Mbps bottleneck link subject to 350 ms fixed-path delay with a single foreground TCP transfer using CUBIC.

The differences between LBE mechanisms were diminished but broadly consistent when competing against NewReno over a 50 Mbps bottleneck link, as shown in Figure 5.12. Yield achieved a mean reduction in foreground transaction time of 23.2% with 50 ms of fixed-path delay but demonstrated no clear decrease in throughput as delay was increased. Performance for Yield was also consistent when competing against additional foreground transfers.

Wireless Networks

Consistent with the findings of the evaluation of existing LBE congestion control mechanisms (described in Section 4.2.6), the performance of Yield over a 802.11n wireless network was consistent with results over a wired network. Figure 5.13 shows the CV (ratio between standard deviation and mean) for the foreground transaction time over wired and wireless links calculated on a per-mechanism basis for each experiment. While the results indicate some additional variability in experiments over wireless networks ($\overline{CV} = 0.07$, compared to $\overline{CV} = 0.05$ over Ethernet), the differences are relatively small.

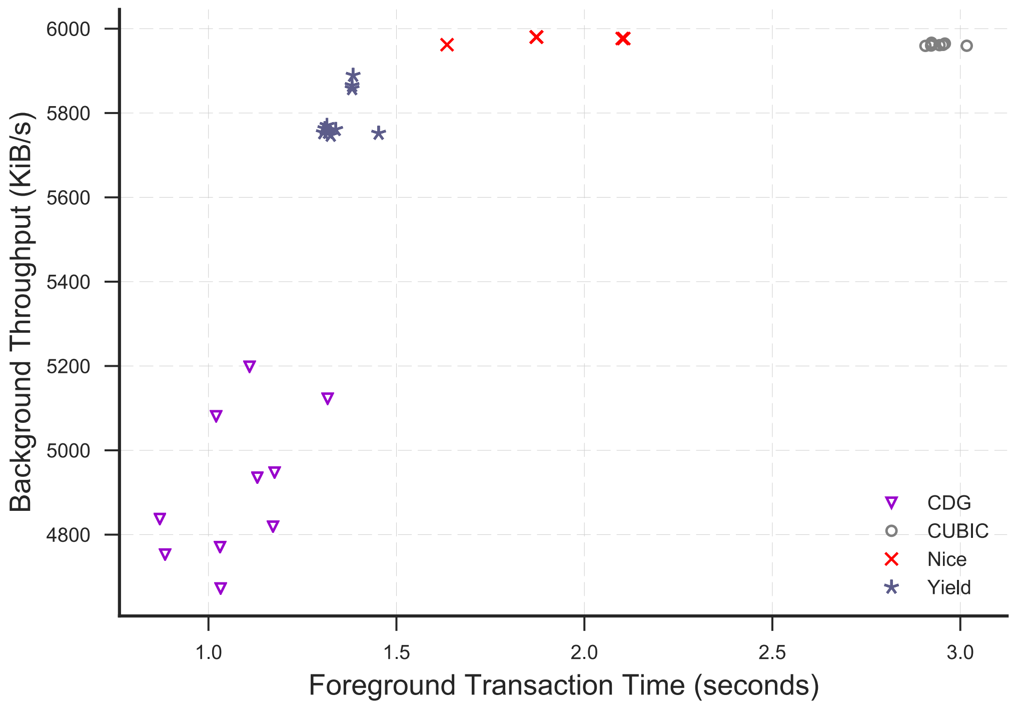


Figure 5.10.: Throughput and foreground transaction time for a download over 50 Mbps/20 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using CUBIC.

A small increase in variability was also observed in throughput readings, as shown in Figure 5.14, where mean CV for wireless networks was found to be 0.08 (compared with $\overline{CV} = 0.03$ over Ethernet). However, the minimal impact of wireless links could primarily be attributed to the ideal channel conditions in the testbed network.

5.6.2 Foreground Throughput

To evaluate Yield's impact on foreground traffic when competing against larger foreground downloads, an additional set of experiments in which the LBE mechanisms competed against a long-lived foreground TCP transfer was carried out.

In these experiments, long-lived foreground and background traffic competed for bandwidth over the same bottleneck link. These experiments were also carried out with the direction of the background transfer inverted. The impact of the LBE mechanisms on foreground traffic was measured based on the median throughput achieved over the course of a run.

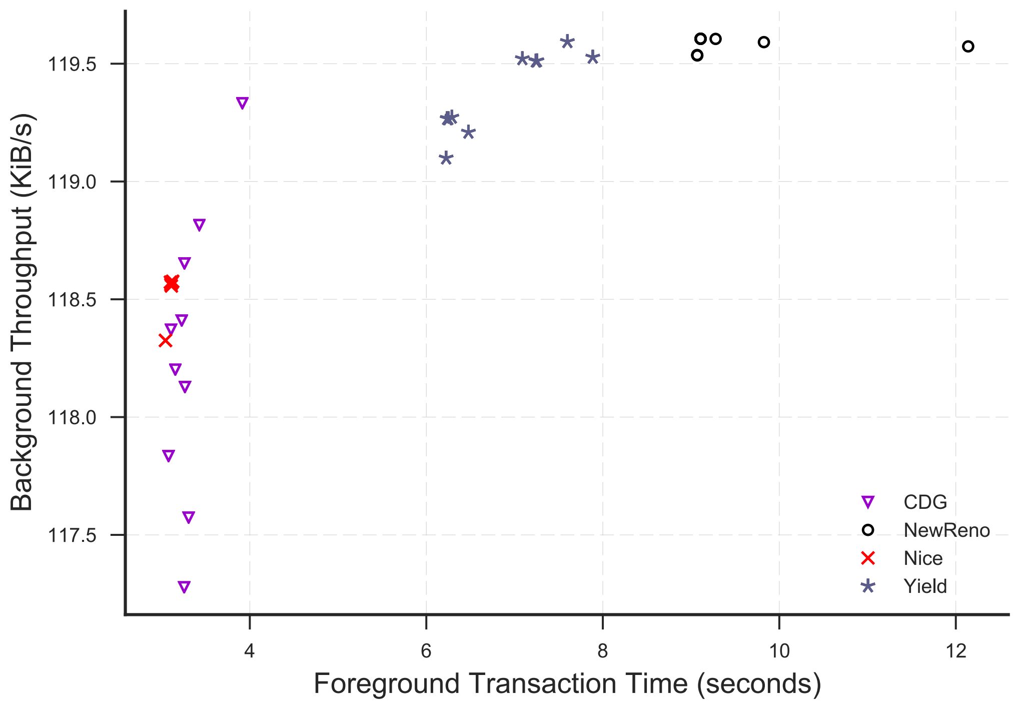


Figure 5.11.: Throughput and foreground transaction time for an upload over a 8 Mbps/1 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

NewReno Foreground Traffic

The performance of Yield when competing against long-lived foreground traffic was consistent with experiments with short foreground transfers. As shown in Figure 5.15, which plots the throughput of the NewReno foreground transfers when competing against a single LBE download over an 8 Mbps bottleneck link, Yield is the least friendly of the LBE mechanisms with median foreground throughput of 640 KiB/s (compared to 767 KiB/s and 735 KiB/s for CDG and Nice, respectively). However, differences between Yield and the other LBE mechanisms became minimal when competing with additional foreground transfers. Median throughput for Yield was 204 KiB/s for each transfer when in competition with four concurrent NewReno transfers (compared to 209 KiB/s for CDG and 203 KiB/s for Nice).

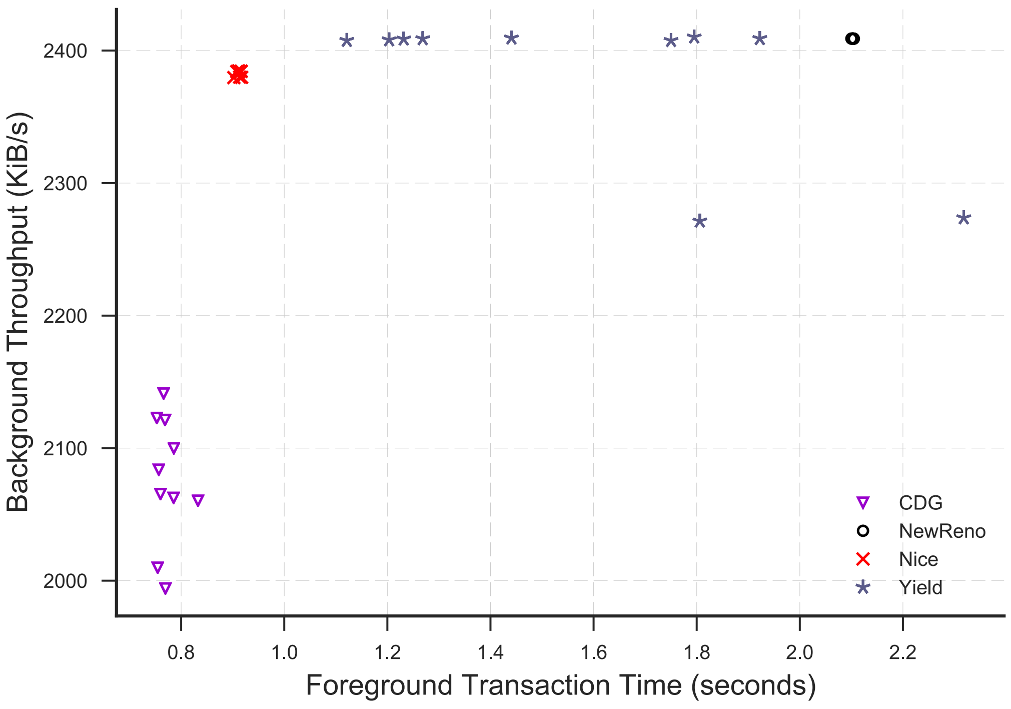
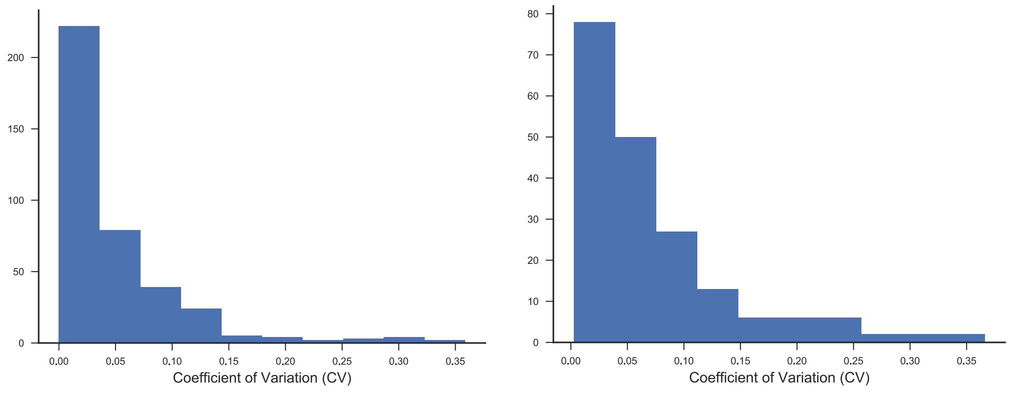


Figure 5.12.: Throughput and foreground transaction time for an upload over a 50 Mbps/20 Mbps bottleneck link subject to 50 ms fixed-path delay with a single foreground TCP transfer using NewReno.

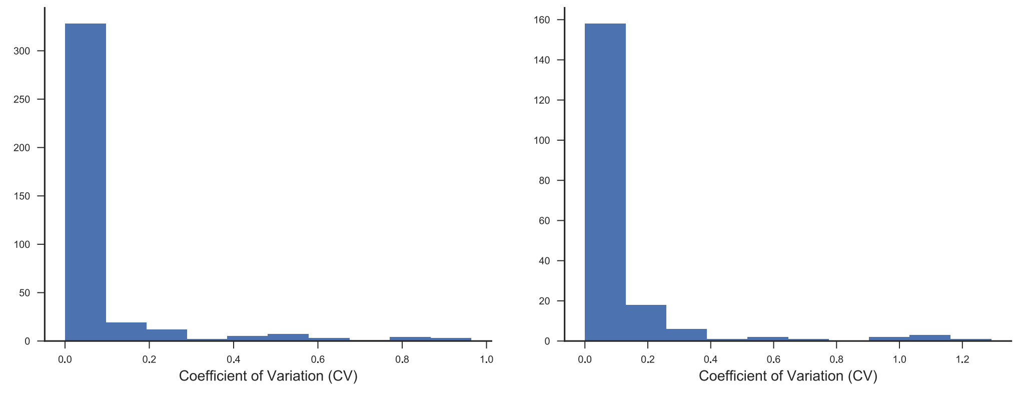


(a) Ethernet network. **(b)** 802.11n wireless network.

Figure 5.13.: Distribution of CV values for foreground transaction time.

Increasing Path Delay

This consistency with experiments utilising short foreground transfers was also present at higher delay settings, as shown for 350 ms fixed-path delay in Figure 5.16. When competing with a single foreground transfer, Yield allows the competing



(a) Ethernet network. **(b)** 802.11n wireless network.

Figure 5.14.: Distribution of CV values for background throughput.

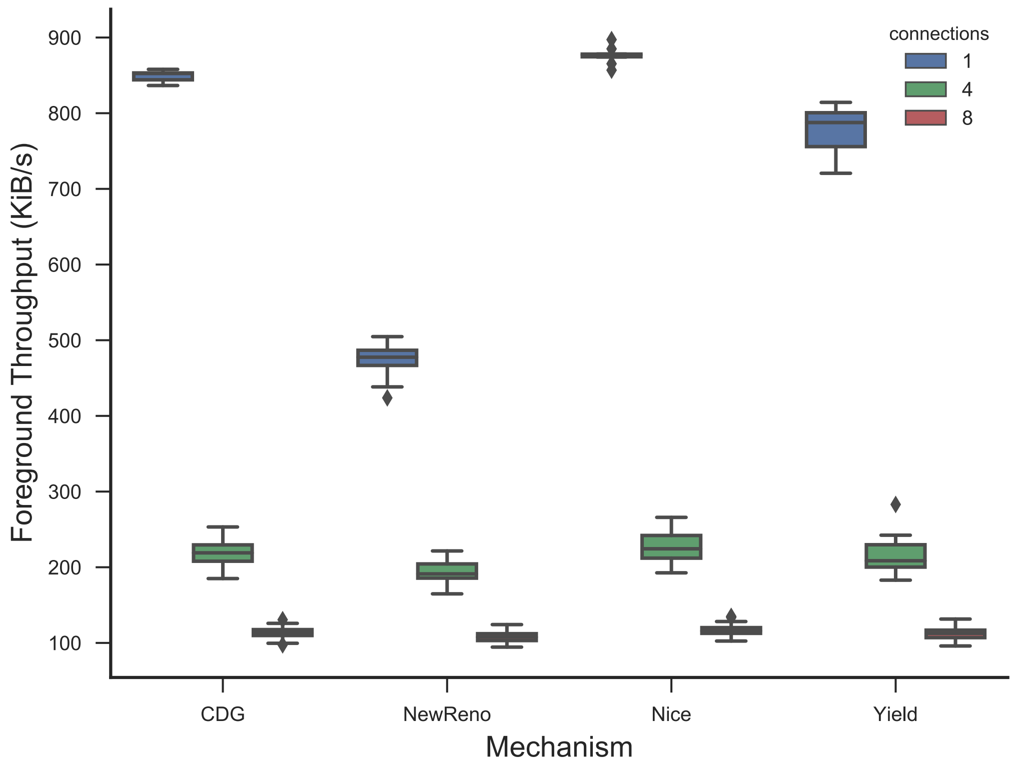


Figure 5.15.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE download over an 8 Mbps/1 Mbps bottleneck link with 50 ms fixed-path delay.

NewReno connection to utilise a greater portion of available throughput than Nice (median foreground throughput of 332 KiB/s, compared to 256 KiB/s). However, the differences were again diminished as additional foreground transfers were introduced.

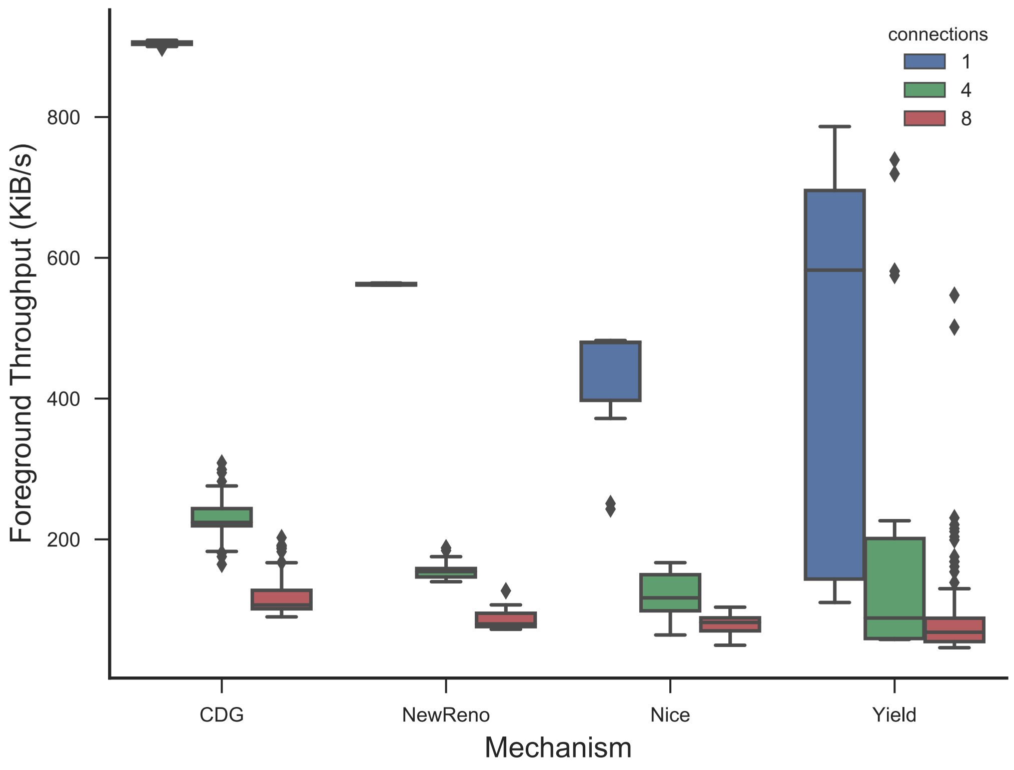


Figure 5.16.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE download over an 8 Mbps/1 Mbps bottleneck link subject to 350 ms fixed-path delay.

Increasing Bottleneck Link Speed

Experiments were also carried out where Yield competed against long-lived foreground transfers over a 50 Mbps bottleneck link. Results for these experiments are shown in Figure 5.17. In these experiments, throughput improvements for NewReno foreground traffic competing with Yield were generally consistent with experiments over the slower bottleneck link. These improvements were consistently smaller than those achieved by CDG, but notably resulted in Yield no longer being the most aggressive of the LBE mechanisms tested as Nice became less friendly.

Yield became more aggressive as path delay was increased to 100 ms and above. With path delay increased, Yield was still able to provide mean improvements in foreground delay of 18.7% and 6.1% for 100 ms and 350 ms, respectively. However, Yield demonstrated limited improvement in experiments with 200 ms path delay with only a 6.2% improvement when competing against four concurrent foreground transfers and a decrease in foreground throughput of 12.7% with eight concurrent foreground transfers.

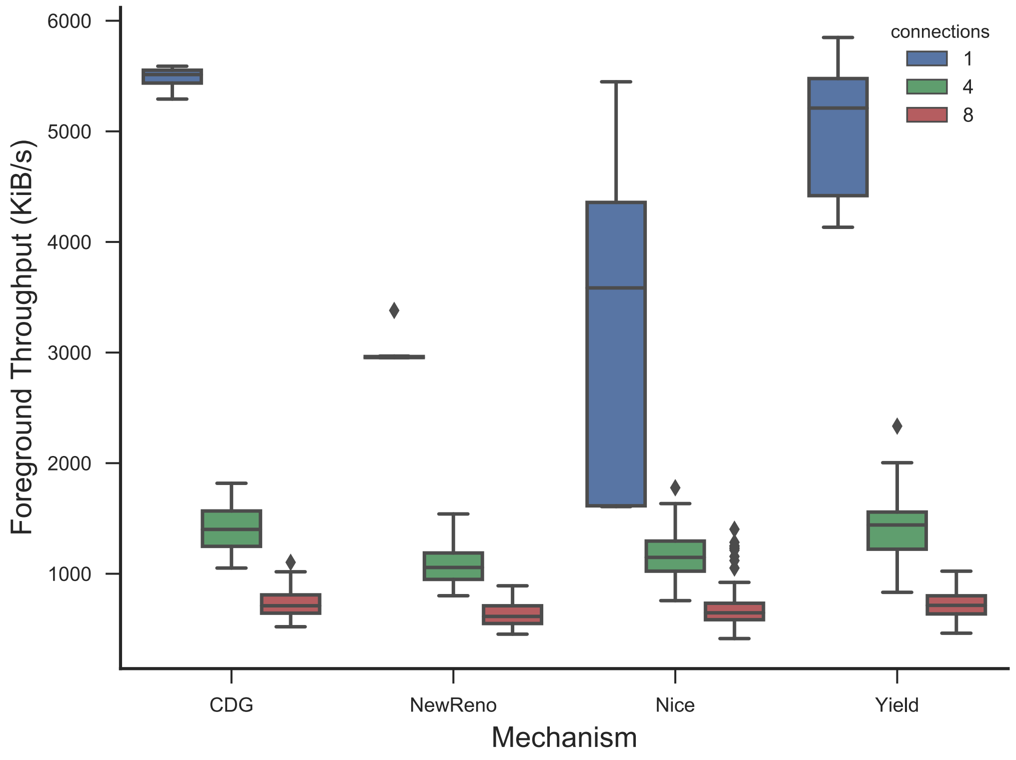


Figure 5.17.: Throughput for one, four, and eight foreground transfers using NewReno competing against a single LBE download over an 50 Mbps/20 Mbps bottleneck link with 50 ms fixed-path delay.

CUBIC Foreground Traffic

Yield's performance when competing against CUBIC foreground traffic was also consistent with experiments using short foreground transfers, as well as those where NewReno was used. As shown in Figure 5.18, Yield was also the least friendly of the LBE mechanisms while competing against a single foreground transfer. However, Yield was still far less aggressive than CUBIC and increased foreground throughput by 105% compared to when CUBIC was used for background traffic. As with experiments using NewReno foreground traffic, differences between Yield and the other mechanisms were minimal with four and eight concurrent foreground transfers.

When competing against CUBIC foreground traffic in high delay settings, Yield performed similarly to experiments where NewReno foreground traffic was used. Foreground throughput for Yield increased by 241.2% compared to CUBIC. Nice and CDG achieved increases of 171.6% and 375.8% compared to CUBIC, respectively.

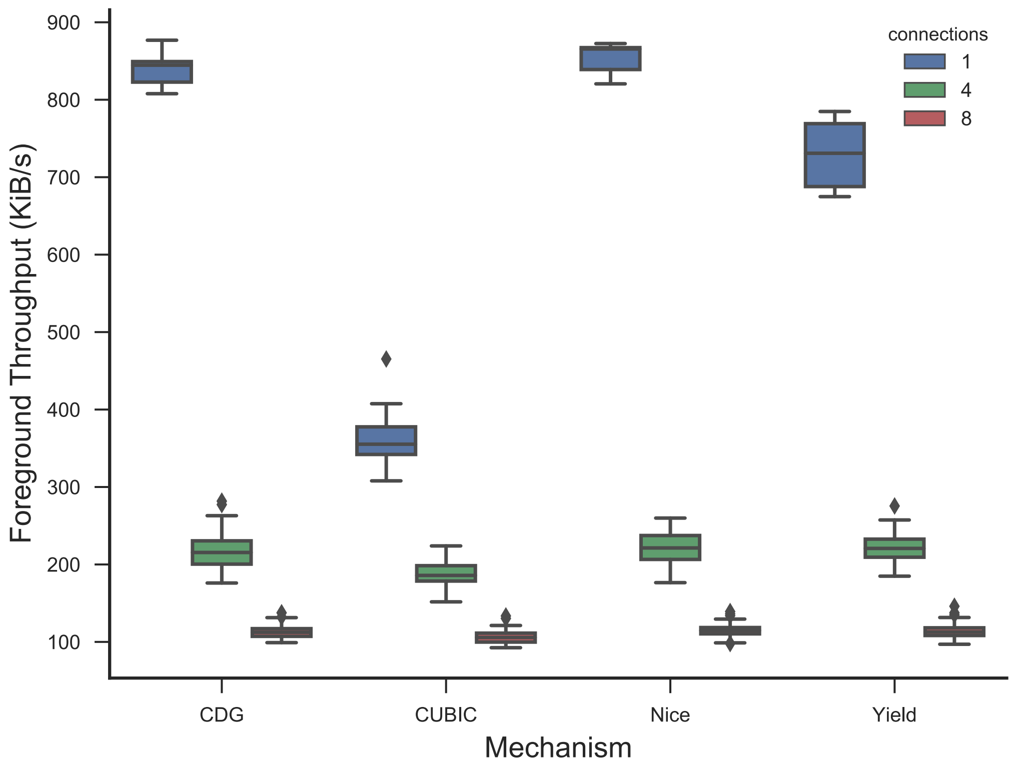


Figure 5.18.: Throughput for one, four, and eight foreground transfers using CUBIC competing against a single LBE download over an 8 Mbps/1 Mbps bottleneck link with 50 ms fixed-path delay.

LBE Upload

The performance of LBE uploads when competing against long-lived foreground transfers was also consistent with that when competing against short foreground transfers. When competing against a single foreground download, Yield demonstrated a 11% improvement in foreground throughput compared to when NewReno was used for the upload. However, this improvement was substantially smaller than those observed for CDG and Nice which demonstrated a twofold increase in foreground throughput.

Yield's performance was similarly consistent when competing against CUBIC foreground traffic, with no appreciable change in foreground throughput compared to when CUBIC was used for background traffic. By contrast, CDG and Nice achieved much larger improvements of 61% and 76%, respectively.

As with previous experiments, the differences between the mechanisms were less evident when additional foreground transfers were introduced. Similar performance trends were also observed in settings with high bottleneck link speeds.

5.6.3 Queuing Delay

The effect of Yield on the delay experienced by foreground and background traffic was also investigated. In particular, any increases in the delay experienced by foreground traffic would likely have an effect on the usability of interactive applications such as streaming video.

For this study, delay was calculated by Wireshark based on the time difference between the transmission of the original packet and the time at which the acknowledgement was received.

Due to the brevity of foreground transfers in the previous scenarios, the impact of LBE mechanisms on delay was only examined in the scenarios where longer foreground transfers were initiated, as well as in the fairness scenario described in Section 3.3.

NewReno Foreground Traffic

Table 5.8 lists the mean and standard deviation of queuing delay for experiments where a single LBE download competed against one, four, and eight concurrent foreground TCP transfers using NewReno. As with Nice and CDG, Yield demonstrated no discernible change to queuing delay when operating in low to medium path delay settings.

With very high fixed-path delay, Yield was observed to provide mean reductions of 13.1% to queuing delay compared to NewReno. This reduction represents the largest of the LBE mechanisms, with Nice achieving a mean reduction of 6.9% and CDG increasing delay by 5% in these experiments.

Yield performed similarly to NewReno in experiments where an LBE upload competed against one or more foreground downloads. This was in contrast to mean reductions of 34.8% and 28.9% for Nice and CDG, respectively.

Exceptions to this performance were observed when Yield competed against a single foreground download in low delay settings ($\leq$ 100 ms), where a mean reduction of 22% to mean queuing delay was observed. Yield reduced queuing delay by 28%

FgConn		1		4		8	
Delay	Mechanism	Mean	Stdev	Mean	Stdev	Mean	Stdev
50 ms	NewReno	125	24.1	135	23.0	143	27.1
	Nice	126	23.6	135	23.2	142	27.2
	CDG	126	23.0	137	24.6	144	27.4
	Yield	126	22.1	135	22.8	144	26.9
100 ms	NewReno	249	49.8	257	44.2	266	45.6
	Nice	249	52.7	262	48.4	268	47.9
	CDG	239	56.8	262	53.7	271	52.2
	Yield	249	49.0	257	48.5	267	43.1
200 ms	NewReno	507	139.8	513	120.6	514	114.8
	Nice	483	111.9	510	119.4	514	115.8
	CDG	462	157.6	509	144.8	526	135.1
	Yield	507	103.1	520	126.2	522	123.0
350 ms	NewReno	900	267.7	916	269.7	905	233.9
	Nice	796	275.6	875	242.2	864	220.3
	CDG	909	245.0	857	343.0	890	322.4
	Yield	833	179.2	771	234.3	761	203.6

Table 5.8.: Median of mean and standard deviation for RTTs (in ms) for a LBE download competing against one, four, and eight concurrent foreground TCP transfers using NewReno for a 8 Mbps/1 Mbps bottleneck link.

while competing against four concurrent downloads with 350 ms path delay, but an increase of 25% was observed when the number of simultaneous downloads was further increased to eight compared to NewReno.

CUBIC Foreground Traffic

When competing against CUBIC foreground traffic, Yield more consistently reduced queuing delay than when competing with NewReno in all delay settings. This result is reflected in Table 5.9, which lists the mean and standard deviation of delay for experiments where a single LBE download competed against one, four, and eight concurrent foreground TCP transfers using CUBIC. In these experiments, Yield reduced mean delay by 4.4% compared to CUBIC. This was in contrast to mean reductions of 2.9% and 5.2% for Nice and CDG.

Yield also achieved small decreases in delay when used for upload traffic with low fixed-path delay, with a mean decrease of 3.5%. However, an increase in queuing delay was observed when Yield shared the bottleneck link with four foreground

FgConn		1		4		8	
Delay	Mechanism	Mean	Stdev	Mean	Stdev	Mean	Stdev
50 ms	CUBIC	143	13.9	147	16.84	152	22.6
	Nice	136	13.0	145	17.36	151	20.5
	CDG	135	16.2	145	17.12	152	21.2
	Yield	137	12.6	146	16.08	151	20.4
100 ms	CUBIC	270	26.1	282	33.49	290	40.9
	Nice	263	28.8	273	32.13	286	39.8
	CDG	253	38.6	274	30.87	284	37.8
	Yield	247	38.1	275	38.13	284	36.7
200 ms	CUBIC	538	77.1	553	81.93	563	98.9
	Nice	510	96.6	533	82.54	560	91.1
	CDG	489	89.3	515	97.60	547	89.6
	Yield	508	91.2	521	95.64	548	81.2
350 ms	CUBIC	948	181.1	968	212.00	987	230.6
	Nice	897	210.0	927	203.84	964	227.4
	CDG	836	228.8	896	247.92	921	263.5
	Yield	881	208.8	893	124.22	927	225.4

Table 5.9.: Median of mean and standard deviation for RTTs (in ms) for a LBE download competing against one, four, and eight concurrent foreground TCP transfers using CUBIC for a 8 Mbps/1 Mbps bottleneck link.

downloads in the presence of 100 ms path delay. By contrast, Nice and CDG demonstrated larger decreases in queuing delay (26.3% and 26.6%, respectively). Yield demonstrated larger decreases in queuing delay when path delay was increased to 200 ms and 350 ms (mean decrease of 27%), although these improvements were still not as substantial as those for Nice or CDG.

Self-Induced Delay

As with experiments where LBE mechanisms competed against NewReno and CUBIC foreground traffic, Yield consistently demonstrated reductions to self-induced latency over NewReno. Table 5.10 lists mean and standard deviation of latency experienced by competing LBE transfers for each of the mechanisms evaluated. Yield reduced mean delay by between 4.4% and 33.2% (with the exception of experiments with two competing Yield transfers and 350 ms fixed-path delay where latency increased by 18%).

	BgConn	2		4		8	
Delay	Mechanism	Mean	Stdev	Mean	Stdev	Mean	Stdev
50 ms	NewReno	124	26.2	133	25.4	142	28.3
	Nice	64	13.3	87	18.2	128	19.8
	CDG	61	9.0	85	15.4	125	22.6
	Yield	83	20.5	99	20.9	144	20.5
100 ms	NewReno	242	58.1	253	55.8	265	53.4
	Nice	144	44.3	144	45.4	193	42.9
	CDG	105	7.2	110	11.0	131	19.4
	Yield	167	52.0	188	52.4	217	48.1
200 ms	NewReno	468	143.9	496	138.8	516	129.3
	Nice	425	75.0	279	71.6	341	96.2
	CDG	204	7.1	207	12.2	211	14.8
	Yield	447	124.7	420	129.6	429	132.2
350 ms	NewReno	725	287.7	896	288.5	872	280.1
	Nice	523	108.7	573	105.1	516	87.2
	CDG	352	5.0	354	8.9	360	18.8
	Yield	856	289.7	705	306.9	797	287.6

Table 5.10.: Median of mean and standard deviation for RTTs (in ms) for two, four, and eight concurrent LBE transfers for a 8 Mbps/1 Mbps bottleneck link.

5.6.4 Intra-Protocol Fairness

Finally the ability of Yield to fairly share available bandwidth amongst multiple concurrent flows was examined. Figure 5.19 plots the Jain's fairness indices for LBE transfers grouped by the number of concurrent LBE transfers for experiments where the mechanisms were subjected to 50 ms of fixed-path delay. While not exceeding the fairness of CDG, Yield demonstrated greater fairness between flows than Nice. This difference was particularly noticeable when eight concurrent LBE transfers were active, where Yield achieved a median fairness index of 0.97 (compared with 0.82 for Nice). Like CDG, Yield also demonstrated consistent fairness regardless of the number of concurrent LBE transfers.

Like Nice, Yield achieved lower fairness indices at higher bottleneck link speeds, as shown in Figure 5.20. However, this reduced fairness remained substantially higher than that achieved by Nice.

Lastly, the impact of fixed-path delay on intra-protocol fairness was examined. Consistent with the observations made in Section 4.5, the results shown in Figure 5.21

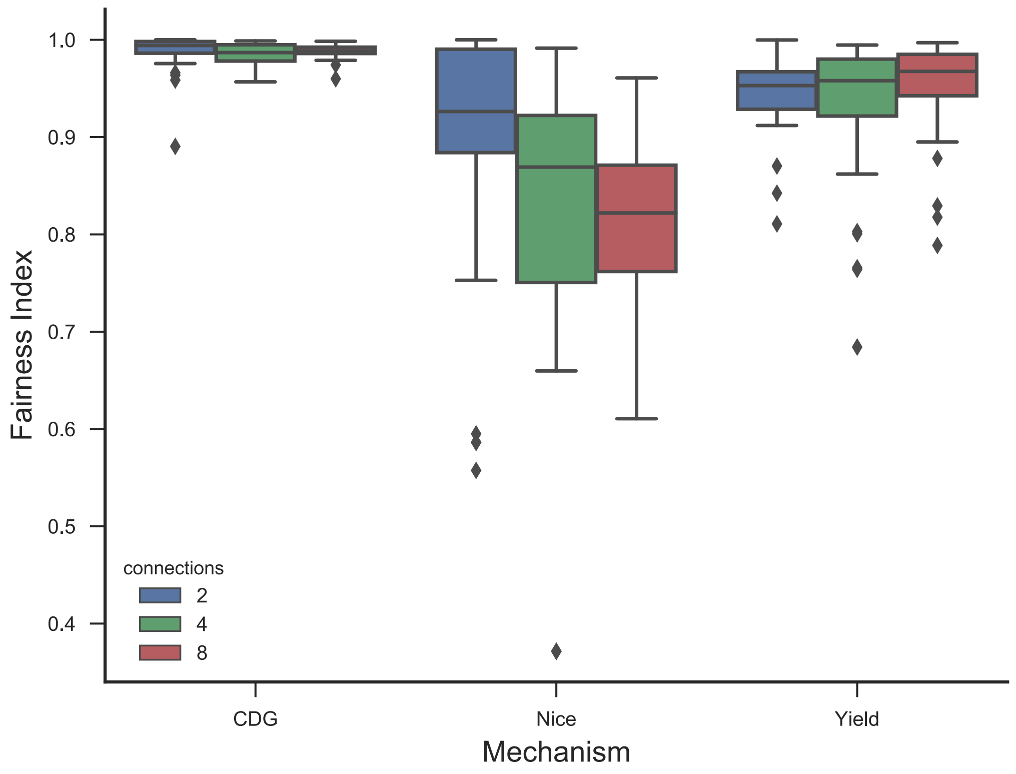


Figure 5.19.: Jain's Fairness indices for LBE transfers subjected to 50 ms fixed-path delay grouped by number of concurrent connections.

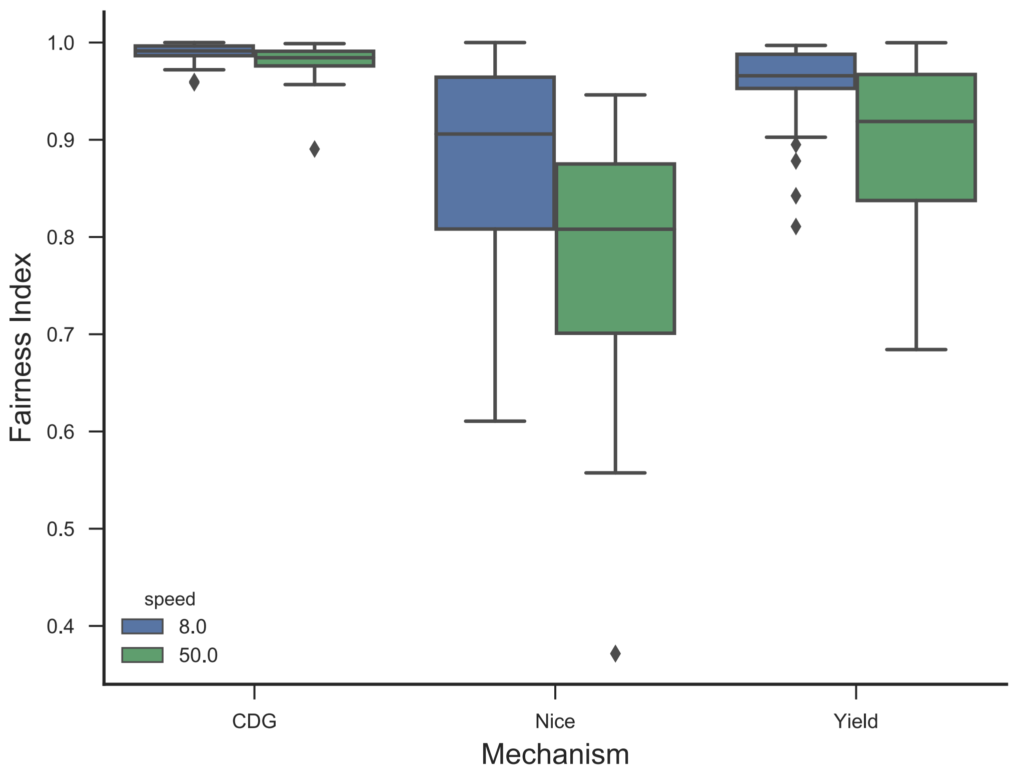


Figure 5.20.: Jain's fairness indices for LBE transfers subjected to 50 ms fixed-path delay grouped by bottleneck link speed (in Mbps).

indicate that intra-protocol fairness for CDG and Nice was reduced as additional delay increased. Yield remained relatively consistent, with median fairness indices increasing slightly at 200 ms and 350 ms (fairness indices of 0.96, 0.92, 0.94, and 0.95 for each of the fixed-path delay settings considered, respectively).

5.7 Discussion

Yield was primarily designed to improve the performance of LBE congestion control mechanisms in high fixed-path delay settings, without significant performance regression in low-delay settings. Improvements to fairness were also considered desirable. The evaluation of Yield indicates that it was largely successful in meeting these design goals.

In low delay settings, Yield typically had a lower impact on NewReno and CUBIC foreground TCP transfers compared to that of Nice when foreground and background traffic traversed the same direction over the bottleneck link. In this sce-

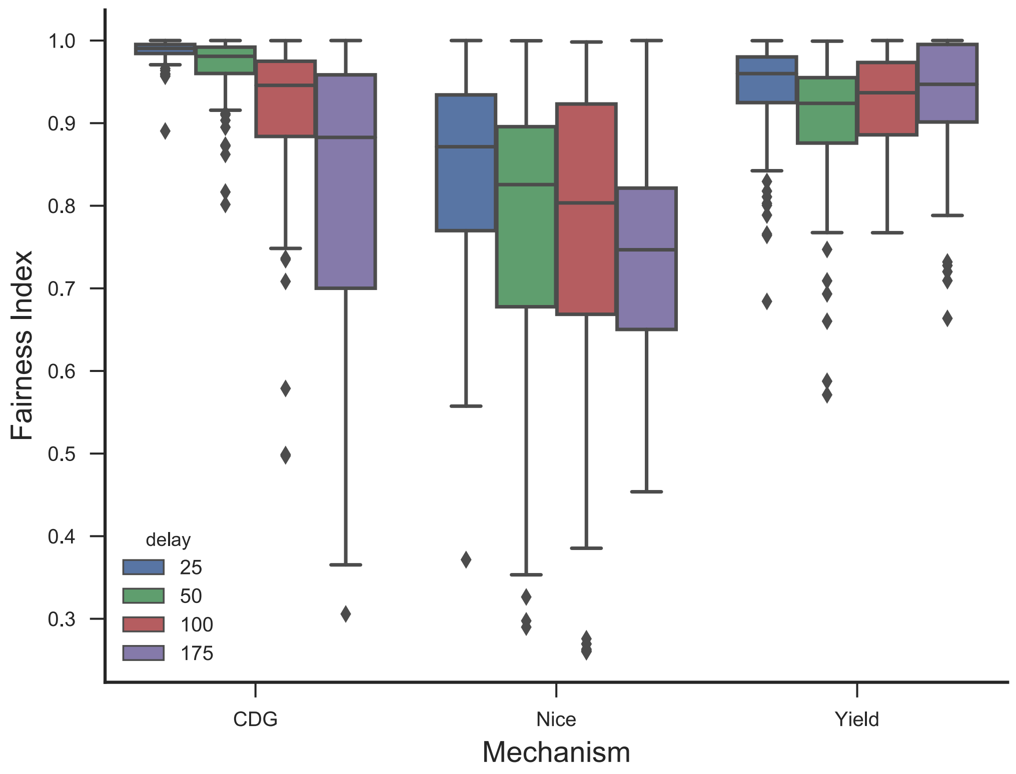


Figure 5.21.: Jain's fairness indices for LBE transfers grouped by fixed-path delay values (in ms).

nario, Yield achieved mean improvements in foreground transaction time of 39.2% and 46.4% relative to NewReno and CUBIC, respectively (compared to 26.7% and 35.9% for Nice). While these reductions were not as large as those demonstrated by CDG, Yield was able to achieve similar improvements in medium delay settings when used for downloads. These improvements to foreground transaction time were achieved while providing throughput similar to Nice.

The results of testing Yield with high fixed-path delay also indicate moderate improvements when competing against NewReno and CUBIC foreground traffic, with mean reductions in foreground transaction times of 13.2% and 32.2%. These improvements were substantially larger than those achieved by Nice, which typically increased foreground transaction time in high delay settings. While Yield achieved lower throughput than Nice in high delay settings, these reductions were much smaller than the penalties experienced by CDG.

The results suggest that Yield was successful in achieving the primary design goal of improving performance in high delay settings, without compromising performance with low path delay. Based on these findings, Yield belongs in the low-impact – high

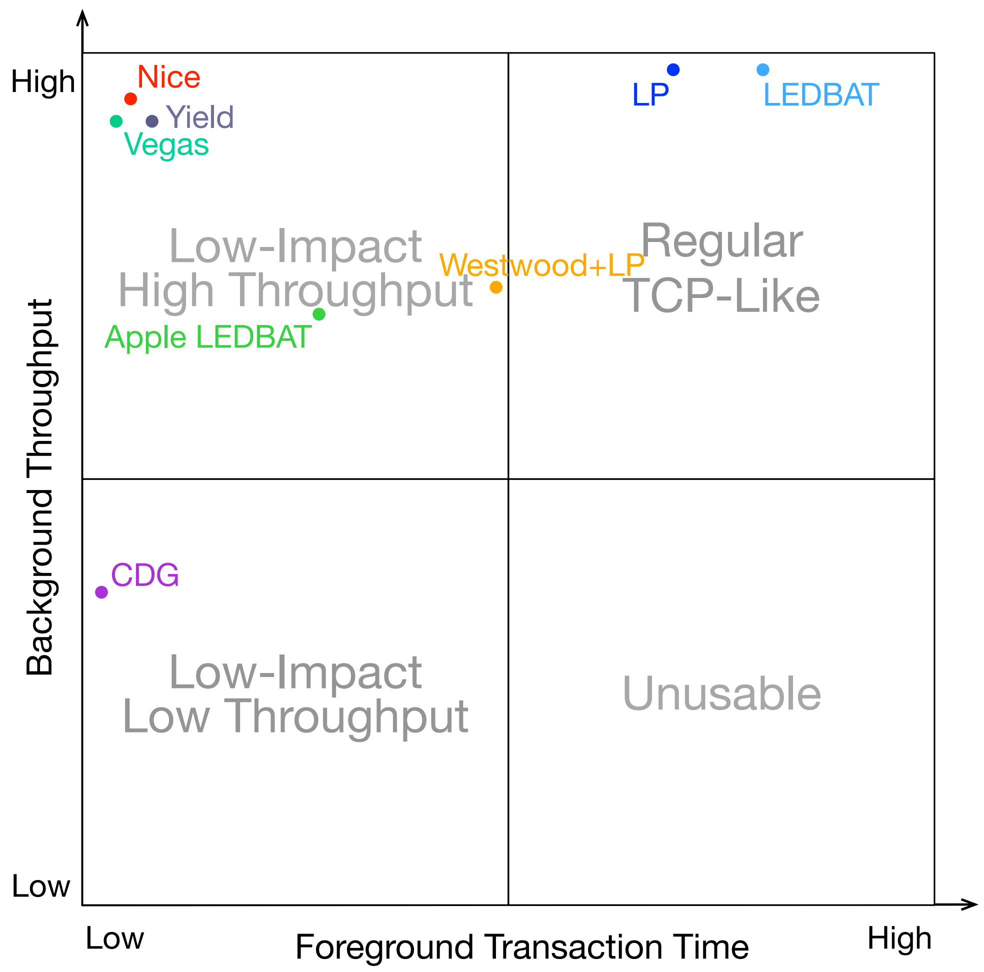


Figure 5.22.: Approximate categorisation of LBE congestion control mechanisms at 50 ms fixed-path delay.

throughput category of LBE congestion control mechanisms in low delay settings (identified in Chapter 4). Figure 5.22 shows an updated matrix of categorisations, including all mechanisms considered in the initial evaluation of LBE congestion control mechanisms described in Chapter 4.

Yield moves to the border of the low-impact – high throughput category and the regular TCP-like mechanisms when subjected to high fixed-path delay, as shown in Figure 5.23, which depicts the approximate categorisations of the LBE congestion control mechanisms in high delay settings.

While not exceeding the fairness of CDG, Yield was also successful in the secondary goal: to improve intra-protocol fairness over Nice. This improvement was evident

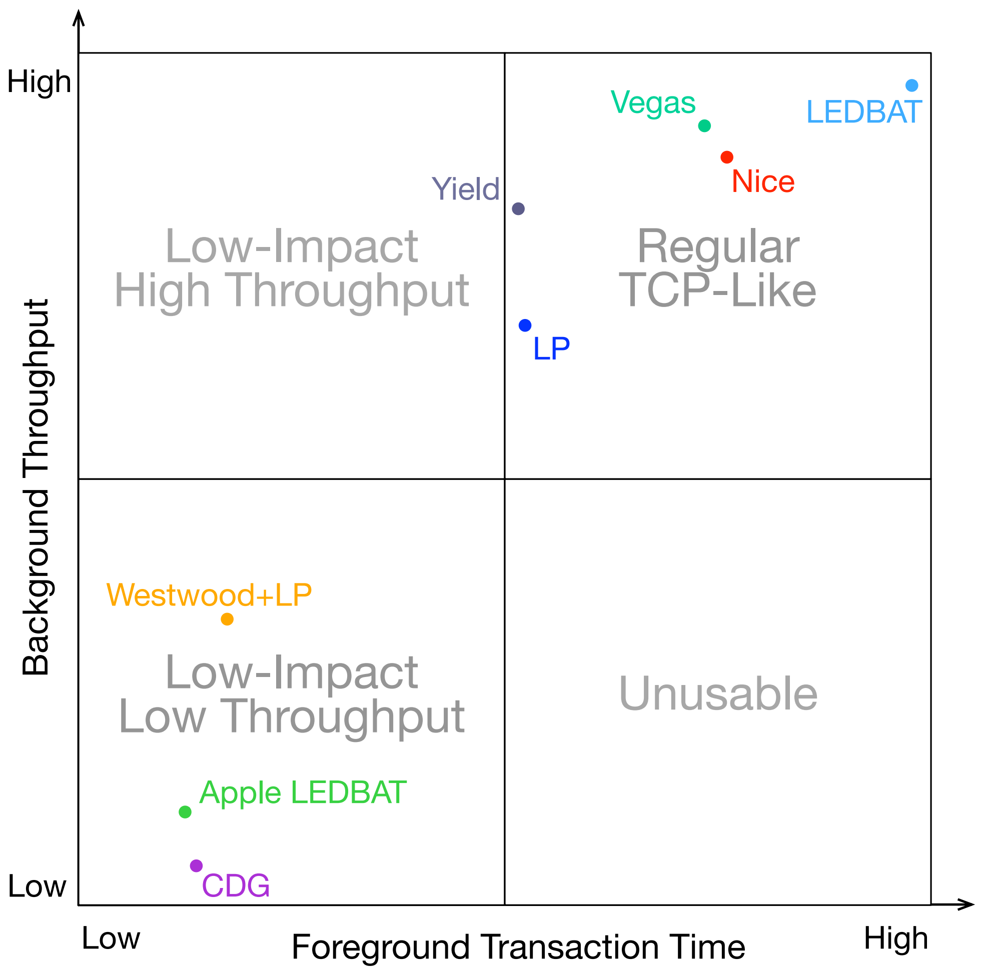


Figure 5.23.: Approximate categorisation of LBE congestion control mechanisms at 350 ms fixed-path delay.

across all experimental settings, with a mean improvement in fairness of 0.15 (and a maximum of 0.69).

However, Yield was not universally successful in demonstrating improvements over existing LBE mechanisms. When Yield uploads competed against foreground traffic, Yield reduced foreground transaction time compared to regular TCP and TCP-like mechanisms (mean reductions of 6% and 23% compared to NewReno and CUBIC, respectively), but was more aggressive than Nice and CDG. The performance of Yield in this scenario was similar to that of Westwood+LP (described in Chapter 4) and could likely be attributed to the use of OWD estimates, rather than the RTT estimates used by CDG and Nice. The use of OWD estimates would, in the upload scenario, likely prevent Yield from detecting and responding to competing traffic as only the acknowledgements from the foreground transfers would traverse the bottleneck link in the direction being measured.

Additionally, Yield achieved substantially lower throughput than normal in a limited number of cases. One example of this issue was seen in Figure 5.7, where a single Yield test achieved background throughput of 3253 KiB/s compared to the median background throughput for this experiment of 5456 KiB/s. This performance issue was caused by Yield being unable to properly increase the size of $cwnd$ once the foreground transfer has completed, due to regular (and sizeable) spikes in the OWD estimate. Based on available information from the testbed, these delay spikes are erroneous estimates produced by the OWD estimation mechanism rather than actual delay increases caused by competing traffic.

While the precise issue with the OWD estimates has not been identified, Kuhlewind and Fisches [36] have previously identified issues with the remote HZ estimation used in the calculation of the OWD estimate. Unlike previous mechanisms using OWD estimates implemented for Linux, Yield also uses these estimates to inform multiple aspects of its operation including modifying the reduction factor in addition to determining whether $cwnd$ should be increased or decreased. This increased reliance on the OWD estimates in Yield, combined with the typically lower delay target, may have exacerbated the impact of any inaccuracy in the OWD estimates.

A future variant of Yield could investigate the performance impact of using RTT estimates in place of OWD as a possible solution to this issue. The estimation of RTT is integrated into the TCP implementation of Linux, and used by existing delay-based and LBE congestion control mechanisms (such as Vegas, Nice, and CDG). While use of RTT was initially discounted due to the possibility for cross-traffic in the reverse direction to cause erroneous $cwnd$ reductions, such a modification could also improve Yield's ability to detect this cross-traffic and reduce foreground transaction times in the simultaneous upload/download scenario. Use of the RTT estimation integrated into the TCP stack for Linux would also simplify the implementation of Yield.

Future improvements to Yield, or other LBE congestion control mechanisms, should further refine the use of the adaptive delay target. While the adaptive delay target utilised by Yield showed promising results, particularly when compared to the high fixed target of LEDBAT, the modifications made to the Eclipse delay target formula limit the adaptability of the target. The modification of s_max to represent the absolute maximum of queuing delay estimates results in an inability to reduce this value, even when estimates of maximum queuing delay have been below this value over a long period of time. As a result, the calculated delay target could be inappropriately high in some scenarios, presenting an opportunity to further refine this mechanism.

A further extension to Yield could also consider the trajectory of queuing delay in order to predict and preempt impending changes in queuing delay before they occur, similar to the approach used by the derivative component of a Proportional-integral-derivative (PID) controller. Early iterations of Yield attempted to implement this approach using the delay trend information already used to control the size of the next $cwnd$ reduction. While this approach was unsuccessful in providing demonstrable improvements over the PI controller-based implementation, a predictive approach in responding to increasing queuing delay could allow $cwnd$ to be reduced more quickly when a foreground transfer begins (particularly in very high delay settings).

5.8 Summary

This chapter has presented Yield TCP, a new LBE congestion control mechanism designed to reduce the impact to foreground traffic in environments with high fixed-path delay without significant throughput penalties. Yield utilises a PI controller to better interpret and respond to changes in queuing delay. In doing so, Yield implements Dynamic Trend-Based Reduction, Adaptive Delay Targeting, and Cross-Traffic Detection.

Yield was implemented for a recent version of Linux, and evaluated against two existing LBE congestion control mechanisms: CDG and Nice. The results of this evaluation suggest that Yield was successful in reducing impact on foreground traffic in high delay settings, without significant penalties to throughput. In low delay settings, Yield had a similar impact on foreground traffic to these LBE mechanisms. Yield also demonstrated significantly better intra-protocol fairness than Nice in all experimental settings.

Future work on LBE congestion control should seek to refine the adaptive delay targeting mechanism used by Yield to provide greater adaptability, as well as improving responsiveness to competing foreground traffic when used for uploads and in the presence of very high path delay.

6 Conclusion

6.1 Overview

This chapter presents the conclusions of the thesis. Section 6.2 presents a summary of the research. Section 6.3 describes the major contributions, while Section 6.4 discusses the limitations of the research. Section 6.5 presents possible directions for future research. Finally, Section 6.6 concludes the thesis by presenting some closing remarks.

6.2 Summary of the Research

A major aim of this research was to substantially improve the understanding of the performance characteristics of LBE congestion control algorithms. To do so, seven LBE congestion control mechanisms were evaluated in a series of scenarios representative of possible real-world usage. The evaluation was carried out using a Linux testbed incorporating wired Ethernet and 802.11n wireless links. These seven algorithms were: Apple LEDBAT, CAIA Delay-Gradient (CDG), Low Extra Delay Background Transport (LEDBAT), Low Priority, Nice, Westwood-LP, and Vegas. Of these, three mechanisms — Apple LEDBAT, Nice, and Westwood-LP — were implemented based on published descriptions and available code fragments to facilitate this evaluation. The performance of the LBE mechanisms was evaluated using traffic and network profiles that typical end-users would encounter.

The results of this evaluation identified two classes of LBE congestion control mechanisms: regular TCP-like mechanisms, and low-impact mechanisms. Of the low-impact mechanisms, CDG has the lowest impact on regular TCP transfers at the expense of throughput. Nice had minimal impact on foreground traffic without the reduction in background throughput associated with other low-impact LBE mechanisms. However, neither mechanism was able to achieve a good balance between low impact on foreground traffic and background throughput in the presence of high fixed-path delay.

The results of the evaluation identified low throughput in high delay settings as a significant limitation of existing LBE mechanisms. Yield TCP is a new LBE TCP congestion control algorithm designed to address this limitation, while also maintaining low impact on regular TCP traffic observed in the low-impact mechanisms such as CDG and Nice. Yield utilises elements of a PI controller to inform its response to changes in queuing delay.

The performance of Yield was compared to low-impact mechanisms identified in the comparison of existing mechanisms: Nice and CDG. The results indicate that Yield is successful in achieving the primary design goal of improving performance in high delay settings, while performing similarly to Nice in low delay settings. While not exceeding the fairness of CDG, Yield also improves intra-protocol fairness over Nice.

6.3 Major Contributions

Through the research described in this thesis, three significant contributions were made to the area of LBE congestion control. These contributions are described in the sections below.

6.3.1 Less-than-Best-Effort Congestion Control Evaluation

Prior to this work, very few studies had evaluated a range of LBE congestion control algorithms and only one had done so independently of a newly proposed algorithm. Additionally, few such studies had evaluated performance of these algorithms outside network simulation tools such as `ns-2`. As such, a major aim of this research was to improve the limited understanding of LBE congestion control performance outside of simulated environments.

The first study of this research evaluated the performance of seven LBE congestion control algorithms in different scenarios in a Linux testbed incorporating wired Ethernet and 802.11n wireless links. The results of this evaluation identified clear performance trends among two categories of algorithms: regular TCP-like mechanisms and low-impact mechanisms. In doing so, this evaluation substantially improved the limited understanding of LBE congestion control algorithm performance and provides a basis for evaluating future algorithms.

6.3.2 Less-than-Best-Effort Algorithm Implementation

To facilitate the evaluation of LBE congestion control algorithms, three algorithms — Apple LEDBAT, Nice, and Westwood-LP — have been implemented for Linux. These implementations, which are described in Section 3.6, were based on published descriptions and available code fragments.

Source code for these implementations is included in Appendix A, and has been published on Github [60]. The availability of these mechanisms will allow future research to include a broader range of LBE congestion control algorithms for evaluation. The published code could also be used as a template for the implementation of future LBE congestion control algorithms.

6.3.3 Yield TCP

Finally, this study proposed a new algorithm for LBE congestion control: Yield TCP. Yield was designed to address the poor throughput of existing LBE algorithms in high delay settings, while avoiding performance regression in low delay settings. To facilitate its evaluation, an implementation of Yield was developed for Linux. This implementation is described in Chapter 5, with source code listed in Appendix B. The implementation of Yield has also been published on Github [61] and is available for use in future research evaluating LBE congestion control.

The evaluation of Yield indicates that it lowers impact on foreground traffic, while achieving relatively high throughput, particularly in high delay settings. Yield also demonstrates improvements in fairness over existing algorithms. In doing so, Yield successfully addresses the lack of LBE congestion control algorithms that successfully balance the need to reduce impact on foreground traffic while providing acceptable throughput. Yield also provides a basis from which future research could seek to make additional performance improvements to LBE congestion control, both through identifying techniques that could provide further improvement, as well as providing a codebase on which these changes could be implemented.

6.4 Limitations of the Study

In this research, the module parameters for the mechanisms were based on default values either from the existing Linux kernel module, or from original proposals. As a result, LEDBAT was observed to behave like a regular TCP congestion control at least partially due to the high default delay target. While this approach resulted in undesirable performance for LEDBAT, it was assumed that the default values would have been selected deliberately as a result of previous experimentation or experiences and would likely be retained in production environments. As such, the algorithms were evaluated based on these default settings. However, future research could attempt to determine whether these parameter values could be further optimised.

Evaluations of LBE congestion control, both existing mechanisms and Yield, were carried out in a testbed network. Use of the testbed network allowed for Linux-based implementations of the LBE mechanisms to be used, as well as permitting a relatively stable environment in which to examine the performance of the algorithms. This testing environment required assumptions to be made regarding the network characteristics. While the LBE algorithms were evaluated under a wide range of settings to ensure broad applicability of the results, it was infeasible to introduce the level of variability present in production networks. As such, further testing under more realistic conditions could be carried out in future research.

While Yield was successful at maintaining low impact on foreground traffic in a number of scenarios, it had limited success when competing against traffic transmitted in the opposite direction. This limitation is likely due to the use of OWD estimates, which only measure delay along the forward path. While this approach prevents Yield from erroneously responding to cross-traffic in the reverse direction, it also prevents Yield from detecting competing traffic in this scenario. Future research should consider the desirability of reacting to competing traffic on the return path, and investigate the possibility of using RTT estimates or other techniques to address this issue.

6.5 Directions for Future Work

As stated in Chapter 1, limited research has been conducted into understanding the performance of LBE congestion control. The evaluation of existing LBE congestion control mechanisms, described in Chapter 4, included seven existing mechanisms. However, algorithms for which only limited information or basis for implementation was available, such as 4CP, Eclipse, and FLOWER, were excluded. These algorithms should be implemented and tested in future work. Such testing would allow for a greater understanding of their performance compared to existing mechanisms, as well as allow for the identification of further modifications that could improve the performance of LBE congestion control.

Internet-based testing was also considered when designing the evaluation of existing LBE mechanisms. Such experiments would allow for testing of LBE congestion control under more realistic conditions, but the added unpredictability would also introduce significant challenges in interpreting the results. As such, these experiments were not pursued. Further evaluations of LBE congestion control, using Internet-based testing, could provide further differentiation between these algorithms that might otherwise not be detectable in a simulated or testbed network.

While Yield was successful in improving performance in high delay settings, it only provided small improvements when used for uploads. As a result, Yield is more aggressive than existing mechanisms such as Nice and CDG in this scenario. Additionally, the adaptive delay target used by Yield is limited by the inability to modify the maximum queuing delay estimate during operation. This limitation could result in the calculated delay target being inappropriately high in some scenarios. Future work on LBE congestion control should seek to further improve performance by addressing these limitations.

Early iterations of Yield also considered using a Proportional-integral-derivative (PID) controller as the basis for design, rather than the PI controller ultimately used. In the PID model, Yield would have considered the trajectory of queuing delay in order to preemptively respond to impending changes before they occurred. While the implementation tested with Yield was unsuccessful in providing noticeable improvements, a predictive approach in responding to increasing queuing delay could improve responsiveness to the introduction of a new foreground transfer (particularly in very high delay settings) and should be investigated further.

6.6 Closing Remarks

This research aims to improve the understanding of the performance characteristics of LBE congestion control, as well as develop and evaluate a new mechanism that addressed weaknesses in existing mechanisms. To address the limited understanding of LBE congestion control algorithm performance, seven existing LBE congestion control algorithms were evaluated.

The results of this evaluation substantially improved understanding of LBE congestion control performance by quantifying the performance of existing mechanisms, and categorising the mechanisms by their performance characteristics. To facilitate this evaluation, three previously proposed LBE congestion control algorithms were implemented and made available for use and extension in future research.

Yield TCP, a new LBE congestion control algorithm, was also developed and implemented to improve throughput in very high delay settings. Yield achieved this goal through improvements in detecting and responding to foreground traffic. In achieving its goal, Yield addresses a major limitation of existing LBE congestion control mechanisms.

References

[1] Cisco Systems, Inc., "Cisco Visual Networking Index: Forecast and Methodology, 2017-2022", Feb. 2019.

[2] Twitch Interactive, Inc. (2016). Twitch 2015 Retrospective, [Online]. Available: `https://www.twitch.tv/year/2015` (visited on Jul. 8, 2016).

[3] A. Kuzmanovic and E. Knightly, "TCP-LP: A distributed algorithm for low priority data transfer", in *22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, 2003, pp. 1691–1701.

[4] G. Carofiglio, L. Muscariello, D. Rossi, and C. Testa, "A hands-on assessment of transport protocols with lower than best effort priority", in *35th IEEE Conference on Local Computer Networks (LCN)*, Oct. 2010, pp. 8–15.

[5] A. Venkataramani, R. Kokku, and M. Dahlin, "TCP Nice: A mechanism for background transfers", *SIGOPS Operating Systems Review*, vol. 36, pp. 329–343, SI Dec. 2002.

[6] H. Shimonishi, T. Hama, M. Y. Sanadidi, M. Gerla, and T. Murase, "TCP-Westwood Low-Priority for Overlay QoS Mechanism", *IEICE Transactions on Communications*, vol. E89-B, no. 9, pp. 2414–2423, Sep. 1, 2006.

[7] S. Liu, M. Vojnovic, and D. Gunawardena, "Competitive and Considerate Congestion Control for Bulk Data Transfers", in *2007 Fifteenth IEEE International Workshop on Quality of Service*, Jun. 2007, pp. 1–9.

[8] H. Adhari, T. Dreibholz, S. Werner, and E. P. Rathgeb, "Eclipse: A New Dynamic Delay-based Congestion Control Algorithm for Background Traffic", in *2015 18th International Conference on Network-Based Information Systems*, Taipei, Taiwan: IEEE, Sep. 2015, pp. 115–123.

[9] S. Q. V. Trang, E. Lochin, C. Baudoin, E. Dubois, and P. Gélard, "FLOWER - Fuzzy lower-than-best-effort transport protocol", in *40th IEEE Conference on Local Computer Networks (LCN)*, Oct. 2015, pp. 279–286.

[10] M. A. Dye, R. McDonald, and A. W. Rufi, *Network Fundamentals: CCNA Exploration Companion Guide*. Indianapolis, Ind.: Cisco Press, 2011.

[11] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Nov. 8, 2011, 1489 pp.

[12] V. Jacobson, “Congestion avoidance and control”, in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM ’88, New York, NY, USA: ACM, 1988, pp. 314–329.

[13] 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.

[14] A. Norberg. (2010). LEDBAT Bittorrent, [Online]. Available: `http://www.bittorrent.org/beps/bep_0029.html` (visited on Sep. 24, 2013).

[15] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, “Low Extra Delay Background Transport (LEDBAT)”, RFC Editor, RFC 6817, Dec. 2012.

[16] Windows Networking Team. (Jul. 18, 2016). Announcing: New Transport Advancements in the Anniversary Update for Windows 10 and Windows Server 2016, [Online]. Available: `https://blogs.technet.microsoft.com/networking/2016/07/18/announcing-new-transport-advancements-in-the-anniversary-update-for-windows-10-and-windows-server-2016/` (visited on Mar. 28, 2017).

[17] Apple Inc. (nd). TCP LEDBAT source code, [Online]. Available: `https://opensource.apple.com//source/xnu/xnu-1699.32.7/bsd/netinet/tcp_ledbat.c` (visited on Mar. 28, 2017).

[18] H. Shimonishi, M. Y. Sanadidi, and M. Gerla, “Service differentiation at transport layer via TCP Westwood low-priority (TCPW-LP)”, in *Ninth International Symposium on Computers and Communications, 2004. Proceedings. ISCC 2004*, vol. 2, Jun. 2004, 804–809 Vol.2.

[19] D. A. Hayes and G. Armitage, “Revisiting TCP congestion control using delay gradients”, in *NETWORKING 2011*, ser. Lecture Notes in Computer Science, vol. 6641, Berlin, Heidelberg: Springer, May 9, 2011, pp. 328–341.

[20] M. Chadburn and G. Lahav. (Apr. 2016). A faster FT.com, [Online]. Available: `http://engineroom.ft.com/2016/04/04/a-faster-ft-com/` (visited on Jan. 23, 2018).

[21] Akamai Technologies, Inc., “The State of Online Retail Performance - Spring 2017”, Technical Report, Apr. 2017.

[22] T. Tsugawa, G. Hasegawa, and M. Murata, “Background TCP data transfer with Inline network measurement”, in *Asia-Pacific Conference on Communications*, Oct. 2005, pp. 459–463.

[23] M. Allman and A. Falk, “On the effective evaluation of TCP”, *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 59–70, 1999.

[24] V. Paxson and S. Floyd, “Why we don’t know how to simulate the Internet”, in *Proceedings of the 29th Conference on Winter Simulation*, ser. WSC ’97, Washington, DC, USA: IEEE Computer Society, 1997, pp. 1037–1044.

[25] R. Jain, “A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks”, *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 5, pp. 56–71, Oct. 1989.

[26] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson, “TCP Vegas: New techniques for congestion detection and avoidance”, in *SIGCOMM ’94 Proceedings of the Conference on Communications Architectures, Protocols and Applications*, ACM, 1994, pp. 24–35.

[27] C. Callegari, S. Giordano, M. Pagano, and T. Pepe, “Behavior analysis of TCP Linux variants”, *Computer Networks*, vol. 56, no. 1, pp. 462–476, Jan. 12, 2012.

[28] D. Ros and M. Welzl, “Assessing LEDBAT’s delay impact”, *IEEE Communications Letters*, vol. 17, no. 5, pp. 1044–1047, May 2013.

[29] H. S. Wong and H. L. Hung. (nd). TCP Low Priority source code, [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_lp.c` (visited on Feb. 5, 2019).

[30] S. Liu, M. Vojnovic, and D. Gunawardena, “Competitive and Considerate Congestion Control for Bulk-Data Transfers”, Microsoft Research, Technical Report MSR-TR-2006-24, Feb. 2006.

[31] 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.

[32] C. L. T. Man, G. Hasegawa, and M. Murata, “Available bandwidth measurement via TCP connection”, in *Proceedings of IFIP/IEEE MMNS 2004 E2EMON Workshop*, 2004, pp. 38–44.

[33] C. L. T. Man, G. Hasegawa, and M. Murata, “ImTCP: TCP with an inline measurement mechanism for available bandwidth”, *Computer Communications*, vol. 29, no. 10, pp. 1614–1626, Jun. 2006.

[34] T. Tsugawa, G. Hasegawa, and M. Murata. (2005). Implementation of ImTCP for FreeBSD, [Online]. Available: `http://www.ane.cmc.osaka-u.ac.jp/~hasegawa/imtcp/imtcp_freebsd.html` (visited on Feb. 15, 2018).

[35] S. Shalunov, “Low Extra Delay Background Transport (LEDBAT)”, IETF Secretariat, Internet-Draft draft-ietf-ledbat-congestion-00.txt, Oct. 22, 2009.

[36] M. Kühlewind. (nd). Homepage of Mirja Kühlewind, [Online]. Available: `http://mirja.kuehlewind.net/code.html` (visited on Mar. 27, 2017).

[37] G. Carofiglio, L. Muscariello, D. Rossi, and S. Valenti, "The quest for LEDBAT fairness", in *IEEE Global Comunications Conference (GLOBECOM)*, IEEE, 2010, pp. 1–6.

[38] G. Armitage and N. Khademi, "Using delay-gradient TCP for multimedia-friendly background transport in home networks", in *38th IEEE Conference on Local Computer Networks (LCN)*, IEEE, 2013, pp. 509–515.

[39] K. K. Jonassen, "Implementing CAIA Delay-Gradient in Linux", Masters Thesis, University of Oslo, Oslo, Norway, May 2015.

[40] S. Ha and I. Rhee, "Hybrid slow start for high-bandwidth and long-distance networks", in *6th International Workshop on Protocols for Future, Large-Scale & Diverse Network Transports (PFLDNeT)*, 2008, pp. 1–6.

[41] K. K. Jonassen. (nd). TCP CDG source code, [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_vegas.c` (visited on Feb. 5, 2019).

[42] D. X. Wei and P. Cao, "NS-2 TCP-Linux: An NS-2 TCP implementation with congestion control algorithms from Linux", in *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator - WNS2 '06*, Pisa, Italy: ACM Press, 2006, p. 9.

[43] S. Floyd and E. Kohler, "Internet research needs better models", *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 29–34, 2003.

[44] C. Bueno. (Jul. 2011). Doppler: Internet Radar, [Online]. Available: `https://www.facebook.com/notes/facebook-engineering/doppler-internet-radar/10150212498738920/` (visited on Apr. 2, 2019).

[45] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi, "Proportional Rate Reduction for TCP", in *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement 2011, Berlin, Germany - November 2-4, 2011*, 2011.

[46] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "LEDBAT: The new BitTorrent congestion control protocol", in *Proceedings of 19th International Conference on Computer Communications and Networks (ICCCN)*, 2010, pp. 1–6.

[47] R. Jain, D.-M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems", *ACM Transactions on Computer Systems*, vol. cs.NI/9809099, 1984.

[48] S. Agarwal. (Aug. 2018). Public Cloud Inter-region Network Latency as Heat-maps, [Online]. Available: https://medium.com/@sachinkagarwal/public-cloud-inter-region-network-latency-as-heat-maps-134e22a5ff19 (visited on Mar. 4, 2020).

[49] S. Souders. (Oct. 2010). HTTP Archive, [Online]. Available: http://httparchive.org/index.php (visited on May 21, 2015).

[50] A. Reid and J. Lorenz, *Working at a Small-to-Medium Business or ISP : CCNA Discovery Learning Guide*. Indianapolis, Ind.: Cisco Press, 2008.

[51] S. Floyd, "Metrics for the Evaluation of Congestion Control Mechanisms", RFC Editor, RFC 5166, Mar. 2008.

[52] B. Briscoe, "Flow rate fairness: Dismantling a religion", *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 63–74, Mar. 2007.

[53] K. Ong. (2017). Less-than-Best-Effort TCP congestion control modules, [Online]. Available: https://github.com/kovalian/lbetcp (visited on Aug. 9, 2017).

[54] S. Hemminger. (nd). TCP Vegas source code, [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_vegas.c (visited on Feb. 5, 2019).

[55] A. Dell'Aera, L. A. Grieco, and S. Mascolo, "Linux 2.4 implementation of Westwood+ TCP with rate-halving: A performance evaluation over the Internet", in *IEEE International Conference on Communications*, vol. 4, Paris, France, 2004, pp. 2092–2096.

[56] 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, Technical Report 150529A, 2015, p. 45.

[57] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 6th ed. Upper Saddle River [N.J.]: Pearson, 2010, 819 pp.

[58] P. Natarajan, G. White, R. Pan, and F. Baker, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem", RFC Editor, RFC 8033, Feb. 2017.

[59] A. Kuzmanovic and E. Knightly, "TCP-LP: Low-priority service via end-point congestion control", *IEEE/ACM Transactions on Networking*, vol. 14, no. 4, pp. 739–752, 2006.

[60] K. Ong, S. Zander, D. Murray, and T. McGill, “Experimental evaluation of Less-than-Best-Effort TCP congestion control mechanisms”, in *42nd IEEE Conference on Local Computer Networks (LCN)*, Singapore: IEEE, 2017.

[61] K. Ong. (2018). Yield TCP, [Online]. Available: `https://github.com/kovalian/tcp_yield` (visited on Aug. 9, 2017).

A Less-than-Best-Effort Code Listing

A.1 Overview

This appendix presents the source code for the implementations of Nice, Westwood+LP, and Apple LEDBAT described in Chapter 3.

A.2 Nice

A.2.1 Per-Acknowledgement Operation

```
void tcp_nice_pkts_acked(struct sock *sk, u32 cnt, s32 rtt_us)
{
    struct nice *nice = inet_csk_ca(sk);
    u32 vrtt;

    if (rtt_us < 0)
      return;

    /* Never allow zero rtt or baseRTT */
    vrtt = rtt_us + 1;

    /* Filter to find propagation delay: */
    if (vrtt < nice->baseRTT)
      nice->baseRTT = vrtt;

    /* Initialise maxRTT to 2*minRTT */
    if (nice->cntRTT == 0)
      nice->maxRTT = nice->baseRTT * 2;
    /* Find the min RTT during the last RTT to find
     * the current prop. delay + queuing delay:
     */
    nice->minRTT = min(nice->minRTT, vrtt);
    nice->maxRTT = max(nice->maxRTT, vrtt);
    nice->cntRTT++;

    if (vrtt > ((100UL - threshold) * nice->baseRTT + threshold *
        nice->maxRTT) / 100UL) {
      nice->numCong++;
    }
}
```

A.2.2 Per-Round Operation

```
static void tcp_nice_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
  struct tcp_sock *tp = tcp_sk(sk);
  struct nice *nice = inet_csk_ca(sk);

  if (nice->fractional_cwnd > 2&&
        nice->nice_timer == nice->fractional_cwnd) {
    /* Send two packets in this RTT then reset the timer */
    tp->snd_cwnd = 2;
    nice->nice_timer = 1;
  } else if (nice->fractional_cwnd > 2) {
    /* Waiting to send packets */
    tp->snd_cwnd = 0;
    nice->nice_timer++;
  }

  if (!nice->doing_nice_now) {
    if (tp->snd_cwnd <= 2&& nice->fractional_cwnd >= 2&&
          nice->fractional_cwnd <= max_fwnd) {
      tcp_reno_fractional_ca(sk, ack, acked);
    } else {
      /* Just do Reno */
      tcp_reno_cong_avoid(sk, ack, acked);
    }
    return;
  }

  if (after(ack, nice->beg_snd_nxt)) {
    /* Do the Vegas once-per-RTT cwnd adjustment. */

    /* Save the extent of the current window so we can use this
     * at the end of the next RTT.
     */
    nice->beg_snd_nxt = tp->snd_nxt;

    /* We do the Vegas calculations only if we got enough RTT
```

```
 * samples that we can be reasonably sure that we got
 * at least one RTT sample that wasn't from a delayed ACK.
 * If we only had 2 samples total,
 * then that means we're getting only 1 ACK per RTT, which
 * means they're almost certainly delayed ACKs.
 * If  we have 3 samples, we should be OK.
 */

if (nice->cntRTT <= 2) {
  /* We don't have enough RTT samples to do the Vegas
    * calculation, so we'll behave like Reno.
    */
  if (tp->snd_cwnd <= 2&& nice->fractional_cwnd >= 2&&
        nice->fractional_cwnd <= max_fwnd) {
    tcp_reno_fractional_ca(sk, ack, acked);
  } else {
    /* Just do Reno */
    tcp_reno_cong_avoid(sk, ack, acked);
  }
} else {
  u32 rtt, diff;
  u64 target_cwnd;

  /* We have enough RTT samples, so, using the Vegas
    * algorithm, we determine if we should increase or
    * decrease cwnd, and by how much.
    */

  /* Pluck out the RTT we are using for the Vegas
    * calculations. This is the min RTT seen during the
    * last RTT. Taking the min filters out the effects
    * of delayed ACKs, at the cost of noticing congestion
    * a bit later.
    */
  rtt = nice->minRTT;

  /* Calculate the cwnd we should have, if we weren't
    * going too fast.
  *
  * This is:
  *     (actual rate in segments) * baseRTT
  */
target_cwnd = (u64)tp->snd_cwnd * nice->baseRTT;
do_div(target_cwnd, rtt);

/* Calculate the difference between the window we had,
  * and the window we would like to have. This quantity
  * is the "Diff" from the Arizona Vegas papers.
  */
diff = tp->snd_cwnd * (rtt-nice->baseRTT) / nice->baseRTT;

if (diff > gamma && tcp_in_slow_start(tp)) {
  /* Going too fast. Time to slow down
    * and switch to congestion avoidance.
    */

  /* Set cwnd to match the actual rate
    * exactly:
    *   cwnd = (actual rate) * baseRTT
    * Then we add 1 because the integer
    * truncation robs us of full link
    * utilization.
    */
  tp->snd_cwnd = min(tp->snd_cwnd, (u32)target_cwnd+1);
  tp->snd_ssthresh = tcp_nice_ssthresh(tp);
  nice->numCong = 0;

} else if (tcp_in_slow_start(tp)) {
  /* Slow start. */
  tcp_slow_start(tp, acked);
} else if (nice->numCong > tp->snd_cwnd / fraction_divisor) {
  /* Nice detected too many congestion events
    * perform multiplicative window reduction.
    */
  if (tp->snd_cwnd > 2&& nice->fractional_cwnd == 2) {
    tp->snd_cwnd = tp->snd_cwnd / 2;
    } else if (nice->fractional_cwnd <= max_fwnd) {
      nice->fractional_cwnd *= 4;
    }

    nice->numCong = 0; // Reset multiplicative decrease counter.
  } else {
    /* Congestion avoidance. */

    /* Figure out where we would like cwnd
      * to be.
      */
    if (diff > beta) {
      /* The old window was too fast, so
        * we slow down.
        */
      if (tp->snd_cwnd > 2&& nice->fractional_cwnd == 2) {
        tp->snd_cwnd--;
      } else if (nice->fractional_cwnd <= max_fwnd) {
        nice->fractional_cwnd+=2;
      }

      tp->snd_ssthresh
        = tcp_nice_ssthresh(tp);
    } else if (diff < alpha) {
      /* We don't have enough extra packets
        * in the network, so speed up.
        */
      if (tp->snd_cwnd >= 2&& nice->fractional_cwnd == 2) {
        tp->snd_cwnd++;
      } else if (nice->fractional_cwnd <= max_fwnd) {
        nice->fractional_cwnd-=2;
      }
    } else {
      /* Sending just as fast as we
        * should be.
        */
    }
  }
```

```
      if (tp->snd_cwnd < 2&& nice->fractional_cwnd == 2)
        tp->snd_cwnd = 2;
      else if (tp->snd_cwnd > tp->snd_cwnd_clamp)
        tp->snd_cwnd = tp->snd_cwnd_clamp;

      tp->snd_ssthresh = tcp_current_ssthresh(sk);
    }

    /* Wipe the slate clean for the next RTT. */
    nice->cntRTT = 0;
    nice->minRTT = 0x7fffffff;
    nice->maxRTT = 0;
    nice->numCong = 0;
  }
  /* Use normal slow start */
  else if (tcp_in_slow_start(tp))
    tcp_slow_start(tp, acked);
}
```

A.3 Westwood+LP

A.3.1 Per-Acknowledgement Operation

```
static void tcp_westwood_pkts_acked(struct sock *sk, u32 cnt, s32 rtt)
{
  struct westwood *w = inet_csk_ca(sk);

  if (rtt > 0)
    w->rtt = usecs_to_jiffies(rtt);
}
```

A.3.2 Per-Round Operation

```
static void tcp_westwood_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
  struct tcp_sock *tp = tcp_sk(sk);
  struct westwood *w = inet_csk_ca(sk);

  u32 ewr_thresh = 0;
  u32 queue_length = 0;
  u32 rtt = 0;

  /* Negate RTT as a factor if delay_loss has no value */
  if (w->delay_loss > 1) {
    rtt = w->rtt;
  }

  /* Check that we have an RTT estimate before computing EWR threshold */
  /* Use delay_min and delay_max until the first EWR event */
  if (w->dmin_avg != w->dmax_avg && w->dmax_avg != 0) {
    queue_length = tp->snd_cwnd - w->bw_est * w->rtt_min / tp->advmss;
    ewr_thresh = (beta * (100 - 100* (rtt << 2) / w->delay_loss) / 100)
          * (100 - 100* w->dmin_avg / w->dmax_avg) / 100;
  } else if (w->delay_min != w->delay_max && w->delay_max != 0
         && !tcp_in_slow_start(tp)) {
```

```
        queue_length = tp->snd_cwnd - w->bw_est * w->rtt_min / tp->advmss;
        ewr_thresh = (beta * (100 - 100* (rtt << 2) / w->delay_loss) / 100)
              * (100 - 100* w->delay_min / w->delay_max) / 100;
      }

      if (queue_length > ewr_thresh) {
        tp->snd_cwnd = tp->snd_ssthresh = tcp_westwood_bw_rttmin(sk);

        /* Update min and max delay averages with values from this EWR window */
        w->dmin_avg = westwood_update_delay(w->delay_min, w->dmin_avg);
        w->dmax_avg = westwood_update_delay(w->delay_max, w->dmax_avg);

        /* Current RTT becomes lowest and highest RTT observed */
        w->delay_max = w->delay_min = w->rtt;
      } else {
        tcp_reno_cong_avoid(sk, ack, acked);
      }

    }
```

A.4 Apple LEDBAT

A.4.1 Per-Acknowledgement Operation

The RFC6817 LEDBAT implementation used as the template for this version of Apple LEDBAT does not specify any new per-acknowledgement behaviour.

A.4.2 Per-Round Operation

```
void tcp_apledbat_cong_avoid(struct sock *sk, u32 ack, u32 acked) {

  struct tcp_sock *tp = tcp_sk(sk);
  struct ledbat *ledbat = inet_csk_ca(sk);

  u32 delay = 0;
  u32 queuing_delay;
  int off_target;
  u32 max_allowed_cwnd;

  /* estimate the remote peers time granularity ->
     doesn't work and therefore not used */
  //estimate_remote_HZ(sk);

  // remember first timestamp of local and remote host as base
  if (ledbat->remote_time_offset == 0)
    ledbat->remote_time_offset = tp->rx_opt.rcv_tsval;
  if (ledbat->local_time_offset == 0)
    ledbat->local_time_offset = tp->rx_opt.rcv_tsecr;

  //calculate current OWD
  //delay * 1000 * 1/HZ; -> Result in [s]. Multiply by 1000 for [ms]
  u32 time = (tp->rx_opt.rcv_tsval - ledbat->remote_time_offset)
     *1000/HZ;
  u32 remote_time = (tp->rx_opt.rcv_tsecr - ledbat->local_time_offset)
     *1000/HZ;
  if (time > remote_time)
  delay = time - remote_time;

// update delays
tcp_ledbat_update_base_delay(sk, delay);
tcp_ledbat_update_current_delay(sk, delay);
ledbat->base_delay = min(ledbat->base_delay, delay);

// calculate queuing delay
if (ledbat->current_delays.buffer!=NULL &&
       ledbat->base_delays.buffer!=NULL) {
  queuing_delay = tcp_ledbat_get_min_from_list(&ledbat->current_delays)
     - tcp_ledbat_get_min_from_list(&ledbat->base_delays);
} else {
  queuing_delay = delay - ledbat->base_delay;
}

/* don't change cwnd is not cwnd-limited */
if (!tcp_is_cwnd_limited(sk))
  return;

/* In "safe" area, increase exponentially. */
if (tp->snd_cwnd <= tp->snd_ssthresh) {
  acked = tcp_slow_start(tp, acked);
  if (!acked)
    return;
}

/* LEDABT cwnd increase/decrease */
off_target = target - queuing_delay;

if (off_target >= 0) {
/* under delay target, apply additive increase */
  tcp_reno_cong_avoid(sk, ack, acked);
} else {
/* over delay target, apply 1/8th cwnd reduction */
  u32 decr;

  decr = tp->snd_cwnd >> 3;
```

```
    tp->snd_cwnd -= decr;
  }

  // From RFC6817: max_allowed_cwnd = flightsize + ALLOWED_INCREASE * MSS
  max_allowed_cwnd = tp->packets_out + acked + ALLOWED_INCREASE;
  tp->snd_cwnd = min(tp->snd_cwnd, max_allowed_cwnd);
  // or
  // cwnd = max(MIN_CWND, min(cwnd, tp->snd_cwnd_clamp));

  // set cwnd
  tp->snd_cwnd = max(MIN_CWND, tp->snd_cwnd);

  // also adapt ssthreash if the cwnd is reduced!
  if (tp->snd_cwnd <= tp->snd_ssthresh)
    tp->snd_ssthresh = tp->snd_cwnd-1;
}
```

Yield TCP Code Listing

B

B.1 Overview

This appendix presents the source code for the Yield TCP implementation described in Chapter 5.

B.1.1 Per-Acknowledgement Operation

```
void tcp_yield_pkts_acked(struct sock *sk, u32 cnt, s32 rtt_us) {

    struct tcp_sock *tp = tcp_sk(sk);
    struct yield *yield = inet_csk_ca(sk);

    u32 time, remote_time;
    s32 trend = 0;

    /* Capture initial timestamps on first run */
    if (yield->remote_time_offset == 0) {
        yield->remote_time_offset = tp->rx_opt.rcv_tsval;
    }
    if (yield->local_time_offset == 0) {
        yield->local_time_offset = tp->rx_opt.rcv_tsecr;
    }

    time = (tp->rx_opt.rcv_tsval - yield->remote_time_offset)
        * 1000 / HZ;
    remote_time = (tp->rx_opt.rcv_tsecr - yield->local_time_offset)
        * 1000 / HZ;

    if (time > remote_time) {
        yield->delay = time - remote_time;
```

```
}

/* Update delay_min and delay_max as needed */
if (yield->delay < yield->delay_min) {
    yield->delay_min = yield->delay;
} else if (yield->delay > yield->delay_max) {
    yield->delay_max = yield->delay;
}

/* Update the smoothed minimum */
if (((yield->delay_min << 3) < yield->delay_smin) ||
    yield->delay_smin == 0) {
    /* overwrite if the latest minimum is below the smoothed */
    yield->delay_smin = yield->delay_min << 3;
} else if (yield->delay_min > yield->delay_smin) {
    /* otherwise update the moving average */
    yield->delay_smin =
        update_delay(yield->delay, yield->delay_smin, 3);
}

/* Update the smoothed maximum */
if (((yield->delay_max << 3) > yield->delay_smax) ||
    yield->delay_smax == 0) {
    /* overwrite if the latest maximum is below the smoothed */
    yield->delay_smax = yield->delay_max << 3;
} else if (yield->delay_max > yield->delay_smax) {
    /* otherwise update the moving average */
    yield->delay_smax =
        update_delay(yield->delay, yield->delay_smax, 3);
}

if (yield->prev_delay != 0) {
/* determine whether delay is increasing or decreasing */
    trend = yield->delay - (yield->prev_delay >> hist_factor);

    if (yield->delay_trend != 0&& trend >
        max((yield->delay_trend / trend_factor) * ct_threshold, 1)) {
            yield->cross_traffic = 1;
            }
        }

        if (trend >= increase_threshold && yield->cross_traffic == 1) {
        /* delay is increasing and cross traffic is present
            so a bigger decrease will be needed */
            yield->reduction_factor -= 1;
            yield->reduction_factor =
                max(yield->reduction_factor, maxc_reduction);
        } else if (trend >= increase_threshold) {
        /* delay is increasing so a bigger decrease will be needed */
            yield->reduction_factor -= 1;
            yield->reduction_factor =
                max(yield->reduction_factor, max_reduction);
        } else if (trend <= decrease_threshold) {
        /* delay is decreasing so make the next decrease smaller */
            yield->reduction_factor += 1;
            yield->reduction_factor =
                min(yield->reduction_factor, min_reduction);
        }

        if (yield->delay < yield->delay_min && yield->delay > 0) {
            yield->delay_min = yield->delay;
        }

        /* current delay reading becomes last seen */
        yield->prev_delay =
            update_delay(yield->delay, yield->prev_delay, hist_factor);

        /* update delay trend history using current */
        yield->delay_trend = update_delay_trend(trend, yield->delay_trend);

    }
```

B.1.2 Per-Round Operation

```
static void tcp_yield_cong_avoid(struct sock *sk, u32 ack, u32 acked) {

    struct tcp_sock *tp = tcp_sk(sk);
    struct yield *yield = inet_csk_ca(sk);

    int target = 0; /* target queuing delay (in ms) */
    u32 qdelay = 0;
    int off_target;

    /* Window under ssthresh, do slow start. */
    if (tp->snd_cwnd <= tp->snd_ssthresh) {
        acked = tcp_slow_start(tp, acked);
        if (!acked)
            return;
    }

    /* Only calculate queuing delay once we have some delay estimates */
    if (yield->delay_smin != 0) {
        qdelay = yield->delay - (yield->delay_smin >> 3);
    }

    /* Calculate target queuing delay */
    target = beta * 100* ((yield->delay_smax - yield->delay_smin) >> 3)
        / 10000;

    off_target = target - qdelay;

    if (off_target >= 0) {
    /* under delay target, apply additive increase */
        tcp_reno_cong_avoid(sk, ack, acked);
    } else {
    /* over delay target, apply multiplicative decrease */
        u32 decrement;

        decrement = tp->snd_cwnd >> yield->reduction_factor;
        tp->snd_cwnd -= decrement;
        /* just decreased, next decrease should be smaller */
        yield->reduction_factor += 1;
    }

    tp->snd_cwnd = max(MIN_CWND, tp->snd_cwnd);

    yield->delay_min = UINT_MAX;
    yield->delay_max = 0;
    yield->cross_traffic = 0;
}
```