UiO : Department of Informatics
University of Oslo

Vulture – Variable Aggressiveness Ultra Low Impact Transport Using Receiver-based Flow-control Mechanism

Research Report 474
Qian Li and Michael Welzl
November 28, 2017

ISBN 978-82-7368-439-4
ISSN 0806-3036

Abstract

Less than Best Effort (LBE) transports are transport protocols that use spare bandwidth left by Best Effort (BE) transports to fulfill their own data transfer tasks. Such kind of protocols can be used by non-delay/bandwidth sensitive applications such as software updating, peer-to-peer file sharing, prefetching, and replication to realize background transfer. By incorporating BE and LBE protocols into the Internet, we can realize flow-based prioritization, which can ensure that the available bandwidth is divided among different applications wisely and give every application an opportunity to run.

We have designed and developed a receiver-side flow-control-based LBE congestion control mechanism. By observing the sign and value of packet interval variations at the receiver side, we can successfully estimate the degree of congestion in the network and distinguish situation (a): the congestion is caused by the LBE flow itself from situation (b): the congestion is caused by the LBE flow as well as other flows. The protocol runs in a conservative mode if it detects other flows, otherwise, it runs in an aggressive mode to quickly reap the available bandwidth.

Testbed evaluation results show that our LBE protocol inflicts low throughput impact on BE flows with which it shares the same bottleneck. The average goodput achieved by a BE flow does not change after an LBE flow is started to operate. An LBE flow can add, on average, 40% round trip delay to a BE flow. Bandwidth utilization ratios of an LBE flow when it runs alone and shares a bottleneck with a constant bit rate BE flow are 99% (stable state, LBE only), 93% (overall, LBE only), and 83% (LBE and BE) respectively. When a BE flow exhibits on-off behavior, the LBE flow's bandwidth utilization ratio decreases with the decrease of the on/off interval. An LBE flow can immediately yield its bandwidth when a BE flow is started. Finally, our LBE flows are fair toward each other. Jain's fairness index is 0.99 for 4, 16, and 64 LBE flows.

0.1 Introduction

Transmission Control Protocol (TCP) [8] is the most important and widely used transport protocol on today's Internet. Many versions of TCP have been proposed during past a few decades. These versions mainly differ in how they detect and react to network congestion. In this report, we refer RFC5681 [1] as standard TCP and the rest as TCP variants. Standard TCP and most of its variants are best effort, which means that a TCP flow strives to obtain a fair share of link bandwidth when it shares the link with other TCP flows. However, such fair shares are not always desirable. Imagine, two Internet applications share one network bottleneck, one application is streaming a Standard Definition (SD) video and another is downloading a large file. Suppose the bottleneck bandwidth is 4Mbps and the minimum bandwidth requirement to stream an SD video is 3Mbps. If both of the applications use best effort TCP and both can obtain a fair share, each of them will get roughly 2Mbps bandwidth, which is not sufficient for the first application to play back the video smoothly. However, if we divide the bottleneck bandwidth between the two applications in another way, for example, 3Mbps for the first application and 1Mbps for the second, both applications can run correctly in parallel just that the second one will experience a lower downloading speed. There are many similar scenarios in real life. For example, surfing the web while downloading software updates, online chatting while sending pictures via an instant messenger and playing online games while sharing music using a peer to peer software. The former applications in the above examples are delay/bandwidth sensitive, that is, they cannot run correctly if their minimum bandwidth/delay requirements are not met. On the contrary, the latter applications can run with arbitrarily low bandwidth and long delay and do not have an upper limit in terms of bandwidth consumption. Therefore, it is of vital importance to prioritize these different types of applications when they share the same network bottleneck, so that the former applications' minimum bandwidth and delay requirements are met.

This demand drives us to devise a Less than Best Effort (LBE) transport protocol to transfer packets for non-delay/bandwidth-sensitive applications in the background. An LBE TCP should be non-intrusive or transparent to Best Effort (BE) TCP flows. That is, an LBE TCP flow should inflict as little delay and throughput impact on a BE TCP flow as possible. To realize transparency to BE flows, an LBE flow should opportunistically utilize the spare bandwidth left by BE flows. When multiple LBE flows share one bottleneck, they should share the spare bandwidth fairly. There are a variety of LBE proposals in literature. [10] provides a survey of pure end-to-end LBE systems. LBE can be realized as TCP congestion control (e.g. [11]) or as an extension to TCP (e.g. [4]). It can be implemented at the sender side (e.g. [14], [5]), the receiver side (e.g., [12], [7]), or at both sides ([11]).

We have designed and developed a Variable aggressiveness Ultra Low impact Transport that Uses REceiver-based flow-control mechanism – Vulture. Vulture uses packet interval variation as congestion signal and receive window as the limiting factor of sending rate. Experiment results show at Vulture has low

impact on BE flows, can achieve acceptable bandwidth utilization ratio under various scenarios, can quickly reap and yield bandwidth, and is fair toward each other.

The report is organized as follows. We review related work in section 0.2. Section 0.3 presents Vulture's system design and building blocks. Section 0.4 evaluates Vulture against its design goals with testbed experiments. Finally, section 0.5 concludes the report and identifies future work.

0.2 Related Work

We are not the first one who has proposed an LBE transport protocol. Similar proposals can be found in literature. [10] provides a survey of early stage LBE systems. This article focuses on Internet-oriented pure end-to-end LBE approaches. The authors classify such LBE systems into delay-based, non-delay-based and upper-layer approaches. Delay-based protocols use round trip delay or one way delay as signal to detect incipient network congestion. Example transports include LEDBAT [11], TCP-NICE [14], and TCP-LP [5]. Non-delay-based protocols use other measurements other than delay as congestion signal. For example, 4CP [6] uses loss as congestion signal. Such kind of transports usually make the sender react to congestion more aggressively than BE TCPs, but cannot detect congestion at its onset. Upper-layer approaches do not modify TCP directly. Instead, they alter a certain variable, for example, receive socket buffer [12] or receive window [7], to make the sender change its sending rate.

FLOWER [13] is a more recent delay-based LBE transport protocol. It is designed to overcome LEDBAT's two performance issues – latecomer unfairness and being too aggressive against BE TCP. FLOWER replaces LEDBAT's P-type controller with a fuzzy controller and adds a peak-valley detector. Simulation results show that FLOWER is capable of solving the performance issues of LEDBAT but at the cost of increased processing time and higher complexity.

[4] is another recent LBE mechanism which is capable of meeting soft deadlines. It works at the sender side as an extension to TCP. It uses the original congestion detection algorithms employed in different TCP congestion control mechanisms, but alters the degree of aggressiveness of a sender's congestion reaction. That is, the sender reacts to congestion more aggressively when the deadline is far, but gradually reduces its aggressiveness as the deadline approaches, therefore, the protocol exhibits varying degree of LBE behavior. Similar to 4CP, it cannot detect congestion at the onset if the congestion control is loss based.

Vulture differs from other work mainly in how congestion is detected. Unlike other delay-based congestion controls, Vulture does not use one way delay or round trip delay as an indicator of congestion. Instead, it uses packet interval variation. One advantage of using packet interval variation is that we do not need to estimate the minimum one way or round trip delay. Obtaining an accurate minimum one way or round trip delay is very challenging in reality. The wrong estimation of such delay is the root cause of the late-comer unfairness and

being too aggressive to short-lived TCP flows issues encountered by LEDBAT [9].

0.3 Design and Implementation

0.3.1 System Overview

Vulture is a flow-control-based receiver-side congestion control. A Vulture receiver uses receive window (rwnd) to control the sending rate of a sender. Upon the receipt of an acknowledgement from a receiver, the sender calculates an effective window (ewnd), which specifies how many bytes the sender can send during one RTT. The effective window is the minimum of the sender's congestion window (cwnd) and the receiver's receive window (rwnd). $ewnd = min(cwnd, rwnd)$. The sending rate of the sender R can be derived by $R = ewnd/RTT$, where RTT is the Round Trip Time between the sender and the receiver. Therefore, by reducing the size of rwnd to a value smaller than cwnd, the receiver can effectively reduce the size of ewnd of the sender, hence, limiting the sending rate R of the sender.

A Vulture receiver uses Packet Interval Variation (PIV) as congestion indicator. To be specific, the receiver monitors the time interval between two data packets. If the interval is significantly altered during transfer, the receiver believes there is congestion along the path. On the other hand, if the interval is kept almost unchanged, the receiver knows that there is no congestion in the network. Furthermore, by analyzing the number and value of positive PIVs, a Vulture receiver can infer if the congestion is caused solely by the Vulture flow itself or also by other flows. If the former is true, the protocol goes into an aggressive mode, in which it can tolerate longer queues. However, if the latter is true, Vulture goes into a conservative mode, in which it only tolerates short queues.

A Vulture receiver makes rwnd moderation policies based on current network congestion estimate. The receiver calculates a new rwnd using the current rwnd moderation policy for each outgoing acknowledgement. The receiver constantly estimates the sender's congestion window (cwnd) so that it can set rwnd to the cwnd estimate when the protocol is switched to LBE mode for the first time. The rationale behind this is that the rwnd value maintained by TCP in the BE mode can be much greater than the sender's cwnd [3]. If we set an LBE protocol's rwnd to be equal to a BE protocol's rwnd, it will make the sender increase its sending rate during the next RTT even if when the network is congested. Apparently, this contradicts the goal of an LBE protocol.

The above are the building blocks of Vulture and will be presented in much details in the following subsections.

Vulture has **five design goals: (1)** correctly estimate spare bandwidth. **(2)** Minimize its impact on BE TCP flows – A Vulture flow should remain transparent to BE TCP flows with which it shares the same bottleneck. In other words, a BE TCP flow should not experience significant throughput degradation

or delay increase when a Vulture LBE flow is running in parallel. **(3)** Maximize bandwidth utilization – Vulture flows should strive to use all available spare bandwidth. **(4)** Quickly yield and reap bandwidth – A Vulture flow should immediately decrease its bandwidth usage as soon as a BE flow increases its bandwidth consumption and quickly increase its bandwidth usage when a BE flow releases more bandwidth. **(5)** Realize intra-flow fairness – Vulture flows should be able to share spare bandwidth fairly.

Vulture is implemented as a Linux kernel module at the receiver side. The kernel module needs TCP, in the kernel core, to capture incoming data packets and outgoing acknowledgements and send relative information to it. We also implemented a user-space library so that applications can start and stop Vulture at any time they need.

0.3.2 Packet Interval Variation based Spare Bandwidth Detection

Unlike other delay-based congestion controls, which use RTT to detect congestion, Vulture uses Packet Interval Variations (PIV) to detect network congestion or spare bandwidth. The algorithm we use is inspired by [2]. The authors of [2] used a series of experiments to show that the time interval between a pair of packet can be significantly altered by network congestion or bottleneck queue along the transmission path. The intervals measured at the receiver end can be much greater or smaller than the ones measured at the sender side if the network is heavily loaded or congested. In comparison, if the network is lightly loaded, the intervals are kept almost unchanged. Figure 1 illustrates this phenomenon.

In order to eliminate noise and make accurate estimation, [2] proposes to use sliding standard deviation and $\epsilon - percentile$. To be specific, the proposed algorithm calculates a standard deviation for every 10 measured PIVs at the receiver. When there are 30 such standard deviations, it calculates the $\epsilon - percentile$, where $\epsilon = 0.3$, of these values. The obtained value is then used as a load estimate of the network.

Although this algorithm can detect network congestion or the existence of a bottleneck queue, it cannot infer which flow(s) built up this queue. Nevertheless, this latter information is of vital importance for Vulture's success. An LBE flow can build up a queue by itself if the LBE sender sends data at a higher speed than the bottleneck can handle. If we react to such queues, it will take the LBE flow a very long time to find the maximum available bandwidth. Figure 2 reveals this problem using a scenario where an LBE sender reduces its sending rate as soon as it detects a queue no matter who builds up the queue. As is shown, the LBE flow cannot fully utilize the spare bandwidth (10Mbps) even if when the BE flow is turned off for almost 30 seconds.

We boldly conjecture that the number and value of positive PIVs measured by a Vulture LBE receiver will increase when the LBE flow's packets are interleaved with packets from other flows due to the fact that the inter packet intervals can be elongated by cross-traffic packets when waiting in the queue. In comparison, when little or no packets from other flows are in the queue, either

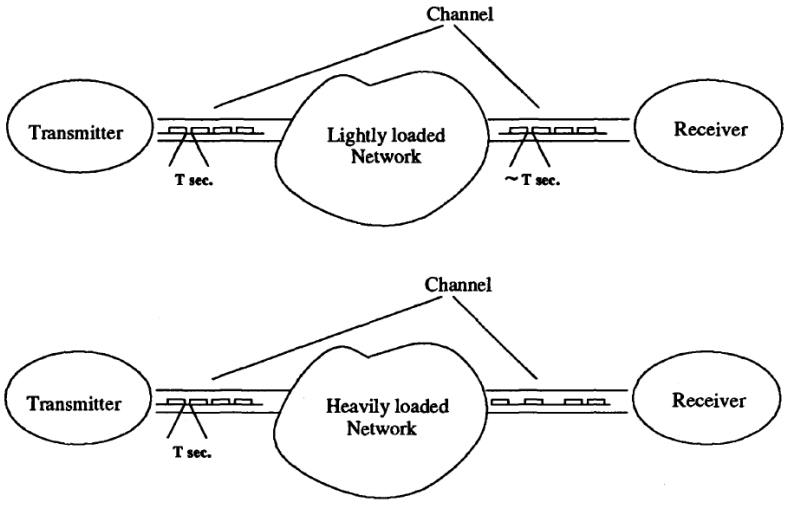

Figure 1: The impact of network congestion on packet intervals. This graph is taken from [2]

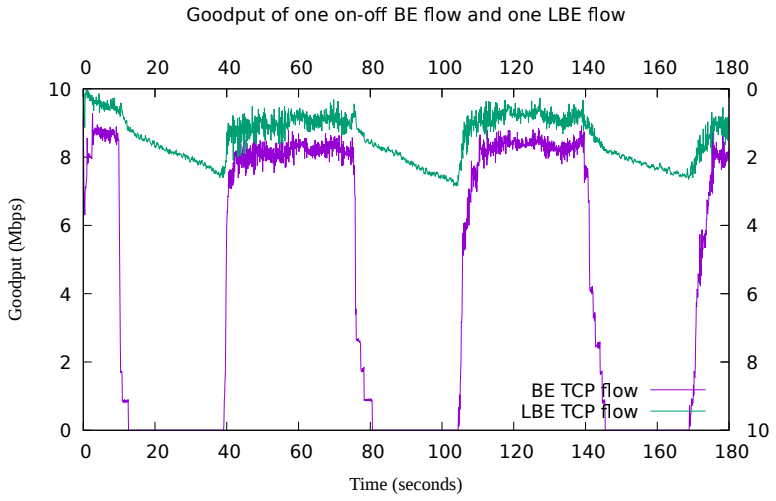

Figure 2: Poor bandwidth utilization by an over cautious LBE flow. Overall bandwidth utilization ratio: 0.21. The LBE flow is plotted upside down

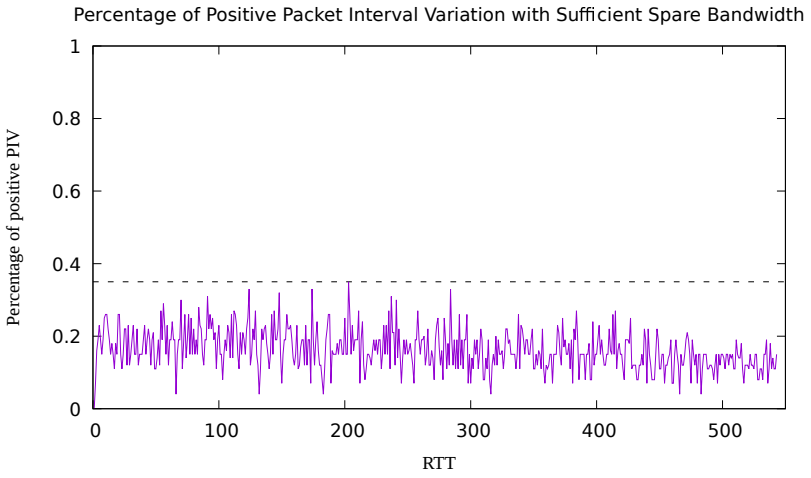

Figure 3: Percentage of positive PIVs with sufficient spare bandwidth. Sending rate: 4Mbps. Spare bandwidth: 7Mbps

because that there is still plenty of spare bandwidth left or because there are no other flows at all, there will be mainly negative PIVs since the packets are compressed together by the queue. If this is true, we can use the number and value of positive PIVs to detect the presence of other flows and if there is still spare bandwidth left.

In order to find out if our hypothesis is true, we conducted a number of experiments. These experiments were ran on a 3-node testbed with 10Mbps bottleneck rate and 100ms RTT. The queuing discipline used at the bottleneck is pfifo. Maximum queue length was set to 86 packets (roughly 1 BDP). We let two BE flows share one bottleneck. One flow sent packets at a constant speed of 3Mbps and the second flow used various sending rates ranging from from 1Mbps to 10Mbps. Reverse traffic was introduced to simulate a real network environment.

Figure 3 shows that positive PIVs only accounts for at maximum 35% of all PIVs measured in one RTT when there are sufficient available bandwidth (sending at 4Mbps when spare bandwidth is 7Mbps). When the LBE sender over utilizes spare bandwidth (sending at 8Mbps when spare bandwidth is 7Mbps), as is shown in Figure 4, the maximum percentage of positive PIVs can reach 73% per RTT. Figure 5 shows the maximum percentage of positive PIVs per RTT under various sending speeds with a 7Mbps spare bandwidth. As we can see, when the sending speed does not exceed the spare bandwidth (7Mbps), the maximum percentage of positive PIVs never exceeds 50%. As soon as the LBE sender sends at a higher speed than the spare bandwidth, more than 50% positive PIVs can be observed in one RTT. These results prove that the number of positive PIVs indeed increases when spare bandwidth is used up.

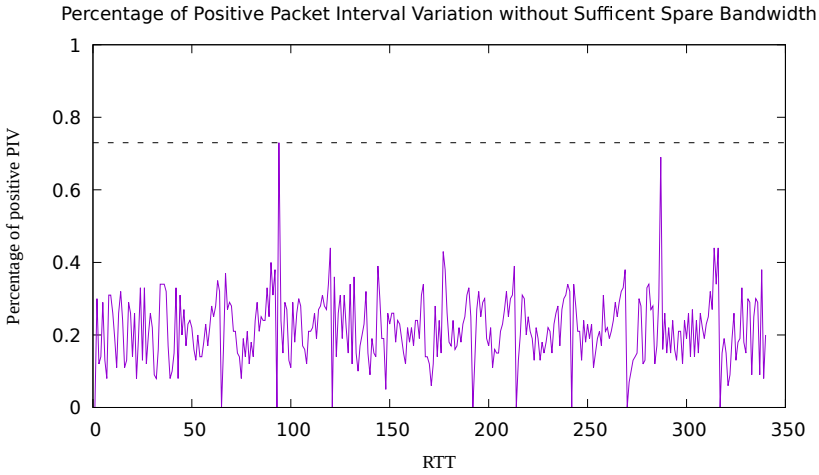

Figure 4: Percentage of positive PIVs without sufficient spare bandwidth. Sending rate 8Mbps. Spare bandwidth: 7Mbps

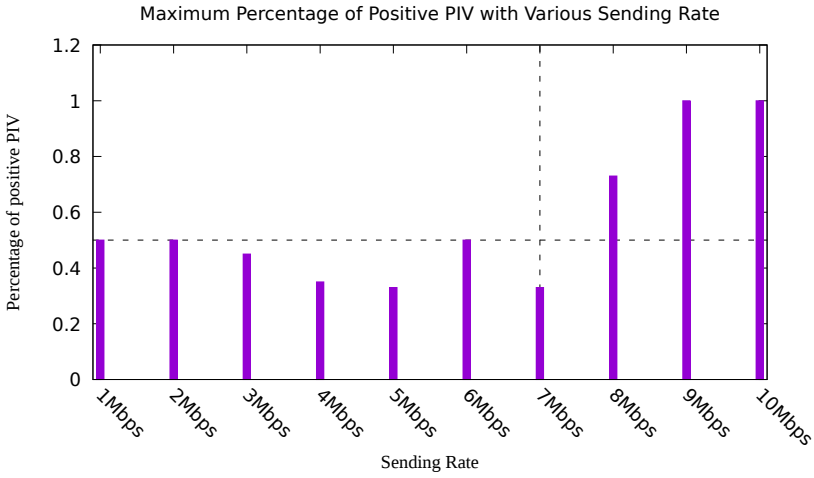

Figure 5: Maximum percentage of positive PIVs with various sending rate. Spare bandwith: 7Mbps

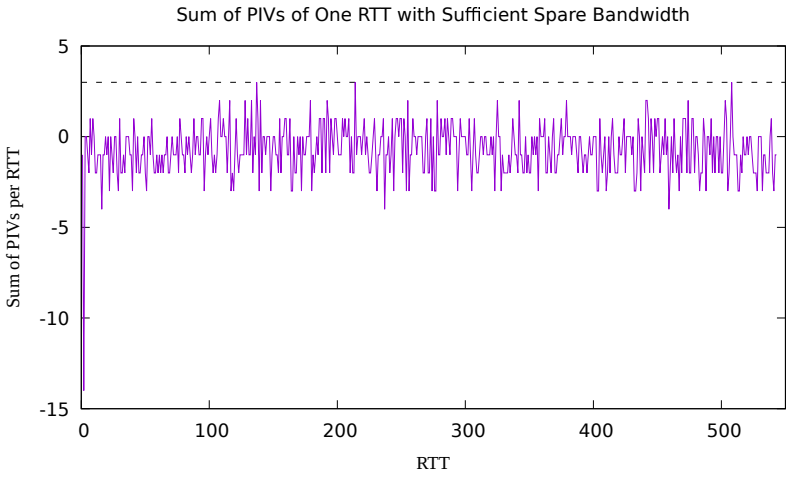

Figure 6: Sum of all PIVs of each RTT with sufficient spare bandwidth. Sending rate: 4Mbps. Spare bandwidth: 7Mbps. Maximum sum: 3

Figure 6 shows the sum of all PIVs of each RTT when the second sender was set to send at 4Mbps. As we can see, the maximum PIV sum of this scenario is 3. In contrast, Figure 7 shows that when the second sender is sending at 8Mbps (greater than the spare bandwidth 7Mbps), the maximum sum of PIVs of each RTT is 41. Figure 8 compares the maximum PIV sums of various sending speeds used by the second flow when spare bandwidth is 7Mbps. It is very easy to discover that using more than the available spare bandwidth can cause significant increase in the value of positive PIVs per RTT. This demonstrates that, when spare bandwidth is over utilized, the receiver can observe more larger positive PIVs per RTT, or more larger PIV sums per RTT.

Figure 9 shows that after including the percentage of positive PIVs and the sum of all PIVs of a RTT as spare bandwidth estimation factors, Vulture's bandwidth utilization ratio is improved from 0.21 to 0.56 – an almost three fold increase.

The network load estimation algorithm proposed by [2] is an application layer solution. It can be too slow for a transport layer protocol like Vulture. The desirable operational period for Vulture is one RTT. That is, Vulture should detect network congestion or bandwidth opportunity and take proper actions within one RTT. In many situations, we cannot get the required number (300) of PIVs within one RTT as proposed by the algorithm. Therefore, we modify the algorithm to use only the PIVs obtained in one RTT. Also, instead of calculating standard deviation and $\epsilon - percentile$, we study PIVs directly. Another modification we have made is that instead of generating probing packets to estimate network load, we directly measure the intervals between the data packets sent by the sender. Algorithm 1 shows how PIVs are calculated.

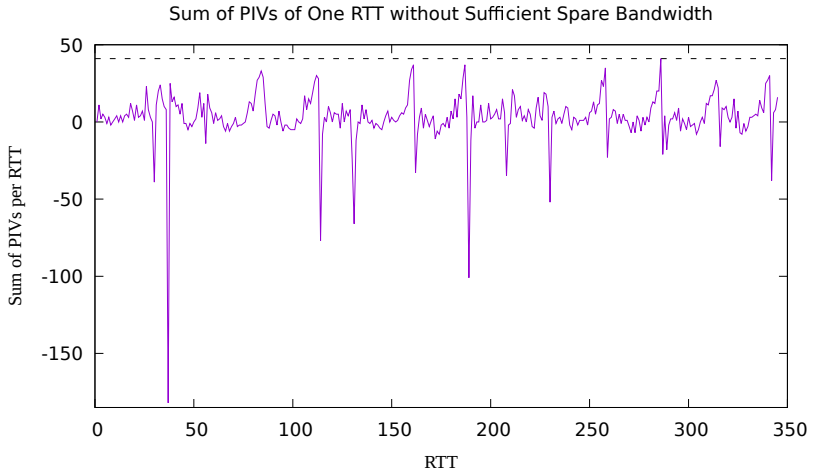

Figure 7: Sum of all PIVs of each RTT without sufficient spare bandwidth. Sending rate: 8Mbps. Spare bandwidth: 7Mbps. Maximum sum: 41

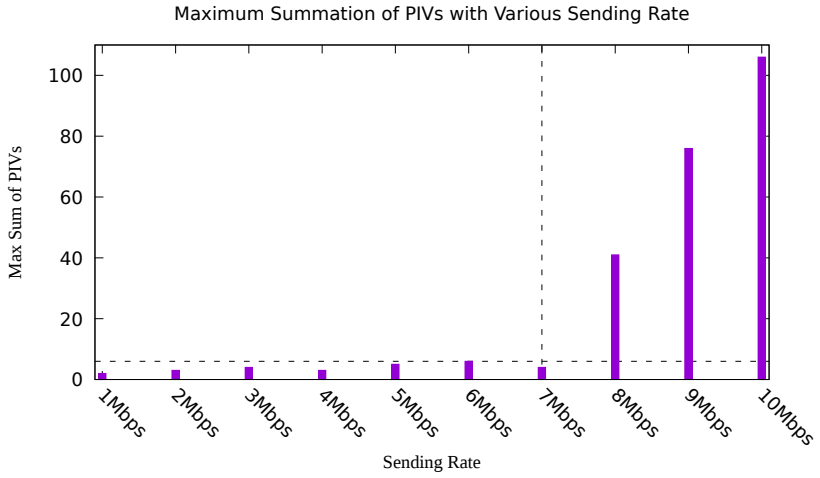

Figure 8: Maximum per RTT PIV sums for various sending rate. Spare bandwith: 7Mbps

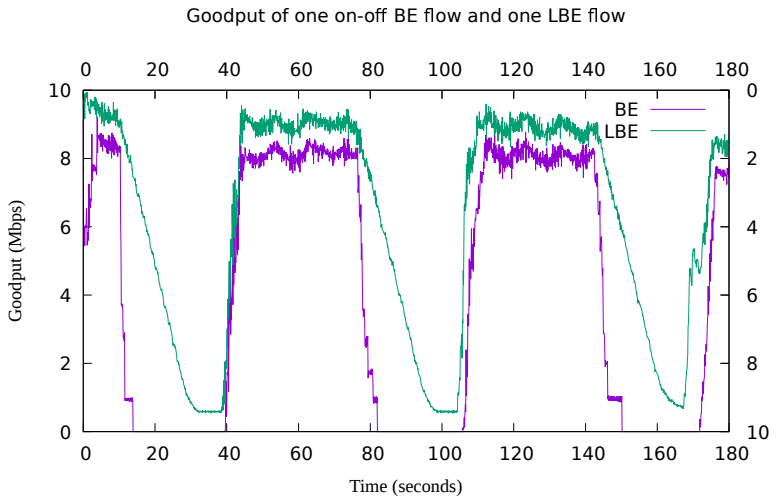

Figure 9: Improved bandwidth utilization after including the percentage and value of positive PIVs as spare bandwidth estimators. Overall bandwidth utilization ratio: 0.56. The LBE flow is plotted upside down

Algorithm 1 Calculating Packet Interval Variations

Variables

psi: packet sending interval, i.e., the difference between the sending time of two data packets

pai: packet arrival interval, i.e., the difference between the arrival time of two data packets

piv: packet interval variation

PSI: a threshold used to filter valid packet pairs. It is set to 5 jiffies in our algorithm.

packetQue: packet queue. Storing information of received data packets

tsval: the sending time of a data packet

tsarv: the arrival time of a data packet

pivQue: the queue used to temporarily store pivs

On the receipt of a data packet: packet

$psi = packet.tsval - packetQue.head.tsval$

if $psi >= PSI$ **then**

  $pai = pkt.tsarv - packetQue.head.tsarv$

  $piv = pai - psi$

  $pivQue.enq(piv)$

  $packetQue.deq(packetQue.head)$

else

  $packetQue.enq(packet)$

end if

PSI is a heuristic value. We discovered that setting PSI to 20 mili-seconds or 5 jiffies can produce the most desirable estimation results. Setting PSI to a very small value may cause the algorithm not be able to distinguish congestion from random noise because the interval variations are too small. On the other hand, a very large PSI value may result in failing of detecting transient queues. Furthermore, the larger the PSI is, the less PIVs we can get per RTT, which adversely affects the accuracy of the spare bandwidth estimation algorithm.

After an RTT is finished, we calculate the percentage of positive PIVs and the sum of all the PIVs of this RTT (see Algorithm 2). If PIV sum is greater than zero, we assume that the spare bandwidth is over utilized, so Vulture enters its conservative mode, in which it is very sensitive to the increase of bottleneck queues. In conservative mode, rwnd is increased very slowly but decreased very quickly. On the other hand, if the sum is less than zero, our algorithm assumes that the spare bandwidth is under utilized, so Vulture enters aggressive mode, in which the receiver can tolerate more queuing delay and increases rwnd more aggressively (see algorithm 3).

Algorithm 2 PIV Analysis

Variables
pivQue: the queue used to temporarily store pivs
ptr: a pointer pointing to the elements of the pivQue
pivSum: the sum of all PIVs in one RTT

On the completion of one RTT
$ptr = pivQue.head$
$pivSum = 0$
while $ptr \neq NULL$ **do**
  if $ptr \rightarrow piv < 0$ **then**
    negative++
  else if $prt \rightarrow piv > 0$ **then**
    positive++
  else
    zero++
  end if
  $pivSum+ = ptr \rightarrow piv$
  $ptr = ptr \rightarrow next$
end while

As we can see, the conservative mode and aggressive mode use different thresholds to classify PIVs into three categories: high, middle, and low. A PIV that is classified as high by the conservative mode can be classified as low by the aggressive mode. Consequently, with the same set of PIVs, the conservative mode algorithm may think the queue is too long and decide to reduce rwnd, but the aggressive mode algorithm may deem this queue is still too short, hence, continue increase rwnd.

Algorithm 3 Deciding on conservative or aggressive modes

Variables
pivCount: how many PIVs are there in last RTT
high: how many PIVs in last RTT are considered too high
middle: how many PIVs in last RTT are considered just fine
low: how many PIVs in last RTT are considered too low
cLow: lower threshold of conservative mode. Set to 1 in our algorithm
cHigh: higher threshold of conservative mode. Set to 2 in our algorithm
aLow: lower threshold of aggressive mode. Set to 3 in our algorithm
aHigh: higher threshold of aggressive mode. Set to 5 in our algorithm

On the completion of one RTT
$ptr = pivQue.head$
if $pivSum > 0 \vee positive > pivCount/2$ **then**
  Enters conservative mode
  while $ptr \neq NULL$ **do**
    if $ptr \rightarrow piv \geq cHight \vee ptr \rightarrow piv \leq -cHigh$ **then**
      high++
    else if $ptr \rightarrow piv == -cLow \vee ptr \rightarrow piv == cLow$ **then**
      middle++
    else
      low++
    end if
    $ptr = ptr \rightarrow next$
  end while
else
  Enters aggressive mode
  while $ptr \neq NULL$ **do**
    if $ptr \rightarrow piv \geq aHight \vee ptr \rightarrow piv \leq -aHigh$ **then**
      high++
    else if $ptr \rightarrow piv \geq -aLowandptr \rightarrow piv \leq aLow$ **then**
      low++
    else
      middle++
    end if
    $ptr = ptr \rightarrow next$
  end while
end if

0.3.3 Receive Window Tuning Policy

We use policies to control how to increase and decrease rwnd. The policy making algorithm is pretty straightforward. If the number of PIVs in the high category accounts for α or more of the total number of PIVs obtained in last RTT, then we think the spare bandwidth is over-utilized. We set rwnd moderation policy to "halve" so that the rwnd will be reduced by half during the next RTT. If α or more of the total number of PIVs are in the low group, the spare bandwidth is deemed to be under-utilized. Then, we set rwnd moderation policy to either "double" or "increase". Increase means rwnd is only increased by one Maximum Segment Size (MSS) per RTT. This policy is chosen over "double" when current rwnd value is higher than rwndThresh. RwndThresh is the threshold value used to memorize the maximum rwnd before last rwnd deduction. RwndThresh estimates the current residual bandwidth of the path. Therefore, when rwnd reaches this value, the receiver should increase rwnd very prudently to avoid any intrusion on BE TCP flows. When rwnd is less than rwndThresh, the "double" policy is chosen, which means that the rwnd will be doubled during the next RTT. Rwnd moderation policy is set to "fix" for all other cases. "Fix" implies that rwnd is not changed until a new moderation policy is made. α is set to 7/8 in our algorithm. Our rwnd moderation policy making algorithm is depicted in Algorithm 4.

After updating rwnd tuning policy, Vulture stops analyzing PIVs until it receives the first data packet that reflects the new policy. This can prevent the receiver from over-increasing/decreasing rwnd caused by analyzing out-of-data information.

0.3.4 Policy Implementation

Before sending out an acknowledgement, TCP queries Vulture for current rwnd value. Upon the receipt of the query, Vulture updates the corresponding flow's rwnd by implementing the current rwnd tuning policy. That is, if the current policy is double (or halve), Vulture increases (or decreases) rwnd by the amount of bytes the outgoing ACK acknowledges. Consequently, at the end of the current RTT, rwnd will be doubled (or halved). We control how many bytes rwnd can be increased with each ACK to avoid that a sudden dramatic increase of rwnd can drastically increase the sender's sending rate, hence cause long delay or packet loss along the path. If the policy is "increase", Vulture increases rwnd by one MSS per RTT. And if "fix" is the current policy, Vulture will keep rwnd unchanged until the policy is changed. Algorithm 5 gives more detailed information on how the policy is implemented.

0.3.5 Congestion Window Estimation

An application program can decide when to start and stop Vulture's LBE service. At the time an application starts LBE, the rwnd maintained by the current BE TCP can be very large. If the network at the time is congested and Vulture

Algorithm 4 Rwnd Moderation Policy Making

Variables
pivCount: how many PIVs are there in last RTT
α: a fractional number. In our algorithm it is set to 7/8
rwndThresh: remembers the maximum rwnd value before rwnd is reduced last time

On the completion of one RTT
if $high > \alpha \times pivCount$ **then**
    the network is congested
    $policy = halve$
else if $low > \alpha \times pivCount$ **then**
    the network is under utilized
    if $rwnd < rwndThresh$ **then**
        $policy = double$
    else
        $policy = increase$
    end if
else
    the network is at its optimal state
    $policy = fix$
end if
On the detection of out of order or lost packets
$policy = halve$

Algorithm 5 Vulture Rwnd Tuning Policy Implementation Algorithm

Before sending out an ACK
if $policy == double$ **then**
    $rwnd+ = acknowledgedBytes$
else if $policy == increase$ **then**
    if rwnd has not been increased in this RTT **then**
        $rwnd+ = MSS$
    end if
else if $policy == halve$ **then**
    $rwnd- = acknowledgedBytes$
else
    leave rwnd unchanged
end if

uses this rwnd as its rwnd's initial value, even if with a policy of halve, it takes Vulture about two RTT's time to reduce rwnd to a desired value. This means that, there are around two RTT's time during which other BE TCP flows are adversely affected by the Vulture LBE flow. This result contradicts Vulture's low impact design goal. In order for Vulture to exhibit LBE behavior as soon as it is started, we need to assign its rwnd a proper initial value.

A value that is equal to or a little bit less than the current cwnd can be a good initial rwnd value, because cwnd estimates the fair share bandwidth of a flow. Therefore, Vulture estimates cwnd at the receiver side. To do so, a Vulture receiver maintains an ack queue for each flow. The tsval (time of sending out the ack) and acknowledged bytes of each ack are stored in the queue in the order of when the acks are sent out. When a data packet arrives, the receiver compares the tsecr (echoed timestamp) carried by this data packet with the tsval of the first ack in the queue. If they are equal, the data packet is probably triggered by this ack. Since TCP uses a coarse grained timestamp, there are chances that multiple acks carry the same tsval. If so, data packets triggered by these acks will all carry the same tsecr. It is very difficult to judge which data packets are triggered by which ack in this situation. Vulture employs a very simple solution. The receiver deducts the length of the data packet newly received from the acknowledged bytes field of the first ack in the queue. If the updated acknowledged bytes becomes zero, the receiver assumes that all data packet triggered by this ack have been received, then it removes the first ack from the ack queue. This may result in temporarily under-estimate of cwnd, but the cwnd estimate can catch up after all the acks that carry the same tsval are removed from the queue. At a specific time, the cwnd estimate can be smaller than the real value, but this can be a favorable estimation error, because by setting rwnd to a value that is less than cwnd, the LBE flow is forced to use bandwidth that is smaller than its fair share, which is in accordance with LBE behavior. Algorithm 6 provides the pseudo code of this algorithm.

As we can see, the algorithm actually estimates how many bytes the receiver receives per RTT. Which implies that when $rwnd < cwnd$, the algorithm actually estimates rwnd, and that when there is packet loss in one RTT, the estimated cwnd will be less than the real cwnd of that RTT because the algorithm does not count lost bytes. It can be very difficult for the receiver to estimate how many bytes are lost during one RTT because the receiver does not know what congestion control mechanism is employed at the sender side and whether the sender is in slow start or congestion avoidance in that RTT. Therefore, the receiver simply ignores the lost bytes. Similarly, this estimation error is acceptable, because assigning rwnd an initial value that is less than cwnd can make the flow exhibit LBE behavior at the start.

0.4 Evaluation

In this section, we present the evaluation results of Vulture. We evaluated Vulture against its design goals. The results are presented in separate subsections.

Algorithm 6 Vulture Cwnd Estimation Algorithm

Variables
cwndEstimate: the estimation of cwnd
ackQue: the queue used to store information of each outgoing ACK
tsval: a timestamp storing the time when a packet is sent
tsecr: a timestamp storing the tsval of an echoed packet
packet.len: length of a data packet
ack.len: how many bytes are acknowledged by the ACK

Upon the receipt of a data packet: packet
while $TRUE$ **do**
  $ack \Leftarrow ackQue.head$
  if $pkt.tsecr < ack.tsval$ **then**
    $cwndEstimate+ = packet.len$
    break
  else if $packet.tsecr == ack.tsval$ **then**
    $ack.len- = packet.len$
    if $ack.len == 0$ **then**
      $ackQue.deq(head)$
    else
      $ackQue.head.len = ack.len$
    end if
    break
  else
    $cwndEstimate- = ack.len$
    $ackQue.deq(head)$
  end if
end while

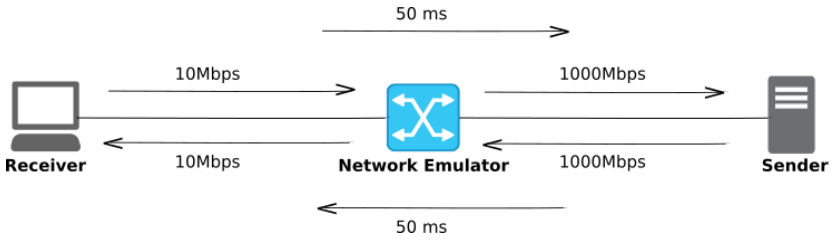

Figure 10: The Topology of the Testbed

0.4.1 Experiment Setup

To generate some preliminary evaluation results, we built a simple three-node testbed (shown in Figure 10). A laptop computer was used as both a client and a receiver. Another laptop was used as a server and a sender. A desktop computer was connected between the sender and receiver as a network emulator. All three computers were running Ubuntu Linux 16.04, kernel version 4.12.8.

The network card of the receiver was configured (using ethtool) to work in the 10Mbps full-duplex mode. The actual achievable goodput however is only approximately 9.41Mbps. In the following subsections, we will use 9.41Mbps as maximum available bandwidth (measured in goodput) and all bandwidth utilization ratios are calculated using this number.

The network emulator was a normal Linux machine with IP-forwarding enabled. It was configured to route packets between the client and server. The client side network card was configured (using ethtool) to work in the 10Mbps full-duplex mode. A pfifo queue with a queue length of 86 packets was added to this card. The server side network card was set to work in the 1000Mbps full-duplex mode. This way, we could create a bottleneck between the receiver and the emulated network. Two 50-millisecond artificial delays were added (using tc) on both directions so that the round trip time between the sender and the receiver was about 100 milliseconds.

The sender's network card worked in the 1000Mbps full-duplex mode. No modifications were made to the sender's operating system. And no special software were install on it except some application programs.

Vulture's kernel module was installed on the receiver. We also modified Linux kernel core on the receiver so that TCP could send required information of each received data packets to Vulture's kernel module and query Vulture for rwnd before it sent out acknowledgements. A user-space Vulture library was also installed on the receiver so that the receiver side application could start and stop LBE mode as necessary.

Segmentation offloading was disabled on both sender and receiver. Send and receive buffers on the two end hosts were set large enough so that they would not become rate limiting factors. TCP's flow caching was disabled on two end machines by setting net.ipv4.tcp_no_metrics_save to 1.

In all experiments conducted, we measured application layer throughput,

namely, goodput. Therefore, the maximum measured goodput can be smaller than the testbed setup, 10Mbps, even if the sender uses a BE congestion control and sends at its full speed. Furthermore, we measure all goodput at the receiver side, so that the measured values do not account for lost packets.

If not otherwise noted, all results presented in this section are the average of 10 runs of exactly the same experiment to eliminate noise.

0.4.2 Cwnd Estimation

In this subsection, we examine the performance of Vulture's cwnd estimation algorithm. The goal is to see if the estimated cwnd values are consistent with the real cwnd values maintained by the sender. By consistent, we mean that when cwnd is increased (or decreased), the estimate should also be increased (or decreased) and the degree of change should be equal. To do so, we let the sender send a large file at its full speed to the receiver for 3 minutes. During the first 45 seconds, the flow was running in BE mode. After this period, the receiver side application started LBE, which made the flow running in LBE mode. After 90 seconds, the receiver side application stopped LBE, and the flow resumed BE mode transport. During the experiments, no cross or reverse direction flows was started. Figure 11 shows cwnd, cwnd estimate, and rwnd over the three periods with Cubic as congestion control. As we can see from the figure, our cwnd estimates can keep up the change of the sender side cwnd although a little bit less than the real values. As we explained in subsection 0.3.5, this underestimation is caused by the fact that our cwnd estimation algorithm does not count lost bytes and this estimation error is acceptable. We can also observe that the algorithm actually estimates rwnd when the flow is running in LBE mode.

In the above experiments, the congestion control algorithm was set to Cubic, next we examine how the cwnd estimation algorithm performs when the sender uses another congestion control. We chose new reno because it is the congestion control used by the standard TCP. We reran the above described experiments but changed Cubic to new reno. The results are illustrated in Figure 12. Our experiment results show that Vulture's cwnd estimation algorithm is independent of congestion controls used by the sender and can achieve equivalent performance.

Finally, we demonstrate that setting the initial rwnd to cwnd estimate can make a flow quickly yield its bandwidth share as soon as it is set into LBE mode. To make this observation, we conducted a set of new experiments, in which two forward direction flows sending data from the server to the client and one reverse direction flow sending data from the client to the server. The reverse direction flow was introduced to add some random queuing delay on the ack path to simulate the real Internet environment. The reverse direction flow was running in BE mode only for the whole experiment. One of the forward direction flows ran in BE mode for the whole experiment (3 minutes) with a constant sending rate of 6Mbps. During the first 45 seconds of the experiment, another forward direction flow was running in BE mode also, then was set to

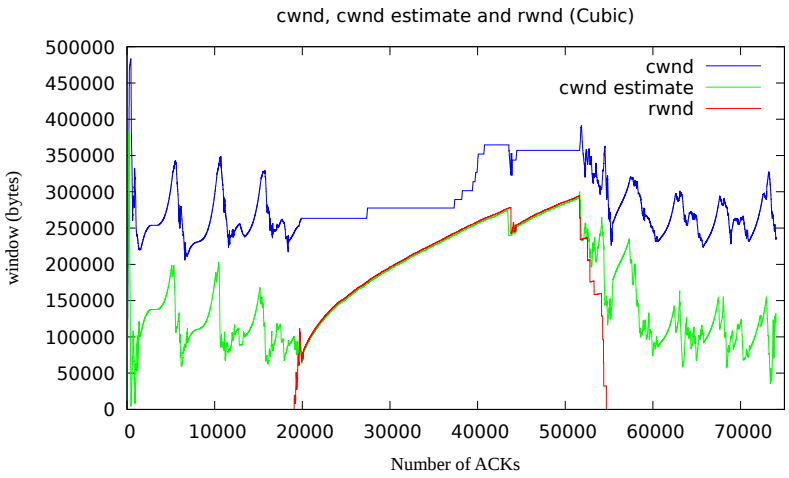

Figure 11: Cwnd estimate for Cubic

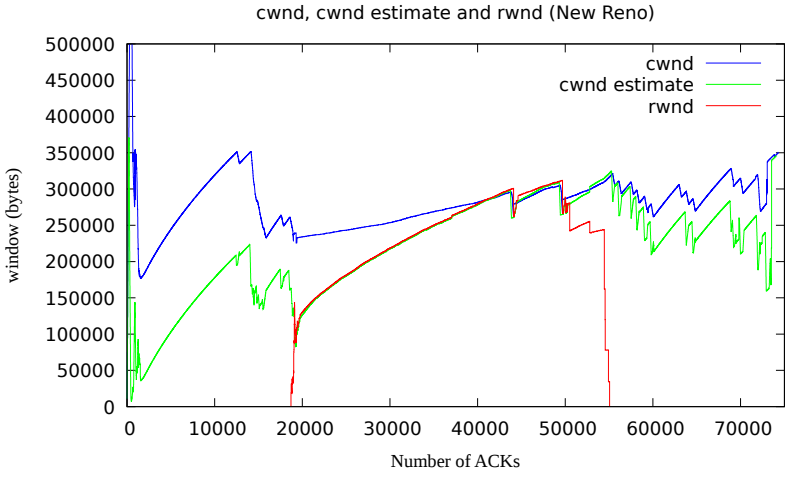

Figure 12: Cwnd estimate for New Reno

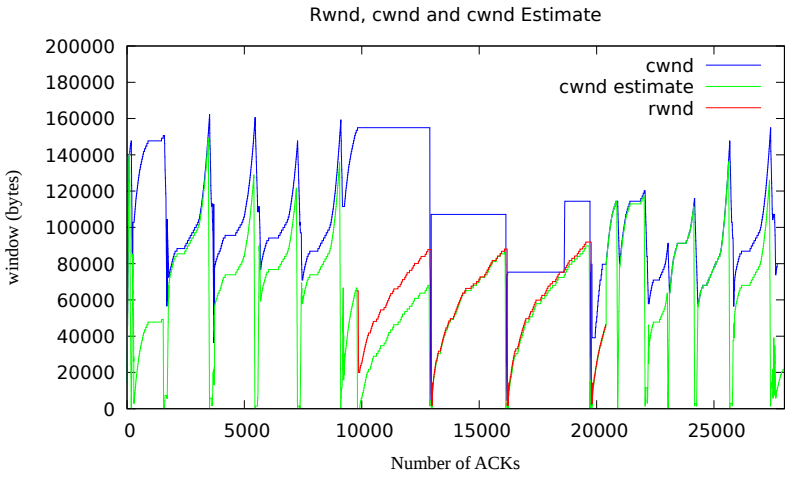

Figure 13: Rwnd is set to cwnd estimation when LBE is started

LBE mode for 90 seconds, and finally was set back to BE mode for the last 45 seconds. Figure 13 plots the cwnd, cwnd estimate, and rwnd of the second forward direction flow from one run only (for presentation clarity purpose). As we can see, when the second forward direction flow is just set into LBE mode (at time 45 seconds or ack number 9795), its rwnd is set to cwnd estimate of that time, which is smaller than the cwnd maintained by the sender. This makes the sender immediately reduce its sending speed to a value that is lower than its fair share.

From Figure 14 we can see the second forward direction flow immediately yields its goodput to the other flow as soon as it enters LBE mode. The two dashed vertical lines denote the start and stop of the LBE mode of the second forward direction flow. The dashed horizontal lines denote the desired bandwidth division between the two forward direction flows during each period. During the first 45 seconds, both flows are in BE mode, so they fairly share the total bandwidth 10Mbps. During the next 90 seconds, the first forward direction flow (labeled as BE TCP flow in the figure) still runs in BE mode, but the second one (labeled as LBE TCP flow) enters LBE mode. Due to the fact that rwnd is set to cwnd estimate as soon as the flow enters LBE mode, the LBE flow can immediately yield its bandwidth to the BE flow. During the second period, the BE flow can send at, on average, 5.58Mbps, which is 93% of its full speed – 6Mbps. In the meanwhile, the LBE flow can make use of 83% of the spare bandwidth achieving an average goodput of 2.82Mbps. In the last 45 seconds, the second forward direction flow resumes BE mode, so it regains its bandwidth fair share.

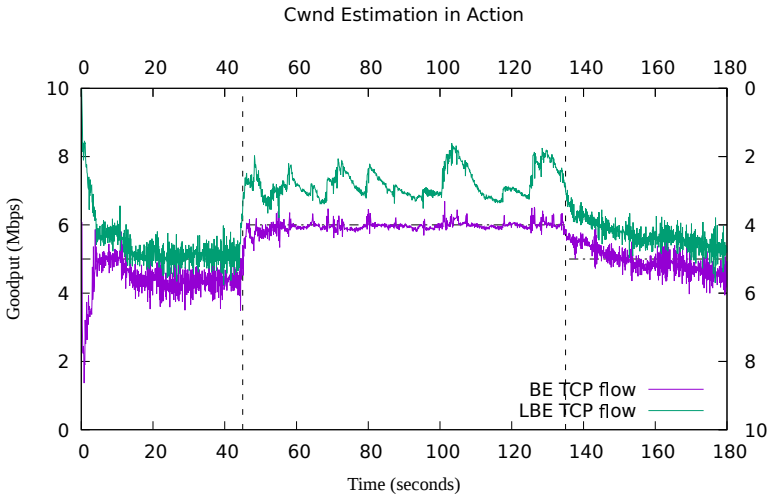

Figure 14: Cwnd estimation in effect. Two flows are plotted head-to-head with the LBE flow upside-down

0.4.3 Impact on BE Flows

One of the main design goals of Vulture is to inflict as little impact as possible on BE flows with which it shares the same bottleneck. In this subsection, we study the throughput and round trip time impact imposed by an LBE flow on BE flows. The experiments were conducted as follows. One forward direction BE flow was sending data from the server to the client at a constant rate of 6Mbps (due to application design reasons, 6Mbps was the average rate, sometimes, we could observe rates higher than 6Mbps, but the maximum rate never exceeded 7Mbps) for the whole experiment duration (3 minutes). Another BE flow was sending data in the reverse direction from the client to the server with the same sending rate to introduce random queuing delay on the ack path. A second forward direction LBE flow was started at time 45 seconds and lasted for 90 seconds. Figure 15 shows the goodput of the forward direction BE flow and LBE flow. The two curves are plotted head to head so that we can clearly see their respective shares of the total 10Mbps bandwidth. The two dashed vertical lines denote the start and the stop of the LBE flow. As is illustrated by the figure, during the first 3 seconds of the start of the LBE flow, the BE flow experiences mild goodput decrease (about 1Mbps). After the initial 3 seconds, the BE flow regains its original goodput but with greater oscillation. The left of the LBE flow does not impose adverse impact on the BE flow. Overall, the average goodput the LBE flow achieves during its active period is 2.84Mbps. The average goodput achieved by the BE flow in the three periods are 5.97Mbps, 5.98Mbps, and 5.97Mbps, respectively. Therefore, the bandwidth utilization ratio of the LBE flow can be calculated by $R = 2.84/(9.41 - 5.98) = 0.83$. As we can see, overall the existence of the LBE flow does not make the BE flow

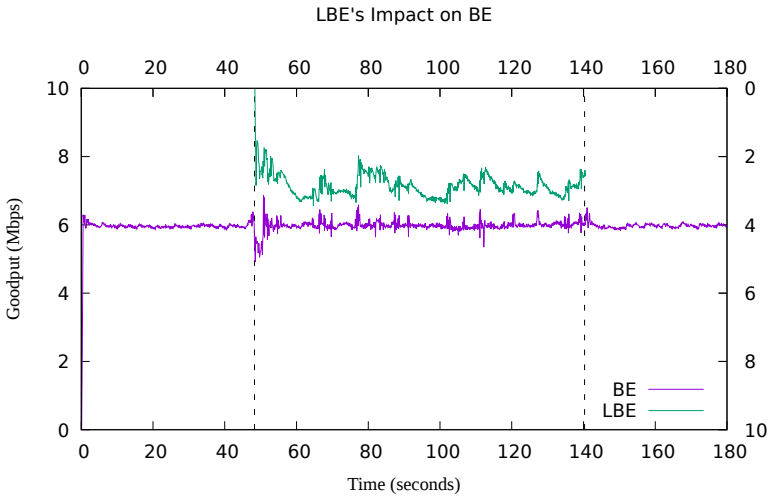

Figure 15: The LBE flow has little impact on the BE flow except the first 3 seconds of its start. Average goodput of the LBE flow: 2.84Mbps. Average goodput of the BE flow during three periods: 5.97Mbps, 5.98Mbps, 5.97Mbps

lose its goodput.

Using the same experiments, we also measured the RTT values experienced by the forward direction BE flow. The average of RTTs from 10 runs are shown in Figure 16. During the first 45 seconds, the BE flow is running alone in the forward direction. It experiences on average 104ms round trip delay in this period. During the next 90 seconds when the LBE flow is also running, the average RTT the BE flow experiences is 146ms. And in the last 45 seconds, the LBE flow is stopped and the BE flow is the only flow passing through the bottleneck, so the average RTT of this period is 104ms. Overall, the LBE flow adds 40% delay to the BE flow.

0.4.4 Bandwidth Utilization

We have examined the bandwidth utilization ratio of Vulture under the scenario where there are two flows, one is a BE flow and another is an LBE flow. In this subsection, we take a look at more scenarios.

In the first scenario, there is only one flow. The server sends data at its full speed to the client for 3 minutes. During the first 45 seconds, the flow is running in BE mode, and during the next 90 seconds LBE mode, in the last 45 seconds back to BE mode. From Figure 17 we can see that the flow experiences a goodput drop after it is set into LBE mode. It regains its goodput after about 20 seconds. The bandwidth utilization ratio after the flow stabilizes in the LBE mode is $9.34/9.41 = 0.99$, where 9.34 is the average goodput achieved during this period of time, 9.41 is the maximum achievable goodput along the route. The bandwidth utilization ratio for the whole LBE mode period is $8.76/9.41 = 0.93$.

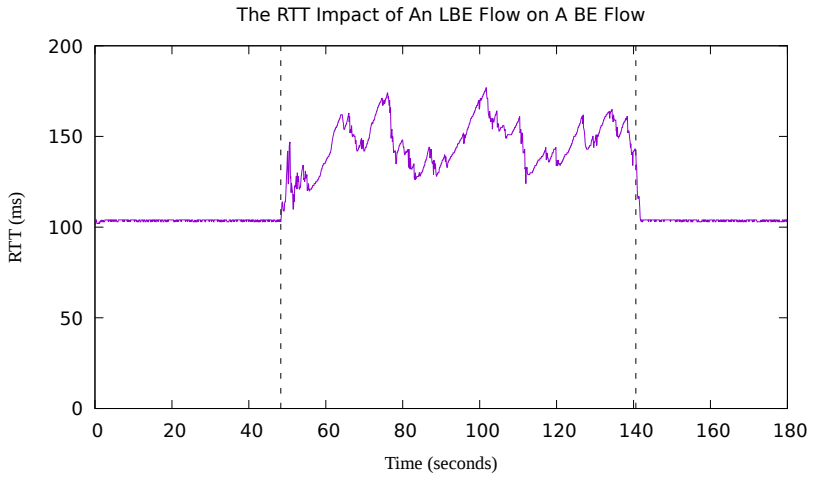

Figure 16: The RTT experienced by the forward direction BE flow. The average RTT of the first period: 104ms, the second period: 146ms, the third period: 104ms. Overall, the LBE flows adds 40% delay to the BE flow

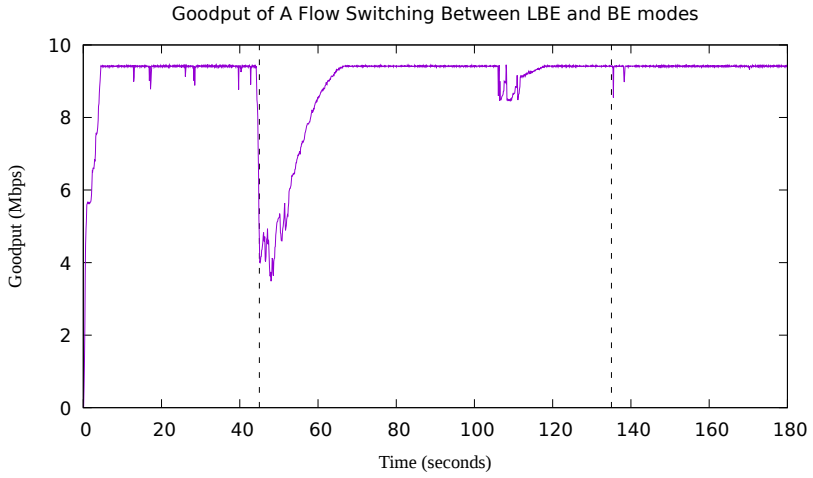

Figure 17: The bandwidth utilization ratio for the whole LBE period is 93%, and for the stabilized LBE period is 99%

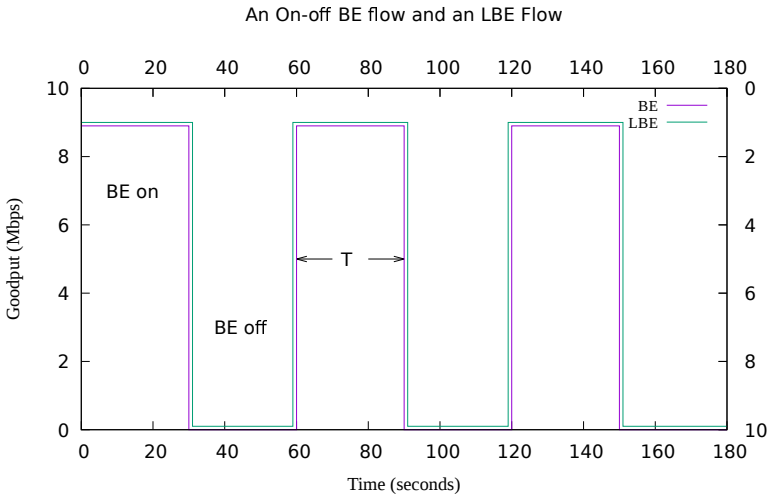

Figure 18: Interaction between an on-off BE flow and an LBE flow. An ideal situation. The LBE flow is plotted upside down

In the second scenario, we use two forward direction flows and one reverse direction flow. Again, the reverse direction flow is used to introduce some random queuing delay on the ack path. One forward direction flow is an on-off BE flow. That is, the sender sends data at its full speed for a period of time, and then stops for a while, and resumes sending. This iterates until the end of the experiment. The duration of on and off periods are set basically the same. Another forward direction flow works in LBE mode from the start to the end and seeks to utilize spare bandwidth left by the on-off BE flow. It is worth noting that the "on" and "off" in this context are only used to specify the BE flow's status. They can not be applied to the LBE flow, which has potentially unlimited bandwidth requirement but can only utilize spare bandwidth left by the BE flow. Figure 18 illustrates how these two forward direction flows interact with each other in an ideal situation. In this ideal scenario, the on/off duration T is exactly 30 seconds. When the BE flow is on, the LBE flow only uses a small portion of the total bandwidth. When the BE flow is off, the LBE flow utilizes almost all the bandwidth. Ideally, The LBE flow can quickly react to the dramatic change of the BE flow from on to off and vice versa. In the figure, the LBE flow is plotted upside down so that we can easily see that the two curves complement each other and the bandwidth is fully utilized by the two flows.

The goal of this test is to see if our LBE protocol can behave like the ideal one. Our focus, in this subsection, is bandwidth utilization. We varied the on/off duration T from 1 second, through 3 seconds, 5 seconds, 10 seconds, to 30 seconds. The measured goodput curve of a BE flow can be a little bit different from the idealized square waves depicted in Figure 18. For example, the

off duration of a real BE flow is less than T seconds and on duration is more than T seconds for most of the times. For each T-duration test, we calculated the average goodput of the BE and LBE flows for each on (or off) period. Then, we averaged all the average goodput of each on (or off) period of the BE and LBE flows. Finally, for all T-duration tests, we obtained one average goodput as the on (or off) period goodput for the BE and LBE flows. When the BE flow is off, the LBE flow is the only user of the bandwidth. Then, its bandwidth utilization ratio can be calculated by $u = G/B$, where G is the final average goodput of the LBE flow during the off period of the BE flow, and B is the total available bandwidth. When the BE flow is on, the bandwidth is divided between the BE and LBE flows. The bandwidth share of the LBE flow is $S_l = G_l/(G_l + G_b)$, and the bandwidth share of the BE flow is $S_b = G_b/(G_l + G_b)$, where G_l is the average goodput of the LBE flow when the BE flow is on, and G_b is the average goodput of the BE flow when the BE flow is turned on. Table 1 presents all the bandwidth utilization ratios and bandwidth shares of all T-duration tests.

Table 1: Bandwidth utilization ratio and bandwidth share

Duration	**BE Off Period**	**BE On Period**	
	LBE BW Util.	**BE BW Share**	**LBE BW Share**
1 sec	0.15	0.89	0.11
3 sec	0.20	0.88	0.12
5 sec	0.25	0.86	0.14
10 sec	0.33	0.86	0.14
30 sec	0.56	0.77	0.23

Figure 19 visualizes these numbers using a bar graph. On the X-axis are the 5 different durations. The Y-axis indicates the summed goodput of the LBE and BE flows of the on (the bar on the right) or off (the bar on the left) period. The average goodput of the LBE flow is plotted above that of the BE flow for the BE on periods. For BE off periods (the bars on the left), only the goodput of the LBE flow is plotted, because the goodput of the BE flow is zero in these periods. The dashed horizontal line in the figure marks the maximum achievable goodput along the route.

As we can observe, the LBE flow's bandwidth utilization improves with the growth of the on/off duration – T. When the BE flow off period is close to 30 seconds, the LBE flow can achieve a 56% overall bandwidth utilization. While when the BE flow off period is less than 1 second, the bandwidth utilization is reduced to only 15% because the LBE flow needs about 20 seconds to find the maximum spare bandwidth. When the BE flow is on, the LBE flow takes a small percent of the total bandwidth (11% – 23%) from the BE flow to fulfill its transport tasks.

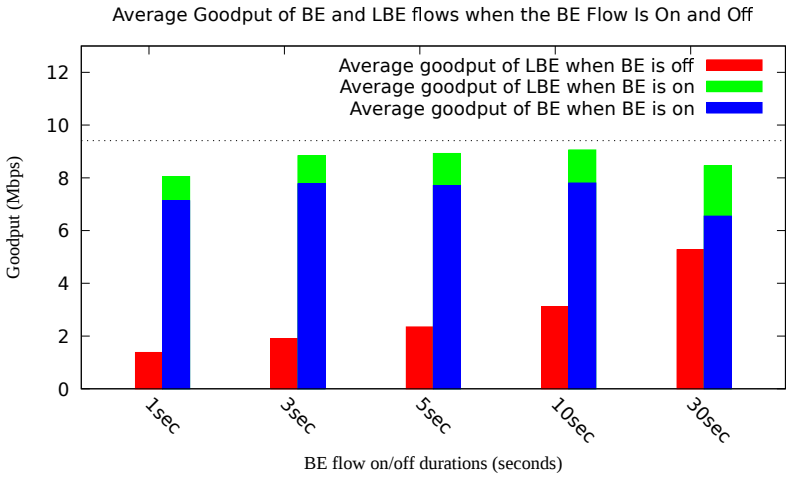

Figure 19: Total goodput achieved by the BE and LBE flows when the BE flow is turned on or off

0.4.5 Swiftness

In this subsection, we focus on how quickly Vulture reacts to bandwidth opportunity changes. This can be examined from four aspects. (1) swiftness in terms of switching from BE mode to LBE mode. (2) swiftness regarding switching from LBE mode to BE mode. (3) How much time it takes for an LBE flow to reap bandwidth newly released by other flows. (4) How much time it takes for an LBE flow to yield bandwidth when other BE flows increase their bandwidth consumption.

As we revealed in previous sections, when a flow is switched from BE mode to LBE mode, it can instantaneously yield its bandwidth and start to use spare bandwidth only (Figure 14 and Figure 17). The time taken to yield the extra bandwidth is almost zero, but the time it takes for the LBE flow to find the maximum available spare bandwidth is in proportion to how much spare bandwidth is left to be used.

We have seen smooth and seamless switch from LBE mode to BE mode. The flow can quickly gain its fair share after entering BE mode (Figure 14) or maintain its original bandwidth if it is already sending at the maximum speed (Figure 17). The switching from LBE to BE mode does not cause system turbulence such as drastic throughput/RTT oscillation.

Vulture can immediately detect extra spare bandwidth released by other flows. Since we control the rwnd growth speed so that the rate increase of the LBE flow will not cause long queuing delay or large amount of packet loss, it may take the LBE flow more time to achieve full bandwidth utilization. As shown in Figure 9, it takes the LBE flow about 20 seconds to discovery a 10Mbps extra spare bandwidth.

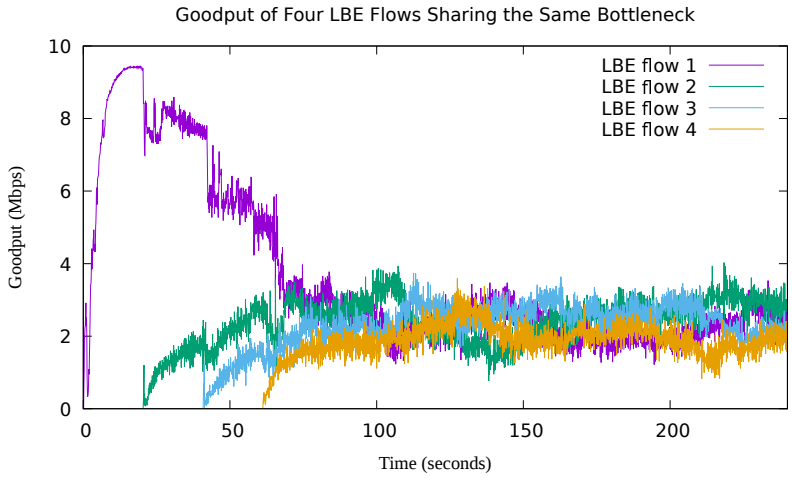

Figure 20: Goodput of four LBE flows. Using 20-second staggered start

From Figure 9 we can also see that the LBE flow yields bandwidth at the same time as the BE flow gains bandwidth. The BE flow can obtain almost all the bandwidth it needs within a few seconds. This demonstrates that the LBE flow can quickly yield its bandwidth to BE flows.

0.4.6 Fairness

Finally, we examine fairness among LBE flows. We do not expect the total bandwidth is fairly shared between an LBE flow and a BE flow, because an LBE flow is designed to only use the spare bandwidth left by BE flows, but we do expect that all LBE flows can fairly share the spare bandwidth left by BE flows.

In this experiment, we let different number of LBE flows compete for the 10Mbps bottleneck bandwidth. No reverse direction flows and BE flows were introduced. The LBE flows were started 20 seconds one after another. The experiments lasted for 3 minutes after all LBE flows were started.

Figure 20 shows that four LBE flows can fairly share the available bandwidth.

Table 2 shows Jain's fairness index for 2, 4, 8, 16, and 64 LBE flows. Jain's fairness index is calculated as follows: $J(x_1, x_2, ..., x_n) = (\sum x_i)^2/n \times \sum(x_i)^2$, where n is the number of competing flows, x_i is the average goodput achieved by the ith flow. A fairness index that is close to 1 means high fairness. As we can see from the table, Vulture LBE flows exhibit high fairness toward each other.

Table 2: Jain's Fairness Index of Various Number of Competing LBE Flows

2 Flows	**4 Flows**	**8 Flows**	**16 Flows**	**64 Flows**
0.96	0.99	0.98	0.99	0.99

0.5 Conclusion and Future Work

Less than Best Effort (LBE) transports are a new type of data transport service. LBE protocols only scavenge spare bandwidth left by best effort transports. It has many potential applications and can contribute to flow prioritization in the Internet.

Vulture is a receiver-side flow-control-based LBE congestion control mechanism designed by us. It has four building blocks. Firstly, it employs a packet interval variation based technique to infer the degree of network congestion and the existence of other flows. Secondly, it uses a policy-based mechanism to tune the receiver's window. Thirdly, the receiver implements the receive window policy in an incremental way. Finally, the receiver estimates the sender's congestion window and set initial rwnd to this estimate. Vulture works in two modes. When there is plenty of bandwidth, it works in aggressive mode, otherwise, in conservative mode.

Vulture has five design goals: (1) Correctly estimate spare bandwidth (2) Minimize its impact on BE TCP flows (3) Maximize throughput utilization (4) Quickly yield and reap bandwidth, and (5) Realize intra-flow fairness.

We have evaluated Vulture against its design goals with a three-node testbed. Our experiment results show that Vulture incurs 0% throughput degradation and 40% extra delay on BE flows that share the same bottleneck with it. Overall bandwidth utilization is above 83% when the LBE flow shares one bottleneck with a constant bit rate BE flow. When the LBE flow shares a bottleneck with an on-off BE flow, its bandwidth utilization ratio decreases with the decrease of the on/off interval. Our LBE flow can quickly yield its bandwidth on the occurrence of a BE flow. And the LBE flows are fair toward each other. Jain's fairness index for 4, 16, and 64 flows is 0.99, 8 flows 0.98, and 2 flows 0.96.

We are planning to test Vulture with real Internet applications and we hope by prioritizing flows with Vulture LBE, the overall performance of applications can be improved. In order to achieve our goals, Vulture may subject to further improvement or enhancement.

Bibliography

[1] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control. RFC5681, September 2009. Available: http://www.rfc-editor.org/rfc/rfc5681.txt. Last access: 2017-11-27.

[2] F. Davoli, O. Khan, and P. Maryni. Estimating the available bandwidth for real-time traffic over best effort networks. *Protocols for High-Speed Networks V*, 1997.

[3] Mike Fisk and Wu chun Feng. Dynamic right-sizing: Tcp flow-control adaptation. In *The 14th Annual ACM/IEEE SC2001 Conference*, 2001.

[4] David A. Hayes, David Ros, Andreas Petlund, and Iffat Ahmed. A framework for less than best effort congestion control with soft deadlines. In *IFIP Networking*, 2017.

[5] Aleksandar Kuzmanovic and Edward W. Knightly. Tcp-lp: Low-priority service via end-point congestion control. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 2006.

[6] S. Liu, M. Vojnović, and D. Gunawardena. Competitive and considerate congestion control for bulk data transfers. In *IEEE IWQoS*, 2007.

[7] P. Mehra, A. Zakhor, and C. De Vleeschouwer. Receiver-driven band-width sharing for tcp. In *IEEE INFOCOM*, 2003.

[8] Jon Postel. Transmission control protocol. RFC793, September 1981. Available: http://www.rfc-editor.org/rfc/rfc793.txt. Last access: 2017-11-27.

[9] David Ros and Michael Welzl. Assessing ledbat's delay impact. *IEEE Communications Letters*, 2013.

[10] David Ros and Michael Welzl. Less-than-best-effort service: A survey of end-to-end approaches. *IEEE Communications Survey & Tutorials*, 2013.

[11] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low extra delay background transport (ledbat). RFC6817, December 2012. Available: http://www.rfc-editor.org/rfc/rfc6817.txt. Last access: 2017-11-27.

[12] N. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. Bershad. Receiver based management of low bandwidth access links. In *IEEE INFOCOM*, 2000.

[13] Si Quoc Viet Trang, Emmanuel Lochin, Cédric Baudoin, Emmanuel Dubois, and Patrick Gélard. Flower – fuzzy lower-than-best-effort transport protocol. In *40th Annual IEEE Conference on Local Computer Networks*, 2015.

[14] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. Tcp nice: a mechanism for background transfers. *ACM SIGOPS Operating Systems Review*, 2002.