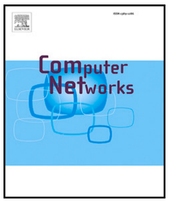
Computer Networks 233 (2023) 109841

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Design, implementation and validation of a receiver-driven less-than-best-effort transport

Marcelo Bagnulo [a,*], Alberto García-Martínez [a], Anna Maria Mandalari [b], Praveen Balasubramanian [c], Daniel Havey [d], Gabriel Montenegro [e]

[a] *U. Carlos III de Madrid, Spain*
[b] *University College London, UK*
[c] *Confluent, USA*
[d] *Microsoft, USA*
[e] *Samsung Research America, USA*

ARTICLE INFO

Keywords:
Less-than-best-effort
Congestion control
LEDBAT++
Receiver-based

ABSTRACT

LEDBAT++ is a congestion-control algorithm that implements a less-than-best-effort transport service. In this paper we present rLEDBAT, a purely receiver-based mechanism to implement LEDBAT++ for TCP. rLEDBAT enables a receiver to select some incoming traffic as less-than-best-effort, managing the capacity of the downlink. We describe the different mechanisms composing rLEDBAT that enable the execution of the LEDBAT++ congestion control algorithm at the receiver. We have implemented and experimentally tested rLEDBAT. We validate that the mechanisms incorporated by rLEDBAT at the receiver are indeed effective to implement a less-than-best-effort transport service at the receiver, as it performs similarly to the original sender-based LEDBAT++.

1. Introduction

In the context of service classes for Internet traffic, the Best Effort traffic is the default class used by Internet traffic and the Less-than-best-effort (LBE) service class is one that obtains smaller bandwidth than best-effort traffic, when sharing a bottleneck with it. LEDBAT++ [1] is a Congestion-Control Algorithm (CCA) that implements a Less-Than-Best-Effort (LBE) service for TCP. By reacting earlier and more aggressively to congestion onset than Cubic [2], LEDBAT++ yields bandwidth to competing Cubic-TCP flows, allowing them to use more of the available capacity. In the absence of competing Cubic-TCP traffic, LEDBAT aims to make an efficient use of the available capacity, while keeping the queueing delay within predefined bounds.

LEDBAT++ is the evolution of the original LEDBAT CCA proposed back in 2012 [3]. Similarly to LEDBAT, LEDBAT++ reacts both to packet loss and to variations in delay. Regarding to packet loss, both LEDBAT and LEDBAT++ react with a multiplicative decrease, similar to commonly used TCP congestion controllers. Regarding delay, both versions of LEDBAT aim for a target queueing delay. When the measured current queueing delay is below the target, they increase the sending rate and when the delay is above the target, they reduce the sending rate.

LEDBAT++ incorporates a number of modifications, designed to overcome several shortcomings of the original LEDBAT algorithm identified over the last few years [4]. Notably, LEDBAT++ uses additive increase/multiplicative decrease (AIMD) instead of the additive increase/additive decrease approach used by LEDBAT in order to address the late-comer advantage problem that affects LEDBAT [5]. LEDBAT++ uses Round Trip Times (RTTs) instead of one-way delays for its calculations to avoid the difficulties with clock frequency and clock drift uncertainties.

Widespread use of LBE transports, including LEDBAT++, for transferring background delay-tolerant traffic is beneficial for competing traffic that is sensitive to latency. Using Cubic-TCP for large background traffic transfers results in long queues sustained for long periods, hindering the performance of other applications with more stringent latency requirements (e.g., real-time applications) [6]. When used for carrying background traffic, LEDBAT++ allows other traffic with more demanding requirements to complete faster and earlier. Also, when there is unused capacity available for background traffic, LEDBAT++ limits the additional delay introduced by the background traffic, aiding latency-sensitive traffic to encounter short queues while transiting. LEDBAT++ is commonly used to carry delay-insensitive background

* Corresponding author.
E-mail address: marcelo@it.uc3m.es (M. Bagnulo).

https://doi.org/10.1016/j.comnet.2023.109841
Received 15 November 2022; Received in revised form 21 April 2023; Accepted 22 May 2023
Available online 9 June 2023

traffic, such as automatic backups and software updates (including Microsoft's operating system updates).

While LBE transports are widely supported in the Internet today, there still exists impediments that prevent its wider adoption. In particular, as any other CCA used in the Internet today, LEDBAT++ is implemented at the sender of the traffic. Therefore, LEDBAT++ is not used when the sender fails to support LEDBAT++.

In this paper we propose **rLEDBAT**, a receiver-based version of LEDBAT++. rLEDBAT is a set of mechanisms that enable the full implementation of the LEDBAT++ CCA in the TCP **receiver**, without requiring any form of LEDBAT++ support at the sender-side.

1.1. r LEDBAT use cases

rLEDBAT empowers the client to enable LEDBAT++ for incoming traffic irrespectively of the server's side support. In many cases, the clients have stronger incentives for deploying LEDBAT++ than servers. This is so because it is fairly common that the client's access link is the bottleneck of the communication. It is reported that 40% of the home network users [7] and for between 25% and 70% of cellular users, depending on the cellular technology [8] suffer from this problem. In these cases, it is the user's delay-insensitive background traffic who is the sole responsible for the long queues and thus for the long delay penalty to the user's own latency-sensitive traffic. This is exacerbated by the excessively large buffers commonly observed in residential access links that result in the bufferbloat effect [9]. Furthermore, rLEDBAT allows the client to selectively enable LEDBAT++ for incoming flows irrespectively of the server's configuration, e.g., the video stream client uses Cubic-TCP/UDP while the software update client uses rLEDBAT.

There are also more sophisticated scenarios where server LEDBAT++ does not protect against large queues. One fairly common of such scenarios involves LEDBAT++-oblivious Content Distributions Networks (CDNs). Consider the case where the source of a file to be distributed (e.g., a software developer that wishes to distribute a software update) enables the LEDBAT++ CCA in the servers containing the source file. However, because the file is being distributed through a CDN which surrogates do not support LEDBAT++,[1] the result is that the file transfers, originated from CDN surrogates will not be using LEDBAT++.

Another frequent scenario where the client is not receiving LEDBAT++ enabled traffic in spite of being enabled at the sender is when there is a LEDBAT++-oblivious middlebox sitting between the LEDBAT++ enabled sender and the client. In this case, the leg between the server and the middlebox is LEDBAT++ enabled while the segment between the middlebox and the client is not. Proxies and other middleboxes are a commonplace in the Internet. For instance, in the case of mobile networks, [10] observes that 25% of the 956 mobile users measured are behind a non-transparent proxy. Enterprise networks usually deploy corporate proxies for filtering and firewalling. In the case of satellite links, Performance Enhancement Proxies (PEPs) are deployed to mitigate the effect of the long delay in TCP connection. All these proxies terminate the TCP connection on both ends and prevent the use of LEDBAT++ in the segment between the proxy and the sink of the content, the client. rLEDBAT enables the use of LBE traffic class for file distribution in these setups.

[1] In our operational experience, the current support for LEDBAT in CDNs is very limited, resulting in LEDBAT rarely used for Windows Updates originated from CDNs. Moreover, it would not only be necessary that the CDN supports LEDBAT++, but it would also require some protocol to allow the content provider to convey to the CDN which content should be delivered using LEDBAT++ and which one should be distributed using Cubic-TCP.

1.2. Contributions

We present rLEDBAT, a receiver-based implementation of a LEDBAT++ congestion control algorithm for TCP. The main contributions of the paper are the following:

- We design the rLEDBAT mechanisms that enable the execution of the LEDBAT++ CCA at the receiver-end of a TCP connection. We implement the proposed rLEBDAT in Linux and make our implementation available as open-source in github.
- Using our rLEDBAT implementation for Linux, we experimentally validate that rLEDBAT performs similarly to the sender-based LEBDAT++ in a wide set of conditions. We test multiple scenarios, including rLEDBAT flows running solo, multiple rLEDBAT flows competing against each other, and rLEDBAT flows competing against TCP using both Cubic and BBR CCA.
- As rLEDBAT behaves similarly than LEDBAT++, they both exhibit the same pitfalls. Notably, both sender-based LEDBAT++ and rLEDBAT fail to yield when competing against BBR flows under certain conditions. We propose a modification of the LEDBAT++ CCA to address the identified shortcoming. We implement the proposed solution in our rLEDBAT implementation, and we verify that it solves the identified issue, i.e., the modified rLEDBAT implementation yields against BBR in all tested conditions.

The rest of this paper is structured as follows. In Section 2 we present the necessary background information regarding LEDBAT++. In Section 3 we describe the rationale for the design of the proposed rLEDBAT mechanism and we describe it in detail. Next , in Section 4 we present our Linux implementation of rLEDBAT. We continue in Section 6 with a description of the experimental evaluation of the rLEDBAT mechanism. Sections 7 and 8 describe and validate the proposed modifications for the LEDBAT++ CCA to address the identified limitations. Finally, Section 10 concludes the paper.

2. LEDBAT++ background

LEDBAT++ [1] is a LBE congestion-control algorithm that reacts both to packet loss and to delay variations.

2.1. LEDBAT++ overview

LEDBAT++ controls the sending rate through the calculation of a Congestion Window, CW. LEDBAT++ updates the CW based on delay variations and packet loss. With respect to packet loss, LEDBAT++ reacts by reducing the CW to half of its value when a loss is detected.

With respect to delay variations, LEDBAT++ aims for a pre-defined queueing delay target, T (defined equal to 60 ms in the specification). LEDBAT++ continuously estimates the current queueing delay, q_d. If the current queueing delay q_d is larger than the target queueing delay T, LEDBAT++ multiplicatively decreases the Congestion Window. Conversely, if the delay is smaller than the target, LEDBAT++ additively increases the Congestion Window.

LEDBAT++ estimates the current queueing delay (q_d) by subtracting the base round-trip-time (RTT_b) from the current RTT (RTT_c). The base RTT is calculated as the minimum RTT observed in the last 10 min of the lifetime of the communication. The current delay is the last RTT measured in the communication. Both values are filtered to eliminate noise by taking the minimum of the last n values, n being at least 4. The current queueing delay is then calculated as:

$$q_d = RTT_c - RTT_b \tag{1}$$

LEDBAT++ defines the GAIN parameter as follows:[2]

$$GAIN = \frac{1}{min 16, CEIL(2 * \frac{T}{RTT_b})} \tag{2}$$

[2] CEIL(X) is defined as the smallest integer larger than or equal to X.

For a T equal to 60 ms, this means that GAIN is equal to 1 for base RTTs larger than 120 ms, equal to 0.5 for base RTTs of 60 ms and equal to $\frac{1}{16}$ for RTTs smaller than 7.5 ms. As it will be described later, the GAIN parameter is used to make LEDBAT++ AIMD less aggressive than the one used by Cubic-TCP, even in cases where the buffers are small. The (unstated) assumption is that networks with small base RTTs are more likely to have shallow buffers (e.g., datacenter networks).

LEDBAT++'s AIMD reacts to changes in the queueing delay by updating its CW as follows: if $q_d < T$, then

$$CW_{n+1} = CW_n + GAIN \tag{3}$$

and if $q_d > T$, then

$$CW_{n+1} = CW_n + max(-\frac{CW_n}{2}, (GAIN - CW_n \cdot (\frac{q_d}{T} - 1))) \tag{4}$$

with CW_n being the value of the Congestion Window computed at RTT n and MSS being the Maximum Segment Size of the TCP connection.

Eq. (3) defines the Additive Increase part. It basically states that the CW increases up to 1 MSS per RTT (being 1 MSS if the base RTT is 120 ms or larger, and less than that for smaller base RTTs). The purpose of this is to ensure that LEDBAT++ increases less aggressively than Cubic-TCP when the base RTT is less than 120 ms. This is especially important when the bottleneck link buffer is small and LEDBAT++ is solely decreasing its CW based on losses.

Eq. (4) describes the Multiplicative Decrease part. By using multiplicative decrease (instead of the additive decrease used in LEDBAT), LEDBAT++ aims to overcome the (un)fairness issues identified in LEDBAT. The multiplicative decrease factor depends on the ratio between the current queueing delay and the Target T, so that LEDBAT++ reacts softly to small excesses in the queueing delay, allowing a smooth operation around the target point. The multiplicative decrease is capped to half, to avoid starving LEDBAT++ in cases of spikes in the delays.

LEDBAT++ performs a slow-start increase at the beginning of the connection. LEDBAT++'s slow-start is similar to the one of Cubic-TCP, in the sense that the CW increases exponentially. However, in order to be less aggressive than Cubic-TCP, instead of doubling the CW every RTT, LEDBAT++ multiples the CW by a factor of $2 \cdot GAIN$, with $GAIN$ being always less or equal to 1. LEDBAT++ exits the exponential growth of the initial-slow start when the measured queueing delay surpasses $\frac{3}{4}$ of the Target T, in order to avoid overshooting.

In addition, LEDBAT++ performs periodic slow-downs to obtain more accurate measurements of the base RTT and overcome the late-comer advantage identified in LEDBAT. This means that periodically LEDBAT++ sets the CW to two MSS during two RTTs and then performs a slow-start increase back to the CW value that it was using before the periodic decrease. An initial slow-down is performed 2 RTTs after exiting the initial slow-start. After that initial slow down, LEDBAT++ performs slow-downs periodically. If we call Tss the time that it takes for the slow-start to ramp back up, then LEDBAT++ performs the next periodic slow down after a period equal to $9 \cdot Tss$.

2.2. LEDBAT++ performance

In previous work, we have experimentally studied the performance of LEDBAT++ [6,11] in different conditions.

LEDBAT++ exhibits two different modes of operation, delay-based and loss-based. When the bottleneck link buffer has enough capacity to generate a queueing delay equal or higher than LEDBAT++'s target delay[3] (T), then LEDBAT++ runs in delay-based mode, while when the buffer is smaller, losses are generated before LEDBAT++ can react to the increase in the queueing delay, so it runs in loss-based mode. We have tested both modes of operation in our previous work.

We have measured the performance of LEDBAT++ when running solo in a bottleneck link [6]. When the buffer is larger than T, LEDBAT++ is able to seize 90% of the available capacity for base RTT smaller than 300 ms. The 10% penalty is caused by the periodic slowdowns built into LEDBAT++. For larger base RTTs, the achieved utilization decreases (e.g., about 80% utilization for base RTT equal to 400 ms). The reason is that, after a periodic slowdown, LEDBAT++ struggles to restore the rate it was sending before the slowdown.

In addition, we measured the fairness between multiple LEDBAT++ flows and we have observed that LEDBAT++ is able to evenly split the available capacity between multiple LEDBAT++ flows (with the same base RTT), irrespectively of when each flow started, solving the late-comer advantage problem [5] of the original LEDBAT specification.

We also tested LEDBAT++ when competing against Cubic [6] and we observed that LEDBAT++ yields in front of Cubic in all tested conditions, effectively implementing an LBE transport in this case.

Finally, we tested LEDBAT++ when competing against both versions of BBR, BBRV1 and BBRv2 [11]. For base RTTs larger than the minimum between the LEDBAT++ target T and the buffer size, LEDBAT++ yields in front of BBR as expected, but for smaller base RTTs, LEDBAT++ does not yield, and consequently it does not behave as an LBE transport. For these small base RTTs, when LEDBAT++ is competing against BBRv1, they evenly split the available capacity, but when LEDBAT++ competes against BBRv2, it is the BBRv2 flow the one that yields (instead of LEDBAT++).

3. r LEDBAT mechanism

rLEDBAT enables a TCP receiver to use the LEDBAT++ CCA to control the sender's rate. The rLEDBAT mechanisms is fully contained at the receiver and the sender is agnostic to rLEDBAT/LEDBAT. As depicted in Fig. 1, rLEDBAT provides the mechanisms that enable the receiver to gather the input required by the LEDBAT++ CCA to calculate the Congestion Window. It also provides the means for the receiver to control the sender's rate. In terms of input for the LEDBAT++ algorithm, rLEDBAT enables the receiver to estimate the queueing delay by measuring the RTT. Also, the rLEDBAT mechanisms enable the receiver to detect packet losses and retransmissions, which also serve as input to the LEDBAT++ algorithm. Finally, the rLEDBAT receiver controls the sender's rate using the Receive Window ($RWND$) field of the TCP header. rLEDBAT defines how to safely convey the Congestion Window computed by the LEDBAT++ CCA using the Receive Window.

3.1. rLEDBAT mechanisms to estimate the base and current RTT

LEDBAT++ estimates the queueing delay by subtracting the base RTT (i.e., the constant components of the RTT) from the current RTT and uses it to compute the Congestion Window, as described in Section 2. rLEDBAT passively measures the RTT by inspecting the incoming and outgoing packets and matching them in pairs. We next describe how rLEDBAT measures the RTT and how it uses the measurements to estimate the current and the base RTT.

Because TCP is a reliable data transfer protocol, when a TCP sender sends data, it expects to receive an acknowledgement back. A TCP sender can naturally leverage on this data-ACK exchange to match outgoing and incoming packets and measure the RTT. Unfortunately, this is not always the case for a TCP receiver. In particular, it is possible for TCP endpoints to behave as pure receivers, meaning that one of the endpoints of the TCP connection is sending data (pure sender) and the other one is only receiving it (pure receiver). In this scenario, there is no data flowing from the rLEDBAT receiver to the sender, making impossible to match data packets with acknowledgements packets to measure the RTT. This is expected to be a very common scenario for rLEDBAT, e.g., downloading an operating system update or any other form of background download.

[3] We call this buffer *larger than* T for short

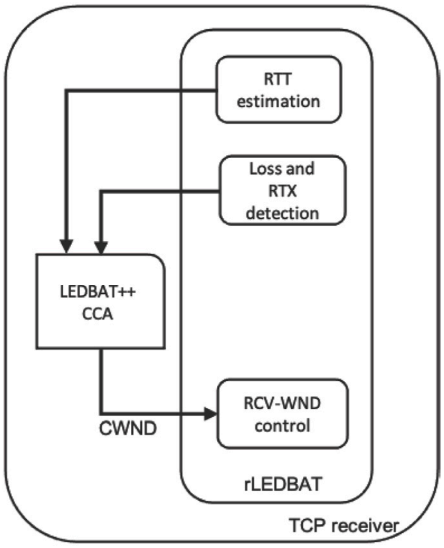


Fig. 1. rLEDBAT architecture at the TCP receiver and interaction with LEDBAT++ CCA.

In order to measure the RTT from a pure receiver side, rLEDBAT relies on the use the Time Stamp (TS) option. By matching the TSVal value carried in outgoing packets with the TSecr value observed in incoming packets, it is possible even for a pure receiver to match outgoing packets with incoming packets and measure the RTT.

This implies that rLEDBAT can only work if the TimeStamp option is enabled by both ends. We measured the Alexa top-500k servers[4] and queried for TimeStamp Option support. We found that 76% of the queried servers support Time Stamps. While not universal, the support is fairly widespread.

There are two reasons why the server may not send a packet immediately back to the rLEDBAT client, artificially increasing the measured RTT. The first reason is when A has no data to send. The second is when A has no available window to put more packets in-flight. We describe next how each of these cases is addressed.

Regarding to the first reason for having an inflated RTT, the lack of data to send in the sender node, we argue that this is rare in the expected rLEDBAT use cases. rLEDBAT will be used mostly for background file transfers, so the sender will have data to send throughout the lifetime of the communication. We propose to address this situation by using a minimum filter of the last k samples when measuring the current RTT to discard the (rare) artificially bloated samples.

Concerning the second reason for having an inflated RTT, the lack of window to send more packets, the limitation can come either from the Congestion Window in the server or from the announced Receive Window from the rLEDBAT client. Normally, the Receive Window will be the one to limit the sender's transmission rate, since LEDBAT++ is designed to be more restrictive on the sender's rate than Cubic-TCP. In any case, if the limiting factor is the Congestion Window in the sender, it is irrelevant if rLEDBAT further reduces the Receive Window due to a bloated RTT measurement, since the rLEDBAT is not actively controlling the sender's rate. To address the case in which the limiting factor is the Receive Window announced by rLEDBAT, the receiver should discard the RTT measurements done while reducing the window and avoid including bloated samples in the queueing delay estimation.[5]

[4] https://www.alexa.com/topsites

[5] The rLEDBAT receiver is aware whether a given TSVal value was sent in a packet where the window was reduced, and if so, it can discard the corresponding RTT measurement.

Finally, depending on the frequency of the local clock used to generate the values included in the TS option, several packets may carry the same TSVal value. If that happens, the rLEDBAT receiver will be unable to match the different outgoing packets carrying the same TSVal value with the different incoming packets carrying also the same TSecr value. However, it is not necessary for rLEDBAT to use all packets to estimate the RTT and sampling a subset of in-flight packets per RTT is enough to properly assess the queueing delay. So, in order to address this issue, rLEDBAT only measures the RTT matching the first outgoing packet with a given value in the TSVal and the first incoming packet with the same value in the TSecr.

Summarizing:

- rLEDBAT relies on TCP timestamps to passively measure the RTT
- rLEDBAT only considers the first incoming and the first outgoing packet with a given timestamp value
- rLEDBAT dismisses RTT measurements done while shrinking the LEDBAT++ window
- rLEDBAT estimates the base RTT (RTT_b) by taking the minimum value observed in the measured RTTs over a (long) period of time.
- rLEDBAT computes the current RTT (RTT_c) applying a minimum filter of the last k samples. rLEDBAT feeds both values to the LEDBAT++ CCA.

3.2. rLEDBAT mechanisms for detecting packet loss and retransmissions

The rLEDBAT receiver is capable of detecting retransmitted packets in the following way. We call $S1$ the highest sequence number correspondent to a received byte of data (not assuming that all bytes with smaller sequence numbers have been received already, there may be holes) and we call $TS1$ the TSVal value corresponding to the segment in which that byte was carried. $S2$ stands for the sequence number of a newly received segment, and we call $TS2$ the TSVal value of the newly received segment. If $S2 < S1$ and $TS2 > TS1$ then the newly received segment is a retransmission. This is so because the newly received segment was generated later than another already received segment which contained data with a larger sequence number. Thus, this segment was lost and was retransmitted.

3.3. rLEDBAT mechanism for controlling the sender through the receive window

In order to empower the receiver to control the sender's rate using the Receive Window, rLEDBAT needs to take into account a number of constraints. First of all, we need to keep in mind that the $RWND$ is used by TCP for flow control. In order to avoid confusion, we will call $fcWND$ the value of the window calculated for TCP flow control purposes.

rLEDBAT uses the LEDBAT++ congestion control algorithm to calculate a Congestion Window which will next convey to the sender using the $RWND$ field of the TCP header. We call $rlWND$ the window value calculated by rLEDBAT. We observe that we now have two different data to convey to the sender, namely $fcWND$ and $rlWND$, and only one TCP field, $RWND$, to send it. In order to honor both of them, rLEDBAT includes the minimum of these two values in the $RWND$ field of TCP. During the rest of the paper, we focus on the case where the $fcWND$ is larger than the $rlWND$, so that rLEDBAT is actively managing the $RWND$ and flow control is not limiting the connection's speed.

When using rLEDBAT, two congestion controllers are in action in the flow of data from the sender to the receiver, the congestion control algorithm of TCP in the sender side and the LEDBAT++ congestion control algorithm executed in the receiver and conveyed to the sender through the $RWND$. In normal TCP operation, the sender uses the minimum of the Congestion Window computed by the CCA running at the sender ($sCWND$) and the Receiver Window ($RWND$) to calculate the sender's window ($SWND$). This is also true for rLEDBAT, as the sender is a regular TCP sender. Because LEDBAT++ is designed to react earlier and more aggressively to congestion than Cubic-TCP congestion control, the $rlWND$ contained in the $RWND$ field of TCP will be in general smaller than the Congestion Window calculated by the TCP sender, implying that the LEDBAT++ congestion control algorithm at the receiver end will be effectively controlling the sender's window ($SWND$). Moreover, this also guarantees that even if the queueing delay is mis-estimated, the flow will never transmit more aggressively than a TCP flow, as the sender's Congestion Window limits the sending rate.

In summary, the sender's window is:

$$SWND = min(sCWND, rlWND, fcWND) \quad (5)$$

There are a few other considerations to be taken into account when calculating $rlWND$ that we describe next.

3.3.1. Reducing the window without shrinking it

The LEDBAT++ algorithm increases or decreases the $rlWND$ based on whether the last estimations of the queueing delay are above or below the target T. If the estimated queueing delay is above T, then the new Congestion Window will be smaller than the current one and there is the possibility that directly announcing in the $RWND$ this smaller value may result in shrinking the window, i.e., moving the right window edge to the left. Shrinking the window is discouraged (see [12]), as it may cause unnecessary packet loss and performance penalty.

To avoid window shrinking, the announced window can be reduced at most in the number of bytes contained in the received packet. This may not always be enough to honor the new calculated value of the $rlWND$. So, in order to reduce the window as dictated by the LEDBAT++ algorithm, the receiver will progressively reduce the advertised $RWND$, always ensuring that the reduction is less or equal than the received bytes, until the target window determined by the LEDBAT++ algorithm is reached.

3.3.2. Window scale option

The Window Scale (WS) option [13] allows increasing the maximum window size permitted by the Receive Window.

Regarding rLEDBAT, the use of the WS option implies that the changes in the window are expressed in the units resulting of the WS option used in the TCP connection (2 to the power of the value of the WS option). This means that the rLEDBAT client will have to accumulate the increases resulting from the different received packets, and only convey a change in the window when the accumulated sum of increases is equal or higher than one Receive Window unit.

Changes in the Receive Window that are smaller than 1 MSS are unlikely to have any immediate impact on the sender's rate, as usual TCP segmentation practice results in sending full segments (i.e., segments of size equal to the MSS). In case large amounts of data are transferred, which is the expected application for rLEDBAT, this requirement of completing a full MSS in the sender or in the receiver makes little difference.

Current WS option specification [13] defines that the allowed values for the WS option are between 0 and 14. Assuming a MSS around 1500 bytes, WS option values between 0 and 11 result in the Receive Window being expressed in units that are about 1 MSS or smaller. So, WS option values between 0 and 11 have no impact in rLEDBAT. WS option values higher than 11 can affect the dynamics of rLEDBAT, since control may become too coarse (e.g., with WS of 14, a change in one unit of the Receive Window implies a change of 10 MSS in the effective window). For the above reasons, we recommend that when rLEDBAT is used, the rLEDBAT client should set WS option values lower than 12. Note that the recommendation for rLEDBAT to set the WS option value to lower values does not preclude the communication with servers that set the

```
1: procedure ReceivePacket
2:     $receivedRTT = computeRTT(sentList, receivedTSecr, receivedTime)$
                            ▷ Looks for first sent packet with same TSval as TSecr, returns time difference
3:     $insertRTT(RTTlist, receivedRTT, receivedTSecr, receivedTime)$
           ▷ Inserts minimum value for a given receivedTSecr - note that many received packets may contain same receivedTSecr
4:     $filteredRTT = minLastKMeasures(RTTlist, K = 4)$
5:     $baseRTT = minLastNSeconds(RTTlist, N = 180)$
6:     $qd = filteredRTT - baseRTT$
7:     $ackedBytes = ackedBytes + receiveBytes$
           ▷ ackedBytes is the number of bytes that can be used to reduce the Receive Window - without shrinking it- if necessary
8:     if $retransmittedPacketDetected$ then
9:         $rlwnd = max(rlwnd \cdot \beta_l, 1)$      ▷ Only once per RTT
10:    end if
11:    if $qd < T$ then
12:        $rlwnd = rlwnd + \alpha * ackedBytes$
13:    else
14:        $rlwnd = max(rlwnd \cdot \beta_d, 1)$      ▷ Only once per RTT
15:    end if
16: end procedure
```

Fig. 2. Procedure executed when a packet is received.

WS option values to larger values, since the WS option value used is set independently for each direction of the TCP connection.

We performed a survey of WS option values normally used in the Internet today. We established TCP connections with the top 200k Alexa servers and observed the WS option values they used. Only 0.18% of the polled servers use WS option values larger than 11, confirming that our recommendation does not result in general in a restriction to current common practices.

3.4. Integrated LEDBAT++ & rLEDBAT algorithm

We next describe how to integrate the proposed rLEDBAT mechanisms and the LEDBAT++ CCA. We describe the integrated rLEDBAT/LEDBAT++ algorithm as two procedures, one that is executed when a packet is received by a rLEDBAT-enabled endpoint, Algorithm 2 and another, Algorithm 3, that is executed when the rLEDBAT-enabled endpoint sends a packet. At the beginning, $rlwnd$ is set to its maximum value, so that the sending rate of the sender is governed by the flow control algorithm of the receiver and the TCP slow start mechanism of the sender, and $ackedBytes$ is set to 0.

The data structures used in the algorithms are as follows. The $sentList$ is a list that contains the TSval and the local send time of each packet sent by the rLEDBAT-enabled endpoint. The TSecr field of the packets received by the rLEDBAT-enabled endpoint are matched with the $sendList$ to compute the RTT. The RTT values computed for each received packet are stored in the $RTTlist$, which contains also the received TSecr (to avoid using multiple packets with the same TSecr for RTT calculations, only the first packet received for a given TSecr is used to compute the RTT). It also contains the local time at which the packet was received, to allow selecting the RTTs measured in a given period (e.g., in the last 10 min). $RTTlist$ is initialized with all its values to its maximum.

4. r LEDBAT implementation

We implemented the rLEDBAT mechanism in Linux. We use `iptables` to insert two code hooks in the Linux kernel (Linux 3.13.0-24 version), one triggered every time a packet is sent, and the other every time a packet is received [14]. The packets to process are filtered according to the destination ports, although other criteria can be used to identify rLEDBAT flows. The receiver hook calls a module that computes the RTT value matching the TSecr of the received packet

```
1: procedure SENDPACKET
2:     if ($rlwnd > rlwndPrevious$) or ($rlwnd - rlwndPrevious < ackedBytes$) then
3:         $rlwndPrevious = rlwnd$
4:     else
5:         $rlwndPrevious = rlwnd - ackedBytes$
6:     end if
7:     $ackedBytes = 0$
8:     $rlwndPrevious = rlwnd$
9:     $rlwnd = min(rlwnd, fcWND)$
                                ▷ Compute the RWND to include in the packet
10: end procedure
```

Fig. 3. Procedure executed when a packet is sent.

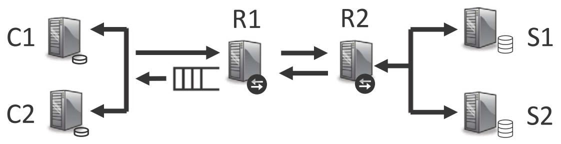


Fig. 4. Experiment setup.

with the TSval of a previously sent one, and it computes the queueing delay. In order to smooth the value of the queueing delay, we use the minimum of the value measured for the last 10 packets received. With this information, it computes the window to be used by the next packet to be sent.

The sender hook calls the Sender Module that computes the minimum of the Receive Window computed by TCP's flow and the one computed by rLEDBAT and includes this value in the Receive Window field of the TCP header of the outgoing packet. The source code for the rLEDBAT implementation is available at https://github.com/net-research/rledbat_module

5. Experimental setup

We next present the setup used to experimentally evaluate the performance of rLEDBAT.

In our experiments, we use the virtualised setup depicted in Fig. 4, which features a *dumbbell* topology, that allows us to generate LEDBAT++, rLEDBAT, Cubic and BBR flows that compete for the capacity of a bottleneck link. It also allows us to easily configure the characteristics of the bottleneck link, including the size of the buffer, the capacity and the delay. We use this topology for our experiments, because, as observed in the design of model-based congestion control algorithms [15], any path, no matter how complex it is, can be accurately modelled from the transport layer perspective as a single link with the RTT of the overall path and the capacity of the path's bottleneck link, which is exactly what this simple topology represents.

C1, C2, R1, R2 and S1 are Linux systems while S2 is a Windows 2019 Server with LEDBAT++ capability. We can configure S1 to use Cubic or any of the two versions of BBR. Cubic and BBRv1 are already available in the Linux kernel and BBRv2 is installed using [16]. Traffic is generated in S1 using the `nc` tool and in S2 using the `ctsTraffic` tool,[6] i.e., bulk transfer type of traffic in both cases. C1 and C1 use Ubuntu 14.04 LTS running our rLEDBAT implementation and will act as pure receiver. It uses `nc` for receiving traffic.

The link connecting R2 with R1 is the bottleneck link of the communications between S1 (S2) and C1 (C2). We set its capacity to different values using the `tbf` queueing discipline for the `tc` traffic control tool. A drop-tail buffer is configured in the R2-to-R1 link, with a size that we vary on different experiments, to represent different network setups. Throughout the rest of the paper, we express the size of the buffer B in milliseconds rather than in bytes. The size of the buffer in bits can be computed as $\frac{C \cdot B}{1{,}000}$, with C being the bottleneck link capacity expressed in bps. For our experiments, we consider two buffer scenarios: a buffer of 500 ms, to enable delay variations larger than the delay target for rLEDBAT/LEDBAT++, and 30 ms, below this target, so that packet losses due to buffer exhaustion may occur even with rLEDBAT/LEDBAT++. The links between S1 (S2) and R2 and the ones between C1 and R1 are configured with (much) larger capacities than the one of the bottleneck. During the experiments, we set the RTT of the path between S1 (S2) and C1 using `tc netem`.

In all the experiments, C1 (C2) connects to S1 and S2 nodes to perform downloads. Each flow is greedy, in the sense that it aims to transmit as much data as possible. Data is transferred using TCP, S1 using BBR or Cubic and S2 using LEDBAT++. When rLEDBAT is not used, TCP flow control never limits the communication rate, as we manually configure a large receiver window. To compute the rates for each flow, we start a `tcpdump` capture in C1. The MSS used is 1,390 bytes while the MTU is 1,456 bytes.

The experiments last for 300 s, as this duration is large enough to accommodate many periodic slow-downs for rLEDBAT/LEDBAT++. The rates reported as results account for the whole period, i.e., they include the start of the transmission. For all the experiments, the maximum buffer occupancy is always reached before 15 s, so that the largest contribution to the performance figures results from the congestion-avoidance phase. Unless otherwise stated, we perform 8 executions for each particular experiment configuration (congestion control mechanism, buffer size, end-to-end RTT value, etc.).

6. Experiments

We perform a number of experiments using the rLEDBAT implementation described in the previous section to validate that rLEDBAT performance. We aim to verify that rLEDBAT provides less-than-best-effort service and that it keeps the delay bounded for a wide range of situations. Moreover, we would like to verify that rLEDBAT performs similarly to the sender based implementation of LEDBAT++. To do that, during the analysis of the results, we compare the results obtained in these experiments with the ones obtained in previous experimental studies of the performance of (sender-based) LEDBAT++, namely [6, 11].

6.1. rLEDBAT and constant bit rate traffic

In this experiment, we measure the delay introduced by an rLEDBAT flow in a constant bit rate communication, such a VoIP communication. An explicit design goal of LEDBAT++, that rLEDBAT should preserve, is that the delay added by a rLEDBAT flow is limited to the target T, 60 ms in our case. For this, we consider a VoIP flow which generates 160-byte packets every 20 ms, for a total rate of 64 kbps. The VOIP flow carries data from S2 to C1. In addition, there is TCP connection between C1 and S1. C1 is using rLEDBAT for this TCP connection. This flow is greedy, in the sense that there is always data from the application to be sent, and it is only limited by the congestion control. We configure the bottleneck link (R1-R2) with a capacity of 1 Mbps. We configure the R1-R2 buffer to accommodate a maximum number of bytes equivalent to 150 ms. The RTT in absence of queueing is 20 ms.

The flow activation sequence for this experiment is the following: at time zero, the VOIP flow starts, 10 s later the rLEDBAT flow is initiated and they both stay for 40 s.

Fig. 5 shows the RTT measured for every VoIP packet exchange. During the first 10 s that the VOIP flow is the only one in the bottleneck and the RTT is in the order of the propagation delay of the path (the transmission delay for these packets is negligible). After the rLEDBAT flow starts, rLEDBAT introduces a variable additional delay that is below the target of 60 ms for most of the packets (91.7% of packets

[6] https://github.com/microsoft/ctsTraffic

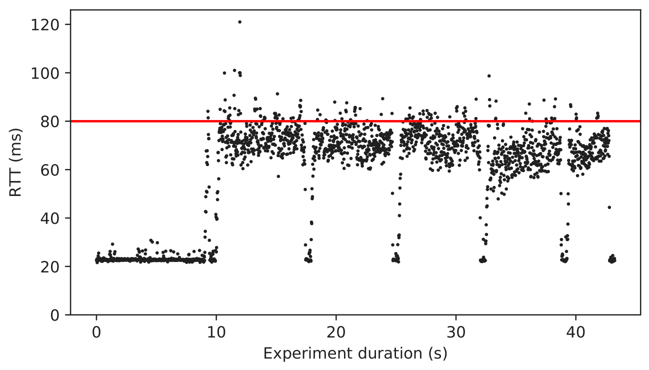


Fig. 5. RTT measured by a VoIP application in VoIP→rLEDBAT experiment, propagation delay=20 ms, buffer=150 ms, capacity 1Mbps.

to be exact). The few packets that appear with a delay higher than the target delay correspond to the packets sent during the before the multiplicative decrease caused when rLEDBAT detects that the queueing delay exceeded the target. We can also observe the periodic slow-downs performed by rLEDBAT.

We now consider the rate that rLEDBAT can achieve in this scenario. We observe that rLEDBAT is able to use 890 kbps of the bottleneck link capacity. Considering that the maximum rate is 954 kbps and that the VOIP flow consumes 64 kbps, rLEDBAT is capable of seizing the whole of the remaining capacity while keeping the queueing delay below the target of 60 ms.

We find similar results when performing experiments using different parameters, including, capacities ranging between 1Mbps and 40 Mbps, RTTs between 10 ms and 300 ms.

6.2. rLEDBAT solo performance

We next measure the performance of rLEDBAT when there is no competing traffic. The buffer for the experiments is large enough to generate a queueing delay larger than the target T. In this case rLEDBAT operates in delay-based mode. Then we run experiments using a buffer smaller than the target T, forcing rLEDBAT to operate in loss-based mode.

6.2.1. Experiments in delay-based mode

We do a set of experiments to measure the rate achieved by rLEDBAT for bottlenecks of different capacity when it is the only traffic in the link. For this experiment we use a buffer with enough capacity create a queueing delay of 500 ms (i.e., larger than rLEDBAT target T of 60 ms) and an RTT of 20 ms. We vary the bottleneck link capacity between 1Mbps and 50 Mbps. The server is using Cubic and the client enables rLEDBAT. The results are presented in Fig. 6. We find that our rLEDBAT implementation is able to seize about 90% of the available capacity. This is consistent with the results observed for sender based LEDBAT++ [6]. The 10% penalty is due to the waste imposed by the periodic slowdowns.

We next measure the rate achieved by rLEDBAT when it is the only traffic in a link, for different RTT values. For this experiment, we keep the buffer value fixed to 500 ms, the capacity set to 20 Mbps and vary the RTT. The results are plotted in Fig. 7. We observe that for base RTTs smaller than 200 ms, rLEDBAT is able to seize 90% of the capacity but that for larger base RTTs the utilization decreases. This is similar to the performance observed for LEDBAT++ [6].

In order to understand this behaviour, we look into the packet traces of one experiment. In Fig. 8 we plot both the flightsize and the RTT for a TCP connection between a Cubic sender and a rLEDBAT receiver over a 20 Mbps link using a buffer of 500 ms, with base RTT of 300 ms. We observe that rLEDBAT struggles to restore the rate after

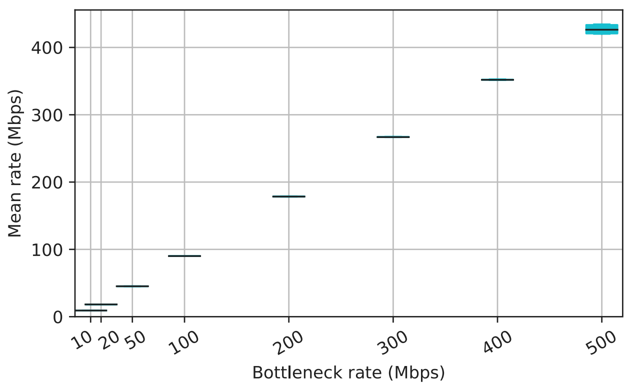


Fig. 6. Rate observed for rLEDBAT running solo in a link with different capacities, propagation delay of 20 ms, buffer set to 500 ms.

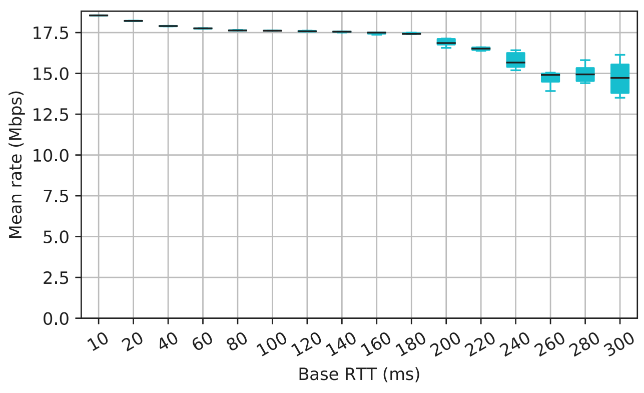


Fig. 7. Rate observed for rLEDBAT running solo in a link with different RTTs, link capacity of 20 Mbps, buffer set to 500 ms.

a periodic slow down. That is, after a periodic slow down, rLEDBAT tries to restore the window before the slowdown using an exponential growth, but the queueing delay exceeds rLEDBAT's target (as observed in the RTT plot), causing rLEDBAT to reduce its rate immediately after exiting the slow start. This behaviour prevents rLEDBAT from using the available capacity. Again, this is not a problem of the rLEDBAT's specific mechanisms, but a limitation of the LEDBAT++ algorithm itself, as reported in [6].

6.2.2. Experiments in loss-based mode

We perform the following experiments using a buffer of 30 ms. Because the buffer is smaller than the target T, rLEDBAT uses packet loss as congestion signal and reacts accordingly. We perform the experiments using a 20 Mbps bottleneck link and different base RTTs. The results are presented in Fig. 9. We can see in the plot that the utilization of the link decreases as the base RTT increases. This is to be expected because rLEDBAT is functioning as a loss-based AIMD, and AIMD mechanisms with the multiplicative factor of 0.5 require a buffer larger than the BDP to be able to seize all the capacity [17]. As the base RTT increases, the BDP also increases and rLEDBAT is able to seize less of the available capacity. Similar results were reported for sender-based LEDBAT++ [6]. However, we can see that rLEDBAT (slightly) outperforms in this case the LEDBAT++ performance reported in [6]. This is to be expected, since while LEDBAT++ reacts to losses (detected by DupACKS or timeouts), rLEDBAT reacts to retransmissions, which naturally occur at least one RTT later, making LEDBAT++ to reduce slightly more aggressively. So, we can conclude that the rLEDBAT mechanisms do not affect LEDBAT++ performance in these conditions.

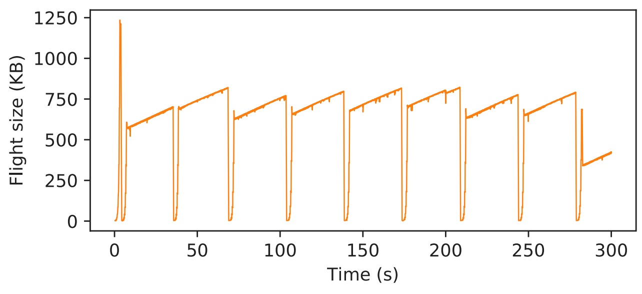
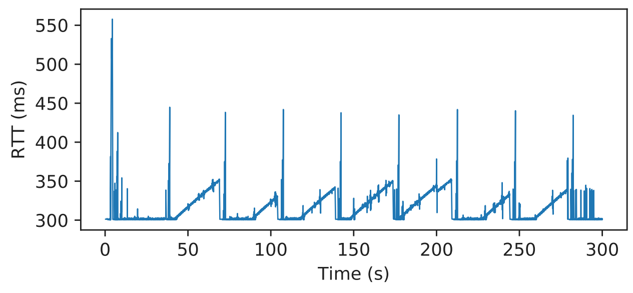



Fig. 8. Flightsize and RTT observed for rLEDBAT running solo in a link with base RTT set to 300 ms, link capacity of 20 Mbps, buffer set to 500 ms.

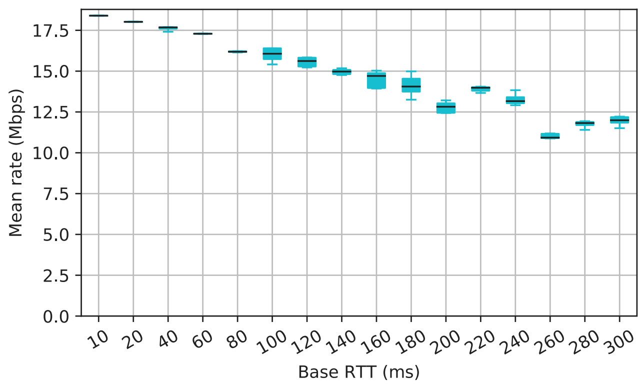


Fig. 9. Rate observed for rLEDBAT running solo in a link with different RTTs, link capacity of 20 Mbps, buffer set to 30 ms.

6.3. Inter-rLEDBAT fairness

Now we analyse the ability of rLEDBAT to achieve fairness when sharing the available bandwidth with another rLEDBAT flow. We perform a series of experiments involving two rLEDBAT flows competing for the capacity of a bottleneck, starting at the same time, for paths with different RTTs. In Fig. 10, we plot the Jain's fairness index [18] for the different experiments and we observe that the rLEDBAT protocol is close to the best case fairness (index equal to 1).

Given that it was reported that the original LEDBAT algorithm suffered from the late-comer-advantage [5], we also perform a series of experiments involving two competing rLEDBAT flows, only that in this case one flow starts 40 s after the other one. The resulting Jain's fairness indexes are presented in Fig. 11. Even though the observed results are slightly lower than the previous case of the simultaneous flows, we confirm that rLEDBAT roughly preserves the fairness achieved by the LEDBAT++, even in the case of two unsynchronized flows.

6.4. rLEDBAT and cubic

For the analysis of the interaction of rLEDBAT with Cubic, similarly than before, we first run experiments with buffers larger than the target T and then with smaller buffers.

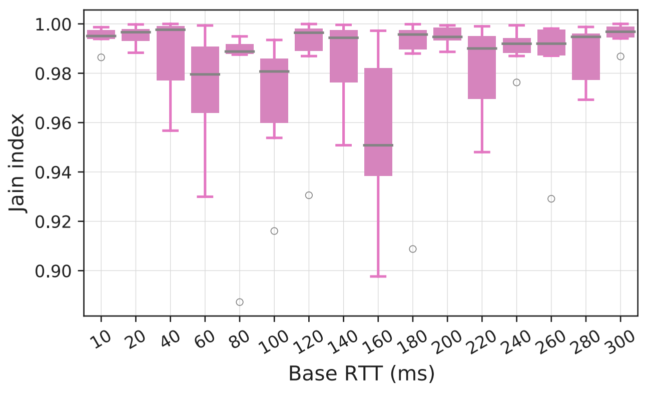


Fig. 10. Jain's fairness index observed for one rLEDBAT flow competing against another rLEDBAT flow, starting at the same time, using a bottleneck link of 20 Mbps with a buffer of 500 ms, varying the base RTT.

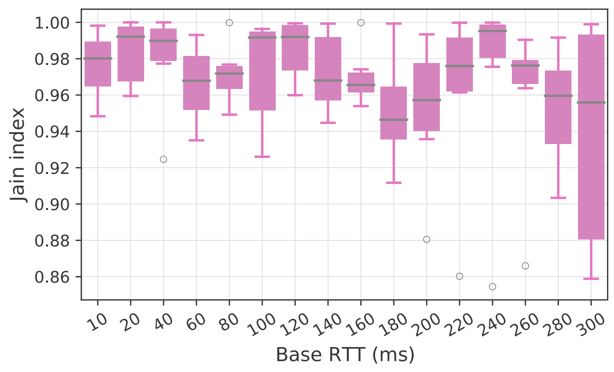


Fig. 11. Jain's fairness index observed for two competing rLEDBAT flows, starting 40 s apart, using a bottleneck link of 20 Mbps with a buffer of 500 ms, varying the base RTT.

6.4.1. Experiments with buffers larger than T

We first consider the case of a rLEDBAT flow competing against a Cubic flow, in a link with 20Mbps of capacity, a buffer capacity of 500 ms and different RTTs. Both flows start at the same time and the last for 300 s. Because the capacity of the buffer is enough to accommodate the amount of packets required to reach the target of 60 ms of rLEDBAT, in this setup, rLEDBAT is expected to detect an increase in the RTT higher than its target, and thus leave most of the bandwidth to Cubic. Results are plotted in Fig. 12. We can see that rLEDBAT behaves as expected as it yields in front of Cubic in all experiments. We also performed the same experiments with a bottleneck link of 500Mbps and we observe similar results.

We next perform a set of experiments varying the capacity of the link. The setup is the same as the previous one, only that the RTT is set to 20 ms and the capacity varies. Results are presented in Fig. 13. We verify that rLEDBAT yields in front of Cubic for all capacities tested.

Finally, we measure the impact of a long-lived rLEDBAT background traffic on the completion time of a shorter Cubic flow. We compare the results against the case when the shorter Cubic flow is running alone and also against the case when both the background flow and the shorter flow are using Cubic. In all experiments, we configure the background flow to start 15 s (in mean) before the shorter flow and terminate after its completion. To set a representative value for the size of the shorter flow, we consider that according to [19], 99.1% of 4 billion flows analysed in the context of a campus network carry less than 262 KB. Similarly, we analyse the 1.4 M flow traces from an Italian nation-wide Internet Service Provider (ISP), collected in [20] in 2017 and we observe that 90% of flows have a size smaller than 50,150 bytes, and 95% of the flows are smaller than 227,500 bytes.

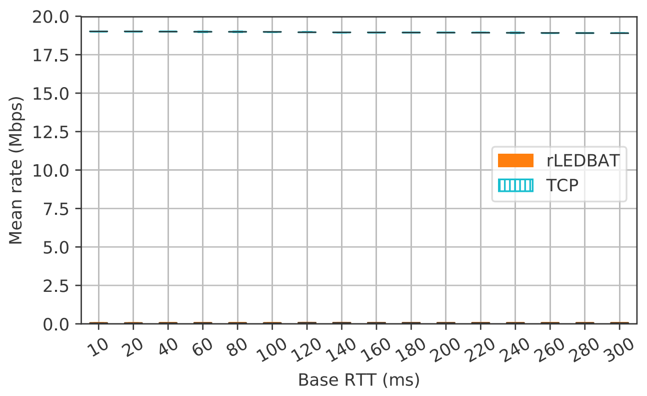


Fig. 12. Rate observed for a rLEDBAT flow competing against a Cubic flow in a 20 Mbps link with buffer set to 500 ms for different RTT values.

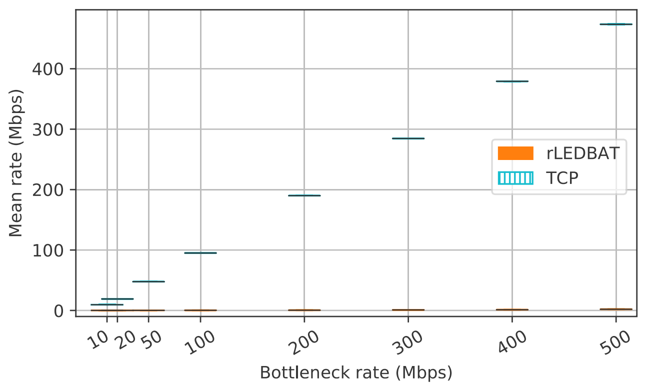


Fig. 13. Rate observed for a rLEDBAT flow competing against a Cubic flow with a base RTT of 20 ms and buffer set to 500 ms for different link capacities.

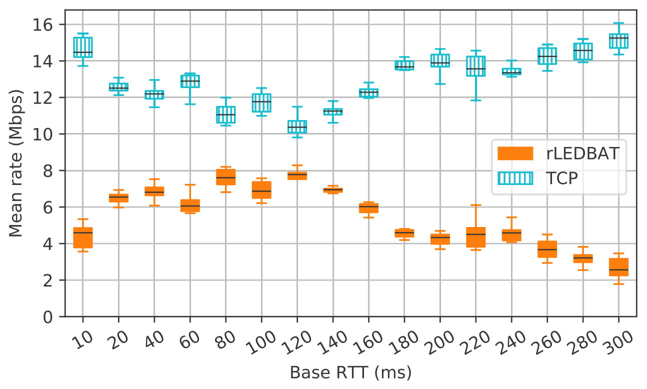


Fig. 14. Rate observed for a rLEDBAT flow competing against a TCP flow using a link of 20 Mbps with a buffer of 30 ms for different RTT values.

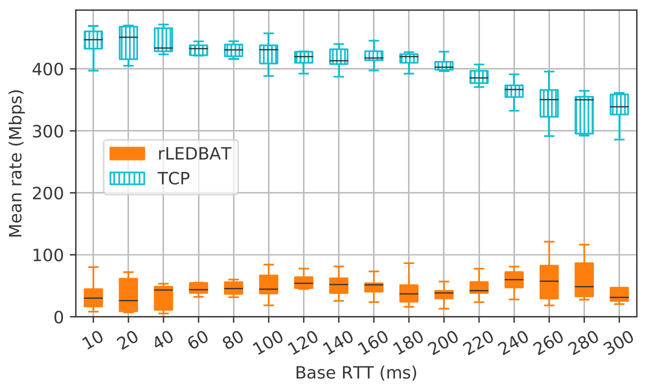


Fig. 15. Rate observed for a rLEDBAT flow competing against a TCP flow using a link of 500 Mbps with a buffer of 30 ms for different RTT values.

Table 1
Mean values for interaction with short-lived TCP flows for different Round Trip Times (RTT), Cubic server.

	20 ms	100 ms	200 ms	300 ms
TCP-Cubic short (solo)	0.16	0.62	1.22	1.82
rLEDBAT client, then TCP-Cubic short	0.56	0.95	1.48	1.84
TCP-Cubic long, then TCP-Cubic short	3.26	4.84	4.76	4.99

Based on this, we next investigate how long-lived rLEDBAT background traffic impacts the completion time of a Cubic flow carrying 262 KB. In Table 1 we present the means for the completion time of a Cubic flow carrying 262 KB of data in three different scenarios: (i) when running alone in the link, (ii) when competing with a long rLEDBAT flow, and (iii) when competing with a long Cubic flow. As we can see there is slight increase in the completion time of the shorter flow when it competes against an rLEDBAT flow compared to the case when it runs solo in the link. However, the completion time when there is a background Cubic flow is between 4 and 20 times larger. We then conclude that using rLEDBAT for background flows also benefits shorter best-effort flows in terms of the completion time.

6.4.2. *Experiments with buffers smaller than T*

In Fig. 14 we plot the rate achieved by a rLEDBAT flow and a Cubic flow competing for the capacity of a 20Mbps link using a buffer of 30 ms for different base RTTs. We see that the rLEDBAT does not fully yield and it is able to seize about 20% of the available capacity while the rest is used by the Cubic flow. This is expected, as both rLEDBAT and Cubic are reacting to losses, but the rLEDBAT AIMD parameters, based on new-Reno, are less aggressive than Cubic ones, allowing Cubic to seize most of the capacity, even when the buffer is smaller than the target T. This is more clearly noted for RTTs larger than 120 ms, because the GAIN parameter of LEDBAT++ decreases, as described in Section 2.1. The observed behaviour is similar to the ones reported for LEDBAT++ and Cubic in [6]. Similarly to the previous experiments using buffers to 30 ms, rLEDBAT is more aggressive than LEDBAT++, because of its delayed reaction to losses (as it reacts to retransmissions).

We also perform experiments using a 500Mbps bottleneck link, shown in Fig. 15. In this case we observe that Cubic seizes significantly more share than rLEDBAT, compared to the case of 20Mbps. This is to be expected, as the bandwidth used for the experiment falls further away from Cubic's TCP-friendly region [2]. As rLEDBAT implements the new-Reno AIMD mechanism, it underperforms compared to Cubic in this scenario.

6.5. *rLEDBAT and BBR*

We also tested the behaviour of rLEDBAT when competing with BBR. There are two versions of the BBR protocol, BBRv1 and BBRv2. BBRv1 is the original protocol defined by Google and currently deployed in all Google and YouTube external servers, as well as in Netflix and many others. Several BBRv1 shortcomings have been reported, including that it is overly aggressive when competing against Cubic as it does not react to packet losses. BBRv2 is an updated version that addresses the identified limitations. It is less aggressive than BBRv1, it reacts to losses and tries to impose a shorter queueing delay. We tested how rLEDBAT performs against both versions of BBR.

6.5.1. *rLEDBAT and BBRv1*

In Fig. 16 we present the results of a rLEDBAT flow competing against a BBRv1 flow in a 20 Mbps bottleneck link with a 500 ms buffer

Table 2
Mean values for interaction with short-lived TCP flows for different Round Trip Times (RTT), BBRv1 server for TCP flows.

	20 ms	100 ms	200 ms	300 ms
TCP-BBRv1 short (solo)	0.17	0.68	1.34	2.03
rLEDBAT client, then TCP-BBRv1 short	0.55	1.04	1.61	2.02
TCP-BBRv1 long, then TCP-BBRv1 short	0.41	1.23	1.85	2.47

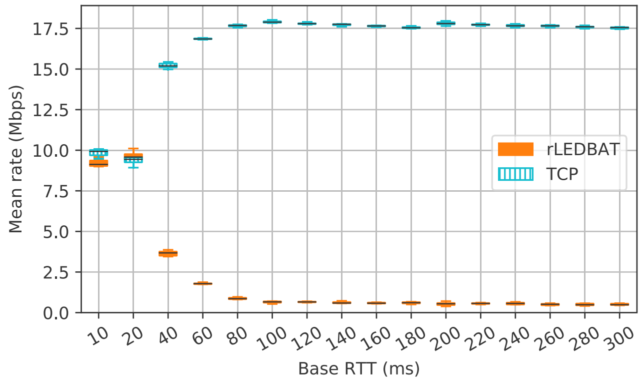


Fig. 16. rLEDBAT and BBRv1 in a 20 Mbps link with 500 ms of buffer, for different RTT values.

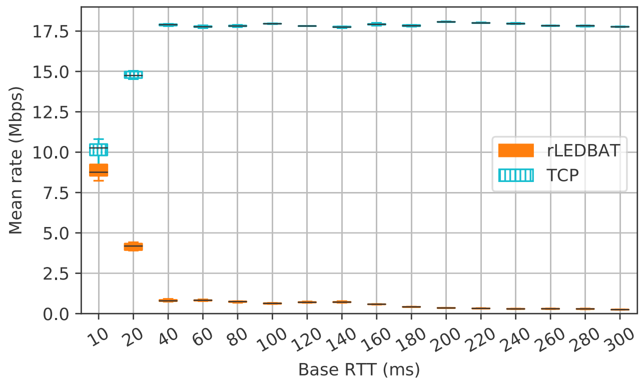


Fig. 17. rLEDBAT and BBRv1 in a 20 Mbps link with 30 ms of buffer, for different RTT values.

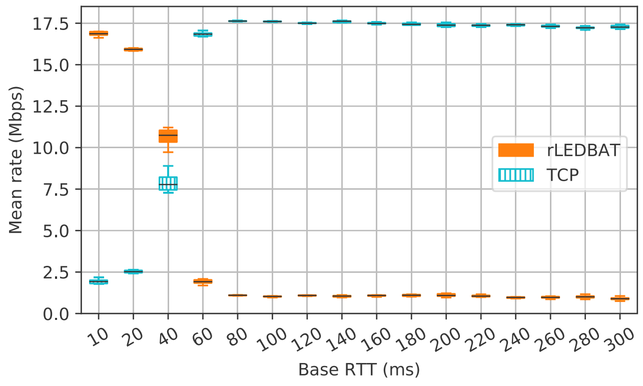


Fig. 18. rLEDBAT and BBRv2 in a 20 Mbps link with 500 ms of buffer, for different RTT values.

for different values of the RTT. We can see that rLEDBAT yields in front of BBRv1 for RTTs larger than 60 ms (i.e., rLEDBAT target T), but that for smaller RTT values, rLEDBAT seizes a significant part of the available capacity. For instance, when the RTT is 20 ms or smaller, they split the capacity evenly. This is the same behaviour that has been reported for LEDBAT++ when competing against BBRv1 [11]. Indeed, when the RTT is smaller than rLEDBAT/LEDBAT++'s target T, BBRv1's cap on the flight size prevents BBR to inject more than twice the Bandwidth-Delay Product (BDP). Thus, BBRv1 cannot inject enough packets to generate a queueing delay equal or larger than rLEDBAT's target T, which in turn allows rLEDBAT/LEDBAT++ to seize part of the available capacity. As such, this behaviour is not due to the rLEDBAT specific mechanisms, but a problem with the LEDBAT++/BBRv1 interaction that has been previously reported [11].

We also quantified the impact of a rLEDBAT background flow on the completion time of a shorter BBRv1 flow. Similarly to the experiments done for Cubic, we measured the completion time of a 262-KB BBRv1 flow in the three scenarios, namely, when running alone in the link, when competing against a long-lived rLEDBAT flow and when competing against a long-lived BBRv1 flow. The results are presented in Table 2. We can see that in general, the smallest completion time are when the shorter flow is alone, followed by the ones with rLEBDAT background traffic and finally the ones with BBRv1 background traffic, as in the Cubic experiments. The only exception to this is the case of RTT of 20 ms, where the experiments with BBRv1 background flow exhibits a shorter completion time than the ones with rLEDBAT background traffic. This is consistent with the explanation that the BBRv1 introduces a queue of an extra BDP, which for the case of RTT of 20 ms is smaller than the one of 60 ms inflicted by the rLEDBAT background traffic, resulting in a lower completion time overall for these cases. The other observation is that the experiments using BBRv1 background flows have a much lower completion time than the ones we measured in the Cubic case (see previous section) and much closer, albeit higher, than the rLEDBAT background traffic. This is consistent with the design of BBRv1 which aims for short queues. So, while there are benefits from using rLEDBAT for background traffic, those are less when compared to BBRv1 than when compared to Cubic.

We next perform another series of experiments with the same parameters but using a buffer of 30 ms, forcing rLEDBAT to operate in loss-based mode. The results are plotted in Fig. 17. We observe a similar behaviour than the case of larger buffer, the main difference being that rLEDBAT yields for base RTTs larger than the buffer (i.e., 30 ms) rather than for base RTTs larger than the target T (60 ms). This also occurs for LEDBAT++ competing against BBRv1 in these conditions, see [11], and it is consistent with the previous experiments. Indeed, when the base RTT is smaller than the buffer capacity, BBRv1 is limited by the flightsize cap, which allows the rLEDBAT flow to seize part of the capacity that the BBRv1 is unable to. When the base RTT is larger than the buffer, then the flightsize cap does not prevent the BBRv1 flow to fill in the buffer and seizes all the capacity, forcing rLEDBAT to yield.

6.5.2. *rLEDBAT and BBRv2*

We next repeat the same set of experiments using BBRv2 instead of BBRv1, see the results in Fig. 18. Similarly to the previous case with BBRv1, rLEDBAT yields for RTTs larger than the 60 ms of the target T; but that for smaller RTTs, the situation differs from the BBRv1 case, as BBRv2 actually yields in front of rLEDBAT in this case. This is consistent with the results reported for BBRv2 and LEDBAT++ in [11], and it is to be expected given that the modifications introduced in BBRv1, which make BBRv2 less aggressive than BBRv1. Specifically, in BBRv2, the flightsize is not only limited by the $2 \cdot BDP$ cap, but also by an explicit flightsize parameter included in the model used by BBRv2, in order to explicitly reduce the queueing delay experienced by BBRv2 traffic. As in the case of BBRv1, rLEDBAT behaves similarly to LEDBAT++ and it inherits its limitations when competing with BBRv2.

The results of the experiments using a buffer of 30 ms are plotted in Fig. 19. In this case, we observe that BBRv2 yields in front of rLEDBAT

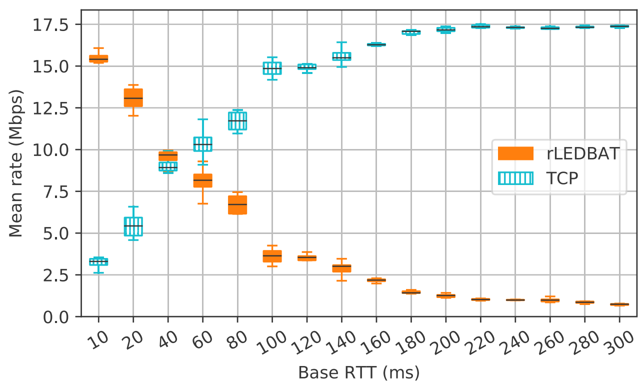


Fig. 19. rLEDBAT and BBRv2 in a 20 Mbps link with 30 ms of buffer, for different RTT values.

Table 3
Mean values for interaction with short-lived TCP flows for different Round Trip Times (RTT), BBRv2 server for TCP flows.

	20 ms	100 ms	200 ms	300 ms
TCP-BBRv2 short (solo)	0.17	0.67	1.33	1.99
rLEDBAT client, then TCP-BBRv2 short	0.54	0.96	1.61	2.02
TCP-BBRv2 long, then TCP-BBRv2 short	0.40	1.03	1.66	2.23

for RTTs smaller than the buffer size (30 ms) and for larger RTTs, the BBRv2 flow is able to seize more capacity. Again, this is the same behaviour observed for LEDBAT++ when competing with BBRv2 [11] under these conditions. The rationale for the observed behaviour is that for base RTTs smaller than the buffer size, the flightsize of the BBRv2 flow is not enough to fill in the buffer (because of BBRv2 inflight limitation) and leaves room for the rLEDBAT flow.

Finally, we measure the completion time of a short BBRv2 flow. Similarly to the experiments with BBRv1 and Cubic, we compare the completion time of a 262-KB BBRv2 flow in the following three scenarios, running solo, competing against a long-lived BBRv2 flow and a long lived rLEDBAT flow. Results are presented in Table 3. We observe a similar behaviour than in the case of BBRv1, only that the benefits of using rLEDBAT diminish. This is expected since BBRv2 is less aggressive than BBRv1.

7. Enhanced LEDBAT++ congestion control algorithm

Our experiments show that the proposed rLEDBAT mechanisms performs similarly to the original (sender-side) LEDBAT++. This includes also the pitfalls we have previously identified regarding LEDBAT++ performance when competing against BBR flows. Specifically, we have verified that neither LEDBAT++ nor rLEDBAT flows yield when competing against BBR flows if the RTT is smaller than the target T.

In this section, we propose simple modifications to the LEDBAT++ congestion control algorithm that correct the observed shortcomings. We implement the proposed modification in our rLEDBAT implementation, and we perform a number of experiments to show that the proposed solution indeed addresses the limitations identified for LEDBAT++ and rLEDBAT when competing against BBR.

7.1. Proposed modifications

In our previous analysis, we posit that BBR flows are unable to seize all the available capacity when competing against (r)LEDBAT(++) flows due to its self-imposed limitation on the flightsize. This limitation is set to one additional BDP, and it is in place to limit the size of the resulting queue. When the RTT is smaller than the target T, the queue generated by the (r)LEDBAT(++) flow is larger than the one BBR is willing to generate, hence BBR yields.

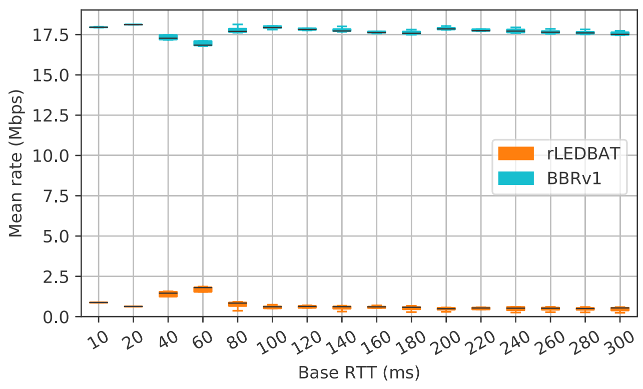


Fig. 20. Modified rLEDBAT and BBRv1 in a 20 Mbps link with different RTTs, buffer set to 500 ms.

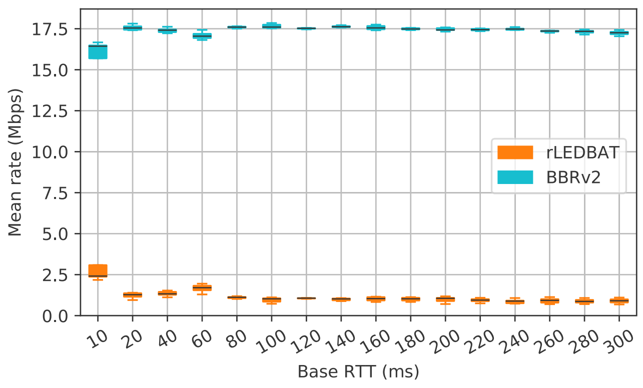


Fig. 21. Modified rLEDBAT and BBRv2 in a 20 Mbps link with different RTTs, buffer set to 500 ms.

In order to address this issue, we propose to limit the queue generated by the (r)LEDBAT(++) mechanisms to the minimum of the target T and the base RTT. This should solve the problem since this would result in (r)LEDBAT(++) refraining from generating queues larger than one BDP, so, it would consequently yield in from of BBR. We believe this would also positive for (r)LEDBAT(++) itself, as it would prevent it from bloating the RTT in relative terms compared to the base RTT for small base RTTs.

8. Experiments with the enhanced LEDBAT++ congestion control algorithm

We implemented the proposed modifications in our rLEDBAT implementation and we repeated the experiments using the updated implementation. Our results show that the proposed enhancements address the identified shortcomings.

Specifically, in Figs. 20, 21, we plot the rate achieved by a rLEDBAT flow and a BBR flow competing for the capacity of a 20 Mbps bottleneck using different base RTTs, and we observe that rLEDBAT successfully yields in all scenarios, against both versions of BBR.

In Fig. 22 we verify that rLEDBAT performance is unaffected by the modifications when running solo in a bottleneck.

9. Related work

We discuss in this section some proposals that enable the receiver to control the rate at which data is downloaded.

Spring et al. [21] manipulate the Receive Window to prioritize flows across bottlenecks at access links. When a new flow starts or leaves, the size of the Receive Window for the remaining flows can

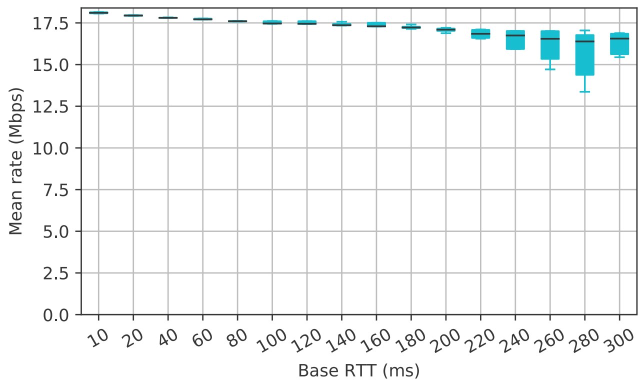


Fig. 22. Modified rLEDBAT running solo in a 20 Mbps link with different RTTs, buffer set to 500 ms.

be changed, either reducing an existing window or increasing it. To compute the amount of window to allocate, they assume known the bandwidth of the access link, which is expected to be the bottleneck for all the communications. The criteria to allocate buffer to applications is to keep delay low for interactive applications and reduce congestion loss by ensuring that few packets are queued in-flight. However, the link bandwidth control mechanism is centralized, as it requires full information and control over all the flows traversing the bottleneck, i.e., it is devised to be implemented in a single host. While this may be a reasonable assumption in certain scenarios, it does not hold when there are competing flows from different sources. rLEDBAT is capable of handling competing flows from different sources.

Mehra et al. [22] propose controlling the Receive Window and the delay of ACK messages to the sender to enforce the rate assigned to particular flows. This solution is suitable for multimedia streams or interactive applications. They measure the smallest rate that provides full link utilization, and distribute the flows to achieve a weighted bandwidth partition according to user preferences. As occurs for [21], the mechanism assumes that a single system manages the flows that traverse the bottleneck, which again is assumed to be close to the receiver.

Key et al. [23] aims to allocate the spare capacity left by best-effort traffic to background flows (less-than-best-effort traffic), and distribute the available capacity fairly between background flows. They formulate a joint control-estimation dual problem with the objective to adjust dynamically the Receive Window to the optimal operation point. Two algorithms are proposed to set Receive Window value, a binary search in which the window is doubled until a condition holds, and a stochastic approximation method which takes into account the presence of noise in the rate measured. The tradeoff between both is stability vs convergence time: the first approach fluctuates around a stable scenario, leading to reduced capacity utilization, while the second takes longer to converge (more than one hundred seconds in the experimental results shown). The main differences with rLEDBAT are that rLEDBAT control is built upon LEDBAT a well-known, widely tested algorithm, allowing to develop rLEDBAT implementation based on existing LEDBAT code. Besides, in rLEDBAT the requirements of delay-sensitive applications are considered by using delay as the main feedback source.

RTAC (Receiver-side TCP Adaptive queue Control, [24]) proposes the manipulation of the Receive Window to tackle with the bufferbloat problem in wireless access networks. Similarly to [23], they formulate a Network Utility Maximization problem. They use the length of the queue of the access link as the price of the problem. RTAC is implemented on Android devices, and the proponents conduct experiments over two different wireless access networks to show the feasibility of a solution based on the control of the Receive Window. The difference with our work is that they aim to allow the flows controlled by RTAC to compete fairly with existing TCP flows, while we aim to differentiate rates according to a less-than-best-effort policy.

10. Conclusions

We presented rLEDBAT, a mechanism to provide Less-than-Best-Effort (LBE) transport services for TCP communications, that is solely based on the receiver side of the communication. Being a receiver-based mechanism, rLEDBAT enables new use cases and new deployment models. In terms of use cases, it allows the client of the communication to manage its incoming traffic, for instance, a mobile device can select which of its incoming traffic is best-effort (e.g., video streaming) and which one is less-than-best-effort (e.g., backup in the cloud of local files). Regarding alternative deployment models, rLEDBAT can be enabled by solely changing the client, allowing for instance to enable LBE software update distribution through CDN surrogates that do not implement LBE transport services as long as the client implements rLEDBAT.

We have implemented rLEDBAT in Linux and we have experimentally tested that the proposed design achieves the aforementioned design goals. We find that rLEDBAT effectively implements a LBE transport at the receiver. rLEDBAT performs similarly than the sender-based LEDBAT++. It exhibits the same performance shortcomings that have previously reported for LEDBAT++, including its inability to seize all available bandwidth for large base RTTs (larger than 200 ms), and to yield when competing against BBR (both BBRv1 and BBRv2) in paths with base RTT smaller than the target T (60 ms).

We propose a modification to the (r)LEDBAT(++) algorithm to address the identified shortcomings. We implemented the proposed modifications in our rLEDBAT implementation and we experimentally validated that the modified algorithm does indeed address the identified pitfalls and yields in front of BBR (both BBRv1 and BBRv2).

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Marcelo Bagnulo reports financial support was provided by European Commission.

Data availability

Data will be made available on request.

Acknowledgements

This work has been partially supported by the EU EC through the STANDICT-CCI project, Grant 951972. Funding for APC: Universidad Carlos III de Madrid (Read Publish Agreement CRUE-CSIC 2022). All authors approved the version of the manuscript to be published.

References

[1] Praveen Balasubramanian, Osman Ertugay, Daniel Havey, LEDBAT++: Congestion Control for Background Traffic, draft-irtf-iccrg-ledbat-plus-plus-01, Internet Engineering Task Force, 2020, Work in Progress.
[2] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, R. Scheffenegger, CUBIC for Fast Long-Distance Networks, 8312 RFC Editor, RFC Editor, 2018.
[3] S. Shalunov, G. Hazel, J. Iyengar, M. Kuhlewind, Low Extra Delay Background Transport (LEDBAT), Request for Comments 6817, RFC Editor, 2012.
[4] P. Balasubramanian, LEDBAT++: Low priority TCP congestion control in windows, in: IETF Meeting 100, Singapore, 2017.
[5] G. Carofiglio, L. Muscariello, D. Rossi, S. Valenti, The quest for LEDBAT fairness, in: Global Communications Conference, 2010, GLOBECOM, 2010, pp. 1–6.
[6] Marcelo Bagnulo, Alberto Garcia-Martinez, An experimental evaluation of LEDBAT++, Comput. Netw. 212 (2022) 109036.
[7] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, Home network or access link? Locating last-mile downstream throughput bottlenecks, in: PAM 2016 - Passive and Active Measurement Conference, 2016, pp. 111–123.

[8] Nimantha Baranasuriya, Vishnu Navda, Venkata N. Padmanabhan, Seth Gilbert, QProbe: Locating the bottleneck in cellular communication, in: Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, in: CoNEXT15, Association for Computing Machinery, New York, NY, USA, 2015.
[9] Jim Gettys, Kathleen Nichols, Bufferbloat: Dark buffers in the internet, Queue 9 (11) (2011).
[10] A.M. Mandalari, M. Bagnulo, A. Lutu, Informing protocol design through crowdsourcing: The case of pervasive encryption, in: 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data, ACM, New York, NY, USA, 2015, pp. 3–8.
[11] Marcelo Bagnulo, Alberto García Martínez, When less is more: BBR versus LEDBAT++, Comput. Netw. (2022) 109460.
[12] R.T. Braden, Requirements for Internet Hosts - Communication Layers, Request for Comments 1122, RFC Editor, 1989.
[13] D. Borman, R.T. Braden, V. Jacobson, R. Scheffenegger, TCP Extensions for High Performance, Request for Comments 7323, RFC Editor, 2014.
[14] S. Protsenko, Print TCP packet data, 2017, https://stackoverflow.com/questions/29553990/print-tcp-packet-data/29584449.
[15] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Van Jacobson, BBR: Congestion-Based Congestion Control: Measuring Bottleneck Bandwidth and Round-Trip Propagation Time, Queue 14 (5) (2016).
[16] N. Cardwell, BBRv2 Implementation for Linux, 2022.
[17] Neda Beheshti-Zavareh, Tiny Buffers for Electronic and Optical Routers (Ph.D. thesis), Stanford University, 2009.
[18] Rajendra K. Jain, Dah-Ming W. Chiu, William R. Hawe, et al., A quantitative measure of fairness and discrimination, 1984, Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA, 21.
[19] Piotr Jurkiewicz, Grzegorz Rzym, Piotr Borylo, Flow length and size distributions in campus internet traffic, 2018, CoRR abs/1809.03486.
[20] Martino Trevisan, Danilo Giordano, Idilio Drago, Maurizio Matteo Munafo, Marco Mellia, Five years at the edge: Watching internet from the ISP network, IEEE/ACM Trans. Netw. 28 (2) (2020) 561–574.
[21] N.T. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, B. Bershad, Receiver based management of low bandwidth access links, in: IEEE INFOCOM 2000, Vol. 1, IEEE, 2000, pp. 245–254.
[22] P. Mehra, A. Zakhor, C. De Vleeschouwer, Receiver-driven bandwidth sharing for TCP, in: IEEE INFOCOM 2003, Vol. 2, IEEE, 2003, pp. 1145–1155.
[23] P. Key, L. Massoulié, B. Wang, Emulating low-priority transport at the application layer: A background transfer service, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 32, ACM, 2004, pp. 118–129.
[24] H. Im, C. Joo, T. Lee, S. Bahk, Receiver-side TCP countermeasure to bufferbloat in wireless access networks, IEEE Trans. Mob. Comput. 15 (8) (2016) 2080–2093.

Marcelo Bagnulo

Alberto García-Martínez

Anna Maria Mandalari works as Assistant Professor at University College Ldonod. Before, she was a research associate in the Dyson School of Design Engineering, at The Faculty of Engineering at Imperial College London. Over the last four years Anna was a METRICS Marie Curie Early Stage Researcher affiliated with the University Carlos III of Madrid (UC3M). Her research interests are related to IoTs, privacy, middleboxes, large-scale Internet measurements, Internet measurements platforms and new Internet protocols.

Praveen Balasubramanian is Director Of Engineering at Confluent. Previously, he was a Principal Development Lead in Core Networking for Microsoft. Praveen in active in the IETF having published an RFC and contributed to many working groups. Praveen has an MS in Computer Science from the University of Texas at Austin.

Daniel Mark Havey is the program manager for Windows networking Data Transports at Microsoft. He attended the University of California Santa Barbara starting in 2006 studying under Dr. Kevin Almeroth and writing a dissertation, "Latency and Bandwidth on the Packet Switched Internet" graduating with an MS/Ph.D. in 2015. Since graduation Daniel has been working on Microsoft specific transport and Web protocols. Daniel's current research interests are low latency transport protocols, Constant System Updates, and cooperative data transport.

Gabriel Montenegro is currently a Principal Engineer at Samsung Research America. Previously, he worked as Principal Program Manager in the Core Networking for Windows group at Microsoft since 2005. Gabriel is active in standards like IETF (past IAB member and working group chair and having co-authored several RFCs) and others (e.g., Wi-Fi Alliance, the USB Implementors Forum, the WiMAX Forum, IEEE 802). Gabriel obtained a BS in EE-Computers from Stanford University and an MS in information engineering from Niigata University in Japan.