Characterising LEDBAT Performance Through Bottlenecks Using PIE, FQ-CoDel and FQ-PIE Active Queue Management

Rasool Al-Saadi*†, Grenville Armitage* and Jason But*
*School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia
†Al-Nahrain University, Baghdad, Iraq
{ralsaadi, garmitage, jbut}@swin.edu.au

***Abstract*—Low Extra Delay Background Transport (LEDBAT) is defined in RFC6817 as a congestion control algorithm for lower than best effort transport service that reacts to both delay and loss congestion signals. LEDBAT allows bulk transfer applications (such peer-to-peer file transfer and software updates) to utilize available capacity in the background while limiting additional forward queuing delays at network bottlenecks to 100ms. New Active Queue Management (AQM) schemes similarly aim for low latencies by dropping or marking packets when the bottleneck queuing delay exceeds thresholds much lower than 100ms. Due to renewed interest in deploying modern AQMs on home broadband services, we experimentally evaluate and characterize the impact of placing PIE, FQ-CoDel and FQ-PIE variants of AQM in the path of flows generated by libutp (a widely deployed UDP-based LEDBAT implementation). We uncover, and propose solutions to, some differences between libutp and RFC6817 that lead to poor utilization and incorrect inter-flow capacity sharing over AQM bottlenecks.**

***Index Terms*—LEDBAT, AQM, libutp, CoDel, PIE**

I. Introduction

Many latency-tolerant applications utilise the Transmission Control Protocol (TCP) for reliable, byte-stream data transport that dynamically adapts to, and makes use of, changing network capacity. However, standard TCP's capacity probing strategy induces potentially-significant queuing delays on all traffic sharing the first-in first-out (FIFO) drop-trail buffers of a conventional router or switch. This creates problems for applications that require low latency for optimal end-user experience (such as multimedia conferencing and multi-player games). The problem worsens when bottlenecks suffer from Bufferbloat [1] (buffering in excess of traffic requirements).

Sometimes referred to as *scavenger class* transport, Low Priority Congestion Control (LPCC) transport protocols have been developed as an alternative to TCP for latency-tolerant applications (such as peer-to-peer file sharing applications) that are willing to take a lower bandwidth share than competing TCP-based traffic on latency-sensitive applications. Some LPCC go further, and aim to cap the bottleneck queuing delays they induce when only sharing with latency-senstive traffic [2].

Low Extra Delay Background Transport (LEDBAT, described in RFC6817 [3]) is a widely deployed, delay-based LPCC that aims to keep forward queuing delay no more than 100ms to minimise interference with other flows. Many popular BitTorrent peer-to-peer file sharing clients use `libutp` [4][1] to provide a UDP-based transport protocol with LEDBAT congestion control [5], [6]; Apple Inc. implemented TCP-based LEDBAT for operating system updates [7]; and Microsoft have LEDBAT in beta-trials for Windows 10 [8].

A complication for LEDBAT is the emergence of Proportional Integral controller Enhanced (PIE) [9], Controlled Delay (CoDel) [10], FlowQueue-CoDel (FQ-CoDel) [11] and FlowQueue PIE (FQ-PIE) [12] Active Queue Management (AQM) schemes. They target loss-based TCP flows by dropping packets when bottleneck queuing delay reaches specified levels well under 100ms, causing TCP to back-off earlier than would occur with FIFO queues. The potential for significant reduction in overall round trip time (RTT) is motivating deployment of these new AQMs at either end of shared, home broadband last-mile services.

However modern AQM actively minimises the delay signal that LEDBAT uses to infer congestion, pushing LEDBAT out of its desirable delay-based congestion control behaviour into more conventional loss-based behaviour.

In this paper we demonstrate and solve the negative impact of ambiguities in RFC6817's definition of LEDBAT, and certain interpretations of RFC6817 by `libutp`, when LEDBAT flows must react primarily to packet losses in low-delay environments. We propose and evaluate a specific enhancement to `libutp`'s use of selective acknowledgements (SACK) to expedite recovery from lost retransmissions. This highlights the importance of transport protocols being specified and implemented carefully for a wide range of network conditions.

The rest of this paper is structured as follow. Section II introduces LEDBAT; summarises the modern AQM algorithms; outlines prior work on LEDBAT with older AQMs and detecting lost retransmissions. Section III introduces our experimental test environment while Section IV introduces flow unfairness and stalling issues we have discovered, along with our proposed solutions. Implications are discussed in Section V and we conclude in Section VI.

II. Background

Here we cover LEDBAT's key characteristics compared to regular TCP, summarise modern AQM schemes that are

[1]A library implementing LEDBAT for the uTorrent Transport Protocol (uTP).

likely to appear in consumer last-mile services, note prior evaluation of LEDBAT's interaction with older AQM schemes, and summarise prior work on prompt detection of loss of retransmitted packets.

A. Low Extra Delay Background Transport (LEDBAT)

LEDBAT infers network congestion when forward queuing delays increase, and early response to rising delay ensures LEDBAT flows typically achieve a lower share of bandwidth than competing TCP flows. LEDBAT has a default *target* of adding no more than 100ms to a path's forward one-way delay (OWD). LEDBAT uses timestamps in each packet to estimate instantaneous forward path one-way delay (OWD_i), so reverse path delay fluctuations do not affect forward path congestion estimation.

As with standard TCP, LEDBAT's congestion window (*cwnd*) caps the number of unacknowledged bytes in flight. In the absence of packet loss, *cwnd* is increased or decreased according to Equation 1, where Δ (Equation 2) is a normalised difference between one-way queuing delay and *target*:

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

$$\Delta = \frac{(target + OWD_{base} - OWD_i)}{target} \tag{2}$$

MSS is maximum segment size, G is the gain scale which determines *cwnd* growth/decline speed (G<1 to make the algorithm less aggressive than standard TCP), and B_{acked} the number of newly acknowledged bytes. OWD_{base} is the minimum OWD_{min} over the last 10 minutes (RFC6817) or 13 minutes (`libutp`), where OWD_{min} is the lowest OWD_i over each one minute period. This smooths the rate of changes in OWD_{base} that could otherwise be caused by delayed ACK, clock skew or re-routing problems. The difference $OWD_q = (OWD_i - OWD_{base})$ estimates the one-way *queuing* delay between sender and receiver.

In response to this delay signal, LEDBAT increments *cwnd* by a maximum of $G \times MSS$ every RTT when $OWD_q = 0$, uses smaller and smaller increments as OWD_q approaches the *target*, and decrements *cwnd* when $OWD_q > target$. In the absence of competing loss-based traffic, LEDBAT flows will stabilise *cwnd* around the point where $OWD_q = target$.

Equation 1 allows LEDBAT to rapidly increase *cwnd* when no congestion is detected while producing low impact on the network when congestion starts to build. This eliminates a dependence on packet losses (and the resulting multiplicative backoffs) while probing for capacity.

When a LEDBAT flow competes with a loss-based (delay-insensitive) flow (such as TCP NewReno) over a conventional FIFO bottleneck, the LEDBAT flow will observe high OWD_q caused by the delay-insensitive flow. LEDBAT will reduce its own *cwnd* once OWD_q exceeds *threshold*, effectively deferring to the regular TCP flow as intended.

In response to packet loss, RFC6817 specifies that LEDBAT halves *cwnd* no more than once per RTT.

B. CoDel, PIE, FQ-CoDel and FQ-PIE

CoDel and PIE are single-queue AQM schemes that aim to control queue delay efficiently while passing short traffic bursts and preserving high long-term link utilisation without the need for parameter tuning.

CoDel distinguishes between bad (persistent) and good (temporary) queues based on the local minimum time spent in the queue (packet sojourn time, $T_{sojourn}$). CoDel enters a *drop state*, in which packets are dropped or marked from the queue's head, when $T_{sojourn}$ remains above T_{target} (default 5ms) for a nominated time $T_{interval}$ (default 100ms). The next drop time is reduced exponentially and another packet is dropped if $T_{sojourn}$ continues to remain above T_{target}; otherwise it exits the drop state. A small T_{target} keeps queuing delay low, while a $T_{interval}$ larger than maximum RTT of flows allows the bottleneck to drain short bursts of packets from its queue.

PIE drops/marks packets on ingress based on a regularly-updated drop probability. Periodically (every 15ms by default), the drop probability is calculated based on the weighted deviation of queue delay from T_{target} (default 15ms), and weighted queue delay trend. PIE calculates the queueing delay by dividing the current queue length by the estimated departure rate. To allow short-lived bursts of packets, PIE ignores the drop probability on packet arrival for a short time (default 150ms) after an idle period.

PIE provides further mechanisms to improve overall performance including 1) turning AQM on and off automatically based on congestion level; 2) ECN threshold (10% default) to protect the queue from unresponsive ECN enabled flows; and 3) drop de-randomisation to prevent too close or too far packet drop. Drop de-randomisation accumulates the drop probability during packet en-queuing and only drops packets when the accumulated probability exceeds a threshold.

FQ-CoDel (FlowQueue CoDel) [11] is a hybrid scheduler-AQM scheme that controls queueing delay and provides relatively fair sharing of the bottleneck capacity. The "FlowQueue" scheduling part of FQ-CoDel protects flows from each other by classifying and hashing the flows into one of 1024 queues (default) and serving them using the Deficit Round Robin (DRR) algorithm. The FlowQueue scheduler provides a short period of priority to lightweight flows (such as DNS queries) to increase overall network response by maintaining lists of new and old queues, where queues in the new queues list have higher priority than those in the old queues list. Latency is controlled in each queue separately using independent instances of CoDel AQM.

FQ-PIE (FlowQueue PIE) is a recent hybrid scheduler-AQM scheme implemented in the FreeBSD operating system [12]. Like FQ-CoDel, this AQM hashes flows to one of N queues, each independently managed by an instance of PIE, and also uses FQ-CoDel's FlowQueue strategy to provide priority to new, short-lived flows. FQ-PIE provides low queueing delay and relatively fair capacity sharing between competing flows.

When a single flow passes through an FQ-CoDel or FQ-PIE bottleneck, the scheduler assigns only one queue and their behaviour is the same as a single flow passing through a CoDel and PIE bottleneck respectively.

C. LEDBAT Over an AQM Managed Queue

Rossi et. al [5] used ns2 simulation to confirm that LEDBAT flows through a FIFO bottleneck with a large, drop-tail queue get a (desirable) lower bandwidth share than TCP flows, and when LEDBAT responds to loss over this same FIFO bottleneck it shares bandwidth equally with TCP flows.

However, any AQM that keeps queuing delays under LEDBAT's default *target* of 100ms will deny LEDBAT flows a sufficient OWD signal with which Equation 1 can control *cwnd*. For example, CoDel's dropping of packets that exceed 5ms of queuing delay (after CoDel's burst tolerance period) means a LEDBAT flow's *cwnd* will be controlled by packet loss signals well before Equation 1 itself causes any *cwnd* reduction. Under such circumstances RFC6817 aims for LEDBAT to behave no more aggressively than a standard loss-based TCP.

RFC6817 also notes that flow-isolating packet schedulers, such as Weighted Fair Queueing (WFQ), provide further protection for latency sensitive flows. Although these outcomes are in line with LEDBAT goals, RFC6817 notes that further study is required to fully understand the impact of AQM on LEDBAT convergence properties.

Gong et. al [13] explore the coexistence of five AQM techniques with different LPCC including LEDBAT, through the use of ns2 simulation and experiments. They found that AQM re-prioritises LEDBAT flows, resulting in almost equal bandwidth sharing between TCP and LEDBAT flows. While [13] extensively analyses the effect of AQM on LPCC algorithms, it does not experimentally explore the impact of the path RTT on LEDBAT flows nor consider modern AQMs.

D. Prompt detection of loss of retransmitted packets

In Section IV we show LEDBAT flows over AQM bottlenecks can stall frequently due to losses of *previously retransmitted* packets. Such losses are, for example, a known issue in TCP over high error rate wireless environments [14], [15], and we summarise some prior solutions here.

Duplicate Acknowledgement Counting (DAC) [16] uses *cwnd* at the moment of fast retransmission to estimate the number of packets in-flight at the time, and hence the maximum number of duplicate ACKs (dupACKs) we should receive for a lost packet. DAC immediately resends the previously retransmitted packet if more than the expected number of dupACKs are received.

Detecting and Differentiating the Loss of Retransmitted Packets (DDLRP) [17] improves on DAC by estimating the packets in flight at the moment of packet loss detection using the difference between the highest sequence number sent and the dupACKs of the lost packet. As with DAC, DDLRP resends the previously retransmitted packet if more than the expected number of dupACKs are received.

Sreekumari et al. [15] propose a modification of RFC 1323's TCP timestamp option processing [18] such that the receiver updates the echo reply field even during loss recovery. The sender logs the timestamp of the retransmitted packet at the start of fast recovery. This retransmitted packet is considered lost, and resent again, upon receipt of a dupACK whose echoed timestamp is greater than or equal to the stored timestamp.

Retransmission Packet Loss Detection (RPLD) [14] uses TCP's *Selective Acknowledgement* option (SACK) [19] to infer retransmissions losses. SACK enables a sender to learn the specific sequence numbers received by the destination after a lost packet. During fast retransmission, RPLD stores the first sequence number and transmission time for the retransmitted and subsequent packets, marking them as either retransmitted or new. When further dupACKs are received containing SACK information, the list of stored sequence numbers and transmission times is traversed to determine if any successfully received packets (indicated in the SACK) were sent after any retransmitted packets were sent. If this is the case, the retransmitted packet is considered lost, and resent again.

Key concepts in RPLD also turn up in RACK ("Recent ACKnowledgment") [20]. RACK aims to efficiently utilise SACK information and time-stamps in TCP packets to expedite detection and recovery from all packet losses.

To varying degrees these proposed solutions need modification to TCP options, are unable to detect multiple losses or add more complexity to the transport protocol's implementation.

III. Test environment

We used a TEACUP-based [21] testbed to explore the impact of using PIE, FQ-CoDel and FQ-PIE on commit 3110314 (2016) of `libutp`. Figure 1 shows our testbed's network topology, with two hosts (Intel Core 2 Duo @ 3GHz, Linux 4.9), one bottleneck router (Intel Core 2 Duo @ 2.33GHz, Linux 3.17.4 or FreeBSD 11-stable), a control host and gigabit Ethernet links. Experiment traffic flows from Host 2 $\rightarrow$ Bottleneck $\rightarrow$ Host1.

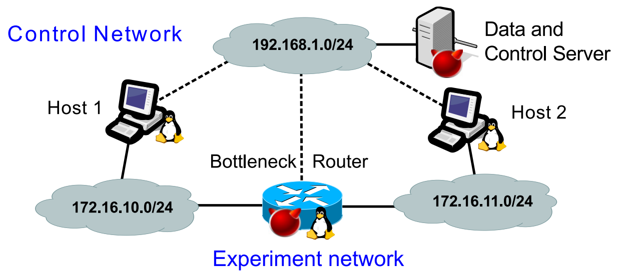


Fig. 1. Testbed topology: Traffic flows from Host 2 $\rightarrow$ Bottleneck $\rightarrow$ Host1

To emulate our path's unloaded RTT (RTT_{base}) and configurable bottleneck bandwidth (B_{rate}), the bottleneck router uses `netem` and `tc` modules in Linux (section IV-B of [21]) when using PIE or FQ-CoDel, and Dummynet and IPFW in FreeBSD ([22]) when using FQ-PIE for specific experiments. We use the default 1000-packet bottleneck buffer for each AQM. We do not test LEDBAT over CoDel as CoDel's authors themselves recommend using CoDel in the context of FQ-CoDel instead of as a standalone AQM.

FreeBSD's PIE and FQ-PIE are based on the latest PIE specification in RFC8033 [9] while the Linux PIE implementation is based on an earlier PIE Internet-Draft [23]. Nevertheless, our PIE experiments use a Linux bottleneck, as Linux is commonly deployed in embedded systems such as home gateways. This makes our observations more likely to match the experience of end users. (The main difference is that RFC8033 is more aggressive in dropping packets, due to specifying a shorter 15ms for both T_{target} and drop probability update interval. Single flow scenarios using FreeBSD PIE or FQ-PIE would likely show slightly different results than with Linux PIE.)

We patched `libutp` to gather detailed internal state (such as current congestion window size and smoothed RTT).

IV. Unbalanced bandwidth consumption and stalling flows

Two problems emerged when `libutp`-generated flows pass through an AQM bottleneck. `libutp`-generated flows can exhibit *inter-flow bandwidth unfairness* by incorrectly consuming more bandwidth than competing standard TCP flows, and regularly stall when passing through a AQM-managed bottleneck under low-to-modest RTT and bandwidth conditions that easily occur in home user contexts.

A. Improving `libutp` for low RTT paths

Our analysis uncovered the following issues with `libutp`'s interpretation of RFC6817:

1) Does not backoff *cwnd* more frequently than once every 100ms regardless of RTT.
2) Equation 1's $G \times MSS$ term was effectively set to 3000 bytes, allowing *cwnd* growth by roughly two MSS rather than one MSS per RTT (faster than it would with standard TCP).
3) Enforces a minimum RTO (Retransmission Time-Out) of one second, and relies on this RTO to recover from loss of retransmissions.

We implemented `libutp-mod` [24] to demonstrate the impact of fixing these issues by modifying `libutp` as follows:

1) Use `libutp`'s existing smoothed RTT estimate ($RTT_{smoothed}$) to allow *cwnd* to backoff once per RTT (per RFC6817) on packet loss.
2) Modify Equation 1's $G \times MSS$ term to ensure *cwnd* grows by only one MSS per RTT.
3) Allow lost retransmissions to themselves be retransmitted long before an RTO is triggered.

Items #1 and #2 are implementation choices that diverge from clear directives in RFC6817. Item #3 relates to how `libutp` detects the loss of a previously retransmitted packet.

RFC6817 does not specify a mechanism for recovery from lost retransmissions, so `libutp` relies on an RTO firing to detect this case and initiate another retransmission, leading to one-second stalls. When LEDBAT flows run over traditional FIFO bottlenecks, the probability of packet drop (in general) and retransmission drop specifically are very low except when induced by competing loss-based traffic. However, in AQM environments LEDBAT packets are frequently and regularly dropped as the AQM tries to control queuing delays well before LEDBAT's own delay *threshold* is reached.

The choice of minimum RTO is not well specified by RFC6817. Simply choosing a smaller minimum RTO value (200ms for example) can reduce `libutp`'s stall time and improve throughput, but does not properly solve the problem.

B. Rapid LEDBAT loss recovery using SACK

Our solution is based on RPLD (Section II-D), optimised for `libutp`'s implementation of SACK (not specified by RFC6817). `libutp`'s initial loss detection and fast retransmission methods are unaltered.

`libutp` acknowledges packets rather than bytes and uses a 32-bit to 128-bit *bitmask* to selectively acknowledge non-sequential arrival of packets. SACK information is only sent if at least one packet is received out of sequence. Given $(dupACK_{seqnum} + 1)$ is the sequence number of the packet deemed lost (P_{lost}), the receiver sets the N^{th} bit in the bitmask of subsequent SACK fields to indicate additional receipt of packet $(dupACK_{seqnum} + 1 + N)$. Each SACK thus updates the sender regarding which of the N packets sent *after* the lost packet also have, or have not, been successfully received.

`libutp` already tracks the most recent transmission time, and number of times sent, of unacknowledged packets. In `libutp-mod` we introduce a new sender-side variable, $maxTA$, to track the transmission time of the most recently sent packet that is known (by ACK or indirectly by SACK bitmask) to have been successfully received. Using the following strategy, `libutp-mod` rapidly detects when a *retransmitted* packet has also been lost and needs to be retransmitted again.

As usual, three dupACKs trigger the first retransmission of P_{lost}. The SACK field of subsequent dupACKs relating to P_{lost} tells the sender about newly received packets (potentially updating $maxTA$) and packets newly declared to be missing (in addition to P_{lost}). If $maxTA$ is greater than the transmission time of any missing packet that has already been retransmitted at least once, we assume the retransmission has been lost and the missing packet is retransmitted again.

C. Inter-flow bandwidth fairness

Figure 2 shows the inter-flow bandwidth unfairness observed when one CUBIC TCP flow and one `libutp`-generated flow share a path with $RTT_{base} = 60ms$, $B_{rate} = 10Mbps$ and PIE, FQ-CoDel or FQ-PIE queue management for 120 seconds. A `libutp`-generated flow consumes a larger share of bandwidth than the CUBIC flow when using PIE, contrary to LEDBAT's goal of being a background or scavenger-class transport. In contrast, our `libutp-mod`-generated flow shares bandwidth much more equitably with the CUBIC flow. The FlowQueue scheduler of FQ-CoDel and FQ-PIE tends to normalise the bandwidth sharing, regardless of whether the LEDBAT flow is generated by `libutp` or `libutp-mod`.

`libutp`'s inter-flow bandwidth unfairness is due to (a) its faster cwnd growth per RTT, and (b) its refusal to reduce cwnd more than once per 100ms even when additional packet drops are detected. The latter point is an issue because modern AQM bottlenecks can lead to paths with RTTs consistently under 100ms – the CUBIC flow is disadvantaged by backing off multiple times over the same time period.

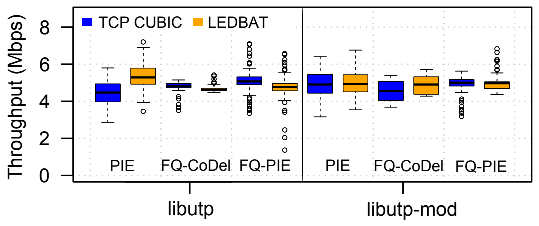


Fig. 2. Throughput of competing LEDBAT and TCP CUBIC flows across PIE, FQ-CoDel and FQ-PIE AQM ($RTT_{base} = 60ms$, $B_{rate} = 10Mbps$)

D. Stalling at $RTT_{base} < 40ms$ and $B_{rate} < 10Mbps$

Stalling of single and multiple `libutp`-generated flows is illustrated by running one, three or five LEDBAT flows for 120 seconds through FIFO, PIE, FQ-CoDel, and FQ-PIE bottlenecks while varying RTT_{base} from 2ms to 40ms and B_{rate} from 250Kbps to 10Mbps in steps of 250Kbps. With a FIFO bottleneck, we saw 100% link utilisation across the parameter range (not shown to save space). Figure 3 shows that with PIE, FQ-CoDel and FQ-PIE, link utilisation results were rather poor. By contrast, Figure 6 shows `libutp-mod`-generated flows achieving far more consistent utilisation across the same range of network conditions.

1) Link utilisation with `libutp`-generated flows: Figure 3a shows noticeable peaks and troughs in a single-flow's link utilisation using FQ-CoDel at certain combinations of low B_{rate} and RTT_{base}, with uniformly bad utilisation when $RTT_{base} \leq 6ms$. When $RTT_{base} = 16ms$ we require $B_{rate} \geq 4Mbps$ for consistently high utilisation, while with $RTT_{base} = 20ms$ we need $B_{rate} \geq 2.5Mbps$ for consistent utilisation. Once RTT_{base} is 40ms or beyond (not shown) we see good utilisation at $B_{rate} < 4Mbps$. The gradual drop-off in utilisation at $B_{rate} \geq 4Mbps$ occurs when path bandwidth-delay product (BDP) exceeds the bottleneck's effective buffer space (discussed further in Section IV-E). Figures 3d and 3g show improved overall link utilisation as the number of concurrent flows increases, because multiplexing reduces the impact of individual flows stalling after retransmission losses.

Figure 3b shows moderate reduction in a single-flow's link utilisation when using PIE AQM. The utilisation decline is most noticeable where $RTT_{base} \leq 40ms$ and $1Mbps < B_{rate} < 5Mbps$. As RTT_{base} and B_{rate} increase the link becomes better utilised. Similar to the FQ-CoDel case, Figures 3e and 3h show overall link utilisation improving with additional concurrent flows.

Figures 3c, 3f and 3i show that using an FQ-PIE bottleneck results in similar degradation in link utilisation as using a PIE bottleneck. Again, the overall link utilisation becomes higher with additional concurrent flows.

2) Underlying problem for `libutp`-generated flows: Two things combine to cause the loss of link utilisation seen in Figure 3. Each AQM keeps bottleneck queuing delay well below LEDBAT's 100ms threshold, forcing `libutp` to primarily respond to loss where it stalls when relying on a one-second RTO to recover from loss of previous retransmissions. Since the AQMs themselves regularly drop packets, this increases the probability of a retransmission being lost and (consequently) traffic stalling.

Figure 4 illustrates the potential for stalls that underlies Figures 3a, 3b and 3c. Here we run a single flow over a path where $RTT_{base} = 2ms$ and $B_{rate} = 2Mbps$, and the bottleneck is FQ-CoDel, PIE or FQ-PIE. Each graph shows the probability (D_p^{th}) of the N^{th} data packet after a packet loss also being lost ("Drop" curve) and the probability (R_p^{th}) of the N^{th} data packet after a loss being the retransmission of the lost packet ("Retransmission" curve).

In Figure 4a we see that for FQ-CoDel a retransmitted packet arriving 10 packets after the original drop has a higher than usual chance of also being dropped. This pattern of packet drop happens because CoDel (the AQM managing the single-flow's FQ-CoDel queue) has a long non-drop phase followed by an aggressive short drop phase, while the retransmission pattern depends on path BDP. If the short drop phase covers the fast retransmission period, there is a high probability for the retransmitted packet to be dropped, causing RTO to be triggered. Such behaviour leads to the dramatic loss in utilisation shown in Figure 3a.

The less dramatic loss of utilisation seen in Figures 3b and 3c can be understood through Figures 4b and 4c, which show less dramatic peaks in R_p^{th} around $N = 10$ and a lower, flatter D_p^{th} for both PIE and FQ-PIE. With a default 20ms[2] threshold, PIE drops packets less aggressively overall, and drops packets more uniformly (based on drop probability) over time, diluting the probability of both original and retransmitted packets being hit. Although FQ-PIE with a single-flow is essentially a single PIE queue, we can see that FreeBSD's more up-to-date PIE includes drop de-randomisation which results in very low D_p^{th} for $N < 5$.

3) Impact of `libutp-mod`: Using the progression of a flow's acknowledgement number over a ten second period, Figure 5 shows the `libutp` flow regularly stalling for one second, a `libutp` flow modified to allow 200ms minimum RTO regularly stalling for 200ms, and a `libutp-mod` flow practically eliminating the stalls. Figure 6 shows how `libutp-mod` improves on the poor performance seen in Figure 3. (The results for three flows are same as for five flows, and hence not shown to save space.)

E. Why utilisation degrades at higher path BDP

We next discuss why for the $RTT_{base} = 40ms$ scenarios, Figure 6a shows a `libutp-mod` flow's utilisation degrading when $B_{rate} > 2Mbps$ whilst Figure 3a shows a `libutp` flow's utilisation only starts to degrade for $B_{rate} > 4Mbps$.

We know that 100% link utilisation requires a transport protocol to keep enough bytes in-flight to equal or exceed the path's BDP ('fill the pipe'). Since LEDBAT halves *cwnd* after detecting packet loss, for continuous 100% utilisation across congestion epochs we need *cwnd* to be no less than BDP after halving. In other words the bottleneck queue ought to have absorbed at least a BDP of additional bytes before

[2]Linux PIE is based on the older specification with a larger threshold

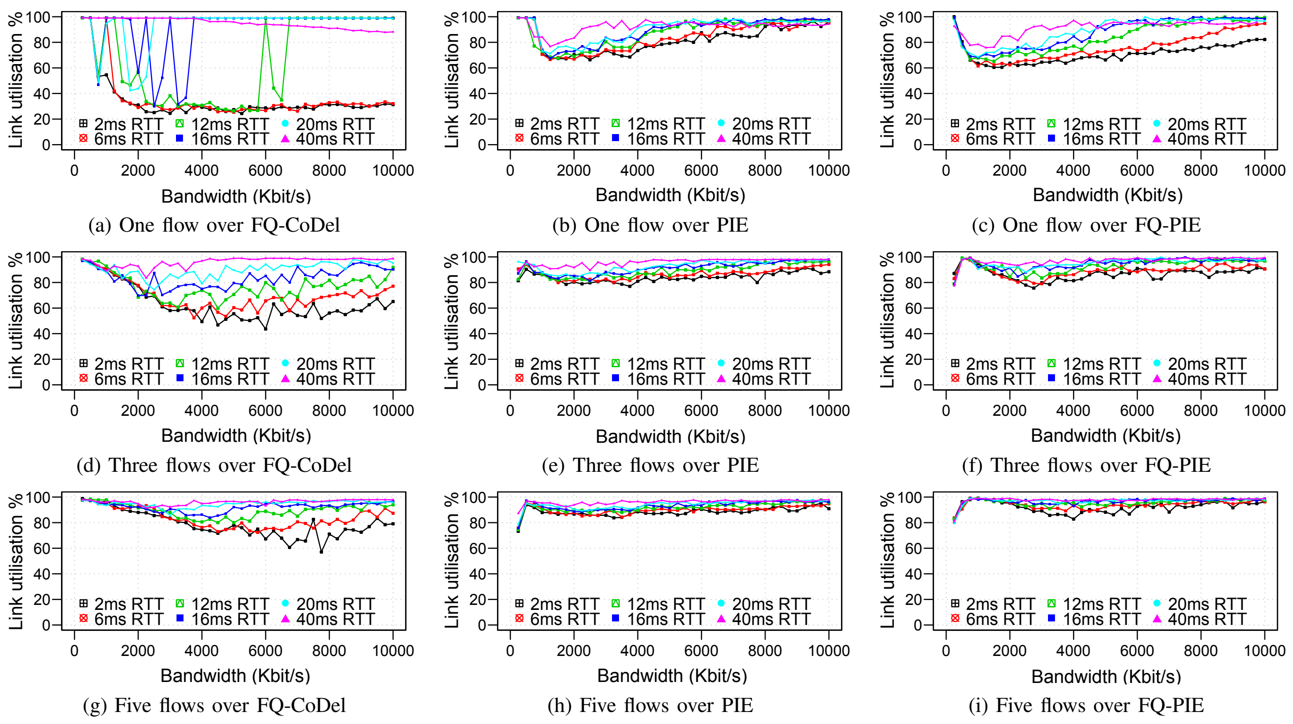


Fig. 3. Link utilisation of `libutp`-generated LEDBAT flows over FQ-CoDel, PIE and FQ-PIE bottlenecks

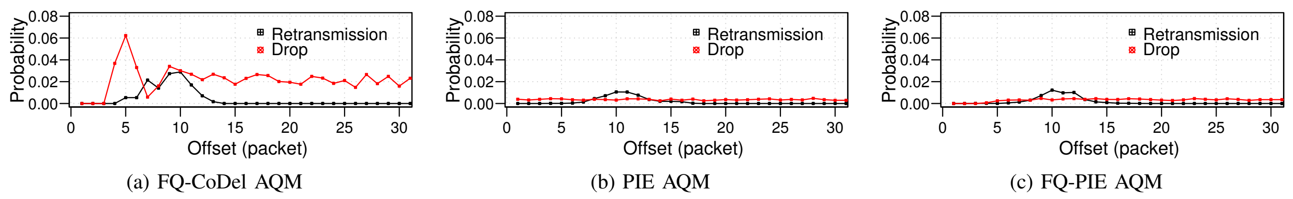


Fig. 4. Probabilities of N^{th} packet being a retransmission and N^{th} packet being dropped (influencing the loss of retransmitted packets)

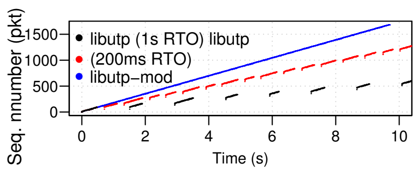


Fig. 5. LEDBAT sequence number versus time – FQ-CoDel

dropping a packet. However, PIE and CoDel emulate small queues over long periods of time (after their short post-idle burst-tolerant periods). A LEDBAT flow's link utilisation will thus drop below 100% when path BDP exceeds the effective size of the AQM queue at the time of each packet drop.

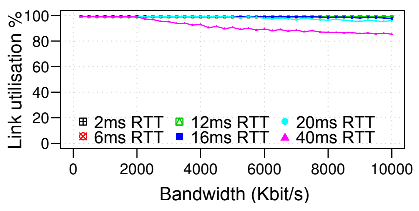
The degree of lost utilisation depends on how long it takes *cwnd* to regain and exceed BDP after packet loss. More relevantly, the BDP (and hence bottleneck speed, for a given RTT_{base}) at which this phenomenon becomes evident depends on the value of *cwnd* at the point of packet loss.

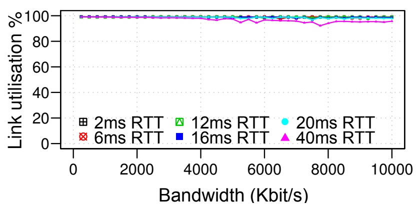
Figure 7 provides a close-up view of the utilisation loss as a function of BDP in Figure 6a. When $RTT_{base} = 40ms$ we clearly see utilisation drop off for $B_{rate} \geqq 2.25Mbps$, and as the BDP gradually increases the utilisation decreases to 85% at $B_{rate} = 10Mbps$. We can also see similar utilisation drop off starting at higher B_{rate} for the lower $RTT_{base} = 20ms$ and $RTT_{base} = 16ms$ cases.

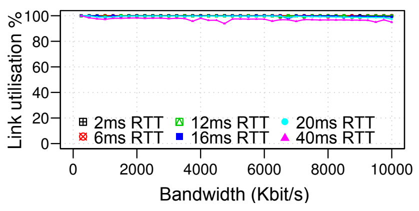
In Figure 3a, when $RTT_{base} = 40ms$ the equivalent utilisation drop-off starts occuring at at higher BDP, $B_{rate} > 4Mbps$. This difference is because `libutp`'s $G \times MSS = 3000$ creates faster *cwnd* growth every congestion epoch than occurs with regular TCP and `libutp-mod`.

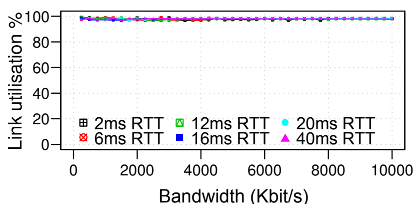
Figure 8 compares *cwnd* versus time for a single `libutp` or `libutp-mod` flow over a path where $RTT_{base} = 40ms$, $B_{rate} = 3Mbps$ and bottleneck uses FQ-CoDel. The `libutp` flow's first packet is dropped at 400ms when *cwnd* reaches 25KiB, while the `libutp-mod` flow's first packet is dropped at 600ms when *cwnd* reaches 22KiB. For every subsequent congestion epoch the `libutp` flow's *cwnd* continues to peak at a higher value than the `libutp-mod` flow's *cwnd*.

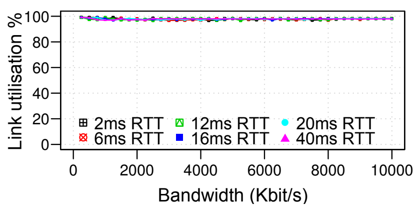
So for BDPs high enough that a `libutp` flow over FQ-CoDel avoids regularly stalling, it will achieve slightly higher utilisation than a `libutp-mod` flow under equivalent circumstances. Rather than being a weakness, `libutp-mod` is doing a better job of being no more aggressive than regular TCP.


(a) One flow over FQ-CoDel


(b) One flow over PIE


(c) One flow over FQ-PIE


(d) Five flows over FQ-CoDel


(e) Five flows over PIE

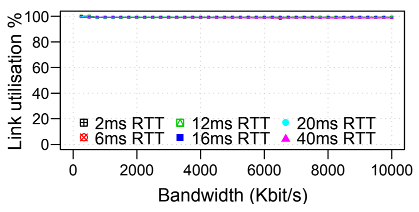

(f) Five flows over FQ-PIE

Fig. 6. Link utilisation of `libutp-mod`-generated LEDBAT flows over FQ-CoDel, PIE and FQ-PIE bottlenecks

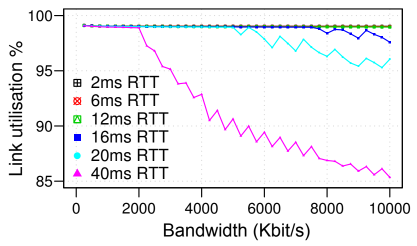


Fig. 7. Close-up of `libutp-mod` flow's utilisation degradation when path BDP exceeds the bottleneck buffering

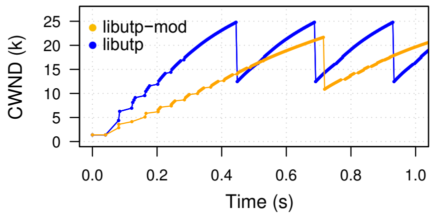


Fig. 8. Comparison of cwnd growth and back-off for `libutp` and `libutp-mod` flows, influencing cwnd after back-off

V. Discussion

Here we summarise our analysis, and consider the consequences of transitioning from `libutp` to `libutp-mod`.

A. *RFC6817 vs Implementations vs AQM*

LEDBAT simultaneously combines delay-sensitive (*cwnd* rising and falling according to Equation 1) and loss-sensitive (halving *cwnd* upon packet loss) congestion control behaviour. A LEDBAT flow induces no more than 100ms of queuing delay, and effectively side-lines itself if queuing delay is pushed over 100ms by competing, delay-insensitive traffic.

An underlying assumption is that bottlenecks have more than 100ms of potential buffering, and packet losses are usually triggered by competing loss-based flows. We have identified issues arising when modern AQMs such as FQ-CoDel, PIE and FQ-PIE impose frequent packet losses on LEDBAT flows while forward queuing delays are $\ll 100ms$.

We tested with path RTTs and sub-10Mbps bottleneck speeds common for consumer internet upstream access, and observed `libutp` stalling often and achieving suboptimal throughput. The root cause is RFC6817's reliance on an RTO to handle loss of retransmitted packets combined with `libutp`'s one second minimum RTO. Our `libutp-mod` successfully demonstrates a minimally-invasive solution building on the simple SACK mechanism already present in `libutp`. Future revisions of RFC6817 ought to address expedited handling of lost retransmissions.

`libutp` grows *cwnd* more aggressively than RFC6817 recommends (by using $G \times MSS = 3000bytes$ in Equation 1), and reduces *cwnd* less aggressively by limiting changes to once every 100ms. This has little downside when operating through large FIFO queues, as `libutp` stabilises at 100ms forward queuing delay or defers to competing TCP flows. But when competing over a single-queue AQM like PIE, `libutp` fails to defer to a regular TCP flow. These two implementation choices should be revisited.

B. *Transition Considerations*

We envisage two transitions – home internet services moving from FIFO to AQM bottlenecks, and `libutp`-based applications upgrading to the enhancements in `libutp-mod`. Given uncoordinated real-world software updates, we expect periods where `libutp` and `libutp-mod` flows from different devices compete with each other and/or regular TCP flows over either FIFO or AQM bottlenecks.

Additional experiments with a FIFO bottleneck (not shown to save space) revealed the following outcomes: (a) `libutp-mod` flows correctly defer to competing loss-based TCP flows; and (b) when `libutp` and `libutp-mod` flows compete in the absence of loss-based flows, the `libutp-mod` flow receives a *lower* long-term throughput share proportional to the ratio of their respective $G \times MSS$ terms. Experiments showed similar throughput unfairness exists when `libutp` and `libutp-mod` flows compete through a PIE bottleneck.

`libutp-mod`'s deferal to `libutp` under these circumstances seems an acceptable trade-off. It is consistent with

LPCC goals, and avoids the non-LPCC behaviour of using $G \times MSS = 3000 bytes$ when competing with loss-based TCP over single-queue AQMs (Figure 2). Furthermore, the G term's impact is negated when `libutp` and `libutp-mod` flows compete through FQ-CoDel or FQ-PIE bottlenecks.

Both `libutp` and `libutp-mod` flows exhibit the 'latecomer advantage' [2] over FIFO bottlenecks, unavoidably due to LEDBAT's infrequent recalculation of OWD_{base}. While `libutp`'s interpretation of RFC6817 out-competes `libutp-mod` in the transitional FIFO bottleneck case, `libutp-mod` flows exhibit preferable LPCC behaviour when competing with loss-based TCPs and using AQM bottlenecks.

VI. Conclusions

LEDBAT is a scavenger class transport protocol with congestion control based on one-way delay measurements to avoid adding more than 100ms of additional forwarding delay over FIFO bottlenecks. As an attractive alternative to TCP for latency-tolerant background file transfers, LEDBAT has been implemented by widely used peer-to-peer applications and commercial software update systems. However, the emergence of new AQM schemes in consumer contexts mean bottleneck queuing delays may be capped well below the levels required to activate LEDBAT's delay-based congestion control. This pushes LEDBAT to rely on loss-based behaviours that our paper demonstrates are (a) not fully specified in RFC6817, and (b) incorrectly interpreted by the widely deployed uTorrent Transport Protocol (uTP) library, `libutp`.

We demonstrate that `libutp` flows can regularly stall and provide unexpectedly degraded performance under specific combinations of low path RTT and bottleneck bandwidth. We further demonstrate conditions under which a `libutp` flow will actually out-compete a CUBIC TCP flow when sharing a PIE bottleneck. Both behaviours run counter to LEDBAT's design goals, and are triggered by the low queuing delay targets of modern AQMs. We propose a number of simple adjustments that eliminate the observed misbehaviours. More broadly, the interaction between modern AQM and LPCC transport protocols is not fully understood and would benefit from further detailed study.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063166.2071893

[2] D. Ros and M. Welzl, "Less-than-best-effort service: A survey of end-to-end approaches," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 898–908, Second 2013.

[3] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low extra delay background transport (ledbat)," *IETF, RFC6817*, December 2012. [Online]. Available: https://tools.ietf.org/html/rfc6817

[4] "libutp - uTorrent Transport Protocol implementation," BitTorrent Inc., 2010. [Online]. Available: https://github.com/bittorrent/libutp

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

[6] "uTorrent Transport Protocol," Bittorrent.org. [Online]. Available: http://www.bittorrent.org/beps/bep_0029.html

[7] "TCP LEDBAT Implementation," Apple Inc. [Online]. Available: http://opensource.apple.com//source/xnu/xnu-3248.60.10/bsd/netinet/tcp_ledbat.c

[8] C. Huitema, D. Havey, M. Olson, O. Ertugay, and P. Balasubramanian, "New Transport Advancements in the Anniversary Update for Windows 10 and Windows Server 2016," Microsoft, Jul 2016. [Online]. Available: https://goo.gl/ZfI8cG

[9] R. Pan, P. Natarajan, F. Baker, and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem," RFC 8033, Feb. 2017. [Online]. Available: https://tools.ietf.org/html/rfc8033

[10] K. M. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," Internet Draft, Mar 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-codel-07

[11] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The flowqueue-codel packet scheduler and active queue management algorithm," Mar 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06

[12] R. Al-Saadi and G. Armitage, "Dummynet AQM v0.2 – CoDel, FQ-CoDel, PIE and FQ-PIE for FreeBSD's ipfw/dummynet framework," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160418A, 18 April 2016. [Online]. Available: http://caia.swin.edu.au/reports/160418A/CAIA-TR-160418A.pdf

[13] Y. Gong, D. Rossi, C. Testa, S. Valenti, and M. Taht, "Fighting the bufferbloat: On the coexistence of {AQM} and low priority congestion control," *Computer Networks*, vol. 65, pp. 255 – 267, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128614000188

[14] N. K. G. Samaraweera and G. Fairhurst, "Reinforcement of tcp error recovery for wireless communication," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 2, pp. 30–38, Apr. 1998. [Online]. Available: http://doi.acm.org/10.1145/279345.279348

[15] P. Sreekumari, S. H. Chung, and W.-S. Kim, "A timestamp based detection of fast retransmission loss for improving the performance of tcp newreno over wireless networks," in *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Oct 2011, pp. 60–67.

[16] D. Kim, B. Kim, J. Han, and J. Lee, *Enhancements to the Fast Recovery Algorithm of TCP NewReno*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 332–341. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-25978-7_34

[17] S. Prasanthi, S. H. Chung, and C. Ahn, "An enhanced tcp mechanism for detecting and differentiating the loss of retransmisssions over wireless networks," in *2011 IEEE International Conference on Advanced Information Networking and Applications*, March 2011, pp. 54–61.

[18] V. Jacobson, R. Braden, and D. Borman, "TCP extensions for high performance," *IETF, RFC1323*, 1992. [Online]. Available: https://tools.ietf.org/html/rfc5681

[19] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Option," *IETF, RFC2018*, October 1996. [Online]. Available: https://tools.ietf.org/html/rfc2018.txt

[20] Y. Cheng and N. Cardwell, "RACK: a time-based fast loss detection algorithm for TCP," Internet Draft, Oct 2015. [Online]. Available: https://tools.ietf.org/html/draft-cheng-tcpm-rack-00

[21] S. Zander and G. Armitage, "TEACUP v1.0 - A System for Automated TCP Testbed Experiments," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 150529A, 2015. [Online]. Available: http://caia.swin.edu.au/reports/150529A/CAIA-TR-150529A.pdf

[22] J. Kua, R. Al-Saadi, and G. Armitage, "Using Dummynet AQM - FreeBSD's CoDel, PIE, FQ-CoDel and FQ-PIE with TEACUP v1.0 testbed," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160708A, 08 July 2016. [Online]. Available: http://caia.swin.edu.au/reports/160708A/CAIA-TR-160708A.pdf

[23] R. Pan, P. N. C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. V. Steeg, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem-draft," Internet Draft, Jun 2013. [Online]. Available: https://tools.ietf.org/html/draft-pan-aqm-pie-00

[24] "libutp-mod - modified uTorrent Transport Protocol implementation," 2017. [Online]. Available: https://github.com/RasoolAlSaadi/libutp-mod