©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay

Steffen Lindner, Gabriel Paradzik, Michael Menth
Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany
{steffen.lindner, gabriel.paradzik, menth}@uni-tuebingen.de

***Abstract*—The idea of an Alternative Best Effort (ABE) per-hop behaviour (PHB) emerged about 20 years ago. It provides a low-delay traffic class in the Internet at the expense of more packet loss than Best Effort (BE). Therefore, ABE is better suited than BE for loss-tolerant but delay-sensitive applications. Furthermore, ABE traffic should not degrade the service for BE traffic in terms of packet loss and delay. Therefore, Internet service providers may leave the choice of using BE or ABE to their customers as they achieve service differentiation without compromising other traffic.**

In this work, we revisit ABE and pursue the fundamental question whether an ABE service is technically feasible, how its service would look like and interact with existing transport protocols? We present a novel scheduler called Deadlines, Saved Credits, and Decay (DSCD) for combined scheduling of BE and ABE traffic. It allows to control ABE's delay advantage over BE and copes with varying bandwidth. We provide an implementation of DSCD in the Linux network stack and demonstrate its efficiency. A side product of the implementation is an efficient approximation of the exponential function in the kernel and a bandwidth estimation method that even works at moderate link utilization. We study DSCD in a semi-virtualized testbed with real networking stacks to understand implications for transport protocols in a BE/ABE Internet. The study analyzes ABE's impact on loss and delay under various conditions and gives recommendations for configuration.

***Index Terms*—Internet Protocol, traffic classes, service differentiation, packet scheduling, Alternative Best Effort (ABE), inter-class fairness**

I. Introduction

Over the last decades, increasing buffer sizes in network devices have led to temporarily long end-to-end delays. This phenomenon is known as bufferbloat [1]. Realtime applications, such as online gaming, video conference systems, or voice over IP applications, rely on small end-to-end delays. Table I gives examples.

TABLE I
Realtime applications with requirements regarding end-to-end delay and packet loss.

Application	E2E delay	Packet loss	Source
Online gaming (FPS)	20 ms – 80 ms	$\leq$ 5%	[2] [3]
Cloud gaming	$\leq$ 50 ms	$\leq$ 5%	[4] [5] [6]
Voice over IP (VoIP)	$\leq$ 150 ms	1% – 3%	[7]

A common solution to this problem is service differentiation, where delay sensitive traffic is prioritized over other traffic. The differentiated services framework (DiffServ) [8] allows for service differentiation in IP networks, offering various per-hop behaviours (PHBs) which can be considered as traffic classes. An example is Expedited Forwarding (EF), where EF traffic is strictly preferred over other traffic classes [9]. However, Internet service providers (ISPs) generally do not leave the choice of the PHBs to their customers because EF traffic may impede BE traffic of other users. If EF traffic accounts for only a small fraction of a link's overall traffic, it is likely to encounter only little loss and delay. In the absence of financial incentives, users possibly send too much EF traffic so that loss and delay objectives of EF traffic may not be met.

In this light, we revisit the idea of an Alternative Best Effort (ABE) service class [10] which has been first proposed in 2000. It causes less packet delay at the expense of increased packet loss but does not degrade treatment of BE traffic with respect to loss and delay. ABE constitutes a PHB that may be particularly attractive for service differentiation in the Internet where end users may choose ABE for low-delay traffic without negatively impacting BE traffic. Furthermore, it may be attractive when strict prioritization, such as the EF PHB, is discussed controversial in the context of network neutrality [11] as ABE does not impact the service for BE traffic.

Various scheduling algorithms [12], [13] have been presented in the past that may be apt to support an ABE PHB, but they suffer from various shortcomings. They are complex so that they have been implemented only in simulations. They require that the link bandwidth is stable and known. While thoroughly investigated by simulations, they did not address the inherent problem that very low ABE traffic rates may lead to unacceptable high packet loss. Supporting a loss versus delay trade-off has also been discussed in the IETF but implementation details were not in scope [14].

The contribution of this work is manifold. We address the fundamental question whether an ABE service class is technically feasible, how it behaves with up to date transport protocols, and whether it can be implemented on modern hardware. To that end, we propose a novel scheduler, named Deadlines, Saved Credits, and Decay (DSCD), for combined scheduling of BE and ABE traffic and implement it in the Linux network stack. It requires the knowledge of the link bandwidth only for secondary tasks and copes with unknown or variable transmission capacity through continuous bandwidth estimation. DSCD is designed to avoid excessive packet loss for low ABE traffic rates as this is most detrimental even to realtime applications.

We investigate the performance of ABE in terms of loss

and delay. We show the impact of configuration parameters, traffic types, and the ABE traffic rate on these metrics. We perform experiments with TCP variants and study both inter-protocol and inter-class fairness when traffic is carried over BE and ABE. The proposed bandwidth estimation method is fast, accurate, and efficient so that it also works at moderate link utilization. For its implementation, we developed a fast approximation of the exponential function in the kernel which may be reused for other purposes.

This paper is structured as follows. In Section II, we review related work. Section III presents the novel DSCD algorithm for joint scheduling of BE and ABE traffic. Afterwards, we point out relevant implementation details of DSCD in the Linux network stack in Section IV. In Section V we investigate the performance of BE and ABE traffic when scheduled with the DSCD scheduler in various networking scenarios. Finally, Section VI summarizes this work and Section VII draws conclusions.

II. Related Work

We review work in the context of traffic prioritization related to DSCD. First, we discuss the ABE schedulers *Duplicate Scheduling with Deadlines* (DSD) and *Delay Segment FIFO* (DSF). Then, we present AQM implementations in the Linux network stack and further traffic differentiation mechanisms.

A. Alternative Best Effort (ABE)

The authors of [12] present the concept of Alternative Best Effort (ABE) as a traffic class similar to BE. ABE provides a bounded-delay service class (green) and a Best Effort (BE) class (blue). Blue traffic achieves the same throughput as in a conventional FIFO system. Green traffic receives priority service whenever possible without harming blue traffic. The concept has also been presented in the IETF [10].

The authors subsequently present *Duplicate Scheduling with Deadlines* (DSD) as a combined scheduler for BE and ABE traffic. DSD utilizes separate, physical FIFO queues for blue and green packets. In addition, it leverages a virtual queue to simulate the queue behaviour of a single FIFO queue that serves both blue and green packets. Based on the fill state of the virtual queue the transmission time for blue packets in a FIFO system is computed, for which the link capacity must be known and stable. This FIFO system transmission time is taken as a deadline for the blue packets in the physical queue. The deadline for green packets in the physical queue is their arrival time plus a maximum tolerable delay. The basic dequeue procedure is as follows. If a blue packet needs to be sent to keep its deadline, it is dequeued and sent. If a green packet passed its deadline, it is dropped. With DSCD, green packets are sent whenever blue packets do not need to be sent. This allows green packets to overtake blue packets when previous green packets were dropped, which effects a delay advantage. DSD is similar to DSCD but utilizes an algorithm to share capacity between both physical queues in a sophisticated way whenever blue or green packets are eligible for transmission. However, the objective for this capacity sharing is debatable as it strives for per-class fairness. As a result, the associated algorithm is complex. The delay, throughput, and packet loss must be tracked so that the algorithm is hard to implement on real forwarding nodes. The performance of DSD has been evaluated by simulations.

B. Delay Segment FIFO (DSF)

Karsten et al. [13] consider multiple traffic classes i with class-specific delay targets D_i. They suggest Delay Segment FIFO (DSF) as an algorithm for scheduling packets such that the queuing delay of the packets is at most their class-specific delay target and at most the delay experienced in a FIFO queue. If these constraints cannot be met, packets are dropped. They use a physical packet queue for each traffic class and a joint slot queue for storing slots that contain the right to send a certain amount of bytes. The slot queue is partitioned into class-specific segments such that the overall capacity of the segments corresponding to the c most stringent traffic classes is $D_c \cdot C$. Thereby C is the link bandwidth which must be known a priori and stable. When a packet arrives, it is added with a deadline to the corresponding packet queue. Moreover, a slot is added to slot queue in a free segment belonging to the packet class of the packet or better. The algorithm is complex and uses a minimal throughput interference index to guarantee some kind of TCP fairness. The authors implement and evaluate DSF for the network simulator ns-3.

C. AQM Implementations in the Linux Network Stack

FQ-CoDel [15] is a packet scheduler and Active Queue Management (AQM) algorithm developed to mitigate the bufferbloat problem. It is based on deficit round robin (DRR) and CoDel and distinguishes between sparse and non-sparse flows. FQ-CoDel stochastically enqueues incoming packets based on their 5-tuple hash into different queues. Each queue is managed by the CoDel AQM. FQ-CoDel is the default queueing discipline in many Linux distributions.

Ramakrishnan et al. [16] present an implementation of FQ-PIE, a flow-based variation of Proportional Integral controller Enhanced (PIE), for the Linux network stack. They compare it to PIE and FQ-CoDel and evaluate the fairness among responsive and unresponsive flows. Further, they evaluate the fairness between different TCP versions, i.e., TCP Cubic and TCP BBR.

CAKE [17] is a network queue managment system designed for the home gateway. It includes bandwidth shaping, queue management, DiffServ handling and TCP ACK filtering. Further, it provides host and flow isolation. CAKE is part of the mainline Linux kernel and was developed for the OpenWrt router firmware.

D. Further Traffic Differentiation Mechanisms

RD [18] proposes two service classes: a high-transmission class and a low queuing delay class. Both classes are implemented by separate FIFO queues on a router. The next transmitted packet is selected according to the intended throughput ratio between both classes. To guarantee delay differentiation, the corresponding queue sizes are dynamical calculated. Queue

sizes and service rates have to be calculated, which requires knowledge of the link capacity.

QJump [19] classifies different latency-sensitive levels. Packets from higher classes are rate-limited but can “jump-the-queue” over packets from lower classes. To provide rate-limitations and some kind of throughput fairness, QJump needs knowledge about the number of network nodes and link speeds. QJump was designed for datacenter applications. The authors evaluate QJump with simulations and a small real-world deployment.

Briscoe et al. [20] give a broad overview of techniques to reduce Internet latency. They categorize latency sources, e.g., caused by too large network buffers, and present their advantages and disadvantages.

Although FQ-CoDel, FQ-PIE, Cake, RD, and other mechanisms provide means for service differentiation, i.e., they enable low-delay forwarding for real-time or low-bitrate traffic, they do so at the expense of BE traffic or large flows. In contrast, the objective of DSCD is that BE traffic is not treated worse than in a pure FIFO system. Further, DSCD could be used in conjunction with existing AQMs such as FQ-CoDel.

III. Scheduling with Deadlines, Saved Credits, and Decay (DSCD)

We give an overview of DSCD's design idea and present its algorithm in detail.

A. Design Idea

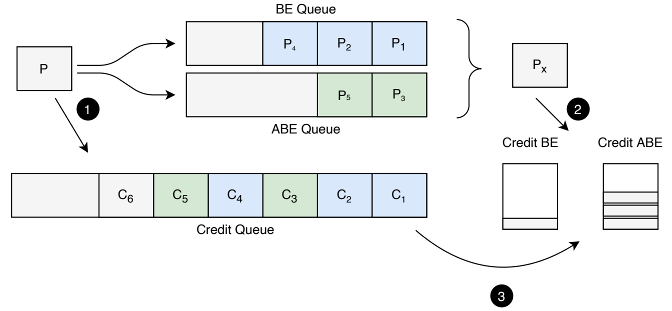


Fig. 1. The DSCD scheduler stores BE and ABE packets in the class-specific packet queues $Q[BE]$ and $Q[ABE]$. For every enqueued packet, a credit element is inserted into the credit queue Qc. The credit counters $CC[BE]$ and $CC[ABE]$ store class-specific credits that are needed for packet dequeue. If these credits do not suffice, new credits are taken from the credit queue.

Figure 1 illustrates the data structure of DSCD that we introduce incrementally. The DSCD scheduler utilizes two FIFO queues: one to enqueue BE traffic ($Q[BE]$) and one to enqueue ABE traffic ($Q[ABE]$). When a packet arrives, it is enqueued into the corresponding class-specific queue based on its DiffServ code point (DSCP). In addition, ABE packets are equipped with a deadline at enqueue, which is the enqueue time plus the delay threshold T_d. When a non-empty BE queue is served, a packet is removed and dequeued for forwarding. When a non-empty ABE queue is served, packets are removed. They are dequeued if their deadline is met, otherwise they are dropped. The unused transmission capacity may be used by subsequent ABE packets, either immediately or later, so that they can take over BE packets without delaying them.

DSCD achieves this behavior by dequeuing packets from the BE and ABE packet queue using so-called credits. To that end, DSCD maintains a FIFO queue Qc for credit elements. Whenever a packet arrives, a credit element with the packet's size and traffic class is inserted into the credit queue ❶. The credit is needed for dequeuing packets. For that purpose, two class-specific credit counters are maintained ($CC[ABE]$, $CC[BE]$). The first packet of a non-empty queue can be dequeued only if the corresponding credit counter is at least the packet's size. If so, the packet can be removed from its queue. If the packet belongs to the ABE class and its deadline has passed, the packet is dropped. Otherwise, the corresponding credit counter is decremented by the packet's size and the packet can be forwarded ❷. If the credit counters of both queues are too low for dequeuing a packet, a credit element is removed from the credit queue and the credit counter of the element's class is incremented by the element's size ❸. The credit of dropped ABE packets remains in the system for some time. With that saved credit, subsequent ABE packets can be served earlier than comparable BE packets, but without delaying BE packets longer than in a pure FIFO system.

This sketch misses some details. First, packets may be lost during enqueue due to limited queue size. Second, low rates of ABE traffic should not experience high packet loss, which requires some extra logic. Third, credit should not be stored for infinite time, even in the presence of congestion. To that end, the credit is devaluated over time according to an exponential function ($exp(-\lambda \cdot \Delta)$) where Δ is the passed time and λ is the decay rate. We configure it via the half-life time $t_h = \frac{\ln(2)}{\lambda}$. After one half-life time t_h only half the credit is still available. Fourth, to simulate the behaviour of a FIFO queue, the credit should vanish when both packet queues are empty. It is drained from the system with link bandwidth C, which is continuously estimated.

B. Algorithm

We formalize the above sketched algorithm using pseudocode and address the missing details. We introduce the data structure of DSCD and describe its algorithms for packet enqueue, packet dequeue, credit devaluation, and bandwidth estimation.

1) Data structures: The data structures of DSCD are illustrated in Figure 1.

- DSCD maintains separate FIFO queues $Q[BE]$ and $Q[ABE]$ to store BE and ABE packets
- and a FIFO queue Qc to store credit elements.
- There are global counters for dequeued credit elements for each traffic class, $CC[BE]$ and $CC[ABE]$, which are zero at system start.
- The credit counter CC_{cq} counts the stored credit in the credit queue Qc.
- Furthermore, the global variable C stores the available bandwidth which is estimated in Algorithm 4 and utilized in Algorithm 3.

- Algorithm 3 also uses the global variable $lastDevaluation$ to record the last devaluation instant which is initialized with $-\infty$.
- Algorithm 4 uses global variables as helpers for bandwidth estimation. S_B, S_T, and $lastPktSize$ are initialized with zero, $backlogged$ with false, and $lastDequeue$ and $lastRateUpdate$ with $-\infty$.

2) DSCD Enqueue: The enqueue operation is given by Algorithm 1. First, saved credit is devaluated, which is described in Section III-B4. Then, a new packet p is dropped if its size together with the overall credit in the system exceeds the buffer size B_{max} (line 2-3). Otherwise, the packet is enqueued. Then, the deadline $p.d$ is set for ABE packets (line 5-6). In the remainder, an element e with the packet's length $p.len$ and class $p.class$ is created. It is added to the credit queue Qc whose credit counter CC_{cq} is incremented. Finally, the packet is added to its class-specific queue.

Algorithm 1: DSCD enqueue routine

```
   Input  : Packet p
1  DevaluateCredit()
2  if p.len + CC_cq + CC[BE] + CC[ABE] > B_max
     then
3  |  drop(p)
4  else
5  |  if p.class == ABE then
6  |  |  p.d = t_now + T_d
7  |  e = new CreditElement(p.len, p.class)
8  |  Qc.add(e)
9  |  CC_cq += p.len
10 |  Q[p.class].add(p)
```

3) DSCD Dequeue: We first explain the principle of the dequeue operation before we go into details.

DSCD has class-specific credit counters $CC[X]$, $X \in \{BE, ABE\}$. When a packet is dequeued, the corresponding credit counter is decreased by the packet size, but it cannot fall below zero. If both credit counters are too low to dequeue a packet, credit elements are removed from the credit queue and the elements' credit it added to the corresponding credit counters. A packet is dequeued as soon as one credit counter is large enough. If both counters are large enough, ABE traffic is preferentially served.

We now look at the pseudocode in Algorithm 2 which dequeues a packet if possible and returns NULL otherwise. First, the credit counter $CC[ABE]$ is devaluated, which is described in Algorithm 3. Afterwards, all ABE packets with violated deadlines are dropped from the ABE queue if they are followed by more than T_q other packets (line 2-4). The second part of the condition avoids packet loss when too few other packets in the packet queue could use the credit of dropped packets. Then, the return packet is initialized with NULL and a dequeue attempt is made only if the system holds at least one packet (line 6). The packet for dequeue is determined in the subsequent loop which ends with a successfully dequeued packet (line 7-16). Within the loop, a

Algorithm 2: DSCD dequeue routine

```
   Output: Next packet to be served
1  DevaluateCredit()
2  while Q[ABE].head.d > t_now and Q[ABE].len > T_q
     do
3  |  drop(Q[ABE].removeHead())
4  end
5  p = NULL
6  if !Q[BE].empty() or !Q[ABE].empty() then
7  |  while p == NULL do
8  |  |  if CC[ABE] ≥ Q[ABE].head.len then
9  |  |  |  p = Q[ABE].removeHead()
10 |  |  else if CC[BE] ≥ Q[BE].head.len then
11 |  |  |  p = Q[BE].removeHead()
12 |  |  else
13 |  |  |  e = Qc.removeHead()
14 |  |  |  CC_cq -= e.credit
15 |  |  |  CC[e.class] += e.credit
16 |  end
17 |  CC[p.class] -= p.len
18 |  EstimateBandwidth(p)
19 return p
```

queue with a sufficiently large credit counter is determined and its first packet is removed (line 8-11). If neither queue has a sufficiently large credit counter, a credit element is removed from the credit queue and the credit counter of the corresponding traffic class is incremented (line 13-15). After successful packet dequeue, the credit counter of the respective class is decremented (line 17). Then, the estimate of the link bandwidth C is updated using Algorithm 4 (line 18). Finally, either the dequeued packet or a NULL pointer is returned.

4) Credit Devaluation: Credit devaluation is needed for two reasons.

First, the overall credit in the system, i.e., the credit in the credit queue Q_c and the counters $CC[BE]$ and $CC[ABE]$, simulates an upper bound of the fill state of an alternative FIFO queue. That is a necessary invariant to ensure that BE packets are not served later than in a comparable FIFO queue. If both packet queues are empty, remaining credit in the system must vanish with the current link bandwidth C.

Second, credit from dropped ABE packets is stored by $CC[ABE]$ and used to send other ABE packets early. Without additional devaluation, credit from dropped ABE packets remains in the system until the end of a congestion period. This may incentivize applications to send unnecessary ABE data to provoke packet loss and leverage resulting credit in order to gain a delay advantage for later ABE traffic. Therefore, we believe that credit should vanish over time, even in the presence of congestion. Further, transport protocols benefiting from lower transmission delay may obtain a throughput advantage via ABE compared to those transmitting over BE, despite increased packet loss. Credit devaluation limits that advantage (see Section V-E) and, thereby, leads to better inter-

class fairness for transport protocols.

As pointed out, credit devaluation is needed in two ways: (1) If the system is empty, ABE's saved credit $CC[ABE]$ is reduced over time by the transmission rate C. (2) ABE's saved credit $CC[ABE]$ decays over time exponentially with rate λ. This pursues the previously discussed objectives.

Algorithm 3: DevaluateCredit

```
1  Δ = t_now − lastDevaluation
2  lastDevaluation = t_now
3  if Q[BE].empty() and Q[ABE].empty() then
4      while !Qc.empty() do
5          e = Qc.removeHead()
6          CC_cq −= e.credit
7          CC[e.class] += e.credit
8      CC[ABE] = max(0, CC[ABE] − C · Δ)
9  else
10     CC[ABE] = CC[ABE] · exp(−λ · Δ)
```

Algorithm 3 performs these operations. First, the passed time Δ since the last devaluation is computed and the global variable $lastDevaluation$ is updated by the current time t_{now}. If both queues are empty, the credit queue is emptied and the credit counters of the corresponding packet queues are incremented (line 4-7). Then, ABE's credit counter $CC[ABE]$ is reduced with the current bandwidth C over time Δ; thereby the credit counter cannot fall below zero. If at least one queue is not empty, ABE's credit counter $CC[ABE]$ is devaluated exponentially over time Δ with rate λ (line 10). We call this mechanism *exponential decay*.

5) Bandwidth Estimation: Algorithm 3 requires an estimate C of the link bandwidth to devaluate credit in the presence of an empty queue. To measure C, an amount of sent bytes is divided by their transmission time. We capture the transmission time of a packet from the time it is dequeued until the next packet is dequeued, provided that there is no idle time in between. We ensure this by considering only packets that leave a non-empty queue, i.e., a backlogged queue.

To cope with varying bandwidth, we accumulate both sent bytes and transmission times by weighted sums S_B and S_T and derive an estimate by $C = \frac{S_B}{S_T}$. We utilize the weighted sum of the moving average UTEMA [21] for that purpose:

$$S_X(t) = S(t_{last}) \cdot e^{-\mu \cdot (t - t_{last})} + X \quad (1)$$

$$t_{last} = t. \quad (2)$$

X is a series of samples at time instants t. $S_X(t)$ is the weighted sum of the sampels at time t. The sum is updated whenever a new sample is available and t_{last} indicates the last update time of the sum. The advantage of UTEMA is that the contribution of the samples considered in the weighted sum decreases exponentially over time with rate μ, i.e., newer samples have a larger impact on the sum than older samples. We apply this concept to the size of sent packets B and their mere transmission times T, which yields S_B and S_T. For configuration, a memory M is used to set the rate $\mu = \frac{1}{M}$.

Algorithm 4: EstimateBandwidth

Input : *Packet p*

```
1  if backlogged then
2      Δ = t_now − lastRateUpdate
3      S_B = S_B · exp(−μ · Δ) + lastPktSize
4      S_T = S_T · exp(−μ · Δ) + (t_now − lastDequeue)
5      C = S_B / S_T
6      lastRateUpdate = t_now
7  if Q[ABE].len + Q[BE].len > 0 then
8      backlogged = true
9  else
10     backlogged = false
11 lastDequeue = t_now
12 lastPktSize = p.len
```

Algorithm 4 translates this concept into pseudocode for a rate estimation procedure that is called at the end of each successful packet dequeue. In the first step of the algorithm (line 1-6), the estimated rate C is updated if the last dequeued packet was backlogged. The elapsed time Δ since the last rate update is computed and used to devaluate the weighted sums of bytes and transmission times (S_B, S_T) which are also increased by the size of the last dequeued packet ($lastPktSize$) and its transmission time ($t_{now} - lastDequeue$). Then, the estimated bandwidth C and the last rate update time $lastRateUpdate$ are updated. The variable $backlogged$ is set to true if there are more packets waiting in some queue, otherwise it is set to false (line 7-10). Finally, the current dequeue time and the size of the dequeued packet are recorded by $lastDequeue$ and $lastPktSize$.

IV. Implementation of DSCD in the Linux Network Stack

Traffic schedulers and AQMs are often evaluated using simulation frameworks such as ns-3 or OMNeT++.

Although simulation frameworks offer a lot of freedom regarding implementation, they are only an approximation of reality. Therefore, the trustworthiness of simulation results for complex protocols, such as TCP, heavily depend on the validity of the simulation model. For this reason, we decided to implement DSCD in the Linux network stack as proof-of-concept implementation and perform experiments with existing protocol implementations, in particular up-to-date TCP variants. Moreover, this implementation demonstrates the practical feasibility of DSCD.

We first introduce some background information on queuing disciplines in the Linux kernel. Afterwards, we present an efficient approximation of an exponential decay function for the Linux kernel. Then we explain the implementation of the exponential decay for the stored ABE credit and the rate estimation as these implementation aspects are most challenging. The overall code for DSCD on Linux is available at Github[1].

[1]https://github.com/uni-tue-kn/dscd-linux-qdisc

A. Use of Queuing Disciplines in the Linux Kernel

Queuing disciplines, also called QDiscs, are part of the Linux network stack and are located in the kernel space. They perform tasks such as traffic shaping, packet classification, or packet dropping. A self-implemented QDisc may perform other, almost arbitrary operations on packets. QDiscs are implemented in the C programming language.

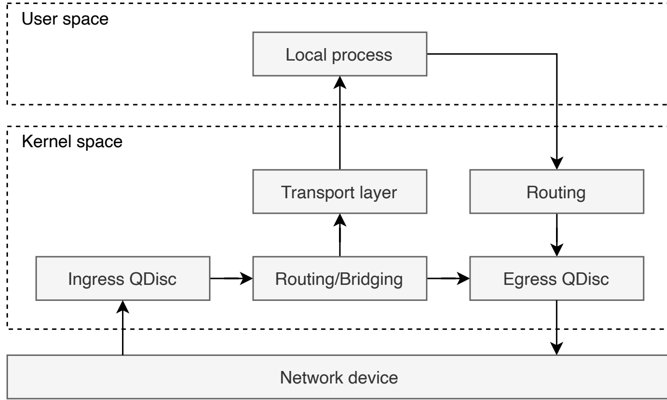


Fig. 2. Packet handling in the Linux network stack.

Figure 2 illustrates the simplified packet handling in the Linux network stack. Incoming packets from the network card are passed to the Linux kernel. Initially, packets are handed to an ingress QDisc. Within an ingress QDisc, packets can be filtered or rate-limited. Afterwards, a routing or bridging decision is taken. If the packet is destined for the host itself, the packet is passed to the transport layer and further to the application process in the user space. Otherwise, the packet is passed to the egress QDisc of the outgoing interface. DSCD is completely implemented as egress QDisc. Both ingress and egress QDiscs provide a standardized interface to the Linux kernel. It includes functions for packet enqueue and dequeue which correspond to the algorithms presented in Section III.

QDiscs also provide the possibility for *chaining*. This means that multiple QDiscs are executed one after another. Chaining is used to separate functionality between different QDiscs, e.g., rate-limiting and classification. QDiscs leveraging chaining are called *classful* and are organized in a tree structure. The Kernel enqueues the packet in the so-called root QDisc. The root QDisc then enqueues the packet into one of its child QDiscs which may enqueue the packet in one of its own child QDiscs. When a packet should be dequeued, the Kernel calls the dequeue routine at the root QDisc which in turn calls the dequeue routine of its child QDiscs.

We leverage the functionality of classful QDiscs within our testbed to combine rate-limiting (with the classful QDisc tbf) and our DSCD QDisc. The use of tbf facilitates the configuration of a controlled and variable bottleneck capacity as described in Section V-A1.

B. Efficient Approximation of the Exponential Function

The algorithms presented in Section III require the computation of an exponential function for credit devaluation and rate estimation. However, the exponential function is not available in the kernel and only integer arithmetic can be used[2]. Therefore, we present an approximation for the multiplication of an integer n with $exp(x)$ that can be efficiently implemented in the Linux kernel. The exponential function can be rewritten as

$$exp(-x) = 2^{-x/ln(2)}. \tag{3}$$

We first propose a piecewise linear function as floating point approximation of 2^{-x} and then we implement $n \cdot 2^{-x}$ with integer arithmetic. Finally, we consider the application of the approximation to exponential decay and bandwidth estimation.

1) Floating Point Approximation of 2^{-x} for Positive Arguments: To approximate the power function $p(x) = 2^{-x}$, we use the following piecewise linear function for $x \geq 0$ which uses interpolation of integer-based sampling points only:

$$f(x) = \frac{\lfloor x \rfloor - x + 2}{2^{\lfloor x \rfloor + 1}}. \tag{4}$$

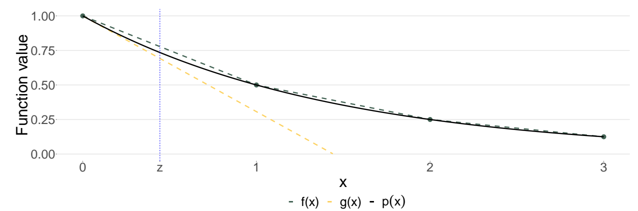
We improve the error for small values of x by another approximation

$$g(x) = 1 - ln(2) \cdot x \tag{5}$$

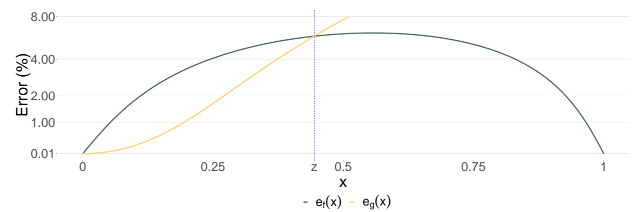
which is based on the derivative of 2^{-x} at $x = 0$. Both approximations are illustrated in Figure 3(a) and the corresponding error functions in Figure 3(b). We combine them to minimize the error by

$$h(x) = \begin{cases} g(x) & 0 \leq x \leq z \\ f(x) & z < x \end{cases} \tag{6}$$

with $z \approx 0.4443$ being the abscissa of the intersection point of both error functions.



(a) Approximation of the power function $p(x) = 2^{-x}$ by the piecewise linear function $f(x)$ and the derivative-based linear function $g(x)$.



(b) Error functions $e_f(x) = f(x) - 2^{-x}$ for the piecewise linear approximation and $e_g = g(x) - 2^{-x}$ for the derivative-based linear approximation.

Fig. 3. Approximation options for the power function $p(x) = 2^{-x}$. We combine them in $h(x)$ to minimize the error.

[2]https://www.kernel.org/doc/html/v5.0/process/howto.html

2) Implementation of $n \cdot 2^{-x}$ with Integer Arithmetic: The argument for the power function is a fractional number x. As the Linux kernel supports only integer arithmetic, we represent the argument by a scaled number $y = x \cdot 2^s$; we denote s the scaling exponent.

We propose the function $a(n, y, s) = n \cdot 2^{(-y/2^s)}$ to multiply an integer n with a power function value where n, y, and s are non-negative integers. We implement $a(n, y, s)$ using function h with argument $y/2^s$ and utilize the approximations

$$ln(2) \approx \frac{2^{12}}{5909} \quad (7)$$

$$z \approx \frac{2^{12}}{9219}. \quad (8)$$

This results in

$$a(n,y,s) = \begin{cases} n - \dfrac{y \cdot n}{5909 \cdot 2^{s-12}} & y \cdot 9219 \leq 2^{s+12} \\ \dfrac{n \cdot (\frac{y}{2^s} + 2) - \frac{n \cdot y}{2^s}}{2^{(\frac{y}{2^s}+1)}} & 2^{s+12} < y \cdot 9219 \end{cases}. \quad (9)$$

Here, divisions imply integer divisions. Therefore, we can substitute $\lfloor x \rfloor$ in $h(x)$ by $\frac{y}{2^s}$ in the formula. We first evaluate numerator and denominator of any fraction prior to division to prevent unnecessary loss of accuracy. For efficiency, every division by 2^k is performed as a bit shift by k bits to the right. Moreover, intermediate results may be stored and reused.

The computation of $n \cdot 2^{-x}$ is achieved by calling $a(n, y, s)$ with the arguments n, $y = x \cdot 2^s$, and s. For the computation of $n \cdot exp(-x)$ Equations (3) and (7) need to be taken into account so that $a(n, y, s)$ is to be called with arguments n, $y = x \cdot 5909 \cdot 2^{s-12}$, and s.

If a floating point number m is to be multiplied with an exponential value, one can scale m to $n = m \cdot 2^{(s_m)}$ with a scaling exponent s_m before applying it to $a(n, y, s)$. The returned number is the result scaled with $2^{(s_m)}$.

3) Application to Exponential Decay and Bandwidth Estimation: We apply $a(n, y, s)$ to implement line 10 in Algorithm 3 (DevaluateCredit) and line 3-4 in Algorithm 4 (EstimateBandwidth). For DevaluateCredit, the rate $\lambda = \ln(2)/t_h$ is configured via the half-life time t_h. Both Δ and t_h are counted in ns. The parameter $y = \Delta/t_h$ is additionally scaled with 2^{20}, i.e., $s = 20$, to gain precision for small values of Δ. Credits are scaled with 2^{10} to limit loss of accuracy for small integers, i.e., $s_m = 10$. For EstimateBandwidth, the rate $\mu = 1/M$ is configured via the memory M. Both Δ and M are counted in ns. The parameter $y = \mu \cdot \Delta / \ln(2)$ is again scaled with 2^{20}, i.e., $s = 20$.

In the following, we derive an upper bound for the relative error in practise. The transmission time for packets with 1490 bytes is 1.2 ms with 10 Mbit/s and 12 µs for 1 Gbit/s. The algorithms are mostly called in these intervals. The half-life time for exponential decay of $t_h = 100$ ms and a memory for bandwidth estimation of $M = 50$ ms correspond to rates of $\lambda = \frac{ln(2)}{t_h} = \frac{6.9}{\text{s}}$ and $\mu = \frac{1}{M} = \frac{20}{\text{s}}$. Therefore, the exponential function is called under relevant conditions with values smaller than 1.2 ms$\cdot \frac{20}{\text{s}} = 0.024$ and the approximation $h(x)$ is called with values smaller than $\frac{0.024}{ln(2)} = 0.034$. The relative error by the approximation for such values is about 0.029% (see Figure 3(b)), i.e., very low. For higher link speeds, e.g., 10 Gbit/s or 100 Gbit/s, the relative error further decreases.

C. Performance of Linux QDisc Forwarding

Modern NICs support multiple transmit (TX) and receive (RX) queues to facilitate highspeed packet processing. Packets are distributed to different queues such that they can be processed by separate CPU cores without interference, e.g., caused by lock mechanisms. This mechanism is called Receive-Side Scaling (RSS). Packets are assigned to a queue using a hash function, e.g., 4-tuple hash over IP addresses and TCP ports[3]. In the following, we investigate the general performance for traffic forwarding with Linux QDiscs. We deploy the pfifo QDisc in a similar testbed as presented in Section V-A1. The bottleneck link has a capacity of 100 Gbit/s. We vary the number of available CPU cores and TX queues[4] and establish 32 TCP flows between the senders and the receiver. Table II shows the L2 throughput, i.e., Ethernet throughput, on the bottleneck.

TABLE II

#CPU cores	#TX queues	L2 throughput (Gbit/s)
1	1	25.86
2	1	34.49
	2	48.45
4	1	38.05
	2	62.73
	4	92.84
8	1	40.29
	2	68.42
	4	96.72
	8	97.07

With a single CPU core (and a single TX queue), only 25.86 Gbit/s can be achieved on L2. The throughput with a single TX queue can be increased by increasing the number of CPU cores. However, the throughput does not linearly increase with the number of CPU cores and converges at 40.29 Gbit/s for 8 CPU cores. This is caused by communication overhead between the CPU cores as the TX queue can only be accessed by a single CPU core at a time. The throughput increases with an increasing number of TX queues and converges at 97.07 Gbit/s for 8 CPU cores with 8 TX queues. A saturation below 100 Gbit/s is reasonable as the L2 throughput does not include preamble and inter-frame gap. The experiment shows that forwarding with 100 Gbit/s requires multiple CPU cores and TX queues on Linux systems. Therefore, we use 8 CPU cores and 8 TX queues when performing experiments at 100 Gbit/s.

As the assignment of packets to TX queues is based on a hash function, packets of the same flow are placed in the same TX queue. If only a subset of the TX queues are used by chance, 100 Gb/s may not be achieved. However, this is unlikely in a 100 Gb/s environment where the number of flows is high.

[3]https://www.kernel.org/doc/Documentation/networking/scaling.txt

[4]The maximum number of TX queues is limited by the number of available CPU cores.

D. Efficiency of the DSCD Implementation

Now, we assess the efficiency of the DSCD implementation. At first sight, the DSCD algorithm has some complexity, but the implementation is efficient. We demonstrate that by the following experiments.

We deploy DSCD in a similar testbed[5] as presented in Section V-A1. The bottleneck link has a capacity of 100 Gbit/s. The delay threshold is set to $T_d = 10$ ms and the half-life time t_h is set to $t_h = 100$ ms. We establish 32 TCP flows between them. Every 2nd TCP flow is labeled as ABE. We measure the overall TCP goodput and average CPU load of DSCD and compare it to existing Linux QDiscs[6] such as FQ-CoDel, FQ-PIE, Stochastic Fair Queuing (SFQ), and pfifo. Table III shows the results.

TABLE III
TCP GOODPUT AND CPU LOAD OF VARIOUS LINUX QDISCS.

QDisc	TCP goodput (Gbit/s)	CPU load (%)
DSCD	89.08	36.27
FQ-CoDel	89.02	38.99
FQ-PIE	89.00	44.21
SFQ	89.03	38.72
pfifo	89.06	35.41

All considered QDiscs achieve approximately the same goodput of ~ 89 Gbit/s. pfifo and DSCD have the lowest CPU load with 35.41% and 36.27% and FQ-PIE the highest with 44.21%. The results show that the DSCD implementation in the Linux kernel is efficient and comparable to other QDiscs.

V. PERFORMANCE EVALUATION

We first explain our performance evaluation methodology. Then we validate the bandwidth estimation algorithm which is part of DSCD. Afterwards, we study DSCD scheduling for non-adaptive traffic, periodic traffic, and TCP traffic, demonstrating the impact of configuration parameters and ABE traffic rates on packet loss and delay. Finally, we investigate inter-protocol and inter-class fairness for different TCP variants in connection with ABE.

A. Methodology

We introduce the methodology for the performance study. We present the testbed, explain experiment organization and performance metrics, and describe how traffic is generated.

1) Testbed: We leverage a semi-virtualized testbed on a host system with 128 GB RAM. We work with KVM-based virtual machines (VMs) that are assigned 4 GB RAM and two cores with 3.2 GHz from an Intel(R) Xeon(R) Gold 6134. The VMs run with Linux kernel 5.10 and have dedicated 10 Gbit/s network cards. Thus, if not mentioned differently, links between VMs have a capacity of exactly 10 Gbit/s. No overbooking is performed on the VM hosts, the NICs are passed through to the VMs using SR-IOV and the CPUs are pinned to physical cores to minimize the influence of virtualization on the experimental results.

[5]The bottleneck VM has 8 cores in this experiment.

[6]We use multi-queuing (8 TX queues) to improve performance.

The logical structure of the testbed is illustrated in Figure 4. Up to five VMs send traffic to a bottleneck VM via dedicated 10 Gbit/s links. The bottleneck VM is connected to a so-called RTT VM via a throttled link which has a capacity of $C = 1$ Gbit/s in most experiments. This is done to perform experiments with software generated traffic (see Section V-A3) and to study DSCDs behavior in a controlled environment. However, as shown in Section IV-D, DSCD supports much higher bandwidths. The rate limitation is achieved with the Linux queuing discipline *tbf* [22] using a rate of 1 Gbit/s and an tbf bucket size of 10 maximum transfer units (MTUs)[7]. This represents the bottleneck of the path and possibly causes congestion. DSCD is deployed at the bottleneck node with a buffer size of $B_{max} = C \cdot 25$ ms $= 3.125$ MB. Thus, packets are queued by DSCD and sent whenever tbf allows. Further default DSCD parameters are a delay threshold of $T_d = 10$ ms, a half-life time of $t_h = 100$ ms, and a queue threshold of $T_q = 1$.

The RTT VM delays packets according to a configured RTT. The Linux queuing discipline Netem [23] is utilized to delay the traffic by RTT time for which the default value is $RTT = 100$ ms.

Table IV summarizes the default configuration for the experiments. Rate limitation by tbf, DSCD scheduling, and delay addition are applied only in one direction.

TABLE IV
DEFAULT CONFIGURATION OF TESTBED AND DSCD ALGORITHM.

Parameter	C	RTT	B_{max}	T_d	t_h	T_q
Value	1 Gbit/s	100 ms	25 ms	10 ms	100 ms	1

2) Performance Metrics and Experiment Organization: The performance metrics in the experiments are packet queuing delay and packet loss on the bottleneck node and end-to-end goodput of TCP flows.

Every experiment, i.e., a set of studied parameters, is executed 30 times and runs for 45 s. Data from the first 15 s of each run are discarded to avoid the impact of a potential transient phase. Data from the last 2 s are removed as not all streams may be terminated simultaneously. For the remaining 28 s we calculate performance metrics. We average them over the 30 runs and calculate 95% confidence intervals. However, we omit them in the figures for the sake of readability as they are very small.

3) Traffic Generation: In the experiments, three different traffic types are utilized. We describe their generation in the following.

a) Non-adaptive traffic with bursts: To investigate basic effects of DSCD scheduling without interactions of transport protocols such as TCP, we apply non-adaptive traffic with bursts. Poisson traffic is a first candidate but does not cause substantial queuing at link speeds of 1 Gbit/s. Therefore, we generate packets with LogNormal-distributed packet inter-arrival times and send them over UDP.

[7]We set the bucket size to a low value to avoid substantial impact on DSCD queuing. We validated that a bandwidth of 1 Gb/s is still achieved with this setting.

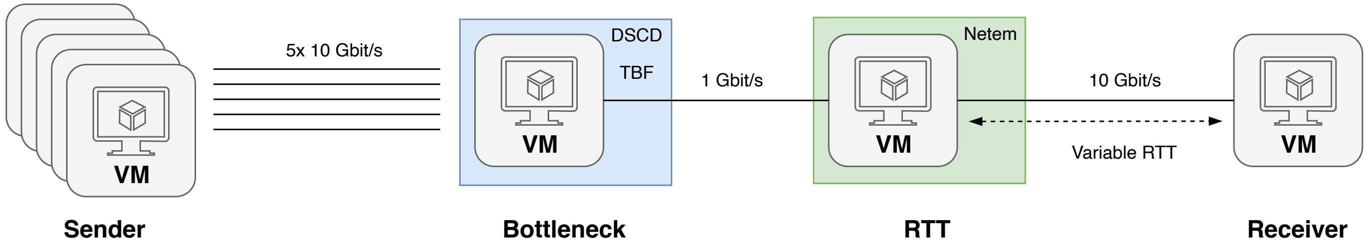


Fig. 4. The performance evaluation is carried out in a semi-virtualiezd testbed. It consists of multiple virtual machines (VMs) on a single server. The VMs are assigned dedicated 10 Gbit/s network cards. Up to five VMs send traffic to a receiver. The path has a bottleneck of 1 Gbit/s and a configurable RTT that are imposed by a bottleneck node and an RTT node. DSCD is deployed at the bottleneck node.

We derive mean and standard deviation of the random variable A, which denotes the inter-arrival time, from the following desired properties. Let C be the capacity of the link and ρ its target utilization. Given a fixed packet size of $B = 1490$ bytes including all headers (20 bytes for IP, 8 bytes for UDP, 14 bytes for Ethernet), we can compute the number of packets N within a 10 ms interval by $N = \rho \cdot C \cdot 10$ ms$/B$. Thus, the expected packet inter-arrival time is $E[A] = 10$ ms$/N$. We set the standard deviation $\sigma[A]$ such that the standard deviation of N inter-arrival times is 5 ms. Thus, the standard deviation of a single inter-arrival time is $\sigma[A] = 5$ ms$/\sqrt{N}$. This traffic is sufficiently bursty. We generate it on a single machine using 40 threads that send packets in a round-robin manner.

To visualize the effect of the resulting arrival process, we experimentally count the number of arrived packets within a 10 ms interval. Figure 5 shows the cumulative distribution function (CDF) of that number for a relative load $\rho \in \{0.95, 1.2\}$ on a link with a capacity of 1 Gb/s. We observe that the number of packets arrived within a 10 ms interval vary substantially around their means (dashed lines). Moreover, there are many 10 ms intervals with clearly less and more traffic arrived than what could be sent within that time (solid line). As a consequence, the generated traffic is bursty and leads to substantial packet queuing. The offered load $\rho = 0.95$ models moderate overload and $\rho = 1.2$ models severe overload. The discussed arrival processes are utilized in the experiments of Section V-C1 and Section V-C2.

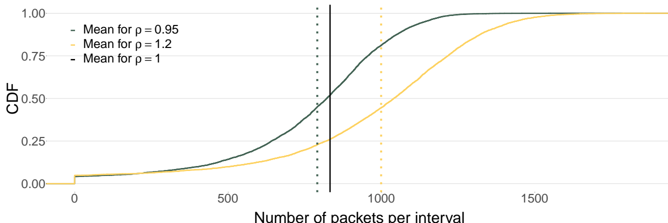


Fig. 5. Cumulative distribution function (CDF) of the number of packets arrived within a 10 ms interval; the LogNormal-distributed inter-arrival times A are set for a relative load of $\rho \in \{0.95, 1.2\}$ on a link with a capacity of $C = 1$ Gbit/s; the vertical lines correspond to mean rates and the link bandwidth.

b) Constant bit-rate traffic UDP traffic: Realtime traffic sent over UDP can be often modelled as periodic constant bit-rate (CBR) traffic. It is a typical candidate to benefit from the ABE traffic class. We leverage *iperf3.9* [24] for the generation of CBR traffic and send it over UDP. The packet size is $B = 1490$ bytes including all headers, and constant inter-arrival times are set to achieve a desired traffic rate.

c) Elastic TCP traffic: Most traffic on the Internet is transmitted over TCP which adapts its transmission rate to the congestion conditions in the network in order to avoid excessive packet loss. It is also called elastic traffic as it utilizes the available bandwidth. Various TCP versions exist and have different properties. Some react primarily to packet loss, e.g., TCP Cubic, others react to increased RTT, e.g., TCP BBR. We leverage *iperf3.9* [24] for the generation of TCP traffic. We utilize both TCP Cubic and TCP BBR by choosing appropriate Linux implementations in the VMs.

B. Validation of the Bandwidth Estimation Algorithm

In Section III-B5 we presented a new bandwidth estimation algorithm. For this study, we set its memory to $M = 50$ ms. We validate the method with the following experiment. We send non-adaptive traffic with bursts as described in Section V-A3 with an load of $\rho = 0.5$ on a link with 1 Gbit/s. The link bandwidth is set by tbf using a burst size of 10 MTUs like in all other experiments. After 5 seconds, the bottleneck decreases to 250 Mbit/s and changes back to 1 Gbit/s after 7 seconds.

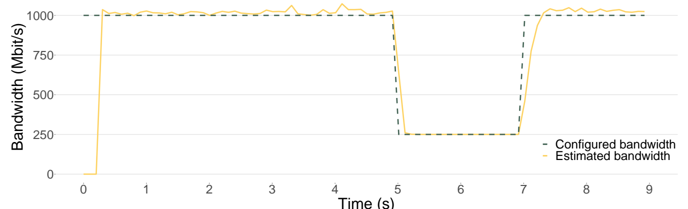
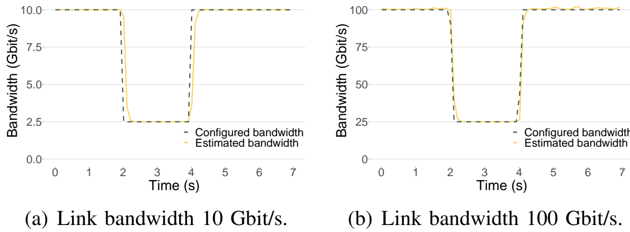


Fig. 6. Bandwidth estimation on a link where tbf-controlled bandwidth starts with 1 Gbit/s, it is 250 Mbit/s after 5 s, and changes back to 1 Gbit/s after 7 s. Non-adaptive traffic with bursts is sent at a rate of 500 Mbit/s.

Figure 6 shows that the estimated bottleneck matches the configured bandwidth very closely. The challenge is the adaptation at 7 s to a larger rate as the utilization is then only 50%. Then, backlogged packets do not occur often, but they are frequent enough as the algorithm leverages every single backlogged packet to update its estimate. Thus, the estimation method is very sensitive in the sense that it recognizes the correct rate even under moderate load. The algorithm works equally well with TCP traffic (32 flows) which is shown in

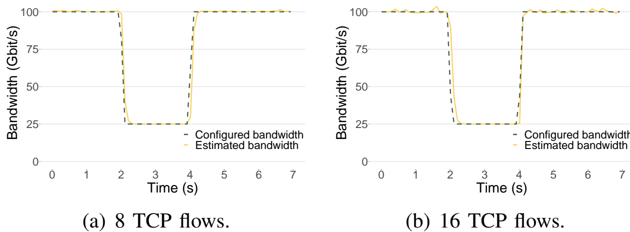
Figure 7(a) and Figure 7(b) for a link with 10 Gbit/s and 100 Gbit/s capacity. Again, the bottleneck changes to 25% of its original capacity after 2 seconds, and changes back after 4 seconds.



(a) Link bandwidth 10 Gbit/s. (b) Link bandwidth 100 Gbit/s.

Fig. 7. Bandwidth estimation with 32 TCP flows; the link bandwidth is throttled to 2.5 (25) Gbit/s between 2 s and 4 s.

The bandwidth estimation algorithm works equally well with 8 and 16 TCP flows[8] as shown in Figure 8(a) and Figure 8(b). With a lower number than 8 TCP flows, there is not sufficient congestion to trigger the bandwidth estimation algorithm. However, this is not a problem for the following reasons. First, less than 8 TCP flows in a 100 Gbit/s environment is rather unlikely. Second, in the absence of congestion, DSCD does not require the estimated bandwidth C. The estimated bandwidth is only used for credit devaluation after a congestion period.



(a) 8 TCP flows. (b) 16 TCP flows.

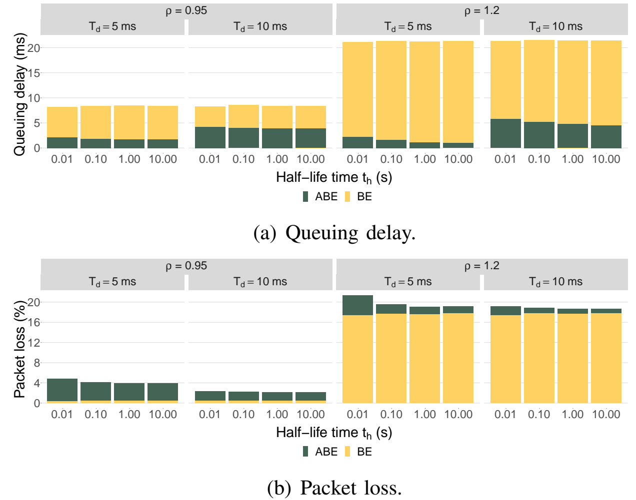
Fig. 8. Bandwidth estimation with 8 and 16 TCP flows; the link bandwidth is throttled to 25 Gbit/s between 2 s and 4 s.

The experiments show that the bandwidth estimation algorithm precisely measures the available bandwidth with the same parameterization (M = 50 ms) for a wide range of bottleneck speeds.

C. Performance of DSCD with Non-Adaptive Traffic with Bursts

We study packet delay and loss for non-adaptive traffic with bursts when being carried over BE and ABE. We first study the impact of DSCD parameters T_d, t_h and then the impact of ABE traffic rate at different link loads ρ.

1) Impact of DSCD's Configuration Parameters T_d and t_h: DSCD is configured with two parameters: the delay threshold T_d for ABE traffic and the half-life time t_h for credit devaluation. We examine their impact with the following experiment. We generate non-adaptive traffic with bursts as explained in Section V-A3 with an offered load of $\rho \in \{0.95, 1.2\}$. We randomly label 90% of the traffic as BE and 10% as ABE. We experiment with different delay thresholds T_d and half-life times t_h. Other parameters are set to the default values in Table IV. We study the experienced queuing delay and packet loss at the bottleneck node separately for ABE and BE traffic. Figure 9(a) and Figure 9(b) illustrate the results.



(a) Queuing delay.

(b) Packet loss.

Fig. 9. Queuing delay and packet loss of BE and ABE traffic for non-adaptive traffic with bursts. The relative overall load ρ, the delay threshold T_d, and the half-life time t_h are varying parameters; other parameters are set as in Table IV.

We first analyze the delay. Figure 9(a) shows that both BE and ABE traffic is only little delayed with moderate overload ($\rho = 0.95$), but ABE traffic experiences less delay than BE traffic. A lower delay threshold T_d reduces the delay for ABE traffic. For severe overload ($\rho = 1.2$), BE traffic is strongly delayed while ABE traffic sees similarly low delays as for moderate overload. Larger half-life times slightly reduce the delay for ABE traffic.

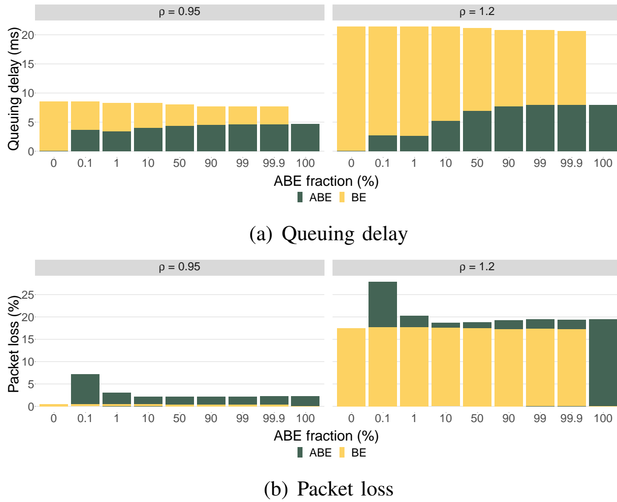
Now, we discuss the packet loss. Figure 9(b) shows that in the presence of moderate overload hardly any BE packets are lost while the packet loss probability for ABE traffic is 2%–4%. The lower the delay threshold T_d, the higher the packet loss. The additional packet loss of ABE is caused by the traffic model. With an offered load of $\rho = 0.95$ the traffic model results in either no congestion (empty queue) or congestion induced by bursts. In the case of no congestion, the stored ABE credit is devaluated with the link rate R. As a result, with $\rho = 0.95$ it is likely that ABE traffic is not able to "save" credit between bursts, i.e., each burst arrives at an empty ABE credit counter.

In case of severe overload, $\frac{1}{6}$ of the traffic cannot be carried due to missing capacity. Therefore, both BE and ABE traffic experience around 17% packet loss at packet enqueue. ABE traffic faces 1%–4% more packet loss than BE traffic, which is mostly due to exceeded deadlines. Larger half-life times slightly reduce the packet loss for ABE traffic. A half-life time of $t_h = 100$ ms leads to clearly less packet loss than $t_h = 10$ ms for severe overload. Longer half-life times lead only to minor improvements. Therefore, we recommend to set the half-life time to $t_h = 100$ ms. This will be confirmed in Section V-E for other reasons.

[8] As long as all TX queues are used.

2) Impact of ABE Traffic Rate: DSCD turns dropped ABE packets into a potential delay advantage for subsequent ABE packets. If no such packets arrive in time, packet drops cannot be leveraged by the ABE traffic class. This may happen in the presence of too little ABE traffic. Therefore, we study the impact of ABE traffic rate on loss and delay.

The experiments are designed similarly as those in Section V-C1. Non-adaptive traffic with bursts is used with the standard configuration of Table IV. We study again an offered load of $\rho \in \{0.95, 1.2\}$ and test different ABE traffic rates by varying the fraction of ABE traffic.



(a) Queuing delay

(b) Packet loss

Fig. 10. Queuing delay and packet loss of BE and ABE traffic for non-adaptive traffic with bursts. The relative overall load ρ and the fraction of ABE traffic are varying parameters; other parameters are set as in Table IV.

We first discuss the packet loss in Figure 10(b). In the absence of ABE traffic, there is no ABE packet loss and only the little BE packet loss is visible. A very small fraction of ABE traffic (0.1% $\approx$ 1 Mbit/s) results in high packet loss of almost 7% and 27% for moderate and severe overload. When the ABE traffic rate is low, packet inter-arrival times are large. When an ABE packet exceeds the delay threshold T_d and the queue threshold T_q, it is dropped but its credit is saved. However, the credit is devaluated over time, either exponentially or linearly. Therefore, saved credit is likely to be vanished by the arrival of the next packet due to the low ABE traffic rate. Then, the next packet may also experience normal queueing delay, exceed the delay threshold, and be lost again.

For non-responsive bursty traffic, the packet loss is very high at a rate of 1 Mbit/s (ABE fraction $\approx$ 0.1%) in spite of the queue threshold $T_q = 1$. In Section V-D2 we will show that this setting is able to prevent excessive packet loss for periodic traffic of that rate. Luckily, realtime traffic is usually periodic. Further, ABE packet loss decreases with a larger fraction of ABE traffic as subsequent packets arrive earlier and can leverage stored credit more efficiently. An ABE traffic rate of 10 Mbit/s (ABE fraction $\approx$ 1%) suffices to achieve significantly lower packet loss.

The packet loss for BE traffic slightly decreases with a larger ABE fraction. With a larger ABE fraction, more ABE traffic is dropped, which reduces load from the system. BE traffic benefits from that with slightly reduced packet loss. In the absence of BE traffic, there is no BE packet loss and only the ABE packet loss is visible.

Figure 10(a) shows that queuing delay for ABE increases with larger ABE fraction. ABE packets can only overtake BE packets. Therefore, an increasing amount of ABE leads to more non-skippable packets and hence to longer queuing delay. Nevertheless, the delay is below the delay threshold T_d. At the same time, BE queuing delay decreases for increasing ABE fraction. This is due to reduced traffic load in the system as more traffic is dropped with larger ABE fraction.

The experiments show that even large fractions of ABE traffic have no negative impact on the performane of BE traffic, which was a design goal for ABE. This is unlike Expedited Forwarding (EF) of the differentiated services framework (DiffServ) [8] where BE traffic suffers if the fraction of EF traffic is too large.

D. Performance of DSCD with Periodic Traffic and TCP Traffic

Now we assume that ABE traffic is periodic UDP traffic and BE traffic consists of TCP Cubic flows. This is a more realistic assumption as many realtime applications send periodic traffic. We first study packet delay and loss for different delay thresholds T_d, ABE traffic rates, and various numbers of TCP flows. Then we focus on small ABE traffic rates and show that the queue threshold $T_q = 1$ is the right means to prevent excessive packet loss.

1) Coexistence of Realtime and Elastic Traffic: We evaluate different sending rates R_{ABE} for ABE traffic and different numbers of TCP flows. We vary the delay threshold T_d and use the default settings from Table IV for other parameters.

(a) Queuing delay

(b) Packet loss

Fig. 11. Queuing delay and packet loss of periodic ABE traffic; ABE traffic rate R_{ABE}, number of TCP flows carried over BE, and delay threshold T_d are varying parameters; other parameters are set as in Table IV.

We first consider the ABE packet loss illustrated in Figure 11(b). For R_{ABE} = 300 kbit/s ABE packet loss is very low, for R_{ABE} = 1 Mbit/s it is large but almost independent of the delay threshold T_d, and for larger ABE traffic rates the packet loss depends on the delay threshold

T_d. This can be explained as follows. For a very low ABE traffic rate $R_{ABE} = 300$ kbit/s, the inter-arrival time of periodic ABE packets is $\frac{1490\text{bytes}\cdot 8\text{ bits}}{300\text{ kbit/s}} = 39.7$ ms. Thus, ABE packets cannot meet previous ABE packets in the queue so that the queue threshold $T_q = 1$ saves them from being dropped due to a passed deadline. Hence, dropping is turned off for ABE traffic and DSCD behaves like a FIFO queue. This is different for LogNormal-distributed inter-arrival times where $T_q = 1$ cannot prevent excessive packet loss that effectively (see Section V-C2). For an ABE traffic rate of $R_{ABE} = 1$ Mbit/s, the inter-arrival time of ABE packets is $\frac{1490\text{bytes}\cdot 8\text{ bits}}{1\text{ Mbit/s}} = 11.92$ ms. Thus, if an ABE packet arrives and meets another ABE packet, that packet will be dropped as it is 11.92 ms old and has exceeded any of the considered delay thresholds $T_d \in \{2, 5, 10\}$ ms. For ABE traffic rates of $R_{ABE} = 3$ Mbit/s or larger, the inter-arrival time of the packets is $\frac{1490\text{bytes}\cdot 8\text{ bits}}{3\text{ Mbit/s}} = 3.58$ ms or smaller. This is short enough so that delay thresholds T_d have an impact on packet loss and lead to different system behaviour. The number of TCP flows influences the congestion level which has also an impact on the packet loss. We observe packet loss values between 0.5% and 1.4% depending on the specific setting. While the number of TCP flows has a non-monotonic impact on packet loss, smaller delay thresholds lead to more packet loss.

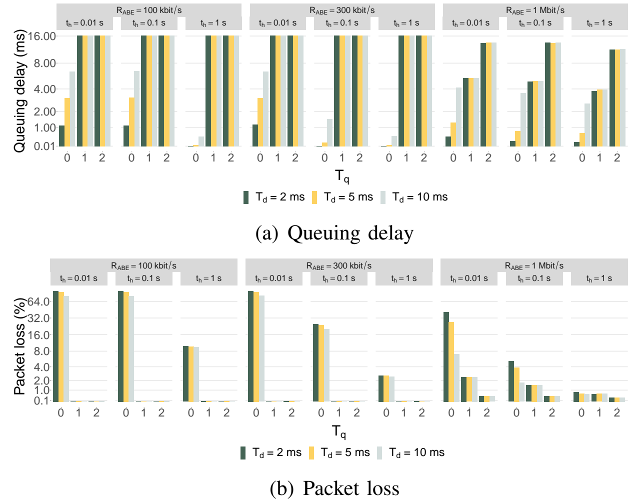
The behaviour of the queuing delay in Figure 11(a) is roughly inverse to the packet loss. For $R_{ABE} = 300$ kbit/s, the queuing delay is about 16 ms which is about the same as for BE traffic. It is lower for $R_{ABE} = 1$ Mbit/s, but it is the same for the different delay thresholds T_d. And for larger ABE traffic rates, the queuing delay clearly decreases with the delay threshold. The number of TCP flows has only a minor impact on the queuing delay of ABE traffic.

2) Impact of the Queue Threshold T_q: Algorithm 2 utilizes a queue threshold T_q to prevent excessive packet loss for low ABE traffic rates. We show that $T_q = 1$ is the appropriate parameter.

We consider various low rates R_{ABE} of periodic ABE traffic and 64 TCP Cubic background flows. We study different half-life times t_h, delay thresholds T_d, and queue thresholds T_q. To obtain reliable results for $R_{ABE} \in \{100, 300\}$ kbit/s, we extend the data collection time to 280 s. Figures 12(a) and 12(b) compile results for packet loss and delay.

In Figure 12(b), we observe for a queue threshold of $T_q = 0$ very high packet loss which is almost the same for any delay threshold T_d. Only for $R_{ABE} = 1$ Mbit/s and $T_q = 0$ the packet loss decreases with increasing half-life time t_h. In contrast, a queue threshold of $T_q \in \{1, 2\}$ keeps the packet loss very low for $R_{ABE} \in \{100, 300\}$ kbit/s and to moderate values for $R_{ABE} = 1$ Mbit/s. Thus, $T_q \in \{1, 2\}$ turns off traffic differentiation in the presence of small ABE traffic aggregates, which saves them from excessive packet loss. For ABE traffic rate $R_{ABE} = 1$ Mbit/s, the packet loss for $T_q = 1$ is larger than the one for $T_q = 2$.

We now discuss the queuing delay in Figure 12(a). For $T_q = 0$, queuing delay is low and scales with the delay threshold T_d. However, that is at the expense of excessive packet loss in case of $R_{ABE} \in \{100, 300\}$ kbit/s. For $T_q \in \{1, 2\}$, the queuing delay is as large as the one of BE traffic as service differentiation is turned off. For $R_{ABE} = 1$ Mbit/s, $T_q = 1$ leads to clearly lower delay than $T_q = 2$ which still turns off service differentiation. Thus, $T_q = 1$ is the appropriate value to save small ABE traffic aggregates from excessive packet loss and enable traffic differentiation for ABE traffic rates of $R_{ABE} = 1$ Mbit/s or larger.



(a) Queuing delay

(b) Packet loss

Fig. 12. Queuing delay and packet loss of periodic ABE traffic in the presence of 64 TCP flows via BE; delay threshold T_d, queue threshold T_q, and half-life time t_h are varying parameters; other parameters are set as in Table IV.

E. The Need for Exponential Decay

Exponential credit decay over time may increase packet loss. Nevertheless, it is helpful for several reasons. First, without exponential decay, a selfish user may send a large burst of redundant ABE data to accumulate credit for later use. When then relevant ABE data is transmitted, it can be sent with low delay thanks to stored credit. Exponential decay of stored credit largely removes the incentive for this selfish behavior. Second, the sum of credits in the system is limited (see Section III), which may lead to packet drop at enqueue. Therefore, stored, unused credit essentially shortens the queue and can cause packet loss although the physical queue is not full. Credit decay frees the system from stored credit over time and thereby extends the queue capacity towards normal. Third, adaptive protocols such as TCP may benefit from shorter delay of the ABE class under some conditions. Then, TCP flows over ABE may achieve a larger goodput than TCP flows over BE due to shorter perceived RTTs. As a consequence, ABE traffic may suppress BE traffic. However, a design goal of ABE is to avoid that. We show that exponential decay helps to achieve that goal.

We perform the following experiment. A single TCP Cubic flow via ABE competes against multiple TCP Cubic flows via BE. We measure the ABE flow's relative goodput compared to the average goodput of the BE flows for different half-life times t_h and for different RTTs. Table V shows the results. A relative goodput above 100% indicates that ABE has an unfair bandwidth share.

TABLE V

GOODPUT OF A SINGLE TCP FLOW CARRIED OVER ABE RELATIVE TO THE AVERAGE GOODPUT OF MULTIPLE TCP FLOWS CARRIED OVER BE. CUBIC IS USED AS TCP VARIANT; THE HALF-LIFE TIME t_h, THE NUMBER OF BE FLOWS, AND THE RTT ARE PARAMETERS.

RTT	#BE flows	t_h			
		10 ms	100 ms	1 s	∞
10 ms	16	2%	99%	167%	201%
	32	2%	129%	198%	226%
	64	4%	157%	244%	267%
	128	6%	164%	281%	329%
30 ms	16	1%	8%	77%	114%
	32	1%	4%	105%	148%
	64	2%	5%	115%	187%
	128	4%	10%	114%	189%

Without credit decay ($t_h = \infty$), the ABE flow takes a clearly unfair traffic share between 114% and 329%. It is larger for a RTT of 10 ms than for a RTT of 30 ms, and it increases with an increasing number of BE flows. For comparison, $t_h = 1$ s causes relative goodputs between 77% and 281%. For $t_h = 100$ ms the relative goodputs are between 99% and 164% in case of a very low RTT of 10 ms, and between 4% and 10% for larger RTT. When the half-life time is too short ($t_h = 10$ ms), the ABE flow achieves only little goodput (< 6%) as credit decays so fast that subsequent packets cannot profit from it sufficiently. Thus, a half-life time of $t_h = 100$ ms limits the unfairness caused by TCP to a moderate degree and leads only to moderate packet loss for ABE traffic (see Section V-D). Therefore, $t_h = 100$ ms is a preferred configuration value for the half-life time.

F. Inter-Protocol and Inter-Class Unfairness of TCP Variants

There is a large number of TCP variants which do not necessarily share bandwidth in a fair manner as they implement different congestion control algorithms. We call this inter-protocol unfairness. In Section V-E we have already shown that flows with the same TCP variant can share bandwidth in an unfair manner when carrying traffic over both ABE and BE. We call this inter-class unfairness. In the following, we first quantify the inter-protocol unfairness between TCP Cubic and TCP BBR. Then we investigate the inter-class unfairness of both TCP variants separately under various networking conditions. We use the default parameters of Table IV in all experiments.

1) Inter-Protocol Unfairness between TCP Cubic and TCP BBR: Inter-protocol unfairness is a well-known phenomenon [25], [26]. We illustrate it in the following experiment. An equal number of TCP Cubic and TCP BBR flows is carried over a single link and we vary the number of flows and the RTT. All traffic is carried over BE. We take the relative goodput of TCP BBR vs. TCP Cubic as a measure of unfairness. Figure 13 shows the results. For low RTT (10 ms), the goodput of BBR is about 3 times the goodput of Cubic. The number of flows has only a secondary impact. For larger RTT (30 ms and 100 ms), the goodput of BBR is 30–100 times larger than the one of Cubic. Thus, inter-protocol unfairness of existing TCP variants can be enormous.

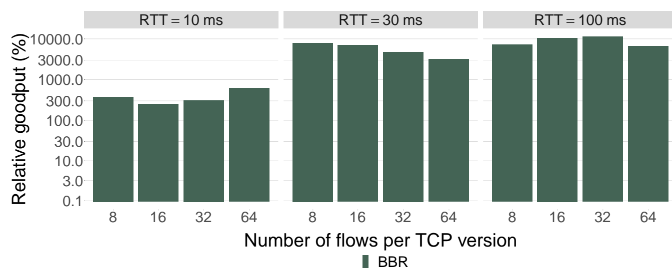
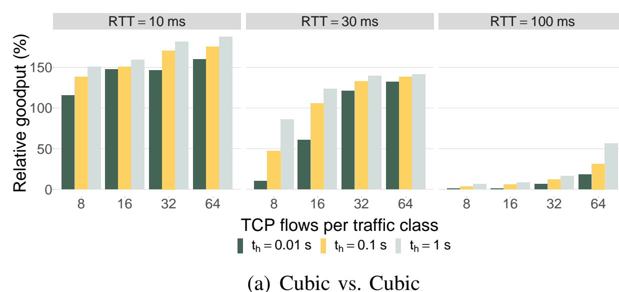
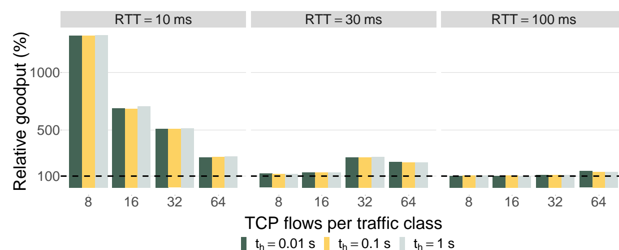


Fig. 13. Goodput of TCP BBR flows relative to the goodput of TCP Cubic flows when being carried over BE; TCP BBR and TCP Cubic have the same number of flows which is a varying parameter as well as the RTT; other parameters are set as in Table IV.

2) Inter-Class Unfairness with TCP Cubic: To quantify inter-class unfairness, we transmit the same number of TCP Cubic flows via ABE and via BE in the system. Apart from that, the experiment setup is the same as before [9]. Figure 14(a) shows the relative goodput for TCP Cubic via ABE vs. TCP Cubic via BE.



(a) Cubic vs. Cubic



(b) BBR vs. BBR

Fig. 14. Goodput of TCP flows via ABE relative to goodput of TCP flows via BE. The experiment is carried out for TCP Cubic and TCP BBR; BE and ABE carry the the same number of flows which is a varying parameter as well as the RTT; other parameters are set as in Table IV.

For an RTT of 10 ms, the relative goodput of ABE vs. BE is between 100% and 185%. The unfairness increases with the number of flows in the system and with increasing half-life time t_h. It is significantly lower than the inter-protocol unfairness between TCP Cubic and TCP BBR for the same RTT. For an RTT of 30 ms, the relative goodput decreases and is clearly below 100% if the number of competing flows is low. For large RTT of 100 ms, the relative goodput is generally below 100%, i.e., TCP senders cannot obtain an unfair traffic share when transmitting over ABE. This is an interesting result as inter-class unfairness is a particular issue at short RTTs

[9]This experiment is slightly different than the similar experiment series in Section V-D where a single ABE flow competes against multiple BE flows.

while inter-protocol unfairness is a particular issue at longer RTTs (see Section V-F1).

We argue why the inter-class unfairness occurs and why the behavior depends on the RTT. Cubic adapts its congestion window based on a cubic function and is mainly influenced by its experienced packet loss. While the congestion window growth of Cubic is independent of the RTT, it still relies on the RTT for timeout calculation. The timeout implicitly affects a parameter for the congestion window growth function. ABE flows experience a relatively lower end-to-end delay (RTT + queuing delay) than BE flows resulting in a higher goodput. This has also been shown in [27], where a smaller RTT leads to higher throughput compared to other Cubic flows with higher RTT. The relative delay advantage vanishes with higher RTTs.

3) Inter-Class Unfairness with TCP BBR: We now look at the inter-class unfairness with TCP BBR. We conducted similar experiments whose results are compiled in Figure 14(b). For small RTT of 10 ms, the relative goodput for ABE flows is between 200% and 1400% depending on the number of flows. The impact of the half-life time t_h is low. Increasing the RTT leads to lower relative goodputs for ABE flows between 100% and 200%. This is a different behaviour than with TCP Cubic. Thus, in case of a predominant deployment of TCP BBR, ABE BBR flows could partly suppress BE BBR flows. However, the problem of BBR suppressing other TCP variants in the current BE Internet is larger and shows that too aggressive congestion control algorithms can be problematic.

The reason why BBR benefits so much from ABE, even at large RTTs, is that its congestion control algorithm does not react to packet loss, which is unlike TCP Cubic. It rather reduces its transmission rate when it notices an increase in the RTT [28]. As, BBR flows via ABE see shorter and more stable end-to-end RTTs due to less queueing delay, they benefit from ABE at any RTT and do not suffer too much from experienced packet loss. The behavior of BBR shows that concepts such as RTT-fairness and influence of AQMs must be considered in the design of congestion control algorithms. As ABE is primarily designed for realtime traffic – and therefore UDP – ABE may be limited to UDP traffic to prevent ABE BBR from suppressing other BE BBR flows. However, this will not work for QUIC-based transport protocols.

VI. Summary and Discussion

We summarize this work and discuss the findings.

A. Novelties of DSCD

The objectives of DSCD are similar to those of DSD and DSF but its properties differ in important aspects.

(1) DSCD has only moderate complexity. A Linux kernel implementation demonstrates its feasibility of 100 Gbit/s links.

(2) DSCD copes with unknown and varying bandwidth while DSD and DSF require a static link bandwidth C for deadline computation. DSCD also measures the link bandwidth C but needs it only to drain credits in the absence of congestion, which is a rather uncritical process.

(3) The conception of ABE is problematic for low rates of ABE traffic. If a packet is dropped due to exceeded delay, there may be no subsequent packet that could leverage that loss for an delay advantage when the queue has been flushed by the next packet arrival. Therefore, ABE traffic aggregates may experience large packet loss with other scheduling algorithms. DSCD prevents this by dropping ABE packets only if there are also other ABE packets in the queue, which essentially turns off service differentiation at low ABE traffic rates.

(4) With DSCD, stored credit decays exponentially over time with half-life time t_h. This avoids that credit can be stored arbitrarily long during a congestion phase. It avoids incentives for selfish users to send more traffic than needed.

(5) Existing algorithms spent lots of effort to pursue approximate fairness for flows sent over BE and ABE. We intend ABE primarily for realtime traffic and not for bulk traffic. Therefore, TCP over ABE may obtain a worse service than TCP over BE. Our objective is even a worse service for TCP over ABE because TCP over ABE should not be able to suppress TCP over BE in the same network. DSCD's exponential decay for stored credit helps to achieve that goal.

B. Performance

We tested DSCD scheduling for BE and ABE traffic using non-responsive traffic with bursts, periodic and TCP traffic, as well as TCP traffic with different variants. We showed that the delay threshold T_d controls the queuing delay for ABE traffic. We recommend to set it to $T_d = 10$ ms as lower values lead to larger packet loss. The queue threshold T_q controls the packet loss and turns of service differentiation in the presence of low ABE traffic rates that are smaller than 1 Mbit/s. The experimental results show that $T_q = 1$ is a good value. The half-life time controls how long credit can be stored so that packet loss and delay decrease with increasing half-life time t_h. If it is too large, then TCP over ABE can obtain significantly larger goodput than TCP over BE under some conditions. Setting $t_h = 100$ ms leads to moderate packet loss and only little inter-class unfairness.

Finally, we quantified inter-protocol and inter-class unfairness (see Section V-F) for multiple scenarios. TCP BBR flows can suppress TCP Cubic flows when being carried over BE, in particular for long RTTs. TCP Cubic flows via ABE can suppress TCP Cubic flows via BE, in particular for short RTTs and the problem vanishes for long RTTs. TCP BBR flows via ABE can suppress TCP BBR flows via BE, also in particular for short RTTs. For long RTTs the advantage diminishes but does not fully disappear. This is mainly the problem of BBR's congestion control as it also causes the observed inter-protocol unfairness.

VII. Conclusion

Alternative Best Effort (ABE) is an alternative traffic class for the Internet. ABE traffic experiences shorter delay than Best Effort (BE) traffic at the expense of more packet loss. This must be achieved without delaying and dropping BE traffic compared to the transmission of the entire traffic with a single FIFO queue.

In this work, we addressed the fundamental question whether an ABE service class is technically feasible, how it

behaves with up to date transport protocols, and whether it can be implemented on modern hardware. To that end, we proposed DSCD as an algorithm for combined scheduling of BE and ABE traffic. We implemented it in the Linux network stack and it is fast enough for 100 Gbit/s links. Side products are an approximation of the exponential function in the kernel, which is useful for moving average computations, and a bandwidth estimation method that works well even at moderate link utilization. We used a virtualized hardware testbed to study the impact of DSCD on packet loss and delay for both BE and ABE traffic under various conditions. ABE traffic faces significantly shorter delay but more packet loss than BE traffic provided that a critical mass of ABE traffic is available (≈ 1 Mbit/s). Otherwise we see approximate FIFO behaviour so that BE and ABE receive a similar service. Under all conditions, the service for BE traffic is not degraded by design. We recommended configuration parameters for DSCD so that packet loss for ABE traffic remains small and that TCP does not get an unfairly large traffic share when sending over ABE.

ABE may be useful for Internet service providers to offer their customers a low-delay traffic class that does not harm other traffic. It may be attractive for net-neutral service differentiation, and it may serve as a bridge towards a low-delay Internet. In future work, DSCD could be implemented with network acceleration techniques such as smart NICs or the Metron platform [29] [30] for higher performance.

Acknowledgement

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/2-1. The authors alone are responsible for the content of the paper.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *ACM Queue*, vol. 9, no. 11, Nov. 2011.

[2] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool, "The Effects of Packet Loss and Latency on Player Performance in Unreal Tournament 2003," in *ACM SIGCOMM Workshop on Network and System Support for Games*, 2004.

[3] S. Liu, M. Claypool, A. Kuwahara, J. Scovell, and J. Sherman, "The Effects of Network Latency on Competitive First-Person Shooter Game Players," in *International Conference on Quality of Multimedia Experience (QoMEX)*, 2021.

[4] A. D. Domenico, G. Perna, M. Trevisan, L. Vassio, and D. Giordano, "A Network Analysis on Cloud Gaming: Stadia, GeForce Now and PSNow," *MDPI Network*, vol. 1, no. 3, 2021.

[5] X. Zhang, H. Chen, Y. Zhao, Z. Ma, Y. Xu, H. Huang, H. Yin, and D. O. Wu, "Improving Cloud Gaming Experience through Mobile Edge Computing," *IEEE Wireless Communications*, vol. 26, no. 4, 2019.

[6] A. Wahab, N. Ahmad, M. G. Martini, and J. Schormans, "Subjective Quality Assessment for Cloud Gaming," *MDPI J*, vol. 4, no. 3, 2021.

[7] "ITU-T Recommendation G.107 : The E-Model, a computational model for use in transmission planning," ITU, Tech. Rep., 2015.

[8] S. Blake, D. L. Black, M. A. Carlson, E. Davies, Z. Wang, and W. Weiss, "RFC2475: An Architecture for Differentiated Services," Dec. 1998.

[9] B. D. et al., "RFC3246: An Expedited Forwarding PHB (Per-Hop-Behavior)," Mar. 2002.

[10] P. Hurley and J.-Y. Le Boudec, "The Alternative Best-Effort Service," https://tools.ietf.org/html/draft-hurley-alternative-best-effort, Jun. 2000.

[11] V. Stocker, G. Smaragdakis, and W. Lehr, "The State of Network Neutrality Regulation," *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 1, 2020.

[12] P. Hurley, J.-Y. Le Boudec, P. Thiran, and M. Kara, "ABE: Providing a Low-Delay Service within Best Effort," *IEEE Network Magazine*, vol. 15, no. 3, May 2001.

[13] M. Karsten, D. S. Berger, and J. Schmitt, "Traffic-Driven Implicit Buffer Management - Delay Differentiation without Traffic Contracts," in *International Teletraffic Congress (ITC)*, Sep. 2016.

[14] J. You, M. Welzl, B. Trammell, M. Kuehlewind, and K. Smith, "Latency Loss Tradeoff PHB Group," https://tools.ietf.org/html/draft-you-tsvwg-latency-loss-tradeoff, Mar. 2016.

[15] T. Høiland-Jørgensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290, 2018.

[16] G. Ramakrishnan, M. Bhasi, V. Saicharan, L. Monis, S. D. Patil, and M. P. Tahiliani, "FQ-PIE Queue Discipline in the Linux Kernel: Design, Implementation and Challenges," in *IEEE Conference on Local Computer Networks (LCN)*, 2019.

[17] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways," in *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2018.

[18] M. Podlesny and S. Gorinsky, "RD Network Services: Differentiation through Performance Incentives," in *ACM SIGCOMM*, Aug. 2008.

[19] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter When You Can JUMP Them!" in *USENIX Syposium on Networked Systems Design & Implementation (NSDI)*, 2015.

[20] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I.-J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl, "Reducing Internet Latency: A Survey of Techniques and Their Merits," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, 2016.

[21] M. Menth and F. Hauser, "On Moving Averages, Histograms and Time-DependentRates for Online Measurement," in *International Conference on Performance Engineering (ICPE)*, 2017.

[22] "QDisc: Token Bucket Filter." [Online]. Available: https://man7.org/linux/man-pages/man8/tc-tbf.8.html

[23] S. Hemminger, "Network emulation with NetEm," *Linux Conf Au*, vol. 844, 2005.

[24] iperf3 team, "iperf3." [Online]. Available: http://software.es.net/iperf/

[25] M. Hock, R. Bless, and M. Zitterbart, "Experimental Evaluation of BBR Congestion Control," in *IEEE International Conference on Network Protocols (ICNP)*, 2017.

[26] Y. Cao, A. Jain, K. Sharma, A. Balasubramanian, and A. Gandhi, "When to Use and When Not to Use BBR: An Empirical Analysis and Evaluation Study," in *Internet Measurement Conference*, 2019.

[27] T. Kozu, Y. Akiyama, and S. Yamaguchi, "Improving RTT Fairness on CUBIC TCP," in *International Symposium on Computing and Networking*, 2013.

[28] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, Sep. 2016.

[29] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV Service Chains at the True Speed of the Underlying Hardware," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[30] G. P. Katsikas, T. Barbette, D. Kostić, J. G. Q. Maguire, and R. Steinert, "Metron: High-Performance NFV Service Chaining Even in the Presence of Blackboxes," *ACM Transactions on Computer Systems*, vol. 38, no. 1–2, 2021.

Steffen Lindner is a Ph.D. student at the chair of communication networks of Prof. Dr. habil. Michael Menth at the Eberhard Karls University Tuebingen, Germany. He obtained his master's degree in 2019 and afterwards, became part of the communication networks research group. His research interests include software-defined networking, P4 and congestion management.

Gabriel Paradzik is a Ph.D. student at the Eberhard Karls University Tuebingen, Germany. He started his Ph.D. in April 2021 at the communication networks research group. His research interests include congestion management and data center networking.

Michael Menth, (Senior Member, IEEE) is professor at the Department of Computer Science at the University of Tuebingen/Germany and chairholder of Communication Networks since 2010. He studied, worked, and obtained diploma (1998), PhD (2004), and habilitation (2010) degrees at the universities of Austin/Texas, Ulm/Germany, and Wuerzburg/Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, as well as resource and congestion management. His recent research focus is on network softwarization, in particular P4-based data plane programming, Time-Sensitive Networking (TSN), Internet of Things, and Internet protocols. Dr. Menth contributes to standardization bodies, notably to the IETF.