How to Network Delay-Sensitive Applications

Pavel Chuprikov
USI Lugano, Switzerland
0000-0002-6673-1143

Kirill Kogan[†]
Ariel University, Israel
0000-0001-5384-1899

***Abstract*—In this paper we study design principles of congestion control supporting low-latency applications. In this regard, we show the attractiveness of a *bounded-delay* model satisfying latency constraints instead of optimizing them. Under this model, we propose two transformations automatically generating policies with finite buffers from policies with infinite buffers while keeping performance guarantees. These transformations allow us to reconsider design principles of congestion control, in particular, avoiding delivery of unnecessary traffic. We study the impact of different policy properties on required buffer sizing and at the extreme case define potential ways to build reliable transports without retransmissions. In addition, we build a taxonomy of management policies for various types of extra knowledge and propose another transformation kind constructing policies that optimize weighted goodput from policies optimizing throughput while keeping performance guarantees. Our analytic results are supported by extensive evaluations demonstrating attractiveness of the proposed design principles.**

I. Introduction and motivation

Recent years have seen a bunch of papers proposing congestion controls for datacenter transports supporting low-latency applications [1], [2], [3], [4], [5]. Very quickly it has become clear that to satisfy aggressive low-latency constraints, design principles of congestion controls should be reconsidered. In this regard, there are several fundamental questions to be raised. What is a service model that should be implemented? Currently, in the case of TCP, all packets, even those that do not satisfy latency requirements, are still reliably transmitted. Most likely these packets will be useless once delivered to low-latency applications.

The second interesting question is related to buffer sizing and its management. The previous common perception was that while bigger buffers can absorb longer and more intensive bursts (in the case of reliable service models also reducing retransmissions), they are not appropriate for low-latency applications. This observation is mostly valid for FIFO-based buffer management inside switches, but not for advanced processing orders transmitting "more desirable" traffic first. In the latter case, infinite buffers can play their role and avoid retransmissions of packets dropped due to congestion while still implementing low-latency requirements. Even with infinite buffers, what are the right processing orders that should be deployed (e.g., shortest remaining processing time, some kind of deadlines, virtual values, etc.)? Clearly, infinite buffers are infeasible in practice but what are those factors allowing to achieve comparable performance with finite buffers? Can these finite buffer sizes be implemented in practice? Requiring ultra low latency under $1\,\mu s$ raises additional feasibility questions: should reliability be implemented through retransmissions? Instead of proposing new transports and comparing their performance on typical workloads, we consider the questions raised above and propose general design principles for congestion control mechanisms with analytic guarantees.

Our contribution: **(1)** In Sections II and III, we carefully show suitability of a *bounded delay* model [6], where every packet p is prepended with a packet value and a maximal offset in time (*slack*), by the end of which, p should be delivered to the application. Thus, instead of optimizing latency, we propose to satisfy latency constraints. This avoids retransmission of unnecessary packets with "expired" slacks and saves extra bandwidth for "preferable" traffic. Advanced processing orders starve less preferable packets inside switch buffers, and these packets naturally die due to expired slacks, thus preventing less preferable traffic from advancing towards congested hotspots. As a result, we claim that congestion control mechanisms should be implemented under mechanisms supporting reliable communication. **(2)** As a by-product, packet slacks implicitly restrict buffer sizes. In Section IV, we analytically study this dependency and introduce transformations allowing to automatically convert online buffer management policies with infinite buffers to online buffer management policies with finite buffers by keeping exactly the same performance guarantees. In addition, we show which properties of processing orders can significantly reduce buffer sizes. The proposed transformations in Section IV show potential ways to achieve reliable communication on the transport layer without retransmissions. **(3)** In Section V, we build a taxonomy of management policies exploiting additional levels of extra knowledge related to retransmission behaviors and compare their impact on overall performance. It can help researchers better understand tradeoffs of different congestion control implementations: "coupled" *vs.* "decoupled", centralized with global view *vs.* fully distributed local. **(4)** In Section VI, we discuss additional universal transformations for the bounded-delay model automatically constructing an online policy optimizing *weighted goodput* from a given online policy optimizing *weighted throughput* preserving performance guarantees. Again our major goal is not to find a "winning" transport but to show attractiveness of the proposed design principles.

The work of Pavel Chuprikov was supported by Swiss National Science Foundation, grant #200021_192121, FORWARD. The work of Kirill Kogan was supported in part by the Israeli Innovation Authority under the Knowledge Transfer Commercialization Program (MAGNETON) file no. 71249, in part by the Ariel Cyber Innovation Center in cooperation with the Israel National Cyber Directorate in the Prime Minister's Office.

[†] Dr. Kirill Kogan died on March 13, 2021 from complications of the COVID-19 disease.

ISBN 978-3-903176-39-3© 2021 IFIP

II. Reliability and delay-sensitive applications

As network resources are limited, *there will always be some level of congestion*, so there will be data that is *not* delivered on time. Thus, for better utilization of resources, the network should *only* be concerned with data that is not yet overdue. This has already been indirectly captured by applying to delay-sensitive applications a different (from non-delay-sensitive) set of metrics, e.g., tail flow completion time (see Homa [5]) and application-level throughput (see D^{2}TCP [7] and D^3 [8]).

A. Adapting existing service models

Traditional *reliability-first* network service models are not well-suited for this goal. To guarantee reliable delivery, they suggest to keep trying to deliver data, no matter how late that data is and whether the application still wants its delivery. If we adapt the traditional service model to delay-sensitive applications by guaranteeing *reliability only as long as delay requirements are satisfied,* we potentially *save network resources* on several levels: state at end hosts, link bandwidth in the network, and buffer space at network devices. In what follows, we discuss how the network may understand that delay requirements are no longer being satisfied—a key precursor to implementing the adapted service model.

B. Knowing when data becomes irrelevant

Applications are the entities that have true understanding of when a particular piece of data becomes irrelevant, such understanding must then reach the network. While there are several options, applications may supply at connection establishment an explicit *slack*—the time frame by the end of which data has to be delivered. Though this requires applications to preform slack estimation, but it allows for a longer decision window, improving optimization opportunities on the network side. Moreover, for low latency applications with less than half-RTT delay requirements, explicit per-packet slack is the *only* way for the information to reach in-network devices on time. For these reasons, our focus, thereafter would be on in-network, packet-level slacks, studied earlier by Kesselman et al. [6].

The slack estimation needed for the chosen approach can already be derived for many delay-sensitive applications from a bound on overall system response time and desired processing time [9], which are derived, respectively, from user studies and the complexity (content) of the request. In addition, infrastructure can infer delay requirements in some cases [10]. In subsequent sections we address the design implications of using the bounded-delay model, and, in particular, its relation to end host control logic and buffer management.

III. Design implications of bounded delay model

When application delay requirements are available to the network as a slack, data transmissions must be scheduled properly to optimize the desired objective. There are two components that come into play when we analyze scheduling: control logic at end-hosts and buffer management policies.

A. Congestion control at end hosts

A part of end-host control logic that is notoriously hard to get right is the congestion control. Congestion control decisions must be made under incomplete and delayed information, which, in relation to delay-sensitive applications, appears to be even more challenging due to very limited reaction window.

Before accepting the challenge, let us make clear, why congestion control was needed in the first place. Without congestion control, reliability requirements would force the network to keep retransmitting data, which anyway would not be able to make it due to downstream congestion. Retransmissions are required to maintain reliability under the following events: buffer overflows due to congestion and packet corruptions. The latter are exceptionally rare in modern data centers, so we ignore them in this work by assuming either that we can tolerate a tiny fraction of packets missing their deadlines due to corruption or that receivers explicitly request retransmission of corrupted packets (e.g., as in [5]). Buffer overflows we can "easily" avoid by placing *"infinite" buffers* at *sufficiently flexible* switches. In order to explain the details rigorously we introduce a formal model.

B. Introduction to the bounded delay model

Our model follows the bounded delay model of [6]. In particular, we assume that every packet arriving at a switch has an intrinsic value $v(p)$ and a slack $\mathrm{sl}(p)$; the objective is to maximizes the total value of packets transmitted *on time:*

$$\textsc{WSwTput} = \sum\nolimits_i v(p_i) \cdot [\mathrm{del}_i(p_i) \leq \mathrm{sl}(p_i)], \qquad (1)$$

where p_i is the ith packet transmitted by the switch, and $\mathrm{del}_i(p)$ is the time between ith transmission event and p's first arrival; $\mathrm{sl}(p)$ is also defined with respect to p's first arrival.

The buffering model is a single queue that holds at most B packets. The queue is managed by a buffer management algorithm ALG. Time is slotted, and we split every time slot into two phases: 1) *arrival:* a set of packets arrives, and ALG must ensure that at most B packets remain in the queue; 2) *transmission:* ALG transmits *one* packet from the queue. We assume $\mathrm{sl}(p)$ and $\mathrm{del}(p)$ to be in time slots, and that queue drops p automatically at the end of the timeslot where $\mathrm{del}(p) = \mathrm{sl}(p)$. For a given sequence of arrivals σ, an algorithm ALG, and an objective function ω, we denote by $\omega(\mathtt{ALG}, \sigma)$ the value of ω earned by ALG's when run on the sequence σ. We would be mostly interested in $\omega = \textsc{WSwTput}$.

C. Choosing processing orders for infinite buffers

At a first glance, infinite buffers seem to contradict our main motivation—support for delay-sensitive applications. If packets follow the FIFO processing order, then bigger buffers, indeed, lead to higher queuing delays and, as a result, are not appropriate for low latency applications. In contrast, non-FIFO advanced processing orders (e.g., priority based as in pFabric [2]) allow to transmit "more critical" packets while absorbing longer and more intensive bursts with bigger buffers.

The FIFO constraint actually makes algorithms provably weaker w.r.t. WSwTput. Let us say that an algorithm A

α*-dominates* an algorithm A' if for each input, A performs no worse than A', and there is at least one input where A performs α times better w.r.t. a given objective. An algorithm A is *equivalent* to A' if A 1-dominates A' and vice versa. To keep track of the buffer size, we denote by $A[B]$ an algorithm operating on a buffer of size B and by $A[\infty]$—an algorithm with an infinite buffer. The following fact quantifies the potential gain for non-FIFO processing orders.

Theorem 1. *For* WSwTPUT *objective and* $B \geq 2$, $OPT[B]$ 1.5*-dominates* $\texttt{OPT}_{\text{FIFO}}[B]$*, where* $\texttt{OPT}_{\text{FIFO}}[B]$ *is the optimal offline FIFO algorithm, and* $\texttt{OPT}[B]$ *is the optimal offline algorithm with no restrictions.*

Proof: On the first time slot two packets arrive: p and q, s.t. $v(p) = v(q) = 1$, $\text{sl}(p) = 2$ and $\text{sl}(q) = 1$. On the second time slot only packet r arrives, s.t. $v(r) = 1$ and $\text{sl}(r) = 0$. FIFO algorithm cannot process all three packets because if it processes r, then it cannot process either p or q. Thus it can earn at most 2, while the priority algorithm can earn 3. ■

D. The infinite-buffers assumption

Assuming that buffers are infinite does not automatically make *all* the logic at end hosts redundant. First, we already discussed corruption-induced retransmissions, they still remain. Second, end hosts can maintain per-flow state useful for scheduling purposes, which is too expensive to keep on each and every switch. Nonetheless, by combining two observations 1) the network has infinite memory for storing packets, and 2) the packet's "timer" starts ticking since the moment of packet's creation, we conclude end-host rate adjustment logic unnecessary for both congestion and flow control.

Neither infinite buffers imply that scheduling is trivial and *all* delay requirements can be satisfied. Since link capacities are still bounded, there are only so many packets that can be transmitted before their slacks start to expire, hence, some drops are unavoidable. For instance, let the link rate be $10\,\text{Gbit/s}$ and assume there *simultaneously* arrive 10 packets of size $1500\,\text{B}$ whose deadlines are all $10\,\mu\text{s}$; clearly, at least two packets are bound to expire. The simplification the infinite buffers bring is that *nothing* can be possibly done at the end host once a packet gets dropped, for "packet is dropped" is now equivalent to "packet is expired", and, as we discussed in Section II, for delay-sensitive applications an expired packet bears no reliability constraints. The one thing that prevents us from forgoing most of the end host logic complexity, including congestion control, is infinite buffers being extremely expensive to build. The next section explains how to emulate buffer management algorithms designed for infinite buffers using only finite buffers.

IV. Replacing an infinite buffer with a finite one

Due to bounded transmission rate, algorithms exploiting infinite buffers are still unable to transmit all packets in all cases, especially, if faced with large bursts of tight-deadline packets. The algorithm is free to choose the set of packets to transmit; which choice depends on the desired objective. We seek the "ideal" set of transmitted packets only among those that can be realized by some buffer management algorithm with infinite buffer. Packet slacks would naturally "sanitize" *unwanted* packets dying due to expired slacks.

Application	Delay req.	B (Theorem 2)	B (Theorem 3)
Web pages	few ms [12]	few 7.76 GiB	few 4.77 MiB
Interactive analytics	$\ll 1\,\text{ms}$ [13]	$\ll$ 7.76 GiB	$\ll$ 4.77 MiB
Social-networks	$200\,\mu\text{s}$ [14]	318.36 MiB	0.95 MiB
Distributed caches	$\approx 10\,\mu\text{s}$ [15]	$\approx$ 838.2 KiB	$\approx$ 48.83 KiB
Machine learning	$\approx 10\,\mu\text{s}$ [15]	$\approx$ 838.2 KiB	$\approx$ 48.83 KiB

Table I: Minimum required buffer size as in Theorem 2 and Theorem 3. The link rate is $40\,\text{Gbit/s}$ and the packet size is $1.5\,\text{KB}$.

In what follows, we are interested in taking an arbitrary $A[\infty]$ and emulating it with respect to a given objective by some other algorithm $A[B]$ with a limited buffer of size B. The goal is to make B as small as possible. Intuitively, B should be implicitly bounded by some function of the maximum slack value. We start with the WSwTPUT objective, which is a natural and yet generic choice for the bounded delay model that has already been studied in the infinite-buffers setting [6].

Theorem 2. *For a maximum slack S and a given online algorithm $A[\infty]$, there is an online algorithm $A[S(S+1)/2]$ equivalent to $A[\infty]$ w.r.t.* WSwTPUT.

Proof: Let $S_2 = 1+2+\ldots+S = S(S+1)/2$. Consider an arbitrary $A[\infty]$. The emulating algorithm $A[S_2]$ has a separate virtual queue of size s for packets with remaining slack being exactly s. The admission behavior of $A[S_2]$ is independent for every virtual queue always preferring higher values. On processing, $A[S_2]$ checks which packet $A[\infty]$ would have sent. Let us denote this packet as p. Then $A[S_2]$ transmits a packet p', s.t. $\text{sl}(p') = \text{sl}(p)$, $v(p') \geq v(p)$ and $v(p')$ is the minimum among all such p'. It remains to show that there would always be at least one such packet in $A[S_2]$'s buffer. For any given point in time, we denote by P'_s and P_s, the sets of packets with slack s in $A[S_2]$ and $A[\infty]$ buffers, respectively. We show by induction on the number of events that at any point in time if we take P_s and P'_s and sort them by decreasing value, then: 1) $|P'_s| \geq \min\{|P_s|, s\}$, and 2) for a pair of corresponding packets p_i and p'_i, $v(p_i) \leq v(p'_i)$ There are the following events: 1) change of a timeslot; 2) packet arrival; and 3) packet transmission. 3) preserves the property by $A[S_2]$'s definition. On 1) a packet can be removed from $A[S_2]$ buffer only if there are already $s+1$ packets with slack s. And on 2 it follows from the two sequences being already sorted and aligned by value, see [11, Lemma 1]. ■

We have applied Theorem 2 to different types of latency-sensitive applications. The results in Table I demonstrate that applications most sensitive to delay can be easily supported with slightly under $1\,\text{MiB}$ of buffer. The bound puts the buffer space for less constrained applications in the order of GiBs, i.e., more than the per-port space available on commodity devices. We must note that the bounds are quite lose due to overly general definition of $A[\infty]$, specifically, one does not include implementation constraints that would arise when running $A[\infty]$ at line rate. For example, $A[\infty]$ (and $A[B]$) in Theorem 2 are allowed to have infinite program memory, which makes little sense when we try to bound buffer space. To be more pragmatic, we should restrict the class of buffer management algorithms to those we could actually consider implementing. Turns out, not only does the bound on buffer sizes gets reduced as a result, but the bound becomes valid for any possible objective unlike Theorem 2. Recent abstractions representing buffer management policies [16], [17], [18] focus on algorithms processing packets according to a *fixed* order (e.g., sorting packets by arrival time, packet deadline, or by packet value). We will refer to such algorithms as *fixed-order*. Interestingly, any fixed-order algorithm with infinite buffer can be emulated by an algorithm whose buffer is *linear* in the value of a maximum slack. The result holds for *any* objective.

Theorem 3. *For a maximum slack S and an online fixed-order algorithm $A[\infty]$, there exists an online algorithm $A[S]$ transmitting exactly the same packet set.*

Proof: Consider an arbitrary online algorithm $A[\infty]$ that processes packets according to an order $\prec$. $A[S]$ also follows $\prec$ on processing, but on admission $A[S]$ keeps only those packets that $A[\infty]$ would have processed if there were *no further arrivals*. Clearly, there cannot be more than S such packets. Our claim is that $A[S]$ processes *exactly* the same set of packets as $A[\infty]$. For that it is sufficient to show that at any point in time, the set of packets that have already arrived and can be potentially processed by $A[\infty]$ (i.e., at least for some arrival) is a subset of packets that are currently in $A[S]$ buffer. It is straightforward to show by induction that any packet's position in $\prec$ order is monotonous w.r.t. arrival sequence, i.e., if we add a new arriving packet, then the position of any other packet will not decrease. The last statement implies that the transmission time cannot move to a later point if we remove some packet from the arrival sequence. Thus, if an already arrived packet p is ever processed by $A[\infty]$, then p is processed by $A[\infty]$ with no further arrivals. ■

For packet slacks, Theorem 3 implies that to emulate fixed-order algorithms with *infinite* buffers for an arbitrary objective, buffer sizes *linear* in the maximal slack value are sufficient. We also present in Table I buffer requirements assuming fixed-order algorithms, i.e., when applying Theorem 3.

Theorem 3 raises an even harder design question. Namely, do we really need to implement reliability at the transport layer if the maximum slack value S is appropriate, given that $A[S]$ is guaranteed to have have no losses due to congestion?

We do not go further by covering corrupted packets or proposing specific transport designs for low latency traffic. All we demonstrate is the potential impact of the model with packet slacks and WSwTPUT objective.

Using slacks in WSwTPUT, we explicitly communicate to the network that there is no need delivering packets once packets have expired. Hence, if we have just enough buffer space, we do not need anymore congestion-induced retransmissions for delay-sensitive applications. Implementing congestion control for delay-sensitive applications is a formidable task; considering sub-RTT delay requirements, it is not even clear how to get the control loop that short (previous works resorted to preallocations [19], which may give suboptimal utilization). The takeaway of this section hints a solution: *congestion control is not required in a delay-sensitive setting*; the problem is reduced to the management of an infinite buffer.

In the following section, we discuss what can be done if retransmissions are unavoidable. We quantify the impact of extra knowledge on the overall performance, we build a taxonomy of different algorithmic classes based on that knowledge, and we propose a universal transformation translating any policy optimizing throughput to a policy optimizing goodput while keeping performance guarantees.

V. Taxonomy of buffer management policies

To properly analyze buffer management policies in the presence of retransmissions, we must ensure that the objective we are using handles retransmissions properly. We model *retransmissions of a packet* p as p being a part of the arrival phase (see Section III-B) for several, not necessarily consecutive, time slots. The WSwTPUT objective (see Equation 1) is not well suited for retransmissions: when for some $i \neq j$ a switch sends the same packet, i.e., $p_i = p_j$, the contribution of that packet to WSwTPUT will be double the actual one.

To make WSwTPUT more appropriate to retransmissions, we prepend to every packet p its *identity* denoted by $\mathrm{id}(p)$, allowing the switch to correctly distinguish packets with the same values and slack. Then, we calculate the *goodput*:

$$\text{WSwGPUT} = \sum\nolimits_{p \in \mathcal{P}} v(p) \cdot [\mathrm{del}(p) \leq \mathrm{sl}(p)], \qquad (2)$$

where $\mathrm{del}(p) = \min\{\mathrm{del}_i(p_i) : \mathrm{id}(p_i) = \mathrm{id}(p)\}$.

Originally, due to end-to-end design principle, congestion controls were mostly based on implicit network feedback and decoupled from buffer management decisions. Recently more and more data-center transports represent monolith solutions combining both control programs at end hosts and buffer management policies inside switches to achieve better performance; e.g., pFabric [2] and DCTCP [1] (to list just a few). In what follows, we address the issue of composing local buffer management policies with congestion control and retaining performance guarantees by capturing the external information available at the switch, yet without restricting congestion control behavior beyond what is necessary.

We consider different knowledge levels that can be exploited by buffer management policies and study the impact these levels have on performance. We start with algorithms that *on arrival of a packet* p learn all p's properties, i.e., value, deadline, and identity—$\mathrm{id}(p)$, but *nothing else*. The class of all such algorithms we denote by $\mathcal{A}^{\mathrm{id}}$. Intuitively, $\mathcal{A}^{\mathrm{id}}$ has minimum knowledge required to optimize WSwGPUT. We contrast $\mathcal{A}^{\mathrm{id}}$ with algorithms studied in the previous works on buffer management, which completely ignore retransmissions [6], [20]. To capture the latter formally, we define a class $\mathcal{A}^{-}$ of algorithms that do not have access to $\mathrm{id}(p)$. In particular, an algorithm $A \in \mathcal{A}^{-}$ cannot distinguish whether two packets

p and q are instances of the same packet as a consequence of retransmissions or these two are just different packets. Naturally, we expect that $\mathcal{A}^{\mathrm{id}}$ has strictly better performance than $\mathcal{A}^{-}$. The following theorem states that packet identities are *absolutely* required for optimization of WSwGPUT.

Theorem 4. *For any $B \in \mathbb{N}$, any algorithm $A[B] \in \mathcal{A}^{-}$, and any $k \in \mathbb{N}$, there exists an algorithm $A'[B] \in \mathcal{A}^{\mathrm{id}}$ that k-dominates $A[B]$ w.r.t.* WSwGPUT.

Proof: Consider $A \in \mathcal{A}^{-}$ and $k \in \mathbb{N}$. Let A' be the algorithm that emulates A, but 1) A' never admits an already processed or duplicate packet; 2) if A' cannot admit the same packet as A, it admits an arbitrary packet; 3) if A' cannot process the same packet as A, it processes an arbitrary packet from the queue. Exactly k packets arrive at each of the time slots $t = 1, 2, \ldots, k^2$. For each packet arriving at $t = i$ we set $v(p) = 1$ and $\mathrm{sl}(p) = k^2 - i$. Packets arriving at $t = 1$ have $\mathrm{id}(p) = 1, 2, \ldots, k$. A processes at most k^2 packets in total, thus, at most for k time slots t A processes all packets arriving t. There must be, then, at least $k^2 - k$ time slots with at least one packet that has been never processed by A. For every packet p processed by A we set $\mathrm{id}(p) \in \{1, 2, \ldots, k\}$, so clearly $A(\sigma) \leq k$. For every packet p that has not been processed by A, we set $\mathrm{id}(p) = k + j$ for some unique $j \in \mathbb{N}$. It follows that $A'(\sigma) \geq k^2 - k$ and the domination follows. ■

On packet arrival, buffer management policies can exploit more information than just packet properties. If they knew the control program used at end hosts, arrival patterns, and how the packets were multiplexed, they could, in theory, know the precise time slots when retransmissions would appear. Formally, assume that the moment a packet p arrives, a switch learns a sequence $\tau(p) = \{t_1, \ldots, t_{k_p}\}$, where t_i is the time slot when p's ith retransmission would reach the switch. The class of algorithms possessing such knowledge, we denote by $\mathcal{A}^{\tau}$. It is no surprise that these can demonstrate even better performance than $\mathcal{A}^{\mathrm{id}}$ if only up to a constant factor of 2.

Theorem 5. *For any $B \in \mathbb{N}$ and any algorithm $A[B] \in \mathcal{A}^{\mathrm{id}}$, there exists an algorithm $A'[B] \in \mathcal{A}^{\tau}$ that 2-dominates $A[B]$ w.r.t.* WSwGPUT.

Proof: Consider an algorithm $A \in \mathcal{A}^{\mathrm{id}}$. Let $A' \in \mathcal{A}^{\tau}$ be an algorithm that fully emulates A, except 1) on admission between two otherwise equal packets A' prefers the one that will never be retransmitted by looking at τ; 2) if the queue is not empty and A does not process a packet, A' processes an arbitrary packet; 3) if A' cannot process the same packet as A, A' processes an arbitrary packet. Next, $2B$ packets arrive at $t = 1$, each packet p has $v(p) = 1$ and $\mathrm{sl}(p) = 2B$. A is bound to drop at least B packets. For every packet p dropped by A we set $\tau(p) = \{1\}$, for any other p we set $\tau(p) = \{1, B\}$. It remains to note that if A earns k then $k \leq B$ and A' earns $k + B$ and the claim follows. ■

Clearly, there exist other kinds of knowledge, some may even include information about yet unseen packets. For instance, we may consider a set of algorithms that at time slot t learn which packets will arrive at timeslots $t+1, t+2, \ldots, t+k$. We also may try to relax $\mathcal{A}^{\tau}$ by assuming that $\tau(p)$ is not revealed immediately on p's arrival but after a certain delay, or is revealed only partially. While fine-grained performance analysis of these and many other knowledge models is important to understand the performance of cooperation between *congestion control and buffer management*, in what follows we restrict ourselves to the extreme case of absolute knowledge that already provides some interesting insights.

Analysis of buffer management behavior with respect to adversarial traffic was traditionally performed using competitive analysis [21]. Performance results in the competitive analysis are usually stated positively: an algorithm `ALG` is called α-*competitive* iff for any $\epsilon > 0$, an optimal offline (*clairvoyant*) algorithm `OPT` does *not* $(\alpha + \epsilon)$-dominate `ALG`. The question with respect to offline algorithms is whether the knowledge of $\tau(p)$ is sufficient for optimality or there exists a performance gap between `OPT` and $\mathcal{A}^{\tau}$. The answer turns out to be the latter.

Theorem 6. *For any $B \in \mathbb{N}$, any algorithm $A[B] \in \mathcal{A}^{\tau}$ is at least $\frac{6}{5}$-competitive (or `OPT`$[B]$ $\frac{6}{5}$-dominates $A[B]$) w.r.t.* WSwGPUT.

Proof: On the first time slot, $2B$ packets with $\mathrm{sl}(p) = \infty$ arrive, among which B packets will be retransmitted on time slot $t_0 = B$ and B packets on time slot $t_1 = 2B$. `ALG` drops at least B packets at $t_0 = 1$, so there exists $i \in \{0, 1\}$ s.t. at least $\frac{B}{2}$ of the dropped packets arrive at t_i, and let us assume with loss of generality that $i = 0$. Then, B new packets with $\mathrm{sl}(p) = \infty$ arrive at t_0. Since `ALG` can accept at most B packets among at least $\frac{3B}{2}$ not-yet-admitted packets arriving at t_0, `ALG` processes $\frac{B}{2}$ packets less than maximum possible, i.e., $3B$. `OPT`, on the other hand, drops all the packets arriving at t_{1-i} and is able to process all packets, so the claim follows. ■

In summary, we have classified buffer management algorithms based on the knowledge they use, and, in addition, we have rigorously quantified the performance gap between different classes in Theorems 4, 5, and 6. In the next section we bound the gap between buffer management policies that know nothing about congestion control implementations at end hosts and policies enjoying a full global knowledge of congestion control algorithms and timing of network events.

VI. Transforming Throughput to Goodput

In this section we present a guarantee-preserving universal transformation constructing online algorithms optimizing WSwGPUT from online algorithms optimizing WSwTPUT.

For an arbitrary online algorithm $A[B]$, we introduce the transformation `TG`$(A[B])$, which essentially ensures that $A[B]$ never sees the same packet twice by maintaining the set $\mathcal{I}_{\mathrm{done}}$ of processed ids, *empty* initially:

- **On arrival:** for an arriving packet p if either $\mathrm{id}(p) \in \mathcal{I}_{\mathrm{done}}$ or $A[B]$'s buffer contains p' s.t. $\mathrm{id}(p') = \mathrm{id}(p)$, drop p; otherwise, follow the $A[B]$'s decision.
- **On processing:** process the same packet p as $A[B]$, and let $\mathcal{I}_{\mathrm{done}} \leftarrow \mathcal{I}_{\mathrm{done}} \cup \{\mathrm{id}(p)\}$

Note, `TG`$(A[B])$ may admit the same packet multiple times, but every packet gets processed only once. The next theorem

	Constraints and existing results				Universal transformation results		
Algorithm	$\mathrm{sl}(p)$	B	$v(p)$	WSwTPut	B	WSwTPut	WSwGPut
`ValueGreedy`	$\mathbb{N}$	$\{\infty\}$	$\mathbb{N}$	$2 \times$ opt., [6]	$\{S, \ldots\}$	$2 \times$ opt., Th. 3	$2 \times$ opt., Th. 3 + Th. 8
	$\{\infty\}$	$\mathbb{N}$	$\mathbb{N}$	opt.	$\mathbb{N}$ $\{\infty\}$		$2 \times$ opt., Th. 7 opt., Th. 8
	$\mathbb{N}$	$\{\infty\}$	$\{1, \alpha\}$	$1 + \frac{1}{\alpha} \times$ opt., [6]	$\{S, \ldots\}$	$1 + \frac{1}{\alpha} \times$ opt., Th. 3	$1 + \frac{1}{\alpha} \times$ opt. Th. 3 + Th. 8
`EDF`	$\mathbb{N}$	$\mathbb{N}$	$\{1\}$	opt.	$\mathbb{N}$ $\{S, \ldots\}$		$2 \times$ opt.,Th. 7 opt., Th. 8
$1/\phi -$ `EDF`	$\{1, 2\}$	$\{\infty\}$	$\mathbb{N}$	$\phi \times$ opt., [6]	$\{2, \ldots\}$	$\phi \times$ opt., Th. 3	$\phi \times$ opt., Th. 3 + Th. 8
`GRQ`	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$	$2 \times$ opt., [22]	$\mathbb{N}$ $\{S, \ldots\}$		$3 \times$ opt., Th. 7 $2 \times$ opt., Th. 8

Table II: The summary of analytic results for automatically generated policies by the proposed transformations.

demonstrates that the above transformation indeed preserves performance guarantees, albeit with a small additive loss. The main idea is to compare the value of the objective function of the two optimal algorithms: one for the original sequence of packets, and one—for the sequence $A[B]$ would see after transformation. We remind, $\mathcal{A}^{-}$ is the set of online algorithms oblivious to packet identities, and $\mathcal{A}^{\mathrm{id}}$—those that are not.

Theorem 7. *if $A[B] \in \mathcal{A}^{-}$ is α-competitive w.r.t.* WSwTPut *then* $\mathtt{TG}(A[B]) \in \mathcal{A}^{\mathrm{id}}$ *is $(\alpha+1)$-competitive w.r.t.* WSwGPut.

Proof: Consider an arbitrary arrival sequence σ with retransmissions and denote by σ' the arrival sequence as seen by $A[B]$ during $\mathtt{TG}(A[B])$ execution. The following holds:

$$\alpha \cdot \mathtt{TG}(A[B])(\sigma) = \alpha \cdot A[B](\sigma') \geq \mathtt{OPT}[B](\sigma'),$$

the last inequality follows from α-competitiveness of $A[B]$. Note, σ' lacks retransmissions for only those packets that $A[B]$ has processed successfully. Thus, if we consider sequence σ'' that adds to σ' precisely those missing retransmissions used by $\mathtt{OPT}[B](\sigma)$, σ'' contains at most one extra copy for every packet processed by $A[B](\sigma')$. The claim follows from:

$$\begin{aligned}(\alpha+1)A[B](\sigma) \geq\ & \mathtt{OPT}[B](\sigma') + A[B](\sigma') \\ \geq\ & \mathtt{OPT}[B](\sigma'') \geq \mathtt{OPT}[B](\sigma).\end{aligned} \quad \blacksquare$$

There are two nice features of Theorem 7: buffer size preservation and independence from packet slack values. The result can be further improved if we assume an upper bound S on packet slacks and large enough buffer, namely, $B \geq S$. The key property we use is that an optimal offline solution that can reorder packets does not need retransmissions anymore, i.e., we can safely assume that $\mathtt{OPT}[B]$ accepts p at the moment p first arrives. To emphasize the necessity for packet reordering, we use $\mathcal{A}_{\prec}$ to denote the set of online algorithms supporting it and $\mathtt{OPT}_{\prec}[B]$ to denote a respective optimal offline algorithm.

Lemma 1. *For any input σ, if $B \geq S$ then there exists an $\mathtt{OPT}_{\prec}[B]$'s solution for σ accepting packets on the first arrival.*

Proof: Consider an arbitrary solution $\mathtt{OPT}'[B]$ and a packet p that first arrives at t and gets processed by $\mathtt{OPT}'[B]$ at time step $\tilde{t}$. $\mathtt{OPT}[B]$ will accept p at t and at any time step $t' \in [t, \tilde{t}]$ p will occupy position $(\tilde{t} - t')$ in $\mathtt{OPT}[B]$ buffer . Since $0 \leq \tilde{t} - t' < \mathrm{sl}(p) \leq S$, at most S packets reside in $\mathtt{OPT}$ buffer simultaneously, and there are no overflows. ■

As a result, we can transfer performance guarantees much easier between the models with and without retransmissions (the backward direction is straightforward—the model with retransmissions represents a more general case).

Theorem 8. *For a set of algorithms $\mathcal{A}_{\prec}$, a maximum slack S, and a buffer size $B \geq S$, if $A[B] \in \mathcal{A}_{\prec}^{-}$ is α-competitive w.r.t.* WSwTPut *then* $\mathtt{TG}(A[B]) \in \mathcal{A}_{\prec}^{\mathrm{id}}$ *is α-competitive w.r.t.* WSwGPut

Proof: Consider an arbitrary arrival sequence σ with retransmissions, and let σ' be the sequence as seen by $A[B]$ during $A[B]^{\mathrm{rt}-}$ execution. We have the following:

$$\alpha \cdot A[B]^{\mathrm{rt}-}(\sigma) = \alpha \cdot A[B](\sigma') \geq \mathtt{OPT}[B](\sigma') \geq \mathtt{OPT}[B](\sigma),$$

where the last inequality follows from Lemma 1 and the fact that σ' preserves first arrivals from σ. ■

Once S gets larger than B, we lose Lemma 1 and waiting for a given retransmission becomes essential to achieving optimal (offline) performance which is accounted for in Theorem 7.

A. Applying transformations

The first transformation that we propose emulates online algorithms with infinite buffers using algorithms with finite buffers. Table 3 in [23] surveys more than twenty relevant algorithms for bounded-delay model in various settings, where most of them assume infinite buffer size and, hence, subjected to Theorem 2 and Theorem 3. For the general case [22] proposes a 2-competitive GreedyQueue (GRQ) algorithm for a finite buffer size. At least for $B \geq S$, we can get this result automatically by applying Theorem 3 to a greedy algorithm with an infinite buffer transmitting the largest value first [6]. The second transformation allows to automatically construct online policies optimizing weighted goodput WSwGPut from policies optimizing weighted throughput WSwTPut, preserving performance guarantees. To our best knowledge, there are no theoretical results providing competitive results for WSwGPut. So we have little to compare with. The shown results in Table II are given only to demonstrate that the proposed transformations are universal tools allowing to experiment with different algorithms during design of congestion controls in the bounded-delay model.

VII. Evaluations

In the evaluation study we aim to understand the effect of WSwTput optimization on network behavior. No less relevant to the design principles of congestion control are the compositional properties of end host programs and buffer management policies. To address these questions, we present a series of experiments performed in a simulated environment.

A. Simulation setup

We start by thoroughly describing different parameters of the evaluation setup. The code for our simulations is based on *NS2* [24] and is made available on *GitHub* [25].

a) Topology: In our experiments we use the same leaf-spine topology as pFabric and PIAS [26], [2]. The network includes 4 spine switches, 9 ToR leaf switches, and links form a full bipartite graph between spines and leaves. There are 16 servers connected to each of the 9 ToR switches, i.e., 144 servers in total. Server-to-ToR link rates are $10\,\mathrm{Gbit/s}$, ToR-to-spine—$40\,\mathrm{Gbit/s}$, buffer size B is 140 packets as in [26]. Every pair of servers exchanges messages independently.

b) Algorithms: The network is governed by a composition of: control program at end hosts, and buffer management policy. To show the interaction between the two, we varied them using options described in Table III. A complete solution is a "buffer, control" pair, e.g., "pFabric, D^{2}TCP" means D^{2}TCP at the end hosts and pFabric at the switches.

Control programs at endhosts	
pFabric	*Data-Center TCP* [1] with pFabric's [2] modifications.
D^{2}TCP	Deadline-aware Data-Center TCP [7]
Buffer management policies	
ECN$^{\mathtt{RED}}$	Tail drop, RED-like ECN marking [1], [7].
pFabric	Shortest-remaining processing time [2].
pFabric$^{\mathtt{edf}}$	Earliest-deadline first [2].
GRQ	The algorithm, presented in [22].

Table III: Endhost and switch components used for evaluations.

c) Performance metric: The ultimate purpose of the bounded-delay model is to improve network performance as perceived by applications. Hence, to properly evaluate such performance, we focused on an application-centric metric *application throughput* [8], [7], [2]. We assume that every application flow comes with two characteristics: a flow size $\mathrm{size}(f)$, and a flow's slack $\mathrm{sl}(f)$. If $\mathrm{del}(f)$ is the completion time of a flow f then the application throughput for a set of flow $\mathcal{F}$ is defined as the number of flows that finish within their slacks: $\textsc{AppTput}(\mathcal{F}) = |\{f \in \mathcal{F} : \mathrm{del}(f) < \mathrm{sl}(f)\}|$.

d) Flow properties: For flow sizes we chose two CDFs derived from real-world data and often used in the literature (e.g., pFabric [2] and PIAS [26]): one for web search applications [1], and the other—for data mining [7]. In addition, we tested one distribution assigning exactly 20 packets and one uniform (as in [8]). Similar, again, to [2] we generated flow slacks $\mathrm{sl}(f)$ from an exponential distribution and assumed to be delay-insensitive, i.e., $\mathrm{sl}(f) = \infty$, flows larger than $200\,\mathrm{KiB}$. For the mean slack we chose $100\,\mu s$ and $1000\,\mu s$ used in previous works. We removed the constraint that a slack has to be at least 125% of minimum flow completion time (used in [2]), thus checking if optimization of AppTput handles extremely tight or impossible deadlines automatically. The distributions are summarised in Table IV.

Flow size distributions, $\mathrm{size}(f)$, pkts			
web_search	data_mining	const	uniform
CDF [1]	CDF [27]	20	$\mathrm{U}(90, 100)$
Flow slack distributions, $\mathrm{sl}(f)$, ms			
normal	tight		
$\mathrm{Exp}(1)$ [2]	$\mathrm{Exp}(0.1)$ [2]		

Table IV: Distributions for flow slacks and sizes, $\mathrm{U}(a, b)$ denotes uniform distribution, $\mathrm{Exp}(\lambda)$—exponential, CDF—cumulative distribution function.

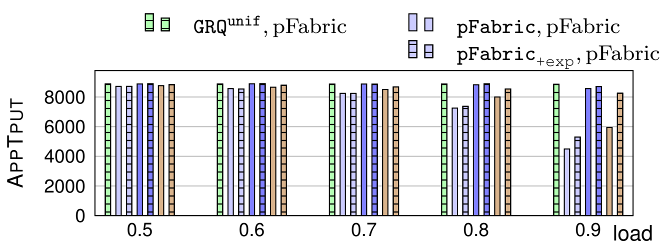
e) Arrival times: Following pFabric [2] and PIAS [26], message arrivals are modelled by Poisson distribution, whose mean depends on the load. Specifically, for a load $x \in (0, 1)$, we derive the mean message arrival rate $\lambda(x)$ as in [26].

B. Throwing away garbage by in-queue expiration

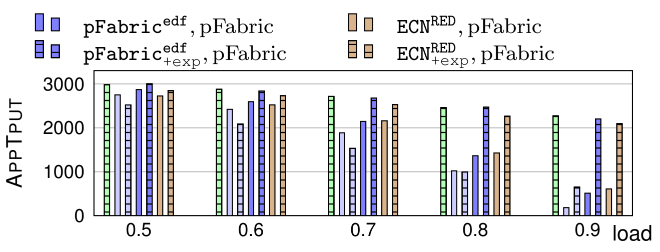
Only GRQ buffer-management policy "natively" supports packet slacks. While pFabric$^{\mathtt{edf}}$'s behavior does depend on deadline imminence, this dependency is rather indirect and ultimately relies on end hosts setting deadline-based priorities. In Figure 1, we show the benefit of taking direct advantage of slacks by augmenting every buffer-management policy with a simple deadline-awareness: *automatically drop expired packets.* The augmented algorithms are denoted by ALG$_{+\mathtt{exp}}$ for respective original ALG. The main observation we make here is that packet expiration can substantially improve performance of otherwise deadline-unaware algorithms on similar-sized messages, e.g., ECN$^{\mathtt{RED}}$ (see Figure 1a and 1b) and pFabric (see Figure 1d at 0.9 load). Interestingly, the performance of pFabric$^{\mathtt{edf}}$ also gets improved (see Figure 1b).

C. Composing switch and end host behavior

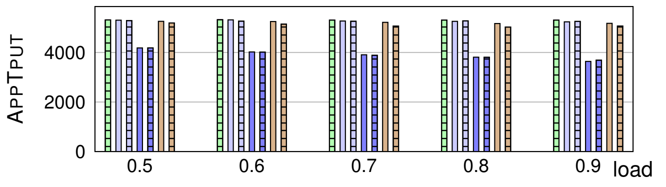
In Figure 2, we study interaction between end-host control and buffer management policies by evaluating eight "buffer, control" pairs. The single best strategy is hard to name, for the behavior varies with different inputs. As an example, pFabric shows very good performance for *data mining* and *web search* workloads (see Figure 2a and 2c) where short messages dominate, and preferring those improves AppTput. On the other hand, when pFabric is facing a workload with little message-size variance, it tends to lose substantially (see Figure 2d and 2b with pFabric). Regarding the interaction between control programs at end hosts and buffer management, we note that deadline-aware D^{2}TCP is not only able to noticeably improve performance of deadline-unaware pFabric and ECN$^{\mathtt{RED}}$ (see Figure 2d and 2b), but



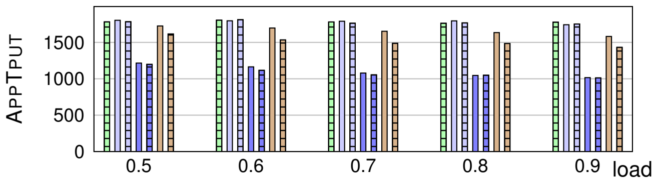
(a) size is `const`, slacks are `normal`



(b) size is `const`, slacks are `tight`

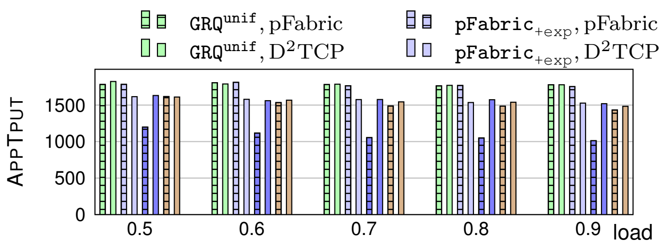


(c) size is `web-search`, slacks are `normal`

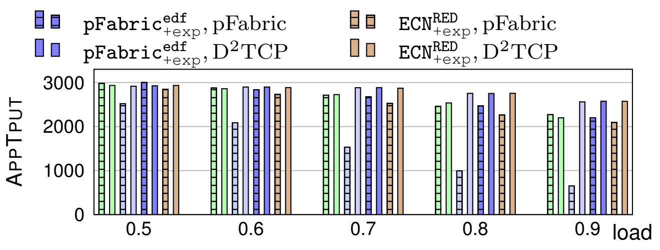


(d) size is `web-search`, slacks are `tight`

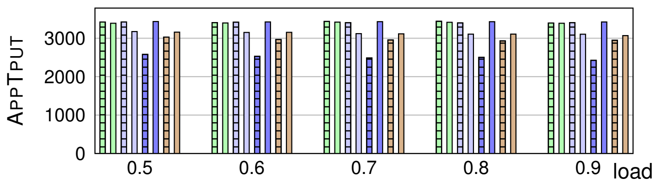
Figure 1: Studying the effect of in-buffer packet expiration on AppTput for different buffer-management strategies.



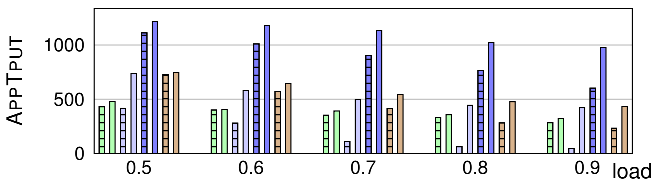
(a) size is `web-search`, slacks are `tight`



(b) size is `const`, slacks are `tight`



(c) size is `data-mining`, slacks are `tight`



(d) size is `uniform`, slacks are `tight`

Figure 2: Studying the compositional properties of endhost control programs and buffer management policies.

also—of `pFabric`$^{\texttt{edf}}$ (Figure 2a–d). `GRQ` is the least affected by end host logic, which we attribute to the main result of universal transformation (see Section VI): `GRQ` is provably close to having full knowledge of end host behavior.

D. When congestion control comes first

In Section IV we raised a question of whether congestion control should be implemented on top of guaranteed reliability or the other way around. So far we were following the former, traditional, path. In the last set of experiments we gauge the cost (or the benefit) the alternative approach can bring.

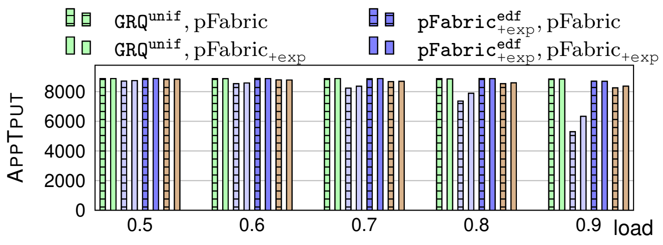
We have modified simulator logic responsible for end-host behavior so that the connection is automatically reset once the message deadline passes. The results are presented in Figure 3, where for an original control program control, we denote the modified version as control$_{+\texttt{exp}}$. Our findings are that the effect on AppTput depends on how sensitive the buffer management policy is to packet slacks. While the difference for `pFabric`$^{\texttt{edf}}$ and `GRQ` is marginal across all runs, for pFabric (see Figure 3a–b) and `ECN`$^{\texttt{RED}}$ (see Figure 3b) there is a significant improvement when the control logic is made deadline-aware.

VIII. Related work

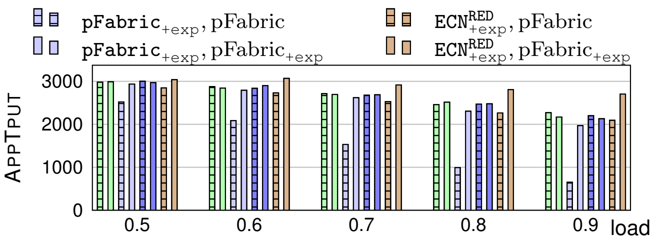
Application-supplied deadlines: Explicit application-supplied slacks have been used recently for end hosts control logic in [28], [8], and [2], which optimized *application throughput*. As we explain in Section II-B, end host-based control can be too slow for ultra-low latency.

Buffer management policies: A good survey of advanced processing orders with admission controls can be found in [23]. Recently, several abstractions expressing buffer management policies have been proposed [17], [29]. All of them support fixed processing orders as in Theorem 3. Another relevant work [30] is investigating when there exist practical algorithms satisfying all deadlines, assuming all are satisfiable.

Design principles: One of the interesting outcomes of [31] studying coexistence of several congestion control properties is that simultaneous optimization of both delay requirements and throughput is not really possible. Similarly to



(a) size is `const`, slacks are `normal`



(b) size is `const`, slacks are `tight`

Figure 3: Studying the consequences of reliability being implemented on top of congestion control

our work, [32] aims to understand the impact of various factors on congestion control performance. [33] studies model-based vs. model-free approaches in congestion controls.

IX. Conclusion

In this paper we show that the bounded-delay model with weighted throughput (goodput) is a good potential basis for design of congestion control supporting low latency. Packet slacks are a natural mechanism to sanitize networks from the "garbage" traffic with expired slacks. We analyze properties of the model and propose several types of transformations allowing tuning performance for specific low-latency requirements. We discuss impact of additional policy properties on required buffer sizing. We show how a service model should be adjusted and define potential ways to implement reliable transports without retransmissions for ultra low-latency traffic. We hope that the proposed study can help other researchers to experiment with the bounded-delay model in various settings.

References

[1] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM*, 2010, pp. 63–74.

[2] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM*, 2013, pp. 435–446.

[3] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *NSDI*, 2018, pp. 329–342.

[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 50:20–50:53, 2016.

[5] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *SIGCOMM*, 2018, pp. 221–235.

[6] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko, "Buffer overflow management in qos switches," *SIAM J. Comput.*, vol. 33, no. 3, pp. 563–583, 2004.

[7] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter tcp (D2TCP)," in *SIGCOMM*, 2012, pp. 115–126.

[8] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM*, 2011, pp. 50–61.

[9] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, p. 74–80, 2013.

[10] J. C. Mogul and R. R. Kompella, "Inferring the network latency requirements of cloud tenants," in *HotOS)*, 2015.

[11] P. Chuprikov, S. I. Nikolenko, A. Davydow, and K. Kogan, "Priority queueing for packets with two characteristics," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 342–355, 2018.

[12] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," *CCR*, vol. 45, no. 4, p. 435–448, 2015.

[13] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," in *SIGCOMM*, 2015, p. 537–550.

[14] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *NSDI*, 2013, pp. 385–398.

[15] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *SIGCOMM*, 2015, p. 523–536.

[16] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *SIGCOMM*, 2016, pp. 44–57.

[17] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. I. Nikolenko, A. Sirotkin, and P. Eugster, "A programmable buffer management platform," in *ICNP*.

[18] A. Saeed, Y. Zhao, N. Dukkipati, E. W. Zegura, M. H. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *NSDI*, 2019, pp. 17–32.

[19] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *SIGCOMM*, 2017, p. 239–252.

[20] A. Fiat, Y. Mansour, and U. Nadav, "Competitive queue management for latency sensitive packets," in *SODA*, 2008, pp. 228–237.

[21] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. Cambridge University Press, 1998.

[22] S. P.Y.Fung, "Bounded delay packet scheduling in a bounded buffer," *Operations Research Letters*, vol. 38, no. 3, pp. 396–398, Sep. 2010.

[23] M. H. Goldwasser, "A survey of buffer management policies for packet switches," *SIGACT News*, vol. 41, no. 1, pp. 100–128, 2010.

[24] "The network simulator — ns2," http://www.isi.edu/nsnam/ns/.

[25] "Code for simulations," https://github.com/pschuprikov/network-delay-sensitive.

[26] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *NSDI*, 2015, pp. 455–468.

[27] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *SIGCOMM*, 2009.

[28] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM*, 2012, pp. 127–138.

[29] A. Saeed, Y. Zhao, N. Dukkipati, E. W. Zegura, M. H. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *NSDI*, 2019, pp. 17–32.

[30] R. Mittal, et al. Rachit Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *NSDI*, 2016, pp. 501–521.

[31] D. Zarchy, R. Mittal, M. Schapira, and S. Shenker, "Axiomatizing congestion control," *POMACS*, vol. 3, no. 2, pp. 33:1–33:33, 2019.

[32] A. Mushtaq, R. Mittal, J. McCauley, M. Alizadeh, S. Ratnasamy, and S. Shenker, "Datacenter congestion control: Identifying what is essential and making it practical," *CCR*, vol. 49, no. 3, pp. 32–38, Jul. 2019.

[33] M. Schapira, "Network-model-based vs. network-model-free approaches to internet congestion control," in *HPSR*, 2018, pp. 1–8.