RTCP - Reduce Delay Variability with an End-to-end Approach

Longbo Huang
IIIS, Tsinghua University
Beijing, China
longbohuang@tsinghua.edu.cn

Yuxing Li
IIIS, Tsinghua University
Beijing, China
yx-li17@mails.tsinghua.edu.cn

Jean Walrand
EECS, University of California
Berkeley, CA, USA
wlr@eecs.berkeley.edu

Abstract—**Many important applications in modern network systems are sensitive to delay and delay variability. To ensure good performance on both metrics, in this paper, we propose a novel transport protocol called Regulated TCP (RTCP), whose design is inspired by the use of auxiliary variables in optimization theory. RTCP is a light-weight protocol and it uses counters in end devices and end-to-end delay measurements for rate adaptation (via window adjustment). We implement RTCP in both `ns-3` and a network testbed under two workload types, i.e., web search and data mining. Our testbed experiments show that compared to existing congestion control schemes such as BBR [1] and PCC [2], RTCP guarantees a better delay performance for urgent flows, while keeping a competitive goodput. Our `ns-3` simulation results also demonstrate that RTCP performs well in a wide range of scenarios. In particular, RTCP achieves a 9.78% throughput increase over BBR, and a 64.22% delay reduction of DCTCP [3] in a random sending scenario. It also reduces the 99th packet delay to less than 56% of other end-to-end TCP versions, including LEDBAT [4], TIMELY [5] and Vegas [6], in the large-scale incast scenario.**

Index Terms—**Transmission Control Protocol, Delay Variability, End-to-end**

I. Introduction

Many important network applications, such as web search, social communication and e-sports, are sensitive to delay and delay variability [7] [3]. For these applications, their performance and user experience can degrade significantly with a small increment in delay or jitter. As a result, various methods have been proposed to tackle these two problems. For instance, D^3 [8] provides a greedy policy to handle deadline constrained flows. The criticality-based approach pFabric [9] achieves a near-optimal performance. Karuna [10] uses flow information to classify flows into different types and operates different schemes. DCTCP [3] and VCP [11] adopt an explicit congestion notification (ECN) [12] based scheme. D2TCP [13] and MCP [14] further include prior knowledge to make DCTCP information-aware. PIAS [15] simulates shortest job first (SJF) without knowing flow size, with the help of strict priority queues in switches. Other schemes based on active queue management (AQM) such as random early detection (RED) [16] and proportional integral (PI) [17] have already been shown not to perform well without statistical multiplexing [3]. Although the solutions above provide good performance in different scenarios, they often require complex intermediate network node configurations for packet transmission. As a result, they may not be directly applicable to commodity products, especially when the middle network devices remain in black box for end-to-end users.

The past few years have also witnessed an optimization-oriented approach for better transport protocol design, which reverse-engineers transport layer protocols based on optimization theory, e.g., [18], [19], and [20]. The key of this method is to convert a *static* objective, often a function of flow utilities, into a *dynamic* control strategy that reacts in real time to network observations. This idea is similar to how a gradient algorithm works. Specifically, one starts with the goal of minimizing a function and ends up with an adjustment scheme. The key step is to formulate the constraints of the optimization problem properly, so that the "gradients" can be observed easily and gradient updates can be translated into rate adaption. For a transmission control protocol, one formulation leads to an end-to-end TCP-like scheme, and the gradient is then the total losses in routers, e.g., Random Early Detection or the accumulative markers, e.g., Explicit Congestion Notification. A different formulation may lead to a delay-based scheme, e.g., TCP Vegas [6], while another formulation is finer and results in a back-pressure algorithm that enables dynamic routing but requires per-flow queuing as feedback between adjacent switches [20]. Different formulations of the constraints result in protocols with varying flexibility and complexity.

In this paper, we adopt the optimization-oriented approach and propose a different formulation that results in a smooth version of TCP that we call Regulated TCP (RTCP). This protocol is smoother than the standard TCP and results in a small delay and delay variability. The protocol achieves a maximum utility of flows, where different classes of flows have different utility functions to capture their different trade-offs between delay and throughput. To achieve optimal utility, we maximize a weighted sum of concave functions of the throughput of flows. Flows that are more urgent get higher weights and therefore an implicit priority. To minimize delay variability and interference between flows, we formulate the optimization problem in a way that the constraints appear at the edges instead of inside the network. Thus, the constraint satisfaction levels appear as shadow prices, and are explicitly visible as counters at servers and measured end-to-end delay by time stamps, making the implementation easy and viable.

ISBN 978-3-903176-28-7 2020 IFIP

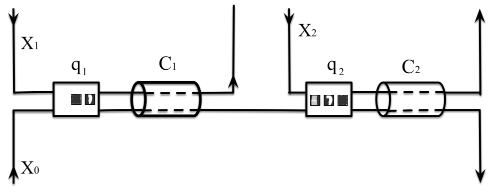


Fig. 1. Three connections share two links. Here, for $i = 0, 1, 2$, x_i indicates the average transmission rate of connection i, in bits per second. $C_j, j = 1, 2$ denote the transmission rate of link j. Each link is equipped with a first-come-first-served (FCFS) queue and q_j is the backlog in the buffer of link j.

We implement a RTCP prototype and test its performance on both a testbed and `ns-3` simulations based on the current TCP stack. We implement RTCP as a kernel module in Linux, while keeping the switches' default settings unchanged. We find that with proper settings, RTCP fits different scenarios well, including random sending, urgent flow preemption test, and over-subscription scenario. It improves the delay performance while keeping the throughput competitive. For example, RTCP achieves a 9.78% throughput increase over BBR, and a 64.22% delay reduction over DCTCP in a random sending scenario; it uses 1.83% throughput loss to achieve an 85.08% delay reduction compared to TIMELY [5] in over-subscription scenarios. We also investigate the influence of parameters on RTCP.

Below, we first provide a simple example to illustrate the design principle of RTCP. Then, we present the general framework and describe RTCP for the network.

II. A SIMPLE EXAMPLE

We provide a simple example to explain the idea of Regulated TCP (RTCP). Consider a simple network as shown in Fig. 1, where three connections share two communication links with transmission rates C_1 and C_2.

A. RTCP Design

We now introduce the formulation that leads to the RTCP protocol. In [18] and [19], Kelly et al. introduced the mathematical framework that leads to TCP. Specifically, consider

$$\begin{aligned} \max: \; & g(\boldsymbol{x}) := \sum_{i=0}^{2} U_i(x_i) \\ \text{s.t.} \quad & x_0 + x_1 \le C_1 \\ & x_0 + x_2 \le C_2. \end{aligned}$$

In this formulation, the functions $U_i(x_i)$ represent flow utilities and are often chosen to be concave increasing functions, such as $\omega_i \log(x_i)$ for a weighted proportionally fair allocation, or $\omega_i x_i^{1-\alpha}/(1-\alpha)$, for an α-fair allocation [21]. The concavity captures the diminishing value of additional rate when the rate increases.

We build our work upon this formulation. However, instead of a dual-based approach as in [18] and [19], we carry out our design by introducing a Lyapunov function, which is first used to stabilize the network in [22], to solve this problem. Specifically, the first step is to rewrite the problem in the following equivalent form:

$$\begin{aligned} \max: \; & g(\gamma) := \sum_{i=0}^{2} U_i(\gamma_i) \\ \text{s.t.} \; & \gamma_i \le x_i, i = 0, 1, 2 \\ & x_0 + x_1 \le C_1 \\ & x_0 + x_2 \le C_2. \end{aligned}$$

The second step, then, is to introduce the Lyapunov function $V(\boldsymbol{Q}, \boldsymbol{H}) := (1/2)\sum_j (Q_j^2 + H_j^2)$, where Q_j is the backlog in the queues at link j and H_j is a virtual backlog indicating how much the actual rate lacks behind the target rate. The two queues evolve according to (assuming a continuous time setting):

$$\dot{Q}_j = x_0 + x_j - C_j, \quad \dot{H}_j = \gamma_i - x_i. \tag{1}$$

The shadow prices H_i are obtained by a gradient algorithm, which shows that they are proportional to the counter values, and the counter for H_i increases at rate γ_i and decreases at rate x_i. Intuitively, if $\gamma_i > x_i$ for some time, then the price of γ_i increases to force it to decrease when solving the maximization problem. The introduction of γ_i in addition to x_i allows for a slack between the "desired rate" γ_i and the actual rate x_i of flow in the network. This slack corresponds to an intermediate queue between the transport layer and the NIC buffer. In practice, this queue reduces the interference of flows of different classes.

To achieve a suitable tradeoff between the small queue (thus a small $V(\boldsymbol{Q}, \boldsymbol{H})$) and high utility, one considers the following combination of the utility of the rate x_i (equivalently represented by γ_i) and the drift of the Lyapunov function:

$$\max: \; g(\boldsymbol{\gamma}) - \alpha \frac{d}{dt} V(\boldsymbol{Q}, \boldsymbol{H}) \tag{2}$$

By making explicit the drift of the Lyapunov function (based on (1)), we see that the problem becomes solving the following problem at every time:

$$\begin{aligned} \max: \; & \sum_{i=0}^{2} U_i(\gamma_i(t)) - \alpha \sum_{j=1}^{2} Q_j(t)[x_0(t) + x_j(t) - C_j] \\ & - \alpha \sum_{i=0}^{2} H_i(t)[\gamma_i(t) - x_i(t)]. \end{aligned}$$

In this expression, $\boldsymbol{x}(t) = (x_0(t), x_1(t), x_2(t))$ is the vector of rates of the connections at time t, and $\boldsymbol{q}(t)$ is the vector of backlogs at the two queues at time t. The parameter α determines the tradeoff between congestion and utility: a large α favors congestion reduction whereas a small α gives a priority to a large utility. Doing so gives rise to the following rate adaptation rules:

$$\gamma_i \in \arg\max U_i(\gamma_i) - \alpha H_i(t)\gamma_i \tag{3}$$

and

$$x_i \in \arg\max x_i(H_i - \alpha Q_{s_i} - \alpha Q_{d_i}). \tag{4}$$

To estimate the performance of this control scheme, let $\boldsymbol{x}^*$ be the maximizer of (2), we have

$$\begin{aligned} & g(\boldsymbol{\gamma}(t)) - \alpha \frac{d}{dt} V(\boldsymbol{Q}(t), \boldsymbol{H}(t)) \\ & \geq g(\boldsymbol{\gamma}^*) - \alpha \sum_{j=1}^{2} Q_j(t)[x_0^* + x_j^* - C_j] \\ & \qquad - \alpha \sum_{i=0}^{2} H_i(t)[\gamma_i^* - x_i^*] \\ & \geq g(\boldsymbol{x}^*). \end{aligned}$$

Here the first inequality comes from the fact that $(\boldsymbol{x}(t), \boldsymbol{\gamma}(t))$ maximizes (3), and the second comes from the fact that $x_0^* + x_j^* \leq C_j$. Integrating this inequality over $[0, T]$ and dividing by T, we conclude that

$$\frac{1}{T}\int_0^T g(\boldsymbol{x}(t))dt - \alpha \frac{1}{T}[V(\boldsymbol{Q}(T), \boldsymbol{H}(T)) - V(\boldsymbol{Q}(0), \boldsymbol{H}(0))] \\ \geq \; g(\boldsymbol{x}^*)$$

Letting $T \to \infty$, we find that

$$\liminf_{T\to\infty} \frac{1}{T}\int_0^T g(\boldsymbol{x}(t))dt \geq g(\boldsymbol{x}^*),$$

which implies that this control achieves the maximum performance asymptotically.

III. General Formulation

We now turn to the general formulation. Consider a network that connects N servers. There are flows x_i going from a server s_i to another server d_i. The goal again is to maximize the sum of flow utilities subject to capacity constrains. The problem is as follows.

$$\begin{aligned} \max : \; & g(\boldsymbol{x}) \triangleq \sum_i U_i(x_i) \\ \text{s.t.} \; & A\boldsymbol{x} \leq \boldsymbol{C}. \end{aligned}$$

In this formulation, A is the routing matrix, where $A_{ij} = 1$ if flow i traverses link j, and $A_{ij} = 0$ otherwise. $(A\boldsymbol{x})_j$ denotes the sum of the flow rates that go through a given link j and C_j is the capacity of that link.

As in the simple example, we go through three steps to convert the problem to the following "utility plus backlog drift" form:

$$\begin{aligned} \max : \; & \sum_i U_i(\gamma_i) - \alpha \sum_j Q_j(t)[(A\boldsymbol{x}_j - C_j)] \qquad (5) \\ & - \sum_i H_i(\gamma_i - x_i). \end{aligned}$$

The RTCP protocol will be designed based on this formulation.

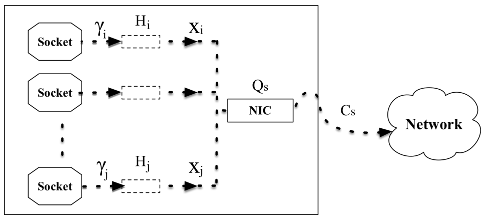


Fig. 2. At the output of sender server s, there is a counter H_i for the deficit between the "desired rate" γ_i and the "actual rate" x_i at each socket. Moreover, there is a queue Q_s in NIC (Network Interface Controller) that all the sending flows share.

A. Full-bisection Scenario

We first examine the case where the network has full bisection bandwidth, i.e., there is no link in the network that is over-subscribed. This is an idealistic situation, but it is a good starting point. Such a network can be a complete Benes network with packet-per-packet randomized routing [23] or it can be any non-blocking Clos networks [24]. In practice, these networks are expensive and one uses versions with over-subscribed link (discussed later).

In this network, the internal capacity constraints are automatically satisfied if they are satisfied at the input and output links. Thus, in (5), the sum capacity over the links j are that of either outputs or inputs of servers. In this expression, the Q_j are backlogs at the output and input links of servers, and no lasting backlog occurs in the switch, keeping the counter stable will enforce the capacity constraints.

In this case, to solve (5), we similarly adopt (3) and (4), i.e.,

$$\begin{aligned} \gamma_i &\in \arg\max U_i(\gamma_i) - \alpha H_i(t)\gamma_i \\ x_i &\in \arg\max x_i(H_i - \alpha Q_{s_i} - \alpha Q_{d_i}) \end{aligned}$$

Here Q_{s_i} and Q_{d_i} denote the backlogs at the source and destination for flow i, respectively. Fig. 2 shows the counters and queues at one sender.

To implement (3), one adjusts the value of γ_i, the rate at which H_i increases, as a function of the deficit x_i. As an example, if we use the logarithm utility function, i.e.,

$$U_i(x) = \omega_i \log(1 + x),$$

then we find that

$$\gamma_i = \max\{0, \frac{\omega_i}{H_i} - 1\}. \qquad (6)$$

Note that γ_i tends to drive H_i to the value ω_i. In practice, one may introduce a rate limit to γ_i that corresponds to the desired rate of the application. Observe that a flow with a large weight typically has a large value of H_i, and its service rate gets matched to its target rate faster.

To implement (4), we use the following rule:

$$x_i = \begin{cases} 0, & \textit{if } H_i < \alpha(Q_{s_i} + Q_{d_i}) \\ x_{\max}, & \text{otherwise} \end{cases}$$

Here, $x_{\max}$ is the maximum rate defined by the internal dam rate. This control turns off x_i when the backlog in the output queue Q_{s_i} and Q_{d_i} exceeds H_i. From our previous discussion about H_i, we see that flows with a small weight are turned off before the other flows. This mechanism prevents the less urgent packets from clogging up the NIC buffer and delaying more urgent flows, similarly for backlogs at the destinations.

B. Over-subscription Scenario

The above describes the full bisection bandwidth case. However, links are often over-subscribed in a practical network. Thus, backlog can appear inside switches. To handle this, note that mathematically, the summation over j in (5) includes internal links. We have seen how to enforce the capacity constraints at the inputs and outputs of the networks using counters. To also enforce the constraints inside the network, we replace

$$Q_j(t)[(A\boldsymbol{x})_j - C_j]$$

in (5) by

$$d_j(t)[(A\boldsymbol{x})_j - C_j]$$

where $d_j(t)$ is the delay through link j. Technically, this step is justified by replacing $\sum_j Q_j^2$ to $\sum_j a_j Q_j^2$ in the original Lyapunov function, where a_j is equal to one for input and output links of the network and $a_j = \delta / R_j$ for an internal link, where R_j is the transmission rate of the link and δ is some constant we can tune during implementation. With this modification, the solution of the maximization objective (4) becomes

$$x_i = \begin{cases} 0, & \textit{if } H_i < \alpha(Q_{s_i} + Q_{d_i} + \delta D_i) \\ x_{\max}, & \text{otherwise} \end{cases} \tag{7}$$

where D_i is the delay experienced by flow i. To measure the packet delay D_i, we use time stamps in the packet headers. We will describe the implementation details in the next section.

IV. Implementation

We have implemented a prototype of RTCP and we now present the three main components of our implementation, including congestion estimation, rate control and congestion smoothing.

A. Congestion Estimation

In RTCP, we adjust the counter increasing and decreasing rates by comparing the counter size and the "degree" of total congestion. In order to keep the RTCP implementation minimal, we assume the end-host delay (interrupts, noisy neighbors, etc.) as a constant and use the change in Round Trip Time (RTT) as an estimation of the total link congestion in our testbed experiments, e.g., as done in [6]. RTT is an available variable calculated in the TCP stack and this approach has been proven feasible and effective in our case.

In our `ns-3` simulations, we evaluated another option to use both the RTT changes, which implies the link congestion, plus the sender buffer size as an indicator of total congestion. The reason to use only the sender buffer is to avoid the potential influence due to the stale receiver buffer information. This also simplifies the RTCP implementation, because we do not need to reserve new bits in packet headers for transmitting the receiver's data to the sender.

B. Congestion Smoothing

Since RTCP reacts to congestion and updates its counters and rates based on network observation, it is critical to minimize the potential impact of fluctuations of the measured statistics. Thus, we adopt an Exponential Smoothing method [25] to update and shape the congestion statistics showed in (8), where S denotes the current delay estimation and N is the newly observed congestion size, i.e.,

$$S = g \times S + (1 - g) \times N \tag{8}$$

The smoothing coefficient is chosen to be $g = 0.5$ to balance the new delay observation and history.

C. Rate Control

We adjust the window size each time RTCP receives an acknowledgement packet, while leaves the other default standards unchanged. The rate control component of RTCP is carried out based on (6) and (7). In particular, we project x's dynamic changes to "congestion window" in network stacks as following:

$$\texttt{tcp\_cWnd} = \begin{cases} \texttt{tcp\_cWnd}/2, & H_i < \alpha(Q_{s_i} + Q_{d_i} + \delta D_i) \\ 2 \times \texttt{tcp\_cWnd}, & \text{otherwise} \end{cases} \tag{9}$$

Here recall that the parameter α determines the trade-off between utility and delay. Another tuning parameter δ represents "tolerance" of internal link delays.

The counter value, on the other hand, can be updated more easily using (6) and (1). To ensure the utility performance, we choose to halve the size of γ when we shall set the counter changing rate to zero, i.e.,

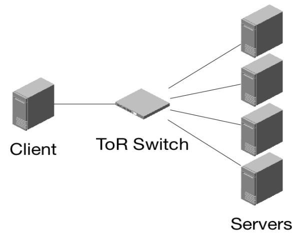
$$\gamma_i = \begin{cases} \texttt{maxRate}, & \text{if } H_i = 0 \\ \gamma_i/2, & \text{if } H_i \geq \omega_i \\ \omega_i/H_i - 1, & \text{otherwise} \end{cases} \tag{10}$$

V. Evaluation

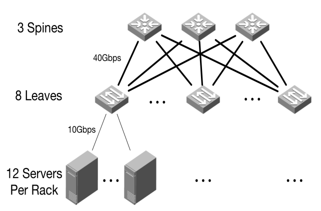
In this section, we present our evaluation setup and results on both a testbed and `ns-3` simulator. We conduct the testbed evaluation on two types of traffic, i.e., web search (WS) and data mining (DM), and measure the performance of RTCP in complex scenarios in `ns-3`.

A. Testbed Experiments

1) Testbed Setup: Our testbed consists of 4 servers and 1 client, connected via a Cisco Nexus 3048 48-port Gigabit Ethernet Switch with 9 MB shared memory. Each server is a Dell Precision Tower 5810 machine with a 4-core Inter(R) Xeon(R) E5-1620 3.5GHz CPU, 4G memory, a 500G hard disk and an Intel Corporation Ethernet Connection I217-LM (rev05) Ethernet NIC. The client uses a Dell OptiPlex 7010 machine with a 4-core Intel(R) Core(TM) i5-3470 3.2GHz CPU, 4G memory, a 500G hard disk and an Intel Corporation 82579LM Gigabit Network Connection(rev04) NIC. The servers and client run CentOS 7.4-64bit with Linux 4.14.9-1 Kernel. The testbed topology is shown in Fig. 3(a), which mimics a simple data-fetching scenario and we keep default network settings,



(a) Topology in testbed experiments.



(b) Spine-leaf in ns-3 simulation.

Fig. 3. (Left) Testbed topology: One client is connected to four servers through a ToR switch with 1Gbps Ethernet links. (Right) ns-3 topology: Spines and leaves are fully meshed with 40Gbps and each leaf switch (ToR) has linked 12 servers with 10Gbps.

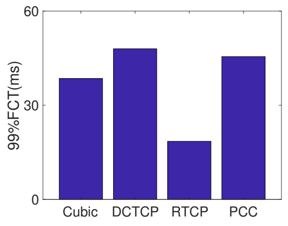
e.g., MTU size and TCP segmentation offload feature remain unchanged. The basic round trip time (RTT) of our testbed is 200 μs.

•**Testbed Parameter Settings.** We keep the suggested default settings for each baseline congestion control algorithms unchanged, and use a default drop-tail policy of each switch for end-to-end congestion control algorithm. To fit different traffic pattern, RTCP uses a fixed $\delta = 125$ and sets different values for variable α in testbed experiments. For example, we set a big α for the WS workload to keep a small buffer occupation, which tends to achieve a better FCT performance for urgent flows. For the DM workload, we use a smaller α because the long flows are heavy and we want to keep an overall better goodput performance.[1]

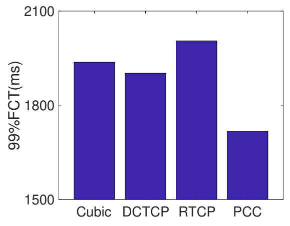
•**Traffic Generation.** We use the traffic generator borrowed from [15], and run congestion control algorithms with two realistic workloads, which include the web search workload given by [3] and the data mining workload offered by [26]. The client application periodically sends requests with a Poisson interval to gather data with specific distribution features from the servers. The server application responds to requests and sends exact data to the client based on the size requested. In our testbed experiments, we use a "separating threshold" 30KB to separate flows between "urgent" which is less than this threshold, and "high-volume" for others. Note that one can also use other separating thresholds to provide specific flows with a higher priority. We set the weight of urgent flows to 2.4×10^4 while give high-volume flows 1.2×10^4. The weights are used to compare with the counter to guide the rate changes.

2) Testbed Experiments Results: We use the topology in Fig. 3(a) to test the performance of the client-server model. In the beginning, we use only one client and one server to run a simple one-on-one data transfer test. The benchmarks we compare RTCP with include Cubic [27], DCTCP, and PCC [2]. We use a fixed load expectation to generate dynamic traffics according to the DCTCP workloads and measure each flow's

[1]Goodput is defined as the ratio of flow size and flow completion time. In simulations, we use throughput defined to be the rate of successful message delivery over a communication channel instead for the convenience of measurement. Note that both goodput and throughput measure how fast messages are transmitted.

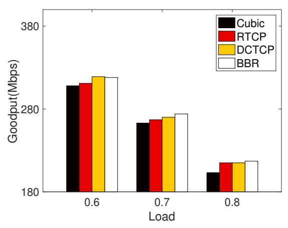


(a) Urgent flows

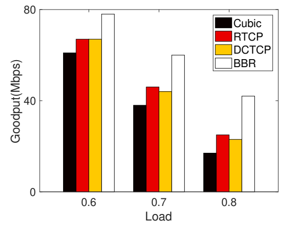


(b) High-volume flows

Fig. 4. Urgent and high-volume flows' 99th FCT in one-to-one fetching. RTCP shows a tradeoff between the urgent and high-volume flows, which throttles the high-volume flows a little to guarantee a small FCT urgent flows. PCC keeps high-volume flows a smaller 99th FCT. DCTCP and Cubic have little FCT difference between urgent and high-volume flows.



(a) WS workload



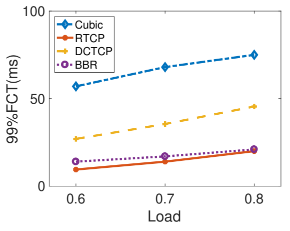
(b) DM workload

Fig. 5. Average goodput under web search (WS) and data mining (DM) workloads in one-to-four fetching. Under WS workload, four TCP versions show comparable goodput but RTCP shows the slowest fall with the increase of load. Under DM workload, BBR shows the best goodput and RTCP performs better than Cubic and DCTCP.

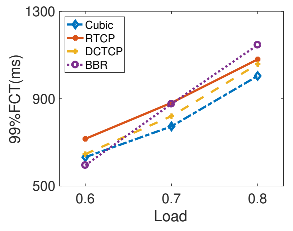
FCT and goodput. For RTCP, we choose $\alpha = 500$.

For different congestion control versions, the average goodput of these two server scenarios are almost the same (around 110Mbps) because the bandwidth is not a competing resources at this time. Fig. 4 shows the flow completion time of urgent and high-volume flows. We find that DCTCP does not show a good FCT for both flows because the link is not congested enough to mark the packets and DCTCP controls the flow with NewReno [28] pattern most of the time. Compared to DCTCP and PCC, Cubic is more friendly with urgent flows and achieves a lower 99th FCT. PCC does not consider delay during its utility function design and we find it only achieves good FCT performance for high-volume flows. RTCP shows a tradeoff between the urgent and high-volume flows: a higher FCT for high-volume flows to keep urgent flows arrive in time. Note that this tradeoff does not affect the goodput compared to other benchmarks.

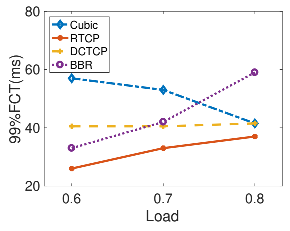
Does this tradeoff work well when scenarios get complex? To answer this question, we use the topology above to run a file fetching test: one client sends requests to four servers randomly and the servers send the exact packets the client requests back. As the average traffic load is moderate (for example, 30% [29]) and a long-term load of over 80% is less likely in practice [15], we test the network workload from 0.6 to 0.8, to investigate whether RTCP is robust to different link congestion status. To match the different flow patterns, we set tuning parameter $\alpha = 800$ for WS workload and $\alpha = 15$ for DM workload. The benchmarks we use in this experiment are Cubic, DCTCP and BBR [1]. Fig. 5 shows the average



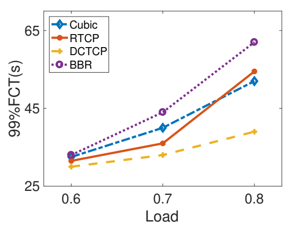
(a) 99th FCT of urgent flows under WS workload



(b) 99th FCT of high-volume flows under WS workload



(c) 99th FCT of urgent flows under DM workload

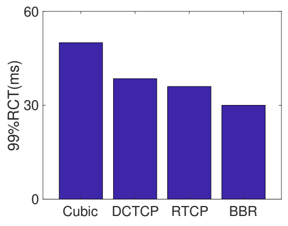


(d) 99th FCT of high-volume flows under DM workload

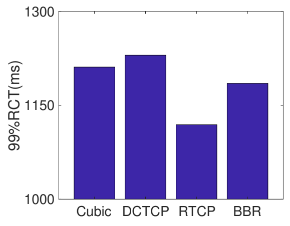
Fig. 6. 99th FCT in one-to-four fetching. In both workloads, RTCP throttles the high-volume flows a little to achieve the smallest FCT for urgent flows. Cubic does not handle urgent flows well but it keeps a good FCT for high-volume flows. BBR shows good performance for urgent flows but as the load increases, high-volume flows of BBR suffer more than urgent flows. DCTCP achieves a mild performance in most cases but figure (d) shows that its simple dynamic is friendly to long flows.

goodput of different versions of TCP under the two workloads. We find that under WS workload, the goodput performance is comparable but as the load grows, Cubic decreases faster than other three versions. Under the DM workload, BBR is better than the other three. Cubic has the worst performance, and RTCP is comparable or slightly better than DCTCP. With this goodput performance, we move our eyes to the 99th FCT, which is showed in Fig. 6. We find that although all four versions of TCP show comparable goodput performance in WS workload, their FCT features are different. RTCP achieves a smaller FCT for urgent flows shown in Fig. 6(a), but it throttles the high-volume flows a little. Cubic works in an opposite way: it keeps a small FCT for high-volume flows but leaves urgent flows a higher FCT. BBR works well at load 0.6, but as the load increases, its FCT for high-volume flows grows faster than others. DCTCP shows a regular performance for both urgent and high-volume flows. In DM workloads, BBR suffers from higher FCT especially when the network gets congested because it seeks for a better goodput performance, which spends lots of heavy-delay packets testing the best sending rate. DCTCP handles the queues well and shows a better FCT for high-volume flows. RTCP keeps its better performance for urgent flows but as the load increases, the sacrifices that high-volume flows make are getting large. An interesting finding is that as the load increases, Cubic prefers to keep the urgent flows a lower FCT. We infer the reason for this FCT decrease is that the frequent high-volume packets timeout gives up the bandwidth for urgent flows, resulting in a smaller FCT as the network becomes more congested.

3) Incast Experiments: Now we study whether RTCP handles the incast problem well. We keep the one client and four servers scenarios, but this time each client packet requests $(1, 2, 8)$ flows according to distribution of $(0.5, 0.3, 0.2)$, i.e., 50% packets requests data from one flow, 30% requests two flows and 20% requests eight flows. To make RTCP sensitive enough for the concurrent packets, we set $\alpha = 10^3$. We measure the request completion time (RCT) to see whether RTCP, Cubic, DCTCP and BBR handle the incast problem well at load 0.8 under the WS workload. Fig. 7 shows the RCT for urgent and high-volume flows. We find that RTCP performs well for both urgent and high-volume flows, especially for high-volume flows as RTCP tries to keep a small queue



(a) Urgent flows



(b) High-volume flows

Fig. 7. 99th RCT (Request Completion Time) in incast experiments. RTCP and BBR outperform Cubic and DCTCP for both flows. BBR outperforms RTCP for urgent flows by a narrow margin as it finds the best sending window, but RTCP outperfroms BBR in high-volume.

occupation in congested scenarios. BBR has a lower RCT for urgent flows. DCTCP keeps a low RCT for short flows but the window adjustment of DCTCP is not friendly to high-volume flows. Cubic shows the worst performance in this incast test.

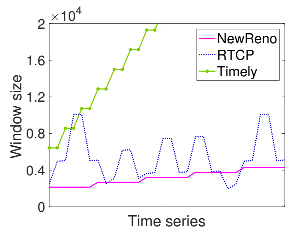
B. `ns-3` *Simulations*

In the previous section, we have conducted RTCP experiments on a testbed. In this section, we present our experimental results in `ns-3` [30], to evaluate the throughput and delay performance of RTCP flows in a system with larger scale.

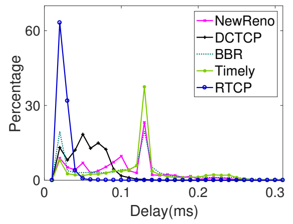
1) Setup: We conduct our simulation with a spine-leaf topology showed in Fig. 3(b). This is a multi-path, multi-bottleneck topology which is commonly adopted in modern data centers [10]. We have 3 spine switches and 8 leaf switches, and each leaf switch is linked to 12 servers. The link bandwidth between the server and leaf switch is set to 10Gbps, and each leaf switch is linked to each spine switch with a 40Gbps bandwidth. As the topology gets complex and we want a better delay reduction, we use $\delta = 10^4$ in simulations if we do not specific the parameter settings in each scenario. The delay of each link is set to $1\mu s$.

One important change from the testbed experiments is that we offer RTCP the sender buffer size in `ns-3` simulations, which means that we allow RTCP to read the current buffer size in NIC for decision making. This implementation is essential and meaningful to evaluate how RTCP performs with more accurate delay estimation.

2) Random sending test: We first initiate the simulation with a random sending test. In this scenario, each server



(a) Window size changes



(b) p.d.f of packets delay

Fig. 8. Window size changes and p.d.f. of packet delay in random sending test. TIMELY accelerates the window increases compared to NewReno; RTCP adjusts the window based on the congestion size measured. Packets under RTCP experience a 99th delay that is only 16% of that under NewReno and an average delay that is 35.78% of that under DCTCP. BBR shows a heavy tail in p.d.f. and NewReno suffers from significant packet loss.

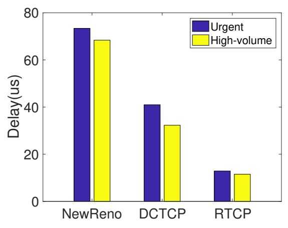
TABLE I
AVERAGE THROUGHPUT AND DELAY PERFORMANCE IN RANDOM SENDING(RS) AND OVER-SUBSCRIPTION(OS) SCENARIOS

Version	Avg. Throughput(Mbps) in RS/OS	Avg. Delay(us) in RS/OS
NewReno	4300.38/2753.2	85.4/129.9
DCTCP	3955.55/2528.9	40.8/56.6
BBR	3936.28/2418.56	87.7/113.6
TIMELY	4355.48/2839.13	107.9/156.8
RTCP	4362.38/2787.08	14.6/23.4

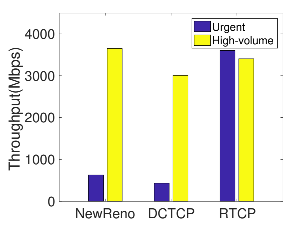
randomly chooses another server to start the data transmission. We keep the setting $\alpha = 500$ for RTCP in this scenario, and use this basic test to see whether RTCP achieves a better performance than NewReno, DCTCP, TIMELY [5] and BBR[2].

We show the average throughput and delay performance in table I and use Fig. 8 to show the window size changes and packets delay performance. We randomly choose one server and plot the window size changes of one determined time series. We compare RTCP window with TIMELY and NewReno, which show the similar throughput performances with RTCP. We find that TIMELY accelerates the window increment compared to NewReno, which explains its attractive throughput and bad delay performance. NewReno increases the window in a slow linear speed. RTCP adjusts the window size based on the congestion size each time receives a packet, which better suits the current congestion condition. NewReno's packet delay distributes from 0 to $0.2ms$ and more than 50% packets suffers from delay beyond $0.1ms$. Most DCTCP packets delay less than $0.1ms$ but DCTCP dose not provide flows with a better throughput. BBR uses probes to find a better sending window, which improves the percentage of packets that experience small delay, but it causes heavy tail for the delay p.d.f. compared to NewReno. RTCP achieves a comparable throughput with NewReno and TIMELY while ensuring a 99th delay that is only 16% of that under NewReno and an average delay that is 35.78% of that under DCTCP. It uses a moderate window changes to achieve best throughput performance and greatly reduces the delay variability compared to other TCP versions. The moderate window size changes of RTCP also indicates that RTCP, which adjusts the window based on congestion rather than packet losses, achieves better fairness between servers than NewReno and TIMELY if all servers adopt the same RTCP congestion control scheme.

[2]Note that the BBR module we used in `ns-3` has implemented most BBR conrol schemes. Yet, it has not supported all recent BBR features. Please check [31] for more details.

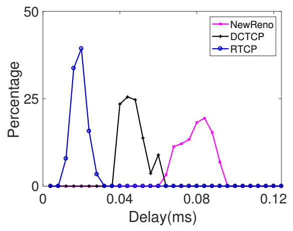


(a) Average Delay

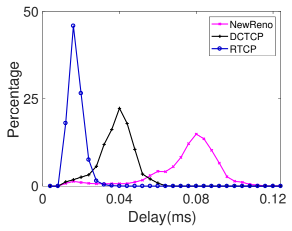


(b) Average Throughput

Fig. 9. Average throughput and delay of urgent flow in the preemption test. RTCP achieves a better delay performance for both flows. NewReno doubles the average delay to tradeoff a throughput increase compare to DCTCP.



(a) Urgent flow p.d.f



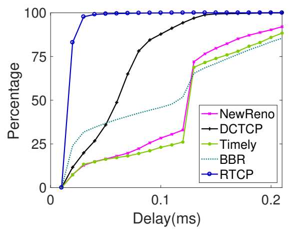
(b) High-volume flow p.d.f

Fig. 10. Packets delay p.d.f. in the urgent flow preemption test. RTCP reduces the 99th delay by almost a half compared to DCTCP, and its delay is only 25% of those under NewReno for both high-volume and urgent flows. All RTCP packets enjoy smaller delay than NewReno and DCTCP.

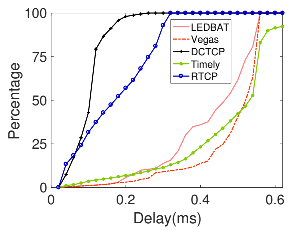
3) Urgent flow preemption test: In this scenario, we initiate one high-volume flow to the right side adjacent server for each server in the topology above. After that, we start one urgent flows at each server which has the same source and destination IP with the high-volume one. Note that in this test, we keep the flow types "urgent" and "high-volume" from testbed experiments to indicate that urgent flows have a higher priority. We study how urgent flows preempt background flows, so as to ensure fast delivery, under different priorities of flows. We keep the mild setting of α unchanged.

Fig. 9 shows the average delay and throughput performance of urgent and high-volume flows. For both urgent and high-volume flows, RTCP achieves a better delay performance: an average delay that is only 31.5% of the average delay under DCTCP for urgent flows and 35.6% for high-volume flows. It also throttles the high-volume flow a little bit (7% throughput compared to NewReno) to provide urgent flow with a good preemption: almost 4 to 7 times increase of throughput compared to NewReno and DCTCP.

We plot the packet delay p.d.f. in Fig. 10. We see that even for high-volume flows which are throttled for urgent flows, RTCP significantly reduces their delay and delay variability. In particular, it achieves a 99th delay that is almost a half of that under DCTCP, and only about 25% of that under NewReno. For urgent flows, RTCP narrows the packet delay distribution



(a) c.d.f in the over-subscription scenario



(b) c.d.f in the large scale incast test

Fig. 11. c.d.f. of packets delay in over-subscription scenario and large scale incast test. The 99th delay of RTCP is significantly better than DCTCP, NewReno, TIMELY and BBR in the over-subscription scenario. In large scale incast, RTCP outperforms the other three end-to-end based TCP versions and narrows the gap from DCTCP.

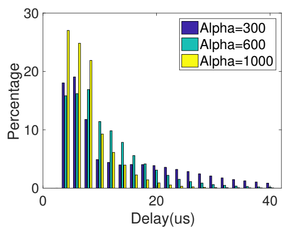
and all RTCP packets enjoy smaller packets delay than under NewReno and DCTCP.

C. Deep dive

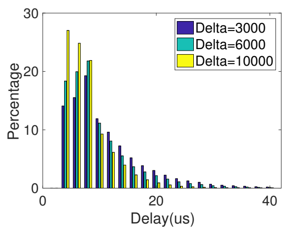
1) Over-subscription scenario: Over-subscription occurs commonly in practice, e.g., due to cost saving or under-provision of network resources. Thus, we also investigate how RTCP performs in the over-subscription scenario. We rerun the random sending test above but links are connected in an over-subscription fashion this time. Specifically, we keep the server port as 10Gbps, but the bandwidth between spine switches and leaf switches is reduced to $\frac{1}{2}$ of the original link capacity. We set $\alpha = 10^3$ in RTCP to make RTCP more sensitive to the backlog. The throughput and delay results are also showed in table I. RTCP handles the congestion window well. Compare to TIMELY, it uses 1.83% throughput loss to achieve 85.08% delay reduction. Fig. 11(a) shows the c.d.f of packet delay. We see that the 99th delay of RTCP is significantly better than DCTCP, NewReno, TIMELY and BBR. From the results, we see that RTCP can effectively reduce packet delay even in this much congested situation.

2) Large-scale incast test: End-to-end protocols may suffer from the incast problem because they use RTT as an estimation of link congestion. Many protocols use switches as a helper of indicating link congestion, such as ECN or priority queues techniques. However, if we do not seek the help of switches, how well can these end-to-end protocols perform? In this scenario, we test how RTCP perform compared to other RTT-based end-to-end protocols, including LEDBAT [4], TIMELY and Vegas. Also, we choose DCTCP, which tackles the incast problem well with the help of ECN, as a baseline. We use the topology above to mimic a 95 to 1 incast test. We set the $\alpha = 5 \times 10^3$ to make RTCP focus on delay reduction. Also, we set the initial window of RTCP as 2 instead of the suggested 10 [32] to avoid backlogging too many packets at the beginning.

Figure 11(b) shows the packets c.d.f. in this scenario. DCTCP shows the best performance with the help of ECN. RTCP reduces the 99th packet delay to about 56% of LEDBAD and Vegas. TIMELY suffers from heavy delay tail compared to other RTT-based congestion control algorithms. This test shows that RTCP can handle the incast problem



(a) The effect of α



(b) The effect of δ

Fig. 12. Comparison of p.d.f with different parameter settings. Increase the parameter values will improve the delay performance.

TABLE II
PARAMETER COMPARISONS FOR RTCP

Parameter Settings	Avg. Throughput	Avg. Delay
$\alpha = 3{\times}10^2$, $\delta = 10^4$	3180.16 Mbps	55.6 us
$\alpha = 6{\times}10^2$, $\delta = 10^4$	3208.76 Mbps	30.2 us
$\alpha = 10^3$, $\delta = 10^4$	2787.08 Mbps	23.4 us
$\alpha = 10^3$, $\delta = 6{\times}10^3$	3028.40 Mbps	24.6 us
$\alpha = 10^3$, $\delta = 3{\times}10^3$	3073.12 Mbps	29.9 us

better than other end-to-end protocols (LEDBAT, TIMELY and Vegas). RTCP shows great flexibility to fit different scenarios.

3) Parameter issues: We have already seen that in the above experiments the α value can be used to achieve a proper utility-delay tradeoff in the network, i.e., a larger α prefers congestion reduction while a smaller α favors overall utility increase. We next try to understand how δ (used in (7) to measure the link backlog) affects the RTCP performance. Table II compares the average throughput and delay of flows with different parameter settings in the over-subscription scenario. Comparing row 1, 2, 4 and 5 with row 3, we find that the decrease of δ can improve the throughput performance but hurt delay, like the parameter α did . Fig. 12(a) and 12(b) shows that the packet delay and jitters are reduced as the decrement of α and δ value. Yet, it shall be noticed that the δ and α values cannot be made arbitrary large or small. A large parameter value may cause an overreaction of RTCP to delay change which improves the delay but hurts the flow throughput severely, while a small value may increase the packet backlog in the intermediate devices which decreases the throughput unexpectedly(as showed in row 1).

VI. LIMITATIONS AND FUTURE WORK

RTCP is a modest congestion control scheme, which performs good fairness among the servers with RTCP implementation. However, RTCP may give up a small fraction of bandwidth in order to achieve a better delay performance when competing with other greedy buffer-occupation algorithms, such as NewReno and BBR.

Many recent works focusing on congestion control try to optimize the control policy with reinforcement learning, for example, [33] and [34]. Using reinforcement learning techniques for finding proper parameters for RTCP for different network scenarios will be an interesting yet challenging future research problem.

VII. Conclusions

In this paper, we propose Regulated TCP (RTCP) based on a novel auxiliary variable solution approach in optimization theory. RTCP uses counters at end devices and end-to-end delay measurements for rate adaptation (via window adjustment). It requires minimal changes to existing network systems and can be implemented with simplest switch configuration. We implement a prototype of RTCP and evaluate RTCP through a series of testbed experiments and `ns-3` simulation. Our results show that RTCP achieves better performance, in terms of packet delay and goodput, compared to existing TCP protocols including DCTCP, BBR, PCC, LEDBAT, TIMELY, Vegas, Cubic and NewReno.

Acknowledgment

The work of Longbo Huang and Yuxing Li is supported in part by the National Natural Science Foundation of China Grant 61672316, the Zhongguancun Haihua Institute for Frontier Information Technology and the Turing AI Institute of Nanjing.

References

[1] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):20–53, 2016.

[2] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. {PCC}: Re-architecting congestion control for consistent high performance. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 395–408, 2015.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[4] Sea Shalunov, Greg Hazel, Janardhan Iyengar, Mirja Kuehlewind, et al. Low extra delay background transport (ledbat). In *RFC 6817*, 2012.

[5] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[6] L.S. Brakmo and L.L. Peterson. Tcp vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 2002.

[7] Ali Munir, Ihsan A Qazi, Zartash A Uzmi, Aisha Mushtaq, Saad N Ismail, M Safdar Iqbal, and Basma Khan. Minimizing flow completion times in data centers. In *2013 Proceedings IEEE INFOCOM*, pages 2157–2165. IEEE, 2013.

[8] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[10] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187, 2016.

[11] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 37–48, 2005.

[12] Kadangode Ramakrishnan, Sally Floyd, David Black, et al. The addition of explicit congestion notification (ecn) to ip. 2001.

[13] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[14] Li Chen, Shuihai Hu, Kai Chen, Haitao Wu, and Danny HK Tsang. Towards minimal-delay deadline-driven data center tcp. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[15] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 455–468, 2015.

[16] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE ACM Transactions on Networking*, 1(4):397–413, 1993.

[17] C. V. Hollot, Vishal Misra, Donald F. Towsley, and Weibo Gong. On designing improved controllers for aqm routers supporting tcp flows. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2001.

[18] Frank Kelly. Charging and rate control for elastic traffic. *European transactions on Telecommunications*, 8(1):33–37, 1997.

[19] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

[20] Scott Moeller, Avinash Sridharan, Bhaskar Krishnamachari, and Omprakash Gnawali. Routing without routes: The backpressure collection protocol. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 279–290, 2010.

[21] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE ACM Transactions on Networking*, 8(5):556–567, 2000.

[22] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12):1936–1948, 1992.

[23] Longbo Huang and Jean Walrand. A benes packet network. In *2013 Proceedings IEEE INFOCOM*, pages 1204–1212. IEEE, 2013.

[24] Jean Walrand and Pravin Pratap Varaiya. *High-performance communication networks*. Morgan Kaufmann, 2000.

[25] Robert G Brown and Richard F Meyer. The fundamental theorem of exponential smoothing. *Operations Research*, 9(5):673–685, 1961.

[26] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[27] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *Operating Systems Review*, 42(5):64–74, 2008.

[28] Sally Floyd, Tom Henderson, Andrei Gurtov, et al. The newreno modification to tcp?s fast recovery algorithm. 1999.

[29] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[30] `ns-3`. https://www.nsnam.org/ Accessed February 5, 2010.

[31] Mark Claypool. Bbr' - an implementation of bottleneck bandwidth and round-trip time congestion control for `ns-3`. https://github.com/mark-claypool/bbr Accessed February 5, 2020.

[32] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing tcp's initial congestion window. *ACM SIGCOMM Computer Communication Review*, 40(3):26–33, 2010.

[33] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 191–205, 2018.

[34] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.