INFOCOM 2023 1570827540

Gemini: Divide-and-Conquer for Practical Learning-Based Internet Congestion Control

Wenzheng Yang*†, Yan Liu†, Chen Tian*, Junchen Jiang‡, Lingfeng Guo†
*State Key Laboratory for Novel Software Technology, Nanjing University, China
†Cloud ARCH & Platform Dept., Tencent, China
‡University of Chicago, USA
{rosswzyang, rockyanliu, lingfengguo}@tencent.com, tianchen@nju.edu.cn, junchenj@uchicago.edu

***Abstract*—Learning-based Internet congestion control algorithms have attracted much attention due to their potential performance improvement over traditional algorithms. However, such performance improvement is usually at the expense of black-box design and high computational overhead, which prevent them from large-scale deployment over production networks. To address this problem, we propose a novel Internet congestion control algorithm called Gemini. It contains a parameterized congestion control module, which is white-box designed with low computational overhead, and an online parameter optimization module, which serves to adapt the parameterized congestion control module to different networks for higher transmission performance. Extensive trace-driven emulations reveal Gemini achieves better balances between delay and throughput than state-of-the-art algorithms. Moreover, we successfully deploy Gemini over production networks. The evaluation results show that the average throughput of Gemini is 5% higher than that of Cubic (4% higher than that of BBR) over a mobile application downloading service and 61% higher than that of Cubic (33% higher than that of BBR) over a commercial network speed-test benchmarking service.**

***Index Terms*—Transport Protocol, Internet Congestion Control**

I. Introduction

Transmission Control Protocol (TCP) has been adopted by default for most applications on the Internet (*e.g.,* file downloading, video streaming, *etc.*), where congestion control (CC) algorithm plays a crucial role in the overall transmission performance of TCP.

Traditional CC algorithms (*e.g.,* Cubic [1] and BBR [2]) are crafted by human experts and widely deployed for production networks. They enjoy two advantages. First, the handcrafted algorithms facilitate easy analysis and debugging by operators, so when a traditional CC algorithm performs unexpectedly, operators can quickly localize and resolve the performance issues. Second, the handcrafted algorithms are usually with negligible computation and memory overhead, which can be easily built into compute-intensive and memory-intensive production services (*e.g.,* content delivery networks). However, due to their "one-size-fits-all" design, these handcrafted algorithms are becoming less capable of satisfying the increasingly diverse application requirements over highly complex network environments.

Recently, learning-based CC algorithms (*e.g.,* Remy [3], PCC [4], Vivace [5], Indigo [6], and Orca [7]) have gained much attraction due to their potential adaptivity to various network environments. Unlike the aforementioned traditional CC algorithms, the learning-based CC algorithms learn the congestion control directly from network environments based on various machine learning models (*e.g.,* reinforcement learning [8], LSTM [9], *etc.*), so intuitively it can adapt to different network environments to achieve higher transmission performance.

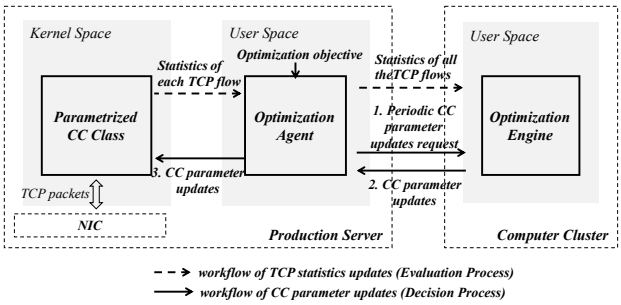


Fig. 1: The architecture of Gemini

However, existing learning-based CC algorithms usually suffer from the following problems for practical deployment over production networks. First, existing learning-based CC algorithms are usually with high computation and memory overhead (see Section V-B2), which directly prevents them from being successfully deployed in production environments as they cannot serve thousands of requests concurrently with reasonable resources. Second, the performance of existing learning-based CC algorithms (*e.g.,* Remy [3] and Indigo [6], *etc.*) rely heavily on the representativeness of training environments, and they may underperform under *unseen* network environments. Given highly complex real-world network environments, it would be difficult, if not impossible, for operators to train the algorithms to cover all the possible network environments, because corner-case network environments are always endless. Finally, since the congestion control is typically a learned black box and is usually hard to be understood, it adds more difficulties to further performance optimization and troubleshooting for the operators.

To address the above problems and enable practical deployment over production networks, we propose a divide-and-conquer framework to design practical learning-based CC

algorithms, called Gemini. Specifically, Gemini consists of the following two major modules:

- **A parameterized CC class, called Fusion**. We model the congestion control algorithm as the *control logic* and the *control parameter*. The control logic determines the reactions to specific events for data transmissions, and the control parameter controls the degree of reaction to specific events. For example, with the occurrence of a congestion event, the congestion window of Cubic will be reduced to a size that equals to β (*e.g.,* 0.7 by default) times the recent maximum non-congested congestion window. So in this case, the window reduction behavior is part of the control logic and the parameter β is the corresponding control parameter. Fusion is handcrafted to keep the control logic of Gemini unchanged, ensuring a white-box design and low computation overhead, and at the same time enables the control parameters of Gemini dynamically adjustable to achieve flexibility for further performance optimization by the online optimization engine.
- **An online optimization engine, called Booster**. Booster is introduced to boost the adaptability and thus improve the performance of Gemini under different network environments by automatically optimizing the control parameters with a given *performance objective* (also known as *utility function*).

In this study, Gemini is fully implemented and readily deployable over production networks. Specifically, Fusion is implemented as a Linux kernel module running in kernel space, and Booster is implemented as an online optimization engine running in the user space in a computer cluster. We compared Gemini with existing congestion control algorithms over both emulated networks and production networks. In emulated networks, we compared Gemini with state-of-the-arts congestion control algorithms in a controlled manner. We showed that Gemini achieved a better performance balance between throughput and delay, and a comparable RTT (round trip time) fairness and Cubic friendliness (see Section V-A). In production networks, we compared Gemini with existing widely deployed congestion control algorithms over two types of production services. We observed Gemini improved the average throughput by 5% compared with Cubic (4% compared with BBR) over a mobile application downloading service and by 61% compared with Cubic (33% compared with BBR) over a commercial network speed-test benchmarking service [10] (see Section V-B).

The remainder of the paper is organized as follows. Section II reviews the related works; Section III presents the design of Gemini; We describe the implementation details of Gemini in Section IV and evaluate its performance in Section V; In Section VI we summarize the study and outlines some future work.

II. Background and Related Work

Numerous CC algorithms have been proposed over the past three decades, including Vegas [11], Cubic [1], BBR [2], and Copa [12]. These CC algorithms are carefully designed, easy to be understood by operators, and their fairness and friendliness have been well analyzed and reported. However, hardwired design is less capable of satisfying the performance requirement of diverse applications over different network environments. Although existing studies have designed a serial of CC algorithms for different scenarios (*e.g.,* Sprout [13] and Verus [14] are designed for mobile networks), their performance deteriorates when being applied to non-target network environments (See Fig. 4).

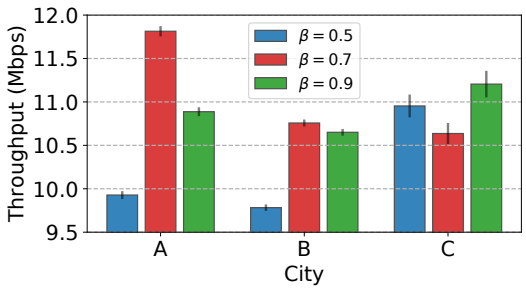


Fig. 2: Throughput variations of Cubic with different control parameter β from production environments over one week

Production service operators usually choose to manually tune the parameters of existing congestion control algorithms (*e.g.,* Cubic or BBR) for performance optimization since it is widely observed that by changing the default parameter of CC algorithms for different network environments we can probably achieve higher performance (as shown in Fig.2, although default parameter β = 0.7 in Cubic performs better in city A and B, we can still further achieve higher throughput in city C by changing it to 0.9.). However, manual parameter optimization for congestion control algorithms is usually time-consuming and cost-ineffective because of continuously emerging new network environments and multiple coupled parameters in the same algorithms, which may interfere with each other. Moreover, the design imperfection of existing algorithms also constrain the potential performance improvement by optimizing the parameters. For example, because of using packet loss as the only congestion signal, Cubic and Reno suffer from throughput degradation under random losses [2] and buffer-bloat [15] under large network queues. BBR suffers from high losses under the scenarios with small network queues and suffers from throughput unfairness under scenarios with large network queues because of its fixing congestion window to be twice of the BDP (short for Bandwidth Delay Product) [16]. By contrast, in this study, we propose a new general congestion *control logic* for performance optimizations to address the above problems.

Learning-based CC algorithms (Remy [3], PCC [4], Vivace [5], Indigo [6], and Orca [7]) have recently attracted much attention due to their potential adaptivity to various network environments. The control logic of these learning-based CC algorithms is automatically learned from training

environments. Thus, they can probably adapt to different network environments. Compared with traditional CC algorithms, learning-based CC algorithms currently offer a potential advantage in adaptability. However, it is usually at the expense of black-box design and a huge cost of model retraining in each environment. Moreover, the convergence, CPU/Memory overhead, and maintainability of learning-based CC algorithms also differ greatly from traditional CC algorithms. These capabilities are as important as adaptability for deployment in production environments. Configurator [17] proposed a practical learning-based optimization framework for the general parameters of CDN service, by contrast our study focus on the practical deployment of optimizations for Internet congestion control algorithms.

III. The Design of Gemini

Gemini (as shown in Fig. 1) consists of two major modules, *i.e.*, a parameterized CC class (called **Fusion** henceforth), which provides the congestion *control logic*, and an optimization engine (called **Booster** henceforth), which serves to optimize *control parameters* for Fusion to adapt to various network environments. To interconnect the above two modules, we further introduce a connector, called optimization agent, serving for data exchange between Fusion and Booster.

The workflow of Gemini is basically an iterative process cycle between *evaluation* and *decision*. Specifically, during the evaluation process, incoming TCP flows are served by Fusion with a set of *control parameters*, this parameter set is randomly initialized at the beginning and further optimized by the Booster afterward. After a preset time period (*e.g.,* 5 minutes) until a sufficient number of TCP flows are collected, the optimization agent will aggregate the performance statistics (*e.g.,* average throughput) of the flows served by Fusion with this control parameter set for the decision process. Then in the decision process, based on the aggregated performance statistics and the corresponding control parameter set, Booster will determine a new set of control parameters to Fusion for serving the next incoming TCP flows for optimizing a given performance objective.

A. The Parameterized CC Class *Fusion*

In this section, we propose a general parameterized CC Class Fusion, which serves the congestion control logic, by incorporating both window-based (*e.g.,* Cubic/Reno) and rate-based (*e.g.,* BBR) algorithm. Moreover, since packet loss alone is not the only good indicator of congestion event, we further introduce RTT variations (*e.g.,* Copa [12] and Vegas [18]) as an extra congestion signal into the algorithm.

Specifically, we use W_t to denote congestion window size at time t and W_0 to denote initial congestion window size. Moreover, Fusion keeps a sliding window (measured in # of RTT, denoted by n) for throughput and RTT sampling, namely, we use $R^{\max}_{t-n,t}$ to denote the measured maximum throughput and $T^{\min}_{t-n,t}$ ($T^{\max}_{t-n,t}$) to denote minimum (maximum) RTT sample. We use L_t to denote the loss rate (estimated by the retransmission rate) at time t. Then the data transmission behavior of Fusion is summarized as follows:

TABLE I: Control Parameters of Fusion

Control Parameter	Description
ω	Initial congestion window size
α	Multiplicative increase factor
γ	Additive increase factor
λ	Multiplicative decrease factor
l	Loss-tolerate threshold
δ	RTT inflation threshold
n	# of intervals for throughput and RTT sampling

- **Congestion Signal:** Once the packet loss rate exceeds threshold l or the queue occupancy rate (estimated by RTT inflation [2]) exceeds threshold δ, a congestion signal is detected.

$$L_t > l \quad \text{or} \quad T^{\min}_{t-1,t} - T^{\min}_{t-n,t} > \delta \cdot (T^{\max}_{t-n,t} - T^{\min}_{t-n,t}) \quad (1)$$

- **Slow Start:** At the slow start phase, Fusion compares the exponentially increased congestion window and estimated BDP (computed by $R^{\max}_{t-n,t} \cdot T^{\min}_{t-n,t}$) and will choose whichever is higher as the new congestion window, denoted by $W_t^{'}$.

$$W_t^{'} = \max(2 \cdot R^{\max}_{t-n,t} \cdot T^{\min}_{t-n,t}, W_t), \quad W_0 = \omega \quad (2)$$

- **Congestion Recovery:** Once a congestion signal or an exit point of slow start is detected [19], Fusion enters into the congestion recovery phase. During this phase, the congestion window size is clipped by the multiplicative-decrease factor λ following the proportional rate reduction algorithm [20], where $0 < \lambda < 1$.

$$W_t^{'} = \lambda \cdot W_t \quad (3)$$

- **Congestion Avoidance:** After the congestion recovery phase, Fusion enters into congestion avoidance phase. During this phase, the congestion window size continues to grow both multiplicatively and additively if the estimated BDP keeps increasing. If not, it grows only additively to occupy more network queues.

$$W_t^{'} = \max(\alpha \cdot R^{\max}_{t-n,t} \cdot T^{\min}_{t-n,t}, W_t) + \gamma \quad (4)$$

  where $\alpha \geq 1$ and $\gamma \geq 1$.

Once the congestion window size is determined, the pacing rate is set according to the current congestion window size W_t and minimum RTT $T^{\min}_{t-n,t}$, *i.e.*

$$R_t = W_t / T^{\min}_{t-n,t} \quad (5)$$

After determining the overall design of control logic, we can further make the corresponding control parameters listed in Table I dynamically adjustable, denoted by

$$\boldsymbol{\theta} := \{\omega, \alpha, \gamma, \lambda, l, \delta, n\} \quad (6)$$

To be more specific, the initial congestion window size has a significant impact for static web objects, and has been updated several times over the years [21]. Multiplicative increase factor

TABLE II: Statistics collected by the optimization agent

Flow-Level Performance Statistics	
$time$	Duration of TCP flow
$size$	Size of TCP flow
rtt_{avg}	Average RTT of TCP flow
rtt_{min}	Minimum RTT of TCP flow
Control Parameters used by Fusion	
$\boldsymbol{\theta}$	Control parameters of TCP for data transmission
Hidden Variables of TCP	
$region_{src}$	Source region of TCP flow
$region_{dst}$	Destination region of TCP flow
ISP_{src}	Source ISP of TCP flow
ISP_{dst}	Destination ISP of TCP flow
$timeStamp$	Time of day when TCP flow ends

α, additive increase factor γ, and multiplicative decrease factor λ determine the aggressiveness of the algorithm. The loss tolerance threshold l and RTT inflation threshold δ allow Fusion expressing the preferences between higher throughput and lower queuing delay. Sliding window size n determines how sensitive the algorithm is to the dynamic change of network environments in a short period. For example, cellular (network states change rapidly) and Ethernet (network states are relatively stable) networks will expect different values of n for optimal performance (see Section V-A for experimental evaluations).

B. The Optimization Engine ***Booster***

The Gemini by nature can be modeled as a black-box optimization problem with its application to TCP performance optimization, where we need to determine a $\mathbf{x}^*$ (*i.e.* the control parameters) that maximizes the objective function $f(\mathbf{x})$ of TCP performance over a *region* $\boldsymbol{\chi}$, i.e.,

$$\mathbf{x}^* = \arg\max_{\mathbf{x}\in\boldsymbol{\chi}} f(\mathbf{x}) \tag{7}$$

This is because we can only observe the results of $f(\mathbf{x})$ with a given $\mathbf{x}$ in real world deployment, rather than knowing any exact expressions of $f(\mathbf{x})$. For example, we don't know what control parameter values lead to higher throughput in production networks.

There are a couple of black-box optimization algorithms (*e.g.,* standard Bayesian optimization (BO) [22], Tree of Parzen Estimator (TPE) [23], Stochastic Radial Basis Function strategy (SRBF) [24] and OnePlusOne evolution algorithm (OnePlusOne) [25], *etc.*), in this study we introduce Bayesian optimization as a basis to our performance optimization due to its fast convergence, low computational overhead and mature implementation library.

BO employs *historical trials* (*i.e.*, set of the $(\mathbf{x}, f(\mathbf{x}))$ pairs) to fit a surrogate model (*e.g.,* Gaussian process) that characterizes this objective function. By maximizing an acquisition function (*e.g.,* GP-Hedge [26]), it can determine which $\mathbf{x}$ to investigate next and iteratively converges to the parameter values that optimize the objective function.

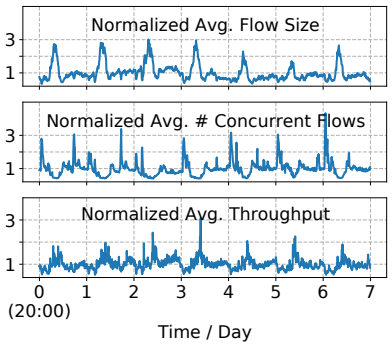


Fig. 3: Daily variations of TCP flow size, # concurrent flows, and throughput normalized by their respective overall mean value from production environments over one week

TABLE III: *Exploration Regions* of the Control parameters.

Control Parameter	Granularity	*Region* $\boldsymbol{\Theta}$	*Region* $\boldsymbol{\Theta}'$
ω	1 packet	[10, 50]	[10, 100]
α	0.01 x	[1, 1.25]	[1, 10]
γ	1 packet	[1, 10]	[1, 100]
λ	0.01 x	[0.5, 0.9]	[0.1, 0.99]
l	0.1%	[0%, 1%]	[0%, 100%]
δ	1%	[1%, 100%]	[1%, 100%]
n	1/4 RTT	[1, 60]	[1, 60]

Since we've observed that in addition to CC algorithms, the performance of TCP is also determined by its own distinct network environments (*e.g.,* region and time of day, as shown in Fig. 2 and Fig. 3, where we called *hidden variables* henceforth). To take hidden variables into the performance optimization, we further introduce Booster based on the design standard BO.

The reason for not applying standard BO to Gemini is twofold. First, in Gemini, directly applying the standard BO will confuse the impacts of hidden variables with that of control parameters on the objective function. To be more specific, if the hidden variables and control parameters have opposite impacts on the objective function, the design of standard BO may mislead the surrogate model to characterize a diametrically opposed relations between control parameter and objective function. Second, applying the BO for each group of flows classified by the hidden variables will increase computation overheads and reduce optimization efficiency, because we need to optimize control parameters for each group of flows and require more time to collect sufficient performance statistics for evaluating objective function robustly.

In Booster, we first employ both control parameters $\boldsymbol{\theta}$ and hidden variables $\mathbf{h}$ to construct the surrogate model to characterize their intrinsic relationship with the objective function. Next, we can get the new values of control parameters $\boldsymbol{\theta}^*$, which maximizes the objective function $f(\boldsymbol{\theta}, \mathbf{h})$ given current hidden variables (denoted by $\mathbf{h}'$ and reported by the optimization agent in Section III-C), *i.e.*,

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}, \boldsymbol{h}=\boldsymbol{h}'} f(\boldsymbol{\theta}, \mathbf{h}) \tag{8}$$

It is worth noting that Booster learns from *historical trials* which are collected from all the working servers running Fusion, so it allows for Booster to directly apply optimization to any new servers running Fusion without scalability issues.

C. The Optimization Agent

As shown in Fig. 1, the agent is introduced to interconnect the parameterized CC class Fusion and the optimization engine running the learning algorithm Booster. Specifically, the agent on each server will periodically (*e.g.,* 5 minutes) aggregate the flow-level statistics (as shown in Table II), the control parameter values used by Fusion and the corresponding hidden variable values and report them to Booster. It is worth noting that only the aggregated statistics (*e.g.,* average throughput, average RTT, flow count, *etc.*) need to be reported, so the cost of data transmission is negligible (about 2MB data need to be transferred from 3K servers every minute, and the average amount of data transferred per server is less than 1KB/minute). Booster will further aggregate these statistics by the control parameter values and the hidden variable values in an incremental manner. At the same time, the agent will also periodically requests new control parameter values with the corresponding hidden variable values and objective function to Booster, then Booster will evaluate the objective function over the aggregated statistics, update its internal surrogate model and generate the new control parameter values to the agent. Once the agent receives the new control parameter updates, it will apply them to Fusion for serving newly incoming TCP flows immediately.

Performance Objective: The agent exposes an interface to operators to specify the performance objective. Previous works (*e.g.,* Vivace and Remy) rely on the form of the objective function to achieve convergence and fairness. However, for Gemini, given control parameter region $\mathbf{\Theta}$ shown in Table III, regardless of the objective form, we can prove if each group of flows with the same hidden variable values uses the same control parameter values, when different flows compete with each other, the corresponding throughput will converge to a unique and fair state under specific assumptions (*e.g.,* single bottleneck and synchronized congestion signal assumptions). Detailed proof is omitted due to space limitation and can be deduced from existing study [27] [28] [29].

It is worth noting that the major congestion control logic design to ensure throughput fairness and convergence are as follows. First, during the congestion avoidance phase, the congestion window increases in both multiplicative and additive ways at the beginning, but only increases additively after the measured BDP stops increasing. Second, the congestion window will be reduced multiplicatively if the congestion signal is triggered.

Without special considerations in convergence/fairness, in this study we set the objective function as the following:

$$f = \frac{1}{N}\sum_{i=1}^{N} Throughput_i - \sigma \cdot Delay_i \tag{9}$$

where N is the number of flows in the report interval; $Throughput_i = size^i/time^i$ is the average throughput of the i_{th} flow; $Delay_i = rtt^i_{avg} - rtt^i_{min}$ is the estimated average queuing delay, where $size^i$, $time^i$, rtt^i_{avg} and rtt^i_{min} are the flow-level statistics reported by the agent; and σ is the preference between throughput and delay. Smaller (larger) σ denotes a preference for higher throughput (lower delay).

IV. Implementation

The parameterized CC class Fusion is implemented as a kernel module in Linux kernel 4.14.105, and we modified scikit-optimize library [30] to support hidden variables for the online optimization engine Booster.

In terms of performance objective, We set three different weights ($\sigma = 0.1$, $\sigma = 1$, and $\sigma = 10$) in Eq. 9 to represent the different preferences between high throughput and low queuing delay. And the *regions* to be explored for the control parameters are set as $\mathbf{\Theta}$ and $\mathbf{\Theta}'$ (See Table III).

(1) *Region* $\mathbf{\Theta}$: The range of each parameter is carefully designed via the tradeoff among adaptability, fairness, friendliness and potential congestion collapse risk in production network based on previous studies [1], [2], [4], [31].

(2) *Region* $\mathbf{\Theta}'$: We set a broader region of control parameters in order to explore the performance limit of Gemini.

V. Performance Evaluation

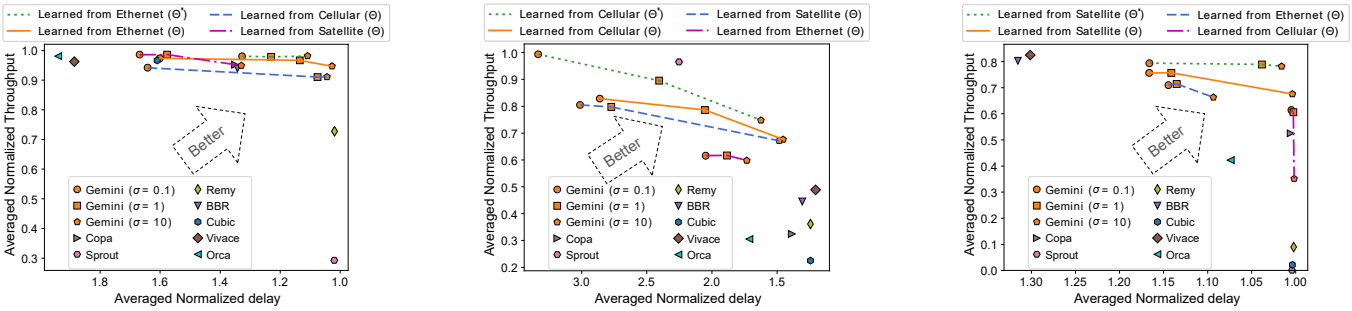
TABLE IV: Details of the three network environments.

Environment	RTT	Bandwidth	Loss rate
Ethernet	150ms	15Mbps	0
Cellular	20ms~250ms	1Kbps~15Mbps	0%~75%
Satellite	800ms	45Mbps	0.74%

In this section, we compare the performance of Gemini with state-of-the-art CC algorithms (*e.g.,* Remy, Vivace, Sprout, Copa and Orca) and widely deployed CC algorithms (Cubic and BBR) over both emulated networks and production networks. We use pantheon [6] to perform our local evaluations and the emulated networks have been covered by the training environments of the prior learning CC algorithms.

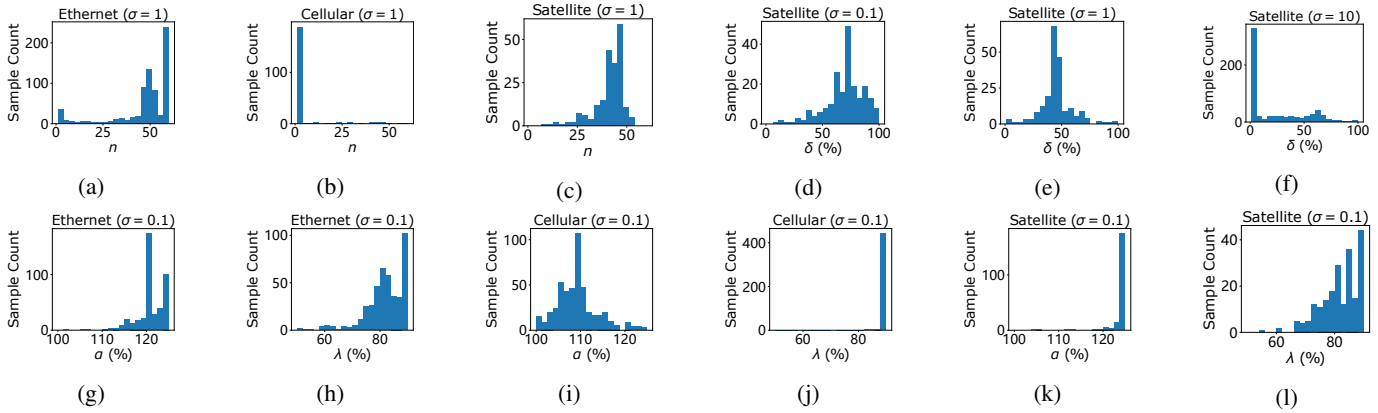
Specifically, under three different emulated network environments (*i.e.* Ethernet [3], cellular [32] and satellite [33], detailed in Table IV), Gemini is able to achieve almost the highest throughput and lowest delay compared with existing algorithm. Moreover, Gemini also exhibits improved adaptability compared to existing learning-based CC algorithms (Remy, Vivace, Orca), despite the significant difference between learning and evaluation network conditions. Finally, our evaluations reveal that Gemini has comparable fairness and friendliness with existing CC algorithms.

Our evaluation over production networks shows that compared with Cubic (BBR), Gemini can achieve around 5% (4%) throughput improvements in a mobile application downloading service, and 61% (33%) improvement in a commercial speed-test benchmarking service. Moreover, the computation and



(a) Ethernet (b) Cellular (c) Satellite

Fig. 4: Performance comparisons in three typical network environments. For each environment, we first evaluate Gemini with three weights ($\sigma = 0.1$, $\sigma = 1$ and $\sigma = 10$). Next, we evaluate Gemini with model learned from different network environments (marked by different line types) to test its adaptivity and robustness. For example, under the Ethernet environment, we evaluate Gemini learned from cellular (blue dashed line), satellite (magenta dash-dot line), and Ethernet (yellow solid line) networks respectively. We also evaluate Gemini learned with a broader space of parameter vector (green dotted line in each sub-figure) in order to explore its performance limits.

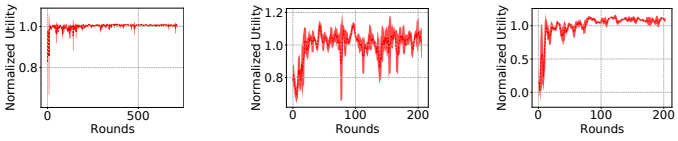


(a) (b) (c) (d) (e) (f)

(g) (h) (i) (j) (k) (l)

Fig. 5: Control parameter distribution (within region $\boldsymbol{\Theta}$) of Gemini

.



(a) Ethernet (b) Cellular (c) Satellite

Fig. 6: Utility convergence of different network environments

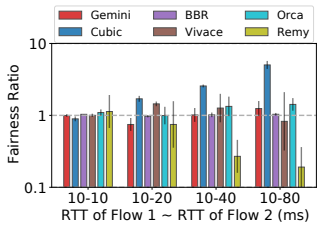
memory overheads of Gemini are comparable with that of traditional CC algorithms and much lower than that of existing learning-based CC algorithms. Next, we compare Booster with existing optimization algorithms, and it reveals Booster achieves a higher performance gains. Finally, the divide-and-conquer design enables us to apply Booster to proprietary TCP for performance optimization and a month-long A/B test results reveal the around 2% throughput improvement achieved by Booster-accelerated proprietary TCP.

A. *Performance Evaluation Over Emulated Networks*

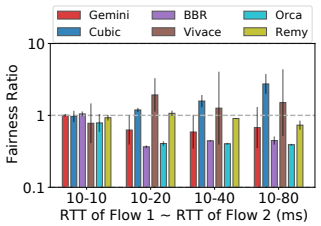
To evaluate the performance of Gemini over different network environments in a controlled manner, we set a linear topology with three machines (the client, router, and server, respectively) and emulate three network environments on the router (Table IV).

1) High Performance in Seen Environments: In Fig. 4a, we can see that Gemini almost achieves the highest throughput for the Ethernet environment (within *regions* $\boldsymbol{\Theta}'$ or $\boldsymbol{\Theta}$), as well as the minimum delay. Although loss-based CC algorithms like Cubic achieve a throughput comparable to Gemini, their delay exceeds that of Gemini, due to their tendency to fill up the queue at the bottleneck. Compared with the rate-based (delay-based) CC algorithms BBR and Copa, Gemini achieves a lower delay as its queuing-delay tolerance is automatically optimized rather than fixed.

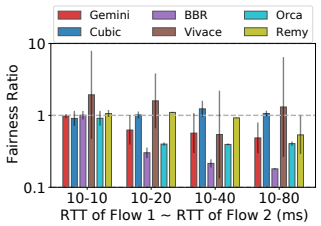
Under the cellular environment (Fig. 4b), the rapidly changing network state (bandwidth, RTT, loss rate) is challenging for CC algorithms to achieve both high throughput and low



(a) 0.5x BDP



(b) 2x BDP



(c) 4x BDP

Fig. 7: RTT fairness comparisons under different link buffer sizes and different path RTTs of competing flows

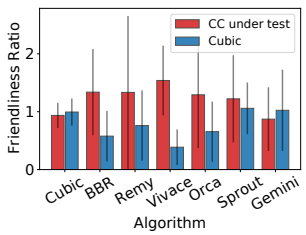


Fig. 8: Cubic friendliness in the random network environments

delay at the same time. Although Gemini is not specifically designed for cellular networks, it achieves the highest throughput (within *region* $\mathbf{\Theta}'$), even compared to Sprout, which is specifically designed for cellular networks. We also observe that the throughput of Orca exceeds that of Cubic. However, the improvement is insignificant as Orca is restricted by the control logic of Cubic. The throughput of Gemini (within *region* $\mathbf{\Theta}$) is slightly lower than that of Sprout. This is attributed to the higher packet loss tolerance of Sprout in the cellular environment, thus it will "steal" more bandwidth from Cubic when competing with Cubic flows for the same bottleneck (Section V-A5).

The satellite environment (Fig. 4c) is also challenging due to the high BDP and stochastic loss. But Gemini (within *regions* $\mathbf{\Theta}'$ and $\mathbf{\Theta}$) outperforms all existing CC algorithms. Although BBR and Vivace exhibit similar throughput with Gemini, their delay is much higher.

2) Robustness in Unseen Environments: In real world deployment, it is common to run CC algorithms over unseen network environments. In order to evaluate the robustness of Gemini, we first train Gemini in one environment, then evaluate Gemini with another two different environments.

We observe that although there is a clear mismatch between the testing and training environments, the performance of Gemini still surpasses that of most existing traditional CC and learning-based CC algorithms. This is because although Booster has adapted the control parameter values of Fusion in training environments, *i.e.* the degree of reaction to specific events during data transfer, Fusion still maintains the same full control logic as traditional algorithms, which maintains its robustness to unseen environments.

3) Convergence of Control Parameters: Since under iterative *decision* and *evaluation* process cycle, Gemini periodically update new control parameters for Fusion, we collect such control parameters and corresponding utility conference in this section, and summarize our key findings as follows.

First, compared to Ethernet and satellite environments (Fig. 5a and Fig. 5c), we find Gemini in cellular environment tends to converge to smaller sliding window n for throughput and RTT sampling (Fig. 5b). This is because the bandwidth in the cellular environment is more dynamic and longer-ago throughput samples are less correlated with the future samples, so closely tracking the most recent throughput can result in higher performance.

Next, since in Gemini the performance objective can trade off between throughput and delay preferences by controlling the value of σ in Eq. 9. Accordingly, we observe that different preferences to delay finally result in different convergence value of RTT inflation threshold δ (Fig. 5d, Fig. 5e and Fig. 5f), *i.e.* higher preference to delay results in a lower RTT inflation threshold and vice versa.

Moreover, we observe that λ (*i.e.* the multiplicative decrease factor in Fig. 5h, Fig. 5j and Fig. 5l) and α (*i.e.* the multiplicative increase factor in Fig. 5g, Fig. 5i and Fig. 5k) in Gemini usually converges to their respective upper bound (*i.e.* 0.9 and 1.25 in region $\mathbf{\Theta}$), this may indicate that by expanding the parameter in Region $\mathbf{\Theta}$, Gemini can achieve higher performance, which also motivate us to expand the parameter region from $\mathbf{\Theta}$ to $\mathbf{\Theta}'$ to explore its performance limit.

Finally, we show the utility (performance objective) convergence behavior of Gemini in Fig. 6. We can see that at the beginning they start at a lower utility, which is expected since the parameters are randomly initialized at the beginning, and after several iterations, they gradually converge to a higher and stable utility. We also observe that because of the higher bandwidth dynamics in cellular network, Gemini in cellular network takes more iterations to converge to a comparatively stable utility and the converged utility is more fluctuated compared with the behavior of Gemini in the Ethernet and the satellite networks.

4) RTT Fairness: In this section, we set up two flows competing on a single bottleneck link, where the path RTT of the first flow is fixed to 10ms, the path RTT of the second flow varies from 10 to 80ms, the bottleneck bandwidth is set to 100Mbps, and the queue length at the bottleneck varies from 0.5x BDP to 4x BDP (1x BDP = 10ms $\times$ 100Mbps).

The *fairness ratio* is employed as the fairness metric [12]:

$$FairnessRatio = AvgTpt_{base}/AvgTpt_x \qquad (10)$$

where $AvgTpt_x$ is the average throughput of the flow with path RTT x (x =10, 20, 40 and 80ms) and $AvgTpt_{base}$ is the average throughput of the flow with based path RTT (set to be 10ms in this experiment).

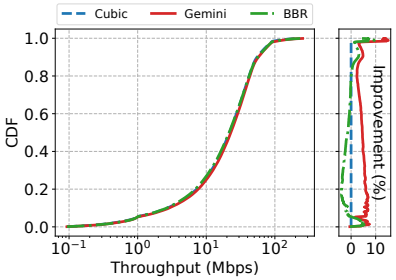


Fig. 9: Throughput comparisons over a mobile application downloading service

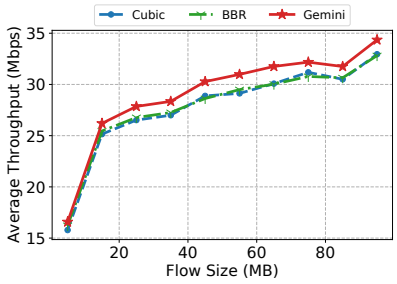


Fig. 10: Throughput comparisons of different flow size over a mobile application downloading service

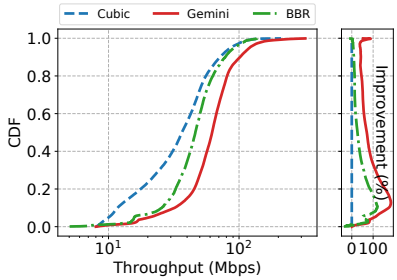


Fig. 11: Throughput comparisons over a commercial speed-test benchmarking service

We show the mean and standard deviation of the fairness ratio for more than 100 repeated experiments in Fig. 7. For each run of Gemini, we randomly sample control parameters $\boldsymbol{\theta}$ from the *region* $\boldsymbol{\Theta}$. We observe that existing CC algorithms exhibit different degrees of RTT fairness. For example, the flow with the shorter path RTT has a larger bandwidth share for Cubic; Orca and BBR are relatively fair at shallow-buffered queues, yet the flow with the longer path RTT has a larger bandwidth share at deep-buffered queues; Remy exhibits a larger bandwidth share for the flow with longer path RTT at shallow-buffered queues. The fairness-ratio variance of Vivace and Remy is much larger than existing CC algorithms, indicating that the converged bandwidth share of the two competing flows varies greatly in the repeated experiments.

By contrast, due to the introduction of both multiplicative-increase factors like BBR (which introduces an additional bandwidth share for long-RTT flows) and additive-increase factor like Reno (which introduces an additional bandwidth share for short-RTT flows), Gemini achieves a medium level of RTT fairness among existing algorithms.

5) Cubic Friendliness: We evaluate Cubic friendliness of Gemini with the same emulated network setting as [12], *i.e.* the bandwidth and path RTT are randomly sampled from 1 to 50Mbps and 2 to 100ms respectively, the queue length on the bottleneck is set between 0.5x and 5x BDP. For each run, we calculate the ratio of its actual bandwidth share to its ideal fair share:

$$FriendlinessRatio = AvgTpt/IdealTpt \tag{11}$$

where $AvgTpt$ is the average throughput of the flows using a specific CC algorithm and $IdealTpt$ is the ideal fair share of each flow competing on the same bottleneck.

We summarize the results in Fig. 8 based on 100 repeated experiments. It is clear that Gemini exhibits a greater level of friendliness with Cubic compared to existing learning-based CC algorithms. By contrast, Vivace is more aggressive than BBR when they both compete with Cubic, while Remy exhibits an unstable performance when competing with Cubic flows across different network environments. Moreover, although the underlying Cubic in Orca can generally control the sending rate, the learning of the window scaling factor does not consider friendliness, so it is not surprising to see the unfriendliness of Orca compared to Cubic.

B. Performance Evaluation Over Production Networks

For evaluations over production networks, we set the performance objective of Gemini as Eq. 9 with the weight factor $\sigma = 0.1$ and *region* $\boldsymbol{\Theta}$ is used as the exploration region of control parameters.

1) Gemini versus Cubic and BBR: We evaluate Gemini in the following two different services. The first one is a mobile application downloading service in a top CDN service provider in China. The flow size in this service varies from Kilobytes to Gigabytes, with over 90% flow size ranges between 1MB and 100MB. Over a 96-hour period of experiment and more than three million requests, Gemini (with an average throughput of 28.34Mbps) achieves around 5% mean throughput improvement over Cubic (with an average throughput of 27.06Mbps) and 4% average throughput improvement over BBR (with an average throughput of 27.22Mbps). We summarize the throughput distribution for each CC algorithm and compare their respective percentiles in Fig. 9. We can see that the throughput of Gemini is generally higher than that of Cubic and BBR for each percentile. Moreover, we observe that though BBR achieves more throughput gains under lower and higher percentiles compared with Cubic, it under-performs in the middle percentiles (*e.g.,* from the 10th to 80th percentile), which may indicate that BBR does not always perform better than Cubic in real-world deployments.

Fig. 10 further investigates the performance of Gemini for different flow sizes. We observe that Gemini outperforms Cubic and BBR regardless of the flow size. Though BBR exhibits advantages over Cubic for small flows, Cubic has similar throughput with BBR on average for large flows.

The second one is a commercial speed-test benchmarking service with thousands of nodes located across nine provinces in China. The nodes in the platform run in a round-robin manner and periodically request a 1MB file from the servers deploying Cubic, BBR and Gemini. In this experiment, the total number of requests for each algorithm is around 30K. We can see from Fig. 11 that Gemini generally outperforms Cubic, BBR for each percentile of the throughput distribution. And

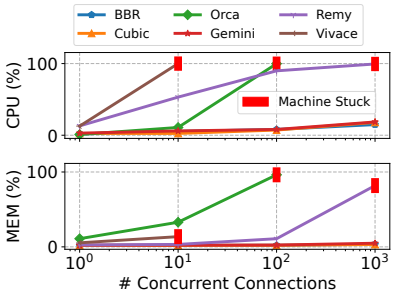


Fig. 12: CPU/Memory consumption of different CC algorithms

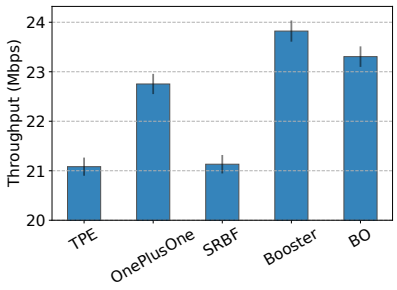


Fig. 13: Throughput comparisons over different learning algorithms

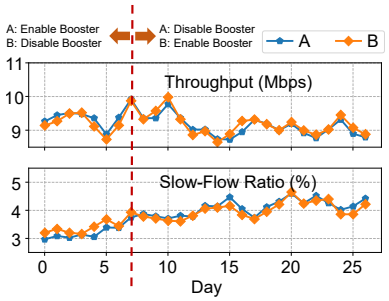


Fig. 14: Performance comparison between proprietary TCP and Booster-accelerated proprietary TCP

we also summarize that Gemini (with an average throughput of 65.85Mbps) improves the average throughput by around 61% for Cubic (with an average throughput of 39.96Mbps) and by around 33% over BBR (with an average throughput of 49.07Mbps). Moreover, we observe that the advantages of BBR and Gemini over Cubic are reduced as the throughput increases, indicating that Gemini and BBR can boost the throughput of flows with poor network environments while the three perform similarly with better network environments.

2) CPU Overhead Comparison: In addition to transmission performance, memory and computation consumption are another two major concerns for the deployment in production networks [7]. Since existing learning-based algorithms cannot be directly deployed over production services and for the sake of performance benchmarking, we further compare the CPU and memory usage of existing algorithms using a virtual machine (8-core, 16-thread 3.6GHZ CPUs and 8GB RAM) with 1Gbps link capacity. We vary the number of concurrent flows and summarize the corresponding CPU and Memory consumption in Fig. 12. It is clear that the computation and memory overhead of Gemini is comparable with Cubic and BBR while existing learning-based CC algorithms suffer order of magnitude higher consumption of CPU and memory.

3) Booster versus existing black-box optimization algorithms: We also compared the performance of Booster with existing black-box optimization algorithms in the production network over another 192-hour period experiment (Fig. 13). We find Booster achieves around 3% higher average throughput than standard BO. This confirms the advantage of Booster due to its ability to incorporate the impact of hidden variables on the performance objective. Moreover, our further investigation reveals that the TPE, OnePlusOne, and SRBF usually fail to converge to stable control parameters in production networks, which indicates further optimization is needed for the applications of these algorithms to production networks.

4) Apply Booster to Proprietary TCP: The divide-and-conquer design of Gemini enables its applications to any other TCP (CC) for performance optimization. In this section, we further apply Booster to a proprietary TCP used by a top CDN service provider for a mobile application downloading service. Specifically, we conducted a month-long A/B test between the proprietary TCP and Booster-accelerated proprietary TCP with two groups (denoted by A and B in Fig. 14) of servers across China. At the end of the A/B test, over 20 million requests are served by each group. We summarize the daily average throughput of each group and its corresponding ratio of slow flows (defined as the ratio of the number of flows with throughput below 100KB/s over the total number of flows) in Fig. 14. Though the load balancer enables an equal share of requests for each group, we switch the CC for each group on day 7 to eliminate the potential system bias in production networks. From the figure, we can see Booster-accelerated proprietary TCP consistently outperforms proprietary TCP with higher throughput (2% improvement on average) and a lower ratio of slow flows (3.5% improvement on average).

VI. Conclusion And Future Work

In this study, we proposed a practical learning-based Internet congestion control algorithm Gemini and deployed it in production networks successfully. Extensive evaluations over both emulated and production networks show that Gemini can significantly outperform existing learning-based algorithms in fairness, friendliness, and CPU/Memory overhead and outperform traditional algorithms in transmission performance. Applying Gemini to more Internet applications (*e.g.,* video streaming and video communications) and more different network environments (*e.g.,* datacenter network) will be conducted in the future works.

Acknowledgment

This research is supported by the Key-Area Research and Development Program of Guangdong Province under Grant number 2020B0101390001, the National Natural Science Foundation of China under Grant Numbers 92067206 and 62072228.

References

[1] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

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

[3] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 123–134, 2013.

[4] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "{PCC}: Re-architecting congestion control for consistent high performance," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 395–408.

[5] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "{PCC} vivace: Online-learning congestion control," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 343–356.

[6] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 731–743.

[7] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: a pragmatic learning-based congestion control for the internet," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 632–647.

[8] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] "Bonree," https://www.bonree.com/bonree/product/server.html, 2022, [Online; accessed 31-July-2022].

[11] 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*, vol. 13, no. 8, pp. 1465–1480, 1995.

[12] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 329–342.

[13] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 459–471.

[14] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 509–522.

[15] H. Jiang, Z. Liu, Y. Wang, K. Lee, and I. Rhee, "Understanding bufferbloat in cellular networks," in *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, 2012, pp. 1–6.

[16] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of bbr congestion control," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 2017, pp. 1–10.

[17] U. Naseer and T. A. Benson, "Configanator: A data-driven approach to improving {CDN} performance." in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1135–1158.

[18] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," in *Proceedings of the conference on Communications architectures, protocols and applications*, 1994, pp. 24–35.

[19] S. Ha and I. Rhee, "Hybrid slow start for high-bandwidth and long-distance networks," in *Proc. PFLDnet*, 2008, pp. 1–6.

[20] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi, "Proportional rate reduction for tcp," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, 2011, pp. 155–170.

[21] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing tcp's initial congestion window," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 3, pp. 26–33, 2010.

[22] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006, vol. 2, no. 3.

[23] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *25th annual conference on neural information processing systems (NIPS 2011)*, vol. 24. Neural Information Processing Systems Foundation, 2011.

[24] R. G. Regis and C. A. Shoemaker, "A stochastic radial basis function method for the global optimization of expensive functions," *INFORMS Journal on Computing*, vol. 19, no. 4, pp. 497–509, 2007.

[25] J. Rapin and O. Teytaud. (2018) Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad.

[26] M. D. Hoffman, E. Brochu, and N. de Freitas, "Portfolio allocation for bayesian optimization." in *UAI*. Citeseer, 2011, pp. 327–336.

[27] Y. R. Yang and S. S. Lam, "General aimd congestion control," in *Proceedings 2000 International Conference on Network Protocols*. IEEE, 2000, pp. 187–198.

[28] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.

[29] S. Varma, *Internet congestion control*. Morgan Kaufmann, 2015.

[30] T. Head, G. L. MechCoder, I. Shcherbatyi *et al.*, "scikit-optimize/scikit-optimize: v0. 5.2," *25th Mar*, 2018.

[31] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 183–196.

[32] "A script based on Linux TC netem to emulate the latency, loss, and bandwidth of a real-world cellular network," *https://github.com/akamai/cell-emulation-util*.

[33] H. Obata, K. Tamehiro, and K. Ishida, "Experimental evaluation of tcp-star for satellite internet over winds," in *2011 Tenth International Symposium on Autonomous Decentralized Systems*. IEEE, 2011, pp. 605–610.