Multi-Objective Congestion Control

Yiqing Ma[1], Han Tian[1], Xudong Liao[1], Junxue Zhang[1], Weiyan Wang[1], Kai Chen[1], Xin Jin[2]
[1]iSING Lab, Hong Kong University of Science and Technology, [2]Peking University

Abstract

Decades of research on Internet congestion control (CC) have produced a plethora of algorithms that optimize for *different* performance objectives. Applications face the challenge of choosing the most suitable algorithm based on their needs, and it takes tremendous efforts and expertise to customize CC algorithms when new demands emerge. In this paper, we explore a basic question: can we design a *single* CC algorithm to satisfy *different* objectives?

We propose MOCC, the *first* multi-objective congestion control algorithm that attempts to address this question. The core of MOCC is a novel *multi-objective reinforcement learning* framework for CC to automatically learn the correlations between different application requirements and the corresponding optimal control policies. Under this framework, MOCC further applies *transfer learning* to transfer the knowledge from past experience to new applications, quickly adapting itself to a new objective even if it is *unforeseen*. We provide both user-space and kernel-space implementation of MOCC. Real-world Internet experiments and extensive simulations show that MOCC supports well multi-objective, competing or outperforming the best existing CC algorithms on each individual objectives, and quickly adapting to new application objectives in 288 seconds (14.2× faster than prior work) without compromising old ones.

CCS Concepts • **Networks** → **Network protocols**; **Transport protocols**; *Network protocol design.*

Keywords congestion control, reinforcement learning, multi-objective

ACM Reference Format:
Yiqing Ma[1], Han Tian[1], Xudong Liao[1], Junxue Zhang[1], Weiyan Wang[1], Kai Chen[1], Xin Jin[2]. 2022. Multi-Objective Congestion Control. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519593

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519593

1 Introduction

Congestion control (CC) is a fundamental, enduring topic in networking research. Decades of study on this topic have produced a plethora of CC algorithms [1, 4, 5, 7, 12, 13, 15, 21, 26, 27, 49, 53, 61, 62]. These algorithms are motivated by new applications that impose different demands on network performance, as well as new technologies that change the underlying Internet infrastructure. The confluence of these two factors requires a CC algorithm to be deliberately designed to optimize for a particular performance objective.

Consequently, applications face the challenge of choosing the most suitable CC algorithm based on their needs. This choice is definitely not easy given the wide range of options, and the subtle differences between them that oftentimes require deep understanding of TCP minutiae. At the same time, whenever new applications with different demands emerge, it takes tremendous efforts and expertise to customize CC algorithms for their new requirements.

In this paper, we explore a basic question: can we design a *single* CC algorithm to satisfy *different* objectives? Traditional CC algorithms [4, 5, 7, 15, 21, 27, 53] are *hand-crafted*. They rely on certain assumptions about the network, and hardwire packet-level events to pre-defined control rules based on human experience. Recent learning-based CC algorithms [1, 26] relieve the burden by applying deep reinforcement learning to automatically *learn* an optimal control policy for a given objective. Yet, satisfying different objectives requires us to maintain one copy for each algorithm (either traditional or learning-based), and pay excessive time to design or train an algorithm each time when a new application with a different objective arrives (§2).

We propose MOCC, the *first* multi-objective CC algorithm that attempts to address this question (§3). The core of MOCC is a novel *multi-objective reinforcement learning* framework for CC to automatically learn the correlations between different application requirements and their corresponding optimal control policies. MOCC explicitly incorporates the performance objective into both the state input and the dynamic reward function, and leverages a new policy neural network with *a preference sub-network* to correlate different objectives with optimal control policies (§4.1). This allows MOCC to effectively establish *a single correlation model* to support different performance objectives. Under this framework, MOCC further applies *transfer learning* to quickly transfer the knowledge learned from past experience to new applications, and optimizes the CC algorithm for a given objective, even if it is *unforeseen*.

MOCC achieves its goal by a combination of offline training (§4.2) and online adaptation (§4.3). In offline training, MOCC is trained over a set of well-distributed landmark objectives to learn the base correlations between application requirements and optimal policies. Then, whenever a new application arrives, MOCC can *immediately* provide a moderate policy using the offline trained model by correlating the application's objective with the landmark objectives, even if it is unforeseen. Meanwhile, MOCC activates online adaptation to transfer the knowledge from the base correlation model to the new application. With transfer learning, MOCC can quickly converge to the optimal policy within just a few training iterations, orders of magnitude faster than training from scratch. In addition, to avoid forgetting the learned policies, we customize the loss function of MOCC online adaptation for both new arrival and old (sampled) applications. This enables MOCC to learn and apply optimal policies for new applications without compromising old ones.

We fully implement MOCC (§5) with a user-space implementation based on UDT [20] and a kernel-space implementation based on CCP [41]. We leverage OpenAI Gym [6] and Aurora [26] to implement the training and adaptation components, and use parallel training to reduce training time. For better portability, we encapsulate all MOCC's functions into one library that is plug-and-play and readily deployable with any networking data paths that include, but not limited to, our user-space and kernel-space implementations.

We evaluate MOCC with extensive simulations and real-world Internet experiments (§6). We show that MOCC well supports multiple objectives, competing or outperforming the best existing CC algorithms (including both traditional ones and recent learning-based ones) on individual objectives (§6.1), and can quickly adapt to new application objectives in 288 seconds, 14.2× faster than prior solution (§6.2). We further demonstrate the benefits of MOCC with three real Internet applications in §6.3, and inspect the fairness and friendliness of MOCC in §6.4. Finally, we deep-dive into various design choices of MOCC and its overhead in §6.5.

2 Background and Motivation

2.1 Diverse Application Requirements

Internet applications have diverse performance requirements for the network, typically characterized by metrics such as throughput, latency, jitter, and packet loss rate [10, 14, 18, 28, 29, 43, 48, 52]. Throughput is the main metric for many applications, in which minimum bandwidth is required to provide good user experience, e.g., HDTV requires (>34Mbps) to play high-definition video without rebuffering [10]. On the other hand, real-time interactive applications usually require low latency, e.g., autonomous driving requires low latency (<15ms) to react to immediate environment signals [11]. For some real-time applications, temporal packet loss is also important, e.g., online video/audio conferencing can only tolerate (<0.1%/1%) packet loss rate [16]. Emerging Internet applications such as augmented/virtual reality may have tight requirements on several metrics simultaneously [37].

Algorithm	Objective
PCC Allegro [12]	$T - \delta RTT$
PCC Vivace [13]	$T^t - b \times \frac{d(RTT)}{dt} - c \times L$
Aurora [26]	$\alpha T - \beta RTT - \gamma L$
Orca [1]	$\frac{T-\varepsilon L}{RTT} / (\frac{T_{max}}{RTT_{min}})$

Table 1. Performance objectives in learning-based CC. T is throughput, RTT is latency, and L is loss rate.

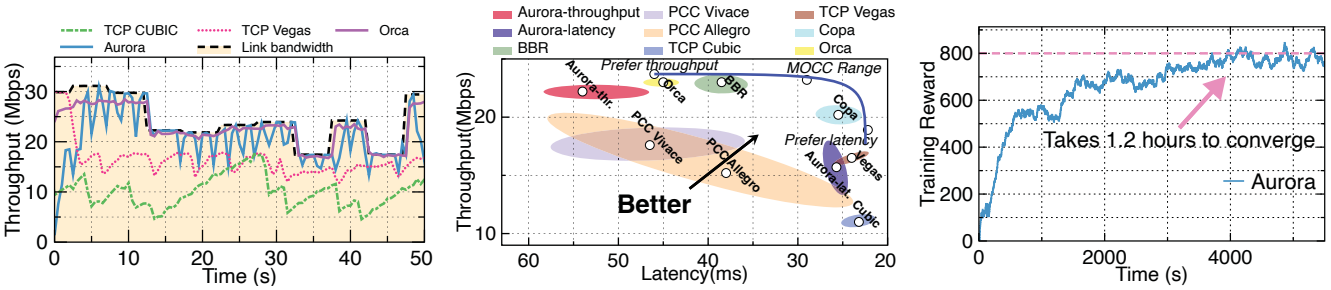
To summarize, these application demands pose different requirements on Internet CC algorithms. Ideally, the CC algorithm should be *multi-objective* to support diverse application requirements simultaneously. However, as we will show subsequently (§2.2), none of existing CC solutions can do this.

2.2 In Pursuit of Multi-Objective CC

We broadly classify existing Internet CC algorithms into two main categories: hand-crafted [4, 5, 7, 15, 21, 27, 53, 61] and learning-based [1, 12, 13, 26]. Traditional CC algorithms hardwire packet-level events to pre-defined control rules based on human experience. The performance objective is implicitly encoded in the mapping, and in many cases, it is hard to infer what is exactly being optimized.

Existing learning-based CC can optimize for a given objective. Recent learning-based CC algorithms can address the above problem of hand-crafted heuristics by explicitly encoding the performance objectives in the reward/utility function and maximizing it through machine learning from network environments. Table 1 lists several reward/utility functions used by state-of-the-art learning-based CC algorithms.The reward function is typically expressed as a combination of metrics such as throughput, latency and loss rate. The coefficient parameters $(\alpha, \beta, \gamma, \delta, \varepsilon, b, c)$ can express the relative importance of these metrics based on application requirements explicitly. Thus, learning-based CC is able to perform well for a particular objective.

We use a simple simulation to showcase this. The setup follows that in Orca [1]. Specifically, we simulate a network in which the one-way delay is 20 ms, the bottleneck link bandwidth varies between 20–30Mbps, and the loss ratio is 0.02%. We compare two traditional CC algorithms (TCP CUBIC and Vegas) and two learning-based CC algorithms (Aurora and Orca). As shown in Figure 1(a), CUBIC and Vegas under-utilize the bandwidth and do not perform well when the link bandwidth changes. In contrast, Aurora and Orca are trained by assigning high weight to throughput in the reward function. As a result, they achieve higher throughput than



(a) Support for throughput-intensive application (b) Support for different objectives (c) Time cost for re-training a new objective

Figure 1. (a) Learning-based CC algorithms can explicitly optimize for a particular objective, and perform better than traditional CC algorithms. (b) Existing learning-based CC cannot support multiple objectives. (c) Existing learning-based CC such as Aurora takes a long time to re-train the model when the objective changes.

CUBIC and Vegas, and the throughput benefits are consistent under changing network conditions.

But, these learning-based CC cannot support multiple objectives. As listed in Table 1, the current learning-based algorithms set relative importance of throughput, latency and loss rate to realize different performance objectives. These coefficient parameters are fixed during training. Thus the trained agent can only optimize for a particular objective at a time. To show this, we reuse the above simulated network setting to evaluate different CC algorithms (Aurora, PCC-Allegro, PCC-Vivace, BBR, Cubic, Vegas, Copa). For Aurora, we apply two models, one trained for throughput (Aurora-throughput) and the other trained for latency (Aurora-latency). We present throughput-delay plot for each CC in Figure 1(b). We take each individual 60-second run as one point, and then compute the $1-\sigma$ elliptic contour of the maximum-likelihood 2D Gaussian distribution that explains the points (refer to Remy [57]). Thus Figure 1(b) shows the throughput-delay performance range for each CC schemes. As shown in the figure, from right to left and bottom to top, these algorithms trace out a path from most latency-optimized to most throughput-optimized. Aurora-throughput provides higher throughput, while Aurora-latency provides lower latency. But each of them can only optimize for one particular objective. In comparison, we propose MOCC, a multi-objective CC that can support different application requirements. The expected ideal performance of MOCC is shown in the blue line. By dynamically adjusting the relative importance of its reward function, MOCC is expected to accommodate different objectives.

For learning-based approaches, hypothetically, one can train a custom model for each performance objective. However, this is undesirable: given the diverse application requirements and with new applications emerging every year, it is hard to cover all performance objectives. And even if possible, one may need to install a copy of algorithm in each device and train the algorithm in real time when the performance objective changes. However, existing learning-based CC algorithms are not *quick-adaptive*. As an example, Figure 1(c) shows that re-training the model of Aurora [26] for a new objective takes more than one hour to converge. Besides these drawbacks, from a scientific point of view, we would like to explore *whether it is possible to design a single CC algorithm to satisfy multiple objectives.*

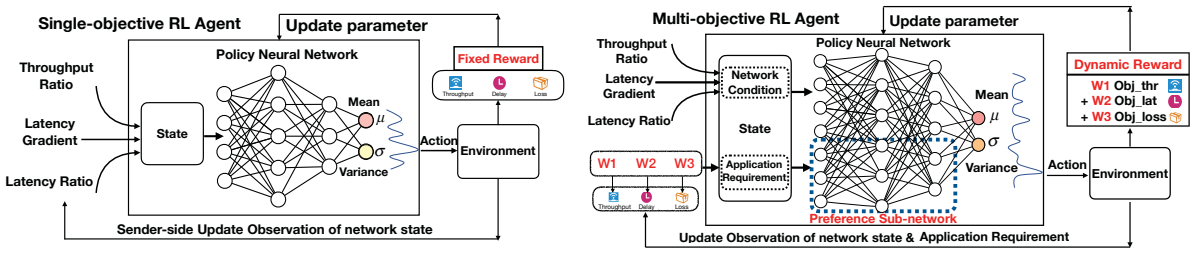
QUIC: QUIC [31] is a user-space transport protocol on top of UDP to improve the transport performance of Internet applications and to enable application-specific customizations. QUIC itself, however, is not tied to a particular CC algorithm. It only provides the *mechanism* to implement application-specific CC algorithms, and an application still needs to specify which CC it uses, which can be either a traditional or learning-based algorithm, to achieve its performance objective. As such, our work is orthogonal to QUIC, and more importantly, we show that we only need a single CC to satisfy different objectives.

2.3 Design Goals

We seek *one single CC algorithm* satisfying all the following three goals simultaneously.

- **Multi-objective**: The algorithm can support different applications with different performance objectives, and provide optimal control policies for individual applications.
- **Quick-adaptive**:The algorithm can quickly adapt to new applications with unforeseen requirements, without compromising performance of old applications.
- **Consistent high-performance**: The algorithm should achieve high-performance in various network conditions without any pre-assumption.

We are inspired by recent trend (notably, Aurora [26] and Orca [1]) to adopt RL for CC, which can readily achieve consistent high-performance. However, the challenge is how



(a) Single-objective CC (Aurora [26])

(b) Multi-objective CC (MOCC)

Figure 2. From single-objective CC to multi-objective CC: Incorporating preference sub-network into MOCC with application requirements explicitly used in both state input and dynamic reward function enables MOCC to learn (and memorize) the correlations between application requirements and the corresponding optimal rate control policies, thus realizing multi-objective, i.e., one single MOCC model can support multiple applications.

to simultaneously support multiple application objectives and quickly accommodate new ones.

3 Multi-Objective Learning for CC

We formulate CC as a sequential decision-making problem under the RL framework. Consider a general RL setting where an agent interacts with an environment. At each time step t, the agent observes some state s_t, and chooses an action a_t. After applying the action, the state of the environment transits from s_t to s_{t+1} and the agent receives a reward r_t. The state transitions and rewards are stochastic and Markovian [56]. The goal of RL learning is to maximize the expected cumulative discounted reward $E[\Sigma_{t=0}^{\infty}\gamma^t r_t]$, where $\gamma \in (0, 1]$ is a factor discounting future rewards.

Figure 2(a) describes the standard way how to apply RL for CC, which reflects the state-of-the-art work Aurora/Orca [1, 26]. Basically, in each time interval, the agent (i.e., sender) observes a set of network metrics such as throughput, latency, and packet-level events, etc., and feeds these values into the neural network, which outputs the action, i.e., the sending rate for the next interval. In the meanwhile, the resulting network performance (e.g., throughput and latency) is measured and passed back to the agent as a reward, which will be used to train and improve the neural network model.

While the above standard RL shows promise, it has a key shortcoming: the algorithm can only optimize for a *single objective* at a time. The crux is that the model (Figure 2a) has no way to recognize and differentiate among multiple different applications. As a result, supporting multiple applications requires multiple different models, and furthermore, adapting to a new application entails retraining the model from scratch which takes time, making it neither multi-objective nor quick-adaptive.

We seek one algorithm to simultaneously support multiple application objectives while quickly adapting to new arrival ones. To this end, we extend the existing single-objective RL approach and establish a multi-objective RL framework for CC (MOCC) that meets all our design goals stated in §2.3.

As a fast-growing research area, MORL(multi-objective reinforcement learning) is a generalized RL framework used for solving multi-objective Markov decision process (MOMDP). MOMDP extends Markov decision process (MDP) by incorporating multiple optimization objectives. A MOMDP can be formalized by the tuple $\langle S, A, P, \vec{r}, \Omega, f_\Omega \rangle$. S and A are the state space and action space. $P(s' \in S | s \in S, a \in A)$ defines the state transition probability. $\vec{r}(s, a)$ consists a vector of reward functions w.r.t. each objective respectively. Ω defines the preference space and $f_\Omega(\vec{r})$ is the preference function producing the integrated reward value with given preference and collected objective-specific rewards. With Ω fixed to a single preference, a MOMDP degrades to a standard MDP and can be solved with single-objective RL algorithm. Under the linear preference function $f_\Omega(\vec{r}) = \vec{w}^T\vec{r}$ in MOCC, the optimal policy set for our MOMDP is called a convex coverage set (CCS), which contains all optimal polices for any given preference $\vec{w}$. The learning goal is to recover the entire CCS, the optimal policy set for all possible application requirements, and apply corresponding one for any given application requirement w.

By contrasting Figure 2(a), our MOCC framework in Figure 2(b) illustrates how it works. From model structure perspective, we make two important changes: (1) we expand the policy neural network by incorporating a *preference sub-network* that explicitly takes application requirements, each denoted by a weight vector of performance metrics, as state input, making our model aware of different objectives in addition to network conditions[1]; and (2) we dynamically parameterize the reward function with the weight vector of the application currently under training, which enables our model to learn the optimal policy for the corresponding objective. As a result, MOCC automatically learns the correlations between application requirements and the corresponding

[1]Researchers have adopted feature vectors to represent multi-user requirements in Video Streaming Dash Approach and achieved significantly better personalized QoE [19, 24].

optimal rate control policies, thus achieving multi-objective (more details in §4.1).

We train our MOCC model through offline pre-training (§4.2) and online adaptation (§4.3). In particular, we leverage transfer learning techniques [3, 9, 32, 45, 54] to speedup offline pre-training as well as adapting to new applications in an online manner. In the offline phase, we pre-train our model with a well-distributed set of landmark weight vectors to learn the correlations between application requirements and optimal policies. This brings two important benefits to the online phase. First, for a new application, MOCC can *immediately* provide a reasonable policy even it is unforeseen, maintaining performance during the transition. Second, transferring from such base correlation model, MOCC is able to quickly converge to the optimal policy for the new application with just a few RL iterations, much faster than learning from scratch (e.g., 14.2× in our evaluation §6). Furthermore, to avoid forgetting the already learned policies for old applications, we modify the loss function of the online learning by optimizing for both the new arrival and sampled history applications, so that our MOCC can recall the learned policies for previous applications.

To summarize, by deliberately architecting and training the model as above, our MOCC framework is able to learn, remember, and apply optimal rate control policies for multiple applications simultaneously while adapting to new ones on-the-fly. In §4 below, we will go deeper into design details.

4 Design

We start by introducing the model architecture that enables MOCC to achieve the multi-objective property (§4.1). Then, we describe our offline training (§4.2) and online adaptation (§4.3) that can quickly adapt MOCC to new applications.

4.1 Model Architecture

To enable multi-objective, MOCC makes two main changes upon the standard RL-based CC: 1) incorporating a preference sub-network into the policy network, and 2) including application requirements in both state input and dynamic reward function. In this way, MOCC can establish the correlations between various application requirements and the corresponding optimal rate control policies.

States: State inputs to MOCC include both application requirements and network conditions. To express application requirements, we use weight vector $\vec{w} = <w_{thr}, w_{lat}, w_{loss}>$ which contains the relative weights of three main performance metrics[2] in CC algorithm: throughput, latency, and packet loss rate. The range of each weight $w_i \in (0, 1)$ and $\Sigma_i w_i = 1$. For example, $<0.8, 0.1, 0.1>$ means that the application desires high throughput, and $<0.4, 0.5, 0.1>$ indicates the application is latency-sensitive but still needs certain throughput.

For network conditions, similar to prior work [13, 26, 49], we use statistics vector $\vec{g}_t = <l_t, p_t, q_t>$ to express the network status at time interval t. Specifically, l_t is sending ratio, defined as packets sent by sender over packets acknowledged by receiver; p_t is latency ratio, the ratio of mean latency of the current time interval t to the minimum observed mean latency in the history; and q_t is latency gradient, the derivative of latency with respect to time. Furthermore, to capture the trends and changes of network dynamics, we use a fixed-length history of network statistics instead of the most recent one (i.e., $\vec{g}_{(t,\eta)} = <\vec{g}_{t-\eta}, \vec{g}_{t-\eta+1}, ..., \vec{g}_t>$ with length $\eta > 0$) as network state input. This improves MOCC by reacting to network dynamics more appropriately [26].

Actions: Upon observing state $s_t = (\vec{w}, \vec{g}_{(t,\eta)})$, the RL agent chooses an action a_t. Then the MOCC sender takes the output a_t to change its sending rate from x_t to x_{t+1} for the next time interval $t + 1$ as follows:

$$x_t = \begin{cases} x_{t-1} * (1 + \alpha a_t) & a_t > 0 \\ x_{t-1}/(1 - \alpha a_t) & a_t < 0 \end{cases} \quad (1)$$

Here α is a scaling factor used to dampen oscillations. Instead of discrete sending rate adjustment, we choose a continuous sending rate adjustment to improve model robustness and achieve faster convergence.

Rewards: The MOCC reward function r_t is dynamically parameterized with the weight vector $\vec{w}$ of application under training, so that the RL agent can capture the requirement of the application. Specifically,

$$Reward: \ r_t = w_{thr} * O_{thr} + w_{lat} * O_{lat} + w_{loss} * O_{loss} \quad (2)$$

in which $O_{thr} = \frac{\text{Measured Throughput}}{\text{Link Capacity}}$, $O_{lat} = \frac{\text{Base Link Latency}}{\text{Measured Latency}}$, and $O_{loss} = 1 - \frac{\text{Lost Packets}}{\text{Total Packets}}$ are three performance measures on throughput, latency and packet loss rate. They are configured to positively relate to the final reward, and normalized to $[0, 1]$ to ensure fairness among each other. We use measured maximum throughput and minimum delay to estimate the Link Capacity and Base Link Latency in the online phase. We note that the estimated normalized statistics are not required during the inference.

Model structure: MOCC adopts the actor-critic method [50], a basic approach to train policy network in RL. The actor-critic method uses two neural networks: the actor network and the critic network (Figure 3). The actor network is used to represent the policy π_θ that maps application requirements and network conditions to action distribution π_θ: $\pi_\theta(\vec{g}_{(t,\eta)}, \vec{w}, a_t) \rightarrow [0, 1]$, where θ represents the adjustable model parameters. The critic network is to evaluate the results of actor network during training by the output value $V^{\pi_\theta}(\vec{g}_{(t,\eta)}, \vec{w})$. After training, the actor network is used as the policy network of MOCC.

[2]Note that our MOCC framework can generalize to any other objectives.

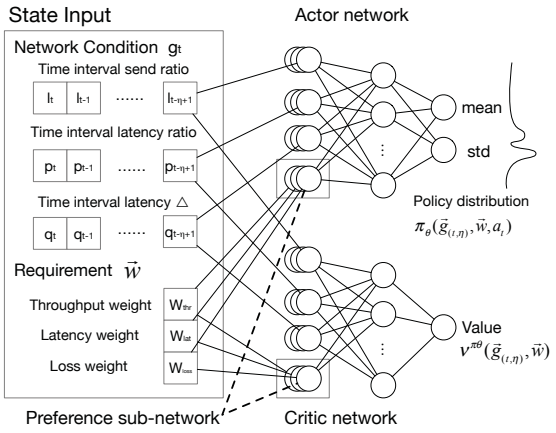


Figure 3. The Actor-Critic model that MOCC uses to generate CC rate control policies.

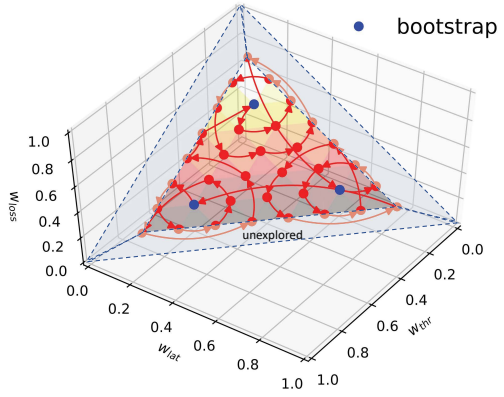


Figure 4. The objective training trajectory for fast traversing, generated by neighbourhood-based objective sorting algorithm (details in §4.2).

To support multiple objectives, MOCC extends both the actor network and critic network with a preference sub-network (PN). PN takes the application weight vector $\vec{w}$ as input, and performs feature transformation to concatenate with the network state $\vec{g}_{t,\eta}$ to feed both networks. Then, the actor network outputs a distribution of the action space for choosing the proper action.

By incorporating the PN, both the decisions made by the actor network and the evaluation given by the critic network are not only based on the network conditions, but also taking the application requirements into consideration. In other words, our MOCC model adopts neural network structure that can recognize different application requirements/preferences, and correlate them with the corresponding optimal policies. As a result, MOCC can learn and apply optimal rate control polices for multiple applications simultaneously, enabling multi-objective.

4.2 Offline Training

Our goal of offline pre-training is to learn the correlations between application requirements and optimal rate control polices, in order to make MOCC quickly adapt to new applications with high accuracy during deployment. In this section, we introduce our two-phase training strategy as well as the policy optimization algorithm.

Two-phase training: To train a multi-objective RL, one straightforward way is to decompose it into multiple single-objective RLs [35]. If we can enumerate all possible objectives and train each of them iteratively, the multi-objective RL can achieve the optimal Convex Converge Set. However, in our case of MOCC, there are infinite possible objectives, i.e., any weight vector that satisfies $w_{thr} + w_{lat} + w_{loss}$=1, $w_i \in (0, 1)$. The problem becomes intractable.

To efficiently train MOCC, instead of exploring the whole objective space, we train on a subset of landmark objectives, say ω, that can produce a satisfying model[3] However, even a moderate ω with tens of objectives will take several days to train. To speedup, we introduce a two-phase training: bootstrapping and fast traversing. In bootstrapping phase, we build a base model by selecting just a small number of objectives to train. In our implementation, we chose 3 to bootstrap with, and our base model can take hours to converge.

Then, in the fast traversing phase, building on the base model, we accelerate the training of the remaining ω-3 objectives by adopting a neighborhood-based transfer learning strategy [32]. This method is based on the observation that when two RLs have close objectives (i.e., similar weight vectors), their optimal solutions are close. Thus, when training a RL, we can speedup by leveraging the solutions of its neighboring RLs.

To do this, we purposely arrange the ω objectives in a neighborhood-based way as shown in Figure 4. We train from one objective to its neighbor iteratively and traverse all the objectives in a cyclic way. Note that each time we do not train an objective until convergence but only for a few steps in order to achieve balanced improvement on all objectives. The whole training completes when the model converges on all objectives. We explain why such two-phase training achieves near-optimal solution in Appendix B. The deep dive discussion in §6.5 shows it effectively speedups the training by 18×.

Neighborhood-based algorithm To speedup the training speed, we design a neighborhood-based algorithm to leverage the solutions of its neighboring RLs. Our algorithm is based on Dijkstra's shortest path algorithm. By constructing an undirected graph G from candidate objectives, we reorder objectives according to their distances from the bootstrapped ones. We construct G with vertices representing candidate weight vectors (all weight vectors

[3]In §6.5, we do a deep-dive discussion of ω and find that ω = 36 achieves a good performance.

satisfying $w_{thr} + w_{lat} + w_{loss}$=1, $w_i \in (0, 1)$, at a given step size), and edges representing the neighborhood relationships. We define two weight vectors to be neighbors if they differ in at most two dimensions and each dimension differs by less than the step size. For example, at the step size of 0.1, <0.2, 0.4, 0.4> and <0.2, 0.5, 0.3> are neighbors, <0.2, 0.4, 0.4> and <0.1, 0.5, 0.4> are neighbors, but <0.2, 0.4, 0.4> and <0.1, 0.3, 0.6> are not neighbors. We add edges between neighbors and set all edge weights to be 1.

Algorithm 1 presents the pseudocode for our neighborhood-based algorithm on G. We iterate on each bootstrapped objective/vertices and apply Dijkstra's algorithm: For the current bootstrapped vertex o, the algorithm extracts the nearest unvisited vertices, puts them into the list L, and updates its unvisited neighbors' distances from o. Finally, L contains a sorted list of objectives that can be used as the training order for our MOCC model.

To accelerate the fast traversing phase, we chose the bootstrapped objectives <0.6, 0.3, 0.1>, <0.1, 0.6, 0.3>, <0.3, 0.1, 0.6> to cover different application requirements as much as possible. Figure 4 illustrates the traversing path.

We do not use weight vectors that have 0 in any metrics/dimensions, whose training models, as we have evaluated, are overaggressive on specific metrics and make no sense for any application.

Policy optimization algorithm: Among a variety of different algorithms for training RL [22, 38, 47], we adopt Proximal Policy Optimization (PPO) [47] as the policy optimization algorithm to train MOCC. It is a policy gradient method updating the model with estimated gradient to maximize the expected total reward. We chose PPO because: 1) it is the state-of-the-art approach and easy to tune; and 2) it performs particularly well on continuous control problem [47], which makes it suitable for deciding the sending rates.

Instead of directly optimizing the expected total reward, PPO optimizes on its lower bound, a surrogate objective function (The lower bound proof is given in [47] and [46]):

$$L^{CLIP}(\theta, \vec{w})_t = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta), \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\right) \hat{A}_t \right] \quad (3)$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|\vec{v}_{(t,\eta)}, \vec{w})}{\pi_{\theta_{old}}(a_t|\vec{v}_{(t,\eta)}, \vec{w})}$ denotes the probability ratio of action a_t compared to the current policy. The term $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)$ clips the probability ratio to the range $[1-\epsilon, 1+\epsilon]$. $\hat{A}_t$ represents the advantage of a specific action over the current policy. The epsilon is a hyperparameter used to decide the clipping. This parameter prevents very large updates.

$\pi_\theta(.|g, w)$is in fact the policy probability distribution, as the entropy function is directly applied on probability distributions.

Advantage function is defined as the difference between empirical total reward applying the action a_t and expected total reward applying policy π_θ:

$$\hat{A}(\vec{g}_{(t,\eta)}, \vec{w}, a_t) = \sum_t \gamma^t r_t - V^{\pi_\theta}(\vec{g}_{(t,\eta)}, \vec{w}) \quad (4)$$

where $V^{\pi_\theta}(\vec{g}_{(t,\eta)}, \vec{w})$ is estimated by the critic network.

To encourage exploration of policy network, as suggested in past works [38], we add an entropy regularization term to the objective function L^{CLIP}:

$$L_t^{CLIP+E}(\theta, \vec{w}) = L^{CLIP}(\theta, \vec{w}) + \beta H(\pi_\theta(\cdot|\vec{g}_{(t,\eta)}, \vec{w})), \quad (5)$$

where $H(\cdot)$ is the entropy function of the probability distribution over actions at each time step. Thus action distribution with higher entropy is preferred, exploring a more diverse set of possible actions.

During offline training, for each step, MOCC's RL agent performs the policy of actor network to generate a network trace for a short period of time. With the trace and collected empirical rewards, we update the critic network following the standard *Temporal Difference* method [51]. Then, the critic network provides $V^{\pi_\theta}(\vec{g}_{(t,\eta)}, \vec{w})$ for computing the advantage function according to Equation 4. Finally, the actor network is updated with gradients computed to maximize

Algorithm 1: Neighborhood-based Objective Sorting Algorithm

input : The undirected objective graph $G = (V, E)$, the bootstrapped vertices O
output : The sorted objective list L

$L \leftarrow []$;
foreach $v \in V$ **do**
  $v.visited \leftarrow False$;
  for $i \leftarrow 1$ **to** $|O|$ **do**
    if *v has edge with bootstrapped vertices O* **then**
      $v.d[i] \leftarrow 1$;
    else
      $v.d[i] \leftarrow \infty$;
for $i \leftarrow 1$ **to** $|O|$ **do**
  $visits \leftarrow \lceil \frac{|V|}{|O|} \rceil$;
  if $O[i].visited = False$ **then**
    Append $O[i]$ to L;
    $O[i].visited \leftarrow True$;
    $visits \leftarrow visits - 1$;
  while *$visits > 0$ and L is not full* **do**
    Find $u \in V$ with minimum $u.d[i]$ and $u.visited = False$;
    Append u to L;
    $u.visited = True$;
    $visits \leftarrow visits - 1$;
    foreach *$w \in$ neighbors of u* **do**
      if *$w.visited = False$ and $u.d[i] + 1 < w.d[i]$* **then**
        $w.d[i] \leftarrow u.d[i] + 1$;

Equation 5. Because the surrogate objective is the lower bound of the expected total reward, optimization on it guarantees the improvement of the policy network on gained reward. As a result, model parameters are updated such that the new policy assigns higher probability to state-action pairs resulting in positive reward advantages, moving towards the optimal policy.

4.3 Online Adaptation

Our offline pre-trained model from §4.2 effectively correlates the application requirements with the optimal policies. This brings two key benefits to MOCC's online adaptation. First, for a new application, MOCC can generate a moderate policy of rate control even the requirement is unforeseen, providing reasonable performance for the new application at the beginning. Second, starting from such moderate model, with transfer learning, MOCC is able to quickly converge to the optimal model for the new application with just a few RL iterations, which is much faster than learning from scratch (e.g., we see over 10 times faster in §6.2). These two benefits enable MOCC to adapt to any new applications on-the-fly.

However, there is one issue: we do not want to compromise the performance of old applications while adapting to new ones. Unlike offline training where all objectives are artificially generated and uniformly distributed, the objective distribution in real environment may have bias: some applications are very frequent, some are rare. Under such a bias, the traditional RL algorithm will overfit to those new frequent applications but gradually forget those old rare ones, which is undesirable.

To avoid this problem, MOCC uses a requirement replay learning algorithm [3]. During the online learning, MOCC stores encountered applications (weight vectors) for a long period of time. For each online training step, the model is trained on both the current objective and an old objective drawn uniformly at random from the pool of the stored applications. We define the online learning objective to be:

$$L_{online}(\theta) = \frac{1}{2} * [L^{CLIP+E}(\theta, \vec{w}_i) + L^{CLIP+E}(\theta, \vec{w}_j)] \quad (6)$$

where $\vec{w}_i$ refers to the current application requirement, $\vec{w}_j$ refers to a sampled old application requirement, and L^{CLIP+E} is the PPO surrogate objective function defined in Equation 5. In this way, MOCC not only learns new applications, but also recalls old applications and reinforces previously learned policies. Thus, MOCC can preserve the learned policies of old applications while adapting to new applications. Our evaluation in §6.2 confirms this property.

5 Implementation

Our implementation of MOCC mainly consists of two components: 1) offline training, and 2) online deployment.

Parameter	Value
Discount factor (γ)	0.99
Learning rate (ϵ)	0.001
Action scale factor (α)	0.025
History length (η)	10
Landmark objectives # (ω)	36

Table 2. Parameter settings

Offline training: Directly training MOCC in real environment is slow considering the actual time cost in real control loops of CC with complex network dynamics [8, 26, 44]. To enable efficient training, we train MOCC in a networking simulator that faithfully mimics Internet links with various characteristics. Our simulator is based on OpenAI Gym [6] and Aurora [26], and further incorporates new design elements in §4.1–§4.2 that are essential to MOCC, such as the encapsulation of application requirements as state input and dynamic reward functions.

The MOCC policy network uses a fully-connected MLP (Multi-layer perceptron) with two hidden layers of 64 and 32 units, respectively, and `tanh` activation function to output the mean and standard deviations of the Gaussian distribution of action. The critic network uses the same neural network structure to estimate the scalar value function. We control the entropy factor β to decay from 1 to 0.1 over 1000 iterations, and set clipping threshold $\epsilon = 0.2$. For the learning rate, we adopt Adam [30], a famous adaptive learning rate optimization algorithm, which consistently outperforms standard SGD method. Important training parameter settings are listed in Table 2. We implement our model architecture with TensorFlow 1.14.0. For PPO implementation, we use an open-source implementation of several reinforcement learning baselines[4].

To further accelerate MOCC's exploration towards optimal solutions for massive objectives, in addition to the two-phase training introduced in §4.2, we also adopt parallel training. We implemented this architecture using Ray [39] and RLlib [33] to build the multiple parallel environments. For compatibility, we leverage Ray API to declare the neural network during both training and testing.

Online Deployment: After the MOCC model was offline trained in the simulator, it needs to be online deployed with the real Internet applications. For better portability, we encapsulate all MOCC's functions into one library. Our library provides three main functions:

- `Register(w)`. Before using MOCC, we should register with it by providing the requirement/preference (weight vector w) of the application.
- `ReportStatus(`s_t`)`. At each time interval, we should report the latest networking status (s_t) to MOCC.

[4] https://github.com/hill-a/stable-baselines

- `GetSendingRate( )`. When sending packets, we use this function to obtain the sending rate calculated by MOCC.

With clean encapsulation, MOCC becomes an easy-to-use module and can be deployed with any networking datapaths.

In our implementation, we integrate MOCC with UDT [20] and CCP [41] to build user-space MOCC and kernel-space MOCC. UDT is a reliable UDP based application level data transport protocol for distributed data intensive applications over wide area high-speed networks. It is a very widely used user-space implementation [1, 12, 13, 26, 59]. The `shim-helper` in UDT interacts with MOCC library to obtain the sending rate.

CCP is a more general solution and enables congestion control outside datapath such as Linux kernel networking stack. CCP can feed the network states from the kernel into MOCC and enforces the derived control action. We use CCP to deploy the trained MOCC model in Linux kernel 4.15.0-74-generic and thus MOCC can support more general-purpose applications. In §6.3, we use MOCC to support 3 real Internet applications: video streaming, real-time communications and bulk data transfer, and we will introduce more implementation details there. Furthermore, we note that MOCC with CCP achieves much lower CPU overhead than that with UDT (§6.5).[5]

6 Evaluation

We evaluate MOCC with extensive simulations as well as real Internet experiments. Our key results are as follows:

- **Multi-objective (§6.1):** Compared with a series of heuristic/learning CC algorithms, MOCC demonstrates its multi-objective performance by competing or outperforming the best existing schemes in supporting 2 common objectives: high throughput and low latency applications (Figure 5), as well as a generalized 100-objective setting (Figure 6).
- **Quick-adaptation (§6.2):** Compared with the state-of-the-art RL CC algorithm Aurora [26], MOCC can adapt to a new application in 4.8 minutes, 14.2× faster than Aurora (Figure 7a). Furthermore, MOCC does not compromise old applications while adapting to the new one, whereas Aurora does, significantly (Figure 7b).
- **Real Internet applications (§6.3):** Among all the algorithms compared, MOCC is the only one that can simultaneously provide high bitrate/throughput for video streaming (Figure 8) and bulk data transfer (Figure 10), while delivering the lowest inter-packet latency for real-time communications (Figure 9).

[5]Currently MOCC leverages CCP to control kernel TCP flows via transferring states and control actions between kernel and user-space. Fully implementing MOCC in kernel requires addressing the problem of float computing, which is left as future work.

	Bandwidth	Latency	Queue size	Loss rate
Training	1-5 Mbps	10-50ms	0-3000 pkts	0-3%
Testing	10-50 Mbps	10-200ms	500-5000 pkts	0-10%

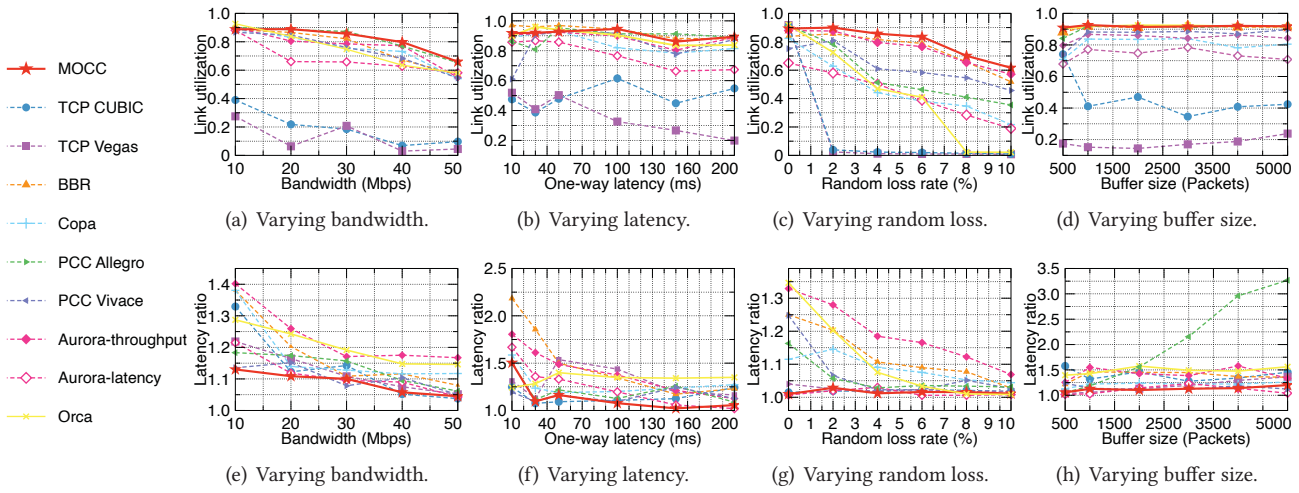
Table 3. Training/testing parameters

- **Fairness and Friendliness (§6.4):** MOCC with the same weight achieves fair share (Figure 11, 12), and MOCC variants with different weights grab different bandwidth according to $weight_{thr}$ (Figure 13). MOCC is friendly among its own variants (Figure 14) and achieves comparable TCP-friendliness as other CC schemes (Figure 15).
- **Deep-dive (§6.5)** into MOCC from different aspects such as hyperparameter setting (Figure 16), CPU overhead (Figure 17), training speedup (Figure 20) and learning algorithm selection (Figure 19) has validated its design efficiency.

Settings: Following §5, we use CCP to deploy our MOCC in both real Internet and Pantheon [59] emulated environment. We train MOCC with varied bandwidth, latency, queue size and loss rate to cover a wide range of network conditions following the settings of prior works [26, 59]. The key parameters used in both training and evaluation are shown in Table 3. In particular, when evaluating MOCC, we use a much wider parameter range beyond training to show the robustness of MOCC.

Schemes compared: We compare MOCC with various CC algorithms, including both handcrafted and learning-based:

1. Aurora [26]: RL based CC algorithm, single-objective RL, Aurora-throughput and Aurora-latency basically use two separate models.
2. Orca [1]: RL based CC algorithm, single-objective RL combined with the classic CC (CUBIC) to achieve low overhead and high performance.
3. PCC Allegro [12]: learning-based, performs *micro experiments* to continuously explore and learn the target sending rate.
4. PCC Vivace [13]: learning-based, extends upon Allegro to achieve better performance.
5. BBR [7]: model-based heuristic, builds an explicit model based on available bandwidth and RTT, and uses the model to control congestion window.
6. Copa [4], delay-based heuristic, computes the target sending rate by estimating minimum delay.
7. TCP CUBIC [21], loss-based heuristic, when packets are dropped, CUBIC modulates its congestion window based on a CUBIC function.
8. TCP Vegas [5], delay-based heuristic, uses RTT as congestion signal and controls congestion window to maintain desired RTT.



(a) Varying bandwidth. (b) Varying latency. (c) Varying random loss. (d) Varying buffer size.

(e) Varying bandwidth. (f) Varying latency. (g) Varying random loss. (h) Varying buffer size.

Figure 5. The multi-objective performance of MOCC in terms of throughput (a-d) and latency (e-h), under various network conditions. Note that the network conditions under evaluation are far beyond the environment where MOCC was trained, demonstrating both its high performance and robustness when adopted in practice.

6.1 Multi-objective Performance

To evaluate MOCC's multi-objective performance in supporting different application requirements, we compare it against the above CC algorithms across different network conditions.

2-objective: We first consider a simple case with two common objectives: high throughput and low latency. Even so, as high throughput and low latency typically conflict with each other, it is not easy to achieve both at the same time in prior solutions (§2.2). However, we show MOCC can achieve both objectives simultaneously, with weight vectors $\vec{w}_1$ =<0.8, 0.1, 0.1> and $\vec{w}_2$ =<0.1, 0.8, 0.1> respectively[6].

The detailed results are shown in Figure 5: (a) to (d) show the bottleneck link utilization with varied bandwidth, one-way RTT, loss rate and buffer size when the application prefers high throughput; (e) to (h) show the latency ratio [58] when the application demands low latency. We have tested MOCC across a wide range of network conditions and applications in Figure 5 to show its' good generalization. In general, MOCC can compete or outperform the best existing CC algorithms and show consistent high performance.

First, for hand-crafted CC schemes, MOCC can at least rival them in one objective and outperform them in the other, or even both. For example, MOCC achieves comparable throughput as BBR, while delivering up to 18.8% (1.12 to 1.38 in Figure 5(e)) lower latency. Furthermore, MOCC outperforms CUBIC in both throughput (at least 1.5× from 0.95 to 0.62 in Figure 5(b)) and latency (15% lower from 1.13 to 1.33 in Figure 5(e)). The reason is that handcrafted CC algorithms generally adopt hardwired policies based on pre-assumptions of network conditions and human experiences, thus hard to achieve optimal application-specific performance. In contrast, RL-based MOCC explicitly considers network conditions in state input and has been trained across a wide range of network conditions. Thus it will swiftly respond to the network and show optimal performance.

Second, we find that learning-based CC algorithms (non-RL), such as PCC Vivace and PCC Allegro, which essentially use online greedy optimization methods, could lead to relatively good performance. Since they avoid the problem of false pre-assumptions and hardwired mapping, they have high flexibility and robustness. Therefore, they could adapt to various network conditions. However, they are easy to trap in local optimum. MOCC uses RL to avoid this problem, and thus outperforms them with up to 1.43× better throughput (0.83 to 0.58 in Figure 5(c)) and 63.2% latency reduction (1.20 to 3.27 in Figure 5(h)) respectively.

Third, we find that RL-based Aurora/Orca show one-sided good performance (e.g., Aurora-throughput shows high throughput along with high latency) from Figure 5(a)5(b). The reason is that they use single-objective RL, which can only optimize for a fixed objective built through empirical reference, thus leading to degraded performance. Compared to RL-based Aurora/Orca, MOCC exceeds Aurora-throughput in terms of throughput (1.09× from 0.89 to 0.81 in Figure 5(a)) while outperforming Aurora-latency in terms of latency (7.5% lower from 1.23 to 1.33 in Figure 5(f)). Default Orca [1] shows similar trend except in the random loss case, due partially to the effect of CUBIC (its heuristic part). These results are expected because single-objective RL cannot simultaneously optimize for both throughput and latency. On the contrary,

[6]Note that we only used these two particular weight vectors as example, and any vectors with similar weight settings would work.

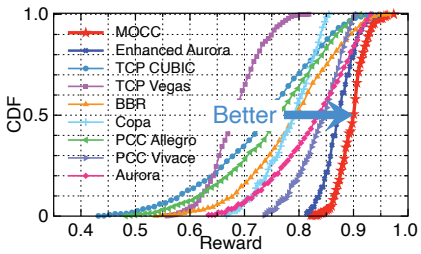
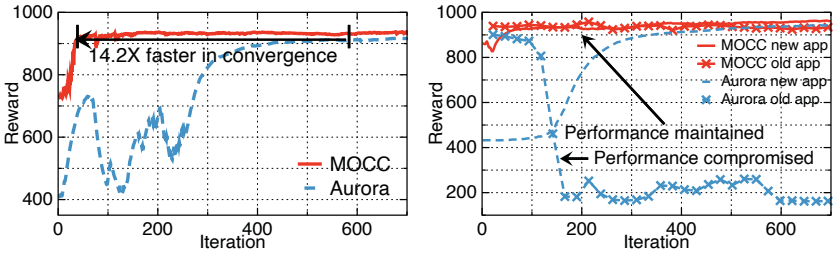


Figure 6. Quantitative CDF of Multi-Objective rewards for all CC algorithms compared.



(a) Quickly adapt to new application

(b) Not compromising old applications

Figure 7. The quick-adaptive property of MOCC. Note that we set the maximum iteration step to be 1000, so the maximum reward gain is 1000 on y-axis.

MOCC uses multi-objective RL to simultaneously support both objectives with one model.

The readers may wonder: *can we pre-train a few variants of Aurora/Orca (with different weights) to achieve multi-objective?*

Multi-Objective: To answer the above question, we consider a more generalized, uniformly-distributed 100-objective setting hypothetically. To make our results visually clear, we unify the performance metrics using *reward* calculated by Equation 2. In this experiment, MOCC only used offline trained model without online adaptation although which we believe is even better. We enhanced Aurora with 10 pre-trained models that best suit these 100 objectives (Orca shares similar property in terms of single-objective RL).

We run MOCC, enhanced-Aurora, and other CC algorithms under 10 different network conditions with 100 objectives, resulting in 1000 different scenarios. Figure 6 presents CDFs of rewards of these 1000 cases for all the algorithms compared. It is evident that MOCC outperforms all other CC schemes (including enhanced-Aurora) in satisfying various objectives across different network conditions. We find that enhanced-Aurora with 10 pre-trained models is secondary to MOCC, but vanilla Aurora with single model cannot perform well. The handcrafted heuristics, as expected, cannot well meet the multi-objective requirements because they are designed with no explicit application requirements in mind and their control policies are hardwired with pre-assumptions.

6.2 Quick Adaptation

To show how quickly MOCC adapts to new applications, we compare it against Aurora. We define the convergence point as 99% of the maximum reward gain.

Figure 7(a) plots the trend of both algorithms in adapting to new applications. The x-axis denotes the number of iterations and y-axis the gained reward. First, we observe that MOCC achieves 1.8× higher initial performance for a new application over Aurora. This suggests that by learning correlations between application requirements and optimal policies, MOCC can provide moderately good polices for applications with unseen requirements. Meanwhile, with transfer learning, such base correlation knowledge brings a 14.2× faster convergence speed (639 down to 45 iterations). This result confirms that MOCC can quickly adapt to new applications that have never been trained before, whereas the single-objective Aurora, without the correlation knowledge, re-trains the model from scratch which takes long time. After the adaptation, MOCC will respond to the learned application quickly with learned policy when it runs again, without further adaptation thereafter.

Figure 7(b) checks whether MOCC will degrade the performance of old applications while adapting to new ones. To do so, we snapshot the models of MOCC and Aurora every 8 iterations, and apply them to the old application to compute the rewards. The curves are illustrative. We observe that MOCC well preserves the performance of the old application with reward loss <5%. This is because MOCC has the correlation model and applies the requirement replay algorithm (§4.3) to recall the old application during online training. In contrast, Aurora, as a single-objective CC model, gradually forget the old application and degrades the performance greatly (916.1 to 156.1) while serving the new application.

6.3 Real Internet Applications

We now showcase the performance of MOCC with 3 real Internet applications: video streaming, real-time communications (RTC), and bulk data transfer. For video streaming, we deploy the video server in AWS Tokyo and fetch the video chunks by our local browser using residential network at HKUST. For the RTC, we deploy the real-time video sender on our residential network and deploy the receiver on our collaborators' network (about 15 hops), ensuring that the video transmits along the real internet. For bulk data transfer, we perform it in the datacenter environment with 1 Gbps interconnects. These applications have different requirements, and we use a single MOCC model to support all of them. We use CCP to deploy our MOCC on Linux kernel 4.15.0-74-generic. We compare it with TCP CUBIC, Vegas and BBR which are built-in algorithms in Linux Kernel and widely used in Internet.

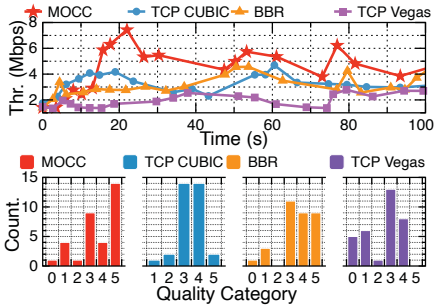


Figure 8. Video streaming

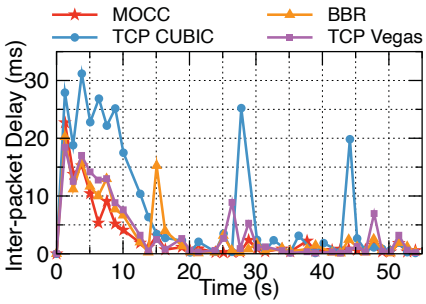


Figure 9. Real time communications

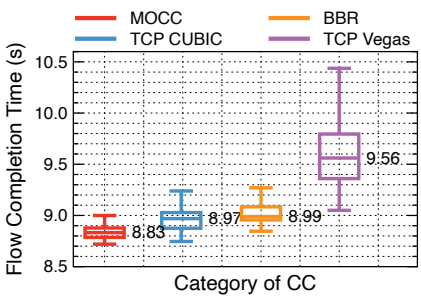


Figure 10. Bulk data transfer

Video streaming: In this experiment, we deployed a video streaming server and use ABR algorithm provided by Pensieve [44]. The best practice of MOCC is to choose the most suitable preference weight vector given by the application itself. we applied $\vec{w}$ =<0.8, 0.1, 0.1> for MOCC because video streaming applications need high throughput and are not sensitive to latency due to the playback buffer. We used our browser on client to play the video from the server via both WiFi and wired networks.

The experimental results are shown in Figure 8. We can see that MOCC continuously outperforms other CC algorithms in terms of throughput Figure 8 (top). Specifically, the average throughput of MOCC is 91.3% (4.4 to 2.3 Mbps) higher than Vegas, 33.3% (4.4 to 3.3 Mbps) higher than CUBIC, 29.4% (4.4 to 3.4 Mbps) higher than BBR. Figure 8 (bottom) shows the number of chunks with different quality levels (higher is better, level 5 is the best) obtained during video streaming. The level of chunk obtained is decided by MPC algorithm and a better networking condition leads to a higher level of chunks. In our experiment, MOCC can obtain the largest number of level 5 chunks compared to others (14 in MOCC vs 9 in BBR, 2 in CUBIC, and 0 in Vegas). The result further shows that MOCC can satisfy the requirement of video streaming application, outperforming the other algorithms.

Real-time communications (RTC): We deployed Salsify (the latest real-time WebRTC) [17] for RTC application. We modified Salsify to work with TCP. We applied $\vec{w}$ =<0.4, 0.5, 0.1> for MOCC because besides throughput, RTC applications also care about latency to avoid lags. We used our browser on client side to set up a conference call with the Salsify server via both WiFi and wired networks.

Figure 9 shows the average inter-packet delay. We observe that MOCC achieves the lowest inter-packet delay and is 21.1% (3.0 to 3.8 ms) better than BBR, 63.1% (3.0 to 7.9 ms) than CUBIC and 26.8% (3.0 to 4.1 ms) than Vegas. The result suggests MOCC can deliver the best performance to RTC applications by keeping low inter-packet delay.

Bulk data transfer: For bulk data transfer, we connected a server and a client by a switch and sent large file via Python. As the file transfer is throughput-hungry, we greedily applied $\vec{w}$ =<1, 0, 0> for MOCC. We transfer a 100MB file for 50 times. We also add a random loss rate of 0.5% to the links to emulate background traffic interference.

Figure 10 shows the results. Compared to others, MOCC achieves the lowest average file transfer completion time and is 1.56% (8.83 to 8.97 ms) lower than CUBIC, 1.78% (8.83 to 8.99 ms) lower than BBR and 7.63% (8.83 to 9.56 ms) than Vegas. Besides, MOCC maintains the most stable performance, and the standard deviation of these 50 measurements is 0.096, while BBR is 0.154, CUBIC 0.123 and Vegas 0.421, respectively. This result shows that MOCC can provide consistent high bandwidth to throughput-intensive applications.

6.4 Fairness and Friendliness

To evaluate the fairness and friendliness of MOCC, we compare MOCC with other CC schemes using Pantheon [59]. Fairness considers the scenarios where all flows use the *same* CC scheme, and friendliness considers those with *different* CC schemes (including MOCC with different weights).

Fairness: We use a canonical setting for evaluating fairness: several flows use the same CC scheme to share a bottleneck link in a dumbbell topology. The link is configured with 12Mbps bandwidth, 20ms RTT and 1 × BDP buffer, and three flows initiates sequentially with a 100s interval. Figure 11 shows the throughput of different flows for each scheme. As expected, MOCC (with the same weight) allocates bandwidth fairly between competing flows. Furthermore, it also achieves fast convergence, because it adjusts the sending rate with a multiplicative factor as defined in Equation 1.

We also use the Jain's fairness index [25] to quantitatively compare the fairness of different schemes for the same setup. A close-to-1 value indicates better fairness. We compute the Jain's fairness index for each second for each scheme, and we also include three variants of MOCC configured with different weights. Figure 12 shows the CDF curve. From the figure, we confirm that: 1) MOCC achieves better fairness compared to other CC schemes in general, and 2) its fairness is irrespective of its weight configuration.

Friendliness: Then we consider MOCC's fairness in competition with existing algorithms. We first evaluate the friendliness of MOCC with different weights. The setup has two

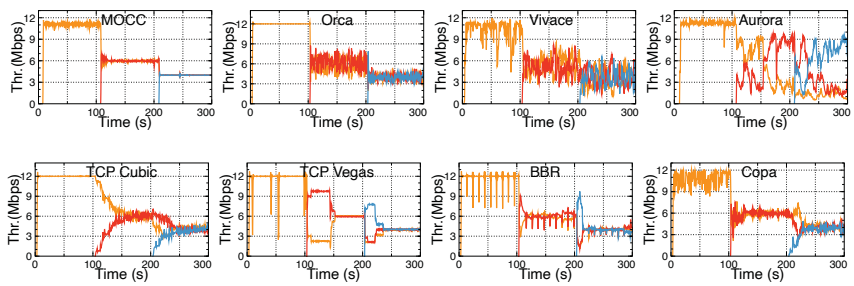


Figure 11. Throughput dynamics of different flows competing one link for various CC

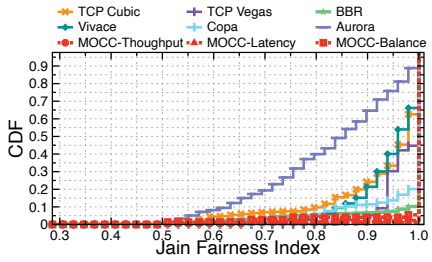


Figure 12. CDF of Jain Fairness Index under dynamics of flows.

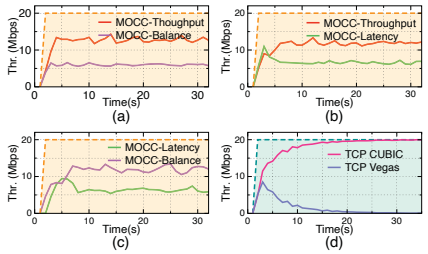


Figure 13. Throughput of MOCC flows with different weights

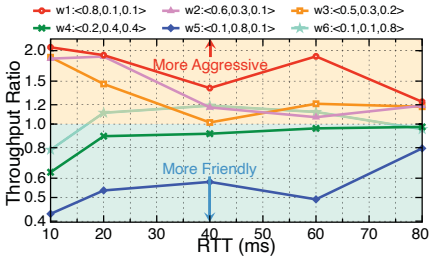


Figure 14. Friendliness ratio of MOCC under different weights

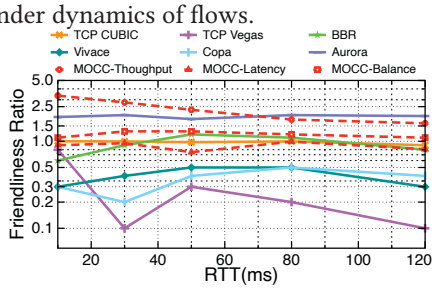


Figure 15. Friendliness ratio of schemes across different RTTs

flows sharing a bottleneck link of 20Mbps bandwidth, 20ms RTT and 1×BDP buffer. We use three MOCC variants, which are MOCC-Throughput, MOCC-Balance, and MOCC-Latency. Figure 13(a)(b)(c) show pairwise competitions of the three variants. These MOCC variants are technically *different* CC schemes, and a variant with a larger $weight_{thr}$ would be more aggressive to get more bandwidth. For comparison, Figure 13(d) shows the result for a TCP Cubic flow vs. a TCP Vegas flow.

MOCC is friendly in the sense that no MOCC flow will grab all bandwidth when multiple MOCC flows with different weights co-exist. This is because all MOCC flows share one objective framework, which is guaranteed to converge to a stable rate configuration [23]. We performed another simulation to further demonstrate this point with more MOCC variants in Figure 14. We fix the bandwidth to 20Mbps and change the RTT from 10ms–90ms. The results show that the throughput ratio varies between 0.43–2.04, which confirms the friendliness of MOCC under different weights.

Finally, we evaluate the friendliness of MOCC with other TCP schemes. We use a common setup that has two flows competing one link. Following the convention of friendliness evaluation in prior work [1, 13], we fix TCP Cubic as the target CC scheme of one flow, and vary the CC scheme of the other flow to compare between them. We use the friendliness ratio as the metric, which is defined by $\frac{Delivery-rate-of-CC-scheme}{Delivery-rate-of-Cubic-flow}$. Figure 15 reports the friendliness ratios of different schemes. The results indicate that MOCC-Throughput is more aggressive in obtaining bandwidth, and MOCC-Balance and MOCC-Latency are more friendly to TCP Cubic. In general, MOCC is comparable to other CC schemes in friendliness.

6.5 MOCC Deep Dive

Finally, we deep-dive into MOCC from some other aspects, including hyperparameter setting, CPU overhead, and training speedup.

Hyperparameter setting: We explore several key hyperparameters that may affect the effectiveness of MOCC, i.e., the learning-related parameters in Table 2. For history length (η) and discount factor (γ), we performed an exhaustive search and obtain similar results with [26]. For learning rate ϵ, we followed the default value suggested in stable-baseline's PPO [47], and we also tried several different values and found that $\epsilon = 0.001$ indeed leads to fast convergence. For the remaining, we discuss the number of pre-grained objective weight vectors (ω) that is unique to MOCC.

The parameter ω causes the tradeoff between the quality of base model and the time cost of training. A larger ω brings better model quality but also increases the training time. To understand the tradeoff, we pre-train MOCC with different ω to study its performance as well as training time. Figure 16 (top) shows the CDF of rewards of MOCC with different number of pre-trained objectives. In general, we can see that the model quality improves as ω increases[7], all the way until $\omega = 36$. We find that $\omega = 36$ has comparable quality as $\omega = 171$, both are within 0.82 to 0.96, outperforming $\omega = 3, 6, 12$ by 3×, 1.5×, 1.2× on average. Meanwhile, the

[7]We vary the step size of the objective weight vectors in terms of 1/4, 1/5, 1/6, 1/10, 1/20 leading to $\omega = 3, 6, 12, 36, 171$.

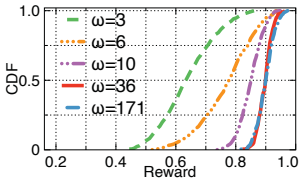


Figure 16. Hyperparameter setting (ω)

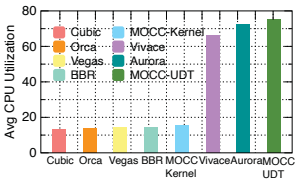


Figure 17. CPU Overhead of different CC schemes

training time of $\omega = 36$ is 5.2 hours, much shorter than $\omega = 171$ (28.2 hours) and reasonably longer time than $\omega = 6$ (2.6 hours). As a result, in this paper we set $\omega = 36$.

Overhead: We evaluate the overhead of MOCC by sending traffic on a 40Mbps link with 20ms RTT and 1 × BDP buffer. We use *taskset* to allocate processes to one CPU and report CPU utilization by *htop*. We exclude the first and the last few seconds for fair comparison. Results in Figure 17 show that User-space MOCC has high overhead similar to Aurora, because MOCC agent repeats model inference in each time interval similar to Aurora. Kernel-space MOCC achieves much lower overhead as Orca [1], because with CCP, the algorithm logic is isolated from the data-path. This decoupling provides CC feedback less frequently and significantly reduces the CPU utilization. Nevertheless, how to reduce the computation overhead of RL-based CC algorithms is still a challenge and may require a refinement of the model architecture and learning process, which we leave for the future work.

Learning algorithm selection and **Training speedup** are left in Appendix C for space limitation.

7 Discussion & Future Directions

Expressing application requirements: In MOCC, an application expresses its requirement as a weight vector over several network-level metrics (e.g.,throughput, latency and packet loss rate), and MOCC trains a model to optimize for the vector. The reason we choose throughput, latency and loss as the three objectives is that they are the most important and common metrics in congestion control. Yet, applications care about application-level objectives, which may not be directly mapped to a weight vector of network-level metrics. It's practical for one application to set its own objective for some specific requirements (e.g. friendliness, jitter) and build their own Multi-objective Congestion Control system. They need to retrain the whole model under the new object setting. In that way, the model architecture and training scheme in MOCC still work for building and training procedure.

Meanwhile, at a high level, the objective weights should be set based on the application-level objectives, e.g., real-time applications should give a higher weight to latency, and bandwidth-intensive applications should give a higher weight to throughput. But how to optimally set the weights to best express an application's requirement still requires human expertise and domain knowledge. We envision a learning-based approach, which learns the mapping from an application-level objective to a weight vector, can be applied to automate this process and reduce human efforts.

Model sharing and Federated learning: For an unseen application, MOCC leverages transfer learning to quickly adapt its model to the new application. Another device may have already run this application and trained a model to optimize the performance. If different devices can share their models, it would further reduce the adaptation time for MOCC. However, sharing models may raise privacy concerns as a trained model may unexpectedly leak a user's traffic pattern and network condition, which could be further used to reveal the user's other sensitive information. This setup is similar to federated learning where a model is trained across multiple decentralized devices. Extending MOCC with privacy-preserving federated learning is an interesting future direction.

Towards a general multi-objective framework for networking: While we focus on congestion control in this paper, we believe the framework behind MOCC is more generic and can be applied to a wide range of networking problems [8, 34, 44, 55]. This framework is particularly relevant given the recent proposals that leverage reinforcement learning to solve networking problems and demonstrate superior performance over traditional heuristics. For example, it can be applied to NeuroCuts [34] to learn to build packet classification trees with multiple objectives on classification time and memory footprint, and be applied to Pensieve [44] to learn adaptive bitrate algorithms with different Quality of Experience (QoE) metrics.

8 Conclusion

This paper established a multi-objective congestion control (MOCC) framework that enables *one single CC algorithm* to effectively support multiple application requirements. To enable multi-objective, MOCC constructs its policy network with a preference sub-network that correlates application requirements with optimal rate control policies. Furthermore, it exploits transfer learning to adapt MOCC to any new applications quickly in an online manner. Extensive simulations and real Internet experiments have shown that MOCC achieves all its design goals.

Acknowledgment

This work is supported in part by the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621, and the National Natural Science Foundation of China (NSFC) under Grant 62172008. We thank our shepherd Mihai Budiu and the anonymous reviewers for their constructive feedback and suggestions. Kai Chen is the corresponding author.

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. 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*. 632–647.

[2] Axel Abels, Diederik Roijers, Tom Lenaerts, Ann Nowé, and Denis Steckelmacher. 2019. Dynamic Weights in Multi-Objective Deep Reinforcement Learning. In *International Conference on Machine Learning*. 11–20.

[3] Axel Abels, Diederik M Roijers, Tom Lenaerts, Ann Nowé, and Denis Steckelmacher. 2018. Dynamic Weights in Multi-Objective Deep Reinforcement Learning. *arXiv preprint arXiv:1809.07803* (2018).

[4] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical delay-based congestion control for the internet. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 329–342.

[5] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. 1994. *TCP Vegas: New techniques for congestion detection and avoidance*. Number 4. ACM.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI gym. *arXiv preprint arXiv:1606.01540* (2016).

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

[8] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. 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*. 191–205.

[9] Xi Chen, Ali Ghadirzadeh, Mårten Björkman, and Patric Jensfelt. 2018. Meta-Learning for Multi-objective Reinforcement Learning. *arXiv preprint arXiv:1811.03376* (2018).

[10] Yan Chen, Toni Farley, and Nong Ye. 2004. QoS requirements of network applications on the Internet. *Information Knowledge Systems Management* 4, 1 (2004), 55–76.

[11] Ron Davies. 2016. *5G Network Technology: Putting Europe at the Leading Edge*. EPRS, European Parliamentary Research Service, Members' Research Service.

[12] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. 2015. {PCC}: Re-architecting Congestion Control for Consistent High Performance. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 395–408.

[13] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. {PCC} Vivace: Online-Learning Congestion Control. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 343–356.

[14] James Durward, Jonathan Levine, Michael Nemeth, Jerry Prettegiani, and Ian T Tweedie. 1997. Virtual reality network with selective distribution and updating of data to reduce bandwidth requirements. US Patent 5,659,691.

[15] Sally Floyd, Tom Henderson, Andrei Gurtov, et al. 1999. The NewReno modification to TCP's fast recovery algorithm. (1999).

[16] François Fluckiger. 1995. *Understanding networked multimedia: applications and technology*. Prentice Hall International (UK) Ltd.

[17] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify: low-latency network video through tighter integration between a video codec and a transport protocol. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 267–282.

[18] Borko Furht. 2011. *Handbook of augmented reality*. Springer Science & Business Media.

[19] Yun Gao, Xin Wei, and Liang Zhou. 2020. Personalized QoE improvement for networking video service. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2311–2323.

[20] Yunhong Gu and Robert L Grossman. 2007. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks* 51, 7 (2007), 1777–1799.

[21] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 5 (2008), 64–74.

[22] Matthew Hausknecht and Peter Stone. 2015. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*.

[23] Elad Hazan. 2019. Introduction to online convex optimization. *arXiv preprint arXiv:1909.05207* (2019).

[24] Liangyu Huo, Zulin Wang, Mai Xu, Yong Li, Zhiguo Ding, and Hao Wang. 2019. A Meta-Learning Framework for Learning Multi-User Preferences in QoE Optimization of DASH. *IEEE Transactions on Circuits and Systems for Video Technology* 30, 9 (2019), 3210–3225.

[25] Raj Jain, Arjan Durresi, and Gojko Babic. 1999. Throughput fairness index: An explanation. In *ATM Forum contribution*, Vol. 99.

[26] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *International Conference on Machine Learning ICML*. 3050–3059.

[27] Cheng Jin, David X Wei, and Steven H Low. 2004. FAST TCP: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, 2490–2501.

[28] Gunnar Karlsson. 1996. Quality requirements for multimedia network services. In *Proceedings of Radiovetenskap ach kommunikation*. 96–100.

[29] Nicholas D Kenyon and C Nightingale. 1992. *Audiovisual telecommunications*. Chapman & Hall, Ltd.

[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[31] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *ACM SIGCOMM*.

[32] Kaiwen Li, Tao Zhang, and Rui Wang. 2019. Deep Reinforcement Learning for Multi-objective Optimization. *arXiv preprint arXiv:1906.02386* (2019).

[33] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. 2017. Ray rllib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381* (2017).

[34] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*. 256–269.

[35] Chunming Liu, Xin Xu, and Dewen Hu. 2014. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 3 (2014), 385–398.

[36] Chunming Liu, Xin Xu, and Dewen Hu. 2014. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 3 (2014), 385–398.

[37] Simone Mangiante, Guenter Klas, Amit Navon, Zhuang GuanHua, Ju Ran, and Marco Dias Silva. 2017. Vr is on the edge: How to deliver 360 videos in mobile networks. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*. 30–35.

[38] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. 1928–1937.

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[40] Hossam Mossalam, Yannis M Assael, Diederik M Roijers, and Shimon Whiteson. 2016. Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707* (2016).
[41] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 30–43.
[42] Sriraam Natarajan and Prasad Tadepalli. 2005. Dynamic preferences in multi-criteria reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*. 601–608.
[43] J-P Nussbaumer, Baiju V. Patel, Frank Schaffa, and James P. G. Sterbenz. 1995. Networking requirements for interactive video on demand. *IEEE Journal on Selected Areas in Communications* 13, 5 (1995), 779–787.
[44] S Tejaswi Peesapati, Victoria Schwanda, Johnathon Schultz, Matt Lepage, So-yae Jeong, and Dan Cosley. 2010. Pensieve: supporting everyday reminiscence. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2027–2036.
[45] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. 2015. Universal value function approximators. In *International conference on machine learning*. 1312–1320.
[46] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International conference on machine learning*. 1889–1897.
[47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
[48] Regina Melo Silveira, Cíntia Borges Margi, LG Gonzalez, E Favero, OD Vilcachagua, Graça Bressan, and Wilson Vicente Ruggiero. 1999. A Multimedia on Demand System for Distance Education. In *International Conference on Technology and Distance Education, Fort Lauderdale-Florida*.
[49] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. 2014. An experimental study of the learnability of congestion control. In *ACM SIGCOMM Computer Communication Review*.
[50] Richard S Sutton, Andrew G Barto, et al. 1998. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge.
[51] Richard S Sutton, Andrew G Barto, et al. 1998. *Introduction to reinforcement learning*. Vol. 2. MIT press Cambridge.
[52] Bohdan O Szuprowicz. 1995. *Multimedia networking*. McGraw-Hill, Inc.
[53] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. 2006. A compound TCP approach for high-speed and long distance networks. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 1–12.
[54] Matthew E Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, Jul (2009), 1633–1685.
[55] Mowei Wang, Yong Cui, Shihan Xiao, Xin Wang, Dan Yang, Kai Chen, and Jun Zhu. 2018. Neural network meets DCN: Traffic-driven topology adaptation with deep learning. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 1–25.
[56] CC White. 2001. *Markov decision processes*. Springer.
[57] Keith Winstein and Hari Balakrishnan. 2013. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 123–134.
[58] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. 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)*. 459–471.
[59] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*.
[60] Runzhe Yang, Xingyuan Sun, and Karthik Narasimhan. 2019. A Generalized Algorithm for Multi-Objective Reinforcement Learning and Policy Adaptation. In *Advances in Neural Information Processing Systems*. 14610–14621.
[61] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. 2019. Congestion control for cross-datacenter networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 1–12.
[62] Gaoxiong Zeng, Jianxin Qiu, Yifei Yuan, Hongqiang Liu, and Kai Chen. 2021. FlashPass: Proactive Congestion Control for Shallow-buffered WAN. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 1–12.

Appendix

A Background on Multi-Objective RL

We briefly review multi-objective reinforcement learning (MORL) techniques [2, 36, 40, 45, 60] we used in this paper. In particular, we introduce **two design choices** and **two enhancements** in order to adopt MORL for the problem of MOCC. To the best of our knowledge, MOCC is the first work to solve the multi-objective CC problem by adopting the MORL framework. We make the following two design choices.

We use multiple-policy approach to optimize for each application preference rather than average preference of all applications. Existing MORL algorithms can be divided into two groups: single-policy approaches and multiple-policy approaches [36]. While single-policy approaches learn a single policy to optimize the average performance among different objectives, multiple-policy approaches learn and maintain a set of optimal policies. A general method adopted by multiple-policy approaches is to collect policies by running standard RL algorithm over different preferences [42]. To simultaneously support multiple existing applications and quickly adapt to new arrival applications while not compromise the performance of old applications, MOCC adopts multiple-policy approach to learn and maintain multiple policies for every application requirement.

We use policy-based rather than value-based algorithms to optimize for CC where the decision space is continuous instead of discrete. The training approach for RL can be divided into two groups: value-based approaches and policy-based approaches. Value-based approaches estimate the value of each state, and take action with highest value estimation. Policy-based approaches directly learn the optimal policy for the task. Generally, policy-based algorithms outperform value-based ones for continuous control problems because policy models can directly output continuous action [47]. Recent MORL algorithms are all value-based approaches [2, 45, 60]. However, to better fit the continuous property of the sending rate in CC, MOCC adopts policy-based algorithm PPO and transfers MORL structures from value-based to policy-based.

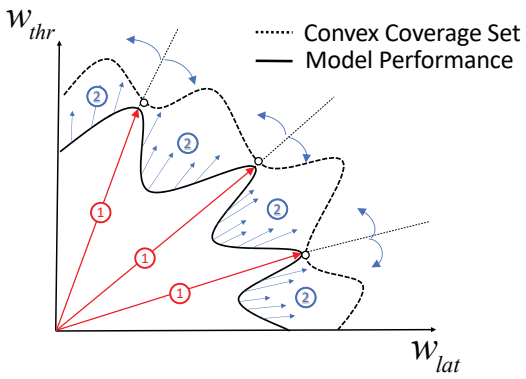


Figure 18. The illustration of how MOCC's performance is improved during the two-phase offline training on the preference space. For the first bootstrapping phase, MOCC learns the optimal policies on a few pivot points. Transferred from these learned policies, we iterate on other objectives in a neighborhood-based way (§4.2) to improve the overall performance in the fast traversal phase.

Besides, based on the above design choices, we also make following two enhancements towards MORL:

Enhance PPO with requirement replay learning algorithm in order not to compromise performance of old applications. DQN-based Conditional Network (CN) proposed in [2] adopts new training policy to adapt to new policy as well as maintain previously learned policies. Based on it, we design the requirement replay learning mechanism for PPO during online learning (§4.3) to recall old applications.

Enhance MORL training with transfer learning to accelerate training speed. Furthermore, with the prospect shown in [45] of using transfer learning to solve new objective faster from similar learned objectives, we design the two-phase offline training scheme with the neighbourhood-based algorithm (§4.2) to unleash the full power of transfer learning and significantly speedup our training (§6.5).

B Two-phase offline training illustration

We use Figure 18 to illustrate why the two-phase training can effectively accelerate the training. Here we only consider the two-dimensional preference space with two performance metrics, throughput and latency, for visual simplicity. Each point in this figure shows a certain objective (a combination of throughput and latency requirements) and the distance to the origin point shows the effectiveness (i.e., how optimally the model can act) of the model.

The dashed curve is convex converge set (CCS), which represents the optimal solution of the task. The solid curve represents the effectiveness of our model. The goal of training is to make the effectiveness (solid line) of our model to approach the optimality (dash line).

After the bootstrapping phase, we have a set of solutions for certain set of objectives, shown as pivot points in the figure. These points are: 1) uniformly distributed, and 2) very close to optimality. The training is also fast because the set is small. Then, in the fast traversing phase, we will determine the rest of the points. Because we already have those pivot points, we can have a very good starting point during training and the effectiveness will not be far away from optimality. Meanwhile, the training can be effectively accelerated by using the information provided by pivot points.

C Deep Dive: Additional Results

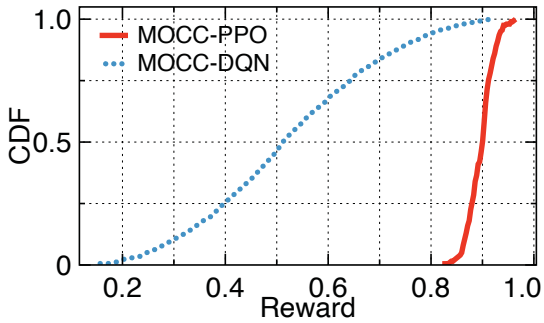


Figure 19. Comparison between learning algorithms

Learning algorithm selection: In this paper, we chose the PPO algorithm as our RL algorithm. An alternative approach is Q-learning [51]. In this experiment, we compare both algorithms to revisit the design decision of using PPO. For this purpose, we implemented a Q-learning version of MOCC, MOCC-DQN. Figure 19 compares MOCC-PPO with MOCC-DQN. We observe that MOCC-PPO significantly outperforms MOCC-DQN by achieving 3× more rewards on average. The reason is that for CC problems, the sending rate is a continuous value. However, Q-learning scales poorly with the continuous action space, causing sub-optimal performance. On the contrary, PPO is able to output continuous action values. So we select PPO to enable a more fine-grained rate control policy.

Training speedup: We evaluate the effectiveness of our training speedup techniques, including both neighborhood-based transfer learning strategy and parallel training. We train MOCC in three ways. First, we treat each single-objective as a standard RL subproblem and train them separately. Second, we use two-phase training with neighbourhood-transfer method (§4.2), without parallel training. Third, based on the second one, we add parallel training. The results are shown in Figure 20. We observe that through transferring across neighbor objectives, we reduce the training time by 18× (6 days 7.2 hours to 8.4 hours), which validates that transfer learning can significantly accelerate the training. In addition,

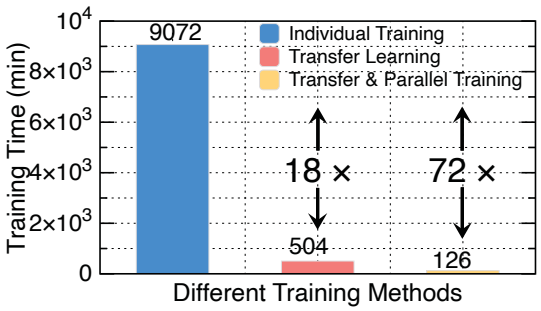


Figure 20. Training speedup techniques

we find parallel training can further speedup the training by 4× (8.4 to 2.1 hours).