Auter: Automatically Tuning Multi-layer Network Buffers in Long-Distance Shadowsocks Networks

Xu He*, Jiahao Cao†, Shu Wang*, Kun Sun*, Lisong Xu§, and Qi Li‡
*Center for Secure Information Systems, George Mason University, VA, USA
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
‡Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
§School of Computing, University of Nebraska-Lincoln, NE, USA
{xhe6, swang47, ksun3}@gmu.edu, caojh15@gmail.com, xu@cse.unl.edu, qli01@tsinghua.edu.cn

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796882

***Abstract*—To bypass network censorship, Shadowsocks is often deployed on long-distance transnational networks; however, such proxy networks are usually plagued by high latency, high packet loss rate, and unstable bandwidth. Most existing tuning solutions rely on hand-tuned heuristics, which cannot work well in the volatile Shadowsocks networks due to the labor intensive and time-consuming properties. In this paper, we propose *Auter*, which automatically tunes multi-layer buffer parameters with reinforcement learning (RL) to improve the performance of Shadowsocks in long-distance networks. The key insight behind Auter is that different network environments require different sizes of buffers to achieve sufficiently good performance. Hence, Auter continuously learns a tuning policy from volatile network states and dynamically alter sizes of multi-buffers for high network performance. We prototype Auter and evaluate its effectiveness under various real networks. Our experimental results show that Auter can effectively improve network performance, up to 40.5% throughput increase in real networks. Besides, we demonstrate that Auter outperforms all the existing tuning schemes.**

I. Introduction

Shadowsocks [1], [2] is a proxy application that has been widely deployed across regions or countries to circumvent Internet censorship. However, due to volatile link status and censorship, such long-distance proxy networks are usually plagued by high latency, high packet loss rate, and unstable bandwidth. For better user experience, it is critical to improve the tunneling connections between the Shadowsocks client and the Shadowsocks server. As network environments cannot be easily controlled, network engineers typically tune a series of network parameters in the TCP/IP protocol stack of end hosts to optimize tunneling connections. For example, the Shadowsocks provider suggests several specific configurations of network-related parameters to improve the tunneling performance, including enhanced congestion control algorithms such as BBR [3], amplified buffer sizes, and a series of enabled tuning mechanism parameters [4].

However, manually tuning network-related parameters falls into two main limitations. First, it relies on hand-tuned heuristics, where engineers fine tune and summarize the parameter configurations for specific network scenarios based on their domain knowledge. This manual tuning process is labor-intensive, time-consuming, and cost-expensive. Second, the one-shot static parameter setting is destined for long-term usage in relatively stable network environments, which does not work well in volatile network environments with varying latency and unstable bandwidth. However, such environments are common in the long-distance Shadowsocks networks. Particularly, our experiments find that even the most aggressive static tuning scheme such as maximizing the buffer size cannot consistently maintain high performance over a long term.

There are several efforts to overcome the limitations of manually tuning network-related parameters. An adaptive tuning scheme uses a feedback control mechanism to adaptively tune the socket buffer using a rule-based model [5]. However, it only works on a single parameter. Another method [6] exploits the evolutionary algorithm to select optimal value for multiple parameters. However, it is still a static tuning method, which may not adapt well to the quickly changing network environments. Recently, deep learning and reinforcement learning (RL) techniques have been adopted in congestion control, due to their flexibility and better performance [7], [8]. However, those solutions mainly focus on tuning a single parameter, and they cannot simultaneously tune multiple network parameters, which are critical to improve the performance of Shadowsocks running in the volatile long-distance networks. Therefore, Shadowsocks providers still rely on empirical results via manual tuning to configure those network parameters.

In this paper, we propose an RL-based tuning system called *Auter* that automatically and dynamically tunes multi-layer network buffers to improve the performance of Shadowsocks. Auter consists of three components including *network perception*, *decision making*, and *policy enforcement*. The network perception component collects a series of information about traffic performance states and transforms them as the inputs to the decision-making agent. The agent calculates the rewards based on the performance states and periodically generates the tuning action policies. The policy enforcement component maps the action policies into buffer size values and hence updates the parameters.

We solve two challenges when applying the RL model into the network parameter tuning task. The first challenge is to find out what network parameters are suitable for continuous tuning at runtime. Our preliminary experiments show that not all network parameters can be adjusted continuously. Actually, changing some parameters (e.g., the Generic Receive Offloading (GRO)) request reboot the Network Interface Card (NIC) to enable the new configuration values [9]. Via our analysis and experimental verification, we identify three types

of network buffers, i.e., *ring buffer*, *socket buffer*, and *TCP socket buffer*, which support runtime tuning and thus could be accommodated in the RL model for dynamic adjustment. Particularly, we discover that the driver-level ring buffer could be dynamically tuned along with the socket buffer and TCP socket buffer to improve the network performance.

The second challenge is on how to decide suitable range and intervals for tuning network parameters. We find that different buffers have different adjustment range. For instance, the adjustable range of the ring buffer is from 64 to 4096 packets, while the range of the socket buffer is from 5120 to 67108864 bytes. Meanwhile, different parameters have specific relationships and constraints. For instance, both the socket buffer and the TCP socket buffer have two adjustable related parameters, i.e., the default and maximum values. However, the default value must be kept less than the maximum value during the tuning process. Besides, we find the ring buffer must maintain a longer tuning interval than the other two buffers. Through studying the network stack implementation and conducting extensive experiments, we determine the adjustable tuning range and interval for each buffer parameter.

We prototype Auter and conduct extensive experiments to evaluate its effectiveness on improving the Shadowsocks performance. We first test the adaptability of Auter under different network scenarios with various latency, bandwidth, packet loss, and flow size. The results show that Auter is robust for various network scenarios, especially for the high latency network. Auter can increase 4.67x throughput in the 40ms latency simulated environment. We also deploy Shadowsocks with Auter on the Azure and Tencent cloud platforms to evaluate its effectiveness in real networks. Auter can achieve up to 40.5% throughput increase and 20.33% throughput increase on average across three regions. We also compare Auter with BBR and other static tuning schemes, and the results show that Auter always outperforms the existing optimization schemes.

In summary, our paper makes the following contributions.

- We propose an automatic and dynamic network buffer tuning system based on reinforcement learning to improve the Shadowsocks performance. It covers multi-layer buffers and can dynamically adapt to complex long-distance networks.
- We design differentiated buffer tuning mechanisms, which tunes ring buffer, socket buffer, and TCP socket buffer with different scale and intervals.
- We prototype a demo of Auter and verify its effectiveness under various network conditions and real networks.

II. Buffer Impacts for Shadowsocks

In this section, we first introduce the communication process of Shadowsocks. Then we present three buffers for data transfers between network applications and NICs. Next, we show how their buffer size impacts the performance of Shadowsocks under different network environments.

A. Shadowsocks

Shadowsocks is a proxy application based on Socks5 protocol [10]. It consists of two components: *SS_local* and *SS_Server*. The communication process of Shadowsocks is shown in Figure 1. Typically, the *SS_Local* is the client deploying at the local host. It interacts with the user applications based on Socks5 protocol and then encrypts the traffic and forwards them to the *SS_Server*. The *SS_server* is deployed on the other side behind the firewall. When receiving the encrypted traffic, *SS_Server* decrypts it and forwards them to the target servers. After getting the responses, *SS_Server* encrypts the traffic again and sends it back to the *SS_Local*. Since Shadowsocks is mainly used to bypass censorship, it is typically deployed across regions and even across countries, facing complex and volatile network environment.

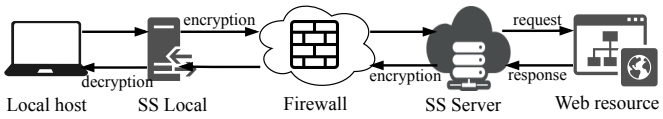


Fig. 1. The communication process of Shadowsocks.

B. Buffers of Network Layers

Socket Buffers. Socket sending and receiving buffers lie between network applications and TCP/IP stack. Intuitively, the larger the socket buffer size, the greater the throughput. However, larger socket buffers require larger memory allocation. When there are hundreds of connections, large socket memory may severely limit the network performance, especially for applications with high concurrency requirements [11]. Hence, a larger socket buffer does not always mean better performance. It should be dynamically adjusted according to applications, memory limitations, and network environments.

TCP Socket Buffers. Besides general socket buffers, TCP connections have their dedicated TCP socket buffers with a set of parameters [min, default, max]. The impact on network performance of the TCP socket buffer is similar to that of the general socket buffer. Nevertheless, the TCP socket buffer parameter has a higher priority than the socket buffer parameter for TCP connections. Moreover, the receiving TCP socket buffer is automatically adjusted by kernels [12] while the sending TCP socket buffer is tuned manually.

Ring Buffers. The ring buffer is the last buffer that data go through before they enter NICs and the first buffer that data go through after they come out from NICs. Small ring buffers can cause the starvation phenomenon, which occurs when the NIC driver wakes to pull packets off for transmission. Besides, when the ring buffer is empty, a transmission opportunity is missed, which reduces the throughput of the system [13]. Network operators may increase the size of the receiving ring buffer to reduce the starvation risk and ensure the high throughput. However, it requires more memory allocation and cause high queuing latency for packets. In some cases, oversized buffers may also cause low throughput (See Table I). Therefore, it is crucial to choose a proper size for the ring buffer according to different network environments.

TABLE I
THE SHADOWSOCKS PERFORMANCE COMPARISON OF HAND-TUNED SCHEMES WITHIN TWO REGIONS

Location	Setting	Ring	TCP socket	Socket	Throughput
ES&WE	default	256	65,536	212,992	161.32 Mbps
	tune-1	512	1,048,576	4,194,304	169.73 Mbps (+5.2%)*
	tune-2	256	87,380	67,108,864	217 Mbps **(+34.5%)**
	max	4096	67,108,864	67,108,864	172 Mbps (+6.7%)
ES&EA	default	256	65,536	212,992	116.4 Kbps
	tune-1	512	1,048,576	4,194,304	146.92 Kbps **(+26.2%)**
	tune-2	256	87,380	67,108,864	107.5 Kbps **(-7.6%)**
	max	4096	67,108,864	67,108,864	127.4 Kbps (+9.5%)

* Compared to the default setting

C. Impact of Buffer Size on Shadowsocks

To show that different buffer sizes take different impacts on Shadowsocks under different network environments, we deployed two Shadowsocks servers in two regions, i.e., the East Asia (EA) and the West (WE). The Shadowsocks client is in the East (ES). It is the typical deployment scheme in practice. We test four groups of hand-tuned buffer parameters in the network environment of the two regions. They are from the default buffer parameters in Ubuntu 18.04 (`default`), the third-party Shadowsocks deployment optimization scheme [14] (`tune-1`), the parameters suggested by the ESnet website from the U.S. Department of Energy [15] (`tune-2`), and the max buffer scheme maintaining maximum values (`max`), respectively. The results are shown in Table I.

As we can see, all the three tuning schemes increase buffer sizes compared to the default settings. However, there is a significant difference on performance gains. The tune-1 scheme performs the best when the server is located in EA and the clients are located in ES. The throughput significantly improves by 26.22% compared to the default scheme. When the server is located in WE, the throughput for the tune-1 scheme only improves by 5.21%. The performance of the tune-2 scheme on the two servers is just the opposite. It works the best in the WE. The throughput increases by 34.52% compared to the default scheme. However, it significantly decreases the throughput in EA compared to the default scheme. The max scheme is neither the best in both servers.

Based on the above results, we have the two observations:

- Different network environments (e.g., distance, routing, and censorship) require different buffer parameters for optimizing Shadowsocks.
- Enlarging the buffer size to the maximum values cannot guarantee the best performance for Shadowsocks.

Hence, buffer parameters need to be dynamically tuned based on different network environments to achieve the best performance of Shadowsocks.

III. AUTER DESIGN

Auter models multi-level buffer tuning as a decision-making procedure with an RL model. Figure 2 shows the system architecture, which deploys an RL agent in the user-space and interacts with the kernel and NIC driver via a series of system tools and APIs. The RL agent consists of three components: *network perception*, *decision making*, and *policy enforcement*. The network perception component collects a series of information about network performance states and transforms them as inputs to the decision-making component. The decision-making component calculates the rewards based on the performance states and generates the tuning actions periodically. Finally, the policy enforcement component maps the action policies into new buffer size values and tunes the configurations in the kernel and NIC driver.

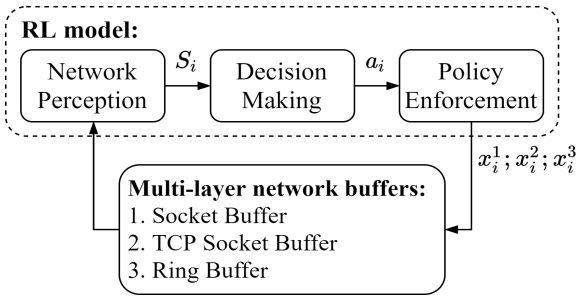


Fig. 2. The architecture of the Auter system.

A. Network Perception

The network perception component constructs a state vector S_i in the i-th RL iteration, which will be fed into the decision making component. It consists of the buffer state S_i^b and the current network performance state S_i^p. As we mentioned in Section II, Shadowsocks' network performance is significantly affected by ring buffer size l_i^r, socket buffer size l_i^s, and TCP socket buffer size l_i^t. Meanwhile, the current network performance can be reflected by throughput θ_i, latency β_i, and packet loss ℓ_i. Hence, we have the following definitions:

$$\begin{cases} S_i^p = [\theta_i, \beta_i, \ell_i] \\ S_i^b = [l_i^r, l_i^s, l_i^t] \end{cases}. \tag{1}$$

Due to network jitters and the burst feature of network flows, throughput, latency, and packet loss may vary significantly in a short time [16]. To avoid overreacting to network environments when adjusting parameters, we obtain θ_i, β_i, and ℓ_i using their average values for each flow. As indicated in [17], the average flow duration of TCP flows is 57.32 seconds. Hence, we calculate the average value per minute. The throughput θ_i is calculated as the total traffic amount sent in the current flow divided by its duration. The latency β_i is calculated as the average time difference between sending packets and receiving the corresponding acknowledgment (ACK) packets. The packet loss rate ℓ_i is calculated as the proportion of packets that need to be resent to the total packets in a TCP connection.

B. Decision Making

After obtaining the state information by the network perception component, the next step is to make a decision on tuning these three buffer parameters. Our model is trained using the popular DQN algorithm [18]. It has simple structure and rapid response, which is suitable to be deployed on a network server

with limited resources and makes tuning decisions quickly. The DQN agent includes two neural networks: the evaluation network and the target network. In the i-th iteration, the policy function $\pi(\cdot)$ generates the action policy a_i while the target network updates the Q function $Q(\cdot)$ that approximates accumulative reward function R_i, in which the basic unit is the reward function r_i. In each iteration, the reward function r_i is calculated according to the current state information and the optimization objective. Thus, r_i continuously accumulates during the iteration process to maximize the accumulative reward function $Q(\cdot)$.

Auter aims to improve the throughput and also takes into account the latency and the packet loss rate. The reward function thus should include all these three performance metrics, which is designed as follows:

$$r_i = x^\theta \cdot r_i^\theta + x^\beta \cdot r_i^\beta + x^\ell \cdot r_i^\ell. \tag{2}$$

Here, r_i^θ, r_i^β, and r_i^ℓ are throughput reward, latency reward, and packet loss reward, respectively. x^θ, x^β and x^ℓ are their corresponding weights. Note that $x^\theta > 0$, $x^\beta < 0$, and $x^\ell < 0$. This is because latency and packet loss actually are the penalty for the system compared to throughput. As throughput, latency, and packet loss have different orders of magnitude, we normalize raw values to obtain r_i^θ, r_i^β, and r_i^ℓ as follows:

$$\begin{cases} r_i^\theta = \dfrac{\theta_i - \theta_{min}}{\theta_{max} - \theta_{min}} \\ r_i^\beta = \dfrac{\beta_i - \beta_{min}}{\beta_{max} - \beta_{min}}, \\ r_i^\ell = \dfrac{\ell_i - \ell_{min}}{\ell_{max} - \ell_{min}} \end{cases} \tag{3}$$

where θ_{min} and θ_{max} are minimum and maximum values of the sampled throughput. However, due to the volatility of Shadowsocks networks, we need to eliminate outliers based on the three-sigma rule [19] before selecting θ_{min} and θ_{max}. β_{min}, β_{max}, ℓ_{min}, and ℓ_{max} are calculated in the same way.

By adjusting x^θ, x^β, and x^ℓ in Equation 2, the decision making component can set different performance optimization goals for different scenarios. For example, a higher throughput is crucial for file transfer protocol (FTP) service; hence we can raise x^θ in that case. However, low latency is required for interactive services such as video games; hence we can further decrease the value of x^β. As the performance expectation of Shadowsocks mainly focuses on throughput improvement, we set the requirement: $x^\theta > \max\left\{\left|x^\beta\right|, \left|x^\ell\right|\right\}$.

C. Policy Enforcement

Policy enforcement tunes the buffer parameters according to the policies generated by the decision making component. Considering the differences between the buffers and the potential side effects of buffer tuning, we design a differentiated tuning mechanism that sets the different scale, range, and time interval for different buffers.

1) Tuning Range and Scale: In each iteration, Auter tunes three buffers according to the action policy provided by the decision making component. However, these three buffer parameters can not be tuned at will since they have different tuning range and scale. We find that the adjustable range is limited by the operating system and hardware. For example, in our test machine (Ubuntu 18.04, 16GB RAM, e1000e NIC driver), the adjustable range of ring buffer size is from 64 to 4,096 packets. While the range of the TCP socket buffer is from 512 to 67,108,864 bytes, which is much larger than that of the ring buffer. Moreover, the relationship of the parameters between sender and receiver gives constraints. From our experiments, we find that the current connection may be interrupted if the sender's TCP sending buffer and the receiver's TCP receiving buffer meet a certain relationship. For example, when the receiver's TCP buffer is 16K, the sender's TCP buffer should be larger than 28K; otherwise, the connection would be cut off. Finally, frequent tuning can also interrupt the current connection for a period of time, especially for the ring buffers. Thus, we set different tuning range and scale for each buffer parameter and update them as follows:

$$\begin{aligned} &b_i = b_{i-1} + \alpha \times a_i, \\ &s.t. \quad b_i,\ b_{i-1} \in [b_{min},\ b_{max}], \end{aligned} \tag{4}$$

where b_i refers to the value of buffer size that is tuned at the i-th iteration. b_i is updated based on the last value b_{i-1} and the agent's output a_i. α is the scale factor, which is distinct for each buffer. We test several options for each scale factor and select the settings that can exhibit high performance throughout in our evaluations.

In addition, there are some special considerations for TCP socket buffer and socket buffer. As introduced in Section II, Linux kernels maintain default and maximum values for socket buffers, and minimum, default, and maximum values for TCP socket buffers. We dynamically tune their default and maximum values. Tuning the default value would decide the actual socket buffer size, while tuning the maximum value can avoid conflicts with the increased default value.

2) Tuning Interval: Besides the tuning range and scale, the tuning interval, which means how often to tune, should also be customized for each buffer. In order to meet the needs of timely response, tuning intervals in existing studies are mainly in millisecond-level [16], [20], [21], [8]. However, a tiny time interval is unsuitable for tuning buffers. First, frequent adjustments on ring buffers may cause side effects on the traffic. our test finds that a single tuning of ring buffer would interrupt a connection for about 2.1 seconds. Moreover, if we continuously and frequently tune the ring buffer, the connection would be unstable or even completely interrupted. Therefore, ring buffer size is not a parameter suitable for frequent tuning. Second, real-time adjustment at a millisecond or even sub-millisecond interval for heavy-load networks may generate high overhead and require very complex architecture [8]. Tuning parameters at a relatively long interval would be better for our system.

Thus, we update buffer parameters at the flow level, which means at least one flow is completed between two adjustments. In each iteration, the perception module collects network state from at least one unbroken flow, which can help us collect enough data to eliminate outliers. The tuning interval depends on the average duration and density of TCP connections. In our evaluation, the range of time interval is from 3s to 10s. Besides, considering the possible side effects of frequent tuning for ring buffers, we further slow down its tuning cycle.

Algorithm 1 shows the differentiated buffer tuning mechanism for the three buffers, which explains how the policy enforcement module tunes the three buffers based on the action policy $a_i = [a_i^r, a_i^s, a_i^t]$ and collects the new state vector $S_i = [S_i^p, S_i^b]$. The algorithm runs in a big loop from Step 1 to 17. In each round, it calculates the new buffer size and determines whether the size exceeds the adjustable range (Step 2-6), and then updates the buffer size. Such tuning is suitable for the socket buffer and TCP socket buffer. For the ring buffer, Steps 9-12 calculate the number of requests for the RL model to increase (cnt_+^r) or decrease (cnt_-^r) the ring buffer size. Meanwhile, we monitor the number of running connections (n_{conn}) in the system by Step 13. Ring buffer tuning meets two prerequisites: (1) the continuous requests to adjust ring buffer exceeds n times; (2) the number of existing connections is less than m, which are determined in step 14-15. Finally, we recollect the state vector $S_i = [S_i^p, S_i^b]$ with Step 16-17 as the new input of the decision-making module.

Algorithm 1 Differentiated Buffer Tuning Mechanism

Input: $a_i = [a_i^r, a_i^s, a_i^t]$, the policy generated by the RL agent
Output: $S_i = [S_i^p, S_i^b]$, the collected state information

1: **for** $i \leftarrow 1$ to T **do**
2: $\quad tmp_i^s \leftarrow b_{i-1}^s + \alpha^s \times a_i^s$
3: $\quad$ **if** $tmp_i^s \in [b_{min}^s, b_{max}^s]$ **then**
4: $\quad\quad b_i^s \leftarrow tmp_i^s$
5: $\quad$ **else**
6: $\quad\quad b_i^s \leftarrow b_{i-1}^s$
7: $\quad \cdots / * Similar\ work\ for\ b_i^t * /$
8: $\quad / * b^r\ monotonically\ increase\ or\ decrease * /$
9: $\quad$ **if** $a_i^r > 0$ **then**
10: $\quad\quad cnt_+^r \leftarrow cnt_+^r + 1$ **and** $cnt_-^r \leftarrow 0$
11: $\quad$ **else if** $a_i^r < 0$ **then**
12: $\quad\quad cnt_-^r \leftarrow cnt_-^r + 1$ **and** $cnt_+^r \leftarrow 0$
13: $\quad n_{conn} \leftarrow the\ number\ of\ running\ connections$
14: $\quad$ **if** $(cnt_+^r > n$ **or** $cnt_-^r > n)$ **and** $n_{conn} \leq m$ **then**
15: $\quad\quad b_i^r \leftarrow tmp_i^r$
16: $\quad S_i^b \leftarrow [b_i^r, b_i^s, b_i^t]$
17: $\quad S_i^p \leftarrow [\theta_i, \beta_i, \ell_i]$
18: $\quad$ **output** $S_i \leftarrow [s_i^p, S_i^b]$

IV. System Implementation

Auter system is implemented with Python 3.7, Tensorflow, and the python package *stable-baselines* [22]. The decision-making component is based on the DQN algorithm [18]. Specifically, Auter applies a basic architecture containing two hidden layers with $(32, 16)$ neurons and $tanh$ activation functions. The inputs and outputs of Auter are low-dimensional vectors so that the lightweight structure is adequate. Meanwhile, the lightweight structure can reduce response time and resource occupation.

We customize the environment template provided by *stable-baselines*. A typical RL model implementation using *stable-baselines* consists of two parts: the policy and the environment. The policy defines the decision-making mechanism, while the environment provides the interface with the target. Our customized environment integrates the network perception and tuning policy enforcement modules. Specifically, Auter observes the traffic on the server in real-time, tunes the buffer size, and generates state (S_i) and reward (r_i) in the environment template. We first capture and analyze packets via *PyShark* package. Next, we tune each buffer according to Algorithm 1. Finally, we modify network parameters at runtime with two tools: *ethtool* that tunes the ring buffer and *sysctl* that tunes the socket buffer and TCP socket buffer.

A. Hyperparameter Configuration

Table II shows the hyper-parameters in our DQN model. The parameters *n_actions* and *n_features* are decided by the system requirements without additional configuration. *learning_rate* would affect the convergence rate of the neural network and the default value is appropriate for our system. *reward_decay* (γ) is a discount factor of the reward, which could directly affect the training effect. In the training process of the network environment, the larger the discount factor is, the faster a stable policy will be learned in the DRL agent. Therefore, we select a large value of 0.9. Other parameters are related to the sample size during the training. As they have little effect on policy generation, we use the default values.

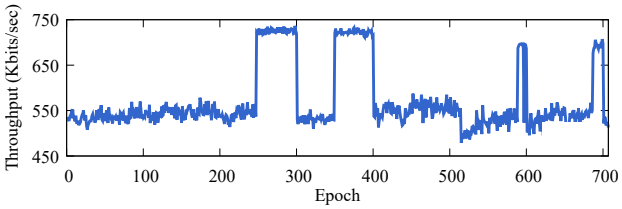
We also test each buffer parameter to determine the step size of tuning. We tune each buffer for 100 epochs with different step sizes. When the step size grows larger than 512 Bytes, the tuning effect is relatively stable. Therefore, we select the step size of 1024 Bytes for TCP socket buffer and socket buffer. For ring buffer, the throughput gain brought from tuning is not obvious since the tuning would result in jitters or even connection interrupts. Due to the relatively small tunable interval of ring buffer, we select the step size of 8.

B. Deployment and Training

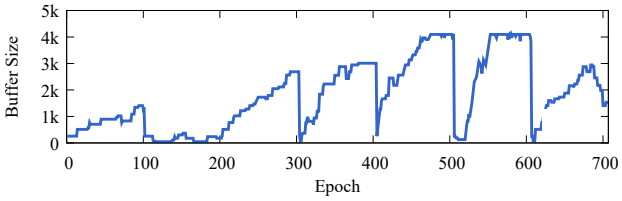
Shadowsocks system consists of two parts: a remote server (*SS_Server*) and a local client (*SS_Local*). *SS_Server*, which is responsible for request parsing and traffic forwarding, is the performance bottleneck of the whole system. Thus, we deploy our system on the *SS_Server*. To train our model, we deploy Shadowsocks system on two local machines (the local and target host) and a VPS (*SS_Server*) to simulate the unstable network environments through Traffic Control (TC) [23]. During the training, all buffer values are reset every 100 epochs to explore optimal values as much as possible. Besides, our training session lasts 4,500 epochs. Figure 3

TABLE II
HYPER-PARAMETERS OF OUR DRL MODEL

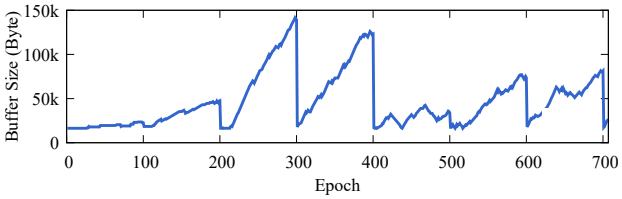
Parameter	Setting Value
n_actions	4
n_features	7
learning_rate	0.1
reward_decay (γ)	0.99
e_greedy (ϵ)	0.9
e_greedy_increment	None
memory_size	500
batch_size	32
replace_target_iter	50



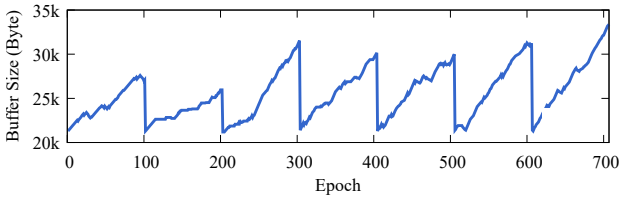
(a) Throughput in training phase.



(b) Ring buffer size in training phase.



(c) TCP socket buffer size in training phase.



(d) Socket buffer size in training phase.

Fig. 3. The training process of Auter.

shows the throughput and three buffer values during our training session. We find that the throughput performance rises and falls suddenly. Also, the knees of the throughput curve can correspond to the values of buffer parameters, indicating the optimal parameter range. There are two humps in the throughput curve from 250 epoch to 400 epoch, which corresponds to the peaks of each buffer in Figure 3 (b-d). This reveals that the reasonable values of the three buffers are the key to performance improvement. Besides, the throughput between 500 epoch and 700 epoch drops dramatically because of the low value of TCP socket buffer. And the continued sluggish throughput in turn makes the ring buffer decline, which shows there is a strong correlation between these buffers.

TABLE III
DEVICE CONFIGURATIONS OF OUR TESTBEDS

Type	Local Host	VPS 1	VPS 2
Number	2	3	1
CPU	E5-2620	E5-2673	E5-2673
Memory	16GB	1GB	1GB
Bandwidth	1 Gbps	1 Gbps	1 Mbps
OS	18.04 LTS	18.04 LTS	18.04 LTS
Provider	Dell	Azure	Tencent
Location	Local	ES (2), WE (1)	EA

V. PERFORMANCE EVALUATION

We conduct experiments under various network conditions to evaluate the system performance. We first evaluate the adaptability of Auter on a local emulated testbed, which is similar to the training environment. Two local machines work as the source and target hosts, and a VPS located at U.S. Eastern plays the role of a remote server. We use the TC tool to emulate the volatile properties of network environments.

We also deploy our system in a real cloud environment to verify its effectiveness. The real testbed consists of three deployment locations. The device configurations are shown in Table III. The *SS_local* is deployed on the VPS in local network, and three *SS_Servers* are deployed in the East (ES), the West (WE) and the East Asia (EA), respectively. to test the system performance under different network environments.

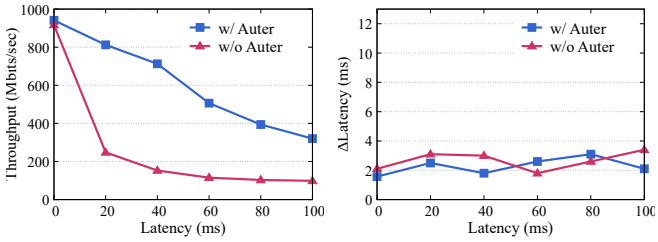
A. System Adaptability

On our local emulation testbed, we quantitatively control four link state metrics: bandwidth, latency, packet loss, and size of TCP flows. Then, we observe the performance of Auter with the change of each single state metric.

Under Different Latency. We verify the adaptability of Auter with different latency by sending fixed-size TCP flows in a single connection on the local testbed. Figure 4 shows that Auter achieves up to a 4.67 times increase in throughput with an extra simulated latency of 40 ms. The growth rate of throughput is the lowest without simulated latency since the bandwidth utilization approaches the topmost level and leaves less room for optimization. Meanwhile, as shown in Figure 4(b), the real delay (Δlatency) does not significantly increase with Auter. The difference between the delays with and without Auter is less than one millisecond on average.

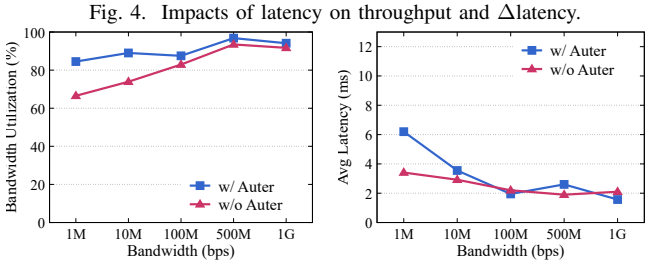
Under Different Bandwidth. We test five scenarios of different network bandwidths. As shown in Figure 5, the average bandwidth utilization increases by 11.84% with Auter. Besides, we notice that the bandwidth utilization is relatively low in the low bandwidth scenarios. In terms of latency, Auter introduces relatively high latency for links with low bandwidth. That is because our trained model prefers to give a policy of increasing the buffer size at most cases.

Under Different Packet Loss. Packet loss is the most common problem when the network link is unstable or congested. Therefore, we test five scenarios with different packet loss



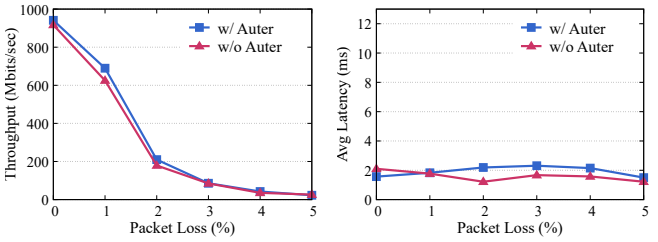
(a) Impact on Throughput (b) Impact on ΔLatency

Fig. 4. Impacts of latency on throughput and Δlatency.



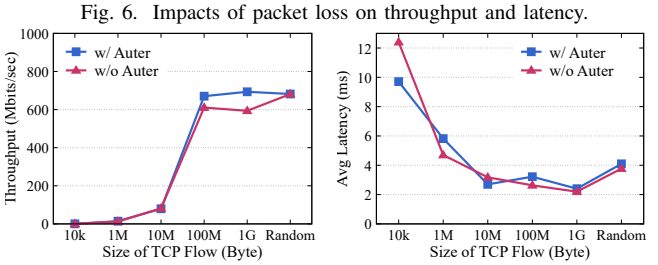
(a) Impact on BW. Utilization (b) Impact on Avg. Latency

Fig. 5. Impacts of bandwidth on throughput and latency.



(a) Impact on Throughput (b) Impact on Avg. Latency

Fig. 6. Impacts of packet loss on throughput and latency.



(a) Impact on Throughput (b) Impact on Avg. Latency

Fig. 7. Impacts of flow size on throughput and latency.

ratios in our experiments. Figure 6 shows that Auter only brings slightly throughput improvement. It is caused by the congestion control algorithm, which would quickly reduce the sending rate when packet loss is detected. When the sending rate is limited, it is hard to improve the performance only by tuning buffers. Besides, the latency remains stable. Auter does not add extra latency, compared with the original latency.

Under Various Sizes of TCP Flow. As different applications usually mean different workloads, we test five flow sizes and random flow combinations. As shown in Figure 7, Auter delivers general performance improvements and is more friendly to large size flows, except for the random flow combinations. This is because a single TCP connection of bulk file transfer would remain stable for a long time after it is established. Thus, the buffer remains stable after being tuned to an appropriate size. For the TCP flows with frequent establishment and short duration, the buffers and congestion windows can hardly adjust in time; hence the performance improvement is limited. Besides, Auter remains stable without introducing extra latency with multiple flows. We also find the average delay increases with the decrease of flow size. It is caused by frequent establishment of TCP flows and hence the performance remains the same with or without Auter.

Summary. The above experiments show that Auter is resilient to various network environments. Specifically, we observe that Auter can improve the throughput under these four network conditions by 296.77%, 11.84%, 10.23%, and 12.96% on average, respectively. Meanwhile, our method only brings no more than 0.8 ms extra delay in Shadowsocks networks. Thus, we conclude that Auter can perform consistently well within complex network conditions, especially for high latency networks where the average throughput increases by 296.77%.

B. Real World Experiments

To evaluate the effectiveness of Auter in practice, we deploy the system in three cloud data centers: The East (ES), the West (WE), and the East Asia (EA). Within the three regions, we compare Auter with existing optimization solutions, including baseline (default setting of Shadowsocks), BBR (the enhanced congestion control), and max buffer scheme [14]. Moreover, we run Auter to handle two traffic modes, i.e., short traffic and long traffic. The experimental results are shown in Table IV.

Comparisons of Different Deployment Locations. We compare the performance of our method with different transmission distances. In Table IV, the performance improvements brought by Auter in the three regions are 30.36%, 28.67%, and 2.07% on average, respectively. When deploying *SS_Local* in ES and *SS_Server* in WE or EA, Auter can bring significant gains. Meanwhile, it has minor impact when deploying the Shadowsocks system in a local region (ES & ES). The difference of performance improvement between different regions is derived from their varying ”lifting space”. When the Shadowsocks system runs without any optimization scheme (baseline), the bandwidth utilization in each region is 16.15% (ES & EA), 10.12% (ES & WE), and 90.72% (ES & ES), respectively. Higher bandwidth utilization means the parameter values in the current system are close to a reasonable range. Hence, the optimization solution cannot bring significant improvements.

Comparisons of Different Optimization Schemes. The existing Shadowsocks suites involve some performance optimization schemes, including adapting BBR instead of default congestion control algorithms and increasing the buffer parameters, e.g., maximizing buffer size [14]. We compare Auter with BBR and the max buffer scheme. As shown in Table IV, the gain brought by Auter is 20.33% on average. However, the max buffer scheme can only bring 8.64% gain on average. The max buffer scheme sometimes even brings negative gain. Such results are from the huge gap of buffer size between the sending and receiving end. Despite some fluctuations, Auter is still better than BBR, achieving 15.93% gain on average. Meanwhile, when combining Auter with BBR, the cooperation

TABLE IV
COMPARISON OF LATENCY AND THROUGHPUT IN REAL WORLD SHADOWSOCKS TESTBED

Traffic mode		Short traffic			Long traffic		
Region		ES & EA	ES & WE	ES & ES	ES & EA	ES & WE	ES & ES
Latency	Baseline	252.16 ms	63.9 ms	3.44 ms	267.3 ms	68.2 ms	2.84 ms
	BBR	243.88 ms	62.7 ms	2.91 ms	252 ms	66 ms	2.63 ms
	Max Buffer	261.3 ms	64.54 ms	2.73 ms	266.58 ms	65 ms	2.96 ms
	Auter	271.04 ms	64.22 ms	3.78 ms	281.12 ms	67.4 ms	2.94 ms
	Auter + BBR	266.14 ms	62.9 ms	2.99 ms	284.3 ms	67.7 ms	2.8 ms
Throughput	Baseline	186.4 Kbps	101 Mbps	947 Mbps	144.33 Kbps	106.3 Mbps	911 Mbps
	BBR	214.62 Kbps (15%)	109.2 Mbps (+8.1%)	**956 Mbps (+1.0%)**	193.68 Kbps (+34.2%)	144.4 Mbps (+35.8%)	923 Mbps (+1.3%)
	Max Buffer	166.32 Kbps (-9.7%)	117 Mbps (+15.8%)	944 Mbps (-0.02%)	169.06 Kbps (+17.1%)	137.1 Mbps (+28.9%)	930 Mbps (+2%)
	Auter	224.08 Kbps (+20.2%)	125 Mbps (+23.8%)	952 Mbps (+0.5%)	202.78 Kbps (+40.5%)	142 Mbps (+33.6%)	932 Mbps (+2.3%)
	Auter + BBR	**241.47 Kbps (+29.5%)***	**128.67 Mbps (+27.4%)**	954 Mbps (+0.7%)	**235.22 Kbps (+63.9%)**	**147.8 Mbps (+39.0%)**	**942 Mbps (+3.4%)**

* Compared to the baseline setting.

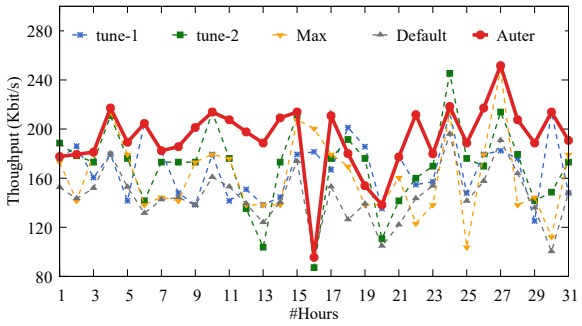


Fig. 8. Long time test for five schemes.

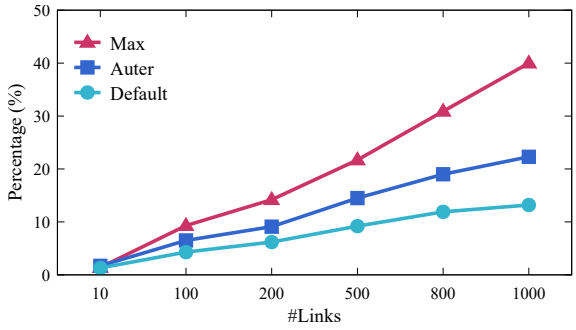


Fig. 9. Comparison of memory consumption.

brings the best performance in all the schemes tested, i.e., 28% on average. Thus, Auter can further improve network performance based on suitable congestion control algorithms.

Moreover, we notice that the delays in all these four optimization schemes are less than the median deviation of the real-time delay in each scenario. This means that even if no optimization solution is deployed, these delays may appear in the current network environment. Therefore, we believe that Auter would not bring additional delay burden.

Comparisons of Different Traffic Modes. The short traffic mode and the long traffic mode are emulated by random website visits and large file transfer, respectively. We find that processing long traffic is generally slower than processing short traffic in Shadowsocks, regardless of regional difference. This is mainly because long traffic usually lasts for longer time and requires higher network stability, which happens to be the weakness of Shadowsocks networks. However, it is worth noting that the performance improvement of our method in processing long flows (24.86% on average) is higher than that in processing short flows (14.83% on average). The *SS_Server* actually plays the role of an intermediate server in the Shadowsocks system for traffic forwarding. The network conditions on both sides of intermediate server are different due to the distance and censorship issues. For the forwarded traffic, the buffers in the network subsystem can alleviate the non-synchronization between the two sides. Therefore, Auter remits the volatility of the network to a certain extent by dynamically adjusting the buffer size.

Comparisons of Throughput for Long Running Time. We test the throughput of 5 schemes deployed between ES and EA within 31 hours, as shown in Figure 8. Basically, all the 4 schemes can improve the performance compared with the default configuration, while the Auter can bring the most performance gain. This is because static parameter configuration is not always optimal in a long period. The continuous tuning of Auter can help the system adapt to the real-time network environments.

Comparisons of Memory Usage. Memory consumption is critical to Shadowsocks since *SS_Server* is usually deployed on VPS with limited resources. We compare the memory usage in three configurations: the max buffer, Auter, and the default buffer size. In Figure 9, the traffic occupied by *SS_Server* inevitably rises with the increase in the number of connections. Although both max buffer and Auter increase memory consumption compared to the default configuration, the upward trend caused by the max buffer is more prominent than that of Auter. When the number of connections reaches 1000, the gap in memory usage has exceeded 15%. The higher memory usage is mainly caused by the larger socket buffer value, which increases the memory usage of a single link. It accelerates the memory consumption when the connection number and connection time increase. The max buffer scheme leads to the largest socket buffer increment. In contrast, Auter increases the socket buffer size to less than 1MB, consuming less memory.

Summary. We have the following observations from our experiments. First, Auter can be applied to improve the long-distance Shadowsocks applications with low bandwidth utilization. Second, it outperforms the max buffer scheme and can cooperate with the BBR to further improve performance.

Third, Auter is more effective for long flows. Fourth, Auter can stably outperform static schemes in long running time. Finally, it consumes less memory than the max buffer scheme.

VI. Discussions

Deploying Auter in Senders and Receivers. We deploy Auter at the *SS_Server*, which acts as the intermediate server in the entire communication. Actually, the parameters of both the sender and the receiver also have a significant impact on the performance of the networking subsystem. For example, the socket receiving buffer can affect the sending rate of the current session since the receiving rate is fed back to the sender by the ACK packets. Hence, we may consider to deploy Auter at both the sender and receiver simultaneously. However, the synchronous parameter tuning at both sides may bring additional overhead, which requires further study.

Extending Auter to Other Applications. Our work focuses on improving the performance of the Shadowsocks application in this paper. It is promising to extend Auter to enhancing other applications that work in unstable network scenarios, such as tunnel networks and high-load data center networks. Moreover, Auter may improve the performance of UDP flows by tuning the UDP socket buffer along with the ring buffers, and we will leave it for future work.

Jointly Tuning More Parameters. In this paper, as a proof of concept, our system jointly tunes the parameters for three network buffers. In complex network systems, there are more parameters that may affect the network performance [13]. As the overall performance is actually determined by the weakest one, it could be further improved if other parameters are tuned. For instance, there are 28 numerical parameters that can be tuned in the Linux kernel of version 5.3. To tune these parameters, we need to understand how they work and their potential side effects.

VII. Related Work

Network Performance Tuning. Parameter tuning is the basic work of network performance optimization [9], [24], [25]. Basically, researchers deeply analyze the networking subsystem and conclude the optimal parameter configurations based on the device status [26], [27]. However, they heavily rely on the operators' experience and domain knowledge. Therefore, self-tuning mechanisms emerge, such as byte queue limits [28] and sending window scaling [12]. Moreover, learning-based methods [7] bring automation and specificity into the TCP performance tuning task. Genetic algorithm is used to search the most suitable set of parameters for the network performance tuning [6]. Besides, automatic adjustment mechanism is designed based on the relation between a single parameter and system performance metrics [5].

The specificity of tuning scheme means the tuning can focus on specific performance metrics [29], [30], [31], applications [32], and usage scenarios [33]. It is hard to satisfy all performance requirements with a single collection of parameter settings. Red Hat develops a project called tuned [33], which provides a series of parameter combinations to adapt to specific usage scenarios, e.g., the wifi environment or the Ethernet connection. Besides, porting the network stack from kernel to user space can also optimize the entire network performance [34]. User-space network stack provides more adjustable space, benefiting to customize the network-related parameters for the application-specific workloads.

Applications of RL on Networking. In network, many optimization problems can be solved by reinforcement learning [35], such as routing [36], scheduling [8], and congestion control problems [20]. The routing problem is a global path optimization problem [36]. The optimal path can be determined via RL, where the state is the throughput between two ends and the reward is the mean delay in the selected two ends. The scheduling problem aims to maximize the forwarding efficiency for packages. To respond promptly, the scheduling is executed with the pre-determined policies, which are adjusted by RL agents [8]. Congestion control is actually a rate control problem. For the RL-based congestion control algorithms, researchers focus on designing a comprehensive reward function [20]. That is because the congestion control algorithms are based on one specific network state signal [37], which may lead to misjudgments and performance reduction.

In addition, RL is also utilized to solve network security problems. Existing studies show that RL can be used to defend against cyber-physical attacks in distributed networks and ad-hoc networks [38]. Also, to cope with jamming attacks, the RL agent can be leveraged to select a proper channel to send the signals, avoiding interference from jammers [39].

VIII. Conclusion

In this paper, we propose a network performance tuning system named Auter, which automatically and dynamically tune multi-layer network buffers to improve the performance of Shadowsocks. Our system contains three components: network perception, decision making, and policy enforcement. The network performance states are collected by the network perception. Based on these states, the tuning policies are generated by the decision making. Finally, the policy enforcement maps policies into new configurations. Due to the different impact patterns of these three parameters on network performance, we design a differentiated buffer tuning mechanism, which would tune each buffer with different scale, range, and interval. We implement and evaluate our system for multiple network scenarios on both the local testbed and cloud testbed. The results in the local testbed show that Auter is robust for various network scenarios, especially in the long-distance network. And the results on the cloud platform prove that our method not only outperforms stably the existing optimization scheme by 20.33% in long time, but also can cooperate with the BBR to further improve the throughput by up to 63.9% compared with the default configuration.

Acknowledgment. The research is partially supported by U.S. ONR Grants N00014-18-2893 and N00014-20-1-2407, U.S. NSF Grant CNS-1815650, NSFC Grant 62132011, and the Shuimu Tsinghua Scholar Program. Qi Li and Jiahao Cao are the corresponding authors.

REFERENCES

[1] Wikipedia contributors, "Shadowsocks wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/w/index.php?title=Shadowsocks&oldid=999060102.

[2] J. Beznazwy and A. Houmansadr, "How china detects and blocks shadowsocks," in *Proceedings of the ACM Internet Measurement Conference*, 2020, pp. 111–124.

[3] 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.

[4] Xiao Guoan, "How to install shadowsocks-libev proxy server on debian 10 vps," 2020, https://www.linuxbabe.com/debian/install-shadowsocks-libev-proxy-server-debian-10.

[5] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," in *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, 1998, pp. 315–323.

[6] B. Gembala, A. Yazidi, H. Haugerud, and S. Nichele, "Autonomous configuration of network parameters in operating systems using evolutionary algorithms," in *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems*, 2018, pp. 118–125.

[7] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, 2018.

[8] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 191–205.

[9] packagecloud, "Monitoring and tuning the linux networking stack: Receiving data," 2017, https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/.

[10] M. D. Leech, "SOCKS Protocol Version 5," Internet Requests for Comments, Tech. Rep. 1928, mar 1996. [Online]. Available: https://rfc-editor.org/rfc/rfc1928.txt

[11] T. Hruby, T. Crivat, H. Bos, and A. S. Tanenbaum, "On sockets and system calls: Minimizing context switches for the socket api," in *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, 2014.

[12] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," Internet Requests for Comments, Internet Engineering Task Force, RFC 1323, May 1992. [Online]. Available: {http://www.rfc-editor.org/rfc/rfc1323.txt}

[13] D. Siemon, "Queueing in the linux network stack," *Linux Journal*, vol. 2013, no. 231, p. 2, 2013.

[14] "Shadowsocks guide 2019 — install + configuration + optimization," 2019, https://fanqiang.network/637.html.

[15] "Linux Tuning," https://fasterdata.es.net/host-tuning/linux/.

[16] 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.

[17] M.-S. Kim, Y. J. Won, H.-J. Lee, J. W. Hong, and R. Boutaba, "Flow-based characteristic analysis of internet application traffic," in *Workshop Chair*.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[19] F. Pukelsheim, "The three sigma rule," *The American Statistician*, vol. 48, no. 2, pp. 88–91, 1994.

[20] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*, 2019, pp. 3050–3059.

[21] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1325–1336, 2019.

[22] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[23] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2017.

[24] packagecloud, "Monitoring and tuning the linux networking stack: Sending data," 2017, https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/.

[25] I. K. Center, "Network performance tuning," 2020, https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaag/wkvm/wkvm_c_tune.htm.

[26] W. Wu, M. Crawford, and M. Bowden, "The performance analysis of linux networking–packet receiving," *Computer Communications*, vol. 30, no. 5, pp. 1044–1057, 2007.

[27] B. H. Leitao, "Tuning 10gb network cards on linux," in *Proceedings of the 2009 Linux Symposium*. Citeseer, 2009, pp. 169–185.

[28] T. Herbert, "bql: Byte Queue Limits," 2012, https://lwn.net/Articles/469652/.

[29] E. Sert, C. Sönmez, S. Baghaee, and E. Uysal-Biyikoglu, "Optimizing age of information on real-life tcp/ip connections through reinforcement learning," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*. IEEE, 2018, pp. 1–4.

[30] N. Arianpoo and V. C. Leung, "How network monitoring and reinforcement learning can improve tcp fairness in wireless multi-hop networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2016, no. 1, p. 278, 2016.

[31] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter, "Expanding across time to deliver bandwidth efficiency and low latency," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 1–18.

[32] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, and J. Zhang, "Reducing web latency through dynamically setting tcp initial window with reinforcement learning," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 2018, pp. 1–10.

[33] "Project "tuned": Daemon for monitoring and adaptive tuning of system devices," Sep. 2020, https://github.com/redhat-performance/tuned.

[34] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 175–186, 2014.

[35] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[36] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1871–1879.

[37] K. A. Yadav and S. Kumar, "A review of congestion control mechanisms for wireless networks," in *2017 2nd International Conference on Communication and Electronics Systems (ICCES)*. IEEE, 2017, pp. 109–115.

[38] A. Ferdowsi, U. Challita, W. Saad, and N. B. Mandayam, "Robust deep reinforcement learning for security and safety in autonomous vehicle systems," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2018, pp. 307–312.

[39] G. Han, L. Xiao, and H. V. Poor, "Two-dimensional anti-jamming communication based on deep reinforcement learning," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 2087–2091.