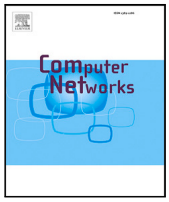
Computer Networks 237 (2023) 110047

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



SmartSBD: Smart shared bottleneck detection for efficient multipath congestion control over heterogeneous networks

Enhuan Dong [a,b,c], Peng Gao [d], Yuan Yang [e,b,f,*], Mingwei Xu [a,e,c,*], Xiaoming Fu [g], Jiahai Yang [a,b,c]

[a] Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
[b] Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China
[c] Quan Cheng Laboratory, Jinan 250103, China
[d] Department of Computer Science, Virginia Tech, VA 24060, USA
[e] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[f] Peng Cheng Laboratory, Shenzhen 518066, China
[g] Institute of Computer Science, University of Goettingen, Goettingen 37077, Germany

ARTICLE INFO

Keywords:
Multipath TCP
Congestion control
Shared bottleneck detection

ABSTRACT

Multipath TCP (MPTCP) has been widely adopted in today's mobile devices. However, two types of congestion control algorithms, uncoupled congestion control (Uncoupled CC) and coupled congestion control (Coupled CC), cannot achieve both bottleneck friendliness and throughput maximization for both of the MPTCP subflow bottleneck sharing scenarios, shared bottleneck (SB) scenario and non-shared bottleneck (NSB) scenario, leading to performance degradation in practice. In this work, we seek to enable efficient MPTCP congestion control, by alternating between Uncoupled CC algorithms and Coupled CC algorithms via smartly detecting whether the two MPTCP subflows share the same bottleneck link. We propose SmartSBD, the first learning-based data-driven approach for shared bottleneck detection, which is accurate, adaptable, and easy-to-deploy. SmartSBD is based on the key insight that the properties of subflows that share the same bottleneck often have similar trends of variation or similar values. In the training phase, SmartSBD collects system logs when MPTCP is running in real-world heterogeneous networks, extracts features, and trains a binary classifier. In the runtime phase, SmartSBD makes periodic predictions on the bottleneck sharing condition of live MPTCP subflows, and uses the prediction results to alternate between Coupled CC and Uncoupled CC. Our evaluations demonstrate that SmartSBD outperforms existing approaches.

1. Introduction

Nowadays, a large number of mobile devices (*e.g.,* smartphones and tablets) have two wireless interfaces, making it possible for mobile clients to access the Internet in multiple ways simultaneously [3]. To fully leverage this valuable feature, multipath TCP (MPTCP) has been developed and is currently under standardization by the Internet Engineering Task Force (IETF) [4]. Compared with regular TCP, MPTCP is able to employ multiple interfaces for one connection concurrently, achieving a higher throughput and a more robust connection. MPTCP has been adopted in a variety of industrial solutions, such as iOS [5], Huawei Link Turbo [6], and RHEL8 [7]. In the research community, numerous studies have been conducted on the multipath transport protocols [1,2,8–13]. As such, we can expect the growing popularity and a wide adoption of this type of technology in mobile devices in the near future. In this paper, we focus on the use case of MPTCP where two endpoints of MPTCP connections consist of a mobile device with two wireless interfaces and a normal server, which means each path between two endpoints is an Internet path with a wireless access link. It is a prominent use case of MPTCP [3], and the condition of the paths employed by MPTCP can be quite heterogeneous because of the different types of wireless links [14]. The *heterogeneous networks* in this paper consist of the heterogeneous paths formed by an Internet wired path with a wireless access link.

Congestion control (CC) plays a critical role in designing multipath transport protocols, given the objectives of throughput maximization and friendliness.[1] For MPTCP, a natural means is Uncoupled CC, which

* Corresponding authors.
E-mail addresses: dongenhuan@tsinghua.edu.cn (E. Dong), penggao@vt.edu (P. Gao), yangyuan_thu@tsinghua.edu.cn (Y. Yang), xumw@tsinghua.edu.cn (M. Xu), fu@cs.uni-goettingen.de (X. Fu), yang@cernet.edu.cn (J. Yang).

[1] We use the term "bottleneck friendliness" in this paper to describe the ability of MPTCP connections to fairly share bottleneck bandwidth with other TCP connections. Such an issue can be termed fairness [1], or TCP-fairness [2] in existing works. However, we use the "bottleneck friendliness" (or simply "friendliness") in this paper because it is more accurate and implies the issue is multipath-specific.

https://doi.org/10.1016/j.comnet.2023.110047
Received 25 June 2023; Received in revised form 14 September 2023; Accepted 27 September 2023
Available online 30 September 2023

Table 1
Goals of two types of congestion control (CC) algorithms in two bottleneck sharing scenarios: shared bottleneck (SB) and non-shared bottleneck (NSB).

	Uncoupled CC		Coupled CC	
	SB	NSB	SB	NSB
Throughput maximization	Yes	Yes	Yes	**No**
Bottleneck friendliness	**No**	Yes	Yes	Yes

enables each subflow of one MPTCP connection to use a TCP CC algorithm (normally, the standard TCP CC algorithm [15]) independently. Thus, each subflow competes for bandwidth as one TCP connection does, resulting in a high throughput. Unfortunately, Uncoupled CC cannot achieve friendliness in certain situations, *i.e.,* when two subflows of one MPTCP connection shares a common bottleneck link. This is because both subflows are aggressive, and the MPTCP connection will obtain more throughput than other TCP connections at the bottleneck link. To address the issue, researchers propose Coupled CC algorithms [8–12], which make each subflow less aggressive, forcing the entire MPTCP connection to compete for bandwidth similar to one TCP connection. Nevertheless, Coupled CC algorithms fail to maximize throughput when there is no shared bottleneck between the two subflows. An MPTCP connection with two subflows (created by two interfaces or two ports with the same interface), should achieve as much throughput as two single-path TCP connections do, if the two subflows do not share the same bottleneck. However, Coupled CC algorithms distribute the ability to compete for bandwidth to the individual subflows of one MPTCP connection, making each subflow less aggressive than a single-path TCP connection. Therefore, in such scenario, Coupled CC algorithms cannot obtain as much throughput as two single-path TCP connections do (maximum throughput). As such, neither of Uncoupled and Coupled CC algorithms can ensure bottleneck friendliness and throughput maximization together for both shared bottleneck (SB) scenario and non-shared bottleneck (NSB) scenario, as summarized in Table 1. For example, Ferlin et al. [1] show that Coupled CC in NSB scenario can induce a throughput reduction of up to 40% compared with Uncoupled CC.

In this work, we seek to enable both bottleneck friendliness and throughput maximization in MPTCP CC. Ideally, a two-path MPTCP connection should achieve as much throughput as two single-path TCP connections do, if the paths do not have the same bottleneck. On the other hand, an MPTCP connection should not harm other regular TCP flows when the two subflows share the same bottleneck.

Intrinsically, the reason why existing Coupled CC algorithms choose to be always conservative is that, MPTCP does not know whether the two subflows share the same bottleneck [8]. Thus, the key is to accurately detect whether the current scenario is SB or NSB. With such knowledge, smart CC algorithms that achieve both throughput maximization and bottleneck friendliness can be developed. For instance, one can simply use Coupled CC in SB scenario and use Uncoupled CC in NSB scenario. In this paper, we design and implement SmartSBD, a first-of-its-kind solution that enables efficient MPTCP CC achieving both bottleneck friendliness and throughput maximization via smart shared bottleneck detection, which periodically alternates between Uncoupled CC algorithms and Coupled CC algorithms according to current scenario detected.

Contributions and Novelty: We propose SmartSBD, a first-of-its-kind solution that enables efficient MPTCP CC achieving both bottleneck friendliness and throughput maximization via smart shared bottleneck detection. We list the contributions and novelty here and then detail them in the rest of this section.

1. We carefully select and develop the machine learning scheme that is *suitable* for shared bottleneck detection from end-hosts. Different from the customized rule-based approach employed in existing works [1,2,16], we propose a data-driven approach to detect shared bottlenecks accurately, *i.e.,* let the data determine what information to use when detecting sharing bottlenecks.
2. We develop and implement a *deployable* MPTCP CC scheme based on shared bottleneck detection results. As lots of existing works [1,2,16–33] have deployment issues, we propose SmartSBD as a pure end-host solution that only requires the standard MPTCP to modify the sender CC and does not need any modifications in networks.
3. We evaluate SmartSBD through comprehensive experiments, and SmartSBD *conforms to our design*.

SmartSBD is suitable for shared bottleneck detection from end-hosts: Since a bottleneck link influences the properties of subflows running through it, the properties of subflows that share the same bottleneck link often have similar trends of variation or similar values. As the end-hosts utilize subflow state variables to describe subflow properties, we first consider how to extract and exploit helpful information included in the subflow state variables to detect shared bottlenecks. However, it is challenging to solve that problem from end-hosts, since the information that the end-hosts can obtain is limited without the help of networks, not all subflow state variables are useful, and even if we know some subflow state variables are useful, how to use them is still an open issue. Though existing works [1,2,16] follow the customized rule-based approach, it is still a daunting task to manually define the rules that can lead to highly accurate shared bottleneck detection. *Our insight is to let the data make decisions. We use a data-driven approach, which can smartly select more important information from the subflow state variables.* Since supervised learning includes an offline training phase, which could provide decision basis for online runtime phases, we choose to use supervised learning algorithms as a first step towards smart shared bottleneck detection and leave other solutions (*e.g.,* online learning) to future work.

In particular, SmartSBD consists of a training phase and a runtime phase. In the training phase, SmartSBD first collects operating system logs from real-world network environments in which MPTCP is running. Next, SmartSBD parses the logs and extracts discriminative features that estimate the similarity of MPTCP subflows from various aspects (*e.g.,* latency trend). Then, SmartSBD trains a classifier based on the extracted features and ground-truth instance labels (SB or NSB). To improve model accuracy, reduce overfitting, and reduce model training time, SmartSBD removes irrelevant features via feature selection based on mutual information [34]. Our demonstration in the paper shows that not only RTT trends but also congestion state trends can be used to detect shared bottlenecks. Finally, SmartSBD performs model selection and hyperparameter tuning to select the best model. In the runtime phase, SmartSBD extracts features from a live MPTCP connection, and uses the trained classifier to make periodic predictions on the bottleneck sharing condition of its subflows. The prediction results are then used by SmartSBD to alternate between Coupled CC (for SB prediction) and Uncoupled CC (for NSB prediction).

SmartSBD is deployable:

To make SmartSBD *deployable, we develop it as a pure end-host solution that only requires the standard MPTCP to modify the sender and does not need any modifications in networks.* Further, SmartSBD *only requires modifications of MPTCP* CC, because it does not need any modifications to other behavior of MPTCP, *e.g.,* the patterns of sending packets or the definitions of header option fields. Moreover, since the CC is usually modularly implemented for transport layer protocols to support multiple CC algorithms, SmartSBD *can be lightly implemented as an MPTCP* CC *module*. Many existing works [1,2,16–33] suffer from deployment issues, and we detail them in Section 6.

We have implemented SmartSBD: ~1200 lines of `Python` code for the training phase, and ~1800 lines of `C` code in the MPTCP Linux network protocol stack [35], as an MPTCP CC module, for the runtime phase. To the best of our knowledge, SmartSBD is the first learning-based data-driven solution for shared bottleneck detection for efficient MPTCP CC, and the runtime phase is completely implemented

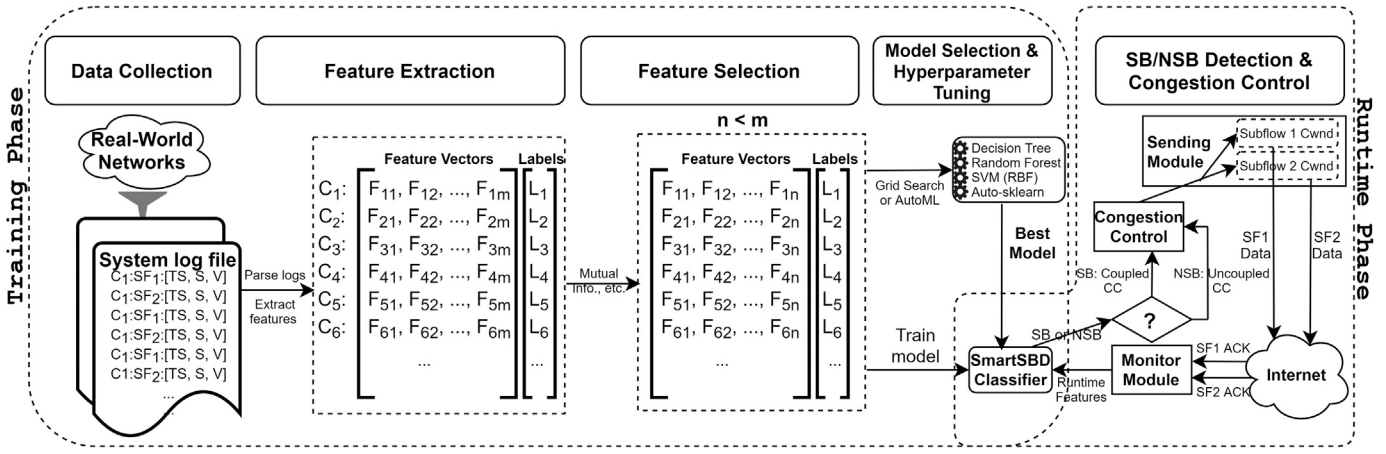


Fig. 1. The architecture of SmartSBD. SmartSBD collects system logs from real-world heterogeneous networks in which MPTCP is running. Each log entry indicates which MPTCP connection it belongs to (C_i denotes the ith MPTCP connection), which subflow it belongs to (SF_j denotes the jth subflow), when it was logged (timestamp, TS), which state variable is modified (S), and the new value of the state variable (V). After that, SmartSBD extracts discriminative features (F_{ij} denotes the jth feature of the ith MPTCP connection) for each labeled MPTCP connection instance (L_i denotes the label). The extracted features are then passed to feature selection and model training.

in the Linux MPTCP protocol stack. Note that it is still an open issue to implement existing data-driven MPTCP CC schemes [31–33] into operating system TCP/IP protocol stack (see Section 6.3).

SmartSBD conforms to our design: We demonstrate the practical efficacy of SmartSBD through a series of evaluations.

1. We evaluate the core of SmartSBD (*i.e.,* the trained classifier). Compared with existing approaches [1,2,16], SmartSBD achieves 47.5% (over [1]), 70.6% (over [2]), and 111.7% (over [16]) improvement in average accuracy (Section 5.1).
2. As it is reported that the performance of machine learning classifiers can drop sharply if they work on unseen datasets [36], we evaluate SmartSBD classifier on different unseen path conditions. The results show that it still obtains overall mean accuracy of 0.861 (Section 5.2).
3. To evaluate if SmartSBD could meet our design goals, *i.e.,* achieving both bottleneck friendliness and throughput maximization, we compare it with existing MPTCP CC algorithms. The results show that SmartSBD achieves 15.3%–123.7% throughput improvement in NSB scenario (Section 5.3.1) and keeps bottleneck friendliness in SB scenario (Section 5.3.2). Moreover, SmartSBD outperforms SBD [1], the existing scheme with accuracy closest to SmartSBD, in terms of adaptability to bottleneck shift (Section 5.3.3).
4. Finally, we evaluate the CPU overhead of SmartSBD. The results show that the overhead is minuscule (Section 5.4).

The rest of our paper is structured as follows. Section 2 shows the architecture of SmartSBD system. Section 3 presents the components of SmartSBD in detail. Section 4 introduces how SmartSBD is implemented. Evaluations are shown in Section 5. Section 7 discusses several issues about SmartSBD. Section 6 introduces related work. Section 8 concludes the paper.

2. SmartSBD Overview

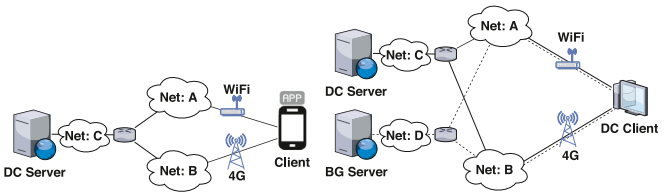
SmartSBD is a learning-based data-driven approach that enables efficient MPTCP CC achieving both bottleneck friendliness and throughput maximization via smart shared bottleneck detection. SmartSBD consists of two phases: an offline training phase and an online runtime phase. In the offline training phase, SmartSBD creates SB and NSB scenarios, runs MPTCP in them, obtains labeled datasets, and trains a classifier. In the online runtime phase, the classifier is used to detect shared bottlenecks.

Fig. 1 shows the overall architecture of SmartSBD, presenting more details. In the offline training phase: (1) SmartSBD collects system logs from real-world heterogeneous networks in which MPTCP is running, in both SB and NSB scenarios (built in advance by creating different bottlenecks), to trace three state variables of each MPTCP subflow (Section 3.1); (2) for each MPTCP connection, SmartSBD parses system logs, extracts discriminative features, and assigns a ground-truth label (SB or NSB) based on whether the logs are collected from SB or NSB scenario (Section 3.2); (3) SmartSBD ranks all features based on estimated mutual information [34] and removes irrelevant features with low ranks (Section 3.3); (4) to obtain the best model, SmartSBD performs model selection and hyperparameter tuning from a set of model candidates (Section 3.4).

In the online runtime phase, the monitor module records state variables and periodically extracts features from both live subflows. The trained classifier then makes predictions periodically based on the features. If the subflows are predicted to share the same bottleneck link, SmartSBD adopts Coupled CC for MPTCP CC. Otherwise, SmartSBD adopts Uncoupled CC (Section 3.5).

SmartSBD is a data-driven solution, and thus its classifier differs for different datasets used in the training phase. Before being used, SmartSBD needs its users to train classifiers first. In this paper, content providers are considered to be the deployers of SmartSBD, since they are the most direct users of SmartSBD, and care most about the network transport behavior. In recent years, many transport-related proposals [37–39] require their users to train before deployment, and their most likely users are the content providers. SmartSBD has the similar requirement. If content providers want to have a high-accuracy classifier with universal adaptability, they need to collect datasets that cover many path conditions. However, our learning-based data-driven approach is able to generate a high-accuracy classifier with size-limited datasets over different unseen path conditions (see Section 5.2). If the content providers want to obtain more specialized classifiers, they can group flows from the users with the same network attributes (*e.g.,* subnet, ISP, city), collect the data from each group, and generate a specialized classifier for each group. Such group-based deployment method has been adopted in many existing studies [40–43]. Like these works, SmartSBD also only requires the modifications at the sender, so the content providers can also deploy SmartSBD following the grouping way. How to deploy SmartSBD is relevant to the user scale and distribution of a content provider, which is beyond the scope of this paper.

In this paper, we mainly consider the shared bottleneck detection problem of two-subflow MPTCP, and the reasons are two-fold. First, the shared bottleneck detection problem of two-subflow MPTCP is a basis for the problem of MPTCP with 3 or more subflows. In Section 7,



(a) For content providers. (b) For our demonstration example.

Fig. 2. Network setups in data collection component. The wireless access links in these network setups are real, so they have wireless characteristics, *e.g.,* random loss.

we show that SmartSBD can be employed as a tool to solve that problem of MPTCP with 3+ subflows. Second, currently deployed MPTCP mainly works with two subflows. Today's mobile devices are commonly equipped with two interfaces (*e.g.,* all generations of iPhone), and each MPTCP connection usually creates two subflows through the two interfaces respectively.

3. Design of SmartSBD

SmartSBD consists of five components: (1) data collection, (2) feature extraction, (3) feature selection, (4) model selection and tuning, (5) runtime detection and congestion control. The data collection component collects system logs from real-world heterogeneous networks in both SB and NSB scenarios to trace three state variables of each MPTCP subflow (Section 3.1); The feature extraction component parses system logs, extracts discriminative features (Section 3.2); The feature selection component ranks all features and removes irrelevant features with low ranks (Section 3.3); The model selection and tuning component provides the best model from a set of model candidates (Section 3.4); The runtime detection and congestion control component records state variables and periodically extracts features from both live subflows, makes predictions based on the features, predicts whether the subflows share the same bottleneck link and adopts Coupled CC or Uncoupled CC based on the prediction results.

In this section, we describe each component of SmartSBD in detail. We also present an example as a demonstration for applying SmartSBD. The SmartSBD classifiers obtained in the example also represent SmartSBD to be implemented and evaluated in Sections 4 and 5. It is worth to note that since we collect datasets over heterogeneous networks in the example, the SmartSBD classifiers implemented and evaluated in our paper are generated for heterogeneous networks.

3.1. Data collection

Network Setup: To deploy SmartSBD, content providers need to collect data and train a SmartSBD classifier first. As shown in Fig. 2(a), content providers need to choose some of their servers as Data Collection (DC) Servers. Data collection can be done while their APP users require content. DC Servers need to have the ability to record the information of MPTCP connections. We have slightly modified MPTCP v0.94 [35] based on `printk` function, to enable DC Servers to record customized log entries into system log files. Note that we consider the clients with WiFi and cellular interfaces, making the paths used by MPTCP connections heterogeneous and making the setup conforming to the prominent use case of MPTCP [3]. As the bottleneck links mainly affect the behavior of CC [44,45], and they can be shared or non-shared for a two-subflow connection, the network setups shown in Fig. 2 support creating different types of bottleneck scenarios (SB or NSB).

Bottleneck Creation: In the training phase, SmartSBD requires labeled datasets, *i.e.,* each instance in the datasets has to be labeled by SB or NSB. In order to collect data with ground-truth labels, SB and NSB scenarios need to be created. For SB scenario, if the DC Server is rented from cloud platforms, SB can be created by limiting the DC Server's bandwidth through the console of the cloud platforms, and if the DC Server is located in a content provider's private networks, DC Server's bandwidth can be limited by configuring the switch that the DC Server connects to. Choose a small value for the DC Server's configured bandwidth to make it lower than the summed bandwidth of the WiFi and 4G links of the Client, and then the two subflows run through SB. Similarly, for NSB scenario, the content provider's APP can throttle WiFi and 4G links to some values with relevant system tools (`tc` or Wonder Shaper [46] for Linux) and the content provider can configure the DC Server's bandwidth larger than the summed bandwidth of the WiFi and 4G links. Then, the two subflows run through NSB links.

Bottleneck Validation: Although similar methods are applied in existing studies [1] to create bottlenecks, we argue that the bottleneck link(s) can still be different from what we expect. Let us consider the network setup shown in Fig. 2(a). When we try to construct SB scenario, we set the DC Server's bandwidth to some value. If the summed available bandwidth of Net:A and Net:B for our MPTCP connection is less than that value, the bottleneck links are still non-shared. For the NSB scenario, we configure WiFi and 4G links to two values, and if the available bandwidth of Net:C is less than the sum of the two values, the bottlenecks of both subflows are still the same one.

In order to mitigate the bottleneck mismatch problem, we introduce our method to validate whether the bottlenecks are configured as we expect. Our method to the bottleneck validation is: for SB scenario, the achieved throughput of the MPTCP connection has to be larger than $\alpha \cdot desired\_T$ $(0 < \alpha < 1)$; for NSB scenario, the achieved throughput of each subflow has to be larger than $\alpha \cdot desired\_T$. During data collection, the content provider needs to collect data from the same group of users many times under the same link configuration. We use the maximum achievable throughput of each subflow/connection in all transmissions for NSB/SB scenario in each group as the $desired\_T$. Only the validated MPTCP connections are included into our datasets.

State Variable Tracking: The path heterogeneity can be captured by subflow state variables. Since each subflow runs on a network path, the path condition impacts the subflow state variables. Thus, the state variables from different subflows are impacted by path heterogeneity. We keep record of three state variables (with timestamps) of each subflow:

1. RTT samples: When the subflow sender receives a new ACK packet, it obtains a new RTT sample.
2. Congestion state: The implementation of TCP CC in Linux employs a state machine to hold and change between multiple states to recover lost packets [47]. The implementation of MPTCP subflow inherits the congestion state machine [35]. A subflow can be in four states: `Open`, `Disorder`, `Recovery`, and `Loss`. The former two can be regarded as "uncongested" states and the latter two can be viewed as "congested" states.
3. ACK numbers of all ACK packets.

When the sender receives new samples of these state variables or it finds one of them is changing, SmartSBD records a log entry into system log files. These state variables estimate the similarity of MPTCP subflows from various aspects: (1) the RTT trends of subflows sharing the same bottleneck link may be similar, because their packets are both in the same queue at the bottleneck link and the queue delay introduced by the bottleneck link would be similar for the RTT samples of the two subflows; (2) the subflows that share the same bottleneck link may lose packets at almost the same time, and the congestion state can be regarded as a loss indicator; (3) the throughputs achieved by the subflows sharing the same bottleneck link may be similar, and ACK numbers can be used to estimate the throughput obtained by each subflow [48,49].

The state variables directly describing bottleneck link properties are quite limited at end hosts. Some other studies [49] also utilize congestion window (cwnd) size to extract features. We intentionally do not

Table 2
Information of Datasets. For each dataset, the # of samples in SB or NSB scenario is 500.

DC server location	BG server location	SB bandwidth	NSB bandwidths (WiFi, 4G)	Run time for each sample
Shanghai	Beijing	8 Mbps	7 Mbps, 3 Mbps	20 s
Guangzhou	Shanghai	8 Mbps	7 Mbps, 3 Mbps	20 s
Silicon Valley	Shanghai	8 Mbps	7 Mbps, 3 Mbps	30 s
Frankfurt	Shanghai	8 Mbps	7 Mbps, 3 Mbps	30 s

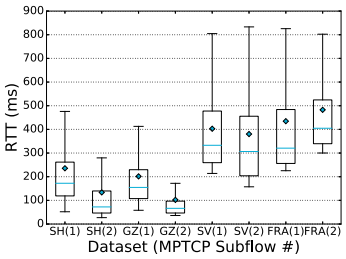


Fig. 3. RTT samples of the two subflows in all datasets. Subflow 1 runs on the 4G link, and subflow 2 runs on the WiFi link. The diamonds show mean values. The ends of the whiskers are set following the definition of Tukey boxplot [51] and we omit outliers.

use it because congestion window is influenced by the path properties (including bottleneck link properties) and CC algorithm both. We think the state variables that can directly show bottleneck link properties would be more helpful and the reached throughput (estimated from the ACK numbers) is better than the expected throughput (estimated from cwnd).

Data Collection for Our Demonstration Example: We set up our data collection infrastructure and collect data from real-world heterogeneous networks. Fig. 2(b) shows the setup for our demonstration example. The DC Server and background server (BG Server) are rented from public cloud platforms. The DC Client is a PC residing at a lab in Beijing. It can access a cellular network (4G LTE provided by China Mobile) and a public WiFi AP (provided by the lab) with a USB cellular modem and a USB WiFi modem respectively. Through the two interfaces, the DC Client can build a two-subflow MPTCP connection with the same server. Since each subflow runs on a different path, the heterogeneity of the paths impacts the connection. The network setup in Fig. 2(b) conforming to the prominent use case of MPTCP [3]. The DC Client establishes an MPTCP connection with the DC Server, which has two subflows, and, in the meanwhile, builds two TCP connections with the BG Server as background traffic running through WiFi and 4G links respectively. The MPTCP connection employs the standard MPTCP CC, LIA [8,9], and the TCP connections use the standard TCP CC, which we refer to Reno [15] in the paper.

All hosts run Ubuntu 16.04. The DC Client employs a stable release of the MPTCP Linux kernel v0.94 [35]. The DC Server runs our modified MPTCP to trace state variables into system logs. We use iPerf 3 [50] to create all MPTCP/TCP connections. The target bandwidths of the two background TCP connections between the DC Client and BG Server are limited to 1.5 Mbps and 3.5 Mbps. As shown in Table 2, the bandwidths of the bottleneck links are set to 1.5-8 Mbps, so we set the target bandwidths of background connections to the values of a similar order of magnitude. As for the bottleneck validation, we used $\alpha = 0.8$ during the data collection, which filtered out ~30% logs.

In our demonstration example, we would like to see whether our approach is powerful enough to build a classifier based on the datasets with very different path conditions. To this end, we deploy the DC Server in multiple locations to get such datasets: Shanghai and Guangzhou in Huawei Cloud [52], and Silicon Valley and Frankfurt in Alibaba Cloud [53]. Table 2 shows information of our datasets. We collect four datasets, and we use the DC Server locations as their names (*e.g.,* Shanghai dataset). The network path conditions of these datasets are quite heterogeneous. For example, Fig. 3 shows the boxplot of two paths' RTT samples in all datasets. As can be seen, RTTs are quite diverse in all datasets. Even in the same dataset, the RTTs of different paths are also very different. Specifically, the median RTT of subflow 1 is ~2.5X than that of subflow 2 in Shanghai dataset. Existing studies [1,2] show that if the variance between the RTTs from different paths becomes larger, the SB/NSB detection would be more difficult.

3.2. Feature extraction

We extract features that estimate the similarity between the two subflows. We use the traces after the two subflows have just entered States `Recovery` or `Loss` within a period of time (T) to compute features for each instance, because before that the subflows are not limited by any links. As for the choice of T, we employ 50 times the larger value of the minimum RTTs of the two subflows. Since the features are extracted in T during the training phase, our final trained model can also make classification every T. It is worth to note that such time period is much shorter than that is used in [1], which is about 175 times the larger value of the minimum RTTs of the two subflows (3.5 s for 20 ms RTT).

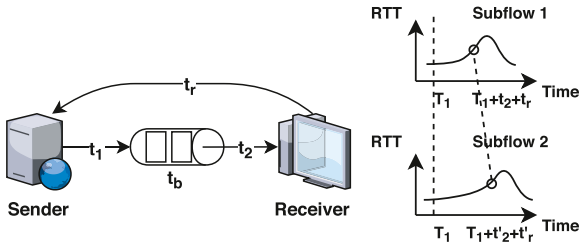
For each subflow of an MPTCP connection, we extract three time series from the traced three state variables. We extract an RTT time series directly from RTT samples. The congestion state is printed when the subflow sender changes it. We use "1" to represent "congested", and "0" to represent "uncongested". Then, we extract a binarized congestion state time series. As for ACK numbers, we compute a new achieved throughput sample from each new ACK through dividing the new acknowledged data amount (the difference between the ACK numbers of the new ACK packet and the previous one) by the inter-arrival time between the new and previous ACK packets. Moreover, we use the exponentially-weighted moving average (EWMA) to smooth the throughput samples, because these samples change quickly and dramatically over time. Using EWMA is also a common technique to smooth throughput samples [48,49], or relevant estimates (*e.g.,* ACK inter-arrival time).

We consider three metrics to characterize the similarity or relevance of these time series: (1) standard deviation (SD), (2) euclidean distance (ED), and (3) dynamic time warping distance (DTWD) [54]. Before we compute these metrics, we need to resample the time series with equidistant time intervals first. Computing these metrics based on resampled time series is reasonable for SD, and required for ED and DTWD. We employ the smaller value of the minimum RTTs of the two subflows as the time interval for resampled time series. Since the RTT and achieved throughput are changing and we only know their samples when we receive ACK packets, we leverage linear interpolation to obtain resampled RTT time series and achieved throughput time series. Different from these two series, we know the values of congestion state at any time during T. So the resampled congestion state can be directly obtained without linear interpolation.

Since SD quantifies the level of variation of a set of data, SD is a proper metric to characterize the similarity for each resampled time series. We use the ratio of the SD values to measure the difference between the two subflows. In time series analysis, ED and DTWD are two well-known metrics to quantify the similarity between time series. They have lots of application scenarios: image processing [55], IoT [56], stock price [57], *etc*. Although ED is much easier to compute, DTWD is more robust to dilatations or shifts across the time dimension

Table 3
All initial features that we extract from each sample. For all ratios, we use the value of the subflow running on WiFi path as the denominator. The rightmost two columns show the mutual information and relevant ranking of all initial features in our example.

Initial features		Abbreviation	MI value	Rank
Distances between one-dimensional time series	ED between subflow RTT time series	RTT_ED	0.1694	3
	ED between subflow congestion state time series	CState_ED	0.1325	5
	ED between subflow achieved throughput time series	Thpt_ED	0	18
	DTWD between subflow RTT time series	RTT_DTWD	0.1830	2
	DTWD between subflow congestion state time series	CState_DTWD	0.1018	9
	DTWD between subflow achieved throughput time series	Thpt_DTWD	0.0049	17
Distances between two-dimensional time series	ED between subflow (RTT, congestion state) vector time series	RTT_CState_ED	0.1281	7
	ED between subflow (congestion state,achieved throughput) vector time series	CState_Thpt_ED	0.0922	10
	ED between subflow (RTT, achieved throughput) vector time series	RTT_Thpt_ED	0.0476	15
	DTWD between subflow (RTT, congestion state) vector time series	RTT_CState_DTWD	0.1503	4
	DTWD between subflow (congestion state,achieved throughput) vector time series	CState_Thpt_DTWD	0.0852	12
	DTWD between subflow (RTT, achieved throughput) vector time series	RTT_Thpt_DTWD	0.0078	16
Distances between three-dimensional time series	ED between subflow (RTT, congestion state, achieved throughput) vector time series	RTT_CState_Thpt_ED	0.0587	14
	DTWD between subflow (RTT, congestion state, achieved throughput) vector time series	RTT_CState_Thpt_DTWD	0.0616	13
Other	The ratio of minimum RTT of subflows	Min_RTT_Ratio	0.1320	6
	The ratio of SD of subflow RTT time series	RTT_SD_Ratio	0.2638	1
	The ratio of SD of subflow congestion state time series	CState_SD_Ratio	0.1164	8
	The ratio of SD of subflow achieved throughput time series	Thpt_SD_Ratio	0.0901	11



(a) We break up RTT into four parts. (b) RTT sample curves.

Fig. 4. A case to show DTWD would be useful for measuring the similarity between two RTT time series. Two data packets from different subflows of the same connection are queued at the bottleneck together, and almost simultaneously leave the same bottleneck link at T_1. We hope their RTT samples (circles in Fig. 4(b)) be aligned to each other before distance calculation.

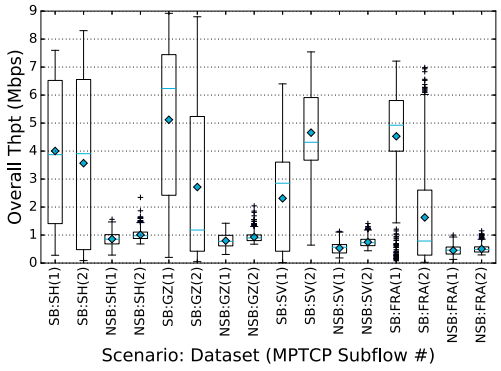


Fig. 5. Overall achieved throughput during *T*. subflow 1 runs on the 4G link, and subflow 2 runs on the WiFi link.

than ED [58]. DTWD is widely used in automatic speech recognition, speaker recognition, or online signature recognition applications [54]. Comparing with ED, the main improvement of DTWD is that it computes the best global alignment between two time series before distance calculation. As the dilatations or shifts across the time dimension also exist in the time series we are focusing on, DTWD is also suitable for measuring the similarity in our problem.

Let us take RTT time series as an example. Consider the case shown in Fig. 4. As shown in Fig. 4(a), RTT has four components, t_1, t_2, t_b, and t_r. While t_b is dominated by the queue length at the bottleneck link, t_1, t_2, and t_r can change over time because of the noise introduced by the devices along the path.

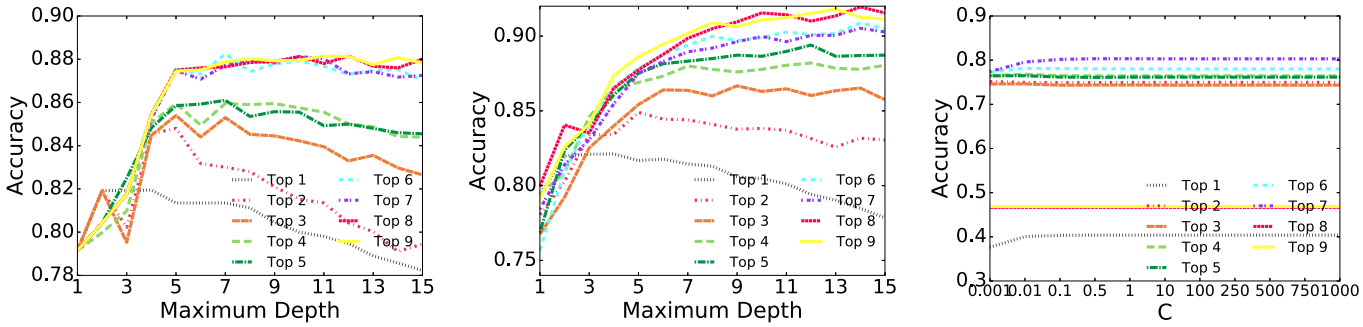
To understand why DTWD helps measure such similarity, consider the scenario that one data packet from subflow 1 and another data packet from subflow 2 are queued at the bottleneck together and almost simultaneously leave the same bottleneck link at T_1. Their related ACK packets would arrive at the sender at $T_1 + t_2 + t_r$ and $T_1 + t'_2 + t'_r$, respectively. When we measure the similarity of both trends, we hope that the two RTT samples obtained from these two ACK packets are aligned with each other, as shown in Fig. 4(b), because they both contain the information of the queuing delay of the bottleneck link at the same moment. When $t_2 + t_r = t'_2 + t'_r$, the two RTT samples are aligned with each other automatically. ED is sufficient to measure the similarity between the time series that satisfy $t_2 + t_r = t'_2 + t'_r$ for all samples. However, when $t_2 + t_r \neq t'_2 + t'_r$ or the values of t_2, t_r, t'_2, and t'_r are influenced by the path noise, we need to first find a best global alignment between two time series before distance calculation, which makes DTWD more suitable.

Since the data point in a time series can be scalar or vector, we combine the three one-dimensional time series into four multi-dimensional time series, as shown in Table 3. We also use the ratio of subflow minimum RTT as a feature since it is a basic property of a subflow.

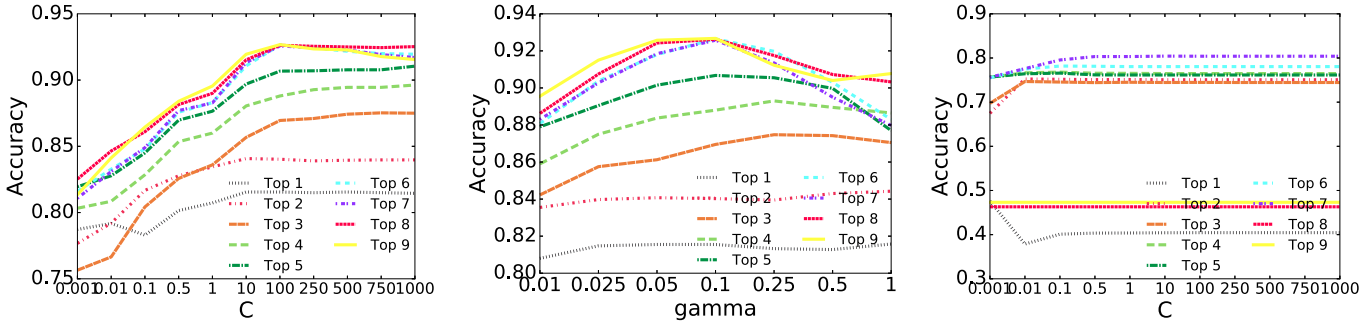
3.3. Feature selection

We have extracted the initial features that we think can be used to estimate the similarity between subflows. However, we do not know whether they are really all useful for our problem and relevant to the target labels. In order to filter out irrelevant features, we score all the initial features by calculating mutual information (MI) between each of them and the target label. Since the target label is discrete and each feature is continuous, their MI is defined as $I(X,Y) = \sum_y \int \log \frac{\mu(x,y)}{\mu(x)p(y)} dx$, where Y is the target label, X is a single feature, $p(\cdot)$ is a probability mass function, and $\mu(\cdot)$ is a probability density function. To estimate the MI, we employ the nearest-neighbor method proposed in [34]. Then we filter out the features with low ranks based on MI estimates.

We apply such method to our example, and the MI estimates are shown in the Table 3. As we can see, all the features that involve the achieved throughput have low ranks, and feature Thpt_ED is even totally independent with the target label. We explore the reason why the features related to the achieved throughput have low ranks. We



(a) Decision tree learning. (b) Random forest learning with 10 trees. (c) SVC learning with linear kernel.



(d) SVC (RBF) learning ($\gamma = 0.1$). (e) SVC (RBF) learning ($C = 100$). (f) Logistic regression learning with liblinear solver.

Fig. 6. Grid search results (mean accuracy) of multiple supervised learning models for top X feature subsets.

suspect that maybe EWMA is not a suitable method to estimate average achieved throughput. To test our hypothesis, we compute overall throughput for each instance during T based on the first and last ACK numbers and their timestamps. The statistical result is shown in Fig. 5. Compared with the overall throughputs achieved in SB scenario, the overall throughputs achieved in NSB scenario have a smaller range of variation. Through further exploration, we find that the default CC algorithm, LIA, cannot make the two subflows share the same bottleneck link evenly in SB scenario. The difference between the overall throughputs of the two subflows can be quite large in SB scenario, which differs from what we expect in Section 3.1. On the other hand, for NSB scenario, the overall throughputs of the two subflows are mainly influenced by the bottleneck bandwidths configured by us. Therefore, the subflows' achieved throughputs can be seriously impacted by an MPTCP CC algorithm (in SB scenarios), or bottleneck bandwidths (in NSB scenarios). Since we do not want our classifier to learn these information, we decide not to use any of the features that involve the achieved throughput.

3.4. Model selection and hyperparameter tuning

To train a good classifier, content providers need to select a classification model and tune its hyperparameters. Two kinds of approaches can be applied to accomplish these tasks: manual approaches and automated approaches. Manual approaches are more traditional. For our problem, content providers can enumerate possible feature combinations based on the feature ranking and compare model candidates with multiple hyperparameters using Grid Search [59], a common approach. Then they could select well-performed models according to comparison results. Comparing with manual approaches, automated approaches do not need human input, but the output of them may be more complicated. In our paper, we employ a famous AutoML system, auto-sklearn (AUTO) [60], which outputs an ensemble of up to 15 kinds of models. Next, we will describe how we apply these approaches in our example.

Table 4
Hyperparameter spaces of the learning models explored in Grid Search. Selected models and parameters are underlined.

Learning model	Hyperparam.	Parameter space
Decision tree	Max depth	1,2,...,**10**,...15
Random forest	Max depth	1,2,...,**10**,...15
SVM with linear kernel	C	x-axis tick values of Fig. 6(d)
SVM with RBF kernel	C	x-axis tick values of Fig. 6(d) (**100** is selected)
	γ	0.01, 0.05, **0.1**, 0.25, 0.5, 1
Logistic regression	Solvers	Liblinear, lbfgs
	C	x-axis tick values of Fig. 6(d)

Grid Search: Following the feature ranking in Table 3, we construct 9 feature subsets with top ranked features. Then, we employ Grid Search (specifically GridSearchCV in [59]) to compare the mean accuracy of multiple classification models with different hyperparameters. Well-known models like decision tree (DT) [61], random forest (RF) [62], SVM [63,64], and logistic regression [65] are explored. Table 4 shows the hyperparameters and their spaces considered in Grid Search. The hyperparameters considered in the Table 4 are essential to the performance of the models. The maximum depth for DT and RF, C for SVM and logistic regression, and γ for SVM (RBF) are used to reduce the risk of overfitting [59,61–65]. The overfitting means that the trained model behaves well on the training data, but it does not generalize on other data (*e.g.,* test data). It is usually caused by high complexity relative to the amount and noisiness of the training data. To avoid overfitting, the mentioned hyperparameters should be carefully tuned to make classifiers perform well. In addition, the solver used by the logistic regression also directly impacts the performance [59,65].

We show the grid search results in Fig. 6. For DT and RF, the mean accuracy becomes close to 90% when the maximum depth is

higher than 6 (Figs. 6(a) and 6(b)). We tried to enlarge the number of trees for RF, but the gain was not significant. For SVM (linear), the maximum mean accuracy is only around 80% (Fig. 6(c)). For SVM (RBF), we find $C = 100$ and $\gamma = 0.1$ achieves the highest mean accuracy. We fix one of them and change the other one to obtain the mean accuracy curves (Figs. 6(d) and 6(e)) for different feature subsets. The results of logistic regression are quite like the results of SVM (linear). Its maximum mean accuracy is also only around 80% (Fig. 6(f)). We have tried both liblinear [64] and lbfgs [66] solvers and their results are similar. The mean accuracy of the logistic regression model with liblinear solver is shown in Fig. 6(f). Among the model candidates, the SVM (RBF) achieves the highest mean accuracy, which is about 92%.

As a result, DT, RF, and SVM (RBF) are more suitable for our example. The reasons are (1) these models can easily reach 90% mean accuracy, and (2) these non-linear models can learn the non-linear decision boundary and generalize well in our datasets. Moreover, we find that enlarging feature subset from top 8 to top 9 did not bring obvious improvement, so we decide to use top 8 features for our demonstration example. For each learning model, we select the configuration of hyperparameters that lead to the best mean accuracy, as shown in Table 4.

AutoML: AUTO [60] does not require people to provide a search space of models and hyperparameters. It automatically builds an ensemble classifier consisting of up to 15 types of models. With more time to train, the output of AUTO can be more accurate. We set the training time to 10 h and use 6 CPU cores (Intel Xeon E5-2603 v4 1.70 GHz) to train it for our example. AUTO is a little better than SVM (RBF). The detailed results are shown in Table 5.

3.5. Runtime phase

Fig. 1 depicts the runtime phase. SmartSBD works when all subflows are in the congestion avoidance stage. SmartSBD has a monitor module recording state variables (with timestamps) of each subflow every T. At the end of each T, the monitor module first computes the features and invokes the classifier. Then, SmartSBD may change the MPTCP CC algorithm to be used in the next T.

4. Implementation

For the training phase, we have implemented SmartSBD with ~1200 lines of `Python` code. We used the model implementations in `python-sklearn` [59] and `python-autosklearn` [60] to train classifiers with all selected models (DT, RF, SVM and AUTO), configured with tuned hyperparameters if applicable. We employ the python class SelectKBest to select features, and we use the python classes, DecisionTreeClassifier, RandomForestClassifier, SVC, and AutoSklearnClassifier, implemented in sklearn.tree, sklearn.ensemble, sklearn.svm, and autosklearn.classification modules to train the selected models respectively. The trained classifiers are saved in some files to be used in the runtime phase.

For the runtime phase, we have implemented SmartSBD (DT) and SmartSBD (RF) with ~1800 lines of `C` code. The implementation of MPTCP in the Linux kernel [35] supports a framework to realize a new CC scheme as a Linux kernel module. Then, we implement SmartSBD (DT) and SmartSBD (RF) as a new MPTCP CC Linux kernel module in MPTCP v0.94 [35], which is based on Linux kernel v4.14. The kernel module reads the classifiers from the mentioned files during module loading, making classifier updates only require modifications of these files without module recompilation. The monitor module of SmartSBD is implemented based on two hook functions: `pkts_acked` and `set_state`. SmartSBD makes classification every T. The classification logic is implemented in `pkts_acked` function. The CC logic is implemented in `cong_avoid`, which adjusts congestion windows according to the MPTCP CC algorithm currently used by SmartSBD. Our implementation employs Reno [15] on each MPTCP subflow (termed "MP Reno") for Uncoupled CC, and LIA [9] for Coupled CC. We also tried other alternatives, such as OLIA [10], but the results of the experiments in Section 5.3 are similar, so we still use LIA, the only MPTCP CC documented in an RFC [8], for Coupled CC.

We tried to implement the classifiers based on SVM or AUTO into Linux MPTCP, however, we have not found a method to support the exponentiation of floating point numbers in Linux kernel modules, which is required by SVM and AUTO classifiers. MPTCP CC schemes have to be implemented as Linux kernel modules, but Linux kernel modules do not support the exponentiation of floating point numbers. Usually, to support the computation of decimal numbers in Linux kernel modules, these numbers are scaled up to save in some integer data types (*e.g.,* int). Thanks to the scaling method, the four basic operations can be supported, however, the exponentiation still cannot be done. Therefore, we only brought SmartSBD (DT) and SmartSBD (RF) into the Linux MPTCP implementation. Moreover, we also implemented SmartSBD (DT) and SmartSBD (RF) into ns-3 [67] to support the evaluation in Section 5.2.

5. Evaluation

In this section, we compare the performance of SmartSBD with existing schemes from multiple perspectives. First, we compare the classifier of SmartSBD with simulated existing shared bottleneck detection schemes [1,2,16] in Section 5.1 on real-world traces (the datasets collected in Section 3.1). Second, to evaluate the accuracy of SmartSBD on different unseen path conditions, we build emulated networks (for Linux implementation) and simulated networks (for ns-3 implementation) with Mininet and ns-3 to assess SmartSBD in Section 5.2. Third, we compare SmartSBD to other MPTCP CC with the network setup in Fig. 2(b) to show SmartSBD can achieve both throughput maximization and bottleneck friendliness in Section 5.3. Finally, we evaluate the CPU overhead of SmartSBD in Section 5.4.

5.1. Accuracy comparison on our demonstration example

As the implementations of existing schemes [1,2,16] are not available, we develope a simulation program to simulate them and obtain their accuracies of our demonstration example. The datasets collected in Section 3.1 include system logs from running MPTCP flows. Our simulation program reads the logs, simulates the detection processes of these schemes and outputs detection results.

Schemes Compared:

- **SmartSBD**: We employ stratified shuffle split cross-validation to evaluate these classifiers. We repeat the shuffling & splitting 10 times for each model, and use 10% datasets as the test set for each turn. For each dataset, we also calculate the mean accuracy of the instances from the same dataset in the test set.
- **DWC** [2]: Loss and delay are adopted as signals to detect shared bottlenecks. We simulate DWC following [2]. For each SB/NSB sample, we run the DWC to compute the time ratio during which the subflows are in the same set/different sets as the accuracy results.
- **SBD** [1]: One-way delay is used as the signal to detect shared bottlenecks. We simulate SBD following [1]. Since we know the subflows are traversing bottleneck(s), we omit the inference in SBD about whether the subflows are transiting the bottleneck(s). The recommended thresholds used in [1] are also employed in our simulations. We use ten observations in the last 3.5 s of each sample to classify.

Table 5
Mean accuracy comparison. The results of each SmartSBD scheme are computed according to the accuracy on the test set. On the contrary, DWC, SBD and TON20's accuracies are obtained from all instances.

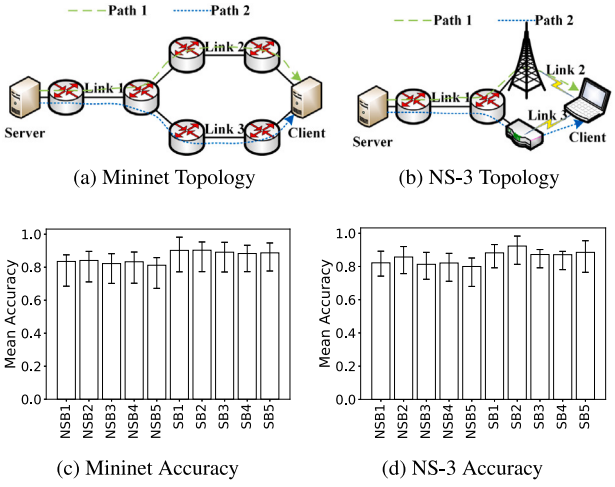
Dataset	Scheme	Mean accuracy of both scenarios	SB	NSB
SH	SmartSBD (DT)	0.974	0.957	0.992
	SmartSBD (RF)	0.981	0.966	0.996
	SmartSBD (SVM)	0.977	0.976	0.979
	SmartSBD (AUTO)	0.990	0.984	0.996
	SBD	0.615	0.728	0.502
	DWC	0.595	0.310	0.880
	TON20	0.376	0.267	0.485
GZ	SmartSBD (DT)	0.955	0.939	0.971
	SmartSBD (RF)	0.957	0.930	0.985
	SmartSBD (SVM)	0.966	0.956	0.975
	SmartSBD (AUTO)	0.968	0.960	0.976
	SBD	0.732	0.750	0.714
	DWC	0.625	0.480	0.770
	TON20	0.454	0.461	0.447
SV	SmartSBD (DT)	0.807	0.794	0.819
	SmartSBD (RF)	0.854	0.866	0.842
	SmartSBD (SVM)	0.907	0.884	0.930
	SmartSBD (AUTO)	0.933	0.930	0.935
	SBD	0.635	0.820	0.450
	DWC	0.535	0.160	0.910
	TON20	0.473	0.154	0.793
FRA	SmartSBD (DT)	0.848	0.809	0.887
	SmartSBD (RF)	0.884	0.864	0.904
	SmartSBD (SVM)	0.900	0.850	0.950
	SmartSBD (AUTO)	0.937	0.907	0.967
	SBD	0.613	0.716	0.510
	DWC	0.490	0.120	0.860
	TON20	0.503	0.183	0.823
ALL	SmartSBD (DT)	0.896	0.875	0.917
	SmartSBD (RF)	0.919	0.907	0.932
	SmartSBD (SVM)	0.937	0.916	0.959
	SmartSBD (AUTO)	0.957	0.945	0.969
	SBD	0.649	0.754	0.544
	DWC	0.561	0.268	0.855
	TON20	0.452	0.266	0.637

- **TON20** [16]: Loss and ECN marks are adopted as signals to detect shared bottlenecks. However, as stated in Sections 1 and 6, ECN may not be enabled for the current Internet. Therefore, we simulate TON20 following [16], except that we only use packet loss signals. For each SB/NSB sample, we run the TON20 to compute the time ratio during which the subflows are in the same Final Judgement state set/other different sets as the accuracy results.

The mean accuracy results are shown in Table 5. SmartSBD achieves an overall accuracy of 0.896–0.957 for different models. Compared with SBD, DWC and TON20, SmartSBD improves the overall accuracy by 47.5%, 70.6% and 111.7% respectively on average. DWC only uses loss signals to trigger group process. However, as we have shown in Table 3, the distances between subflow congestion states are less powerful than the distances between subflow RTTs. This means, if the subflows sharing the same bottleneck do not lose packets together in a short time, DWC would believe that the subflows are running on distinct bottlenecks even if the distances between subflow RTT time series are very small. As a consequence, DWC tends to believe that the subflows are running on distinct bottlenecks as can be seen in Table 5. Our datasets are collected from real-world networks, which is more challenging and different from the ns-2 simulation networks used in [2]. SBD is better than DWC because it uses delay based signals instead of loss signals. As depicted in Table 3, delay signals are much more powerful than loss signals. However, the accuracy of SBD in our simulations is much lower than that in [1]. The reasons are two-fold. First, wireless links are less reliable and more dynamic than wired links [14]. SBD is only evaluated over the real *wired* networks in [1]. Second, there is a greater difference between path RTTs in our demonstration. For instance, the median RTT of subflow 1 is ~2.5X than that of subflow 2 in Shanghai dataset, however, SBD is only tested with similar RTT paths in [1]. The simulated TON20 only utilizes the loss signals. Without the help of other signals, TON20 is not robust enough on our datasets. As we have stated before, the two paths in our demonstration are quite diverse, and the results show that TON20 cannot handle such situations. In conclusion, SmartSBD can smartly select more important features and is more suitable for challenging and complicated real-world heterogeneous environments.

5.2. Accuracy on different unseen path conditions

Our trained model's universal adaptability is subject to the size of the datasets used in the training phase. If the datasets could cover a huge number of path conditions, the trained model would be more general. However, we wonder whether our approach can train a high-accuracy classifier with size-limited datasets (*e.g.,* the datasets in our demonstration example) for common but different path conditions (different from those in our demonstration example). We generate 10 NSB/SB scenarios with Mininet [68] and ns-3 simulator [67]. Figs. 7(a) and 7(b) show the topologies. Two Mininet/ns-3 hosts act as the server and the client. We configure the bandwidths of Links 1–3 and the base RTTs (the minimum RTT or the RTT without queuing delay) of the two paths to create the path conditions we need. For the RTT configurations, only Links 2 and 3 are configured delay, while other links (between devices including hosts and switches) are not configured, *i.e.,* 0 s. The parameters are presented in Table 6. To emulate/simulate different path conditions brought by heterogeneous paths, we configure the parameters of the two paths with quite different values. Moreover, the path conditions are different from those created in Section 3.1. We install the linux implementation of SmartSBD on Mininet hosts for Mininet emulations and use the ns-3 implementation of SmartSBD on



(a) Mininet Topology (b) NS-3 Topology

(c) Mininet Accuracy (d) NS-3 Accuracy

Fig. 7. The topology used in Section 5.2 and accuracy results.

Table 6
Path parameters in Section 5.2.

Cases	Bandwidth (Mbps)			Base RTT (ms)	
	Link 1	Link 2	Link 3	Path 1	Path 2
NSB1	10	30	50	100	40
NSB2	10	30	50	100	20
NSB3	10	30	50	40	40
NSB4	10	30	50	40	20
NSB5	10	30	50	20	20
SB1	50	30	10	40	20
SB2	50	30	10	20	20
SB3	50	10	30	100	20
SB4	50	10	30	40	20
SB5	50	10	30	40	40

ns-3 hosts. Each experiment runs 60 s, and we repeat 10 times for each case. The results are shown in Fig. 7(c). Although the overall mean accuracy for all cases, 0.858, is less than those in Table 5, considering the different unseen path conditions, our learning-based data-driven approach is still promising for shared bottleneck detection.

5.3. Throughput maximization and bottleneck friendliness

We evaluate the performance and bottleneck friendliness of the implementations of SmartSBD (DT) and SmartSBD (RF) in the Linux kernel. Since the results are similar, we only show those of SmartSBD (DT). We employ the network setup shown in Fig. 2(b) to create heterogeneous networks.

5.3.1. NSB scenarios (throughput maximization)

We still use the setup shown in Fig. 2(b) and the method described in Section 3.1 to create the MPTCP flow and background flows. We still deploy the DC Servers and BG Servers in four locations following the demonstration example shown in Section 3.1. We configure the bandwidths of the two bottlenecks according to Table 2.

The throughput results are shown in Fig. 8. Note that the MP Reno also acts as the Uncoupled CC of SmartSBD, which is illustrated in Section 4. As we have stated in Section 1, Uncoupled CC cannot ensure bottleneck friendliness in the SB scenario, so it is not allowed to be used in practical deployments [8], where different subflows of one MPTCP connection may share the same bottleneck. In this subsection, we create the NSB scenario so that the MP Reno can be considered for the purpose of experiments instead of usage in real-world networks. Both SmartSBD and MP Reno outperform Coupled CC algorithms: LIA, OLIA, BALIA and MPCC. SmartSBD improves 15.3%–123.7% throughput compared with Coupled CC algorithms. SmartSBD achieves similar throughput as MP Reno thanks to its accurate detection. According to the results in Table 5 and Fig. 8, we can also find that the more accurate the detection is, the higher the throughput SmartSBD would achieve. As seen in Fig. 8, MP Reno has a slightly higher throughput than SmartSBD, but that does not mean MP Reno can be employed in real-world networks. As we have mentioned in Section 1, in real-world networks, the subflows of one MPTCP connection may have the same bottleneck (SB scenario), and Uncoupled CC including MP Reno will not keep the bottleneck friendliness in such scenario, thus harming other flows. Therefore, Uncoupled CC is not allowed to be used in practice [8]. Coupled CC algorithms always think that the subflows may run through the same bottleneck, so they reduce the increment of subflow congestion window for each ACK, then the whole MPTCP connection becomes friendly with other TCP connections in the SB scenario. However, in the NSB scenario, they would have much lower throughput.

5.3.2. SB scenarios (bottleneck friendliness)

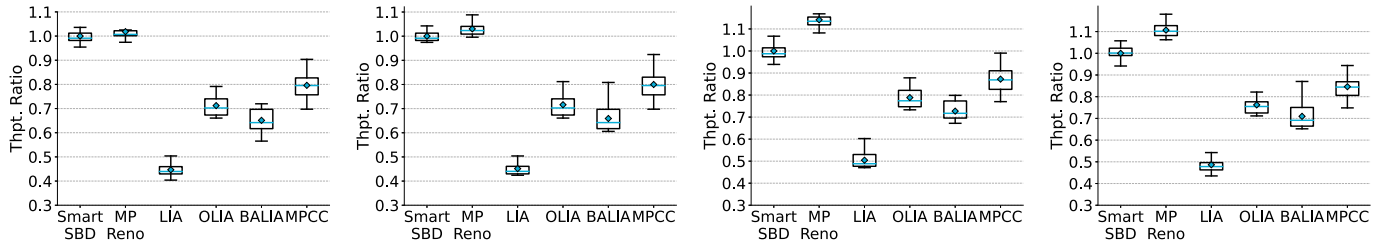
We use a slightly modified setup to introduce bandwidth competence at the SB bottleneck. The setup is based on what is shown in Fig. 2(b) but we introduce a new client (BG Client) into the setup. The BG Client is located at the same lab with the DC Client. When the DC Client establishes an MPTCP connection with the DC Server, the BG Client also generates a TCP connection with the DC Server. These two connections would compete for the bandwidth at the SB bottleneck, which is set to 16 Mbps.

The throughputs achieved by LIA, OLIA, and BALIA can be regarded as benchmarks to measure the ability of SmartSBD to hold bottleneck friendliness since they are tied to TCP standard AIMD mechanism and have already shown excellent bottleneck friendliness in a vast number of existing works [1,2,8–12]. Since using Uncoupled CC (MP Reno) in SB scenario is known to be bottleneck unfriendly, we only compare SmartSBD with Coupled CC algorithms. If SmartSBD is able to keep the friendliness, it would achieve a similar throughput. The goal of all CC algorithms in this subsection is not the highest throughput. Since the two MPTCP subflows share the same bottleneck (SB scenarios) and there is another TCP connection competing bandwidth at the bottleneck, the bottleneck friendliness of the MPTCP CC algorithms is embodied by not taking too much bandwidth at the bottleneck. Fig. 9 shows SmartSBD gains only slightly higher bandwidth than other benchmarks. Especially, SmartSBD only gets 8.1% more mean bandwidth than the default MPTCP CC, LIA. If SmartSBD made mistakes frequently, it would obtain much more bandwidth. As SmartSBD is accurate in most of time, it holds the friendliness well. It is worth noting that SmartSBD is more friendly than MPCC, an online-learning MPTCP CC, derived from the PCC framework [45,69]. MPCC gains 18.1%–20.2% than other Coupled CC algorithms, more aggressive than SmartSBD.

5.3.3. Shifting bottleneck scenarios

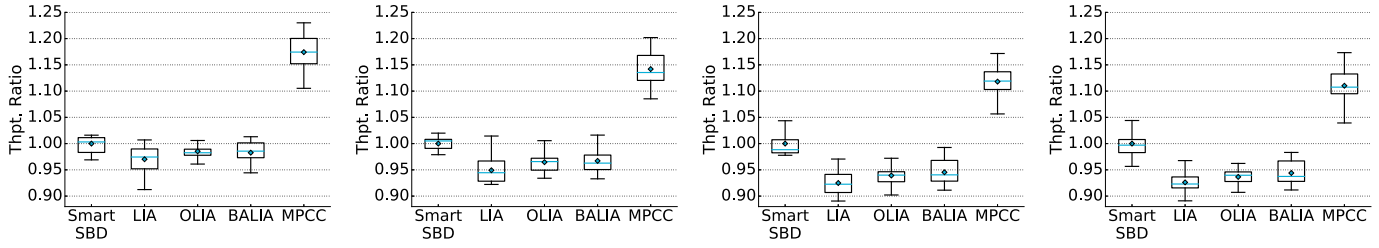
SmartSBD periodically makes decisions, which enables itself to adapt to bottleneck shifts. We construct shifting bottleneck scenarios, which continue 180 s. In the first 60 s and the last 60 s, we use the SB setup described in Section 5.3.2. In the middle 60 s, we construct a NSB scenario by limiting the bandwidths of the two bottlenecks following Table 2. Since in our shifting bottleneck scenarios, subflows may run through the same bottleneck, and using Uncoupled CC (MP Reno) in SB scenario is known to be bottleneck unfriendly, we only compare SmartSBD with Coupled CC algorithms in this subsection.

We use the mean throughput for each second as the results, depicted in Fig. 10. SmartSBD needs some time to make decisions (~3 s for SH, ~4 s for GZ, ~11 s for SV, ~16 s for FRA). The duration of decision making is determined by T, which is 50 times the larger value of the minimum RTTs of the two subflows, first introduced in Section 3.2.



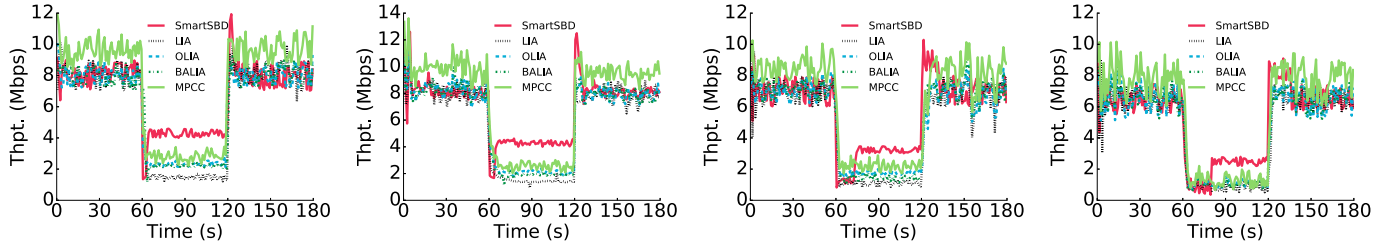
(a) SH (b) GZ (c) SV (d) FRA

Fig. 8. Throughput of MPTCP flows in NSB scenarios (We omit the outliers.). The results are normalized to the means of SmartSBD. Compared with Coupled CC algorithms, SmartSBD improves throughput in NSB scenarios. For all the boxplots in this paper, the ends of the whiskers are set according to the definition of Tukey boxplot [51], and the diamonds show mean values.



(a) SH (b) GZ (c) SV (d) FRA

Fig. 9. Throughput of MPTCP flows in SB scenarios (We omit the outliers.). The results are normalized to the means of SmartSBD. SmartSBD achieves only slightly higher bandwidth than other algorithms, which indicates it keeps bottleneck friendliness in SB scenarios.



(a) SH (b) GZ (c) SV (d) FRA

Fig. 10. Throughput of MPTCP flows in shifting bottleneck scenarios. SmartSBD achieves higher bandwidth in NSB phases and keeps bottleneck friendliness in SB phases. We use the mean throughput for each second as the results.

It is worth to note that such time period is much shorter than that is used in [1], which is about 175 times the larger value of the minimum RTTs of the two subflows (3.5 s for 20 ms RTT). According to recent measurement studies [70–72], more and more cloud datacenters are built, and the end-to-end latencies have decreased from 100 ms to 10–25 ms. Moreover, [71,72] show that 53 countries can access the cloud with RTTs less than 20 ms, including most areas of Europe and North America, and two countries with large populations, China and India. With the help of CDN techniques, the clients in these countries usually access their local datacenters resulting in the mentioned small RTTs, and SmartSBD can make detection less than 1 s in such cases (minimum RTT less than 20 ms). In the evaluation, we rent servers identical to those used in the data collection phase, and these servers are accidentally far from the client, leading to high RTTs, more than 100 ms on average. However, considering the help of widely deployed CDNs, most clients in the real world are much near the servers, resulting in much less detection delay of SmartSBD. In NSB phase, after the change of CC, SmartSBD achieves a notably higher throughput because it finds out the two subflows are running through distinct bottlenecks, and thus SmartSBD employs Uncoupled CC. In SB phase, after the change of CC, SmartSBD has a similar throughput as LIA, indicating it keeps bottleneck friendliness. Compared with SmartSBD, MPCC exhibits worse bottleneck friendliness in SB phases, while less throughput in NSB phase.

5.4. CPU overhead

The CPU overhead introduced by SmartSBD consists of three parts: recording of state variables, feature computing and classification. State variable recording is triggered when the sender receives an ACK packet or the congestion state of one subflow is changing. That overhead is neglectable, because the existing sender TCP would also do some state updates and records at the same time. Then we focus on the overhead of feature computing and classification. Our trained model makes classification every T. At the end of each T, SmartSBD first computes features and then makes classification based on these features.

We consider two methods to estimate CPU overhead: (1) Linux `perf`; (2) execution time printing. Linux `perf` is a sampling-based tool providing a per-function breakdown of CPU cycles [73]. With `perf`, we could obtain the percent of CPU used by feature computing and classification. With printed execution time, we could know their running time. We evaluate the CPU overhead for the experiments in Section 5.3.

Table 7
CPU overhead introduced by SmartSBD in each T.

CPU usage	Mean	Max.	Execution time	Mean	Max.
Feature computing	0.023%	0.03%	Feature computing	2.23 ms	4.41 ms
Classif. of DT	0.032%	0.04%	Classif. of DT	4.47 ms	6.26 ms
Classif. of RF	0.035%	0.05%	Classif. of RF	4.70 ms	7.36 ms

The CPU overhead is shown in Table 7. As we can see, both feature computing and classification take no more than 0.05% CPU overhead, which is minuscule and similar to the CPU overhead of other functions relevant to congestion control. The mean execution time of feature computing is 2.23 ms. The mean execution times of classification are 4.47 ms and 4.70 ms for SmartSBD (DT) and SmartSBD (RF) respectively. Since T is the 50 times the larger value of the minimum RTTs of the two subflows and the RTT between end hosts is usually about a few milliseconds to hundreds of milliseconds [14], the CPU overhead of SmartSBD (DT) and SmartSBD (RF) is minuscule and acceptable. It is worth to note that SmartSBD has less computation overhead than other RL-based MPTCP CC schemes, *e.g.,* DRL-CC, because SmartSBD classifier is only invoked once in T while DRL-CC queries the agent whenever the subflow congestion window changes, and the calculation of the SmartSBD classifier is less complex than the agent of DRL-CC.

6. Related work

We classify the related works based on their orientation into three categories, as shown in Sections 6.1 to 6.3. Then, we further divide the related works of one category into several groups. For each group, we first briefly introduce their core ideas, then we indicate their drawbacks. For all the schemes presented in this section, they either cannot ensure bottleneck friendliness and throughput maximization together for both SB scenario and NSB scenario, or have deployment issues.

6.1. Shared bottleneck detection only based on the information from end-hosts

For the schemes of shared bottleneck detection only based on the information from end-hosts, according to their design goals, we divide them into two groups: (1) For the MPTCP; (2) For multiple connections/flows.

For MPTCP: DWC proposed in [2] and SBD proposed in [1] detect shared bottlenecks only based on the end-host information, and they are designed for the MPTCP protocol. Hassayoun et al. [2] create a set of rules to identify SB/NSB based on round trip time (RTT) and packet loss, which are measured by the sender, *i.e.,* the server. Ferlin et al. [1] think that RTT is not accurate enough, and they create rules based on latency variations collected by cooperation between the sender and receiver. The detection of SBD is conducted by the receiver, which periodically makes decisions and feeds them back to the sender.

Although they can cooperate the MPTCP protocol, they still have some limitations. First, the accuracy of rules created by these approaches are highly dependent on particular network conditions. As shown in Section 5, the accuracy of the approaches drops sharply, from 75% to 56% (Hassayoun et al. [2]) and from 90% to 65% (Ferlin et al. [1]), respectively, when evaluated with real-world traces. These results imply that creating proper rules to identify SB/NSB with high accuracy is complicated even in one certain network environment, and transforming the rules to identify SB/NSB with high accuracy in a different network condition is hardly accomplishable. Second, which subflow properties (or state variables) have helpful information for the shared bottleneck detection task is still an open issue and depends on researchers' decisions. Third, they still suffer from deployment issues. The approach proposed in [2] was only implemented in a simulator, where a complicated state machine is introduced into the MPTCP subflow CC component. Meanwhile, the implementation of Linux MPTCP subflow also includes a CC state machine [35,47], and there is no identical state in the two state machines, so how to merge them is unsolved. As for SBD, coordination between the sender and receiver [1] makes the deployment difficult, and a dishonest receiver may incur unfriendliness. Considering the client is usually the receiver in many applications, allowing a selfish client to make decisions can result in unfriendliness. For example, the client can be customized to pursue higher throughput by always feeding back the information that all subflows are running on distinct bottlenecks to make the server employ the Uncoupled CC, even though the subflows are sharing the same bottleneck.

For multiple flows: Though the above schemes focus on the problem of shared bottleneck detection in one single connection of one single flow, some schemes detect shared bottlenecks across several connections or flows [17–21]. They also create rules to detect shared bottlenecks, and their rules are based on the information of delay or packet loss events that can be collected from end-hosts. The schemes in [17,19–21] inject probe packets following self-defined patterns, and the scheme in [18] requires defining new TCP option fields in ACK packets.

Introducing these schemes into MPTCP has several drawbacks. First, the subflow of standard MPTCP sends packets if it has been scheduled some packets and its congestion window does not limit. However, if we implement the schemes of [17,19–21] into MPTCP, the subflow either has to send empty probe packets when there is no packet to send, or delay the packets that should be sent in order to comply with the patterns of sending probe packets. Such modifications are huge because they may change the timing of sending packets or the content of sending packets. These proposals present integration overhead that could hinder their wide spread adoption. Second, the scheme proposed in [18] requires defining new TCP option fields in ACK packets. Such protocol modifications need a consensus of TCP/MPTCP community since the TCP option field is length limited and all options have to be understood by all TCP/MPTCP implementations. Third, all these proposals [17–21] only consider persistent shared/non-shared bottlenecks. However, bottleneck shifts should be considered for MPTCP CC. Finally, these proposals [17–21] assume the paths sharing the same bottleneck have a similar delay, which is unrealistic in real-world networks.

6.2. Shared bottleneck detection based on the information from networks

For the schemes of shared bottleneck detection based on the information from networks, according to their design goals, we divide them into two groups: (1) For the current Internet; (2) For the networks with centralized authorities.

For the current Internet: Since the measurements of RTT and packet loss events at end-hosts can be inaccurate, some researchers propose to explicitly share more information from the current Internet to help the end-hosts. Wei et al. [16] proposed a scheme including a ECN-based shared bottleneck detection method (TON20). TON20 utilizes ECN (Explicit Congestion Notification) [74] to detect shared bottlenecks. Their key idea is that the subflows that are ECN-marked at the same time have a higher chance of sharing the same bottleneck. TON20 requires all the devices, including endpoints along network paths, to enable the ECN mechanism, *i.e.,* all the forwarding devices and end hosts not only support the ECN mechanism but also activate it via reasonable configurations. Moreover, a new research group, Path Aware Networking RG (Panrg) [22], has been established by Internet

Research Task Force (IRTF) recently in order to share more network information with end-hosts.

The main drawback of these schemes [16,22] is that they cannot be assumed enabled on the current Internet. Although recent reports show that many end hosts enable the ECN mechanism [75] and some forwarding devices support it [76], to the best of our knowledge, on the Internet, how to detect ECN-activated forwarding devices are deployed is still an open issue, which makes people not know the exact number and proportion of deployed ECN-activated forwarding devices. Thanks to the huge benefits of the ECN mechanism, we believe that most Internet paths will enable it in the future, but it may not be enabled for the current Internet. As for the studies relevant to Panrg, they are still preliminary and it would take quite a long time to deploy these in-network schemes.

For the networks with centralized authorities: Since it is hard to detect shared bottlenecks only according to each end-host's information accurately, some SDN-based schemes are proposed [23–30]. Their ideas are pretty similar: with the assistance of centralized controllers, MPTCP subflow management can be improved. They make the SDN controller tell end-hosts how many subflows each MPTCP connection can create, how to generate the subflows to distribute them on disjoint paths, how to use the built subflows, how to dynamically add or delete subflows, *etc*. Moreover, they also propose new methods to support the communication between the SDN controller and each end-host, because existing SDN controllers do not provide any method to communicate with end-hosts directly. The main drawback of these schemes is that they can only be used on networks with centralized authorities, *e.g.,* SDN networks.

6.3. Shared bottleneck-unaware MPTCP CC

For the MPTCP CC algorithms that do not rely on the shared bottleneck detection, they can be described as two groups: Coupled CC and ML-based CC.

Coupled CC for MPTCP: Several Coupled CC algorithms has been proposed [8–12]. They are designed to conservatively adjust subflow CC windows to avoid possible bottleneck unfriendliness, even though there is no risk (*i.e.,* NSB). Though they share the design goal of bottleneck friendliness, their performance may differ in different scenarios, *e.g.,* MPCC is more unfriendly than other Coupled CC algorithms. As indicated in Section 1, the main drawback of the Coupled CC algorithms is that they fail to maximize throughput when there is no shared bottleneck between two subflows (*i.e.,* NSB).

Machine Learning based MPTCP CC: Recently, machine learning techniques have been applied to improve CC. Existing studies mainly focus on TCP [37,38,40,41,45,48,69,77]. MPTCP CC has also been revisited with machine learning techniques. MPCC is an online-learning MPTCP CC, derived from the PCC framework [45,69]. We evaluate it in Section 5.3. Reinforcement learning (RL) method has also been applied to enhance MPTCP CC. DRL-CC [31], SmartCC [32] and MPLibra [33] are based on RL and require at least one agent running on each host/server to provide RL-based MPTCP CC service.

For these schemes, they either cannot solve the problem faced by MPTCP CC, or have deployment issues. As shown in Section 5, MPCC cannot keep bottleneck friendliness well in SB scenarios. As for other RL-based schemes, commodity Linux-based devices cannot support complex computations in the Linux kernel introduced by RL. The agent of DRL-CC is implemented as a user-space process running Tensorflow [78]. Every time a subflow congestion window changes, the CC part in the kernel needs to query the agent, which may introduce high overhead. The overhead of DRL-CC is not evaluated in [31]. DRL-CC is very complicated, integrating several machine learning techniques, such as Deep Neural Network (DNN) and Long Short-Term Memory (LSTM [79]), *etc*, and its implementation is not available online. Some parts of DRL-CC (*e.g.,* the state collection module) are not introduced in detail in [31], making it hard to realize it correctly. SmartCC and MPLibra have only been implemented in ns-3 [67], and how to introduce their RL part into the Linux kernel MPTCP stack is still an open issue. Moreover, SmartCC does not consider the bottleneck friendliness issue (the TCP-friendliness issue in [32]).

7. Discussion

Extension to 3 + Subflows: We focus on the shared bottleneck detection problem of two-subflow MPTCP in previous sections, however, in this section, we introduce a solution to solve that problem of MPTCP with 3+ subflows based on SmartSBD. Let S be the set of all subflows of one MPTCP connection. At the end of each T, find a partition (P) of S that meets two requirements: (1) any two subflows selected from two different subsets do not share the same bottleneck; (2) for all partitions satisfied (1), P has the maximum number of subsets. Then, for each $s \in P$, couple all the subflows in s to use a Coupled CC algorithm, if $|s| > 1$; let the subflow in s use a Uncoupled CC algorithm, if $|s| = 1$. In this solution, SmartSBD can be used to test whether two subflows share the same bottleneck.

Different MPTCP schedulers: We use the default MPTCP packet scheduler, LowRTT [80], in this paper. Though a prior study [81] has shown that LowRTT can interact adversely with the MPTCP CC, we have not found LowRTT has any notable influence on SmartSBD.

Application of SmartSBD: Although some existing works focus on a more general problem of detecting the shared bottleneck link [17–21], which means their solution may be used in different use cases, SmartSBD is specifically designed for MPTCP, like [1,2]. According to Section 3.3, SmartSBD requires two state variables (RTT samples and congestion state) of each subflow to conduct detection. The state variables are strongly associated with TCP flows or MPTCP subflows. The RTT samples are obtained from new ACK packets, and the subflow congestion state is defined by the congestion state machine of Linux TCP/MPTCP implementation [35,47]. Moreover, the shared bottleneck detection problem of MPTCP CC requires the solution to detect bottleneck shifts instead of persistent shared/non-shared bottleneck scenarios. As shown in Section 5.3.3, SmartSBD supports shifting bottleneck detection. Therefore, SmartSBD is tightly coupled with the need of MPTCP CC, passively collects state variables, and does not require other behavior modification. Besides MPTCP, SmartSBD can also be introduced into a multipath extension of the emerging transport protocol, QUIC [82]. Multipath QUIC (MPQUIC) inherits CC from MPTCP CC [13,83], and each QUIC subflow also has the mentioned state variables. Maybe the detection method of SmartSBD can be applied to other shared bottleneck detection tasks unrelated to multipath transport. For example, the operators of CDNs or cloud platforms may adopt SmartSBD so as to enhance the quality of their services transparently from users. However, this paper focuses on the shared bottleneck detection problem in multipath transport like SBD or DWC.

Though the usage of SmartSBD needs cooperation with content providers, the performance gain is convincingly large. As indicated in Section 5.3.1, SmartSBD can lead to up to 123.7% throughput improvement in NSB scenario, compared with existing MPTCP CC. We think the NSB scenario is more common than the SB scenario in practice. In recent years, many researchers have studied how to improve the performance of transport protocols on wireless networks [84–88], because wireless links are prone to be the transport bottleneck. For the multihomed devices, if one wireless access link becomes a bottleneck, the two paths do not share the same bottleneck (*i.e.,* NSB scenario). Thanks to SmartSBD, the aggregated throughput of two paths is significantly improved in the NSB scenario, which could motivate content providers to use SmartSBD. Additionally, though SmartSBD is designed for heterogeneous networks, and the demonstration is also built for heterogeneous networks, it could be used in traditional wired

networks. Actually, the two baseline schemes, SBD and DWC, were only evaluated over wired networks [1,2], and Section 5 shows that SmartSBD outperforms them. We believe SmartSBD can be applied over wired networks.

Network Setups in Real-world Heterogeneous Networks: The network setups shown in Fig. 2 are used in the training phase of SmartSBD and in Section 5. Compared with the network setups employed in other papers [1,2,16], the network setups in Fig. 2 are different. In this paper, we consider the prominent use case of MPTCP [3], which includes different wireless links on different Internet paths, so both topologies Figs. 2(a) and 2(b) have 4G and WiFi links. The network setups in other papers do not include wireless links [1,2,16]. Specifically, (1) the network setup used in [1] is constructed over NorNet [89], a real-world, large-scale multi-homing testbed. The method Ferlin et al. used to create bottlenecks is similar with ours: The links close to endpoints are leveraged to create bottlenecks because they can be configured more easily by the endpoints or researchers than other links. However, as we have mentioned, their topology does not involve any wireless links. (2) The network setups used in [2] are based on the ns-2 simulator. The simulated topology does not have wireless links either. (3) The network setups used in [16] are based on a self-built testbed and ns-3 simulator. All the links are wired, and all the background traffic is generated, while part background traffic in our paper and [1] is real Internet traffic.

Besides collecting system logs from real-world heterogeneous networks, like what we have done in our demonstration example, we also considered building simulation or emulation networks with manually generated traffic to collect the data needed for training. However, all the existing simulators or emulators may lack fidelity, so we choose to build network setups in real-world heterogeneous networks to collect the data. We believe such a requirement is worthwhile. If a high-fidelity simulator/emulator is developed and widely used in the future (Note that the simulator/emulator has to be able to provide almost real wireless links and almost real background traffic.), we think the data collection can be done with it instead of constructing network setups in real-world networks.

8. Conclusion

In this paper, we propose SmartSBD, a first-of-its-kind solution that enables efficient MPTCP CC achieving both bottleneck friendliness and throughput maximization via smart shared bottleneck detection. Different from the customized rule-based approach employed in existing works [1,2,16], SmartSBD is a data-driven approach to detect shared bottlenecks accurately. Moreover, to make SmartSBD easy to deploy, we propose SmartSBD as a pure end-host solution that only requires the standard MPTCP to modify the sender CC and does not need any modifications in networks. We evaluate SmartSBD through comprehensive experiments. Compared with existing approaches [1,2,16], SmartSBD achieves at least 47.5% improvement in average accuracy. For unseen path conditions, SmartSBD still obtains overall mean accuracy of 0.861. Compared with existing MPTCP CC algorithms, SmartSBD achieves 15.3%–123.7% throughput improvement in NSB scenario and keeps bottleneck friendliness in SB scenario. Moreover, SmartSBD outperforms SBD [1], the existing scheme with accuracy closest to SmartSBD, in terms of adaptability to bottleneck shift. Last but not least, the CPU overhead of SmartSBD is minuscule.

CRediT authorship contribution statement

Enhuan Dong: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft. **Peng Gao:** Methodology, Writing – review & editing. **Yuan Yang:** Writing – review & editing. **Mingwei Xu:** Supervision, Writing – review & editing, Funding acquisition. **Xiaoming Fu:** Writing – review & editing. **Jiahai Yang:** Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The research was supported by the National Natural Science Foundation of China under Grant 62002192, Grant 62221003, and Grant 62172251.

References

[1] S. Ferlin, Ö. Alay, T. Dreibholz, D. Hayes, M. Welzl, Revisiting congestion control for multipath TCP with shared bottleneck detection, in: IEEE INFOCOM, 2016.
[2] S. Hassayoun, J. Iyengar, D. Ros, Dynamic window coupling for multipath congestion control, in: IEEE ICNP, 2011.
[3] O. Bonaventure, C. Paasch, G. Detal, RFC 8041: Use cases and operational experience with multipath TCP, 2017.
[4] IETF MPTCP working group, http://datatracker.ietf.org/wg/mptcp/documents/.
[5] Apple opens multipath TCP in iOS11, https://support.apple.com/en-us/HT201373.
[6] Link turbo aggregation technology on Huawei honor V20, https://www.gizmochina.com/2018/12/26/exclusive-how-link-turbo-works-on-the-honor-v20/.
[7] Linux MPTCP upstream project, URL https://github.com/multipath-tcp/mptcp_net-next/wiki/.
[8] C. Raiciu, M. Handly, D. Wischik, Coupled congestion control for multipath transport protocols, 2011, RFC 6356.
[9] D. Wischik, C. Raiciu, A. Greenhalgh, M. Handley, Design, implementation and evaluation of congestion control for multipath TCP, in: USENIX NSDI, 2011.
[10] R. Khalili, N. Gast, M. Popovic, J.-Y.L. Boudec, MPTCP is not Pareto-optimal: Performance issues and a possible solution, IEEE/ACM Trans. Netw. (2013).
[11] Q. Peng, A. Walid, S.H. Low, Multipath TCP algorithms: Theory and design, in: ACM SIGMETRICS, 2013.
[12] T. Gilad, N. Rozen-Schiff, P. Godfrey, C. Raiciu, M. Schapira, MPCC: Online learning multipath transport, in: ACM CoNEXT, 2020.
[13] Q.D. Coninck, O. Bonaventure, Multipath QUIC: Design and evaluation, in: ACM CoNEXT, 2017.
[14] Y.-C. Chen, Y. Lim, R.J. Gibbens, E.M. Nahum, R. Khalili, D. Towsley, A measurement-based study of multipath TCP performance over wireless networks, in: ACM IMC, 2013.
[15] M. Allman, V. Paxson, E. Blanton, RFC 5681: TCP congestion control, 2009.
[16] W. Wei, K. Xue, J. Han, D. Wei, P. Hong, Shared bottleneck-based congestion control and packet scheduling for multipath TCP, IEEE/ACM Trans. Netw. (2020).
[17] M.Y. Murtaza, M. Welzl, On the accurate identification of network paths having a common bottleneck, Sci. World J. (2013).
[18] O. Younis, S. Fahmy, Flowmate: Scalable on-line flow clustering, IEEE/ACM Trans. Netw. (2005).
[19] M. Kim, T. Kim, Y. Shin, S. Lam, E. Powers, A wavelet-based approach to detect shared congestion, IEEE/ACM Trans. Netw. (2008).
[20] D. Rubenstein, J. Kurose, D. Towsley, Detecting shared congestion of flows via end-to-end measurement, IEEE/ACM Trans. Netw. (2002).
[21] H. K., A. Bestavros, J. Byers, Robust identification of shared losses using end-to-end unicast probes, in: IEEE ICNP, 2000.
[22] Path aware networking RG (panrg), https://datatracker.ietf.org/rg/panrg.
[23] Z. Jiang, Q. Wu, H. Li, J. Wu, Scmptcp: SDN cooperated multipath transfer for satellite network with load awareness, IEEE Access (2018).
[24] H. Nam, D. Calin, H. Schulzrinne, Towards dynamic MPTCP path control using SDN, in: IEEE NetSoft, 2016.
[25] M. Sandri, A. Silva, L. Rocha, F. Verdi, On the benefits of using multipath TCP and openflow in shared bottlenecks, in: IEEE 29th International Conference on Advanced Information Networking and Applications, 2015.
[26] S. Zannettou, M. Sirivianos, F. Papadopoulos, Exploiting path diversity in datacenters using MPTCP-aware SDN, in: IEEE ISCC, 2016.
[27] K. Joshi, K. Kataoka, Sfo: Subflow optimizer for MPTCP in SDN, in: ITNAC, 2016.
[28] T. Wang, M. Hamdi, eMPTCP: Towards high performance multipath data transmission by leveraging SDN, in: IEEE GLOBECOM, 2018.
[29] F. Alharbi, Z. Fei, An SDN architecture for improving throughput of large flows using multipath TCP, in: IEEE CSCloud / IEEE EdgeCom, 2018.

[30] F. Becerra, D. Mejía, I. Bernal, Solving MP-TCPś shared bottlenecks using a SDN with OpenDayLight as the controller, in: IEEE ANDESCON, 2018.

[31] Z. Xu, J. Tang, C. Yin, Y. Wang, G. Xue, Experience-driven congestion control: When multi-path TCP meets deep reinforcement learning, IEEE J. Sel. Areas Commun. (2019).

[32] W. Li, H. Zhang, S. Gao, C. Xue, X. Wang, S. Lu, Smartcc: A reinforcement learning approach for multipath TCP congestion control in heterogeneous networks, IEEE J. Sel. Areas Commun. (2019).

[33] H. Yu, J. Zheng, Z. Du, G. Chen, Mplibra: Complementing the benefits of classic and learning-based multipath congestion control, in: IEEE ICNP, 2021.

[34] B.C. Ross, Mutual information between discrete and continuous data sets, PLoS ONE 9 (2) (2014).

[35] C. Paasch, S. Barre, et al., Multipath TCP in the linux kernel, 2020, http://www.multipath-tcp.org.

[36] Z.-H. Zhou, Open-environment machine learning, Natl. Sci. Rev. 9 (8) (2022).

[37] S. Abbasloo, C. Yen, H. Chao, Classic meets modern: a pragmatic learning-based congestion control for the internet, in: ACM SIGCOMM, 2020.

[38] F. Yan, J. Ma, G. Hill, D. Raghavan, R. Wahby, P. Levis, K. Winstein, Pantheon: the training ground for internet congestion-control research, in: USENIX ATC, 2018.

[39] H. Mao, R. Netravali, M. Alizadeh, Neural adaptive video streaming with pensieve, in: ACM SIGCOMM, 2017.

[40] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, J. Zhang, Reducing web latency through dynamically setting TCP initial window with reinforcement learning, in: IEEE/ACM IWQoS, 2018.

[41] X. Nie, Y. Zhao, Z. Li, G. Chen, K. Sui, J. Zhang, Z. Ye, D. Pei, Dynamic TCP initial windows and congestion control schemes through reinforcement learning, IEEE J. Sel. Areas Commun. (2019).

[42] J. Jiang, S. Sun, V. Sekar, H. Zhang, Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation, in: USENIX NSDI, 2017.

[43] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, B. Sinopoli, CS2p: Improving video bitrate selection and adaptation with data-driven throughput prediction, in: ACM SIGCOMM, 2016.

[44] N. Cardwell, Y. Cheng, C.S. Gunn, S. Yeganeh, V. Jacobson, BBR: Congestion-based congestion control, ACM Queue (2016).

[45] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, M. Schapira, PCC vivace: Online-learning congestion control, in: USENIX NSDI, 2018.

[46] Wonder Shaper: Command-line utility for limiting an adapter's bandwidth, https://github.com/magnific0/wondershaper.

[47] S. Arianfar, TCP's congestion control implementation in linux kernel, in: Proceedings of Seminar on Network Protocols in Operating Systems, 2012.

[48] K. Winstein, H. Balakrishnan, TCP ex machina: Computer-generated congestion control, in: ACM SIGCOMM, 2013.

[49] Y. Kong, H. Zang, X. Ma, Improving TCP congestion control with machine intelligence, in: Proceedings of the Workshop on Network Meets AI & ML, ACM, 2018.

[50] The iPerf3, a speed test tool, http://iperf.fr.

[51] M. Frigge, D. Hoaglin, B. Iglewicz, Some implementations of the boxplot, Amer. Statist. 43 (1) (1989) 50–54.

[52] Huawei cloud, https://www.huaweicloud.com.

[53] Alibaba cloud, https://www.alibabacloud.com.

[54] H. Sakoe, S. Chiba, Dynamic programming algorithm optimization for spoken word recognition, IEEE Trans. Acoust. Speech Signal Process. (1978).

[55] T. Rath, R. Manmatha, Word image matching using dynamic time warping, in: IEEE CVPRW, 2003.

[56] C. Aleman, N. Pissinou, S. Alemany, G. Kamhoua, Using candlestick charting and dynamic time warping for data behavior modeling and trend prediction for MWSN in IoT, in: IEEE Big Data, 2018.

[57] T. Thongmee, H. Suzuki, T. Ohno, U. Silparcha, Finding strong relationships of stock prices using blockwise symbolic representation with dynamic time warping, in: IEEE INISTA, 2014.

[58] M. Cuturi, M. Blondel, Soft-DTW: a differentiable loss function for time-series, in: International Conference on Machine Learning, 2017, pp. 894–903.

[59] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[60] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, in: Advances in Neural Information Processing Systems, 2015, pp. 2962–2970.

[61] L. Breiman, Classification and Regression Trees, Routledge, 2017.

[62] L. Breiman, Random forests, Mach. Learn. (2001).

[63] C. Chang, C. Lin, LIBSVM: A library for support vector machines, ACM Trans. Intell. Syst. Technol. (2011).

[64] R. Fan, K. Chang, C. Hsieh, X. Wang, C. Lin, LIBLINEAR: A library for large linear classification, J. Mach. Learn. Res. (2008).

[65] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education Limited, Malaysia, 2016.

[66] R. Malouf, A comparison of algorithms for maximum entropy parameter estimation, in: Proceedings of the 6th Conference on Natural Language Learning-Volume 20, Association for Computational Linguistics, 2002, pp. 1–7.

[67] The NS-3 network simulator, URL http://www.nsnam.org/.

[68] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, N. McKeown, Reproducible network experiments using container-based emulation, in: ACM CoNEXT, 2012.

[69] M. Dong, Q. Li, D. Zarchy, P. Godfrey, M. Schapira, PCC: Re-architecting congestion control for consistent high performance, in: USENIX NSDI, 2015.

[70] R. Singh, A. Dunna, P. Gill, Characterizing the deployment and performance of multi-CDNs, in: ACM IMC, 2018.

[71] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, J. Kangasharju, Pruning edge research with latency shears, in: ACM HotNets, 2020.

[72] L. Corneo, M. Eder, N. Mohan, A. Zavodovski, S. Bayhan, W. Wong, P. Gunningberg, J. Kangasharju, J. Ott, Surrounded by the clouds: A comprehensive cloud reachability study, in: WWW, 2021.

[73] perf: Linux profiling with performance counters, https://perf.wiki.kernel.org/index.php/Main_Page.

[74] K. Ramakrishnan, S. Floyd, D. Black, The addition of explicit congestion notification (ECN) to IP, 2001.

[75] D. Murray, T. Koziniec, S. Zander, M. Dixon, P. Koutsakis, An analysis of changing enterprise network traffic characteristics, in: APCC, 2017.

[76] QoS: Congestion avoidance configuration guide, URL https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_conavd/configuration/xe-16/qos-conavd-xe-16-book.pdf.

[77] T. Meng, N. Schiff, P. Godfrey, M. Schapira, PCC proteus: Scavenger transport and beyond, in: ACM SIGCOMM, 2020.

[78] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016, arXiv preprint arXiv:1603.04467.

[79] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.

[80] C. Paasch, S. Ferlin, O. Alay, O. Bonaventure, Experimental evaluation of multipath TCP schedulers, in: ACM SIGCOMM Workshop on CSWS, 2014.

[81] B. Arzani, A. Gurney, S. Cheng, R. Guerin, B. Loo, Deconstructing MPTCP performance, in: IEEE ICNP, 2014.

[82] IETF QUIC working group, https://datatracker.ietf.org/wg/quic/documents/.

[83] Q. Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, O. Bonaventure, Pluginizing QUIC, in: ACM SIGCOMM, 2019.

[84] Z. Meng, Y. Guo, C. Sun, B. Wang, J. Sherry, H. Liu, M. Xu, Achieving consistent low latency for wireless real-time communications with the shortest control loop, in: ACM SIGCOMM, 2022.

[85] T. Li, K. Zheng, K. Xu, R. Jadhav, T. Xiong, K. Winstein, K. Tan, TACK: Improving wireless transport performance by taming acknowledgments, in: ACM SIGCOMM, 2020.

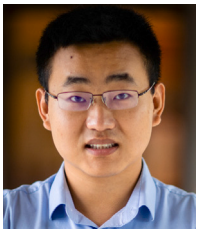
[86] S. Park, J. Lee, J. Kim, J. Lee, S. Ha, K. Lee, Exll: An extremely low-latency congestion control for mobile cellular networks, in: ACM CoNEXT, 2018.

[87] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, C. Görg, Adaptive congestion control for unpredictable cellular networks, in: ACM SIGCOMM, 2015.

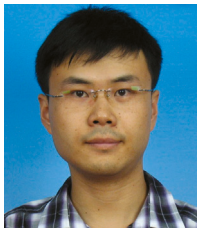
[88] K. Winstein, A. Sivaraman, H. Balakrishnan, Stochastic forecasts achieve high throughput and low delay over cellular networks, in: USENIX NSDI, 2013.

[89] NorNet: A real-world, large-scale multi-homing testbed, URL https://www.nntb.no.

Enhuan Dong received the B.E. (2013) degree from Harbin Institute of Technology, Harbin, China, and the Ph.D. (2019) degree from Tsinghua University, Beijing, China. He was a visiting Ph.D. student at the University of Goettingen in 2016–2017. He is an Assistant Research Professor in the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network security, network operations and network transport.



Peng Gao received the B.E. in Electrical and Computer Engineering at Shanghai Jiao Tong University in 2013, and the Ph.D. in Electrical Engineering at Princeton University in 2019. He is an Assistant Professor in the Department of Computer Science at Virginia Tech. His research interests include the domains of threat protection, threat intelligence, blockchain, trustworthy AI, and privacy-preserving systems.



Yuan Yang received the B.Sc., M.Sc., and Ph.D. degrees from Tsinghua University. He was a visiting Ph.D. Student with The Hong Kong Polytechnic University from 2012 to 2013. His major research interests include computer network architecture, routing protocol, and green networking. He holds an assistant research professor position with the Department of Computer Science and Technology, Tsinghua University.

Mingwei Xu received the B.Sc. degree and the Ph.D. degrees from Tsinghua University. He is a full professor in the Institute for Network Sciences and Cyberspace and the Department of Computer Science at Tsinghua University. His research interest includes Internet architecture, high-speed router architecture and network security.

Xiaoming Fu received the Ph.D. degree in computer science from Tsinghua University, China, in 2000. He is a full professor in the University of Goettingen. His research interests are architectures, protocols, and applications for networked systems, including information dissemination, mobile networking, cloud computing, and social networks.

Jiahai Yang received the B.Sc. degree in computer science from Beijing Technology and Business University, and the M.Sc. and Ph.D. degrees in computer science from Tsinghua University, Beijing, China. He is currently a Professor with Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network management, network measurement, network security, and cloud computing.