FlowBot: A Learning-Based Co-bottleneck Flow Detector for Video Servers

Jinhui Song*, Bo-Rong Chen*, Bowen Song†, Anyu Ying‡, Yih-Chun Hu*
*University of Illinois Urbana-Champaign, USA †University of California, Davis, USA ‡Carnegie Mellon University, USA

***Abstract*—Recent research has proposed that Content Delivery Networks (CDNs) can use better bandwidth allocation to improve video streaming services through congested links. Because CDNs are usually not located at the bottleneck link, shared bottleneck (co-bottleneck) detection on the video servers is necessary for joint flow shaping and the Quality of Experience (QoE) improvements. However, co-bottleneck detection is challenging in such environments due to the large number of flows, possible network topologies, and traffic patterns. Current detectors fail to balance detection accuracy, speed and overhead, and suffer performance degradation in the scale of thousands of flows on each video server. We propose FlowBot, a novel model-based passive co-bottleneck detector designed for deployment on a video server. FlowBot uses Siamese model to learn flow representations, and combines the training procedure with its clustering algorithm to continue to provide strong performance with up to thousands of flows. Our evaluations show that FlowBot can achieve consistently high accuracy (over 70% F1 with around 90% precision) in most tested scenarios, while maintaining a short detection delay of 3 s and overhead similar to the fastest benchmark algorithms.**

***Index Terms*—Shared bottleneck detection, Siamese model, one-way delay**

I. Introduction

Video streaming continues to grow as a dominant usage of the Internet in recent years [1], and optimization of Content Delivery Networks (CDNs) and video traffic is a trending research topic. Researchers have improved several aspects of CDN-based video streaming, with the goal of improving user experience: intra-CDN traffic [2], application-level optimizations, *e.g.,* Dynamic Adaptive Streaming over HTTP (DASH) [3], [4], and the routing of user requests to the CDN edge servers, *e.g.,* anycast [5]. Despite these advancements, on-path network bottlenecks still persist. End-to-end control [6]–[8] can improve video streaming by letting the server control the bandwidth usage of each co-bottlenecked flow, *e.g.,* stabilizing bursty video streams to improve Quality of Experience (QoE), or enforcing a sender-defined policy on the streams. Co-bottleneck detection forms a critical part of such systems.

There are two types of co-bottlenecks between video servers and clients. A *server-side co-bottleneck* can arise due to imbalances in edge-server load (which can follow a heavy-tailed distribution [9]), or because the CDN's anycast algorithm may be unaware of server load, resulting in co-bottleneck at the edge servers [5]. In contrast, *downstream co-bottlenecks* occur within an access Internet Service Provider (ISP) either due to the ISP's link capacity or some bandwidth constraint. For instance, when a large number of users at the same ISP watch the same video during some event, the ISP's link might be the bottleneck. Thus, a good co-bottleneck detector must handle both scenarios for end-to-end flow control in video servers.

An important feature of a CDN edge server is that it handles many video streams simultaneously. For instance, each Open Connect Appliance (OCA) [10] that Netflix provides to ISPs has 18–190 Gbps throughput. Sending 1080p or 2k video at 10–20 Mbps [11] reflects thousands to tens of thousands of flows per OCA. This scale is challenging for co-bottleneck detection.

Co-bottleneck detection is a long-studied problem with two distinct categories: active detection and passive detection. Active detection approaches [12]–[14] send packets to probe the links along the flow path. Pathneck [15] sends packets to measure the delay of some partial paths, loading the links with a packet train, and finding the link most affected by that load. A major disadvantage of these approaches is high levels of additional traffic, which is not suitable for large-scale scenarios.

Passive detection approaches [16]–[23] do not send any extra traffic on the forward path, but instead passively analyzes signals such as One Way Delay (OWD), Round Trip Time (RTT), Inter Packet Arrival Time (IPAT), loss, etc. These schemes exploit the correlation between the flows' performance across the co-bottleneck link; for example, two flows sharing a co-bottleneck link experience similar queueing delays on that link, resulting in similar OWD variation patterns, since queueing delays are the largest source of variance in OWD. Co-bottleneck detection is most commonly used in multi-path applications such as MPTCP [24] and RMCAT [23], which uses a detector to detect multiple subflows that share a co-bottleneck link. In these use cases, the detector is designed for a small number of subflows, as the number of subconnections is small (around 2–10). To our knowledge, previous work neither has satisfactory accuracy, nor is it well-evaluated, in scenarios with thousands of flows.

To address these challenges, we propose FlowBot, a novel passive co-bottleneck detector for video servers, based on a Siamese network [25]. In FlowBot, each flow first goes through a classifier to determine if it is bottlenecked, and then passes one Convolutional Neural Network (CNN) model to extract representations, which are then clustered to detect the co-bottleneck groups. We use a simulated dataset with sufficient variety to train the model, and use simulated and real test data to evaluate it. The advantages of FlowBot include: (i) **Accuracy at scale:** Unlike the most recently published algorithm Shared Bottleneck Detection (SBD) [17], we jointly choose the threshold in training loss and the distance threshold in the clustering, and can therefore maintain high accuracy for up to 1200 flows and 30 bottlenecks in our tests. (ii) **Computational**

979-8-3503-0322-3/23/$31.00 ©2023 IEEE

feasibility: The representation size is designed to be small to allow for efficient detection, allowing FlowBot to scale to thousands of flows (iii) **Generalizability:** By using a simulated dataset, FlowBot generalizes well to various real scenarios without seeing any real data during the training.

Our contributions include the following:

- We generate a novel simulated dataset for co-bottleneck detection study including 600 runs (15M triplet samples) with various network topologies, traffic patterns, and ground truth labels;
- We propose the first framework based on a Siamese model for co-bottleneck detection problem, yielding a detector with better generalizability and flexibility;
- We implement FlowBot, which achieves consistently high accuracy with over 75% F1 and over 90% precision in most tested scenarios using an improved clustering technique with the smallest overhead;
- We evaluate FlowBot comprehensively on both simulation and real experiments with built topologies using Google Cloud Platform (GCP), GENI [26], and CloudLab [27].

This paper focuses on wired networks, and we do not evaluate our approaches on wireless links which impose unique challenges including random changes to OWD, the possibility of non-congestive loss, and bitrate variability.

II. Motivation

This section motivates FlowBot by discussing potential applications, previous approaches, and how they cannot satisfy the requirements of these scenarios.

A. Application Scenarios

Co-bottleneck detection is useful for applications such as *end-to-end flow shaping* and *network diagnosis*. In *end-to-end flow shaping*, a content provider aims to improve some metric (*e.g.,* QoE) by reallocating traffic among video streams sharing a co-bottleneck [6], [8]. Because such systems decrease one flow's traffic to increase another flow's traffic, their action is friendly to competing flows only if both flows share a co-bottleneck link; thus, flow shaping systems need a co-bottleneck detector to precisely determine which flows can be shaped together. In *network diagnosis*, the output of co-bottleneck detection can serve as a hint for load balancing to improve CDN performance [28]. By detecting a server-side co-bottleneck, the CDN can adjust its load balancing strategy to avoid congestion at the edge servers. The detector can also provide co-bottleneck information to cloud diagnosis systems such as BlameIt [29], suggesting the location of the bottleneck link.

B. Previous Work

Co-bottleneck detection approaches can be classified into two categories: active or passive approaches. Active approaches send packets to measure the delay of some partial paths [12], [14], [15], loading the links with a packet train, and finding the link most affected by that load. Passive approaches use signal processing techniques to process end host measurements such as OWD (or RTT), loss, and IPAT to infer the co-bottleneck among flows. There are two types of passive approaches. (i) Correlation-based approaches [18], [20], [21] exploit the correlation of flow signals, such as OWD or loss rate, when the flows share co-bottleneck link. An example is Delay Correlation with Wavelet denoising (DCW) [18], which uses wavelet denoising to remove noise before calculating the correlation. Such approaches usually have difficulty handling path lag diversity, because diverse lags result in asynchronous signals at the end hosts, making correlation more difficult. They also require $\Theta(n^2)$ work to compute pairwise correlations between flows. (ii) Clustering-based approaches [16], [17], [19], [30] cluster flows into co-bottleneck groups based on some representations of the signals. An example is SBD [16], [17], which uses summary statistics like skewness, variability, OWD, and loss rate to represent each flow, then clusters these representations to obtain co-bottleneck groups. The accuracy of such approaches is highly dependent on the representation quality and parameters such as the clustering threshold, which are often scenario-dependent and hard to determine.

TABLE I: Co-bottleneck detection approaches.

Literature	Active/ Passive	Signal	Overhead Type
IPAT-based [19], [30]	P	IPAT	Deployment
Correlation-based [20]	P	OWD, loss	Computation
DCW [18]	P	OWD	Computation
Pathneck [8], [15]	A	RTT	Network
TSLP [12], [14]	A	latency, throughput	Deployment, network
QProbe [13]	A	IPAT	Network
Last mile detector [31]	P	RTT, IPAT	Deployment
SBD [16], [17]	P	OWD, SLR	-
SBDV [21]	P	Congestion interval	Computation
SB-FPS [22]	P	ECN	Deployment
FlowBot (proposed)	P	OWD, SLR	-

C. Requirements

To support large-scale applications, the detector must provide two properties.

Accuracy at scale. Based on the problem scope, the detector should scale to thousands of flows. As opposed to traditional detection, where false positives and false negatives are equally bad, we favor *precision* (fewer false positives) to reduce the number of wrongly clustered flows, because especially in the flow shaping scenario, incorrectly added flows will induce unfriendly congestion at the bottleneck link. Most existing passive co-bottleneck detection methods are designed to support smaller-scale multipath applications such as MPTCP, and do not consider large-scale use cases. For example, SBD is designed for up to 20 flows, and its performance is known to degrade severely when the number of parallel bottlenecks increases. Maintaining good accuracy at scale remains a challenge for co-bottleneck detectors.

Low overhead. In these applications, the detector should not introduce too much extra overhead, especially since the per-flow network overhead is multiplied by a large number of flows. We consider three types of overhead: (i) Network overhead. Active approaches introduce extra overhead by sending many probing packets, which does not scale well to a large number

of flows. (ii) Computational overhead. Some correlation-based passive methods such as DCW require the server to pairwise-correlate the raw signals such as OWD and loss rate, resulting in high computational requirements. (iii) Deployment overhead. Some passive approaches deploy extra observers to collect signals like IPAT, but the deployment of such observers may be difficult in real networks.

In summary, the detector should be able to detect the co-bottleneck with high accuracy at large scale with low overhead. However, to the best of our knowledge, existing methods cannot simultaneously satisfy both requirements. Thus, we propose FlowBot to address these challenges for co-bottleneck detection for large-scale applications, and compare it with prior work in Table I.

III. System Design

This section introduces the general design of our co-bottleneck detection system, then describes each component.

A. Overview

Our passive co-bottleneck detector (illustrated in Fig. 1) first extracts some time-series data, then removes the non-bottleneck flows, and finally clusters the data to determine co-bottlenecked groups. To remove non-bottleneck flows, FlowBot feeds the statistics of the time series into a binary congestion classifier to determine whether or not the flow is bottlenecked. Then, FlowBot uses a representation for clustering instead of the pairwise correlation to avoid the $\Theta(n^2)$ cost of pairwise correlation. An accurate representation must capture flow features of potentially heterogeneous bottleneck links. Reducing data to such a representation is intrinsically harder for traditional signal processing approaches, as most of them are built on some assumptions about the bottleneck link, which are not theoretically comprehensive. By contrast, FlowBot uses a Siamese model [25] for feature extraction, *i.e.,* each flow's signals are processed by the same model to extract its representation. The model is trained using triplet loss [32], which reduces the intra-group distance while increasing the inter-group distance. These representations are then clustered using the Unsupervised Nearest Neighbors (UNN) algorithm [33] to obtain the co-bottleneck groups.

We can set the number of output dimensions to be larger than previous work to increase the model's representation ability, and thus increase accuracy, but not too large, to keep the overhead low. We can use large datasets to cover many networking scenarios to produce a model with good generalization ability for real networks. We keep the measurement window as small as 1.5 s to ensure fast detection. Furthermore, FlowBot's double-margin triplet loss margin parameters can be used to inform the thresholds used for clustering, meaning that FlowBot representations are inherently normalized, and a given radius in the clustering algorithm can achieve a good balance between precision and recall.

B. Dataset Generation

A high-quality dataset is critical to train FlowBot for various scenarios. The features we choose for our dataset are OWD, RTT and, Short-term Loss Rate (SLR), which best characterize signals from the queue. To balance the variety and quality of the data, we design a family of network topologies and use the ns-3 network simulator [34] to obtain simulated data, monitoring the queues to determine ground truth labels for bottlenecks. We carefully construct training sets for both the binary congestion classifier (§ III-C) and the Siamese model (§ III-D). We describe dataset generation in § IV and V.

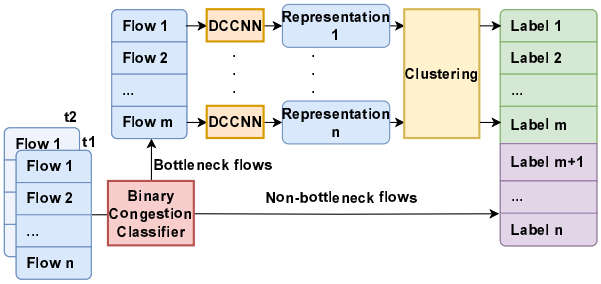

Fig. 1: Architecture of FlowBot.

C. Binary Congestion Classification

Though bottlenecked flows experience varied network characteristics due to queue variations over time and location, non-bottleneck flows usually have similar patterns, with low loss rates and relatively steady OWDs. We remove non-bottlenecked links before clustering to reduce the number of false positives. Since non-bottleneck flows have clear features, we use the flow statistics (*e.g.,* mean and variance) of OWD, RTT, and SLR as input features, build a binary congestion classifier using random forest, and construct a dataset for training and testing. We describe the dataset and training of the classifier in § V.

D. Model Training

FlowBot uses a Siamese model [25] with double-margin triplet loss [32].

Siamese model with DCCNN. To train a good representation for clustering, we use a Siamese model [25]. As a common technique for clustering and multiclass classification problem, Siamese model is a type of neural network that includes multiple identical sub-models. It learns the similarity among samples using triplet loss and gives a representation, making it a great fit for our flow clustering problem. Furthermore, Siamese models are generally well-suited to smaller datasets.

For each interval, the signals of each flow are fed into the CNN model to extract that flow's representation. Within the Siamese architecture, we use the Dilated Causal CNN (DCCNN) [35] as our model for feature extraction. DCCNN is an efficient CNN model that uses different dilations to keep track of sequential dependencies at different time scales. The exponentially increasing dilations enable the model to capture the long-term dependencies without having too many neurons. To further reduce the dimensionality, we use global average pooling instead of local max pooling.

The model architecture is shown in Table II. The input is of shape $[N_{batch}, L, n_{in}]$, where N_{batch} is the batch size (256 by default), L is the sequence length (300 for 1.5 s), and n_{in} is

TABLE II: DCCNN model architecture.

Layer(s)	Output Shape
Conv1d(n_{in}, 64, k=13, dilation=1), ReLU()	$(N_{batch}, 64, L_1)$
Conv1d(64, 64, k=13, dilation=2), ReLU()	$(N_{batch}, 64, L_2)$
Conv1d(64, 64, k=13, dilation=4), ReLU()	$(N_{batch}, 64, L_3)$
AvgPool1d(1), Flatten(), Linear(64, n_{out})	(N_{batch}, n_{out})

the number of input features, *i.e.,* two when OWD and SLR are used, and $n_{out} = 16$ is the representation dimensionality. Unlike in § III-C, we do not use RTT here, as RTT is much noisier than OWD in practice. We subtract the mean value from the OWD signal to avoid the impact of different path latencies.

Double-margin triplet loss. The loss function we use is a modified version of double-margin triplet loss [32].

$$L(a,p,n) = 0.5\ \mathrm{ReLU}(\alpha^2 - \min\{d(a,n), d(p,n)\}^2) + 0.5\ \mathrm{ReLU}(d(a,p)^2 - \beta^2), \quad (1)$$

where a, p, n are the output vectors of anchor, positive, and negative samples, respectively. Here $\alpha = 1.0, \beta = 0.2$ are distance thresholds that control the expected inter-cluster and intra-cluster distances, respectively. Compared to the original triplet loss, which only has one margin for the difference between $d(a,n)$ and $d(a,p)$, our loss has better control of cluster representations, and uses $\min\{d(a,n), d(p,n)\}$ to enforce a stricter intra-cluster distance. Compared to SBD algorithms [16], [17], our training produces representations with *relative* magnitude, where intra- and inter-cluster distances are normalized to α, β, and our radius for clustering can be chosen accordingly.

E. Clustering

The clustering algorithm is critical to accuracy when we have a large number of flows. In [17], Chinese Whispers, *i.e.,* unsupervised one-nearest-neighbor, is used: Each node is assigned the class label of the closest neighbor, except when the distance between them exceeds a threshold. This algorithm is fast, but each node can only explore one other node, and in practice tends to form smaller clusters when iteration ends, thus decreasing the detection's recall. Instead, we use unsupervised nearest neighbors [33], which uses more neighbors to determine one node's label. The algorithm has two parameters: number of neighbors $n_{neighbors}$ and radius r. For each node n, the algorithm first chooses $n_{neighbors}$ neighbors within radius r; then it labels n using the most common label among those neighbors. Since the inter-cluster distance varies depending on signal quality and model performance under different scenarios, we use $r = 0.4$ based on the intra- and inter-cluster distances from training loss in § V to remove the nodes that are too far away. We use $n_{neighbors} = \min\{9, a \cdot n_{flow}\}$, where a is 0.1 by default. The number of nearest neighbors is chosen to balance accuracy and speed: if it's too small (1–2), then it forms clusters are spread out, resulting in low recall; if it's too large, then (i) precision is lower; (ii) overhead is large and thus not scalable. When there are only a few flows and the cluster size is about 9, limiting the number of neighbors to 10% of all flows avoids precision degradation in such scenarios.

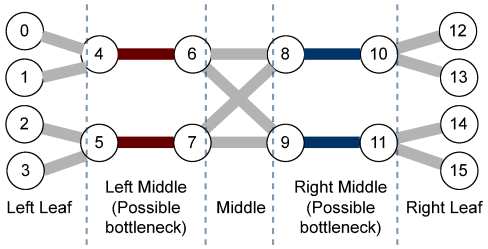

Fig. 2: Example Extended Dumbbell topology.

IV. Simulation

This section describes how we generate our dataset using ns-3 simulation [34].

A. Overview

To train a performant neural network to detect co-bottlenecks in the real Internet, we need to train on a dataset with sufficient variety with *correctly labeled* bottlenecks. As a result, we turn to simulations, because: (i) it is the most flexible approach for constructing various network topologies and observing the bottleneck queue to get ground-truth labels, even compared to software-defined networks, which can only deliver data in real-time (*i.e.,* one second of data takes one second to generate), and would still be largely constrained to experimenter-generated cross-traffic, and (ii) to the best of our knowledge, there is no existing real-world dataset that satisfies our requirements. Simulation allows us to control topologies and queues to obtain and change the ground truth, and allows us to generate a large amount of data across various scenarios. *Critically, though we train FlowBot using only simulation results, we evaluate FlowBot on the real Internet, so any lack of fidelity and representativeness of our simulations does not inherently limit FlowBot;* rather, FlowBot's Internet performance (§ VI-C) with real (not experimenter-generated) cross-traffic demonstrates that our simulations are *accurate enough* for our training needs.

B. Topology

We considered a number of possible topologies for our experiments. We could use *the classical dumbbell topology*, with a middle link constricted to form the co-bottleneck link, or with the leaf links constricted to create a bottleneck link; this topology creates clear ground truths, but also creates trivial outputs. We could use *a random topology*, with automatic inference of co-bottleneck ground truth based on queue backlogs, but such scenarios introduce the additional complexity of comparing queue sizes along each flow's path, and many flows may traverse no bottlenecks, or more than one bottleneck, complicating training. Instead, we combine these two schemes into the *Extended Dumbbell topology* (EDB): we extend the dumbbell topology, but ensure flexible settings of all parts of the flow path. A sample topology with 2x2 bottlenecks is shown in Fig. 2.

This topology combines two dumbbell structures with a fully connected middle section; as a result, there are five layers of links. The first layer is the left leaf links that connect left leaf senders (node 0–3) to the gateways (node 4 and 5); the second

layer is the left middle links, which are candidate server-side bottleneck links (link 4–6, 5–7); the third layer is the fully-connected middle links, emulating the core Internet (link 6–8, 7–8, 6–9, 7–9); the fourth layer is the right middle links, which are candidate client-side bottleneck links (link 8–10, 9–11); and the last layer is the right leaf links connecting gateways to the right leaf receivers (node 12–15). The mutable components of the topology include the number of left or right middle links, and the number of left or right leaf links for each gateway; we discuss these parameterizations in § IV-D. All link characteristics can be configured, as discussed in the § IV-C.

One important advantage of using EDB is that it provides a good abstraction of typical network topologies and bottleneck cases in CDN scenarios, yet is easy to control. We believe the simulated dataset generalizes because co-bottlenecked flows are identifiable from the characteristics of the bottlenecked flows, rather than the topology directly: (i) the model's input is time-series from each flow, which can be distinguished not by hop-by-hop changes but only by the aggregate path metrics, so the effect of link changes on dataset is not direct; (ii) the topology does affect the flow characteristics, but most of the impact on main signals (delay variation, SLR) is caused by the bottleneck link. These factors allow EDB to be a good enough abstraction to obtain flow-wise generalizability for our dataset.

The two most common bottleneck locations are server-side and client-side. Server-side bottlenecks can be caused by insufficient bandwidth resources and imbalanced load on the server, and client-side bottlenecks are typically caused by the last-mile ISP or an access link bandwidth limit. These bottlenecks are represented by left and right middle links respectively. The fixed bottleneck locations simplify monitoring of the queue backlog and bottleneck behavior control for co-bottleneck analysis.

C. Links

We set the bandwidth of non-bottleneck links to be large enough to support the simulated traffic load. The number and bandwidth of bottleneck links are chosen based on the scenario, with a smaller number and larger bandwidth for left bottlenecks. § V discusses these scenarios.

Varying path delay is critical for generating realistic data. With our topology, we choose delays so that each middle link models a long path through the core network, and all other links model a one-hop link, so that the overall OWD distribution is realistic for each flow. We first set the delay of one-hop links from a Gaussian distribution $N(0.5, 0.01)$, then sample the middle links' delay from $LogNormal(2.5, 1.0)$, which has 5, 50, and 95 percentile latencies of 1.7 ms, 12 ms, 90 ms respectively. We choose the access leaf link randomly from $[1, 3]$ for left leaf links and $[1, 5]$ for right leaf links. The bottleneck queue discipline is chosen randomly between PIE [36] and CoDel [37]. The queue size is chosen uniformly at random between 100 to 500 packets.

D. Flows

Each flow is used to model one user's video stream, so we choose a rate from a discrete distribution with probabilities set based on the measured proportions of YouTube played bitrates from Table III of [38]. Since we want to saturate bottleneck links, the number of users is chosen from a Normal distribution with mean and standard deviation set to achieve a certain level of congestion for a given cross traffic load. Each simulation lasts 30 s. To focus on scenarios where flows share the same link (regardless of whether or not they are co-bottlenecked), we start and end the flows within the first and last three seconds of the simulation, so that flows always overlap during the middle 24 s. The underlying transport protocol is TCP.

E. Cross traffic

We model cross traffic using Poisson Pareto Burst Process (PPBP) [39], [40], which sends bursts based on a Poisson process with rate λ, with each burst duration following a Pareto distribution determined by Hurst parameter H, and the mean duration T_{on}. Each burst is a flow with constant bitrate r. The average aggregate rate of the bursts is $r_{aggregated} = \lambda \cdot r \cdot T_{on}$. We set each flow to be a TCP flow with $r = 10$ Mbps, and set T_{on} to 547 ms for the each burst flow based on the measurements from [41]. The Hurst exponent H is randomly sampled from $[0.5, 0.9]$ [40], [42], [43]. Since T_{on}, r are determined, λ and $r_{aggregated}$ are calculated. We use a parameter r_{cross} to control the ratio of $r_{aggregated}$ to the total link bandwidth and then implicitly calculate $r_{aggregated}$ and λ accordingly. For each bottleneck link, we define congestion level as the ratio of the total required flow rates (including cross traffic) to the link bandwidth. Thus, the congestion level is determined by r_{cross} and the total user rate. In different scenarios, either r_{cross} or the total user rate can change, the details which are discussed in § V and shown in Tables III and V.

F. Ground truth

We monitor the queue lengths of each left and right middle link, which are the only potential bottleneck links, and then for each flow, we determine the bottleneck after the simulation. During a specific time interval, suppose the queue lengths of the left and right middle link are q_1, q_2 respectively. Let $q_{Th} = 50$ be the threshold of a significant bottleneck. If $q_1 < q_{Th}, q_2 < q_{Th}$, then the flow is labeled as non-bottlenecked; otherwise, the flow is labeled to be bottlenecked by the link with the larger backlog between q_1, q_2. We do not consider flows that traverse multiple bottlenecks as they are rare in large-scale networks; the benefit of such is limited but the detection cost is high. Thus, we do not consider scenarios having both $q_1 > q_{Th}, q_2 > q_{Th}$.

V. Detector Training

This section describes dataset generation and training for the non-bottleneck binary congestion classifier and the Siamese model for co-bottleneck detection.

Dataset generation. For each simulation run, we collect each flow's time series OWD, RTT, and SLR, and the queue backlogs of each left-middle and right-middle link, every 5 ms. The time series are divided into detection intervals (*e.g.,* 1.5 s). In each interval, we find the ground truth label for each flow as described in § IV-F. For the training set, we build triplets by

TABLE III: Training set settings ("-" for non-bottleneck).

Scenario	Left middle links & bw (bps)	Right middle links & bw (bps)	Congestion level	Cross traffic ratio
Left small	2∼4 , 300∼500 M	2∼6 , -	$N(1.15, 0.1)$	0.2∼0.8
Left large	2, 0.5∼1 Gbps	8∼16 , -	$N(1.15, 0.1)$	0.2∼0.8
Right small	2∼4, -	2∼6 , 150∼300 M	$N(1.15, 0.1)$	0.2∼0.8
Right large	2∼3, -	8∼16 , 150∼300 M	$N(1.15, 0.1)$	0.2∼0.8
Binary Classification	1, -	4∼16 , 150∼200 M	0.9∼1.05	0.5∼0.65

TABLE IV: Accuracy of classifiers for bottleneck classification.

	Naïve	Logistic Regression	KNN	SVM	Decision Tree	Random Forest
Bottleneck F1	0.79	0.21	0.89	0.74	0.87	0.88
Non-bottleneck F1	0.87	0.73	0.91	0.85	0.90	0.91

sampling two flows with the same label and one with a different label. We sample $2 \cdot \max\{2 \cdot n_{pos}, n_{neg}\}$ triplets for each cluster. For the test set, we prepare the flows' data and ground truth.

We construct two datasets: one for co-bottleneck detection, and a smaller one for non-bottleneck flow classification. Table III shows the settings, the first four rows being for co-bottleneck detection, and the last row for non-bottleneck flow classification.

Training sets. For co-bottleneck detection, we generated 540 scenarios, aiming to be more general with fewer constraints. We design two categories of scenarios: server-side bottlenecks (left bottlenecks), and client-side bottlenecks (right bottlenecks). Left bottlenecks scenarios emulate cases bottlenecked at the edge server due to limited bandwidth or poor load balancing, so we use fewer left middle links and more bottleneck bandwidth. Right bottlenecks scenarios emulate bottlenecks at the ISP or access links, so we use more right middle links and less bottleneck bandwidth. For all scenarios, we use a wide range of cross-traffic ratios and congestion levels to ensure data quality and variety for training. For non-bottleneck flow classification, we generated 60 runs, focusing on scenarios with unbottlenecked flows, with emphasis on diversity in links and congestion level.

Validation and test set. Table V and § VI describe our validation scenarios. For each scenario, we generate 24, 20, 20, 80, and 80 runs respectively.

Training the binary congestion classifier. For the classifier, we extract the four features from each interval: OWD_{std}, RTT_{std}, SLR_{mean}, SLR_{std}, and evaluated several classification models, as shown in Table IV. We choose Random Forest for higher non-bottleneck detection accuracy, to improve Siamese model performance.

Training the Siamese model. We train and test the Siamese model using the dataset described above, split with a 8:1 ratio. From Eq. 1, we define $\mathrm{ReLU}(\alpha^2 - \min\{d(a,n), d(p,n)\}^2)$ as the negative loss, and $\mathrm{ReLU}(d(a,p)^2 - \beta^2)$ as the positive loss. After convergence, the test loss is 0.206, with 0.233 positive loss and 0.178 negative loss, corresponding to intra- and inter-cluster distances of 0.525 and 0.907, respectively. The difference between the distances shows that our model is well trained. The distance threshold r (§ III-E) should be less than intra-cluster distance to avoid interference from other clusters; we find $r = 0.4$ works well in practice.

VI. Evaluation

We implemented FlowBot in Python which is available at https://github.com/PrinceS17/flowbot. This section introduces the experimental setup and presents the accuracy results of FlowBot under both ns-3 simulation and in real experiments.

A. Experiment Setup

Each experiment lasts 30 s, with flows starting in the first 3 s and ending in the last 3 s. In each detection interval, the detector predicts co-bottleneck groups for each flow, which we call *predicted flow labels*. We compare the predicted labels with ground truth labels to calculate accuracy. Each box aggregates all detection intervals across four runs of each network setting.

Benchmark algorithms. Among passive detection algorithms using only end-host signals, we choose DCW to represent correlation-based algorithms, and use SBD as it is the latest and best clustering-based algorithm. We use two variants of SBD, rmcatSBD and dcSBD, which use the same representations but cluster using threshold-based hierarchical clustering and Chinese Whispers respectively. To obtain a more comprehensive understanding of the characteristics of FlowBot as compared to previous approaches, we compare FlowBot with three benchmark algorithms: DCW, rmcatSBD, and dcSBD in both simulated and real tests.

We implement DCW as described in [18]. In our scenarios, the original threshold 0.512 is too low, so we choose a threshold of 0.8 to provide better precision and F1. We implement rmcatSBD and dcSBD as in [17], and parameterize them as in [44] and [17] respectively.

Metrics. We use F1, precision, and recall between predicted labels and ground truth labels as our accuracy metrics. Unlike in traditional classification, co-bottleneck detection only needs flows to share the same label, and the value of the label is irrelevant. Thus, we permute the predicted and truth labels to maximize the level of matching before calculating precision and recall.

For example, consider a scenario with ground truth clusters $C = \{c_1, \ldots, c_m\}$ and predicted clusters $C' = \{c'_1, \ldots, c'_n\}$, where each flow $i \in [1, N]$ has ground truth label $l_i \in C$ and predicted label $l'_i \in C'$. Let $\mathrm{Mo}_k, \mathrm{Mo}'_k$ be the modal label in in c_k, c'_k respectively, and $\mathrm{TP}_k, \mathrm{TP}'_k, \mathrm{FP}_k, \mathrm{FP}'_k, \mathrm{FN}_k, \mathrm{FN}'_k$ be true positives, false positives, and false negatives of c_k, c'_k respectively, *e.g.,* TP_k are the correctly grouped flows in c_k, *i.e.,* flows with $l'_i = \mathrm{Mo}'_k$. Then precision p and recall r are given by

$$p = \sum_{k=1}^{n} \frac{|c'_k|}{N} \frac{\mathrm{TP}'_k}{\mathrm{TP}'_k + \mathrm{FP}'_k} = \sum_{k=1}^{n} \frac{|\{i \in c'_k | l_i = \mathrm{Mo}_k\}|}{N} \quad (2)$$

$$r = \sum_{k=1}^{m} \frac{|c_k|}{N} \frac{\mathrm{TP}_k}{\mathrm{TP}_k + \mathrm{FN}_k} = \sum_{k=1}^{m} \frac{|\{i \in c_k | l'_i = \mathrm{Mo}'_k\}|}{N} \quad (3)$$

with $F1 = 2pr/(p + r)$. In real-world applications, precision means fewer incorrectly grouped non-bottlenecked flows, and

TABLE V: Test scenarios settings ("-" for non-bottleneck).

Scenario	Left middle links & bw (bps)	Right middle links & bw (bps)	Congestion level	Cross traffic ratio	Flows
Path lag	4, 200 M	4, -	1.2	0.3∼0.4	120
Cross traffic load	1, -	4 , 60∼120 M	1.1	0.2∼0.6	80
Overall link load	1, -	4 , 60∼180 M	0.8∼2.5	0.5∼0.8	40
Scalability w/ left bottlenecks	1∼4, 40∼770 M	4, -	1.2	0.3	50∼250
Scalability w/ right bottlenecks	1, -	4∼16, 20∼400 M	1.2	0.3	100∼500

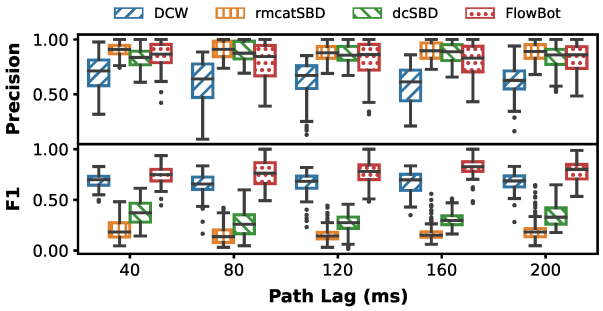

Fig. 3: Detector accuracy vs. path lag.

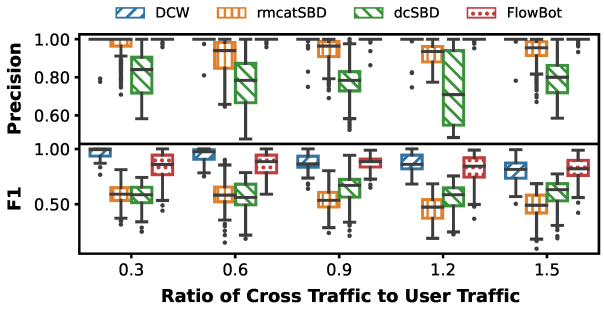

Fig. 4: Detector accuracy vs. cross-traffic load.

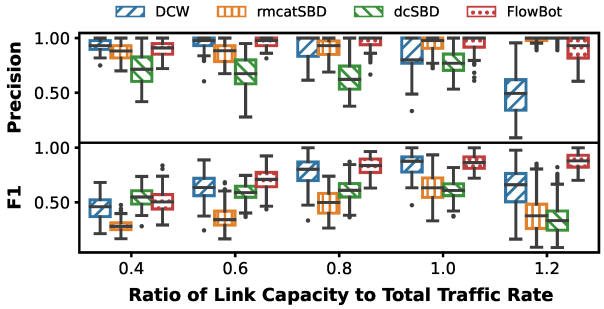

Fig. 5: Detector accuracy vs. link capacity.

recall describes better co-bottleneck detection. In many applications, precision is more critical as incorrect detection hurts normal flows. We report F1, precision, and recall separately to evaluate their suitability for various applications.

B. Simulation

We first evaluate FlowBot using ns-3 simulation, under network conditions that are harder to construct in real networks, to compare with the benchmark algorithms. We use the extended dumbbell topology (§ IV) with varying bottleneck link position and bandwidth, number of users, and cross traffic load. As described in Table V, we construct test scenarios varying several properties: path lag, cross traffic load, and bottleneck link load, as well as a scalability test. We also test the detection speed and overhead using the two largest scenarios from the scalability test. As stated in § I, we do not consider wireless links.

1) Path lag: To evaluate the impact of diverse path latency, we use a topology with four bottlenecked left middle links and four right middle links. Users traversing each bottleneck link either have 10 ms or 50–210 ms right leaf link delay. Fig. 3, shows that FlowBot has the best average F1 (over 0.7) and great precision (over 0.85). Also, increasing path lag does not have much impact on performance, except DCW's precision, because (i) correlation isn't degraded enough to break DCW's threshold 0.8; (ii) SBD's aggregate statistics are not sensitive to path lag; (iii) FlowBot is robust, with lower signal quality requirement than DCW due to varied training scenarios.

2) Varied cross traffic load: These scenarios vary the ratio of cross traffic to user traffic at a fixed congestion level of 1.1. We use four right bottleneck links, each with 20 users, each user averaging 5 Mbps, for a total rate of around 100 Mbps per bottleneck link. Fig. 4 shows that the four detectors from best to worst are DCW, FlowBot, dcSBD, and rmcatSBD, though DCW and FlowBot have slightly degraded performance with increasing cross traffic, because increased cross traffic comes with increased burstiness at the bottleneck queue, increasing the noise in the OWD signal.

3) Bottleneck link load: These scenarios evaluate the detectors by varying the load on each bottleneck. We use the same topology as the previous scenario, except that 10 users request 50 Mbps user traffic with 100 Mbps cross traffic. We vary each bottleneck link capacity from 60 Mbps to 180 Mbps to vary congestion levels, and plot the results in Fig. 5. The left side shows that all evaluated algorithms perform badly when the link is too congested, as the queue is always full, reducing signal quality. With increasing link capacity, FlowBot has the best F1 and precision, even when link capacity is higher than requested, showing an effective handoff from the congestion classifier to the DCCNN model.

4) Scalability test: These scenarios set either all left middle links, or all right middle links, to be bottleneck links. As in Table V, we vary the total number of flows and the number of parallel bottlenecks while fixing the congestion level at 1.2 on each link by varying the bottleneck bandwidth.

Varying bottleneck count. We vary the number of bottlenecks and plot the F1 scores in Fig. 6. FlowBot and DCW have better F1 (over 0.7 with 3+ bottlenecks), demonstrating stability at a larger scale, whereas SBD algorithms have a decreasing and much worse F1, showing that they are not suitable for large numbers of bottlenecks and flows.

Varying number of flows. We use 3 left and 12 right bottlenecks, varying the number of flows, and plot the results in Fig. 7. In the left-bottleneck scenario, (i) all algorithms perform worse with increasing flow count, though FlowBot remains the best and most stable, with F1 over 0.75 and precision close

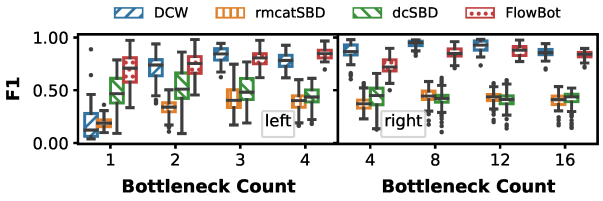

Fig. 6: F1 vs. number of bottlenecks with 150 flows for left bottlenecks, and 300 flows for right bottlenecks.

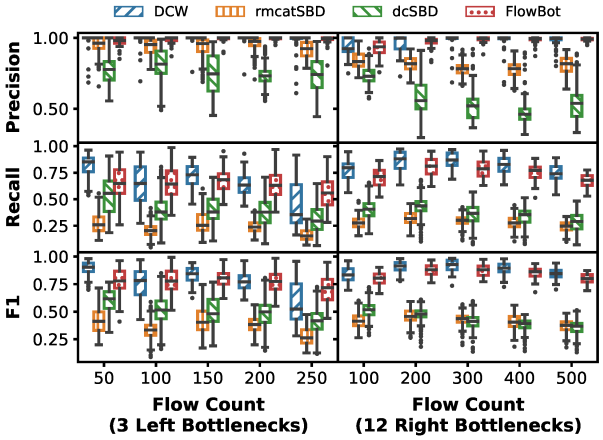

Fig. 7: Accuracy vs. number of flows with 3 left bottlenecks and 12 right bottlenecks.

to 1.0; (ii) SBDs become worse in both precision and recall, because their representations cannot support a large number of flows; (iii) DCW has good precision but its recall decreases significantly, showing that its fixed threshold cannot detect all co-bottlenecked flows when the cluster size is large. In the right-bottleneck scenario, FlowBot and DCW are still the two best algorithms, with precision near 1.0 and F1 over 0.75. SBDs have bad recall and even worse precision than in the left bottleneck scenario due to more bottlenecks.

5) Detection Speed: *Detection delay* represents the time from when flow i first enters a ground-truth label y to when the detector first begins consistently labeling it y (that is, as long as the detector continues to label it y for a period of time, *i.e.,* 3 s). This metric focuses on sensitivity; accuracy was previously discussed (§ VI-B4). Since SBD algorithms use a short interval (350 ms), we deem them to have consistently labeled a flow once they have correctly labeled the first interval, as long as three out of the next five labels continue to be correct. We measure detection time for the two largest scenarios with left and right bottlenecks from § VI-B4, and plot the results in Fig. 8. FlowBot performs best, with less than 3 s delay in detection, and DCW also does well. SBD algorithms are quite slow due to their low recall.

6) Computational Overhead: We consider *computational overhead* to be the fraction of a single core needed for running detection, which we compute as the single-core processing time per interval divided by the length of the interval. Fig. 9 shows that FlowBot and SBD algorithms have low overhead (around 10 cores for 1000 flows) because they use clustering algorithms with the same time complexity. DCW, which performs

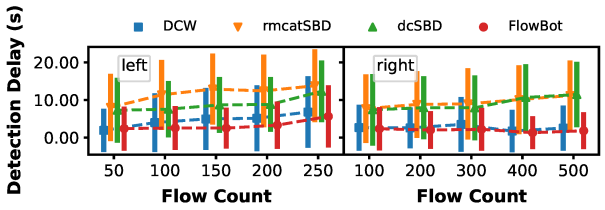

Fig. 8: Detection delay of FlowBot with four left bottlenecks and 16 right bottlenecks.

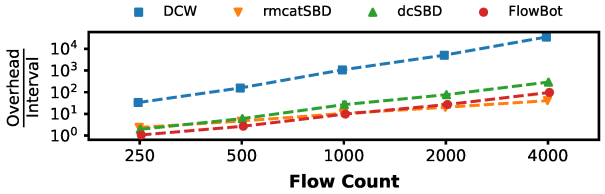

Fig. 9: Detection overhead vs. number of flows.

expensive pairwise comparison, has an overhead two orders of magnitude higher. Our results show that FlowBot has the highest accuracy and nearly the lowest overhead. The detector implementations in our experiments are not optimized, so real-world overhead can be further reduced.

C. Real Experiments

1) Setup: Our real-world testbed, illustrated in Fig. 10, has over 30 bottlenecks, with nodes drawn from GCP, CloudLab [27], and GENI [26]. The app servers in GCP send traffic to the clients in GENI; the bottlenecks are 100 Mbps last-mile links on GENI; and the cross traffic is generated by d-itg [45] running on CloudLab with bursty mode. We create GENI instances in different locations to provide different paths as listed in Table VI. Every 5 ms, we sample the OWD, and RTT, and compute the SLR for each flow from kernel-space data structures. We then send the time series data to the detectors for offline detection. We evaluate FlowBot using iperf3 and video streaming flows; Table VII lists our settings.

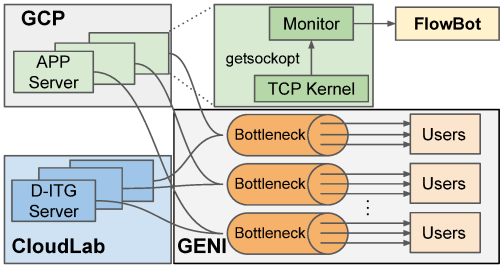

Fig. 10: Real-world experiment platform for FlowBot.

TABLE VI: Location of GENI testbeds in Internet experiments (d_1, d_2 are delays to GCP Iowa and CloudLab Utah in ms).

Location	d_1, d_2	Location	d_1, d_2	Location	d_1, d_2	Location	d_1, d_2
UCLA	48, 19	Stanford	48, 20	NYU	29, 52	Princeton	29, 52
Clemson	26, 54	Rutgers	27, 53	Cornell	35, 48	Case Western	41, 41
Georgia Tech	40, 54	Illinois	26, 36	Wisconsin	24, 38	Kentucky	20, 43
Kettering	22, 45	UMich	16, 39	UChicago	11, 34	Missouri	23, 26
VCU	27, 54	UW	44, 17	UKYMCV	21, 43	UTDallas	18, 35
Hawaii	99, 73	UCSD	49, 16	UMKC	29, 31	Sox	39, 55
OSU	19, 43	OhioMetro	21, 68	NYSERNet	40, 50	MOXI	18, 40
GPO	35, 56	Colorado	12, 12	CENIC	47, 21	Virginia Tech	30, 57

TABLE VII: Settings for Internet experiments.

Scenario	Bottlenecks	Flows per link	Flow rate (bps)	Cross traffic ratio
iperf3 (1)	4∼16	30	max 2.5 M	0.25
iperf3 (2)	16	20∼35	max 2.5 M	0.23∼0.6
Video (1)	4∼16	35	avg 2.3 M	0.23
Video (2)	30	20∼40	avg 2.8∼4.7 M	0.05
Partial bottlenecks	2∼8	5, 10	avg 6.1 M, 4.3 M	0

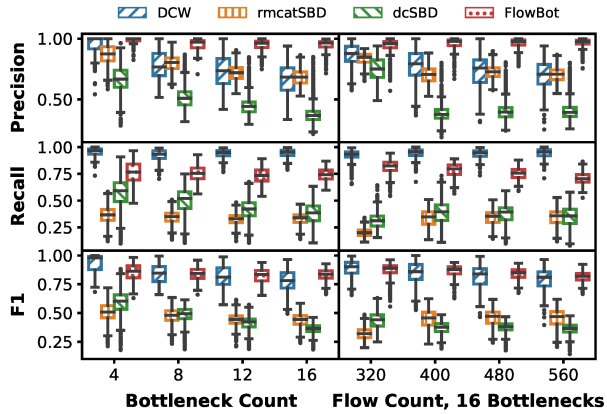

Fig. 11: Accuracy vs. number of bottlenecks (left) and flows (right) with iperf3 flows.

2) Test with iperf3: We use iperf3 to generate user traffic with a maximum bandwidth of 2.5 Mbps to simulate application-limited behavior. The results are shown in Fig. 11, with the left showing bottleneck count variation, and the right showing flow count variation.

Different number of bottlenecks. We first vary the number of bottlenecks. The left side of Fig. 11 shows that DCW and FlowBot have a similar F1, but FlowBot becomes the best (over 80%) with more bottlenecks. DCW has low precision and high recall, indicating that the signals from different co-bottleneck groups are too correlated. DCW and SBD's precisions decrease significantly with an increasing number of bottlenecks (over 30% worse), showing that they cannot accurately separate different clusters as the number of clusters increases.

Different number of flows. We then set 16 bottlenecks and vary the number of flows from 320–560. The right side of Fig. 11 shows that FlowBot outperforms the benchmark algorithms, with over 80% F1 and 95% precision, whereas dcSBD's precision drops to 60% at 560 flows, because its flow representations are insufficiently distinguishable.

3) Test with video streaming flows: We evaluate FlowBot with video streams to test its compatibility with realistic applications. We place an HTTP server [46] with dash.js 4.6.0 [47] in GCP, and stream Big Buck Bunny [48] from 144p to 1080p. Clients on GENI run dashc [49].

Varying number of bottlenecks and flows. Fig. 12 is divided into two parts; the results on the left show scenarios with 35 flows per bottleneck link, where the number of bottlenecks varies from 4–16; on the right, we set up 30 bottlenecks and vary the number of flows to up to 1200. DCW shows high precision

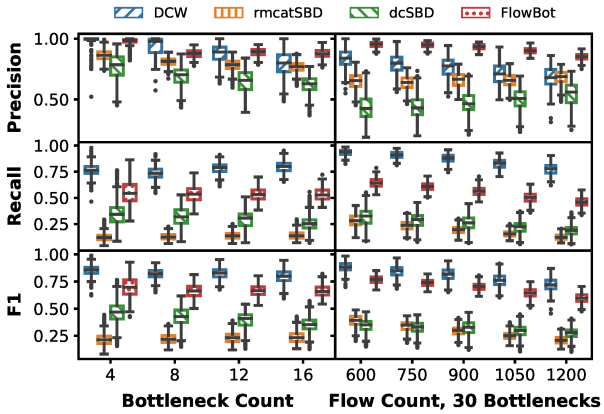

Fig. 12: Accuracy vs. number of bottlenecks (left) and flows (right) with video streams.

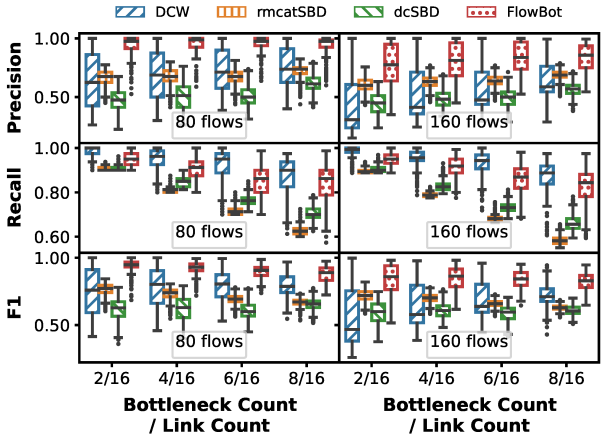

Fig. 13: Accuracy vs. partial bottlenecks with video streams.

but low recall, showing that video applications generate signals that are quite uncorrelated. DCW is the best in F1 and recall, while FlowBot has the best precision and second-best recall and F1 (90% precision and around 70% F1), with 5% better precision than DCW, and over 25% better recall and F1 than dcSBD. Overall, FlowBot provides strong precision and good F1 in all cases.

Partial bottleneck links. We further test FlowBot when not all links are bottlenecked, by varying the number of bottlenecks from 2 to 8 out of 16 total links; this reflects the real-world environment, where most flows are not bottlenecked. Fig. 13 shows that FlowBot has the highest precision and F1 (15% higher than others), and has consistently high precision no matter how many bottlenecks there are, showing strong robustness to environments where not all flows are bottlenecked.

4) Representation Analysis: This section analyzes the representations of FlowBot to evaluate clustering quality.

Representation quality. We first choose one interval from the four-bottleneck scenario, compress the 4-D and 16-D representations of SBD and FlowBot into a 2-D latent space using multi-dimensional scaling (MDS) [50], and plot them together with OWD in Fig. 14. SBD's representations do not separate clusters well, while FlowBot's representations do. The difference in representation quality accounts for FlowBot's over

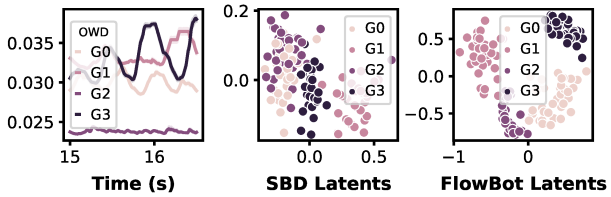

Fig. 14: OWDs and latents from video stream traffic (4 bottlenecks, 15–16.5 s).

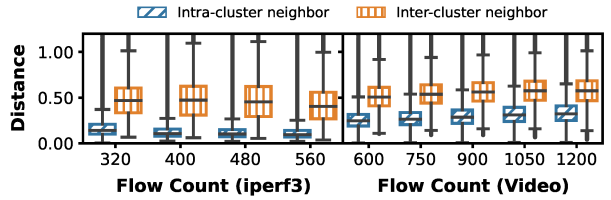

Fig. 15: Intra- and inter-cluster neighbor distances with iperf3 and video experiments.

20% improvement in recall and F1 in the four-bottleneck result in Fig. 12.

Scalability with more flows. For a given flow i of cluster C, we define *intra-cluster neighbor distance* / *inter-cluster neighbor distance* as the average distance between the representation of i and the nine closest neighbors from C and from the nearest cluster to C. We plot these distances for the experiments of varying number of flows in both iperf3 and video streaming scenarios, i.e., the right side of Fig. 11 and 12, shown in Fig. 15. The results demonstrate that there is a clear gap between the intra- and inter-cluster distances for most flows. The iperf3 experiment shows a good trend regardless of the number of flows; in the video experiment, the intra-cluster distances get a little worse with an increasing number of flows due to the bursty nature of video streams, but FlowBot's representation maintains a significant gap even with 1200 flows.

VII. Discussion

Due to resource constraints in simulation and real testbeds, our evaluations do not scale to clusters with multiple video servers. This section considers some aspects of FlowBot feasibility in such cases.

Larger flow count. § VI-B6 shows that FlowBot's overhead is linear in flow count and remains smallest compared to other techniques. There may be a limit to the number of flows FlowBot can accurately support due to the finite representation space, but § VI-C4 shows the trend of the difference between intra- and inter-cluster distances remains good, indicating that the limit would be reached at larger scales.

Higher bandwidth and signal quality. Higher-bandwidth links in CDNs reduce the granularity of the delay signal, making detection harder. However, if the queue length is proportional to the bandwidth-delay product (BDP) as is generally advised, the delay signal will reflect queue occupancy identically to a lower-bandwidth equal-latency link; *i.e.,* a 90%-full 1BDP queue with 1 ms latency will always have the same queueing delay whether the link is 1 Gbps or 100 Gbps. Furthermore, any degradation in signal quality impacts all algorithms, including FlowBot, and FlowBot's higher-quality representation and flexible training could help mitigate the reduction in signal quality.

VIII. Related Work

Table I lists the co-bottleneck detection algorithms reviewed here.

Correlation-based approaches. RTT correlation can be used to measure the similarity between flows traversing the same bottleneck link [20], as their RTTs should share the same bottleneck queueing delay component, which is a major component of RTT variation. In [20], the authors modify this approach by comparing the cross-correlation among flows with the auto-correlation of the flow itself, noting that the packets of different flows are more correlated than packets of the same flow because they interleave at the bottleneck queue.

Clustering-based approaches. Designed for MPTCP applications, SBD [16], [17] uses summary statistics such as skewness, variability, frequency of the OWD as well as the packet loss rate to represent each flow, and then clusters the statistics to detect the co-bottleneck groups. Specifically, SBD uses a time interval of 350 ms, and a long interval of M (typically 30) intervals, maintains the average OWD over short and long intervals, and calculates the statistics accordingly. SBD addresses the path lag problem by using summary statistics instead of the raw time series, but it incurs a long processing time especially at the start as enough samples are collected for a reliable decision.

Co-bottleneck detection using other signals. Inter-packet arrival time can be used for co-bottleneck detection [13], [19], [30], [31] Packets traversing the same bottleneck link should have a similar pattern of inter-packet arrival time, thus the detector can minimize the entropy of a group of observed inter-packet arrival time to find the clusters. Congestion interval variance [21] can be used since co-bottlenecked flows should have aligned congestion observation time, even with the existence of strong path lags. SB-FPS [22] uses ECN signals to detect co-bottleneck for coupled congestion control for MPTCP, since ECN-marked packets from the same bottleneck should arrive within a similar time interval.

IX. Conclusion

In this paper, we propose FlowBot, a novel co-bottleneck detector based on a Siamese model. By constructing a simulated dataset with sufficient variety, FlowBot can obtain a better flow representation, which it uses for clustering. Extensive evaluations in both simulated and real environments using GCP and GENI show that FlowBot can achieve a high detection accuracy and a fast detection speed without increasing the computational overhead, especially for large-scale network scenarios, when compared to correlation-based approaches such as DCW and clustering-based approach such as SBD.

Acknowledgments

We thank our shepherd Arvind Narayanan and the anonymous reviewers for their valuable comments which improved the paper. We thank Tiancheng Qin for advice on model training. This work is supported in part by Google Cloud Research Credits award GCP216419339.

REFERENCES

[1] Cisco, "Vni complete forecast highlights," 2018.

[2] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang, "Practical, real-time centralized control for cdn-based live video delivery," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 311–324.

[3] T. Stockhammer, "Dynamic adaptive streaming over http– standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*, 2011, pp. 133–144.

[4] V. Joseph and G. de Veciana, "Nova: Qoe-driven optimization of dash-based video delivery in networks," in *IEEE INFOCOM 2014-IEEE conference on computer communications*. IEEE, 2014, pp. 82–90.

[5] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye, "Analyzing the performance of an anycast cdn," in *Proceedings of the 2015 Internet Measurement Conference*, 2015, pp. 531–537.

[6] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen, "Server-based traffic shaping for stabilizing oscillating adaptive streaming players," in *Proceeding of the 23rd ACM workshop on network and operating systems support for digital audio and video*, 2013, pp. 19–24.

[7] F. Cangialosi, A. Narayan, P. Goyal, R. Mittal, M. Alizadeh, and H. Balakrishnan, "Site-to-site internet traffic control," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 574–589.

[8] B.-R. Chen, Z. Liu, J. Song, F. Zeng, Z. Zhu, S. P. K. Bachu, and Y.-C. Hu, "Flowtele: Remotely shaping traffic on internet-scale networks," in *18th International Conference on emerging Networking EXperiments and Technologies*, 2022.

[9] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 2020, pp. 1–6.

[10] Netflix, "Netflix open connect appliances," 2023 (accessed Aug 18, 2023). [Online]. Available: https://openconnect.netflix.com/en/appliances

[11] Google, "Youtube recommended upload encoding settings," 2023 (accessed Aug 18, 2023). [Online]. Available: https://support.google.com/youtube/answer/1722171?hl=en

[12] M. Luckie, A. Dhamdhere, D. Clark, B. Huffaker, and K. Claffy, "Challenges in inferring internet interdomain congestion," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014, pp. 15–22.

[13] N. Baranasuriya, V. Navda, V. N. Padmanabhan, and S. Gilbert, "Qprobe: Locating the bottleneck in cellular communication," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015, pp. 1–7.

[14] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy, "Inferring persistent interdomain congestion," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 1–15.

[15] N. Hu, L. Li, Z. M. Mao, P. Steenkiste, and J. Wang, "Locating internet bottlenecks: Algorithms, measurements, and implications," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 41–54, 2004.

[16] S. Ferlin, Ö. Alay, T. Dreibholz, D. A. Hayes, and M. Welzl, "Revisiting congestion control for multipath tcp with shared bottleneck detection," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[17] D. A. Hayes, M. Welzl, S. Ferlin, D. Ros, and S. Islam, "Online identification of groups of flows sharing a network bottleneck," *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 2229–2242, 2020.

[18] M. S. Kim, T. Kim, Y. Shin, S. S. Lam, and E. J. Powers, "A wavelet-based approach to detect shared congestion," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 293–306, 2004.

[19] D. Katabi, I. Bazzi, and X. Yang, "A passive approach for detecting shared bottlenecks," in *Proceedings Tenth International Conference on Computer Communications and Networks (Cat. No. 01EX495)*. IEEE, 2001, pp. 174–181.

[20] D. Rubenstein, J. Kurose, and D. Towsley, "Detecting shared congestion of flows via end-to-end measurement," *IEEE/ACM Transactions On Networking*, vol. 10, no. 3, pp. 381–395, 2002.

[21] W. Wei, Y. Wang, K. Xue, D. S. Wei, J. Han, and P. Hong, "Shared bottleneck detection based on congestion interval variance measurement," *IEEE Communications Letters*, vol. 22, no. 12, pp. 2467–2470, 2018.

[22] W. Wei, K. Xue, J. Han, D. S. Wei, and P. Hong, "Shared bottleneck-based congestion control and packet scheduling for multipath tcp," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 653–666, 2020.

[23] S. Islam, M. Welzl, and S. Gjessing, "Coupled Congestion Control for RTP Media," RFC 8699, Jan. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8699

[24] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," Tech. Rep., 2011.

[25] G. Koch, R. Zemel, R. Salakhutdinov *et al.*, "Siamese neural networks for one-shot image recognition," in *ICML deep learning workshop*, vol. 2, no. 1. Lille, 2015.

[26] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A federated testbed for innovative network experiments," *Computer Networks*, 2014.

[27] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The design and operation of cloudlab." in *USENIX Annual Technical Conference*, 2019, pp. 1–14.

[28] A. Aghdai, M. I.-C. Wang, Y. Xu, C. H.-P. Wenz, and H. J. Chao, "In-network congestion-aware load balancing at transport layer," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–6.

[29] Y. Jin, S. Renganathan, G. Ananthanarayanan, J. Jiang, V. N. Padmanabhan, M. Schroder, M. Calder, and A. Krishnamurthy, "Zooming in on wide-area latencies to a global cloud provider," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 104–116.

[30] D. Katabi and C. Blake, "Inferring congestion sharing and path characteristics from packet interarrival times," *Mass. Inst. Technol., Cambridge, MA, MIT-LCS-TR-828*, 2001.

[31] S. Sundaresan, N. Feamster, and R. Teixeira, "Home network or access link? locating last-mile downstream throughput bottlenecks," in *International Conference on Passive and Active Network Measurement*. Springer, 2016, pp. 111–123, this work determines if the bottleneck is in the home network or the access link. If access link, then inter-packet arrivals are smoothed, but otherwise it's bursty.

[32] K. Ho, J. Keuper, F.-J. Pfreundt, and M. Keuper, "Learning embeddings for image clustering: An empirical study of triplet loss approaches," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 87–94.

[33] O. Kramer, *Dimensionality reduction with unsupervised nearest neighbors*. Springer, 2013, vol. 51.

[34] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," *Modeling and tools for network simulation*, pp. 15–34, 2010.

[35] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[36] G. White and P. I. C. E. PIE, "Internet engineering task force (ietf) r. pan request for comments: 8033 p. natarajan category: Experimental cisco systems," in *PIE*, 2017.

[37] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," RFC 8289, Jan. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8289

[38] H. Nam, H. Schulzrinne, H. Nam, K.-H. Kim, H. Schulzrinne, M. Varela, H. Nam, H. Schulzrinne, T. Mäki, H. Nam *et al.*, "Youslow: What influences user abandonment behavior for internet video?" *Columbia University Rcp*, 2016.

[39] M. Zukerman, T. D. Neame, and R. G. Addie, "Internet traffic modeling and future technology implications," in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, vol. 1. IEEE, 2003, pp. 587–596.

[40] D. Ammar, T. Begin, and I. Guerin-Lassous, "A new tool for generating realistic internet traffic in ns-3," in *4th international ICST Conference on Simulation Tools and Techniques*, 2012.

[41] P. Jurkiewicz, G. Rzym, and P. Boryło, "Flow length and size distributions in campus internet traffic," *Computer Communications*, vol. 167, pp. 15–30, 2021.

[42] T. Karagiannis, M. Molle, and M. Faloutsos, "Long-range dependence ten years of internet traffic modeling," *IEEE internet computing*, vol. 8, no. 5, pp. 57–64, 2004.

[43] A. Biernacki, "Improving quality of adaptive video by traffic prediction with (f) arima models," *Journal of communications and networks*, vol. 19, no. 5, pp. 521–530, 2017.
[44] D. Hayes, S. Ferlin, M. Welzl, and K. Hiorth, "Shared Bottleneck Detection for Coupled Congestion Control for RTP Media," RFC 8382, Jun. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8382
[45] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.
[46] E. W. Server, "mongoose," 2023 (accessed Jan 9, 2023). [Online]. Available: https://github.com/cesanta/mongoose
[47] Dash-Industry-Forum, "dash.js," 2023 (accessed Jan 9, 2023). [Online]. Available: https://github.com/Dash-Industry-Forum/dash.js/
[48] Blender, "Big Buck Bunny," https://www.youtube.com/v/YE7VzlLtp-4, 2009, accessed April 9, 2023.
[49] A. Reviakin, A. H. Zahran, and C. J. Sreenan, "Dashc: A highly scalable client emulator for dash video," in *Proceedings of the 9th ACM Multimedia Systems Conference*, 2018, pp. 409–414.
[50] J. D. Carroll and P. Arabie, "Multidimensional scaling," *Measurement, judgment and decision making*, pp. 179–250, 1998.