Adaptive Network Protocol Selection: A Machine-Learning Approach

by

Hamidreza Anvari

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science
University of Alberta

Abstract

We introduce *optimization through protocol selection (OPS)* as a technique to improve bulk-data transfer on shared wide-area networks (WANs). Instead of tuning the parameters of a network protocol, our empirical results show that the selection of the protocol itself can result in up to four times higher throughput in some key cases. However, OPS for the foreground traffic (e.g., TCP CUBIC, TCP BBR) depends on knowledge about the network protocols used by the background traffic (i.e., other users). Yet, global knowledge can be difficult to obtain in a dynamic distributed system like a WAN.

Therefore, we introduce and evaluate a machine-learning (ML) approach for recognizing the background mix of protocols on a shared network. We build and empirically evaluate several ML classifiers, trained on local round-trip time (RTT) time-series data gathered using *passive probing* or *active probing*, to recognize the mix of TCP CUBIC versus TCP BBR congestion control algorithms (CCAs) in the background with an accuracy of up to 95%. Then, a decision process selects the best protocol to use for the new foreground transfer, so as to maximize throughput while maintaining fairness (i.e., OPS).

Lastly, we describe the design, implementation, and evaluation of iPerfOPS, the first tool that uses OPS to perform bulk-data transfer. The new tool is a substantially modified version of the well-known iPerf tool, and is an end-to-end implementation that incorporates previous research results. Our evaluation of iPerfOPS shows a bandwidth utilization close to the fair (i.e., equal) sharing if used with an appropriate probing pattern.

Preface

The research included in this thesis is an original contribution by 'Hamidreza Anvari'. Parts of this thesis have been published, as follows in chronological order:

1. H. Anvari and P. Lu, "Large transfers for data analytics on shared wide-area networks," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16, Como, Italy: Association for Computing Machinery, 2016, 418–423, ISBN: 9781450341288. DOI: 10.1145/2903150.2911718. [Online]. Available: https://doi.org/10.1145/2903150.2911718.

2. H. Anvari and P. Lu, "The impact of large-data transfers in shared wide-area networks: An empirical study," *Procedia Computer Science*, vol. 108, pp. 1702–1711, 2017, International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.05.211. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050917308049.

3. H. Anvari *et al.*, "Machine-learned classifiers for protocol selection on a shared network," in *Machine Learning for Networking*, É. Renault *et al.*, Eds., Cham: Springer International Publishing, 2019, pp. 98–116, ISBN: 978-3-030-19945-6.

4. H. Anvari and P. Lu, "Learning mixed traffic signatures in shared networks," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya *et al.*, Eds., Cham: Springer International Publishing, 2020, pp. 524–537.

5. H. Anvari and P. Lu, "Active probing for improved machine-learned recognition of network traffic," in *Machine Learning for Networking*, É. Renault *et al.*, Eds., Cham: Springer International Publishing, 2021, pp. 122–140.

6. H. Anvari and P. Lu, "Machine-learned recognition of network traffic for optimization through protocol selection," *Computers*, vol. 10, no. 6, 2021, ISSN: 2073-431X. DOI: 10.3390/computers10060076. [Online]. Available: https://www.mdpi.com/2073-431X/10/6/76.

7. H. Anvari and P. Lu, "iPerfOPS: A tool for machine learning-based optimization through protocol selection," in *Machine Learning for Networking*, Cham: Springer International Publishing, 2022.

In addition, over the course of implementing the outcome of this thesis as a software artifact, the following software code contributions has been published to the open-source software community:

1. *Iperf bug fix (pr#1112): Missing features in the library api*, https://github.com/esnet/iperf/pull/1112

2. *Iperf bug fix (pr#1113): Incorrect termination test in bytes/blocks mode*, https://github.com/esnet/iperf/pull/1113

3. *Iperf bug fix (pr#1114): Fix and optimize test termination condition check*, https://github.com/esnet/iperf/pull/1114

4. *Iperf bug fix (pr#1115): Incorrect data transfer in file mode*, https://github.com/esnet/iperf/pull/1115

5. *Iperf bug fix (pr#1116): Auto adjusting test-end condition in file mode*, https://github.com/esnet/iperf/pull/1116

6. *Iperf+: Extending iperf with reliable file transfer and network socket reuse*, https://github.com/hanvari/iperf-reliable

To Fatemeh, Hossain, and My Parents

Acknowledgements

I would like to sincerely thank my supervisor, Professor Paul Lu, for being a great advisor, and a good friend. Thank you for your valuable guidance, mentorship, and ongoing support, both at personal and academic levels, through my years of study.

I must also express my profound gratitude to my wife, Fatemeh Ghods, for providing me with constant love, support, and continuous encouragement. Also thank you to my son, Hossain, for being my constant source of joy and inspiration.

I am sincerely thankful to my parents, for their care, sacrifice, and unconditional love. Your encouragement and unparalleled support over the years has been a great source of motivation for me towards achieving my goals.

I should also thank all my dear friends from the Iranian community in Edmonton for their support and compassion throughout the years. Special thanks to MohammadHossein Bitarafan for his dedicated support and life-long friendship.

This research was supported through funding provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), Huawei, and ScotiaBank.

Table of Contents

List of Tables

List of Figures

Abbreviations

ALCF Argonne Leadership Computing Facility.

BBR Bottleneck-Bandwidth and Round-Trip Propagation Time.

BDP Bandwidth-Delay Product.

BG Background.

BIC Binary Increase Congestion Control.

BtlBW Bottleneck Bandwidth.

CCA Congestion Control Algorithm.

CTCP Compound TCP.

CWND Congestion Window.

dec-POMDP Decentralized Partially-Observable Markov Decision process.

DMZ Demilitarized Zone.

DTW Dynamic Time Warping.

FDT Fast Data Transfer.

FG Foreground.

FTP File Transfer Protocol.

Gb/s Gigabits per Second.

GNB Guassian Naive Bayes.

GridFTP Grid File Transfer Protocol.

iPerf-DT iPerf for Data Transfer.

iPerfOPS iPerf for OPS.

K-NN K Nearest Neighbors.

KNN K Nearest Neighbors.

LAN Local-Area Network.

LHC Large Hadron Collider.

LHCOPN LHC Optical Private Network.

LLNL Lawrence Livermore National Laboratory.

LoLa (TCP) Low Latency.

Mb/s Megabits per Second.

ML Machine Learning.

MLP Multi-layer Perceptron.

MPTCP Multi-Path TCP.

MS Milliseconds.

NB Naive Bayes.

NERSC National Energy Research Scientific Computing Center.

NFS Network File System.

NIC Network Interface Card.

OPS Optimization through Protocol Selection.

OSCARS On-demand Secure Circuits And Advance Reservation System.

PCC Performance-oriented Congestion Control.

PDS Parallel Data Streams.

PRR Proportional Rate Reduction.

QUIC Quick UDP Internet Connection.

RBF Radial Basis Functions.

RBUDP Reliable Blast UDP.

ReLU Rectified Linear Unit.

RTT Round-Trip Time.

SCTP Stream Control Transmission Protocol.

SKA Square Kilometer Array.

SLA Service Level Agreement.

SSH Secure Shell.

SVM Support Vector Machine.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UDT UDP-based Data Transfer.

VoIP Voice Over IP.

WAN Wide-Area Network.

WLCG Worldwide LHC Computing Grid.

Chapter 1

Introduction

Knowledge about the state of a data network can be used to achieve high performance. For example, knowledge about the protocols in use by background traffic might influence which protocol to choose for a new foreground data transfer. Unfortunately, global knowledge can be difficult to obtain in a dynamic distributed system like a wide-area network (WAN). Therefore, we introduce and evaluate a machine-learning (ML) approach to network performance, called *optimization through protocol selection (OPS)*, depicted in Figure 1.1. Using local round-trip time (RTT) time-series data (Step 1), a classifier predicts the mix of protocols in current use (Step 2). Then, a decision process selects the best protocol to use for the new foreground transfer (Step 3), so as to maximize throughput while maintaining fairness (Step 4).

Distributed, cloud-based, computing often requires dozens of gigabytes-to-terabytes of data (and more in the future, of course) to be transferred between local and cloud storage. Hybrid public–private clouds and clouds that span geographically separated data centers can often have WANs connecting key systems. If the data-transfer stages are bottlenecks, the overall data-processing pipeline will be affected. For example, a community of over 8,000 physicists across more than 40 countries have been served with advanced particle physics data, at the scale of petabytes per year, generated at the Large Hadron Collider (LHC) at CERN [14]. Similarly, the Square Kilometer Array (SKA) radio telescope project has generated data at approximately 400 Tb/s,

which is sent to a supercomputer center 1000 km away for analysis [15, 16].

High-performance protocols and tools exist for data transfer over WANs. For example, GridFTP [17], a well-known tool for transferring files across WANs, was developed as part of the LHC project at CERN. While much of the related work on high-performance tools has been in the context of dedicated (non-shared) networks, many networks are likely to be shared in practice, containing highly dynamic traffic workloads over any given period of time. Hence, prior performance evaluations on private networks could not characterize the mutual impact of a tool or protocol (foreground traffic) against other data streams (background traffic) on the same network path. Yet, there are many scenarios (e.g., cloud systems, or institutional networks) where not only are the administrators interested in how much performance is possible in the foreground, but also how large of an impact is made on background traffic. As a result, applying the existing data-transfer tools, without appropriate protocol selection and tuning, may result in unexpected behavior or sub-optimal performance on bandwidth-sharing networks.

The Transmission Control Protocol (TCP) is the backbone of the Internet because of its ability to reliably transfer data while sharing the network among competing data streams. TCP achieves this ability through its Congestion Control Algorithm (CCA), designed to adjust the data sending rate in response to variable network conditions. The early work on TCP CCA were mainly focused on two goals. First, managing the traffic flow to efficiently fill the network bandwidth while avoiding from overloading the network (i.e., congestion). Second, fair sharing of bandwidth against other TCP users on the network (TCP friendliness). While the proposed CCAs were adequate for the smaller bandwidths and lower latencies of the networks at the time, they proved inefficient with the emerge of high bandwidths, non-wired mediums, and larger scale of cross-country networks. In particular, there are well-known challenges with using TCP/IP on WANs for transferring large data files [18].

A variety of CCAs (a.k.a., TCP flavors) were developed over the past decades to

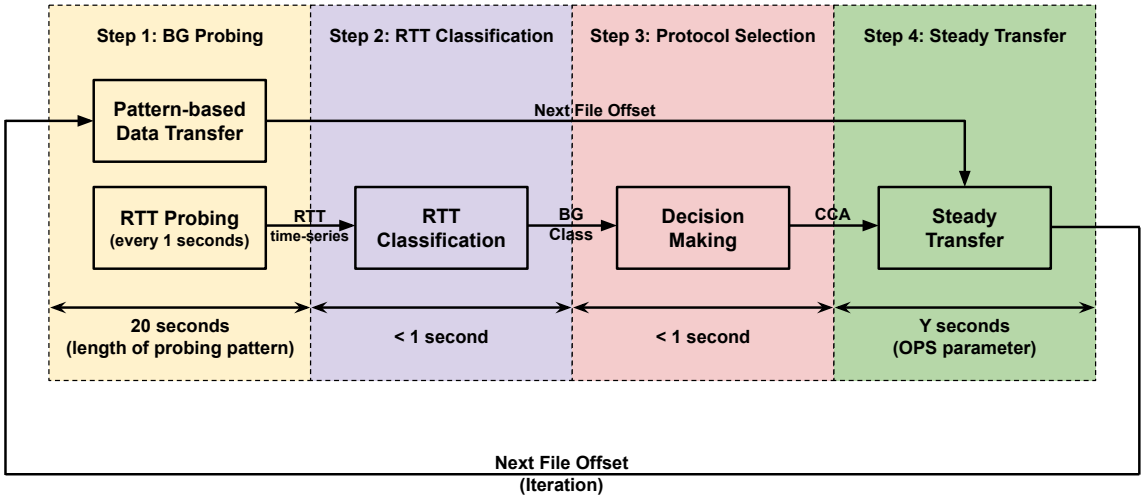


Figure 1.1: Iterative OPS operation cycle, comprised of four main steps: (1) Background Probing (active/passive), (2) RTT Classification, (3) Protocol Selection, (4) Steady Transfer (cf., Figure 6.1 for more details).

improve different aspects of TCP performance and fairness [19]. Despite proposal for several new CCAs over the past few years, TCP CUBIC [20] (with a few improvements) is still the dominant CCA on the Internet, because of its ability to efficiently utilize bandwidth on larger networks (performance) while remaining TCP-friendly (fairness). More recently, the researchers at Google introduced TCP BBR [21] as a performant CCA, gaining quick interest and popularity. Instead of blindly pushing the flow rate and reactively pulling off when packet loss happens, as in TCP CUBIC, TCP BBR probes, and aims at, the available bottleneck bandwidth, presenting promising end-to-end performance improvements on the network. Yet, several fairness concerns are reported on TCP BBR when it competes against TCP CUBIC (or other BBR streams) [22, 23]. several proposals proposed by the research community on addressing those inefficiencies with BBR [24–26].

The reported fairness and performance issues with competing streams of different TCP flavors (CCAs), yet again, emphasizes the importance of protocol selection on shared networks. Even though the particular issues with BBR could be transient and get addressed by the BBR v2 [27] [1], the overarching concept of potentially negative

[1]While BBR version 2 has been under development [28], at the time of conducting this research,

Table 1.1: Summary of chapters' (Ch.) topics and highlights.

Ch.	**Topic**	**Highlights**
1	Study introduction, thesis outline, and main contributions	ML-based BG classification, OPS for adaptive protocol selection
2	Background and related work on high-performance data transfer and shared networks.	Performance challenges on high-BDP networks, Bandwidth sharing could impact performance
3	Methodology, Emulated testbed, Traffic generators	Dumbbell topology, High-BDP paths, Bursty traffic patterns
4	Performance implications of bandwidth sharing on high-performance protocols (throughput, fairness)	Inconsistent performance and poor fairness of high-performance protocols (GridFTP vs. UDT) or CCAs (CUBIC vs. BBR)
5	Active/passive BG probing and ML-based recognition of protocol mixture on shared networks	Classifying mixtures of CUBIC and BBR with 95% accuracy using RTT time-series
6	OPS decision making strategy for adaptive protocol selection on shared networks	Significant throughput and fairness improvement (compared to worst case) with OPS
7	iPerfOPS: building an iPerf-based tool for OPS with active probing with interactive and script modes	Substantial modification to the well-known iPerf tool, performance close to equal bandwidth sharing using iPerfOPS
8	Concluding remarks, key observations, lessons learned, limitations and future work	Effectiveness of adaptive protocol selection / Future work: expansion of ML classifiers for more networks and protocols mixtures

mutual impact between different data streams and protocols on a shared network still stands. Nonetheless, global end-to-end knowledge about the of the background traffic can be difficult to obtain in a dynamic distributed system like a WAN.

In this research, outlined in Table 1.1, we first study the performance implications of using different data transfer tools and TCP CCAs on a shared network, in the presence of different profiles of background traffic. We then investigate applying ML techniques towards identifying the mixture of background traffic, building strategies and tools for data transfer optimization through protocol selection (i.e., OPS).

Following the introduction to our study in this chapter, Chapter 2 provides the background and some of the related work from research literature. It will discuss the high-performance protocols, as well as the fairness implications on a shared network. In Chapter 3 we introduce our experimental methodologies and the common testbed configurations used through the following chapters. We setup an emulated network with a dumbbell topology, allowing us to model competing streams on an end-to-end WAN path with a bottleneck bandwidth and longer RTT.

In Chapter 4, we compare the performance of a few well-known tools and CCAs (e.g., GridFTP, UDT [29], TCP CUBIC [20], TCP BBR [21]) over a bandwidth-sharing, emulated, WAN. We show that GridFTP and UDT, as representative high-performance data transfer tools, could unfairly affect the competing traffic on a shared network. In addition to its unfair traits (e.g., Figure 4.9), GridFTP could experience significant throughput drop in the presence of particular patterns of background traffic. For our benchmarking we study both synthetic and application-generated data-transfer workloads. A noteworthy result of our experiments is how poorly GridFTP performs when competing with UDP-based traffic. In particular, GridFTP's throughput drops to less than 10% of the theoretical maximum when competing with bursty UDP traffic (Figure 4.1). Furthermore, the effective throughput of TCP CUBIC when

BBR version 1 is still the only stable version publicly available; hence the one used in this study for all the experiments and evaluations.

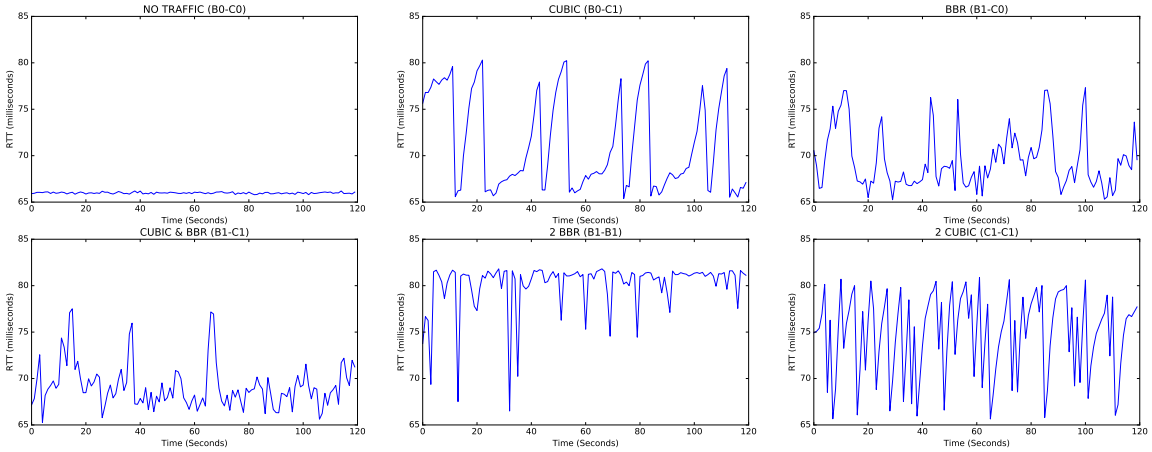


Figure 1.2: Samples of RTT time-series data generated using non-intrusive (passive) probing (cf., Chapter 5 for more details).

competing against TCP BBR could drop to less than 10% of its best, representing the aggressive impact of BBR on CUBIC (Figure 4.5).

Having shown the impact of the distinct background patterns on the performance of different protocols, in Chapter 5, we build a ML pipeline for identifying the mixture of the background workload on the network. Considering the mainstream use of TCP transport on the network, we choose TCP CUBIC versus TCP BBR as the scope for the ML task. First, we devise two end-to-end mechanisms, intrusive (active probing) and non-intrusive (passive probing), for capturing representations of the network signal(s) for distinct patterns of the background traffic. We use end-to-end latency, namely RTT, as a proxy for workload intensity on the network at any given time. Measured over regular time intervals it forms RTT time-series, allowing us to capture both the temporal pattern of the protocol mixture and the intensity of the background workload (Figure 1.2). Second, we examine several ML models, training classifiers for recognizing the background traffic on the network by taking the RTT time-series as the input. The classification performance results of up to 95% accuracy (Figure 5.7) shows the effectiveness of using RTT time-series for recognizing the background mixtures of TCP CUBIC, TCP BBR, or both.

To be clear, for this research we limit the scope of our study to TCP CUBIC and TCP BBR as the two popular CCAs on the data networks. Generalizing the classifier models for larger mixtures of streams and different CCAs and protocols (e.g., QUIC) is a direction for the future extension of this study. Furthermore, evident by the emergence history of new TCP CCAs, new CCAs are expected to be proposed in the future. Therefore, the mixture of CCAs on the actual networks would change over time. New classification models are needed to be trained as new CCAs get emerged or widespread over time.

In Chapter 6, we discuss OPS for optimizing the end-to-end data transfer performance. Using the trained ML classifiers, OPS recognizes the background traffic, followed by using a decision model to either pick TCP CUBIC or TCP BBR for data transfer (Figure 1.1). In OPS strategy we iteratively probe the RTT (Step 1), classify the RTT time-series (Step 2), and pick a protocol, CUBIC or BBR (Step 3), for transferring data (Step 4). Furthermore, we introduce *constructive* active probing where, instead of dummy traffic, we transfer real user data during the probing phase.

Our early evaluation of OPS shows up to four times improvement compared to the worst case for throughput (Figure 6.9). At the same time, the fairness measure (Jain Index) is improved by about 50% when OPS is used for protocol selection compared to the worst case for fairness (Table 6.3).

Following the promising results of our proof-of-concept for OPS, in Chapter 7 we describe the design, implementation, and evaluation of iPerfOPS tool as an end-to-end implementation of OPS strategy for bulk-data transfer. Using a substantially modified version of the well-known iPerf tool, iPerfOPS incorporates the findings and results of our previous chapters. iPerfOPS introduces, (1) a reliable data-transfer capability to iPerf, and (2) an end-to-end implementation of OPS. Our evaluation results for iPerfOPS shows the importance of the probing pattern when using (constructive) active probing, where selecting an appropriate probing pattern could result in a bandwidth utilization close to the fair (equal) sharing (Figure 7.10).

It is a challenging task to optimize a data-transfer task for both throughput and fairness. If throughput is the only deciding metric, we may use a greedy approach to probe each available CCA and to pick the one with maximum throughput for steady transfer. However, a greedy approach can hurt fairness. Unlike throughput, it is not feasible to measure fairness from a local end-point perspective. OPS enables optimization for throughput-and-fairness (i.e., both throughput and fairness) via its multi-step process for probing, discovering, and protocol selection, followed by the steady data transfer.

In a summary, the main contributions of this thesis are:

1. **A novel ML-based approach for adaptive protocol selection:**

   Knowledge about the protocols in use by the background traffic might greatly influence which protocol to choose for a new foreground data transfer. Yet, such a global knowledge can be difficult to obtain. We introduce and evaluate OPS, a novel ML-based approach to network performance. Using either passive or active probing, a classifier predicts the mix of TCP-based protocols in current use by the background workload. Then, we use that insight to build a decision process for selecting the best protocol to use for the new foreground transfer, so as to maximize throughput while maintaining fairness. The OPS approach's throughput is four times higher than that achieved with a sub-optimal protocol choice. Furthermore, the OPS approach has a Jain fairness index of 0.96 to 0.99, as compared to 0.60 to 0.62 if a sub-optimal protocol is selected (Section 6.5).

2. **An effective passive/active probing method for traffic recognition:**

   We introduce and provide a proof-of-concept of two network probing profiles, *passive probing* and *active probing*. These probing schemes enable us to gather representative end-to-end insights about the network, without any global knowledge present. Such an insight could potentially be utilized for different real-world use-cases, including workload discovery, protocol recognition, performance

profiling, and more. With *passive probing*, we measure local, end-to-end RTT in regular time intervals, forming time-series data. We show that such a non-intrusive probing strategy will result in distinct time-series signatures for different background protocol mixtures. *Active probing* is an intrusive extension of passive probing, adding a systematic and deliberate perturbation of traffic on a network for the purpose of gathering more distilled information. The time-series data generated by active probing improves our ability to distinguish the time-series for different workloads, evident by our ML evaluation (Section 5.6).

3. **Empirical quantification of the poor performance of the combinations of some popular network protocols on shared networks:**

   Many data networks are likely to share bandwidth, carrying data using a mix of protocols (both TCP-based and UDP-based). Yet most previous work on bulk-data transfer have (1) focused on TCP-only traffic, and (2) assumed a dedicated bandwidth, or negligible effect of background traffic. We detail an empirical study to quantify the throughput and fairness of high-performance protocols (TCP CUBIC, TCP BBR, UDP) and tools (GridFTP, UDT), in the context of a *shared network* (Section 4.1). Our mixed-protocol workload includes TCP-based foreground traffic with both TCP-based and UDP-based background traffic. In fact, our unique workload has led to interesting results.

1.1 Concluding Remarks

In this chapter we reviewed the motivation for our study, the performance implications on bandwidth-sharing networks. We introduced our thesis for OPS as an effective solution for adaptive protocol selection on shared networks, reviewing the high-level design for OPS (Figure 1.1) which we will investigate with more details in the following chapters. We also introduced some of the key contributions and highlight of the results for the following chapters.

Chapter 2

Background and Related Work

An important goal of many network protocols, exemplified by TCP, is to provide reliable data transfer over the network. The main technique for implementing reliability is via delivery acknowledgement and resend (a.k.a., re-transmit). If a data packet is not acknowledged by a specific time-out after sending, the sender will *resend* it. Upon receipt of an acknowledgement, the sender may increase the sending rate toward saturating the network capacity (Figure 2.1a) [30].

TCP wants to maximize the sending rate without causing unnecessary packet loss due to congestion, so the *congestion control algorithm (CCA)* of TCP adjusts the packet sending rate in an effort to match the available bandwidth. CCAs use a combination of a ramping-up strategy for the send rate, and a strategy for decreasing the send rate, based on particular events or factors. Examples of those factors include the receipt of acknowledgements for previously sent packets, when a packet loss is detected, when end-to-end latency is increasing, and more.

Most of network protocols, or CCAs, directly or indirectly, depend on RTT to adjust the sending rate and to pace putting data packets on the wire. On the one hand, the reactive CCAs such as CUBIC are Ack-clocked. Those (re-)adjust their `cwnd` every time they receive an acknowledgement (Figure 2.2). The time interval from sending a packet until receiving the corresponding acknowledgement is bounded by the end-to-end RTT. On the other hand, proactive CCAs such as Vegas or BBR

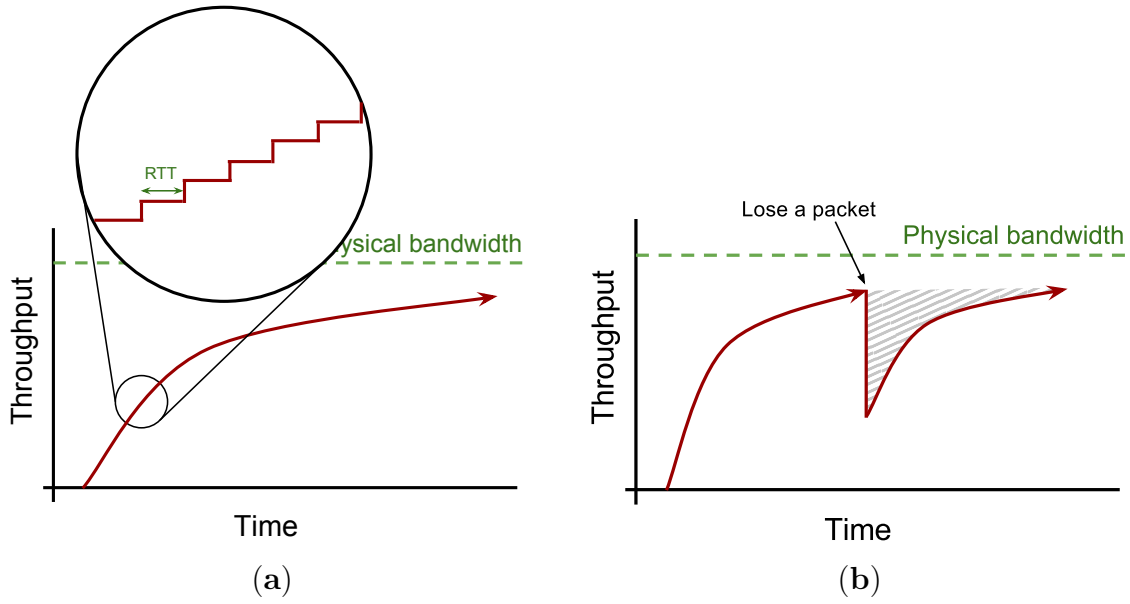


Figure 2.1: Congestion control behavior in TCP. (**a**) Additive Increase over Time. (**b**) Multiplicative Decrease in case of Packet Loss.

rely on the RTT variation as a signal of network congestion to adjust their sending rate.

One inherent challenge in designing CCA algorithms is to provide efficient bandwidth utilization [19, 31]. Based on the described packet delivery acknowledgement, for higher RTTs it takes longer for the CCA to increase the sending rate. In addition, the higher the bandwidth, the longer it takes to fully utilize the available physical bandwidth. As a result, the performance of classic network protocols can be sub-optimal in the networks with high bandwidth-delay product, so-called high-BDP networks, identified to have high capacity (i.e., bandwidth), and long latency (i.e., delay, represented by RTT) [32].

Moreover, a packet loss aggravates the low-utilization problem in high-BDP networks. For some CCAs, a lost packet is used as an indication that the network is overloaded, so the CCA drops the sending rate to avoid further network congestion (Figure 2.1b). It is worth noting that, in practice, data does become corrupted or is dropped due to router buffer overflows, contention, noise or other effects.

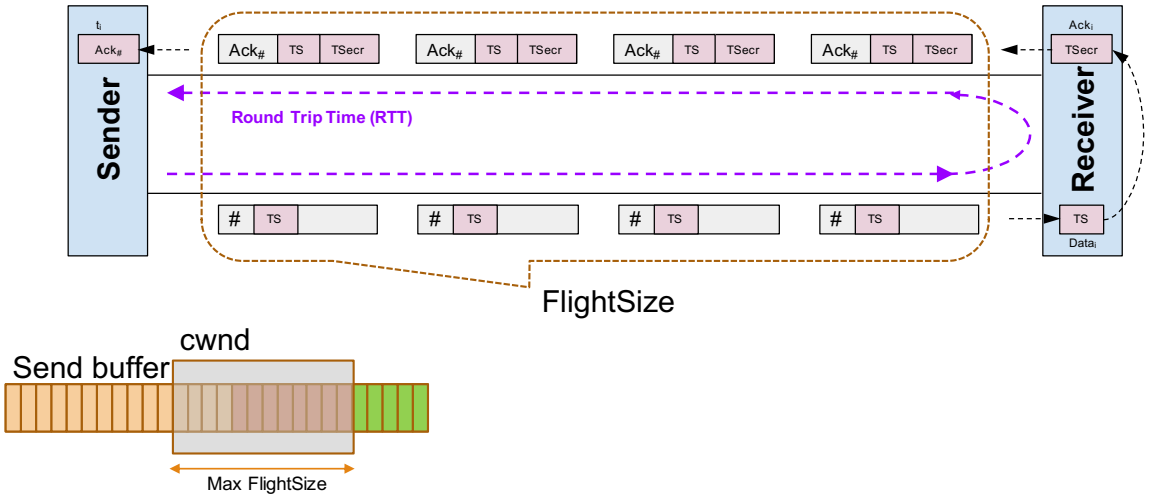


Figure 2.2: The basic TCP mechanism for tracking packet acknowledgements to adjust congestion window (`cwnd`) and to pace sending rate.

2.1 High-Performance Data-Transfer Techniques

Some common strategies to overcome performance issues in high-BDP networks are through (1) parallel TCP channels, (2) Custom UDP-based, reliable protocols, and (3) Enhancing a TCP CCA.

Parallelism and Data Striping: One of the most popular techniques to optimize data transfers over WANs is to establish multiple connections in parallel, striping data over them between two ends (Figure 2.3a). As a result, if some streams experience a decrease in the sending rate due to packet loss, the overall rate will not experience a significant drop due to other concurrent streams sending at full rate (Figure 2.3b) [33, 34]. This technique is well investigated and widely applied on top of the TCP protocol. There are several research projects which provide parallel protocol solutions for wide-area data transfer, ranging from prototypes to full-fledged tools and frameworks. GridFTP is one of the most popular data transfer protocols in this category [17]. On a private, non-shared network, TCP can be overly conservative, therefore GridFTP (in effect) parameterizes the aggressiveness of claiming bandwidth for a specific data transfer. Specifically, the number of parallel streams used by GridFTP is a run-time parameter. If the network is private, aggressiveness is not necessarily a problem.

Parallel Data Streams (PDS) is another tool that stripes data over either clear text (TCP) or encrypted (SSH) parallel connections [35]. Multi-path TCP (MPTCP) is another proposal to utilize multiple connections but in a different dimension. Rather than establishing parallel connections over the same network connection, MPTCP advocates for creating concurrent connections over different interfaces available to the host [36, 37].

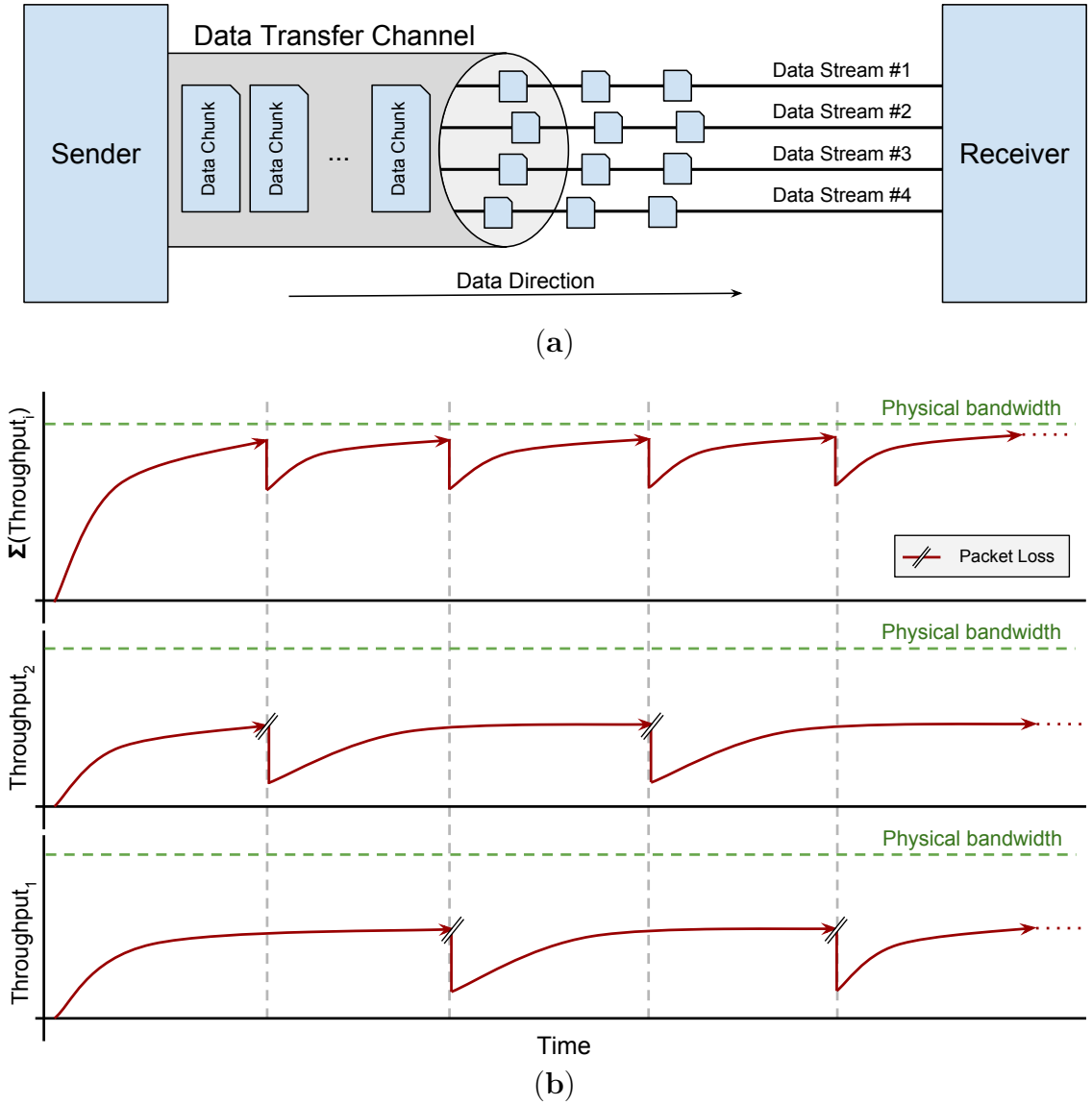


Figure 2.3: Schematic of Parallel Data Stream Behavior. (**a**) Connection Model. (**b**) Congestion Window Dynamic for 2-parallel-stream connection.

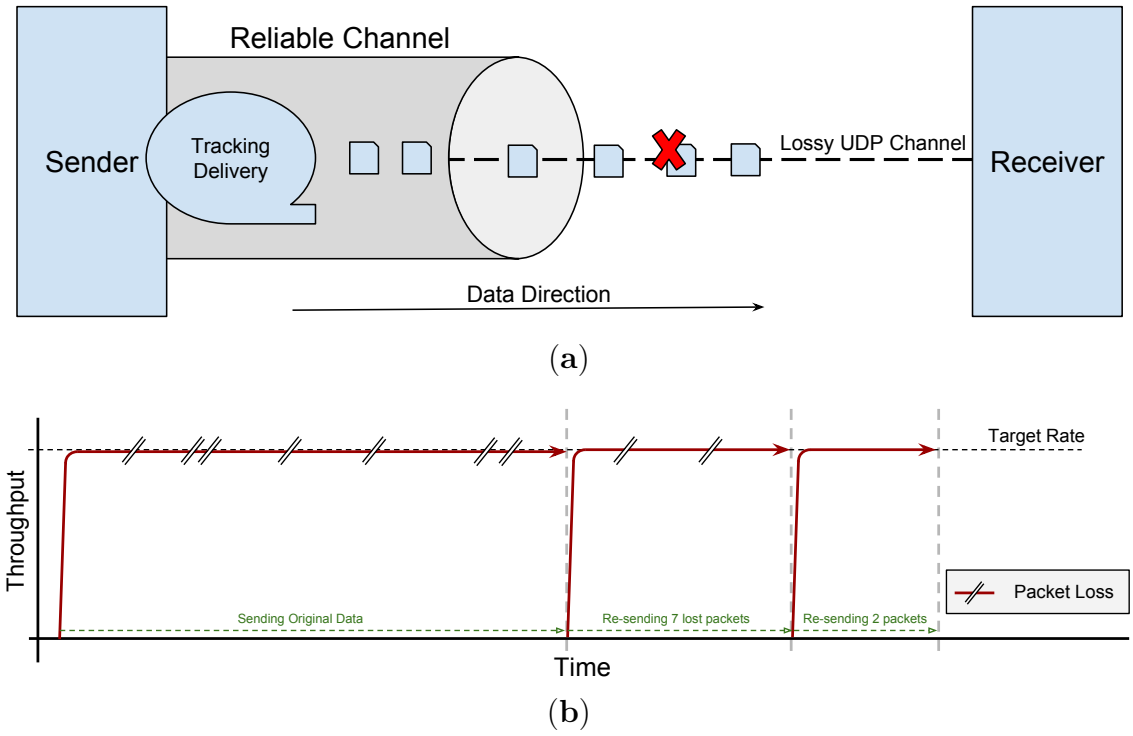


Figure 2.4: UDP-based Data Transfer. (**a**) Schematic of UDP-based Data Transfer. (**b**) A Sample of Congestion Control in UDP-based Tools: RBUDP.

UDP-based Protocols: Some reliable data-transfer protocols are based on unreliable UDP [29, 38, 39]. Arguably, these approaches use UDP because modifying the CCA within TCP is more restrictive. A new CCA within TCP might require kernel-level changes to either the sender or the receiver. Therefore, pragmatically, layering on top of UDP with a user-level implementation has been popular. UDT is one of the most popular data transfer protocols in this category [29].

The reliability of the data transfer is implemented as a user-level solution (Figure 2.4a). Compared to TCP-based tools, one of the strengths of this approach is the flexibility and wider range of design options, which provides a platform for applying different ideas and heuristics toward improving the data-transfer performance. As an example, the congestion control algorithm for one UDP-based protocol, Reliable Blast UDP (RBUDP) [39], is illustrated in Figure 2.4b. The pitfalls of all UDP-based approaches include, first, both end hosts must be configured by installing the user-level protocol

stack prior to data transfer. User-level changes are easier than kernel-level changes, but still require configuration. Second, sub-optimal design decisions and implementations may result in high resource requirements and CPU utilization from user-level procedures and task handlers. Furthermore, UDP-based protocols may not be able to take advantage of various hardware-based TCP offload engines and mechanisms.

Enhanced TCP CCAs: Another approach to solving the under-utilization problem is to modify TCP, improving the efficiency of CCA algorithms. This approach has been addressed by many research papers and several high-performance TCP variants were proposed, such as NewReno, High-speed TCP, BIC, Compound TCP (CTCP), CUBIC [40], Proportional Rate Reduction (PRR) [41], BBR [21], LoLa [42], and Copa [43]. Modifying TCP is effective and at the same time challenging, because of the possible compatibility problems with the middleboxes along the path and deployment challenges.

2.2 Bandwidth Utilization Models in WANs

One important property of computer networks is the distinction between *dedicated* and *shared* bandwidths. In this section, we briefly review common network utilization models.

Dedicated Networks. In this model, a private network is constructed between two or more end-points. This approach is only feasible for a niche sector in research or industry. Examples of this approach include Google's B4 private world-wide network [44] and Microsoft's private WAN resources [45].

Bandwidth Reservation in a Shared Network. Another approach to establish dedicated bandwidth, is to employ scheduling and bandwidth reservation techniques as a means to provide dedicated bandwidth over a common network path. For example, the On-demand Secure Circuits and Advance Reservation System (OSCARS) is one technique for bandwidth reservation [46]. An example of the state-of-the-art research in this category is the *On-demand Overlay Network*. This framework provides

dedicated bandwidth using bandwidth reservation techniques [47]. In this model, aggressiveness is a desired property for the network protocols for the efficient utilization of the reserved (dedicated) bandwidth.

Shared Bandwidth. In contrast to dedicated bandwidth, bandwidth-sharing networks are still the common practice for a large portion of research and industry users. Two common challenges are attributed to shared WANs. First, network resources are being shared between several users at a time, which results in dynamic behavior and workloads on the network. One consequence of this dynamic behavior is the emergence of periodic burstiness of the traffic over the network [48, 49]. This burstiness may result in various levels of contention for network resources, which could lead to an increased packet loss rate and therefore decreased bandwidth utilization. Second, in the networks with shared bandwidth, the criteria changes for an ideal data-transfer tool. In this case, aggressiveness is not necessarily considered a good quality and there are some implications which have to be taken into account. The protocols should provide a good trade-off between efficiency and being fair to other traffic (i.e, background traffic).

2.3 Does Background Traffic Matter?

Networks are not always private. Bandwidth-sharing networks are still the common case for a large number of research and industry users. The workload on shared networks tends to be highly dynamic. At any given time, a varying number of endpoints would be transporting traffic on the network. Each endpoint may be running one or more applications, concurrently communicating data on the network.

When utilizing the shared network, different users, and possibly different applications per user, may aim for distinct, sometimes contradicting, performance metrics. For example, a backup service trying to copy a database snapshot to a NFS-mounted network drive would prioritize throughput. At the same time a video-conferencing application, or a VoIP service, utilizing the same bandwidth would give priority to lower

latency and minimized jitter. Such a distinct requirements and performance criteria, would motivate different protocol settings and end-host configuration. For example, following the provided scenario, the backup service would use TCP transport, possibly conducting parallel TCP streams (as in tools such as GridFTP and Apache Hadoop's DistCp) to maximize the throughput. In contrast, the video-conferencing service may use UDP-based transport, optimized for low-latency communication as opposed to high-throughput. In such a condition, the use of different protocols and tools optimized for different use-cases could cause drastic impacts on each other, causing applications to experience degraded performance.

While the TCP transport tends to predominate the network traffic, different hosts might be configured with different, sometimes inconsistent, default CCA algorithms, causing degraded, potentially unfair, network allocation. In addition, from a bandwidth management point of view, the contention among end-hosts and applications could amplify the congestion situation, diminishing the aggregate bandwidth utilization while increasing end-to-end latency.

Furthermore, UDP-based traffic tends to increase by the presence and evolution of UDP-based data transfer tools (e.g., UDT, QUIC), as well as the emergence of prioritizing low-overhead, low-latency, communication over reliability and high-throughput (e.g., video conferencing, VoIP, streaming).

2.4 Fairness in Bandwidth-Sharing Networks

Fairness, at a high level of abstraction, is defined as a protocol's tendency and commitment toward utilizing an equal share of bandwidth with other concurrent traffic streams. In general, fairness is a desired quality for data transfer tools and protocols on a shared network. For example, in proposing new TCP CCAs the fairness against the other TCP variants is studied and referred to as TCP Friendliness.

In particular cases, when, and if, allowed by service level agreements (SLA), and for short periods of time, it is useful to have a tool that can aggressively utilize the

available bandwidth on the WAN, even if the sharing is unfair for a limited amount of time. The ability to utilize all of the available bandwidth is a matter of having the right mechanism or tool. Choosing to be unfair in sharing bandwidth is a matter of policy.

There are a number of proposals for quantifying fairness. *Jain fairness* is one well-known fairness index which serves as a measure of how fair are the traffic streams in sharing network resources [50]. The value of this metric falls between zero and one, representing the worst and the perfect fairness, respectively. The Jain index is calculated using the equation provided in Equation (2.1), where x_i represents the allocated resource (i.e., bandwidth) to the i-th consumer (i.e., traffic stream).

$$\mathcal{J}(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n. \sum_{i=1}^{n} x_i^2}. \tag{2.1}$$

2.5 Dynamic Workload and Latency Variation

The presence of background traffic of various patterns on the network could cause unexpected impacts on the throughput performance, both for plain TCP streams (iPerf) and high-performance data transfer tools (GridFTP and UDT).

This performance variation, could be attributed to the dynamic workload on the shared networks. The presence of competing workloads on the network could overwhelm the bottleneck link (i.e., the link on the path with the lowest bandwidth), causing the buffer build-up at the routers and increasing the end-to-end RTT. The constant addition and removal of data streams will result in RTT variation.

To showcase the variation in network latency (i.e., RTT) caused by dynamic workload on the shared networks, we conducted a simple end-to-en experiment, quantifying the effect of network congestion on latency, using the standard ping tool. We transport constant traffic on the network transferring data using TCP CUBIC, and at the same time transporting the square-wave pattern background traffic through the same bottleneck link. Figure 2.5 depicts the variation in end-to-end RTT on our testbed

over a 5 minutes time window. While the static RTT on the network is 128 ms, the observed latency experienced increases of up to 100 ms (i.e., between 128 ms and 228 ms) as a result of network congestion. This increased RTT could have a further negative impact on the network utilization. The study of RTT dynamics in shared networks and its impact on performance metrics constitutes one of the interesting directions for extending this study.

As earlier discussed, most of the network protocols, or CCAs, directly or indirectly, rely on RTT as a signal to adjust the sending rate and to pace putting data packets on the wire. On the one hand, the reactive CCAs such as CUBIC are Ack-clocked. Those (re-)adjust their `cwnd` every time they receive an acknowledgement. The time interval from sending a packet until receiving the corresponding acknowledgement is bounded by the end-to-end RTT. On the other hand, proactive CCAs such as Vegas or BBR rely on the RTT variation as a signal of network congestion to adjust their sending rate.

In either case, the observed RTT variation would cause inconsistent impact on CCA algorithms, making under- or over-estimation of the RTT. As a result, different CCAs, depending on their internal logic and their reliance on the RTT, will manifest different types of performance degradation.

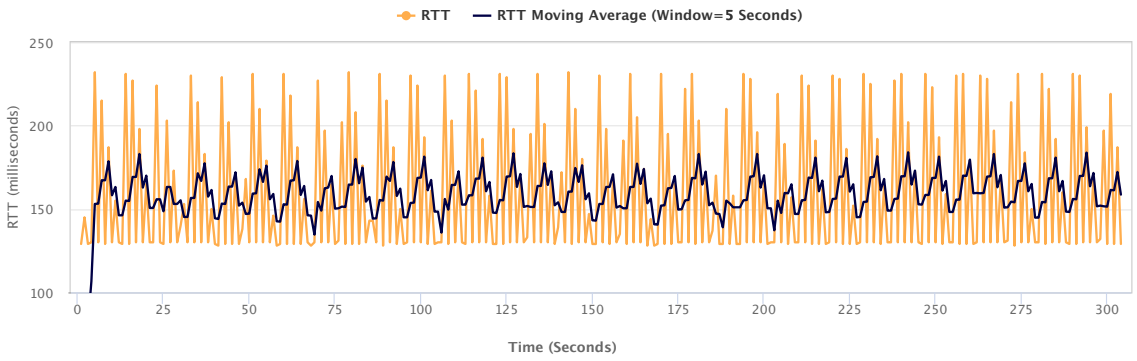


Figure 2.5: RTT Variation (via ping) over Time, Due to Congestion.

2.6 Modelling the Network Topology

Network topology defines how end-nodes are connected through network links and intermediate nodes. The layout, capacity, and bandwidth of the intermediate elements in a path will impact the performance of the network traffic. Therefore, it is important to identify and appropriately model network topology for studying various aspects of a networked system.

Depending on the type of the study, different levels of abstraction might be needed to build representative topology models. For example, in end-to-end performance studies, a simple dumbbell model would represent a network path, including an end-to-end RTT and a global bottleneck bandwidth as the contributing factors to a data stream performance [51]. Alternatively, in performance studies where internal network signals are in use, a more detailed network model might be needed to appropriately distinguish the properties of different network segments and hops (RTT, Bandwidth, contention, etc.) [52].

2.7 High-Performance Data-Transfer Tools

The literature on high-performance, bulk-data transfers on WANs can be categorized into three main areas: (1) application-layer data transfer tools, (2) high-performance TCP variants, and (3) network design patterns.

2.7.1 Application Layer: Large Data-Transfer Tools

Application-layer bulk-data transfer tools run on top of the underlying transport protocols without any modifications to the base protocols. These tools are based on either TCP or UDP transport protocols.

TCP-based Tools

As we mentioned earlier, GridFTP is the most well-known and widely used TCP-based tool. It extends the File Transfer Protocol (FTP) to provide a secure, reli-

able, and high throughput file-transfer tool based on parallel TCP connections. The three most important parameters of GridFTP are (1) pipelining for transferring large datasets consisting of small files, (2) parallelism for achieving multiples of the throughput of a single stream for a big file transfer, and (3) concurrency for sending multiple files simultaneously through the network channel [17].

GridFTP originally used a naive round-robin scheduling approach to assign application data blocks to outgoing TCP streams. Thus, different transmission rates of individual TCP streams will cause significant data block reordering and requires a large buffer memory at the receiver to handle out-of-order data blocks. A weighted round robin scheduling algorithm is proposed in [53] to solve this problem while preserving the throughput of parallel TCP connections.

Determining the optimal number of TCP streams is also another challenge addressed by several proposals [33, 54, 55]. Those proposals try to find a model that estimates the throughput of data transfer based on RTT, message size, and packet-loss rate of each TCP stream. Then, an optimization process is applied for maximizing the throughput, towards finding the optimal number of parallel TCP connections.

There exist other TCP-based high-performance tools. Those tools mainly use the same technique of establishing parallel TCP connections to improve throughput performance. Fast Data Transfer (FDT) [56] and BBCP [57] are two of the popular tools in this category.

UDP-based Tools

The most important challenge of using UDP for large data transmissions in WANs is that it is not a reliable protocol. It means that there is no guarantee that the data is completely received at the destination. To solve this, several research works propose implementing UDP-based tools such as UDT [58] and RBUDP [39] that utilize UDP transport while implementing reliability at application layer.

2.7.2 Network Design Patterns

The Science DMZ [59] is a network design pattern to address different bottlenecks for data-intensive science. The authors separate the network to LANs and WANs. Since LANs are general-purpose, they add some overheads to bulk-data transfers (e.g., firewalls). To reduce these kinds of overheads, they separate the path of bulk-data traffic from the other traffic in the network and use a set of dedicated nodes called Data Transfer Nodes (DTNs) along the path to fully optimized them for high-performance data transfers. Together, the design pattern and tools form a set of best practices for high-performance data transfers across WANs, and how to implement the interface with the LANs are often the destination for the data.

The Science DMZ design pattern has been implemented at various institutions, including the University of Florida, the University of Colorado, the Pennsylvania State University and National Energy Research Scientific Computing Center (NERSC) [60].

2.8 Effects of Sharing Bandwidth

While a significant part of literature concerns the performance optimization of data communication over dedicated networks, there are a relatively small number of research projects in which, directly or indirectly, the network environment is considered to be shared. One example of such research projects is the study of distributed systems in the context of shared network environments [61]. This work argues the importance of considering background traffic when evaluating distributed systems which are deployed on top of a shared network.

Another area of research which has considered the shared nature of networks is the research of estimating available bandwidth, where the notion of cross traffic (e.g., a mix of foreground and background traffic) has been investigated [62].

2.9 Traffic Generators

While a parameterized synthetic traffic generator could represent a wide range of states in terms of traffic patterns and other characteristics, it is desirable to expose tools and techniques to the mixture of data traffic running across real networks. There are a range of tools in the literature, which are developed to capture the packet traffic going across a network, and to replay the recorded trace in another environment. Such a technique is quite useful to run traffic patterns from the real world, and at the same time being able to reproduce the exact same patterns using the same set of traces. There are a large number of research projects which investigate the problem of capturing and generating network traffic, either to serve as workload in evaluation of end-point systems or to generate traffic in network studies. One category of such tools are designed to capture the observed network traffic and to replay the exact same trace in the future, such as Epload [63]. Another interesting group of projects in this category concern creating representative models of network traffic. Such a model is then used to generate traffic patterns which adhere to the statistical characteristics of captured traces and, at the same time, are not exactly replaying the same trace [64, 65].

2.10 Big Data Transfer: High-Profile Use Cases

An increasing number of real-world projects exist in which timely high-performance data transfer is essential to project efficiency and success. These projects usually involve gathering or generating large amounts of data which needs to be transferred to high-performance computing and research centers around the world for processing.

In the area of earth system and climate change studies (http://www.ipcc.ch/index.htm accessed on 8 June 2021), the Coupled Model Intercomparison Project (CMIP-5) generates large amounts of data which incorporates collecting environmental measurements of temperature, carbon emission, and other factors. This data, at the scale of

terabytes and petabytes, is to be communicated with several research groups around the world. In a particular case, a 10 TB subset of this dataset was collected from three sources, namely ALCF, LLNL, and NERSC, and transferred to Portland, Oregon over a 20 Gb/s network [66]. The maximum achieved bandwidth is reported as 15 Gb/s using GridFTP over a dedicated (virtual circuit) network.

In another case, the Large Hadron Collider (LHC) project from the high-energy physics community relies on GridFTP for sharing the collected collisions data with research centers around the world over its computing grid infrastructure [67]. A dedicated optical-fiber network, called LHC Optical Private Network (LHCOPN), connects CERN to each of the Tier 1 centers around the world. This network provides near real-time access to LHC generated data, which scales up to 15 petabytes for every year. The Worldwide LHC Computing Grid (WLCG) is claimed to be the world's largest computing grid [14].

2.11 Machine Learning for Networking

Two common challenges in networking problems include, first, probing the network to discover the existing workload and available resources, and, second, efficient utilization of the available resources (i.e., bandwidth). Probing the available network resources and workload is a key insight for the efficient utilization of the network. This probing is ideally performed from an endpoint perspective, viewing the network as a black box. Estimations could range from probing available bandwidth, to inferring the workload and the type of the traffic on the network. There are a number of studies in this area, mainly investigating the possibility of applying ML techniques for bandwidth estimation in high-speed networks [62, 68]. In a different vein, ML techniques have been used for designing or optimizing network protocols. Remy used simulation and ML to create a new TCP CCA, via a decentralized partially-observable Markov decision process (dec-POMDP) [51]. The resulting RemyCC is a look-up table to help end-nodes decide to either send data packets to the network or wait [69].

Performance-oriented Congestion Control (PCC) is another recent study where an online learning approach is incorporated into the structure of the TCP CCA [70]. Online convex optimization is applied to design a rate-control algorithm for TCP, called Vivace [71].

2.12 Concluding Remarks

In this chapter we have reviewed the background and related work for this study. We reviewed the basics of flow control and reliable data transfer in TCP, followed by discussing well-known techniques and tools for high-performance data transfer on WANs. In addition to the throughput performance of data transfer, we discussed the fairness considerations on a bandwidth-sharing network. Following the concepts discussed here, in the next chapter we will introduce the networking testbed and our methodology for evaluating protocols on an emulated shared WAN.

Chapter 3

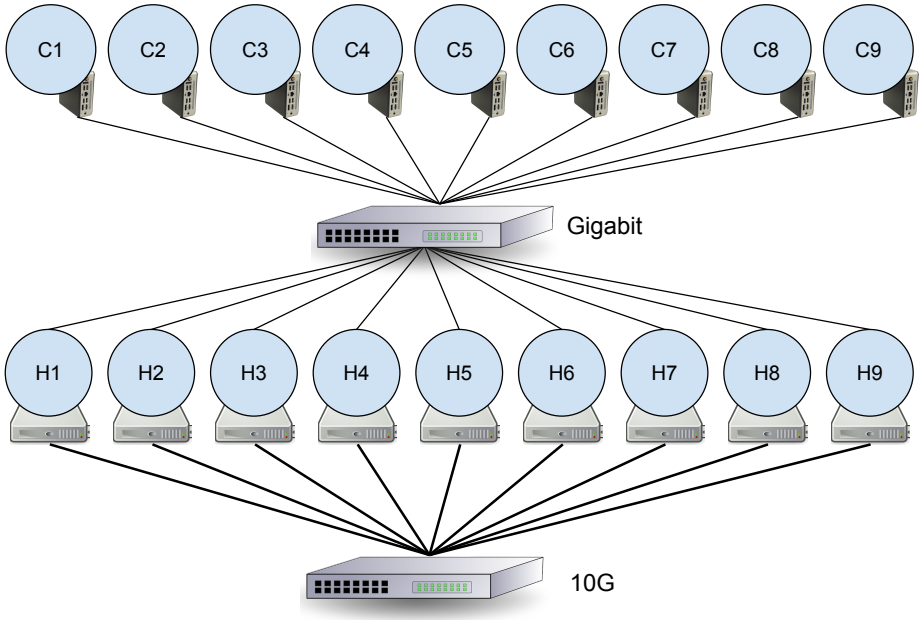
Testbed Network Design and Methodology

In this chapter, we introduce our methodology to investigate the performance impacts of cross traffic on bulk-data transfer in the context of shared high-BDP networks. Our methodology consists of the set up and configuration of the testbed as well as designing test scenarios for conducting background traffic across the network.

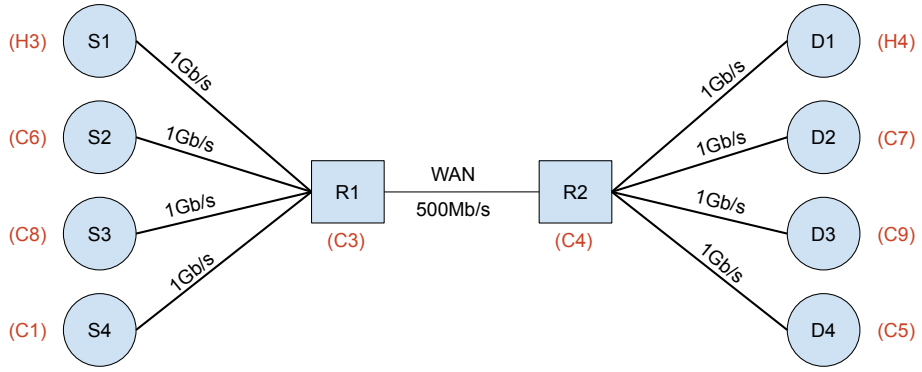
There are some simplifications in our current methodology. First, we limit our study to the well-known dumbbell network topology. Second, we only use a maximum of two streams of background traffic (of different protocols). A study of larger numbers of background streams remains to be done in the future work. Third, we limit the type of background traffic to constant or bursty synthetic patterns. Web traffic or other application-traffic patterns are also outside the scope of this research.

3.1 Testbed Configuration

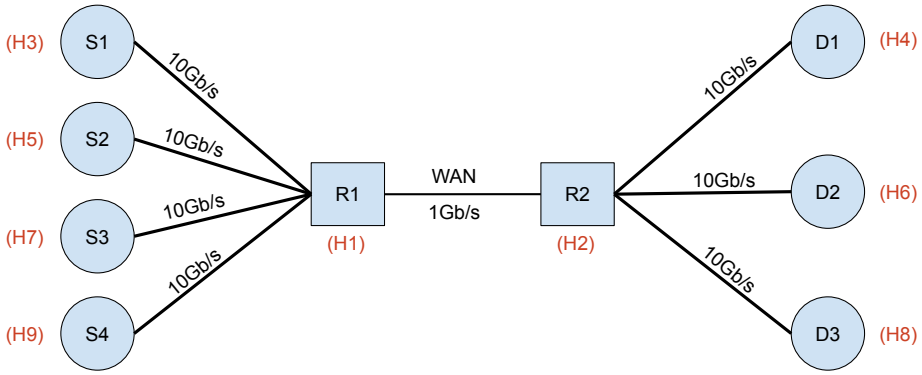
While the ideal environment for our network experiments is a real WAN, it does not offer enough control for the exploration and tuning of different network parameters and to study their effect. Therefore, we designed an emulated high-BDP testbed, implemented on a research computing cluster at the University of Alberta. The experiments are being conducted over a physical network, making sure we capture the full impact of the network stack at the end-nodes, while emulating the bandwidth



(a)



(b)



(c)

Figure 3.1: Testbed Network Architecture. (**a**) Physical Network Configuration. (**b**) Overlay-1: Logical Dumbbell Network (BW = 1 Gb/s, BtlBW = 500 Mb/s). (**c**) Overlay-2: Logical Dumbbell Network (BW = 10 Gb/s, BtlBW = 1 Gb/s).

capacity and RTT delay for the high-BDP bottleneck link (Figure 3.1). Our physical testbed configuration is depicted in Figure 3.1a. All nodes are connected using a 1 Gb/s connection, while a subset of nodes are also interconnected using a 10 Gb/s connection over a separate NIC interface.

Our high-BDP testbed employs a dumbbell topology, implemented as a logical overlay network over our physical infrastructure. We deploy two configurations of the overlay network: *Overlay-1* (Figure 3.1b), and *Overlay-2* (Figure 3.1c). The network specification for the two overlay networks are provided in Figure 3.2a. Each overlay network consists of 7 or 8 end-nodes grouped into 2 virtual LANs, and 2 machines as virtual router nodes. The virtual routers are configured to relay traffic between the two LANs, while applying the desired configuration for bandwidth and delay to the network link, emulating a high-BDP path.

Dummynet [72], a well-known open-source network emulator, is deployed on nodes $R1$ and $R2$ for implementing the router functionality. We use the latest stable version, tag *ipfw-2012*, from 16 January 2015 [1]. Our configuration mimics a high-BDP network between the two LANs. We simultaneously transfer various traffic patterns between Si/Di node pairs where different traffic streams compete for the bandwidth on the emulated link (i.e., R1 to R2).

The network configuration parameters are summarized in Figure 3.2a. For Overlay-1, all links (except the intended bottleneck) are rated at 1 Gb/s and the emulated bottleneck bandwidth between $R1$ and $R2$ is set to 500 Mb/s. For Overlay-2, network links (except the intended bottleneck) are rated at 10 Gb/s, and the emulated WAN bandwidth between $R1$ and $R2$ is 1 Gb/s. While 40 Gb/s and faster WANs are currently available as the state-of-the-art, our observations and results are mainly orthogonal to the raw bandwidth of the underlying network (e.g., Sections 4.1 and 4.2).

The hardware configuration for Overlay-1 and Overlay-2 nodes are provided in

[1] https://github.com/luigirizzo/dummynet accessed on 8 June 2021.

Parameter	**Overlay-1**	**Overlay-2**
LAN Bandwidth	1 Gb/s	10 Gb/s
WAN Bandwidth	500 Mb/s	1 Gb/s
RTT (Chapter 4)	128 ms (R1 ⇔ R2)	
RTT (Chapters 5-7)	65 ms (R1 ⇔ R2)	
Router buffer size	6 MB (R1 & R2)	
TCP CCA	CUBIC, BBR	

(**a**)

Node(s)	CPU (Model/Cores/Freq.)	RAM
$S1$, $D1$	AMD Opt. 6134 / 8 / 2.30	32 GB
$S2$, $D2$	Intel Ci3-6100U / 4 / 2.30	32 GB
$S3$, $D3$	Intel Ci3-6100U / 4 / 2.30	32 GB
$S4$, $D4$	AMD E2-1800 / 2 / 1.70	8 GB
$R1$, $R2$	AMD A8-5545M / 4 / 1.7	8 GB

(**b**)

Node(s)	CPU (Model/Cores/Freq.)	RAM
$S1$, $D1$	AMD Opt. 6134 / 8 / 2.30	32 GB
$S2$, $D2$	Intel Xeon E5620 / 16 / 2.40	32 GB
$S3$, $D3$	Intel Xeon X3323 / 4 / 2.50	4 GB
$S4$	Intel Xeon E5620 / 16 / 2.40	32 GB
$R1$, $R2$	Intel Xeon X3323 / 4 / 2.50	4 GB

(**c**)

Figure 3.2: Testbed Hardware Configuration. (**a**) Parameter space. (**b**) Overlay-1: Nodes Configuration. (**c**) Overlay-2: Nodes Configuration.

Figure 3.2b and Figure 3.2c respectively. All the nodes run Linux CentOS 6.4 Linux, using kernel version `4.12.9-1.el6.elrepo.x86_64`. Unless otherwise mentioned, CUBIC is the default TCP CCA in use by all the nodes.

For both Overlay-1 and Overlay-2 networks, the end-to-end delay (i.e., RTT) are set to either 65 milliseconds (ms) or 128 ms between the two local networks. For all the experiments in Chapter 4 the RTT is fixed to 128 ms, taken from a real network path connecting a computational center in Eastern Canada (located in Sherbrooke, Quebec) to a research center in Western Canada (located at the University of Alberta). In contrast, the RTT is fixed to 65 ms for all the experiments in the rest of the chapters (i.e., Chapters 5, 6, 7), representing a wider range of mid- to large-size networks.

The router queue buffer size at $R1$ and $R2$ are set to 6 MB for both Overlays. The 6 MB is roughly equal one BDP for the Overlay-1 network (0.8×BDP with RTT 128 ms and 1.5×BDP for RTT 65 ms), which would prevent the effects of a shallow buffer (<< BDP) while avoiding the extended delays resulting from a deep queue size (>> BDP). Due to technical issues with the unmodified Dummynet library for Linux, we were not able to adjust the buffer size beyond 6 MB to match the BDP for the Overlay-2 network. However, the buffer size does not appear to be the primary rate-limiting bottleneck in our Overlay-2 experiments, since Overlay-2 foreground throughputs are greater than 500 Mb/s (for example, in Figure 4.2, to be discussed below). For the future work we are looking to overcome this limitation.

Pragmatically, the (C3, C4) and (H1, H2) node pairs were used as routing nodes in Overlay-1 and Overlay-2, respectively, to avoid having to change Dummynet parameters between 1 Gb and 10 Gb experiments. The Dummynet and routing parameters are left unchanged for (C3, C4) and (H1, H2), and only the overlay network changes between experiments. We confirmed that the 8 GB of RAM for Overlay-1 routing nodes and 4 GB for Overlay-2 routing nodes are sufficient our experiments. Even with our most intensive workloads, we observed no memory paging on the routing

```
ip addr add 192.168.253.11/24 dev eth0
route add -net 192.168.254.0 netmask 255.255.255.0 gw cool06
```

(**a**)

```
ip addr add 192.168.253.16/24 dev eth0
ip addr add 192.168.252.16/24 dev eth0
route add -net 192.168.254.0 netmask 255.255.255.0 gw cool07
#disable ICMP redirect packets
echo 0 | tee /proc/sys/net/ipv4/conf/*/send_redirects
```

(**b**)

Figure 3.3: Emulated network configuration commands for (**a**) end-nodes. (**b**) virtual router nodes.

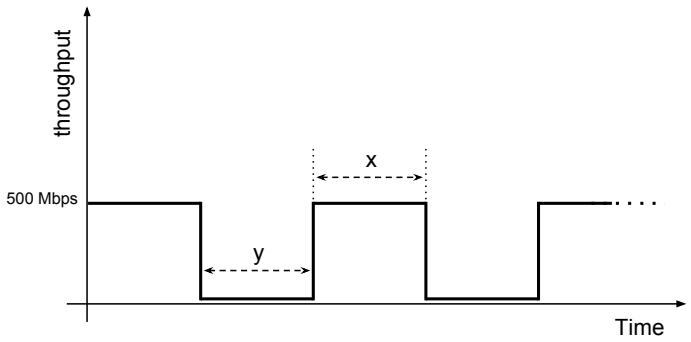
nodes (i.e., C3, C4, H1 and H2), with memory utilization at around 20%.

We transfer 14 GB worth of data, which is large enough of a transfer for the protocols to reach steady state in terms of throughput. The specific value of 14 GB of data was a practical limit, given the available main memory on our testbed nodes, and our desire to perform memory-to-memory data transfers. All data transfers are from memory-to-memory, to avoid disk I/O as a possible bottleneck. Unless otherwise mentioned, all reported results are the average of five runs, with error bars representing the standard deviation from the average.

Figure 3.3 lists the set of commands for configuration of the end-nodes (Figure 3.3a) and the virtual router nodes (Figure 3.3b).

3.2 Foreground Traffic

We experiment with a variety of sources for foreground traffic. To quantify the impact of different protocols, we use both TCP- and UDP-based traffic, generated using standard tools and applications. It includes iPerf, NFS-based data transfers, GridFTP, and UDT. We also extend this list to include popular TCP CCAs, namely CUBIC and BBR.



(**a**)

Background Traffic	**x / y (seconds)**	**Traffic Generator**
BG-TCP-Const.	10 / 0	iPerf-TCP[2]
BG-SQ-UDP1	10 / 10	iPerf-UDP
BG-SQ-UDP2	10 /10	RBUDP
BG-SQ-TCP	10 / 10	iPerf-TCP
Bursty-UDP(x,y)	x / y (user-input)	iPerf-UDP

(**b**)

Figure 3.4: Characterization of Synthetic Background Traffic. (**a**) Generated traffic pattern. (**b**) Configuration settings for generated traffic.

3.3 Background Traffic

One important requirement for studying shared networks is the ability to generate various (and, arguably, representative) background traffic patterns. One important property of background traffic is the adjustable burstiness of data streams at different time scales [61]. Here we investigate scenarios with both synthetic and application-based background traffic.

Synthetic Traffic. For this study, we developed a traffic generator script which is capable of creating traffic with adjustable burstiness at different time scales. A synthetic workload can be easier to describe, automate and reproduce than application-generated workloads. The features of our traffic generator are summarized in Figure 3.4. It can be set to create constant or parameterized square-wave traffic patterns, using either TCP or UDP transport channels.

Our study uses the following tools:

1. Netperf [73]: used as a standard benchmarking tool.
2. iPerf [74]: used as another standard benchmarking, capable of conducting multi-stream traffic.
3. GridFTP: the multi-stream TCP-based protocol.
4. UDT: the UDP-based reliable protocol.
5. RBUDP: a high-performance UDP-based data transfer tool [39]

We developed the following bandwidth-sharing scenarios to study the effect of background traffic on data-transfer tasks. The use of a square waveform is intended to be a simplified approach to non-uniform, and bursty traffic pattern. The key patterns of the background traffic, summarized in Figure 3.4b, are:

1. No background: as with data transfers on dedicated networks.
2. Uniform TCP (*BG-TCP-Const*): A constant, single-stream TCP connection between two LANs.
3. Square-waveform UDP 1 (*BG-SQ-UDP1*): A square waveform of 10 s data burst followed by 10 s of no data, using iPerf-generating UDP traffic.
4. Square-waveform UDP 2 (*BG-SQ-UDP2*): same as previous scenario, except that data bursts are generated using RBUDP transferring a 1 GB file between two LANs.
5. Square-waveform TCP (*BG-SQ-TCP*): same as previous scenario, except that data bursts are generated using iPerf-generating TCP traffic.
6. Variable length bursty UDP (*Bursty-UDP(x,y)*): the UDP traffic is generated based on a square waveform pattern while data burst and gap time lengths are parameterized as user inputs.

Application Traffic. While synthetic traffic could represent a wide range of real-world traffic patterns, we further extend our experiment scenarios to conduct real application traffic patterns. We employ NFS for generating the patterns of application traffic. There are no standard benchmarks or workloads for background traffic; NFS has the advantage of being familiar to many Unix-based environments, it is representative of classic client-server systems, and it is easy to set up without a lot of additional software or data files. Although NFS is not a typical WAN workload, many WAN data transfers will span a local-area network (LAN) segment shared with client-server applications, such as NFS.

We use *cp*, filesystem copy, to generate network traffic between an NFS client and server. This configuration enables us to study (1) the effect of NFS as background traffic on foreground traffic, and (2) the impact of other data-transfer tasks as background traffic on NFS performance as foreground traffic.

Historically, NFS used UDP as the transport protocol. However, for many years now, NFS can be used over TCP links. For our experiments, we use NFS over TCP because we had issues with repeatability of results when using NFS over UDP.

3.4 Concluding Remarks

In this chapter we have introduced our emulated WAN as the testbed for conducting benchmarking and evaluation experiments in the rest of this study. We created a dumbbell topology as an overlay network on top of a computing cluster, allowing us to emulate the bottleneck bandwidth and RTT, as present in WANs. We prepared two different configurations for our testbed, Overlay-1 and Overlay-2, using 1 Gb/s and 10 Gb/s network connections. This will enable us to study different impacts of cross-protocol competition at different rates of end-host and bottleneck bandwidth. Moreover, in order to enable us studying the impact of background traffic on different protocols, we presented several synthetic profiles of TCP- and UDP-based workloads with square-wave or configured bursty patterns.

Chapter 4

Protocol Benchmarking on Shared Networks

As previously discussed (Section 2.3), the performance of a protocol on a shared network may experience degradation depending on the presence of background traffic of different protocols and patterns. The performance degradation may manifest in terms of different performance metrics, such as throughput or fairness.

In this chapter, we evaluate a few popular TCP-based and UDP-based protocols and CCAs for their performance on a shared network. Our benchmarking reveals the notable throughput inefficiency for those tools in the presence of different patterns of background traffic introduced in Chapter 3. Furthermore, we show the unfair impact of different TCP CCAs, CUBIC and BBR in particular, when competing for the bandwidth. We will discuss throughput and fairness as the two main metrics of performance.

In particular, we look at four different scenarios, studying the inter-operation of heterogeneous workloads and the performance impact of the cross traffic (i.e., background traffic) on a shared network path:

1. **Throughput: high-performance tools.** We examine GridFTP (TCP-based) and UDT (UDP-based) data transfer tools and the impact of different patterns of background traffic on their performance. While GridFTP presents better performance against TCP-based background traffic, it tends to suffer signifi-

cant throughput degradation in the presence of UDP-based bursty traffic. In contrast, UDT manifests a higher throughput running against UDP-based background mixture.

2. **Throughput: TCP CUBIC vs. TCP BBR.** We investigate the impact of heterogeneous TCP CCA algorithms on each other. In particular, we look at the two popular CCA algorithms: CUBIC as the default CCA in most of the Linux and Windows machines, and BBR as a modern BDP-based CCA widely used recently. Our results show the unfair behavior of BBR against CUBIC when competing for bandwidth.

3. **Fairness: NFS workload for data upload.** We study the impact of bulk-data transfers and high-performance protocols on NFS client-server communication. Our results show that transferring bulk data using GridFTP and UDT could significantly degrade the NFS network users who share the same bandwidth.

4. **Fairness: high-performance tools impact on iPerf.** We study the impact of data-transfer tasks on the performance of a regular TCP-based, single-stream, user. We quantify the throughput and fairness impact on the user, further showing the unfair attributes of GridFTP and UDT.

As we will see, in the context of a shared network, the throughput performance of a protocol or tool may experience drastic degradation, up to 80% in some cases (Figure 4.1, 4.5), depending on the presence of background traffics of different protocols or varying patterns. At the same time, the use a bandwidth-aggressive tool could substantially affect the fairness in a collaborative network, a Jain fairness index of 0.65 (Table 4.2).

A summary of all the results and the corresponding testbed in use is provided in Table 4.1.

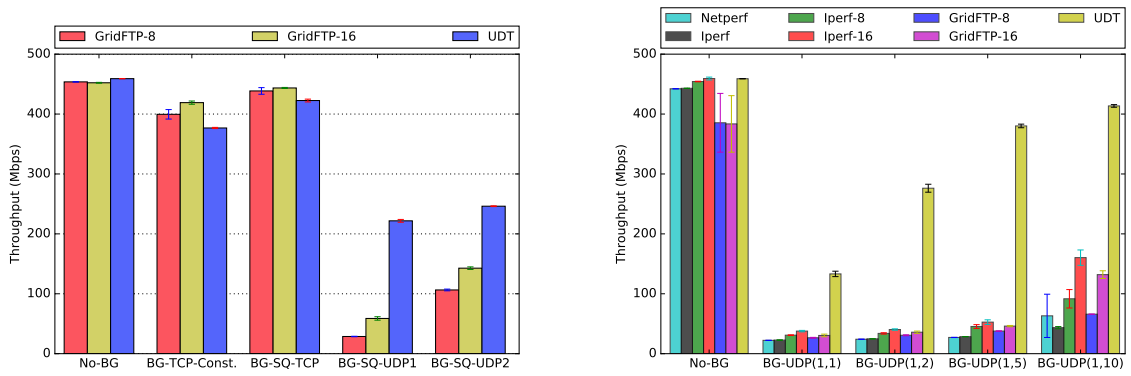
Table 4.1: A summary of experimental results provided. Overlay-1 and Overlay-2 network testbeds used from Figure 3.1.

#	Figure(s)	Network	Protocol Mixture	Background Traffic
1	Figure 4.1	Overlay-1	TCP vs. UDP	Synthetic Bursts
2	Figure 4.2	Overlay-2	TCP vs. UDP	Synthetic Bursts
3	Figure 4.3	Overlay-1	TCP vs. UDP NFS-over-TCP	Application (NFS)
4	Figure 4.5	Overlay-1	CUBIC vs. BBR	Synthetic Bursts
5	Figure 4.6	Overlay-2	CUBIC vs. BBR	Synthetic Bursts
6	Figure 4.7	Overlay-1	CUBIC vs. BBR	Constant (iPerf)
7	Figure 4.8	Overlay-1	TCP vs. UDP	Constant (iPerf, GridFTP, UDT)
8	Figure 4.9a	Overlay-1	TCP vs. UDP	Constant (iPerf, GridFTP, UDT)
9	Figure 4.9b	Overlay-2	TCP vs. UDP	Constant (iPerf, GridFTP, UDT)

4.1 Throughput: High-Performance Data Transfer Tools on Shared Networks

To investigate the behavior of two high-performance data-transfer tools, namely GridFTP and UDT, in the presence of background traffic, we experiment with different synthetic patterns of TCP- and UDP-based background traffic. The performance numbers for transferring data over a single TCP connection (iPerf) is provided as the baseline. For GridFTP, we conduct the data-transfer tasks in two configurations with 8 and 16 parallel connections, denoted by GridFTP-8 and GridFTP-16 respectively.

Scenario One. In the first scenario (Figures 4.1a and 4.2a), we transfer 14 GB worth of data between $S1$ and $D1$ (Figure 3.1), while using our synthetic traffic gener-



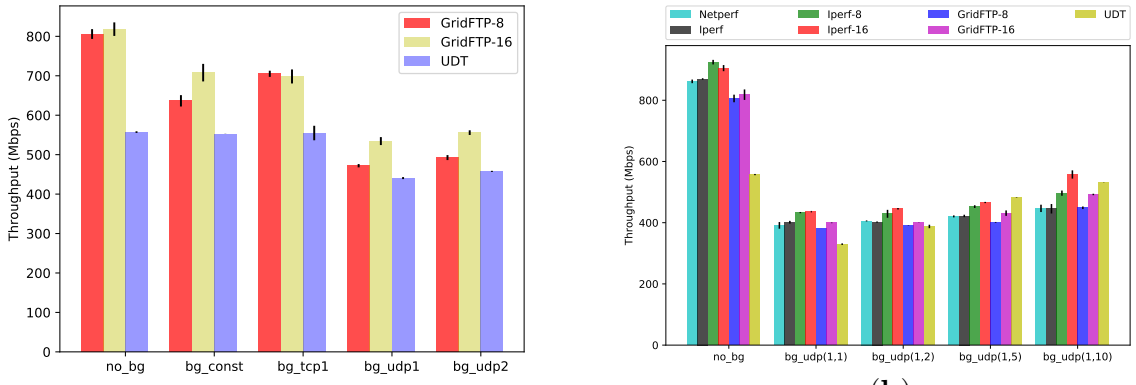
(**a**) (**b**)

Figure 4.1: (Overlay-1) Effect of TCP and UDP traffic competing for bandwidth (BW = 1 Gb/s, BtlBW = 500 Mb/s, RTT = 128 ms). (**a**) Scenario One: Square-wave pattern background traffic. (**b**) Scenario Two: Bursty UDP background traffic.

ator (Section 3.3) to run background traffic between $S2$ and $D2$, traversing the same bottleneck. The background traffic patterns include constant TCP stream, square-wave TCP and UDP streams, and square-wave RBUDP stream (cf., Figure 3.4b). In all cases, the background traffic is a single stream, periodically cycling ON and OFF in square-wave mode (Figure 3.4a).

The performance in absence of background traffic (No-BG) is also provided as a performance baseline. Both GridFTP and UDT have robust performance in the presence of TCP-based background traffic, comparable to the baseline of No-BG. However, when competing with UDP-based background traffic patterns, GridFTP and UDT start to experience significant performance degradation. UDT, as compared to GridFTP, claims a higher throughput in the presence of UDP-based background traffic. The results for the first scenario under Overlay-1 and Overlay-2 testbeds are provided in Figures 4.1a and 4.2a, respectively.

Scenario Two. UDP traffic constitutes a non-trivial share of Internet traffic in terms of telephony, media streaming, teleconferencing, and more, reinforced by the recent surge of people working remotely from home [75]. In addition, using UDP as the basis for designing new transport protocols (e.g., QUIC) is common [76]. In the second scenario (Figures 4.1b and 4.2b), to further investigate the performance



(**a**) (**b**)

Figure 4.2: (Overlay-2) Effect of TCP and UDP traffic competing for bandwidth (BW = 10 Gb/s, BtlBW = 1 Gb/s, RTT = 128 ms). (**a**) Scenario One: Square-wave pattern background traffic. (**b**) Scenario Two: Bursty UDP background traffic.

inefficiencies in the presence of bursty UDP traffic, we expose the same foreground data transfer to more dynamic UDP-based background traffic patterns.

To generate this background traffic, we use our traffic generator in parameterized mode (Section 3.3), denoted by UDP(x, y), under several configurations. The results are depicted in Figures 4.1b and 4.2b for Overlay-1 and Overlay-2 networks, respectively. We included four configurations of UDP(1,1), UDP(1,2), UDP(1,5), and UDP(1,10), where the UDP-based traffic cycles ON every 1, 2, 5, and 10 seconds respectively, transferring a 1-second burst of UDP traffic before cycling OFF. This scenario further uncovers the sensitivity of GridFTP traffic to the bursty UDP cross traffic, experiencing a drop in throughput as low as 10% of its best case. In summary, the observed performance drop is the typical behavior of multi-stream TCP-based protocols in the presence of UDP traffic bursts. UDT proves to be more robust than GridFTP in this scenario, which makes UDT a better option for shared networks with considerable UDP-based traffic over the network. In this scenario Netperf is used as a baseline, representing a single TCP stream (similar to iPerf).

Scenario Three. In the third scenario (Figure 4.3), we use application-based traffic, generated using NFS, as the background traffic. The nodes $D2$, $D3$, and $D4$

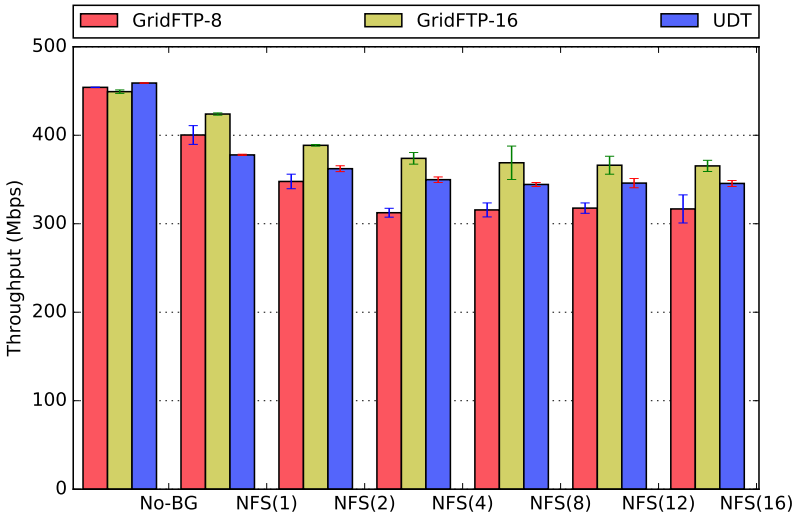


Figure 4.3: (Overlay-1, Scenario Three) Effect of NFS background traffic on throughput. NFS(x) refers to x parallel NFS clients.

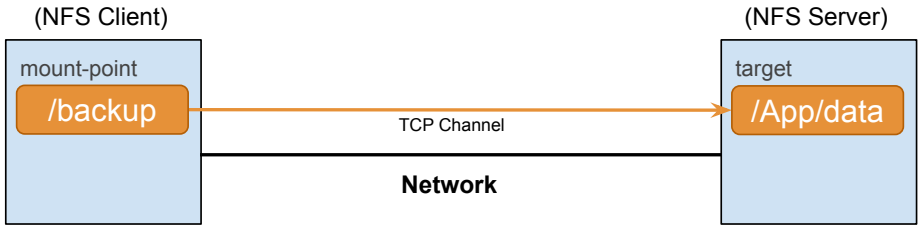


Figure 4.4: Scenario Two: NFS mount-point over network.

are the NFS servers, while $S1$, $S2$, and $S3$ are NFS clients (Figure 4.4). We deploy several configurations with a variable number of NFS clients, between 1 and 16, all on nodes $S1$, $S2$, and $S3$. To generate traffic over the network, we copy 1 GB worth of data from local memory to the NFS mounted folder. This workload could represent a file-backup workload, file transfer between remote folders, or other use-cases of NFS.

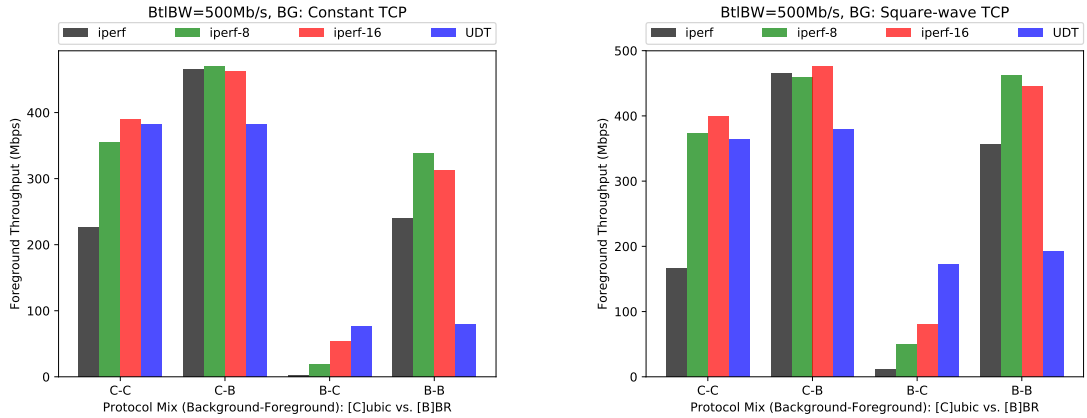
By design, GridFTP and UDT aggressively contend for network bandwidth, to the point of being unfair, and are not experiencing performance loss with NFS background traffic. The performance results of the effect of NFS as background traffic on GridFTP and UDT are presented in Figure 4.3 for Overlay-1 network. This experiment was only conducted on Overlay-1 network, but we expect the results to be qualitatively similar for Overlay-2 network. We revisit this scenario in Section 4.3 to study the impact of bulk-data transfers on NFS performance.

4.2 Throughput: TCP CUBIC vs. TCP BBR

So far, we have been utilizing CUBIC as the default TCP CCA for all the experiments. In this section, we study the impact of different TCP CCAs on each other by introducing TCP BBR into the network. BBR is a relatively new CCA designed at Google and has been gaining popularity over the past few years [21]. Instead of blindly pushing the flow rate and reactively pulling off when packet loss happens, as with reactive TCP CUBIC, the BDP-based design of TCP BBR probes, and aim at, the available bottleneck bandwidth, presenting promising end-to-end performance improvements on the network. Yet, there are several fairness-related concerns, and reported issues, with TCP BBR design when competing against TCP CUBIC (or other BBR streams) [22, 23]. Here, we try to quantify that impact in terms of throughput performance. Instead of GridFTP, we establish a similar configuration using the iPerf tool, with a variable number of parallel streams.

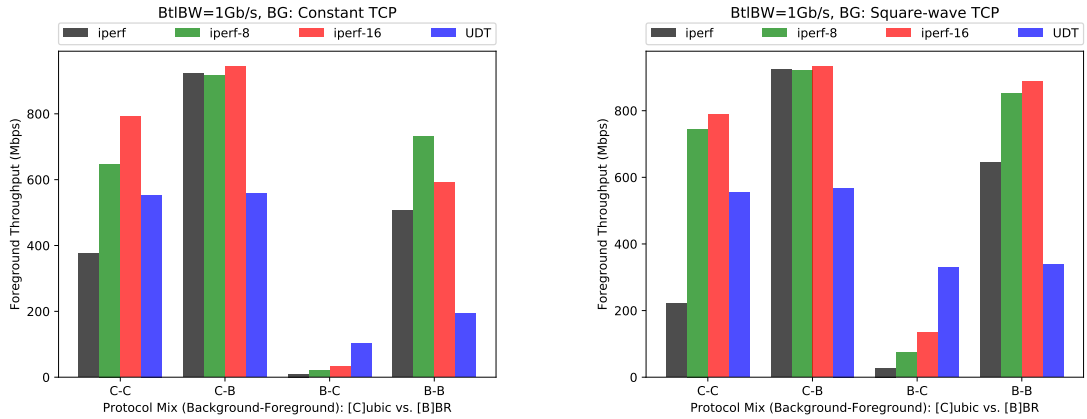
The foreground performance results for the TCP BBC and TCP CUBIC interoperability on Overlay-1 and Overlay-2 networks are provided in Figures 4.5 and 4.6, respectively. The results are provided under two scenarios for background traffic: first, constant TCP stream (Figures 4.5a and 4.6a), and second, square-wave TCP stream (Figures 4.5b and 4.6b). Furthermore, Figure 4.7 provides the throughput results for both background and foreground streams for the mixtures of up to three streams of constant TCP CUBIC versus TCP BBR competing for bandwidth. Note that iPerf is used for conducting constant traffic, where foreground traffic (FG) is started with some delay compared to the background traffic (BG).

The results follow a similar pattern across both testbeds. For both constant and square-wave background patterns, TCP BBR manifests an adverse impact on TCP CUBIC, used either for the foreground or background streams.



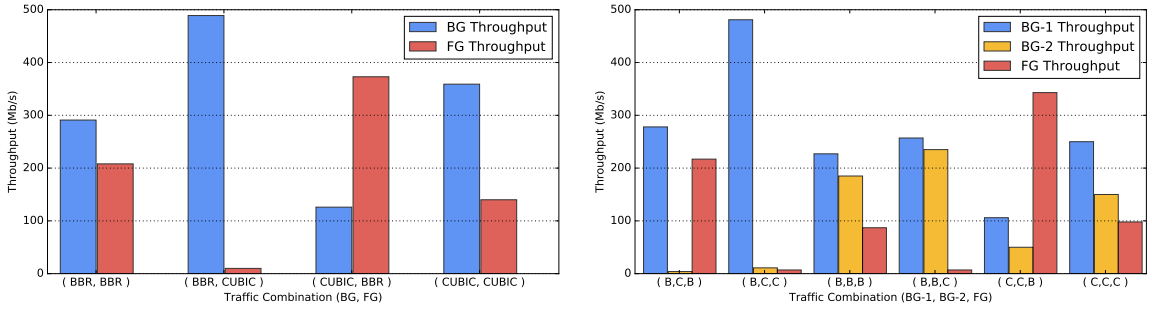
(**a**) (**b**)

Figure 4.5: (Overlay-1) TCP CUBIC versus TCP BBR competing for bandwidth. (**a**) Background traffic: Constant TCP. (**b**) Background traffic : Square-wave TCP.



(**a**) (**b**)

Figure 4.6: (Overlay-2) TCP CUBIC versus TCP BBR competing for bandwidth. (**a**) Background traffic: Constant TCP. (**b**) Background traffic: Square-wave TCP.



(**a**) (**b**)

Figure 4.7: (Overlay-1) Constant TCP CUBIC versus TCP BBR using iPerf (FG has less live time). (**a**) Two competing streams (**b**) Three competing streams.

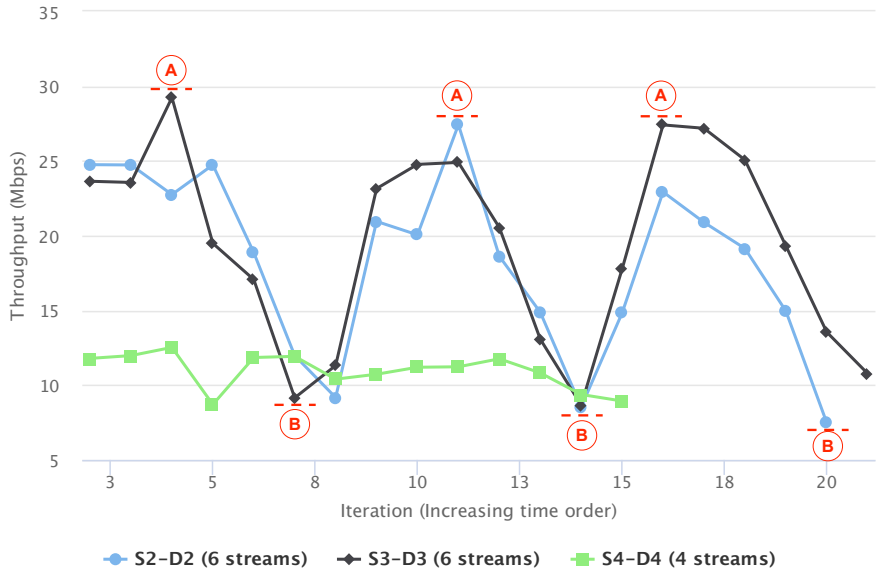
4.3 Fairness: NFS Performance on Shared Network

In this section, we investigate the effect of bulk-data transfers on the performance of NFS client-server communication. Our approach is identical to the third test scenario in Section 4.1, but instead we consider the NFS as foreground traffic. We only present the configuration with 16 concurrent NFS connections and streams. We also experimented with fewer than 16 streams but the performance characteristics are similar to the case with 16 streams. The choice of 16 NFS streams matches GridFTP-16's use of 16 TCP streams. The choice of three node-pairs is the maximum possible on our testbed.

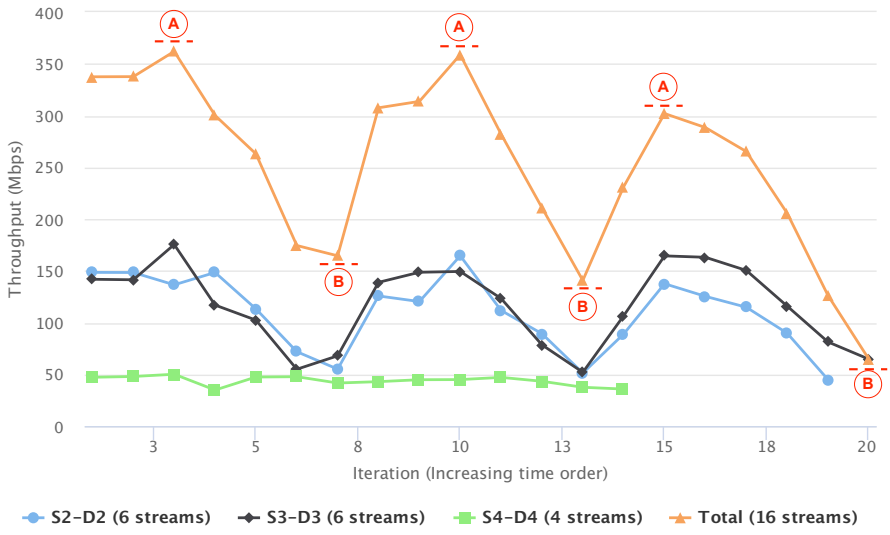
In our test scenario, a total of 16 NFS connections are established between three pairs of nodes ($S2$, $D2$), ($S3$, $D3$), ($S4$, $D4$), repeatedly transferring 1 GB files over the network (copying file to NFS mount points). Each NFS foreground data transfer is one iteration. The baseline, copying a local 1 GB file to a mounted NFS folder, takes about 20 s (at 410 Mb/s) in our Overlay-1 network, with no background traffic. We will study how this performance is affected in the bandwidth-sharing scenario.

For background traffic, we conduct a series of bulk-data transfers between $S1$ and $D1$ over the same high-BDP path, consecutively transferring 14 GB data using iPerf-TCP (1 stream), GridFTP-8 (8 streams), GridFTP-16 (16 streams), and UDT (not based on streams). Therefore, each background run transfers 56 GB in total (i.e., 4×14 GB) and the progression from iPerf-TCP to UDT represents a growing aggressiveness of the background traffic. As in previous test scenarios, we run the same set of tests three times. Therefore, the foreground NFS traffic is a series of iterations of 1 GB data transfers. The background traffic is three runs of 56 GB data transfers.

The results are presented in Figure 4.8. For better readability, the results are aggregated per NFS node-pair of foreground traffic.



(**a**)



(**b**)

Figure 4.8: Impact on NFS: Throughput of NFS-over-TCP with TCP and UDP background traffic (Overlay-1 Network). (**a**) Average NFS Throughput, Per Node-Pair. (**b**) Aggregate NFS Throughput, Per Node-Pair.

Figure 4.8a reports the average throughput of NFS foreground data transfers between each node-pair. The x-axis corresponds to time, specifically consecutive iterations of NFS data transfer in the foreground. The y-axis is throughput (in Mb/s). The individual three lines represent the throughput for a given node-pair of NFS server and client (i.e., three node-pairs: S2-D2, S3-D3, S4-D4). Note that the number of streams varies between node-pairs.

The average throughput of each NFS stream (i.e., average of either 4 or 6 streams) is approximately 29 Mb/s in the best case (marked as circle-'A' in Figure 4.8a), dropping to less than 10 Mb/s in the worst case (marked as circle-'B' in Figure 4.8a).

The three peaks (marked as circle-'A') correspond to the start of each three background runs, starting with iPerf (only a single TCP CUBIC stream). As shown in earlier experiments, TCP CUBIC is fair in bandwidth sharing. Recall that we are using NFS over TCP CUBIC, so this is a scenario of sharing between TCP CUBIC streams. The 3 troughs (marked as circle-'B') correspond to the extreme case of transferring data using GridFTP-16 and UDT. As discussed before, GridFTP-16 is aggressive in capturing bandwidth, and is unfair in sharing the network with the foreground NFS traffic. The total accumulated throughput of the NFS streams are reported in Figure 4.8b. The orange line (the highest curve) represents the overall bandwidth utilization of 16 NFS connections. One caveat on utilizing NFS in our experiments is that the NFS is not designed to be used over high-BDP networks; so it might not be optimized for such a testbed. However, its communication pattern could be representative of various TCP-based application traffic over the network, with the same impacts from the bulk-data transfer tasks.

To summarize this test scenario, we observe that the GridFTP and UDT could impose considerable performance degradation on standard network traffic, which could hurt the notion of fairness over the shared networks. We further discuss the fairness issues in Section 4.4.

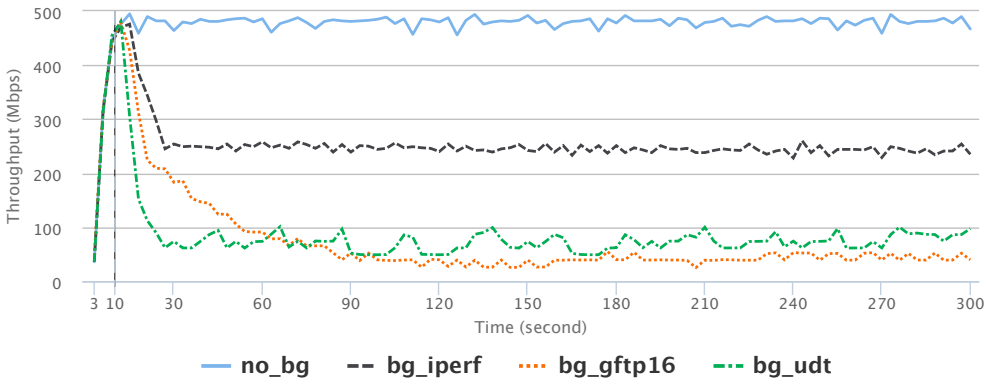
Table 4.2: Throughput (Mb/s) and Jain fairness (0∼1) of well-known protocols on Overlay-1 Network. As evident by calculated Jain indices, GridFTP and UDT are considerably unfair to single TCP streams, GridFTP representing the lowest fairness index.

Type	**Stream**	**no_bg**	**bg_iperf**	**bg_gftp16**	**bg_udt**
Dedicated	Background	-	478	473	481
	Foreground (iPerf)	474	474	474	474
Shared	Background	-	238	415	398
	Foreground (iPerf)	474	253	75.6	85.3
	Jain fairness	**-**	**1**	**0.68**	**0.7**

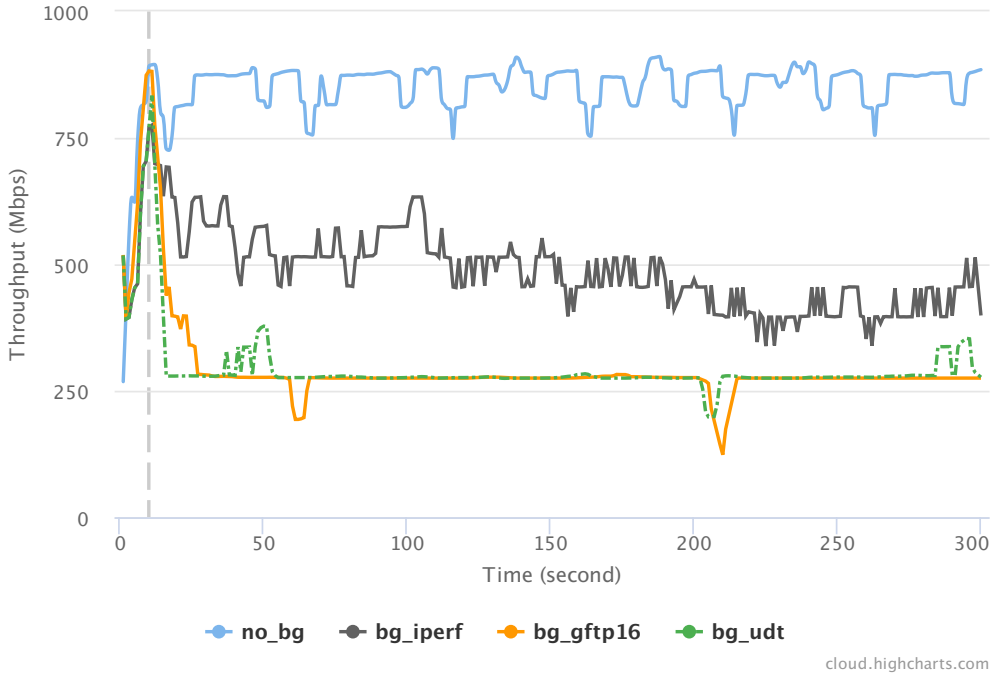
4.4 Fairness: Impact of High-Performance Tools on iPerf TCP

As discussed in Section 2.4, fairness is a measure of how consistently a network stream shares the bandwidth with other streams, either of the same or heterogeneous protocol. One form of ideal fairness is that all traffic streams consume the same amount of shared bandwidth.

To verify the effect of GridFTP and UDT on background traffic in shared networks, we conducted a series of test for benchmarking the effect of these tools (as background traffic) on a single, classic TCP stream (as foreground traffic). CUBIC CCA is used for all TCP-based streams. The results for Overlay-1 and Overlay-2 networks are presented in Figure 4.9a,b, respectively. The foreground single TCP connection is established using the iPerf tool. The performance of this connection is measured in four different scenarios in terms of background traffic type: no background (no_bg), single TCP (bg_iperf), GridFTP with 16 concurrent TCP streams (bg_gftp16), and UDT (bg_udt). For each scenario, the foreground connection was running for 300 seconds while the background traffic was a file transfer of 14 GB (which takes roughly the same time when run by itself).



(**a**)



(**b**)

Figure 4.9: Throughput of a standard TCP CUBIC stream in the presence of various background traffic for (**a**) Overlay-1 Network. (**b**) Overlay-2 Network.

Table 4.3: Throughput (Mb/s) and Jain fairness (0∼1) of well-known protocols on Overlay-2 Network. As evident by calculated Jain indices, the fairness issues with GridFTP and UDT are consistently manifested regardless of the varying BDP (bottleneck-bandwidth and RTT).

Type	**Stream**	**no_bg**	**bg_iperf**	**bg_gftp16**	**bg_udt**
Dedicated	Background	-	848	811	625
	Foreground (iPerf)	872	872	872	872
Shared	Background	-	412	738	591
	Foreground (iPerf)	872	485	109	161
	Jain fairness	**-**	**0.99**	**0.64**	**0.75**

The quantified throughputs along with the calculated Jain fairness index are represented in Tables 4.2 and 4.3. It is worth noting that the background connections were established with a 10 s delay from the foreground traffic, which is represented with a vertical dashed line in Figure 4.9.

The extreme effect of the GridFTP and UDT protocols on the foreground protocol is significant. For example, in Figure 4.9, notice how the green and orange lines drop well below under 100 Mb/s after 60 seconds, because the background traffic is aggressively occupying the whole available bandwidth. Both GridFTP and UDT protocols tend to occupy more that 80% of available bandwidth, allowing less than 20% of bandwidth to be used by the other concurrent traffic stream.

While this behavior is consistent with the design principles of those tools, from a traffic-management point of view this is not a fair resource allocation between two or more concurrent connections over the network. Not only does it affect the other traffic using the shared WAN, it also may result in poor performance of other network applications running on the LANs. One immediate remedy to this last issue is to make a separation between the traffic going over a WAN and the LAN, as discussed in [77].

4.5 Concluding Remarks

In this chapter we showed that background traffic can reduce the throughput performance and fairness for foreground traffic. We studies the impact of a variety of synthetic and application (i.e., NFS) background traffic patterns on the performance of data transfer tools (iPerf, GridFTP, UDT) and TCP CCAs (CUBIC vs. BBR).

We showed that, in the context of a shared network, the throughput of a data transfer could experience drastic degradation, up to 80% in some cases, in the presence of bursty background traffic (e.g., GridFTP vs. bursty UDP traffic, Figure 4.1). Furthermore, in case of all-TCP traffic mixture, the utilization of heterogeneous CCAs, CUBIC versus BBR, may yield undesired performance impacts such as unfair bandwidth sharing (e.g., TCP CUBIC vs. TCP BBR, Figure 4.5). Finally, the presence of throughput-oriented tools such as GridFTP and UDT may impose high impacts on the performance of NFS-based application data transfers (e.g., Figure 4.8).

In conclusion, when communicating data on a shared network same high-performance tool or CCA may not always result in the best performance. Whether on a collaborative network (e.g., a university campus) or a competitive network (e.g., multiple organizations sharing bandwidth) the performance priorities might change, for example pure throughput versus throughput-and-fairness (i.e., both throughput and fairness matter). But for all cases, based on different type and patterns of the background traffic, one should consider using different tools and protocols for optimizing on their desired performance metrics.

One major challenge in deciding on a data transfer tool or protocol is that the type of background traffic is mostly unknown to the end hosts. In the following chapter we will examine the possibility of recognizing the type of background traffic on a shared network, followed by an effort to adopt such an insight towards adaptive protocol selection.

Chapter 5

Machine-Learned Classification of Network Protocols

As discussed in the previous chapter (e.g., Table 4.1), in the context of a shared network, the throughput and fairness of protocols could experience drastic degradation, depending on the mixture and patterns of the background traffic. Hence, obtaining knowledge about the type and mixture of the background traffic could enable us to utilize more appropriate protocols, better suiting the network conditions and yielding better performance. In other words, if we can recognize the mixture of background traffic, instead of using a static foreground protocol, we could adaptively choose the foreground protocol in a way to improves throughput and fairness.

In this chapter, we investigate building method and tools for recognizing the type and mixture of the background traffic on a network, from an endpoint perspective. Such knowledge would then enable us to make informed decision on the use of appropriate protocols and tools for the foreground. For example, knowing a bursty UDP-based background traffic is present, one could decide to, instead of GridFTP, use UDT for bulk-data transfer (Section 4.1). Or in the case of different TCP CCAs, in the context of a collaborative shared bandwidth, the knowledge about a TCP CUBIC background traffic would imply using TCP CUBIC, acquiring optimum[1] bandwidth

[1]In this context, optimum bandwidth is not defined as maximizing the throughput (pure throughput). It is defined as obtaining a fair share of bandwidth, equal to the other traffic streams on the network (throughput-and-fairness).

while keeping fair with the other users on the network (Section 4.2). In contrast, in the presence of TCP BBR background traffic, one could decide to use TCP BBR for a new data stream, keeping up to the speed with the other traffic.

However, global knowledge can be difficult to obtain in a dynamic distributed system like a shared network. Probing the available network resources and workload is a key insight for the efficient network utilization. Pragmatically, such probing should be performed from a local perspective, viewing the network as an end-to-end black box.

Can ML classify the background traffic in bandwidth-sharing networks? This question serves as our main motivation to design a ML pipeline for recognizing the mixture of traffic workload on a shared network. The characterization of network background traffic is a novel approach in our study, discussed in further detail in upcoming sections. We start by defining target classes for TCP-based traffic mixtures (Section 5.1). Note that UDP-based protocols are not studied from this point on. Then we introduce our passive/active probing techniques for capturing representative network signals to be used for classification (Section 5.2). Our ML pipeline starts by probing the background traffic, gathering RTT training data (Section 5.3) followed by data pre-processing and preparation (Section 5.4). Then, we present and discuss our classification models trained for identifying the mixture the background traffic (Section 5.5). We wrap up this chapter by presenting the classification performance for the trained models, gaining up to 95% accuracy (Section 5.6).

5.1 Scope: TCP CUBIC vs. TCP BBR

While a diverse collection of reliable and unreliable tools and protocols are in use on the Internet, TCP-based protocols are the predominant type of traffic on data networks. For the purpose of building our classification models, we only consider TCP-based background traffic, under the simplified assumption that TCP CUBIC and TCP BBR are the only protocols available.

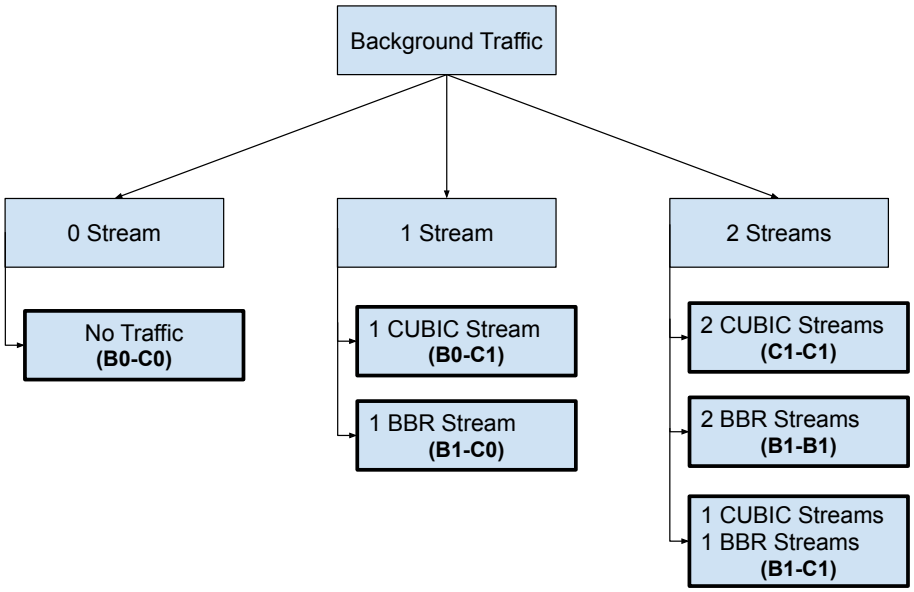


Figure 5.1: Six classes of background traffic to be recognized. Mixtures of up to two streams of TCP CUBIC, TCP BBR, or both (bold boxes).

In particular, we will consider six distinct classes of background traffic to train a classifier for them, depicted in Figure 5.1 (bold boxes). These six classes include various mixtures of up to two streams of TCP CUBIC and TCP BBR, where either or both are present on the network.

Generalizing the classifier models for larger mixtures of streams and different CCAs and protocols (e.g., QUIC) is a direction for the future extension of this study. Furthermore, evident by the emergence history of new TCP CCAs, new CCAs are expected to be proposed in the future. Therefore, the mixture of CCAs on the actual networks would change over time. New classification models are needed to be trained as new CCAs get emerged or widespread over time.

5.2 Recognizing Background Workload: Passive vs. Active Probing

Recognizing the available network resources and workload could provide key insights towards efficient management and utilization of the network. Probing network workload is ideally performed without requiring global information, viewing the network

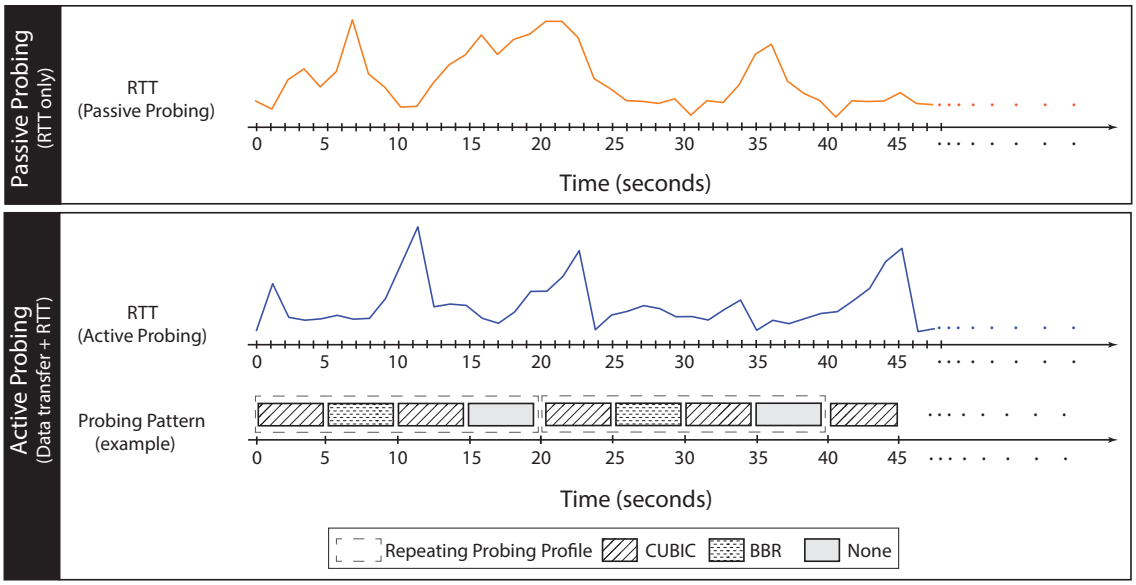


Figure 5.2: Passive and Active Probing: RTT measurement (ping) combined with probing pattern based on bandwidth interference (iPerf).

as an end-to-end black box. Such an insight could potentially be utilized for different real-world use-cases, including workload discovery, protocol recognition, performance profiling and more. In our particular use-case, such knowledge would allow better, more-efficient data transfer through choosing the appropriate protocol for data-transfer tasks (cf., Chapter 6).

We introduce two probing techniques, *passive* (non-intrusive) and *active* (intrusive), trying to generate representative end-to-end signals for the network workload (Figure 5.2). As we will see active probing outperforms passive probing. Both probing techniques, as discussed below, generate RTT time-series as the representative signal for the network workload. Samples of generated RTT time-series using both techniques are presented in Figure 5.3. We later use those signals in a ML task to build and train classification models for recognizing the background workload.

Passive Probing. Network protocols may follow different, sometimes contradictory, design principles. As such, we hypothesize that different mixtures of protocols would imply distinct workload patterns on the network. To quantify those workload patterns, we adopt RTT as the representative, end-to-end signal for the network

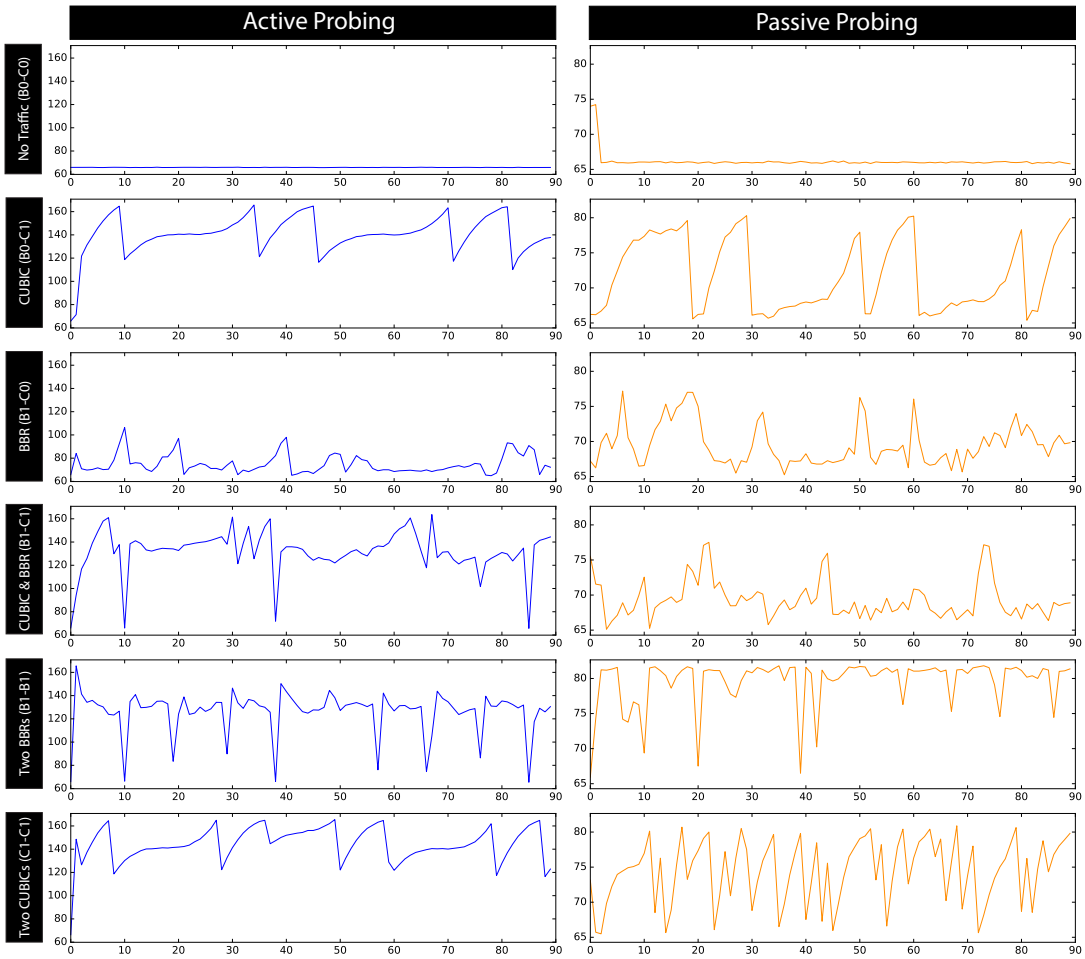


Figure 5.3: Illustrative examples: active vs. passive probing impact on RTT signal: 90 s time-series signatures per traffic mixture. Active probing changes time-series patterns. Note the different ranges for Y-axis across columns.

workload over time. We periodically measure local, end-to-end RTT in regular time intervals, forming time-series data (Figure 5.2, top section). Passive probing is a non-intrusive mechanism, solely relying on a minimal RTT measurement through end-to-end pinging. It does not involve interfering the network traffic in any noticeable way. As we will see, such a probing strategy will result in distinct time-series signatures for different background protocol mixtures.

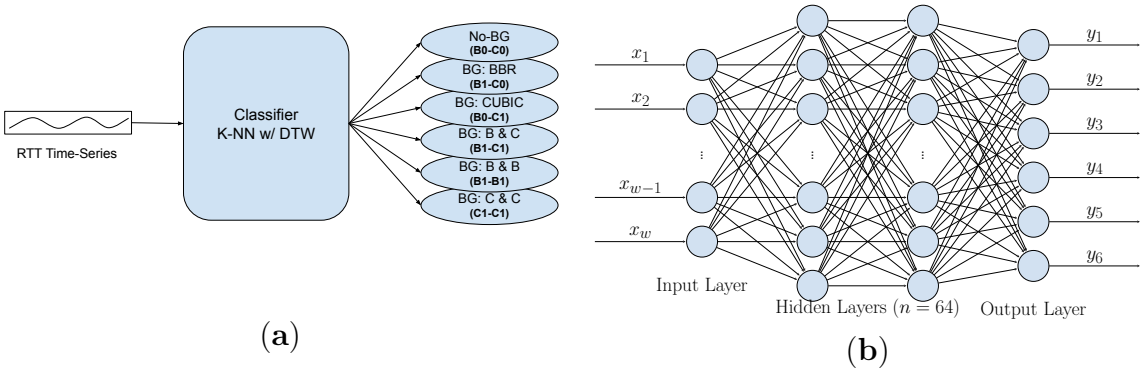
Active Probing. As an extension of passive probing, we add a systematic and deliberate perturbation of traffic on a network (i.e., intrusive mechanism) for the purpose of gathering more discriminative information (Figure 5.2, bottom section).

We conduct active probing in the form of short bursts of traffic, while concurrently RTT is being probed. To trigger background streams to reveal their unique patterns, as opposed to a single traffic burst, we use a sequence of short traffic bursts with different configurations (e.g., different CCAs). We refer to such a sequence of short bursts as an *active probing pattern*, illustrated as dashed boxes in Figure 5.2. The time-series data generated by active probing improves the ability to distinguish the time-series for different workloads, evident by our ML evaluation (Section 5.6). As we will discuss in Chapter 6, the data bursts of active probing can be read from the data file(s) staged to be transferred, hence making it constructive perturbation of bandwidth. We refer to such setting as *constructive active probing* (cf., Section 6.3).

5.3 Gathering RTT Training Data

For a classification problem, we need to gather sufficient data for training our model. We generate RTT time-series using both passive and active probing schemes (Section 5.2). Our target classes to be decided are different mixtures of CCAs in background workload per Figure 5.1. We designed a controlled experiment to gather RTT data as a time-series, with a 1-second sampling rate, over a one hour period of time. To smooth out the possible noise in the measured RTT value, each probing episode consists of sending ten ping requests to the other end-host, with 1 ms delay in between, and the average value is recorded as the RTT value for that second. We repeated this experiment for the six traffic combinations in Figure 5.1.

For gathering data, we use the Overlay-1 testbed (Figure 3.1b) to conduct traffic between the two LANs, sharing the same bottleneck bandwidth. We conduct the background traffic between the $S1$ and $D1$ pair of nodes. In the cases where we need a second stream of background traffic, the $S2$ and $D2$ pair of nodes are used. Finally, the $S3$ and $D3$ nodes are used to probe the end-to-end RTT. For all data-transfer tasks, we used the iPerf tool for generating TCP traffic of the desired CCA. Samples of the generated time-series data are presented in Figure 5.3.



(**a**) (**b**)

Figure 5.4: Classification Models for Background Traffic Classification. (**a**) Inference Workflow. (**b**) Network Structure for MLP neural network.

In the case of active probing, we defined the active probing pattern as a sequence of three segments of traffic bursts (CUBIC for 5 s, BBR for 5 s, CUBIC for 5), followed by a silence period of 5 s. We use the abbreviation *C5-B5-C5-S5* for referring to this pattern. Each traffic burst will send data to the network at full speed as regulated by the CCA in use.

5.4 Data Preparation and Pre-processing

To prepare the gathered data (cf., Section 5.3) for ML, we partition the one-hour RTT time-series into smaller chunks. We developed a script to partition the data based on a given parameter w, representing a window size measured in data points (which also corresponds to a window in seconds, given the one second gap between RTT probes). We experimented with both non-overlapping and overlapping (or sliding) windows. Due to sub-optimal performance of our trained models with overlapping (sliding) windows, we only present and discuss results using non-overlapping windows.

5.5 RTT Time-Series Classification

In this section we train a few classifiers for recognizing the mixture of the background traffic (Figure 5.4a). Classification is a common technique in ML [78]. Here we investigate several supervised classification models, namely k-nearest-neighbors (K-NN),

a multi-layer perceptron (MLP) neural network, Support Vector Machines (SVM), and naive Bayes (NB) [78]. For this study, we train classification models for RTT-based representations of the workload, generated using passive and active probing schemes (Figure 5.3). Even though time-series RTT data violates the assumption of independent features in the feature vector, we still evaluate SVM and NB.

K-NN. For K-NN classification, the k closest training samples to the input time-series are selected and then the class label for the input is predicted using a majority vote of its K-nearest-neighbors' labels. k is the input parameter to this model. As we will discuss shortly, we use K-NN along with DTW as the distance measure. DTW has a warping parameter for which we tested different values, settling on 10.

MLP. A multi-layer perceptron (MLP) is a type of feedforward artificial neural network, consisting of one or more fully-connected hidden layers. The activation output of the nodes in a hidden layer are all fed forward as the input to the next layer. While more sophisticated models, such as LSTM and ResNet, exist for time-series classification [79], for the purpose of building a proof-of-concept model we limit our study here to a simple MLP (Figure 5.4b). Our MLP network consists of two fully-connected hidden layers of size 64. The input layer is of size w (the length of RTT time-series vector) and the output layer consists of six nodes, generating a on-hot-encoded prediction (*one* for the predicted class and *zero* for the rest). The nodes of the hidden layers employ Rectified Linear Unit (ReLU) activation, while the output layer uses a Softmax activation.

SVM. In Support Vector Machine (SVM), a hyperplane or a set of hyperplanes, are constructed in high-dimensional space. The hyperplanes are then used to classify the unseen data-points to one of the predefined classes. The hyperplanes are constructed in a way to maximize the marginal distance with the training samples of each class. Our implementation of SVM follows the default configuration from the *scikit-learn* library, using Radial Basis Function (RBF) as the kernel.

NB (GNB). A Naive Bayes (NB) classifier is a probabilistic model, assigning a

probability for each class conditional to the feature values of the input vector. Notably, the NB model assumes the features to be independent, while for time-series data the data-point values could be dependent on the previous ones. Yet, we include NB as another point of comparison. We use the *Gaussian Naive Bayes* (GNB) implementation from scikit-learn library.

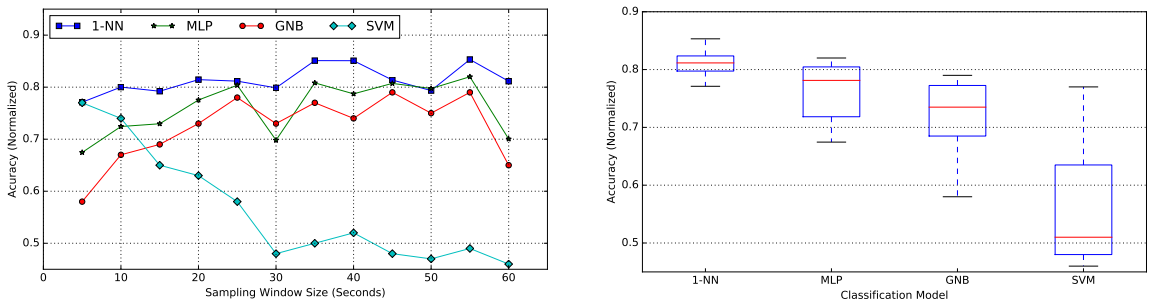
Distance Measure. Time-series classification is a special case where the input data being classified are time-series vectors. While Euclidean distance is widely used as the standard distance metric in many ML applications, it is not well-suited for time-series data, since there is noise and distortion over the time axis [80]. More sophisticated approaches are needed for comparing time-series data. Two common approaches for handling time-series data are, (1) quantizing a time-series into discrete feature vectors [81], and, (2) using new distance metrics [82].

For the K-NN classification model, we utilize the well-known *dynamic time warping* (DTW) distance metric for the time-series data. DTW was originally used for speech recognition. It was then proposed to be used for pattern finding in time-series data, where Euclidean distance results in poor accuracy due to possible time-shifts in the time-series [83, 84].

To estimate accuracy and overfitting, we use 5-fold cross-validation, where 20% of the data from each traffic class are in a hold-out set, not used for training, and then tested with classifier. The reported results are the average accuracy over the five folds on the cross-validation scheme. The classification performance results for all models are provided and discussed in Section 5.6.

5.6 Evaluation: Classification Performance

In this section, we provide and discuss the performance evaluation results of the developed classifier for characterizing the mixture of background traffic (out of the six classes of background traffic in Figure 5.1). We first discuss the classification performance when passive probing is used. We then introduce and compare the



(**a**) (**b**)

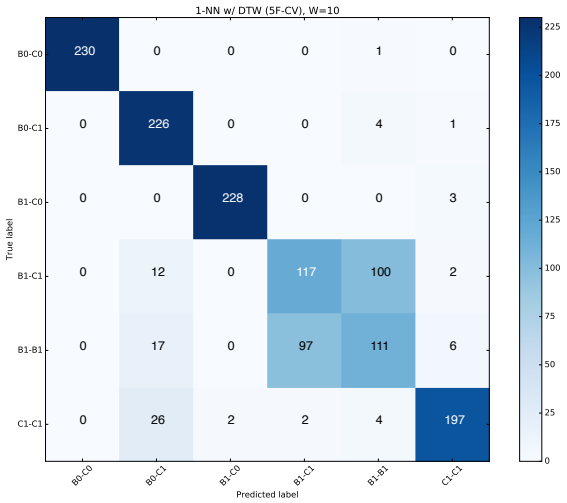
Figure 5.5: (Passive Probing) Performance of classification models for varying time length w of the input time-series (5-Fold Cross-Validation). (**a**) Classification accuracy. (**b**) Box Plot for classifiers accuracy.

results for classifiers trained using the time-series of active probing.

Selecting the appropriate time-length parameter w is a trade-off between the amount of information provided to the classifier and the RTT probing latency. On the one hand, the larger the w becomes, the more information will be provided to the classifier, providing a better opportunity to recognize the traffic pattern. On the other hand, a larger w means it would take more time for the probing phase of a real system to gather RTT data to use as input to a classifier. To make an appropriate decision about the parameter value w, we do a parameter-sweep experiment where we calculate the classification accuracy for all the classifiers, varying parameter w from 5 s to 60 s, with a 5 s step.

Passive probing. Figure 5.5a represents the average accuracy per window size w (in seconds), calculated for 1-NN, MLP, GNB, and SVM classifiers. We include the box plot for each classification model in Figure 5.5b. In a box plot, the dashed lines mark the minimum and maximum value, where the solid box range represents the 25th and 75th percentiles respectively. The horizontal line crossing the solid box marks the median value.

For the K-NN model, we examined three different configurations with k equal to 1, 5, or 10, where the 1-NN represents a better overall accuracy. Hence, we use K-NN with $k = 1$, where the single closest neighbor to the input would be used for predicting



(a)

	Precision	Recall	F1-Score	Support
B0-C0	1.00	1.00	1.00	231
B0-C1	0.80	0.98	0.88	231
B1-C0	0.99	0.99	0.99	231
B1-C1	0.54	0.51	0.52	231
B1-B1	0.50	0.48	0.49	231
C1-C1	0.94	0.85	0.90	231
Avg / Total	0.80	0.80	0.80	1386

(b)

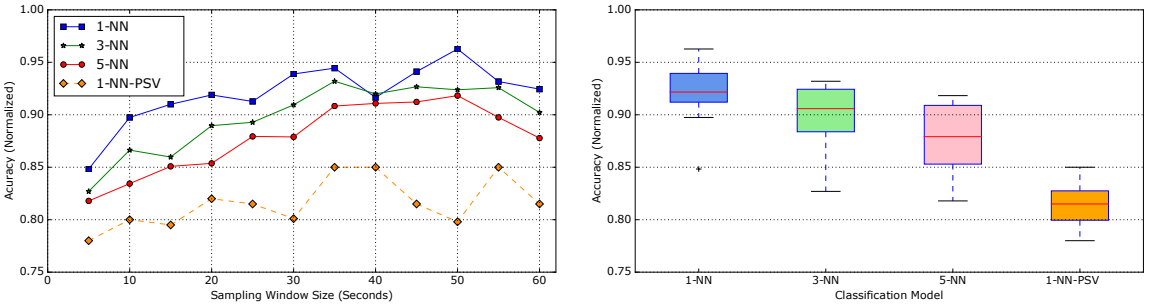
Figure 5.6: (Passive Probing) K-NN (DTW) Performance ($w = 10$, $k = 1$) (5-Fold Cross-Validation). (**a**) Confusion Matrix. (**b**) Classification Report on Accuracy-related metrics.

the class label. So for the comparison in Figure 5.5 only 1-NN variation is provided.

Comparing performance of different classifiers, the 1-NN classifier outperforms the other models in most cases. According to the box plot, 1-NN presents the most consistency in accuracy for varying window sizes, with a narrow variation in observed performance. SVM tends to have the best performance after 1-NN for smaller window sizes. However, its accuracy drops significantly as we increase the time-series length. MLP and GNB show a similar accuracy pattern, following 1-NN with a slight margin.

For reference, the detailed *confusion matrix* and classification report for a single configuration (1-NN, $w = 10$) are provided in Figure 5.6.

Active probing. We now introduce the performance results for classifiers trained using active probing. As discussed before, K-NN outperforms the rest of the classifiers for all windows sizes. So for the rest of discussions we only provide results for K-NN



(**a**) (**b**)

Figure 5.7: (Active Probing) Accuracy of classification models for window size w of the input time-series with Active Probing (5-Fold Cross-Validation). 1-NN-PSV represents Passive Probing performance as the baseline (Figure 5.5). (**a**) Classification Accuracy per window size. (**b**) Box Plot for Classifiers Accuracy.

classifiers of different k parameter (1, 3, 5). Across all the figures, best viewed in color, corresponding colors are used to highlight the corresponding variation of the K-NN classifier model.

The accuracy results for different classifiers per time length w of the input time-series are provided in Figure 5.7. Along with the results for classifiers trained using active probing, the results for the *1-NN* classifier with passive probing (labelled as 1-NN-PSV) are also provided for reference and comparison. All active probing classifiers outperform passive-probing classifiers by between 7% (for w = 5) to 16% (for w = 50) (depending on window size) (Figure 5.7a). This performance improvement verifies our hypothesis that systematic perturbation using active probing will reinforce unique patterns in RTT time-series for different background traffic mixtures.

Across different configurations for active probing classifiers, the 1-NN tends to offer a better overall accuracy (Figure 5.7a). According to the box plot, 1-NN presents the most consistency in accuracy for varying window sizes, with a narrow variation in observed performance (Figure 5.7b).

To further analyze the performance consistency and variation, the accuracy results along with accuracy variation across five folds are provided in Figure 5.8 for all four classifiers (1-NN, 3-NN, 5-NN with active probing, and 1-NN with passive probing).

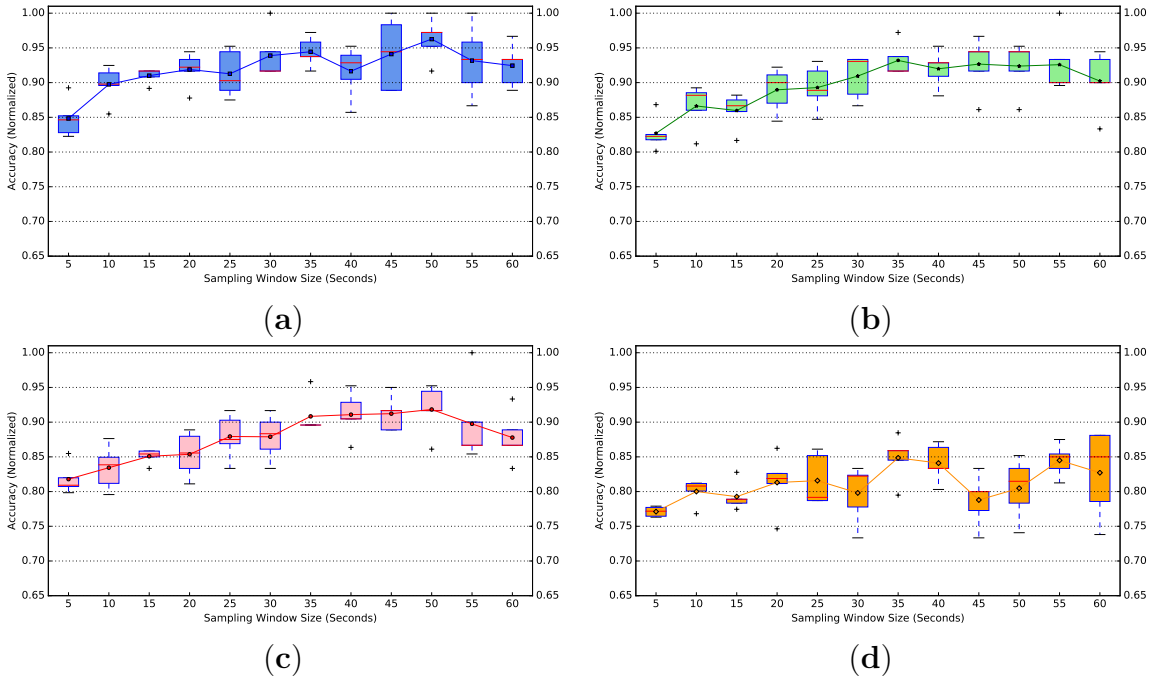


Figure 5.8: Accuracy variation (BoxPlot) of classification models for varying time length w of the input time-series per k in K-NN with DTW—with and without Active Probing (5-Fold Cross-Validation). (**a**) 1-NN w/ DTW with Active Probing. (**b**) 3-NN w/ DTW with Active Probing. (**c**) 5-NN w/ DTW with Active Probing. (**d**) 1-NN w/ DTW with Passive Probing.

For better interpretability, instead of standard error bars, the box plot is provided across five folds for each window size. In most cases, 1-NN (Figure 5.8a) outperforms 3-NN (Figure 5.8b) and 5-NN (Figure 5.8c) with active probing.

To better realize the classifiers' performance and their challenges in predicting traffic classes, the confusion matrices for $w = 10$ are provided in Figure 5.9, corresponding to the four classifiers in previous figures. As expected, the classifiers with active probing consistently manifest more diagonal confusion matrices, representing more accurate predictions across 6 classes. 1-NN with active probing (Figure 5.9a) and 1-NN with passive probing (Figure 5.9d) correspondingly represent the highest and lowest performance across all classifiers.

To take a close look at how active probing helps improve the quality of prediction, the isolated version of confusion matrices for 1-NN with and without active probing

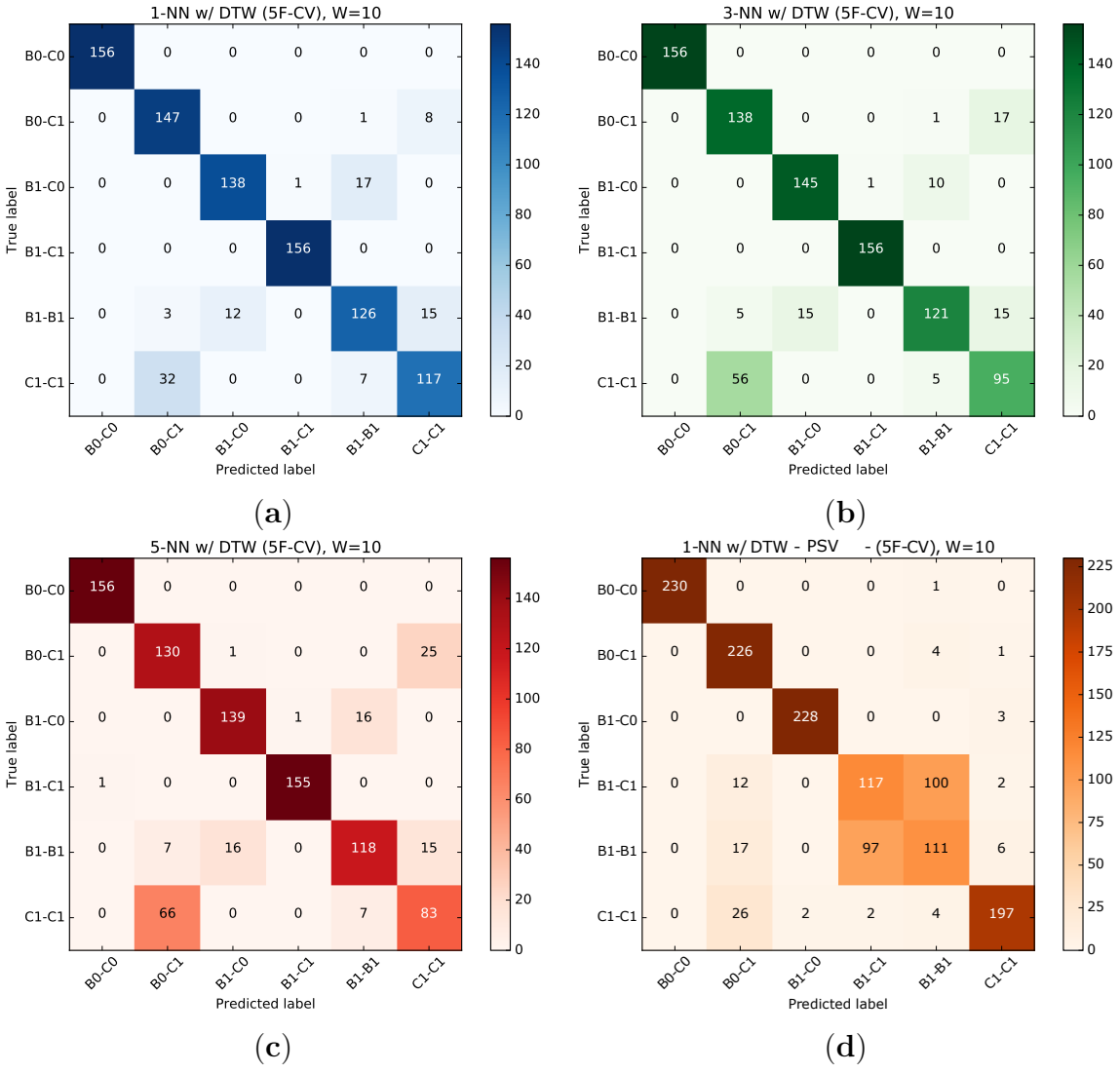
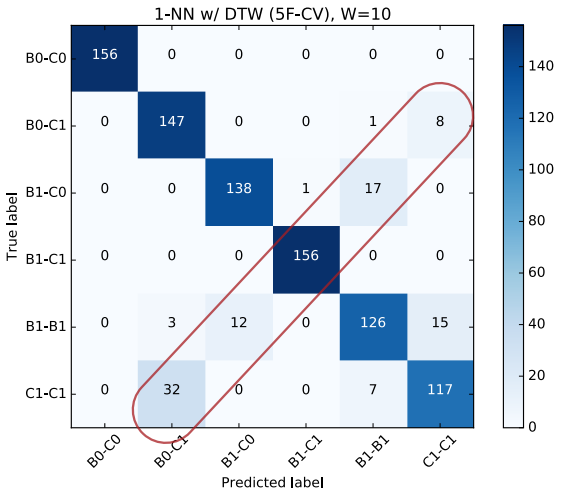


Figure 5.9: Confusion Matrix for classification models for $w = 10$ of the input time-series per k in K-NN with DTW, with and without Active Probing (5-Fold Cross-Validation). (**a**) 1-NN w/ DTW with Active Probing. (**b**) 3-NN w/ DTW with Active Probing. (**c**) 5-NN w/ DTW with Active Probing. (**d**) 1-NN w/ DTW with Passive Probing.



(**a**)

	Precision	Recall	F1-Score	Support
B0-C0	1.00	1.00	1.00	156
B0-C1	0.81	0.94	0.87	156
B1-C0	0.99	1.00	1.00	156
B1-C1	0.83	0.81	0.82	156
B1-B1	0.92	0.88	0.90	156
C1-C1	0.84	0.75	0.79	156
Avg / Total	0.90	0.90	0.90	936

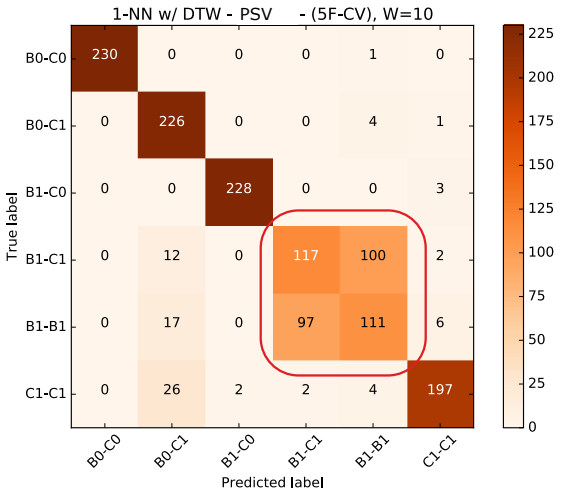
(**b**)

Figure 5.10: K-NN (DTW) Performance ($w = 10$, $k = 1$) with Active Probing. (**a**) Confusion Matrix. (**b**) Classification Metrics.

along with other classification reports are provided in Figures 5.10 and 5.11, respectively. As one concrete observation, active probing has resolved the confusion in making a distinction between B1-C1 and B1-B1 classes (annotated in Figure 5.11a). The distinction between these two classes is important, since it affects our ability to decide if we have homogeneous or heterogeneous protocols in use by the background traffic. With active probing, the confusion in determining CCAs is mostly eliminated, and only the less critical confusion remains the distinction between the number of streams of the same protocol running on the network (annotated in Figure 5.10a).

5.7 Concluding Remarks

In this chapter we examined the ability to recognize the type and mixture of background traffic on a shared network. We have developed techniques to probe the end-to-end network path, producing RTT time-series data. We then put those time-series



(**a**)

	Precision	Recall	F1-Score	Support
B0-C0	1.00	1.00	1.00	231
B0-C1	0.80	0.98	0.88	231
B1-C0	0.99	0.99	0.99	231
B1-C1	0.54	0.51	0.52	231
B1-B1	0.50	0.48	0.49	231
C1-C1	0.94	0.85	0.90	231
Avg / Total	0.80	0.80	0.80	1386

(**b**)

Figure 5.11: K-NN (DTW) Performance ($w = 10$, $k = 1$) with Passive Probing. (**a**) Confusion Matrix. (**b**) Classification Metrics.

data towards training classification models to identify the mixture of background traffic based on the observed end-to-end RTT time-series. Our evaluation results showed that, for the mixtures of TCP CUBIC and TCP BBR, a classification model is able to identify the mixture of the protocols in use with up to 95% accuracy.

One future direction to expand on this chapter is to train classifiers for a larger number of streams. Moving towards larger mixtures of background traffic, the target classes for training the classifier might be adjusted to subjectively represent different aspects of the aggregate behavior of protocols in use (e.g., proactive vs. reactive CCAs), as opposed to predicting the exact mixture of CCAs present on the network.

In the next chapter we will try putting our classification model towards an adaptive protocol selection to optimize the data transfer on a shared network.

Chapter 6

OPS: Optimization through Protocol Selection

In this chapter we discuss optimization through protocol selection (OPS) as a strategy for optimizing data transfer performance through a decision making process (i.e., protocol selection) on shared networks. As discussed in Chapter 4, knowledge about the protocols in use by background traffic might influence our choice of protocol for new foreground data transfer. In Chapter 5 we devised a ML-based method for generating such knowledge about the protocol mixture in the background traffic. OPS uses the classifiers from previous chapter to build an iterative strategy for optimizing the data transfer throughput, while maintaining fairness, through protocol selection (Figure 6.1).

Informed by the scope of our available classifiers, we build OPS to handle mixtures of TCP CUBIC and TCP BBR as the background. Clearly, the same strategy may be extended for a wider range of protocols in the future. Our proof-of-concept implementation of OPS shows up to four times improvement compared to the worst case for throughput (Figure 6.9a, middle vs. right bars). At the same time, the fairness measure (Jain Index) is improved by about 50% when OPS is used for protocol selection compared to the worst case for fairness. (Table 6.3).

We start by providing detailed insights about OPS operation cycle (Section 6.1). Then we discuss the decision making process in Step 3 (Section 6.4), as well as the

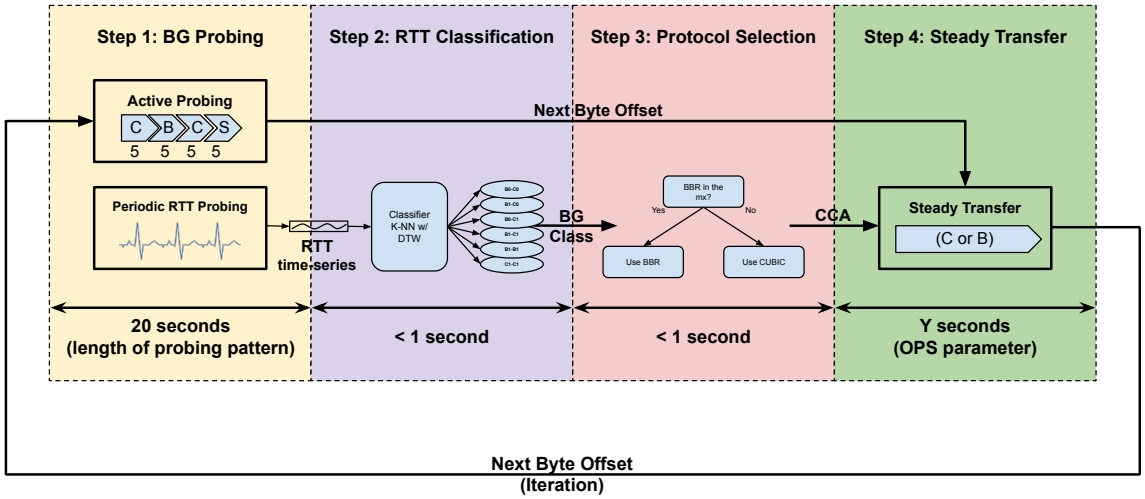


Figure 6.1: Iterative OPS operation cycle, comprised of four main steps: (1) Background Probing (§ 6.1.1), (2) RTT Classification (§ 6.1.2), (3) Protocol Selection (§ 6.1.3), (4) Steady Transfer (§ 6.1.4) (This is an extended version of Figure 1.1).

data transfer stages in Step 1 (i.e., active probing) and Step 4 (Section 6.3). Finally we build and evaluate a proof-of-concept implementation for OPS (Section 6.5).

6.1 OPS Operation Cycle

As depicted in Figure 6.1, OPS iterative design consist of four steps: (1) probing background through periodic RTT probing (passive or active), (2) classifying the generated RTT time-series, (3) deciding on a protocol for data transfer, and (4) an episode of steady data transfer.

6.1.1 Step 1: Background Probing

Probing the RTT is the first step in recognizing the mixture of background traffic. As discussed in Section 5.2, passive or active probing policies are available to use for this step. For passive probing the only task is to periodically probe the end-to-end RTT at every second, forming a RTT time-series to be used by the following step for classification. For all OPS implementations in this research we rely on the *ping* utility provided by the operating system for probing RTT.

For active probing, parallel to the periodic RTT probing, we conduct an segment-

wise data transfer using the specified probing pattern (e.g., C5-B5-C5-S5). As we will discuss in Section 6.3, in conducting the active probing we use data from the intended file for data transfer, hence we refer to it as *constructive active probing*. Assuming a large data file, in each episode of active probing we only transfer a fraction of the whole file. Hence we need to keep track of the last byte offset transferred, so we could resume the data transfer in the Step 4 (tracked as "Next Byte Offset" in Figure 6.1).

Depending on the configured active probing pattern, or w parameter for passive probing, the time length of this phase will be determined. Under the current configurations the length of this step will be 20 s. However, the length of probing phase may be reduced through applying more sophisticated probing patterns, or via gathering more metrics than RTT alone. We will further discuss this topic as part of the future work in Chapter 8.

6.1.2 Step 2: RTT Classification

In this step we use one of the ML models from Chapter 5 (Figure 5.4a) for classifying the background mixture. It will take the resulting RTT time-series from Step 1 as the input and predict the class of background mixture.

It is important to note that depending on the use of passive or active probing, and specific probing length (w) or probing pattern, a separate classifier model should be trained. The classification model is composable, meaning that it may be replaced by alternative models for the future without impacting the rest of the steps. Further details on the design and training of classifiers are available in Chapter 5.

Depending on the classification model the inference time might be slightly different. Using the classifiers we trained the length of this step (i.e., inference time) should be about 1 second or less.

6.1.3 Step 3: Protocol Selection

This step consists of a decision making model to map the classification result from Step 2 to one of the available protocols (i.e., CCAs) to be used for data transfer. The presence of this step provides an opportunity to compose a simple or complex decision logic towards protocols selection. In more complex scenarios this decision making could rely on an external context, in addition to the classification result, for protocol selection. For the scope of this research we only use the classification result to select a protocol. We will use a simple one-level decision tree for this step. We will discuss more details on building this decision model in Section 6.4.

Considering the simple design for the decision model in our study, it only takes a fraction of a second for this step.

6.1.4 Step 4: Steady Data Transfer

Employing the protocol decided at Step 3, the main episode of data transfer takes place in Step 4. It includes transferring data for Y seconds, where Y is a configuration parameter for OPS. At the beginning of this phase the file offset is set according to the "Next Byte Offset" if available from last iteration, or from Step 1 in the case of active probing. This will set the data transfer to resume transferring data towards completion. If Y seconds expire before reaching the end of file, the transfer is interrupted, passing the last byte offset of transferred data to the next iteration.

The length of this step is equal to parameter Y, and we use *OPS-Y* notation to refer to the OPS invocation of different values for Y parameter.

6.2 OPS Conceptual Overheads

Following the introduction to the OPS operation cycle, in this section we discuss the conceptual overheads of the OPS operation cycle per Figure 6.1.

1. **Step-wise overheads.** For the Step 1, assuming active probing is in use, we

transfer data for the whole period of probing phase except for the silence segments, denoted by S in the probing pattern. The silence segment is an overhead since no data is being transferring during this time. The other overhead in this step is the change of CCA in use across segments of data transfer, causing a transient decrease in sending rate when transitioning to new CCAs.

For Step 2 and Step 3, they each only take a fraction of a second to execute. in our current design (1-NN w/ DTW in Step 2 and 1-level decision tree in Step 3). This sub-second latency is negligible compared to the time spent for steady data transfer (i.e., Step 4). If more complex classification and decision models are going to be used in the future, more sophisticated inference runtime environments, hardware or software, might be considered to keep the latency of these steps minimal.

Finally, after deciding about a CCA to use, in Step 4 we will transfer data without any control intervention required from OPS until the steady transfer expires after Y seconds.

2. **Amortized overhead.** The end-to-end performance gain of informed protocol selection outweighs the overhead of individual steps through the OPS cycle. Still, the amortized overhead of the Steps 1 through 3 may be further controlled through adjusting the Y parameter, the length of steady data transfer for Step 4. The total length of time spent for Steps 1 through 3 is near constant. Hence, increasing the Y parameter will decrease the amortized overhead of the rest of the steps in the end-to-end OPS operation cycle. As we will see in Chapter 7 we will use values of up to 360 seconds for Y parameter, making the amortized overhead of the silence segments of probing pattern and sub-second latency of classification and decision steps relatively negligible (cf., Figure 7.10, OPS-360).

   It is worth noting that picking the right value for Y parameter is a trade-off. Larger values for Y would reduce amortized overhead by reducing the frequency

of the probing cycle. However, it would also degrade OPS responsiveness on networks with highly dynamic workloads. In contrast, smaller values for Y would imply more frequent iterations of OPS cycle which increases the overhead. However, more frequent iterations improves the OPS latency in response to a changing mixture of background traffic (cf., Figure 7.10, ALT-30).

6.3 Constructive Active Probing for Data Transfer

Active probing is an intrusive method which includes conducting traffic on the network (Section 5.2). It perturbs the current traffic on the network through short bursts of data transfer (i.e., probing pattern), triggering the existing streams to better reveal the unique signatures of their underlying CCA. However, sending such random traffic on the network (even in small bursts) could be considered as unproductive use of bandwidth, wasting the available network resources.

Constructive active probing is a mechanism for addressing the wasteful use of bandwidth in active probing. It uses real user data (instead of random dummy traffic) for the short bursts of active probing, hence the term *constructive*. One approach to realize this mechanism is to define a global offset variable during the data-transfer task, tracking the next byte offset in data file to be used for either active probing (Step 1) or steady transfer (Step 4). This is marked as "Next Byte Offset" in Figure 6.1.

When initiating each data transfer segment (for either Step 1 or Step 4), the offset variable value is used as the data file offset to start transferring data. At the end of each data transfer segment the offset variable is updated to track the next byte offset to be used by subsequent data transfer segments for resuming the file transfer. We should note that if passive probing is configured, the offset variable will be only used by Step 4 across iterations towards completing the file transfer.

We will implement constructive active probing mechanism in Chapter 7 for building iPerfOPS.

6.4 Decision Making for Protocol Selection

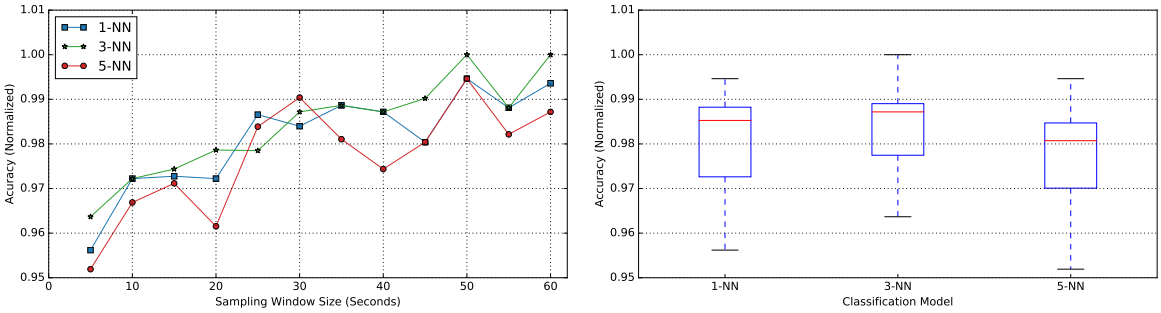
As earlier discussed, a decision making model is needed in Step 3 of OPS for applying the classification output (Step 2) towards selecting an appropriate protocol for data transfer (Step 4). In other words, we need to map the recognized traffic class to one of the protocols available for data transfer. In this section we review such decision making model in more details. We introduce two alternative decision models built on top of classifiers' output from previous chapter, as summarized in Table 6.1. Then we pick one model as the decision model for protocol selection in OPS.

6.4.1 Two Decision Scenarios

The ultimate goal for building a traffic classifier is to generate useful insights, supporting better decision making under different networking scenarios. In this section, we define two decision making scenarios, as it might arise. We then study how accurately the discussed classifiers would perform in decision making. For both scenarios, we take the direct output of the background classifiers (with active probing) and reduce it to generate the desired decision results.

Table 6.1: Recognizable background mixtures via classification models. Six classes of up to two streams of TCP CUBIC or TCP BBR (cf., Figure 5.1).

Traffic Class	BBR Stream#	CUBIC Stream#
B0-C0	0	0
B0-C1	0	1
B1-C0	1	0
B1-C1	1	1
B1-B1	2	0
C1-C1	0	2



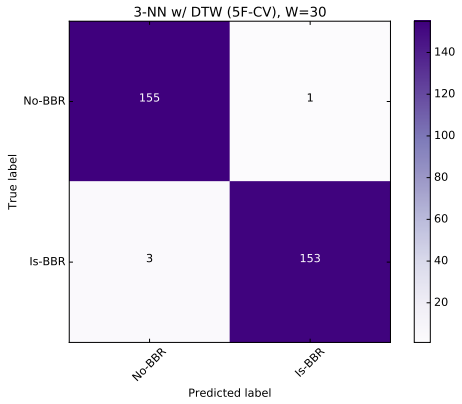
(a) Classification Accuracy per window size (b) Box Plot for Classifiers Accuracy

Figure 6.2: Decision Scenario 1: Is BBR present or not? Accuracy of classification models for varying time length w with Active Probing (5-Fold Cross-Validation)

Decision Scenario 1: Is BBR Present or Not?

In the first scenario, we consider the problem of deciding whether any TCP BBR traffic exist in the background mixture or not. The motivation for this decision is that there is a known fairness issue when TCP BBR competes against TCP CUBIC on a shared network, as discussed in Section 4.2. As a result, identifying whether TCP BBR traffic exist on the network would help both end-users (to decide on which protocol to use) as well as network administrators (for example, to define regulations for bandwidth allocation).

We used the classification performance from Chapter 5 to generate the accuracy results for this decision model. The accuracy performance and sample classification metrics for this decision scenario using active probing are provided in Figure 6.2 and Figure 6.3 respectively. The accuracy of this decision scenario is between 95% and 99%, representing near perfect performance in identifying if BBR CCA is present in the network or not. While this is a simplistic scenario, the promising, near-perfect, results yield a great opportunity in scaling and deploying this model for decision making.

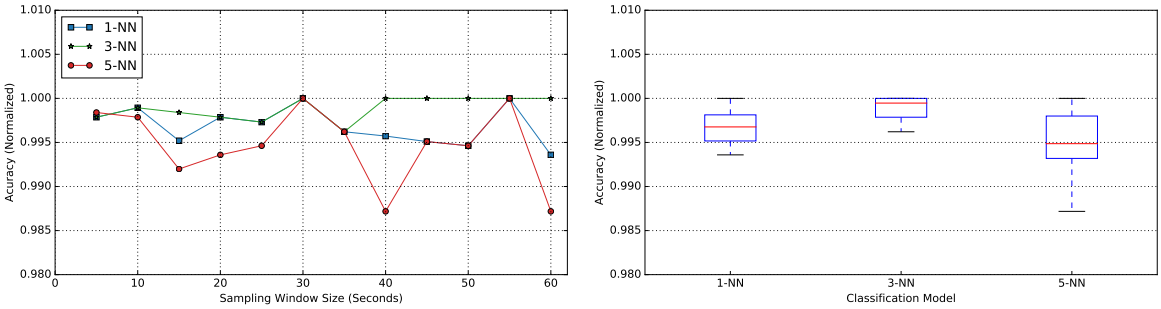


(a) Confusion Matrix

	Precision	Recall	F1-Score	Support
No-BBR	0.99	0.98	0.99	156
Is-BBR	0.99	0.98	0.99	156
Avg / Total	0.99	0.98	0.99	312

(b) Classification Metrics

Figure 6.3: Decision Scenario 1: Is BBR present or not? K-NN w/ DTW performance ($w = 30$, $k = 3$) with Active Probing



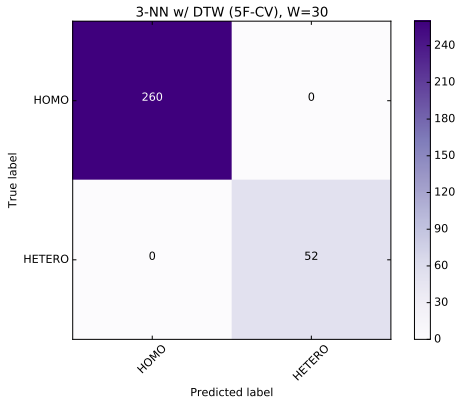
(a) Classification Accuracy per window size

(b) Box Plot for Classifiers Accuracy

Figure 6.4: Decision Scenario 2: Homogeneous or Heterogeneous CCAs? Accuracy of classification models for varying time length w with Active Probing (5-Fold Cross-Validation)

Decision Scenario 2: Homogeneous or Heterogeneous CCAs?

In second scenario, we aim to answer a more general question: whether the mixture of background traffic consists of homogeneous or heterogeneous protocols. Similar to the last scenario, this scenario would have potential use-cases among users, administrators, or infrastructure providers. Similar to the Scenario 1, applying our K-NN based model with active probing yields promising, near perfect, performance in this scenario as well. The results are provided in Figure 6.4 and Figure 6.5.



(a) Confusion Matrix

	Precision	Recall	F1-Score	Support
HOMO	1.0	1.0	1.0	260
HETERO	1.0	1.0	1.0	52
Avg / Total	1.0	1.0	1.0	312

(b) Classification Metrics

Figure 6.5: Decision Scenario 2: Homogeneous or Heterogeneous CCAs? K-NN w/ DTW performance ($w = 30$, $k = 3$) with Active Probing

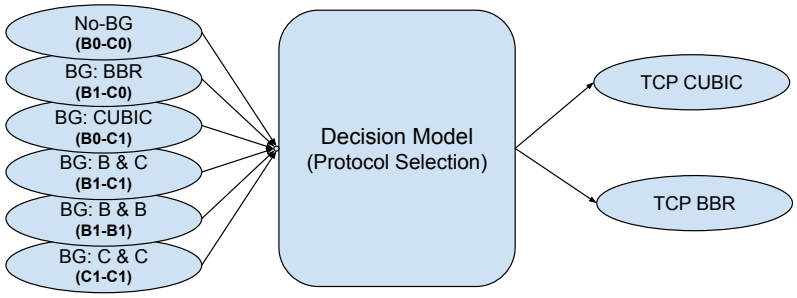


Figure 6.6: OPS Decision making: mapping classification result to a CCA

6.4.2 OPS Decision Model

One key step in the OPS workflow (i.e., Step 3) is to decide on a protocol to be used for data transfer for optimizing the overall performance. This decision making should be informed by the result of the background classification (Step 2). We need to map each predicted class to an appropriate action, namely, to choose a protocol from the catalog of available CCAs (Figure 6.6). For the scope of this study our catalog of available protocols includes TCP CUBIC and TCP BBR.

In the last section we reviewed two possible decision making scenarios. We employ the first decision scenario, whether BBR is present in the background or not, for OPS protocol selection. This decision scenario is directly applicable to output of RTT classification, allowing us to answer the question about TCP BBR presence with high

	FG: BBR	FG: CUBIC
BG: BBR	Good Throughput Good Fairness	Poor Throughput Poor Fairness
BG: CUBIC	Good Throughput Poor Fairness	Good Throughput Good Fairness

Figure 6.7: Interaction matrix for TCP CUBIC vs. TCP BBR.

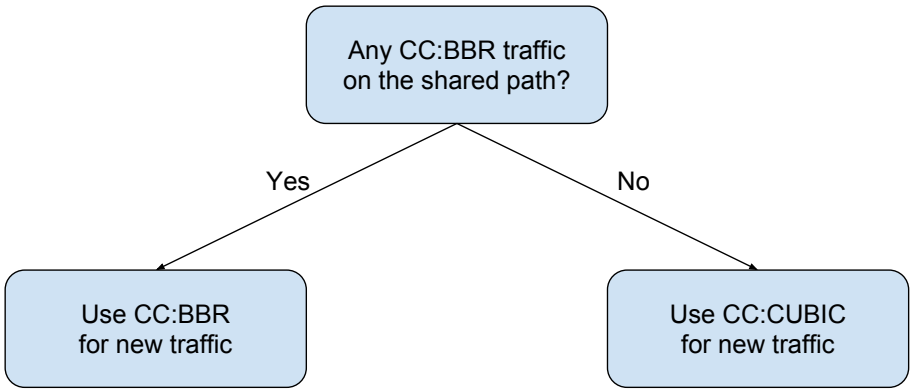


Figure 6.8: Binary decision model for choosing a CCA for new data transfer.

accuracy (Figure 6.2). We could then use the answer to this question along with our findings from Chapter 4 to inform our protocol selection.

Per Section 4.2, the interaction between TCP CUBIC and TCP BBR could be summarized as the matrix presented in Figure 6.7. Following the balanced throughput and fairness when we have homogeneous CCAs on the network (all CUBIC or all BBR) we build our ultimate protocol selection model to pick TCP BBR if BBR is already present in the background. Otherwise we will use TCP CUBIC to maintain a CUBIC-only mixture on the network. This decision logic is presented in Figure 6.8 as a one-level decision tree. Evident by the ML research literature, a model as simple as a one-level decision tree could perform sufficiently well in many cases [85, 86].

The intuition behind this decision strategy is as follows: BBR significantly hurts CUBIC performance on the network (cf., Figure 4.7). Hence, if we discover that the

Table 6.2: Steps configuration for evaluating OPS performance.

Step	**Configuration**	**Parameters**
(1) Background Probing	Passive Probing	RTT Resolution = 1 s RTT time-series length = 20
(2) RTT Classification	K-NN w/ DTW	k = 1 max-warping = 10
(3) Protocol Selection	One-Level Decision Tree	If BBR present, use BBR Otherwise, user CUBIC
(4) Steady Transfer	Single-stream TCP	Length of transfer = 60 s

existing traffic on the network includes at least one BBR stream, we use BBR for the new data-transfer task; BBR will be fair to the other BBR traffic on the network, and it also will be able to claim a fair share of bandwidth for data transfer. In contrast, if we predict the network traffic to be CUBIC only, we choose CUBIC for the new data transfer. This scheme will claim a reasonable share of the bandwidth from the network, while not hurting the other users.

The performance results of applying this decision model to a data-transfer tasks will be discussed in Section 6.5.

6.5 Evaluation: OPS for Protocol Selection

In this section, we put all the steps together to build a proof-of-concept OPS pipeline for protocol selection. The configuration used for OPS steps in this section are summarized in Table 6.2. For background probing (Step 1), we limit the scope for OPS evaluation in this section to passive probing. The active probing evaluation will be discussed in the next chapter where we build a stable implementation of OPS (i.e., iPerfOPS).

For RTT classification (Step 2), we evaluate OPS performance based on the 1-NN classifier with passive probing. Notably, higher accuracy for active probing implies

that the overall performance of our OPS software when extended to use active probing will be at least similar to, and more likely to be better than, the results we present in this section. For consistency with the active probing configuration in the future, we use $w = 20$ as the length of RTT time-series (i.e., length of probing). Per Figure 5.5a the classification accuracy for 1-NN with $w = 20$ is 82%. It presents a reasonable accuracy relative to the size of training data we used, and we predict that by increasing the size of training data, or training more sophisticated models, this accuracy would gradually increase.

For protocol selection (Step 3), we use the one-level decision tree as discussed in Section 6.4.2. It decides either TCP CUBIC or TCP BBR based on whether TCP BBR is already present on the network or not. The decided CCA is then used for steady data transfer (Step 4), where iPerf tool is used for transferring data for 60 seconds, i.e. OPS-60.

Using the discussed configuration (Table 6.2), we conducted a series of experiments, running a 30 min long static (i.e., non-changing) traffic over the network as the background traffic of either TCP CUBIC or TCP BBR. We repeated this test three times, each time using one of the following strategies for sending foreground traffic. (A total of six experiments were conducted with a combination of two background and three foreground schemes.):

1. Fixed: CUBIC. Always TCP CUBIC is used for data-transfer phase.

2. Fixed: BBR. Always TCP BBR is used for data-transfer phase.

3. Adaptive (i.e., OPS): Dynamically determines which protocol to be used for each data-transfer cycle.

Figure 6.9 presents the average throughput of background (BG) and foreground (FG) streams per each combination, also provided along with fairness measure in Table 6.3. In Figure 6.9a, by introducing BBR traffic in Fixed:BBR (middle bars) the

shared bandwidth of background CUBIC drops drastically, resulting in poor fairness with a Jain index of 0.62. In contrast, the Adaptive (OPS) mode (right-hand-side bars) dynamically probes the network condition and adjusts the network protocol, reclaiming four times more bandwidth, comparable to the ideal case of Fixed:CUBIC scenario with a Jain index of 0.99. Similarly, in Figure 6.9b, the Fixed:CUBIC mode (left-hand-side bars) suffers from extreme unfair bandwidth sharing in presence of background BBR (Jain index: 0.60). Using the Adaptive (OPS) (right-hand-side bars) significantly improves on fairness, again reclaiming four times more share of the bandwidth (Jain index: 0.96).

While the accuracy of the particular classifier we utilized is 82% (Passive Probing, Figure 5.5a), the outcome is quite satisfactory, compared to the extreme cases of blindly choosing a sub-optimal protocol for data communication over the network. On the one hand, choosing CUBIC to use over a BBR-present network will significantly degrade our data-transfer performance. On the other hand, choosing BBR over a CUBIC-only network would severely hurt the other network users sharing the same organizational or research bandwidth. This in turn might imply negative consequences to other traffic flows of our own, such as, delaying a whole data communication pipeline because of an aggressive traffic stream over the network.

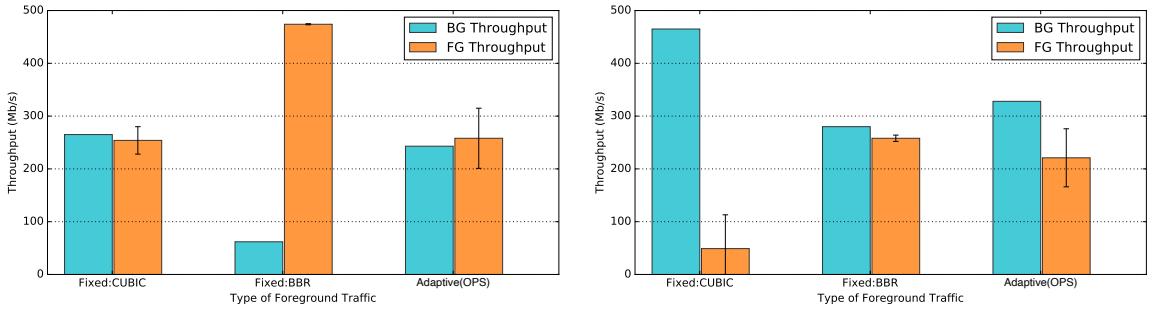


Figure 6.9: Average throughput of fixed versus adaptive (OPS) foreground protocol selection, in the presence of static (non-changing) background. (**a**) Background Traffic=CUBIC, OPS improves fairness while maintaining a reasonable bandwidth by using CUBIC in foreground. (**b**) Background Traffic=BBR, OPS maintains fair throughput by using BBR in foreground.

Table 6.3: OPS performance: throughput vs. fairness for different combinations of TCP CUBIC and TCP BBR for background (BG) and foreground (FG)

BG	FG	Throughput (BG,FG)	Jain Fairness	Figure
CUBIC	CUBIC	265 Mb/s, 254 Mb/s	0.99	Fig. 6.9a, left
	BBR	62 Mb/s, 474 Mb/s	0.62	Fig. 6.9a, middle
	OPS	243 Mb/s, 258 Mb/s	0.98	Fig. 6.9a, right
BBR	CUBIC	465 Mb/s, 49 Mb/s	0.60	Fig. 6.9b, left
	BBR	280 Mb/s, 258 Mb/s	0.97	Fig. 6.9b, middle
	OPS	328 Mb/s, 221 Mb/s	0.96	Fig. 6.9b, right

6.6 Concluding Remarks

In this chapter, we introduced OPS as new model for dynamic protocol selection in the context of shared networks. In OPS operation model we periodically probe the background traffic on the network using regular RTT probing and use a classification model to classify the resulting time-series, hence identifying the mixture of traffic in terms of protocol and CCAs. The identified background mixture is then used in a decision making process (decision model) to select an appropriate protocol or CCA to improve data transfer performance (throughput and fairness) on the underlying shared network. We have implemented a complete proof-of-concept OPS cycle, evaluating the performance using synthetic data-transfer tasks.

It is worth noting that while the particular CCAs we used here for OPS evaluation might get evolved, or deprecated in favor of others, the OPS strategy for adaptive protocol selection (as opposed to fixed protocol) would still remain relevant. It could significantly contribute to the performance of data-transfer tasks at any time assuming appropriate classification model and sufficiently-sized protocol catalog are available.

In the following chapter, we will introduce and discuss iPerfOPS, our end-to-end implementation of OPS as a Linux-based data transfer tool.

Chapter 7

iPerfOPS: An Adaptive Protocol Selection Tool

Following the OPS proposal from previous chapter, in this chapter, we describe the design, implementation, and evaluation of iPerfOPS, the first tool that uses OPS to perform bulk-data transfer. End-to-end evaluation of iPerfOPS yields a throughput improvement of more than double compared to the worst case (CUBIC used against BBR). Furthermore, the cross-stream fairness is improved, reaching close to the fair (equal) sharing for some configurations (e.g., OPS-100, 100 denotes length of steady transfer in seconds).

iPerfOPS is a substantially modified version of the well-known iPerf tool [74]. It is an end-to-end implementation that incorporates the results from the previous chapters. In brief, we adopt iPerf due to its extensive benchmarking insights and its highly configurable fixed-time and fixed-size traffic conduction. Moreover, iPerf is a well-optimized open-source tool, with great portability across different platforms.

Through this chapter, we present the high-level design as well as implementation details of iPerfOPS. First we provide a more in-depth discussion on adopting iPerf as the underlying data transfer tool and the extensions we added to iPerf to support OPS use-cases. Then we provide the high-level architecture of iPerfOPS as a standalone, end-to-end, data transfer tool. Finally, we conduct evaluation and provide end-to-end performance results for iPerfOPS.

7.1 Development Strategy: Adopting iPerf Library

In our efforts to implement OPS as an end-to-end, production-ready, data-transfer tool we considered two options: First, build OPS from scratch, using the native Linux interfaces and libraries (e.g., network sockets, files). Second, leverage an existing tool with a programmable API or library, and reuse its high-level data transfer constructs and abstractions. While the first option provides finer control for building underlying constructs and workflows from scratch, the second option empowers us to take benefit from a wide range of well-designed and stable community-provided tools. The use of existing tools could also positively contribute to the overall quality of our resulting tool, while saving us on required development efforts. For the second option, building on an existing tool, there are two main categories: data-transfer tools, and network benchmarking tools.

Since OPS is designed to improve data transfer efficiency, the data transfer tools seem to be the best fit for this purpose. TCP-based tools, such as GridFTP [17], are immediately capable of serving as the data transfer channel of a given TCP CCA. Furthermore, UDP-based data transfer tools, such as UDT [29], provide ultimate flexibility in fine tuning and customization of CCAs to be used for data transfer. Meanwhile, the lack of two important features in data transfer tools renders them insufficient for building OPS.

First, data transfer tools are incapable of conducting fixed-time transfers. They are mainly built for transferring a full file to the completion. While they may be tweaked, by the means of I/O redirects or otherwise, to partially transfer a file, they will still function based on the byte-count and size of file (i.e., fixed-size). However, the OPS strategy requires the capability of transferring data blocks in a fixed-time fashion to allow conducting different patterns for active probing, and to transfer data for a fixed period of time before re-probing the background traffic.

Second, data transfer tools usually function as black-box tools, merely providing

any insights or metrics of the data transfer performance. In most cases, the end-to-end throughput, reported directly or via the proxy of transfer time, is the only performance metric available. Those tools usually use various internal metrics and counters to optimize the data transfer, but such metrics remain internal to the tool and are not accessible, or meaningful, to be used as performance proxies to the outside world.

Network benchmarking tools, contrary to data transfer tools, mostly provide flexibility in terms of end condition (fixed-time and fixed-size). In addition, those tools by design provide a larger range of performance metrics. Some of the well-known tools in this group include Netperf [73], Nuttcp [87], and iPerf [74]. The available performance metrics based on the tool in use could include average throughput, instantaneous (interval) throughput, jitter, latency, and more. The presence of described properties makes this group of tools a desirable candidate to be utilized for implementing the OPS.

iPerf is a network benchmarking tools with active community and on-going development and maintenance. In addition, iPerf serves its full functionality via an API library, making it a great networking software for building new tools. Lastly, iPerf is an open-source software, enabling us to make any necessary adjustment to its source code. Hence, we decided on using iPerf as the underlying library for implementing OPS. We should note that iPerf has a legacy version known as iPerf2, which is now replaced with iPerf3. Throughout this chapter any references to iPerf means iPerf3, unless otherwise mentioned.

For OPS, iPerf has two limitations, one functional and one performance-related. First, the functional inadequacy, is the lack of reliable file transfer. iPerf supports reading data from a file for probing the network. However, it does not implement reliability in file transfer, and terminates the data transfer socket as soon as the termination criteria is reached on the sender (specified byte-count or seconds-length). Second, the performance inadequacy, is that iPerf creates and destroys network sockets

per invocation, making it I/O-intensive if there are multiple subsequent invocations happening (e.g., in active probing patterns for OPS). In the following section we will discuss and report on our approach on modifying and extending iPerf to address these inadequacies.

7.2 iPerf-DT: iPerf for Data Transfer

As discussed in the previous section, iPerf contains two inadequacies to accommodate OPS implementation: The lack of reliable file transfer (functionality), and the lack of network socket reuse (performance). Since the two inadequacies are internal to iPerf, we need to address and fix them in the iPerf source code. We first provide a detailed report and analysis on the internal architecture and operation cycle of iPerf. Then we present our solution to the inadequacies in the following sections. For distinction, we use *iPerf-DT* to refer to our modified version of iPerf.

7.2.1 iPerf Architecture and Operation State-Machine

iPerf design follows a client-server model, operating based on a stateful application-level protocol and using distinct control channel and data channel(s) for client/server coordination. The operation sequence of a standard iPerf data transfer benchmarking is provided in Figure 7.1. The iPerf state-machine is also provided in Figure 7.2.

iPerf client and iPerf server use control channel for three distinct use cases:

1. Initial test setup: swapping test parameters and initiating the task.
2. Ongoing sync: during the test client and server use control socket to inform and prompt the other party to transition to a new state.
3. Communicating results: at the end of the data transfer client and server swap their statistics before terminating the task.

First, the iPerf is started on in server mode on destination node. iPerf server by default binds and listens to port 5001 (which may be customized). Once the server

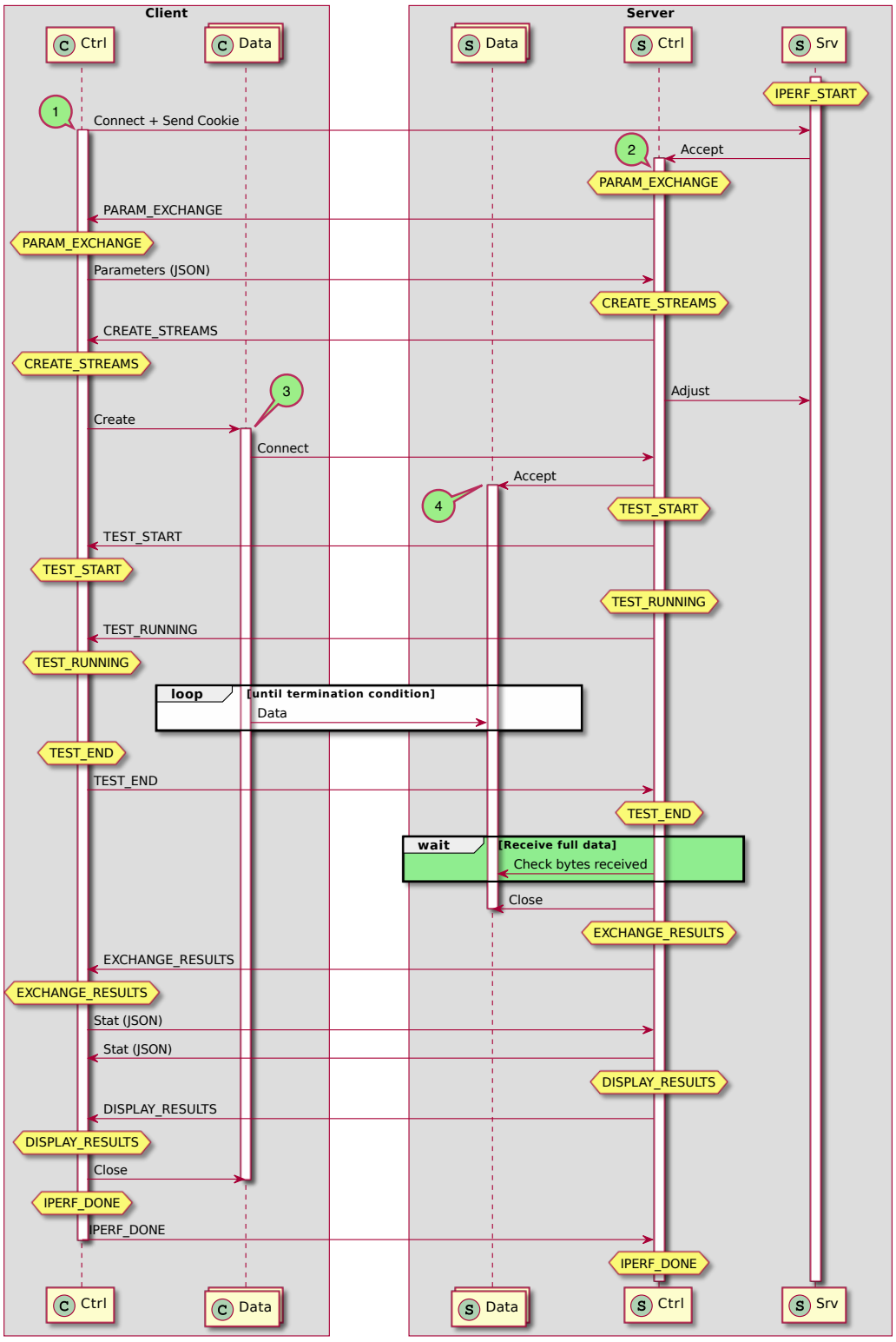


Figure 7.1: iPerf operation sequence diagram. The yellow labels indicate what the active state is at each point during the operation for client and server (Figure 7.2). Annotations in green color indicate the new extensions in iPerf-DT (vs. original iPerf), addressing functional (§ 7.2.3) and performance (§ 7.2.4) inadequacies.

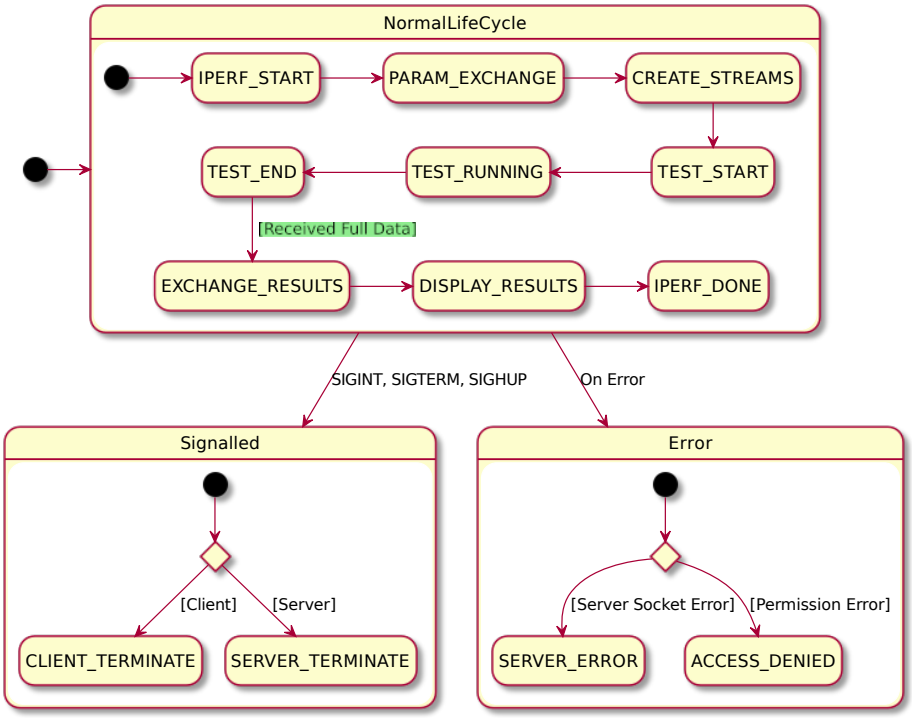


Figure 7.2: iPerf State Machine. Both client and server follow the same state transitions. The state transitions are mostly coordinated by the server (Figure 7.1), except for `TEST_END` and `IPERF_DONE`, or if client is signalled (`CLIENT_TERMINATE`).

is up, the other nodes may run iPerf in client mode taking the server IP for traffic conduction. Then, iPerf client starts by communicating the server instance. After an initial hand-shaking between the client and the server, data channel(s) are created and the requested benchmarking takes place which involves a fixed-time or fixed-size data transfer using the provided configuration. A few of relevant iPerf configurations include the following:

1. Transport protocol: it supports TCP, UDP, and SCTP. Default is TCP.

2. Number of data streams: it supports single or parallel data streams for data transfer. Default is a single stream.

3. Congestion Control (CCA): in case of TCP and SCTP transports, the CCA could be selected. The available options and default CCA are the same as the host running iPerf.

4. Data direction: the data transfer might be directed from client to server, or vice versa. Default is from client to server.

5. Data source: the data on the sender could be randomly generated on the fly, or a file may be used to read data from. Default is random data.

6. Data destination: the received data on the receiver could be discarded, or a file may be specified for writing the received data. Default is discarding the data.

7. End condition: a byte count (fixed-size) or the time length in seconds (fixed-time) may be specified as end condition. Default is fixed-time for 10 seconds.

The relevant iPerf configuration options, along with the configuration we will use for OPS are summarized in table 7.1.

Table 7.1: Relevant iPerf configuration options for implementing iPerfOPS. (C is for Client, S is for Server)

Configuration	**Options**	**Default**	**iPerfOPS**
Transport Protocol	TCP, UDP, SCTP	TCP	TCP
# of Data Streams	Single or Parallel	1	1
CCA	Same as host	Same as host	CUBIC, BBR
Direction	C $\rightarrow$ S	C $\rightarrow$ S	C $\rightarrow$ S
Data Source	Random, File	Random	File
Data Destination	Discard, File	Discard	File
End Condition	Bytes, Seconds	10 Seconds	Seconds

iPerf server is designed as a single-tasked service, meaning that it could only engage in a single data transfer at any given time. As soon as a client request arrives and the connection is accepted as a control channel, the server becomes unavailable and any new clients trying to connect while the current server engagement is not completed will get denied with an error message. Once the current experiment is completed, both client and server communicate results and then terminate all sockets and cleanup. Server then goes back to listen for new incoming connection requests.

For both fixed-time and fixed-size data transfers, the sender puts data on the data channel socket(s). When the termination condition is reached, client sends a signal for server (`TEST_END` state) indicating the end of test. at this point server immediately closes the data sockets (without waiting for the whole data to be received), and signals the client that it's ready for exchanging the results (`EXCHANGE_RESULTS` state). This behavior on the server side is the core cause for the unreliability of data transfer in iPerf (i.e., the functional inadequacy).

As introduced in the earlier sections, iPerf does not support keeping the connection open between client and server for conducting more than one data transfer. Even for the same pair of client and server nodes, they will both create and destroy network sockets (control and data channels) per each data-transfer task, which could become a performance overhead for transfers with several subsequent data transfer between the same pair of client and server node (i.e., the performance inadequacy).

With this brief review on iPerf internals, we will discuss our solutions for the two inadequacies in the following sections.

7.2.2 Bug Fixes for the Original iPerf

While working on the iPerf source code to identify and fix the inadequacies specific to OPS, we identified a few logical and implementation bugs in the official iPerf project source code [8–12, 88, 89]. Those bugs were not specific to OPS logic, rather generic logical bugs which have not been identified by the iPerf development community yet.

On the one hand, those bugs were within the scope of the official iPerf project. On the other hand, fixing those issues are necessary to allow us implementing our OPS-specific adjustments and extensions to iPerf. So we have addressed and fixed those bug, and our bug fixes were merged back into the mainstream (*master* branch) repository for iPerf project.

In this section we briefly review those bug fixes.

Missing features in the API library. iPerf library offers its functionality

through two alternative interfaces: CLI use and Library access. Some of the features already available through CLI use were identified to be missing int the library. In particular, the important missing features for our project were the functions to get or set the CCA through library access. We have added the missing functionality to the iPerf API library [8].

Incorrect termination test in bytes/blocks mode. While running iPerf tests in bytes or blocks mode (i.e., fixed-size mode), we identified that iPerf tends to send a larger number of bytes than specified as the termination condition. After careful inspection and code review, we discovered a logical bug in the main data transfer loop were causing iPerf to not stopping right at the desired bytes count. We have adjusted the logic for termination test which fixed the problem [9].

Inefficient test termination condition. Since iPerf provides both fixed-size and fixed-time modes, its test termination check should carefully cover all possible cases. In particular, when the tests were conducted in fixed-time mode where source data were read from a file, the termination test condition were causing incorrect stop point to end the test. We have updated the test termination logic and simplified it to sorely rely on a *test-done* flag which would cover all the existing test modes, while making it straightforward in the future to add new termination logic specific to each mode [10].

Incorrect data transfer size in file mode. When transferring data from a file, the iPerf reads data from file into a buffer and sends data from buffer to the network, repeating this cycle until done. The main logic for sending data from file was not accurately considering the end-of-file condition, hence sending an incorrect amount of data in the last iteration (last chunk of file in the buffer). We have fixed the logic to test for the end-of-file and appropriately handle the non-full buffer while accurately setting the test-done flag [11].

Auto adjusting test end condition in file mode. In this case we implemented an enhancement for tests where data is read from a source file. As earlier mentioned,

the two standard modes for test termination in iPerf are fixed-size and fixed-time. In the case of transferring data from a file we have added a new mode, allowing the iPerf to auto adjust the length of the data-transfer task to match the file size [12]. For distinction we refer to this new mode as fixed-file.

7.2.3 Functional Extension: Reliable File Transfer

The bug fixes reviewed in the previous section were necessary building blocks in order to get iPerf sending the exact number of bytes from a source file to the network. However, as we discussed in Section 7.2.1, the iPerf's operation logic on the server side is designed to prematurely terminate the data channels without waiting for the full data to be received. To fix in inadequacy and closing the loop on reliable file transfer, we have made two modifications.

First, in the beginning of a data transfer in the fixed-size (including fixed-file mode), we send the total bytes count from client for the server. As a result, the server will know the size of data it is expecting to receive from the client.

Second, we have adjusted the operation cycle on the server for handling the `TEST_END` prompt sent by the client. The adjustments are highlighted in Figure 7.1, as the green wait box on the server side between `TEST_END` and `EXCHANGE_RESULTS` labels. Upon receiving the `TEST_END` prompt, the server enters a timeout loop and periodically checks the number of bytes successfully received on the data channel. It delays closing data channel until the full data is successfully received. The server, then, terminates the data channel and sends `EXCHANGE_RESULTS` prompt for the client. It is worth noting that in order to avoid interfering with the expected behavior in the original iPerf, the reliable data receive is only triggered if a new flag is set (using switch `-r` on the CLI, or a new library function.

Furthermore, in order to accommodate partially transfer data from a file on the client through different intervals (as with the OPS probing patterns), we added a new functionality to iPerf, allowing the sender to partially send data from a file by seeking

```
# start server
receiver$ ./iperf -s -F /path/to/destination.file,append -r
```

(**a**)

```
# send file
sender$ ./iperf -c receiver -F /path/to/source.file,15000 -t5
```

(**b**)

Figure 7.3: iPerf-DT Sample CLI use for reliable file transfer (**a**) Starting the server, specifying a file to write the received data, setting the flag to append data to the file instead of rewrite (`",append"`) and requesting reliable transfer (`"-r"`). (**b**) Initiating transfer on the client, specifying a file to send data from after seeking to specific offset in file (`",15000"`), and transferring data for 5 seconds (`"-t5"`). It will finish after 5 seconds of transferring data, or if end of file is reached in less than 5 seconds. When duration flag omitted it will send the whole file and then terminate.

to a location and sending specified bytes count. We also adjusted the server to add a mode to append the received data to the end an existing file, rather than replacing the existing file as implemented in the original iPerf.

As a result, the iPerf is now fully reliable in transferring data and writing them to the file on the server. A sample of using iPerf-DT for reliable file transfer is provided in Figure 7.3, seeking to an offset in the source file on the sender along with appending data to file on the server side.

Motivated by making it available to the larger iPerf community, we initially tried to merge the reliability feature into the iPerf mainstream repository. However, the corresponding pull requests to the official iPerf repository were dismissed by the owner, due to adding new scope to the project and requiring extra maintenance and support effort [88, 89]. Hence, we have made the reliable iPerf-DT available as an open-source software on GitHub [13].

7.2.4 Performance Extension: Pooling Network Sockets

The periodic probing interval in the OPS and the idea for constructive active probing implies a sequence of short data transfer episodes until the data transfer is completed.

As such, the efficiency of creating those short data transfer episodes is a critical quality towards achieving production-ready performance.

However, as we already discussed in Section 7.2.1, iPerf creates, configures, and destroys control and data sockets per each transfer which makes it inefficiently resource-intensive for our use-case. The green circles numbered one through four in Figure 7.1 indicate the points where sockets of different types (Control or Data) are being created or accepted during an iPerf data transfer. So we started investigating on how to eliminate this inefficiency. We plan to use iPerf as a library in OPS, using instances of its client and server for data transfer episodes until the transfer is completed. As such, if we manage to preserve the network sockets when initially created, they will be kept alive for the whole duration of a OPS invocation.

One common pattern to predictably control the overhead in applications with high frequencies of socket use is to establish a *Socket Pool*. In this pattern a special data structure takes the ownership and accounts for creating, maintaining, and reusing network sockets. When our application requires a socket it will request the socket pool. If an open and unused socket is available to the desired destination it will be allocated for the use. Otherwise a new socket is created. When we are done with the socket, it will be returned to the socket pool and marked as available for any subsequent request for a socket.

While a socket pool seems to be the right fit to optimize the socket use in iPerf, there exists a challenge in adopting this pattern with iPerf: iPerf concurrently uses control and data socket(s) between a client-server pair of nodes, following significantly different semantics for each of control and data sockets. So we need to coordinate client and server on which existing socket to be used for control or data channel for a new episode of data transfer. Furthermore, due to the relatively complex state-machine in iPerf and the highly formalized use of control socket for communicating the state transitions, we ideally should not disrupt the client-server control protocol as it might make it incompatible with the original iPerf library for the future updates.

Table 7.2: The wrapper functions implemented for socket operations in iPerf-DT

Wrapper Function	**Behavior Adjustment**
TSocket()	If a socket of requested type is available in registry it will be marked as in-use and returned. Otherwise a new socket is created, added to registry and returned.
TAccept()	If a socket of requested type is available in registry it will be marked as in-use and returned. Otherwise a new connection is accepted, added to registry and returned.
Close()	The in-use flag is removed for the socket (It does not get closed)
Bind()	If this is a re-used socket we skip as it is already binded, otherwise we call bind()
Connect()	If this is a re-used socket we skip as it is already connected, otherwise we call connect()
Getsockopt()	only for logging purpose. It calls getsockopt()
Setsockopt()	If new socket call setsockot(). If a re-used socket skip except for setting TCP_CONGESTION option
Listen()	only for logging purpose. It calls listen()
Select()	The most heavy wrapper logic. Based on the direct information from SocketRegistery and which socket types are in use it decides on the order of checking FDs for availability and overrides FD flags if needed.

In order to overcome the challenge of coordinated socket reuse, we have devised a *Typed* Socket Pool, where each socket will be identified as either Control or Data socket. Since we exactly need one of each socket type for any data transfer, it would then suffice to keep track of one instance of each socket type to be reused through the lifespan of an OPS invocation. In addition, to minimize the refactoring efforts and divergence from original iPerf, we intend to keep this socket pooling and reuse mostly transparent to the iPerf modules and functions.

In our ultimate solution we have created wrapper functions for the basic socket library functions. The wrapped library functions are summarized in table 7.2. Each wrapper function named and signature is identical to its corresponding library func-

tion, but named with Capital case initial letter. For the two special functions that create a new socket, `socket()` and `accept()` we also created typed wrapper functions (`TSocket()` and `TAccept()`) which take an additional parameter *SocketType*. The types methods help the wrapper functions to return the existing socket of the desired type when requested, which enables the wrapper functions to remain agnostic about the iPerf internal logic of control versus data channels. We also implemented a data structure `SocketRegistry` for pooling typed sockets. It is worth noting that the SocketRegistery is a simplified version of socket pooling for the special case of iPerf library where we only need one of each socket type at a time, and we only have a single data-transfer task at any given time. Further implementation details are omitted here, but might be inspected in the public open-source iPerf-DT repository [13].

7.3 iPerfOPS Architecture

iPerfOPS is designed as a modular software and following a client-server model. The high-level modular design of iPerfOPS is provided in Figure 7.4. The state chart for iPerfOPS is also provided in Figure 7.5. Upon invocation, iPerfOPS starts an infinite loop of data transfer episodes until the end of file is reached. Each data transfer episode, consists of two phases, probing phase and Steady transfer phase. The probing phase follows passive or active policies (cf., Section 5.2) based on the provided CLI parameters. The result of probing is then used to classify the background traffic and selecting a CCA (e.g, CUBIC or BBR) to be used for the subsequent steady phase. The steady phase uses the provided CCA for transferring data from the file for a configured time length of Y seconds (Figure 7.5. The longer the steady state, the fewer and longer transfer episodes it will take for a file transfer, which in turn implies less frequent probing phases. We will further discuss the impact of steady phase length in the Section 7.6.

In discussing iPerfOPS we take iPerf-DT as a black box dependency, being a reliable and high-performance data transfer library used for implementing iPerfOPS. We will

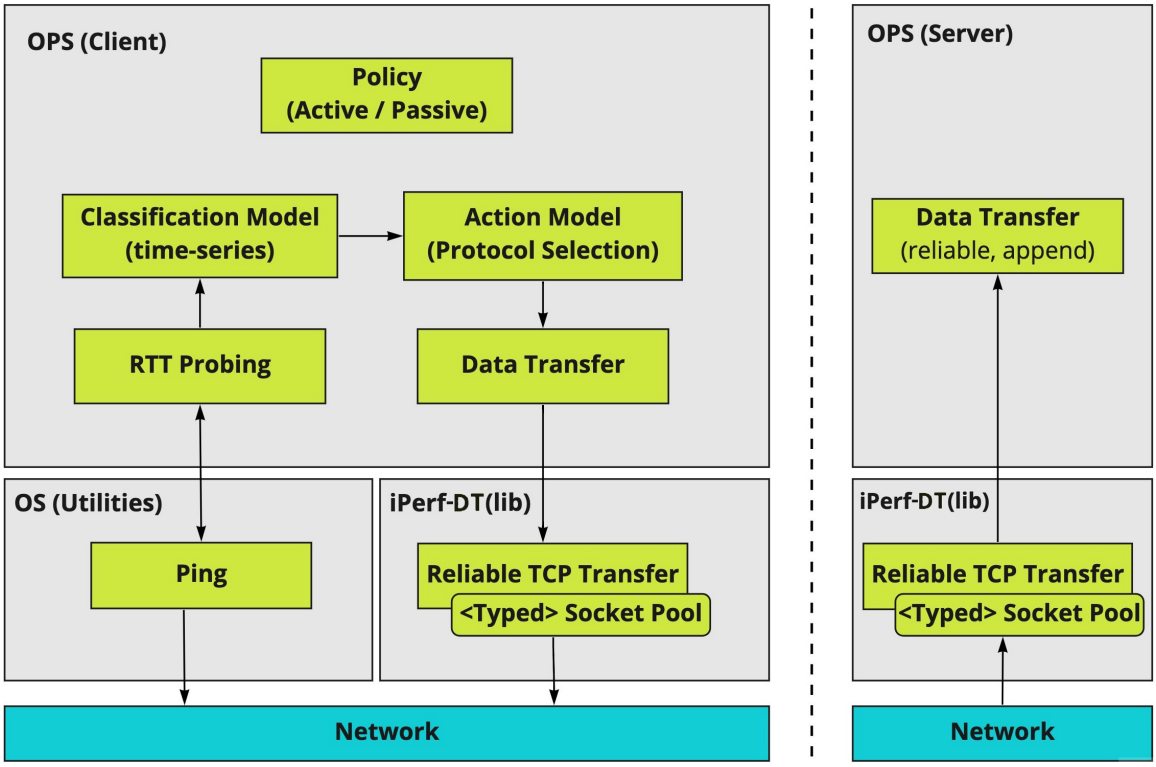


Figure 7.4: iPerfOPS Client/Server Architecture

next review the structure and functionality of the server and the client and their corresponding modules.

7.3.1 iPerfOPS Server

The server relies on iPerf-DT to accept incoming connections, and to reliably write the received data to the user-specified file. It operates to append-mode, always appending data to the end of the target file. Server design follows a simpler logic compared to the client. It is run by providing a port number as well as the file path for storing the received data. The server remains agnostic about the state of the data transfer whether it is in probing phase or steady phase of data transfer. The client handles the logic for different episodes of data transfer.

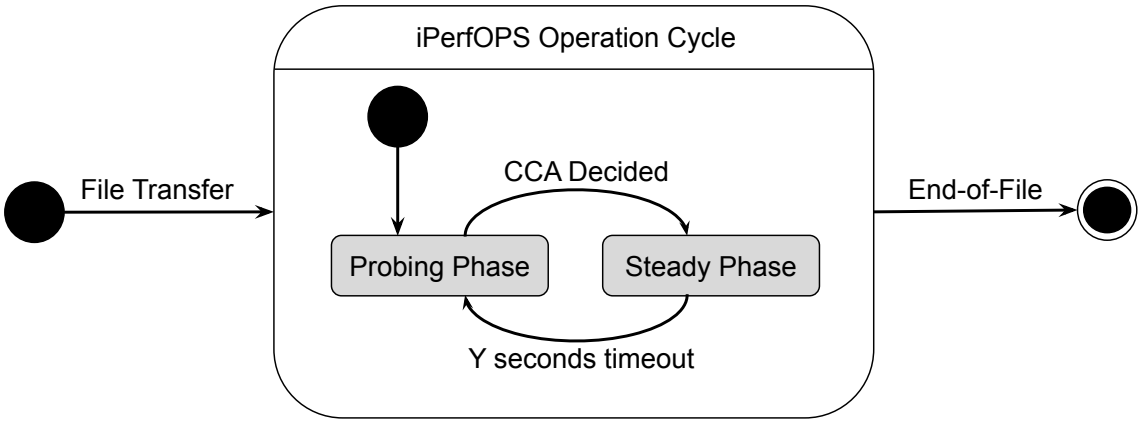


Figure 7.5: iPerfOPS state chart (coordinated an the client side)

7.3.2 iPerfOPS Client

The iPerfOPS client orchestrates the episodes of data transfer, interleaving steady phases of data transfer with periodic probing phases, as discussed in Chapter 6 (c.f., Figure 6.1). It relies on the *ping* utility provided by the operating system for probing RTT and generating RTT time-series (Step 1). The RTT time-series are then passed to the classification module (Step 2). The classification results are then passed to protocol selection module (Step 3) where a decision is made on the CCA to be used for the following steady phase of data transfer (Step 4). At the end of each data transfer segment, the last byte offset of the partially transferred file is passed to the next data transfer segment for continuing the data transfer.

While our current implementation contains a minimal viable version of functionality, the modular design of iPerfOPS allows us to independently extend and improve each module independently in the future. For example, the classification module, which now operates based on the 1-NN with DTW (Section 5.5), may be extended in the future to use more sophisticated classic or deep time-series classification models. Similarly, the action module which currently contains a simple logic may be extended with more advanced decision making models for picking a transfer protocol, considering more parameters such as network configuration, policies, and more.

```
# Server CLI Options
Usage: ./iperfops-server -p <server-port> [options]
 -h     (show help)
 -d     (enable debug mode)
 -f <destination-file-path>
```

(**a**)

```
# Client CLI Options
Usage: ./iperfops-client [options] [ -- <ops-command> ]
 -h     (show help)
 -d     (enable debug mode)
 -V     (enable Verbose mode)
 -s <server-host>
 -p <server-port>
 -c <cca>
 -k <k-for-knn>
 -S <steady-duration> (zero for one-time probing)
 -P <active-pattern>
 -l <log-file>[,<log-line-label>]
```

(**b**)

Figure 7.6: iPerfOPS CLI options for (**a**) server and (**b**) client. if `<ops-command>` is provided for the client it will be invoked in script mode, otherwise it will start the interactive prompt.

7.4 Usage: Interactive and Script Modes

We have implemented iPerfOPS with two possible operation modes to better fit different use-cases: the interactive mode and script mode. The CLI options for and usage guide for invoking iPerfOPS are provided in Figure 7.6 and a sample invocation commands (in script mode) are provided in Figure 7.7. On the server side the file path should be provided to store the received data, and optionally enabling debug mode to see execution details. On the client side, there is a wider range of possible options to configure different aspects of OPS data transfer. When invoking the client (Figure 7.6b), if `<ops-command>` is provided it will be invoked in script mode, otherwise it will start the interactive prompt.

Interactive mode. The interactive mode starts a REPL (read, evaluate, print

```
# start server
receiver$ ./iperfops-server -p 5555
```

(**a**)

```
# start client (sending file)
sender$ ./iperfops-client -s receiver -p 5555 -S 0 \
-l /path/to/logfile,b1c0-s0 -V --- send adaptive \
/path/to/source.file
```

(**b**)

Figure 7.7: Sample iPerfOPS invocation in script mode. (**a**) Starting server on port 5555. (**b**) Starting the client. As `<ops-command>` is provided, the client will run in script mode, running the provided command and terminating. The listed client command will only do a single data transfer episode (`-S 0`), meaning it will only probe once and then indefinitely running in steady phase using the decided CCA until the end of file transfer.

result, loop) prompt, allow the user exploring its features and trying different operations. A sample screenshot of the iPerfOPS interactive user-interface is provided in Figure 7.8. Interactive mode provides commands for setting options configurations, as well as commands for probing background traffic or transferring files. If the configurations (server IP, port, CCA, etc.) are provided through the initial invocation they will be set and available in the interactive prompt. Otherwise all the options are configurable through the iPerfOPS prompt too. It is worth noting that since the data transfer is coordinated through client, the interactive mode is only available on the client side.

Script mode. If `<ops-command>` is provided during the client invocation (Figure 7.6) the iPerfOPS will run in script mode. The provided invocation in Figure 7.7 includes a command, hence it will be executed in script mode. In script mode the provided OPS command is executed and then iPerfOPS is terminated immediately. In the script mode all the required configurations must be provided through CLI arguments, server host and port at the minimum. The script mode is ideal for use-cases where the exact task is known for data transfer, and it well supports automation through scripting.

```
**** iPerfOPS ****
use 'help' to learn about available commands.

OPS>> help
run <time-length>
set server <server-host>
set port <server-port>
set cca <tcp-cca>
set knn <K>
set pattern <pattern>
set steady <seconds>
help
list config
list cca
list pattern
probe <passive|active> <length>
send <adaptive|normal> <file-path>
quit

OPS>> list config
Server host: 192.168.55.39
Server port: 5555
CCA in use: default (cubic)
K for KNN: 1
Active Probing Pattern: c5-b5-c5-s5
Steady duration: 30

OPS>> send adaptive /dev/shm/source.file
```

Figure 7.8: iPerfOPS client interactive mode

7.5 OPS Runtime Overheads

In Section 6.2 we reviewed the conceptual overheads of the OPS cycle. In this section we discuss how those overheads extend to iPerfOPS. Furthermore, we review the system-level runtime overheads, and their potential impact on scalability, of our iPerfOPS implementation.

1. **Data-transfer steps.** For Step 4 we conduct traffic to the network using different CCAs. As discussed, the system and TCP 3-way handshake overheads are minimized through a socket pooling mechanism. However, when switching between CCAs (i.e., CUBIC and BBR) the internal state and counters of the TCP socket might be in a cold-start state (e.g., small `cwnd`). Nonetheless, this

CCA transition only happens every Y seconds at the most. We use Y values between 100 s and 360 s for evaluation (cf., Figure 7.10). As discussed before, there is trade-off in setting the Y parameter. For example, setting Y to 360 implies the best case of amortized overhead compared to lower Y values. At the same time, in the worst case it would imply up to 360 s of using a wrong CCA for steady data transfer (caused by inference error in classification or decision steps, or a change in the background CCA mixture).

Changes to the CCA in Step 1 have similar effects on the internal state of network sockets as discussed above for Step 4. Furthermore, the CCAs are switched more frequently during Step 1, corresponding to distinct segments of the active probing pattern. However, as we will discuss in Chapter 8, one direction for the future work is to simplify the probing phase to take less time and fewer switches between CCAs.

2. **Classification and Decision steps.** Each of the classification (i.e., Step 2) and decision-making (i.e., Step 3) steps only take a fraction of a second to execute, making them negligible compared to the steady data transfer. For Step 2, we currently utilize a Python implementation of 1-NN w/ DTW classifier. The efficiency of the classifier may be further improved by a new implementation in a compiled language such as C or C++. For Step 3, we use a 1-level decision tree which uses the classification of Step 2 to choose a CCA for Step 4. Considering Step 3 simple structure, it only takes a fraction of a second to execute.

3. **System-level overheads.** OPS incurs minimal CPU and memory overhead. Data-transfer tasks during Step 1 (probing) and Step 4 (steady transfer) are exactly the same case as writing data to a socket in typical network applications. In Step 2 our current 1-NN w/ DTW incurs minimal CPU overhead for the classification inference. In Step 3 the 1-level decision tree is a simple `if` statement. The computation for Steps 2 and 3 take a fraction of a second and

is negligible compared to the 20 seconds of Step 1 and the Y seconds of Step 4. If more complex classification and decision models are to be used in the future, additional software and hardware optimizations can be considered.

After choosing a CCA to use, Step 4 of OPS transfers data using the standard network socket until the Y seconds expires. During the steady data transfer, the only overheads are the usual user-space to kernel overheads associated with network sockets.

7.6 End-to-End Evaluation

In this section we discuss data-transfer experiments and report on the end-to-end performance results for iPerfOPS.

7.6.1 Experimentation Setup

We use the same testbed as presented in Chapter 3, and the same test scenarios as discussed in Section 6.5.

Background traffic. In our previous OPS evaluations (Section 6.5) we have studied the OPS performance in the presence of a constant background pattern, either CUBIC, BBR, or a mixed of both. To further study the adaptive end-to-end performance of iPerfOPS, we add new test scenarios where the background traffic alternates between CUBIC and CCA in regular time intervals. We use `ALT-X` to identify those configurations, where `X` represent the time interval in seconds that one CCA runs before toggling the CCA.

All the reported results in this chapter present data-transfer tasks, transferring 14 GB data files. The average throughput over five runs are provided, and the error bars present standard deviation.

Probing Patterns. While both passive and active probing policies are supported by iPerfOPS, in this section we only present and discuss constructive active probing

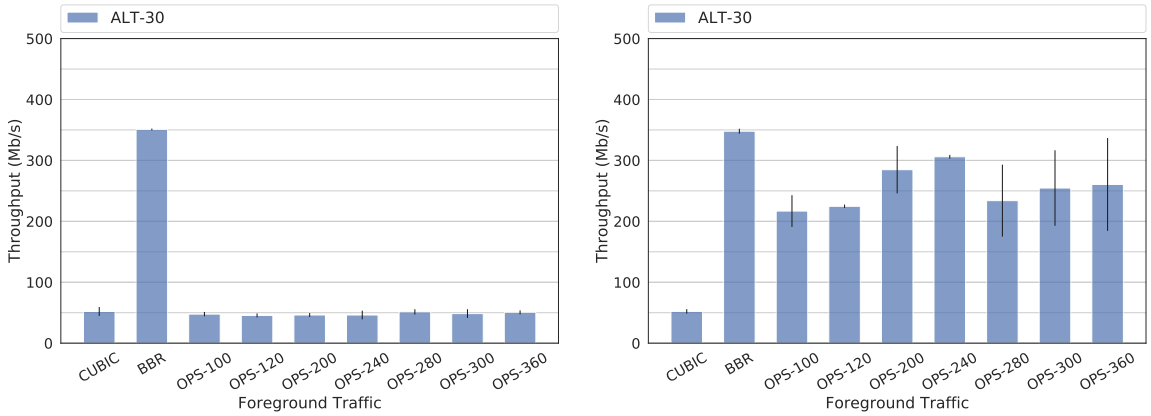


Figure 7.9: iPerfOPS Performance in presence of dynamic (alternating) background traffic. (**a**) Active probing pattern: C5-B5-C5-S5 (non-effective) (**b**) Active probing pattern: S5-C5-S5-B5 (effective). All results are average over five runs, transferring 14 GB random file. The results correspond to ALT-30 background pattern (single constant TCP stream of background traffic, toggling between CUBIC and CCA every 30 seconds). OPS-Y represents foreground traffic of probing network followed by sending data steady for Y seconds and repeat.

policy as it outperforms passive probing policy in terms of higher classification accuracy (Section 5.6). So far we have introduced the C5-B5-C5-S5 as the only active probing pattern available during our evaluations in Section 6.5. However, in order to better study and identify the importance and capability of distinct probing patterns to drive classification and decision making, in this section we add a second probing pattern to better identify the impact of silence periods in the probing patterns. The new probing pattern is S5-C5-S5-B5, where we have 5 seconds silence period before sending 5 seconds of CUBIC or BBR traffic as part of the probing pattern. With this new pattern we aim to better identify the RTT pattern of background traffic without our probing interference. The performance results will be provided in the following section.

7.6.2 Results: The Impact of Distinct Probing Pattern

Figure 7.9, especially part (b), shows the throughput improvement gained by using iPerfOPS. In (b), all of the (dynamic) OPS-[100,360] bars have much higher through-

put (all between 210 and 305 Mb/s) than the CUBIC bar (about 50 Mb/s). Given the 500 Mb/s bottleneck bandwidth, the 250 Mb/s level represents a fair share of the network. At 50 Mb/s, the CUBIC bar represents much less than a fair share of the bandwidth, and is the result of a static choice of CUBIC for the foreground CCA.

The BBR bar (almost 350 Mb/s) shows how unfair BBR is against CUBIC, in which the ALT-30 background traffic alternates between CUBIC and CCA every 30 seconds.

Figure 7.9(b) shows that OPS can detect, classify, and then select an appropriate CCA for the foreground traffic that is effective in maximizing throughput. Note that the OPS task is non-trivial because the background traffic dynamically changes CCA (between CUBIC and CCA) every 30 seconds. Also, an important practical, implementation detail is that a S5-C5-S5-B5 active probing pattern (part (b)) is required for OPS to be effective.

In contrast, the probing pattern C5-B5-C5-S5 (Figures 7.9(a)) is ineffective in improving performance, presenting virtually identical performance as fixed CUBIC, the probing pattern S5-C5-S5-B5 (Figures 7.9b) is effectively improving the data-transfer performance by several orders of magnitude compared to fixed CUBIC CCA, while treating the cross traffic more fairly compared to the aggressive BBR.

The distinct results between the two different active probing patterns shows the effect, and the importance, of the probing pattern in use by the OPS for probing phase. The probing pattern C5-B5-C5-S5 which was effective in improving performance in our proof of concept experiments (Chapter 6), proves ineffective in our end-to-end evaluation of iPerfOPS. In contrast, the pattern S5-C5-S5-B5 improves the ability of generated RTT time-series in identifying background mixture, hence improved throughput performance compared to CUBIC and better fairness compared to BBR. We may attribute, among other reasons, the improved performance for S5-C5-S5-B5 to the added silence periods prior to conducting traffic using different CCAs. This enables the generated RTT time-series to better manifest the unique patterns for

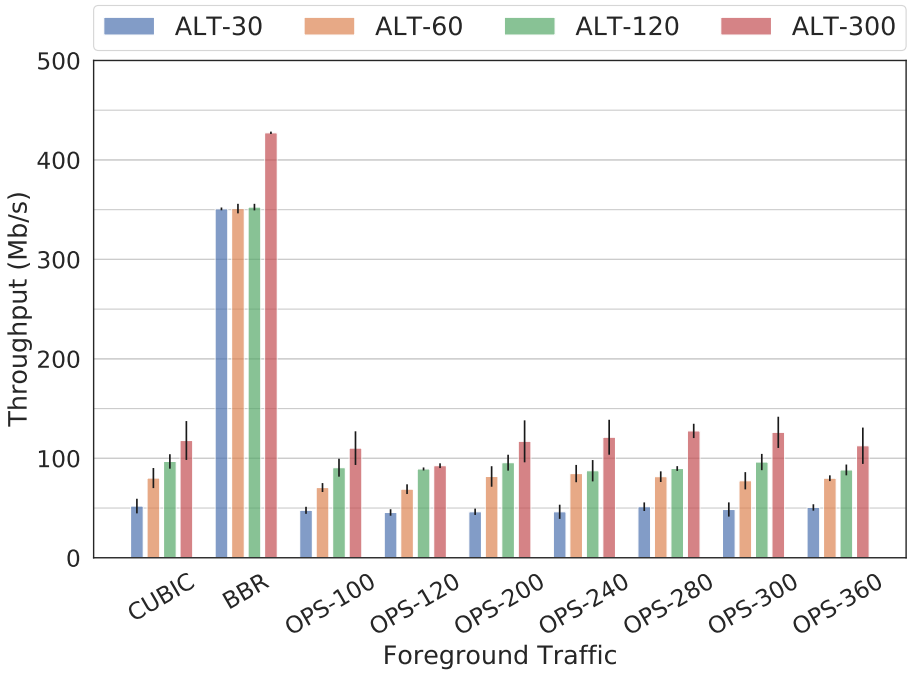
different types of background traffic.

Furthermore, increasing the steady state length, improves the performance for the effective pattern S5-C5-S5-B5 (Figure 7.9b). The longer steady state periods mean less silence periods in total,hence better throughput performance. However, there is a trade-off when setting the steady state length for OPS. The shorter the steady state, the more frequent probing phases will be, and OPS sooner gets the opportunity to identify the changes in the background traffic patterns and adjust the CCA accordingly. In contrast, the longer the steady state, the less time is spend on probing phase, hence less time in silence periods and overhead of segment-wise active probing patterns. Hence the performance could improve if the background traffic is not changing too often.

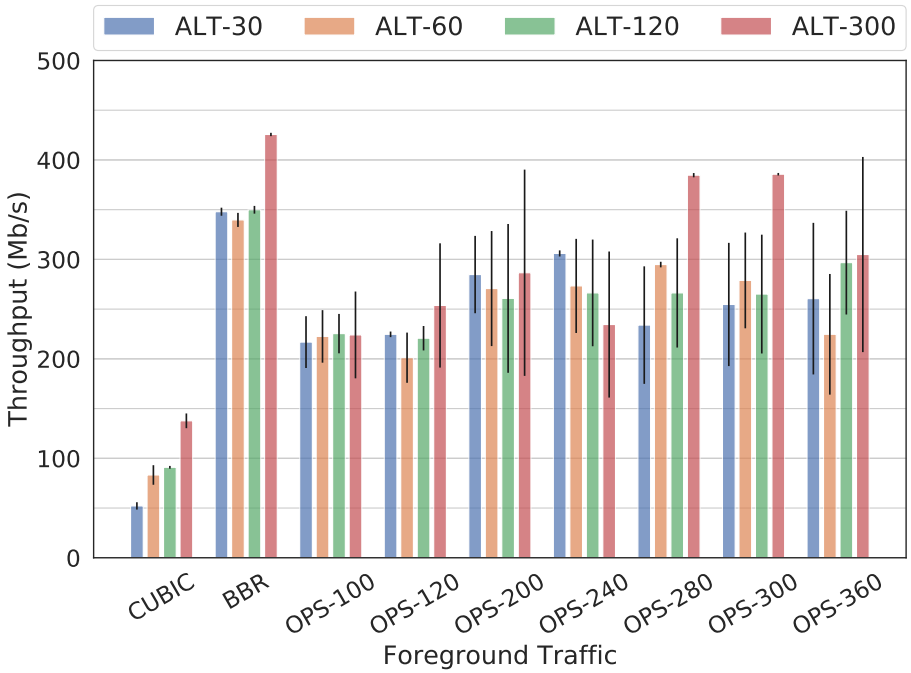
One general rule of thumb is to set steady state to longer values if the frequency of the change in background traffic is low. For example on private networks or institution bandwidths where the pattern and type of bandwidth usage is more predictable. However, for public shared bandwidths or networks with a larger number of users the shorter steady state could help OPS to better adapt to frequent changes to the background traffic.

To further study the effect of the frequency of alternating background CCA, we have conducted the same experiment for different configurations for X in ALT-X background traffic pattern. The results for X values of 30, 60, 120, and 300 are provided in Figure 7.10. The throughput performance is calculated in presence of fixed and alternating background traffic. All results are average over five runs, transferring 14 GB data files containing random binary data. ALT-X represents background traffic of alternating CUBIC and BBR every X seconds. OPS-Y represents foreground traffic of probing network followed by sending data steady for Y seconds and repeat.

Figure 7.10(a) presents results in probing pattern C5-B5-C5-S5 in use, while Figure 7.10(b) presents results for probing pattern S5-C5-S5-B5. While there are throughput variations based on the standard deviation (error bars), the overall performance



(**a**)



(**b**)

Figure 7.10: iPerfOPS Performance in presence of dynamic (alternating) background traffic. (**a**) Active probing pattern: C5-B5-C5-S5 (non-effective) (**b**) Active probing pattern: S5-C5-S5-B5 (effective). All results are average over five runs, transferring 14 GB random file. ALT-X represents background traffic of alternating CUBIC and BBR every X seconds. OPS-Y represents foreground traffic of probing network followed by sending data steady for Y seconds and repeat.

trend remains the same across different configurations of ALT-X for the background traffic.

Once again (as with Figure 7.9), Figure 7.10(b) shows that OPS is effective in achieving fair throughput against dynamically changing background traffic, this time with greater interval variability. And, once again, the probing pattern of S5-C5-S5-B5 is key to enabling OPS to properly detect, classify, and perform OPS.

While in current OPS design the steady state length is specified as a user-provided configuration, one possible improvement for the future is to auto-adjust the steady state length. So when OPS identifies changes in background traffic in subsequent probings, it should decrease the steady state length to better capture dynamic patterns of background traffic. But if subsequent probing phases result in the same background traffic class, this implies less dynamic background patterns and hence we should increase the steady state length.

7.7 Concluding Remarks

Prior to this chapter we have been discussing motivation and the effectiveness of background classification through proof-of-concept experiments. In this chapter, we have demonstrated the effectiveness of OPS for data transfer in a real tool and under end-to-end empirical evaluations. We have discussed the design, implementation, and evaluation of iPerfOPS, an implementation of the OPS strategy. We show that iPerfOPS provides a tangible performance improvement for end-to-end data transfer when configured with appropriate probing patterns. Our tool extends the functionality of the original iPerf with the OPS functionality, but we also add both reliable and efficient data transfers.

Our empirical evaluation of iPerfOPS shows the effectiveness of OPS. It also reveals the importance of the probing pattern when using (constructive) active probing. While selecting an appropriate probing pattern (i.e., S5-C5-S5-B5) could result in a bandwidth utilization close to the fair (equal) sharing, other patterns (e.g., C5-B5-

C5-S5) could result in sub-optimal performance.

One future direction for this study is to further explore the impact of different probing patterns on the classification and end-to-end data transfer performance. The distinct performance of the two probing patterns in this chapter reveals the opportunity to further improve the end-to-end performance by crafting new combinations of protocols, and silence periods, for the probing pattern. Furthermore, our current study included up to three data streams on the network (one foreground and two background). Another future direction is to study a larger number of streams on the network, possibly with different schedules for the stream inter-arrival timing.

Chapter 8

Concluding Remarks

We have introduced OPS, optimization through protocol selection, as a technique to improve bulk-data transfer on shared WANs. Instead of just tuning the parameters of a network protocol, our empirical results show that the selection of the CCA and the network protocol itself can result in up to four times higher throughput in some key cases (e.g., Figure 6.9b, foreground throughput for Adaptive (i.e., OPS) versus Fixed:CUBIC).

It is a challenging task to optimize a data-transfer task for both throughput and fairness. If throughput is the only deciding metric, we may use a greedy approach to probe each available CCA and to pick the one with maximum throughput for steady transfer. However, a greedy approach can hurt fairness. Unlike throughput, it is not feasible to measure fairness from a local end-point perspective. OPS enables optimization for throughput-and-fairness (i.e., both throughput and fairness) via its multi-step process for probing, discovering, and protocol selection, followed by the steady data transfer.

OPS for the foreground traffic depends on knowledge about the network protocols used by the background traffic (i.e., other users). Unfortunately, global knowledge can be difficult to obtain in a dynamic distributed system like a shared WAN. Therefore, we trained ML models for classifying the local RTT time-series data, gathered using *passive probing* and *active probing*, to recognize the mix of network protocols in the

background, with an accuracy of up to 95% (e.g., Figure 5.8a, w = 50 s).

Finally, we built an end-to-end data transfer tool based on OPS strategy, iPerfOPS, implemented as a substantially modified version of the well-known iPerf tool. iPerfOPS operates in two interactive and script modes, providing flexibility to be used in automated data transfer pipelines as well as for interactive and exploratory networking probing and data transfer. The evaluation results for iPerfOPS shows that, depending on the use of appropriate probing pattern, iPerfOPS offers improved cross-stream fairness, reaching close to the fair (equal) sharing.

In the following sections we review some of important observations in our study, as well as the key lessons learned. We wrap up this research by discussing the limitations of this work, followed by proposing a few directions for the future work.

8.1 Throughput vs. Fairness on Shared Networks

Decades of research and tuning of network protocols (e.g., TCP) and congestion control algorithms (e.g., CUBIC, BBR) have been in the service of fair sharing with high performance of foreground and background network traffic. Our empirical results show that sharing is not fair under some combinations of use-cases and protocols. In particular, due to specific design decisions, various TCP CCAs could be unfair to each other. An example of this unfair sharing includes BBR streams' unfair behavior against CUBIC streams. Furthermore, the interaction of TCP and UDP traffic can lead to unfair sharing and low performance. Tools that have been built on top of TCP (e.g., GridFTP) and UDP (e.g., UDT, QUIC) to address various performance problems, can aggravate the sharing and performance problem for other traffic.

As a result of our experiments, here we provide two main observations regarding throughput and fairness:

1. **Throughput:** A poor choice of foreground protocol can reduce throughput by up to 80% (Figure 4.1a, GridFTP reduced throughput versus UDT). Different

combinations of protocols on shared networks have different dynamic behaviors. One might naturally assume that all network protocols designed for shared networks interact both reasonably and fairly. However, that is not necessarily true.

The mixture of CUBIC and BBR, both based on TCP, can lead to poor throughput for CUBIC streams competing against BBR (e.g., Figures 4.5 and 4.6). Using TCP for the foreground, with UDP as the background, results in poor foreground throughput (e.g., Figures 4.1 and 4.2). For example, our empirical results show that if the competing background traffic is UDP-based, then UDT (which is UDP-based itself) can actually have significantly higher throughput (Figure 4.1a, BG-SQ-UDP1 and BG-SQ-UDP2; Figure 4.1b all except for No-BG).

2. **Fairness:** Jain fairness (Section 2.4) varies by up to 0.35 (Table 4.3, between 0.64 to 0.99) representing poor fairness for some combinations of tools and protocols. On the one hand, using an aggressive tool would highly degrade the performance of the cross traffic (e.g., Jain index of 0.68 in Table 4.2, and 0.64 in Table 4.3 for GridFTP). On the other hand, depending on the type of background traffic, the performance of a traffic stream would be impacted (e.g., iPerf foreground in Tables 4.2 and 4.3) . Of course, using GridFTP is a voluntary policy choice, and we quantified the impact of such tools on fairness. GridFTP can have a quantifiably large impact on other network traffic and users. Anecdotally, people know that GridFTP can use so much network bandwidth that other workloads are affected. However, it is important to quantify that effect and the large dynamic range of that effect (e.g., a range of 22 Mb/s, from a peak of 30 Mb/s to a valley of 8 Mb/s of NFS throughput, due to GridFTP; discussed above and in Section 4.3). Similarly, in a purely TCP-based mixture of traffic, TCP BBR has a large impact on other TCP CCAs, TCP CUBIC in

particular (Figures 4.5 and 4.6).

8.2 Fixed vs. Adaptive Protocol Selection

While global knowledge about the mixture of traffic on a shared network is hard to obtain, our ML-based classifier with passive probing is able, with up to 85% accuracy, to identify different mixtures of TCP BBR and TCP CUBIC background traffic (Figure 5.5). Furthermore, active probing (as compared to passive probing) improves ML-based classifiers to better distinguish between different mixtures of TCP-based background traffic, with up to 95% accuracy (Figure 5.7).

Without OPS, selecting a fixed protocol or tool might yield sub-optimal performance on different metrics of interest (throughput, fairness, etc.). In contrast, through investigating the background workload (e.g., using passive or active probing techniques) and adaptively selecting appropriate protocol (i.e., OPS) we could gain overall performance improvement (Figure 6.9).

While the particular CCAs we used in this research for OPS evaluation might get evolved, or deprecated in favor of others, the OPS strategy for adaptive protocol selection (as opposed to using a fixed protocol) would still remain relevant. It could significantly contribute to the overall performance of data-transfer tasks at any time, assuming appropriate classification model and sufficiently-sized protocol catalog are available.

8.3 Discussion

Throughout this research we have studied and discussed a few aspects of high-performance data transfer on bandwidth-sharing networks. In this section we reiterate our key findings and provide a summary of lessons learned from this research:

1. **One protocol's throughput might significantly degrade depending on the background mix of protocols.** Despite the conventional wisdom about

using particular protocols, such as GridFTP, for achieving the best performance, each protocol might differently react to distinct mixtures of background traffic on a shared network. Hence, one tool or protocol that offers great performance in one use-case might suffer significant under-performance in some other scenarios. For example, GridFTP is not always the highest performing tool for bulk-data transfers on *shared* networks. Our results show that if there is a significant amount of UDP background traffic, especially bursty UDP traffic, GridFTP is negatively affected, dropping to 10% of the theoretical maximum (Figure 4.1).

2. **Fairness should to be accounted for as a trade-off with the aggressiveness of data-transfer tools.** Unless bandwidth reservation capabilities are present for allocating dedicated bandwidth for a data-transfer task, the more aggressive the protocol gets, the larger the impact on the other network users would become. For example, the use of GridFTP with a large number of parallel streams would drastically impact the users accessing their data on an NFS-mounted network drive(Figure 4.8). In a collaborative network environment where groups of users belong to the same institution, the wise choice of protocol set for a shared network could help to improve the overall performance (per user and aggregate throughput) while allowing appropriate bandwidth share for the rest of the traffic (Fairness), resulting in a better performance equilibrium.

   Furthermore, in a purely TCP-based mixture of traffic, the partial upgrade to a new TCP CCA while other nodes still use older CCAs could drastically degrade the user experience for a group of users. For example, TCP BBR has a large impact on other TCP flavors, TCP CUBIC in particular (Figures 4.5 and 4.6).

3. **Periodic RTT probing could expose the nature of the traffic on an end-to-end path.** The internal logic of how a network protocol operates is not available to the outside world. However, as a protocol pushes traffic to the

network or decides to pull off for any reason it directly impacts the size of data queued at the routers. Moreover, the queue build up at the routers is directly impacting the observed end-to-end latency, measured by RTT. So, assuming we know the baseline latency of the network, probing the RTT at any given time could provide insights about how congested is the network, representing the instantaneous network load. While there is no clear way to capture the baseline latency, when we regularly measure the RTT it reveals the latency variation, potentially representing the temporal patterns of the network load over time (e.g., Figure 2.5). In contrast, the particular state transitioning and behavior model of a network protocol (or a mix of protocols) could result in particular patterns of latency. Hence, the periodic RTT probing, forming RTT time-series, providing the opportunity to capture the unique patterns of particular mixtures of protocols, exposing the nature of the traffic to be analyzed, through machine learning or otherwise (e.g., Figure 5.3).

4. **Adaptive protocol selection aids optimization for throughput and fairness on shared networks.** When transferring data over a shared network, it is beneficial, or even necessary, to investigate the mix of background workload. Distinct settings of background workload could manifest different levels of impact on different protocols. Hence selecting a fixed protocol or tool might yield sub-optimal performance on different metrics of interest (throughput, fairness, etc.). In contrast, through investigating the background workload (e.g., using passive or active probing techniques) and adaptively selecting appropriate protocol (e.g., using OPS) we could gain overall performance improvement (Figure 7.10).

8.4 Limitations and Future Work

Limitations. In this research study we strove to build models sufficiently sized to validating our hypotheses and verifying our design decisions. Yet, there are a few limitations of this research which we would like to call out in order to inform the direction for possible improvement and extension in the future work:

1. In applying ML for background recognition, we have limited the scope to TCP CUBIC and TCP BBR only. However, more realistically there are more diverse ranges of CCAs running on a shared network, either TCP-based or UDP-based.

2. Our study is limited to a maximum of two streams of traffic in the background, while the streams on a real network may count up to tens or hundreds of stream. With larger number of streams it could tend to be harder to recognize the mixture, particularly identifying the exact mixture of protocols.

3. In building our classification models we have gathered training data from a single network with fixed bandwidth and RTT properties. This could be limiting the capability of the trained model if deployed for a networks with different properties. We have not verified this limitation as we have used the same network for evaluating the trained models.

Future Work. Informed by those limitations, there are a few areas of improvement which we recommend to be considered for the future work:

1. **Diversifying the training data for a wider range of networks.** The training data we used for building classification models were limited to a fixed network bandwidth and RTT configuration. This could limit our model performance on the networks with different bandwidth and RTT properties. CCAs tend to behave differently as the BDP on network path varies, hence manifesting distinct signatures of RTT time-series. Therefore, one potential direction

to expand this study is to gather lager training datasets collected over a wider range of network BDP configurations (i.e., bottleneck bandwidth and RTT), combined with a parameter sweep for the buffer size at the bottleneck router.

2. **Enhancing ML models to deal with larger mixtures of streams and CCAs.** The classification models in this research were designed to predict the exact mixture of the background traffic of up to two streams, of either TCP CUBIC or TCP CCA. While this was sufficient to prove the effectiveness of OPS mechanism, on real shared networks there could exist a larger number of streams sharing the bandwidth.

   Moreover, even though TCP CUBIC and TCP BBR are among the most popular TCP CCAs on the Internet, the emerge of newer TCP CCAs (e.g., Copa) or even UDP-based protocols (e.g., QUIC) increases the possibility of facing those protocols on the networks. Hence, another direction for future work consists of gathering data and training models for traffic mixtures with a larger number of streams, and of better diversified protocols in the mixture, possibly with different schedules for the stream inter-arrival timing. When moving towards more complex mixtures of background traffic, the target classes might be adjusted to represent the type of protocols in use (e.g., proactive vs. reactive CCAs) rather than predicting the exact streams and CCAs present on the network. Furthermore, more advanced classification models (e.g., Deep networks, LSTM, ResNet) may be used to better generalize for unseen RTT signatures.

   Additionally, inspired by the premise of reinforcement learning, the classification and decision process of OPS may be further optimized. A feedback loop may be established from Step 4 to Step 2 and to Step 3 for sharing the observed performance of the decided protocol. Then, using an appropriate reward function we could incrementally improve the classification and decision-making performance. Since the sender end-point could only measure throughput (and

not cross-stream fairness), additional caution should be taken to ensure the decision-making process is not gradually evolving towards throughput-only objective (as opposed to the desired objective of throughput-and-fairness).

3. **Alternate design choices for OPS.** There are some interesting variations of the basic 4-step OPS cycle presented in this research.

   First, the Step 1 (Background Probing) probing pattern does not need to be a simple sequence of known CCAs, nor does it need to be fixed at 20 seconds. In fact, the CCAs used in the probing pattern do not necessarily have to be the same CCAs as in current use. Moreover, one might consider a new or modified CCA (not just vanilla CUBIC or BBR) designed specifically to improve classification accuracy. Such a special-purpose CCA could still constructively transfer data during Step 1, while reducing the time and overhead of the step, and lead to a better classification in Step 2.

   Although we have not yet experimented with the idea, it is future work to try a progressive probing pattern, with early termination. If a classifier (i.e., overlap Steps 1 and 2) already has high confidence in the class of the background CCA, the pattern can be stopped before all four segments. Assuming 5 seconds per segment, Step 1 could be abbreviated to just 5, 10 or 15 seconds. Even the 5 second parameter itself is not necessarily fixed, other than in our experiments.

   Also, classification accuracy in a shorter time-frame might be improved by using more features than just an RTT time-series. Other metrics, such as throughput, latency, or packet loss count can be considered.

   Second, the Y parameter in Step 4 (Steady Transfer) does not have to be a constant. Similar to how congestion windows grow if there is a steady state, the value of Y can vary and grow larger if the background CCA is observed to be unchanging. But, if the background CCA is changing more frequently, Y can be reduced, thus trading-off the amortization benefits of a larger Y for the

dynamic responsiveness of a smaller Y.

Third, especially when using multiple features, other classifier algorithms might be re-considered in the future. As discussed previously, we chose 1-NN w/ DTW for Step 2 because it met our needs. We experimented with MLPs and other options (Chapter 5) and chose the simplest algorithm that worked. However, a progressive probing pattern, multiple features, and a special-purpose CCA for classification might benefit from a more sophisticated machine-learning strategy in our future work.

Bibliography

[1] H. Anvari and P. Lu, "Large transfers for data analytics on shared wide-area networks," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16, Como, Italy: Association for Computing Machinery, 2016, 418–423, ISBN: 9781450341288. DOI: 10.1145/2903150.2911718. [Online]. Available: https://doi.org/10.1145/2903150.2911718.

[2] H. Anvari and P. Lu, "The impact of large-data transfers in shared wide-area networks: An empirical study," *Procedia Computer Science*, vol. 108, pp. 1702–1711, 2017, International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.05.211. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050917308049.

[3] H. Anvari, J. Huard, and P. Lu, "Machine-learned classifiers for protocol selection on a shared network," in *Machine Learning for Networking*, É. Renault, P. Mühlethaler, and S. Boumerdassi, Eds., Cham: Springer International Publishing, 2019, pp. 98–116, ISBN: 978-3-030-19945-6.

[4] H. Anvari and P. Lu, "Learning mixed traffic signatures in shared networks," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya *et al.*, Eds., Cham: Springer International Publishing, 2020, pp. 524–537.

[5] H. Anvari and P. Lu, "Active probing for improved machine-learned recognition of network traffic," in *Machine Learning for Networking*, É. Renault, S. Boumerdassi, and P. Mühlethaler, Eds., Cham: Springer International Publishing, 2021, pp. 122–140.

[6] H. Anvari and P. Lu, "Machine-learned recognition of network traffic for optimization through protocol selection," *Computers*, vol. 10, no. 6, 2021, ISSN: 2073-431X. DOI: 10.3390/computers10060076. [Online]. Available: https://www.mdpi.com/2073-431X/10/6/76.

[7] H. Anvari and P. Lu, "iPerfOPS: A tool for machine learning-based optimization through protocol selection," in *Machine Learning for Networking*, Cham: Springer International Publishing, 2022.

[8] *Iperf bug fix (pr#1112): Missing features in the library api*, https://github.com/esnet/iperf/pull/1112.

[9] *Iperf bug fix (pr#1113): Incorrect termination test in bytes/blocks mode*, https://github.com/esnet/iperf/pull/1113.

[10] *Iperf bug fix (pr#1114): Fix and optimize test termination condition check*, https://github.com/esnet/iperf/pull/1114.

[11] *Iperf bug fix (pr#1115): Incorrect data transfer in file mode*, https://github.com/esnet/iperf/pull/1115.

[12] *Iperf bug fix (pr#1116): Auto adjusting test-end condition in file mode*, https://github.com/esnet/iperf/pull/1116.

[13] *Iperf+: Extending iperf with reliable file transfer and network socket reuse*, https://github.com/hanvari/iperf-reliable.

[14] *Worldwide lhc computing grid*, http://home.cern/about/computing/worldwide-lhc-computing-grid, 2021-06-08.

[15] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul, "Efficient data transfer protocols for big data," in *E-Science, 2012 IEEE 8th International Conference on*, IEEE, 2012, pp. 1–9.

[16] M. G. Labate, M. Waterson, G. Swart, M. Bowen, and P. Dewdney, "The square kilometre array observatory," in *2019 IEEE International Symposium on Antennas and Propagation and USNC-URSI Radio Science Meeting*, 2019, pp. 391–392. DOI: 10.1109/APUSNCURSINRSM.2019.8888528.

[17] W. Allcock *et al.*, "The globus striped gridftp framework and server," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05, Washington, DC, USA: IEEE Computer Society, 2005, pp. 54–, ISBN: 1-59593-061-2. DOI: 10.1109/SC.2005.72. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.72.

[18] C. Barakat, E. Altman, and W. Dabbous, "On tcp performance in a heterogeneous network: A survey," *Comm. Mag.*, vol. 38, no. 1, pp. 40–46, Jan. 2000, ISSN: 0163-6804. DOI: 10.1109/35.815451. [Online]. Available: http://dx.doi.org/10.1109/35.815451.

[19] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, "Host-to-host congestion control for tcp," *Communications Surveys Tutorials, IEEE*, vol. 12, no. 3, pp. 304–342, 2010, ISSN: 1553-877X. DOI: 10.1109/SURV.2010.042710.00114.

[20] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008, ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105.

[21] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *ACM Queue*, vol. 14, September-October, pp. 20 –53, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=3022184.

[22] Y. Zhang, L. Cui, and F. P. Tso, "Modest bbr: Enabling better fairness for bbr congestion control," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00 646–00 651. DOI: 10.1109/ISCC.2018.8538521.

[23] S. Ma, J. Jiang, W. Wang, and B. Li, *Fairness of congestion-based congestion control: Experimental evaluation and analysis*, 2017. DOI: 10.48550/ARXIV.1706.09115. [Online]. Available: https://arxiv.org/abs/1706.09115.

[24] W. Pan, H. Tan, X. Li, and X. Li, "Improved rtt fairness of bbr congestion control algorithm based on adaptive congestion window," *Electronics*, vol. 10, no. 5, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10050615. [Online]. Available: https://www.mdpi.com/2079-9292/10/5/615.

[25] F. Yang, Q. Wu, Z. Li, Y. Liu, G. Pau, and G. Xie, "Bbrv2+: Towards balancing aggressiveness and fairness with delay-based bandwidth probing," *Computer Networks*, vol. 206, p. 108 789, 2022, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2022.108789. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622000226.

[26] Y.-J. Song, G.-H. Kim, and Y.-Z. Cho, "Bbr-cws: Improving the inter-protocol fairness of bbr," *Electronics*, vol. 9, no. 5, 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9050862. [Online]. Available: https://www.mdpi.com/2079-9292/9/5/862.

[27] N. Cardwell *et al.*, "Bbrv2: A model-based congestion control," in *Presentation in ICCRG at IETF 104th meeting*, 2019.

[28] *Tcp bbr v2 alpha/preview release*, https://github.com/google/bbr/blob/v2alpha/README.md, 2019.

[29] Y. Gu and R. L. Grossman, "Udt: Udp-based data transfer for high-speed wide area networks," *Computer Networks*, vol. 51, no. 7, pp. 1777 –1799, 2007, Protocols for Fast, Long-Distance Networks, ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2006.11.009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128606003057.

[30] J. Kurose and K. Ross, *Computer Networking: A Top-down Approach* (Always learning). Pearson, 2013, ISBN: 9780132856201. [Online]. Available: https://books.google.ca/books?id=fWg5pwAACAAJ.

[31] Z. Wang, X. Zeng, X. Liu, M. Xu, Y. Wen, and L. Chen, "Tcp congestion control algorithm for heterogeneous internet," *Journal of Network and Computer Applications*, vol. 68, pp. 56 –64, 2016, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.03.018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804516300327.

[32] S. Yu, N. Brownlee, and A. Mahanti, "Comparative performance analysis of high-speed transfer protocols for big data," in *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, 2013, pp. 292–295. DOI: 10.1109/LCN.2013.6761252.

[33] T. J. Hacker, B. D. Athey, and B. Noble, "The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, ser. IPDPS '02, Washington, DC, USA: IEEE Computer Society, 2002, pp. 314–, ISBN: 0-7695-1573-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=645610.661894.

[34] E. Altman, D. Barman, B. Tuffin, and M. Vojnovic, "Parallel tcp sockets: Simple model, throughput and validation," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006, pp. 1–12. DOI: 10.1109/INFOCOM.2006.104.

[35] N. Eghbal and P Lu, "A parallel data stream layer for large data workloads on wans," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications*, IEEE, 2020, pp. 1–9.

[36] C. Paasch and O. Bonaventure, "Multipath tcp," *Commun. ACM*, vol. 57, no. 4, pp. 51–57, Apr. 2014, ISSN: 0001-0782. DOI: 10.1145/2578901. [Online]. Available: http://doi.acm.org/10.1145/2578901.

[37] S. Habib, J. Qadir, A. Ali, D. Habib, M. Li, and A. Sathiaseelan, "The past, present, and future of transport-layer multipath," *Journal of Network and Computer Applications*, vol. 75, pp. 236 –258, 2016, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.09.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804516302028.

[38] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-27, Feb. 2020, Work in Progress, 174 pp. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27.

[39] E. He, J. Leigh, O. Yu, and T. A. DeFanti, "Reliable blast udp: Predictable high performance bulk data transfer," in *Proceedings of the IEEE International Conference on Cluster Computing*, ser. CLUSTER '02, Washington, DC, USA: IEEE Computer Society, 2002, pp. 317–, ISBN: 0-7695-1745-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=792762.793299.

[40] M. A. Alrshah, M. Othman, B. Ali, and Z. M. Hanapi, "Comparative study of high-speed Linux TCP variants over high-BDP networks," *Journal of Network and Computer Applications*, vol. 43, pp. 66–75, 2014.

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

[42] M. Hock, F. Neumeister, M. Zitterbart, and R. Bless, "TCP LoLa: Congestion control for low latencies and high throughput," in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, 2017, pp. 215–218. DOI: 10.1109/LCN.2017.42.

[43] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, Apr. 2018, pp. 329–342, ISBN: 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/arun.

[44] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: ACM, 2013, pp. 3–14, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486019. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486019.

[45] H. H. Liu *et al.*, "Efficiently delivering online services over integrated infrastructure," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA: USENIX Association, 2016, pp. 77–90, ISBN: 978-1-931971-29-4. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/liu.

[46] C. Guok, D. Robertson, M. Thompson, J. Lee, B. Tierney, and W. Johnston, "Intra and interdomain circuit provisioning using the oscars reservation system," in *Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference on*, 2006, pp. 1–8. DOI: 10.1109/BROADNETS.2006.4374316.

[47] L. Ramakrishnan, C. Guok, K. Jackson, E. Kissel, D. M. Swany, and D. Agarwal, "On-demand overlay networks for large scientific data transfers," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 359–367, ISBN: 978-0-7695-4039-9. DOI: 10.1109/CCGRID.2010.82. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2010.82.

[48] C. Hong *et al.*, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: ACM, 2013, pp. 15–26, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486012. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012.

[49] H. Jiang and C. Dovrolis, "Why is the internet traffic bursty in short time scales?" In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '05, Banff, Alberta, Canada: ACM, 2005, pp. 241–252, ISBN: 1-59593-022-1. DOI: 10.1145/1064212.1064240. [Online]. Available: http://doi.acm.org/10.1145/1064212.1064240.

[50] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984, vol. 38.

[51] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: ACM, 2013, pp. 123–134, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486020. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486020.

[52] J. Ros-Giralt *et al.*, "On the bottleneck structure of congestion-controlled networks," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, Dec. 2019. DOI: 10.1145/3366707. [Online]. Available: https://doi.org/10.1145/3366707.

[53] T. J. Hacker, B. D. Noble, and B. D. Athey, "Adaptive data block scheduling for parallel TCP streams," in *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on*, IEEE, 2005, pp. 265–275.

[54] E. Yildirim, D. Yin, and T. Kosar, "Prediction of optimal parallelism level in wide area data transfers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 12, pp. 2033–2045, 2011.

[55] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante, "Modeling and taming parallel TCP on the wide area network," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, IEEE, 2005, 68b–68b.

[56] *Fdt: Fast data transfer tool*, http://monalisa.cern.ch/FDT/.

[57] *Bbcp: Multi-stream data transfer tool*, http://www.slac.stanford.edu/~abh/bbcp/.

[58] Y. Gu and R. Grossman, "Udtv4: Improvements in performance and usability," English, in *Networks for Grid Applications*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, P. Vicat-Blanc Primet, T. Kudoh, and J. Mambretti, Eds., vol. 2, Springer Berlin Heidelberg, 2009, pp. 9–23, ISBN: 978-3-642-02079-7. DOI: 10.1007/978-3-642-02080-3_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02080-3_2.

[59] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The Science DMZ: A network design pattern for data-intensive science," *Scientific Programming*, vol. 22, no. 2, pp. 173–185, 2014.

[60] *Esnet website*, https://www.es.net/science-engagement/knowledge-base/case-studies/science-dmz-case-studies/, Accessed: 2021-06-08.

[61] K. V. Vishwanath and A. Vahdat, "Evaluating distributed systems: Does background traffic matter?" In *USENIX 2008 Annual Technical Conference*, ser. ATC'08, Boston, Massachusetts: USENIX Association, 2008, pp. 227–240. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404031.

[62] Q. Yin and J. Kaur, "Can machine learning benefit bandwidth estimation at ultra-high speeds?" In *Passive and Active Measurement: 17th International Conference, PAM 2016, Heraklion, Greece, March 31 - April 1, 2016. Proceedings*, T. Karagiannis and X. Dimitropoulos, Eds. Cham: Springer International Publishing, 2016, pp. 397–411, ISBN: 978-3-319-30505-9. DOI: 10.1007/978-3-319-30505-9_30. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30505-9_30.

[63] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is spdy?" In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA: USENIX Association, Apr. 2014, pp. 387–399, ISBN: 978-1-931971-09-6. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/wang.

[64] K. V. Vishwanath and A. Vahdat, "Swing: Realistic and responsive network traffic generation," *IEEE/ACM Trans. Netw.*, vol. 17, no. 3, pp. 712–725, Jun. 2009, ISSN: 1063-6692. DOI: 10.1109/TNET.2009.2020830. [Online]. Available: http://dx.doi.org/10.1109/TNET.2009.2020830.

[65] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 151–160, Jun. 1998, ISSN: 0163-5999. DOI: 10.1145/277858.277897. [Online]. Available: http://doi.acm.org/10.1145/277858.277897.

[66] R. Kettimuthu *et al.*, "Lessons learned from moving earth system grid data sets over a 20 gbps wide-area network," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, Chicago, Illinois: ACM, 2010, pp. 316–319, ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851519. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851519.

[67] W. Liu, B. Tieman, R. Kettimuthu, and I. Foster, "A data transfer framework for large-scale science experiments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, Chicago, Illinois: ACM, 2010, pp. 717–724, ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851582. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851582.

[68] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1026–1039, Aug. 2010, ISSN: 1063-6692. DOI: 10.1109/TNET.2009.2037812. [Online]. Available: http://dx.doi.org/10.1109/TNET.2009.2037812.

[69] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: ACM, 2014, pp. 479–490, ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626324. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626324.

[70] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015, pp. 395–408, ISBN: 978-1-931971-218. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/dong.

[71] M. Dong *et al.*, "PCC vivace: Online-learning congestion control," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, Apr. 2018, pp. 343–356, ISBN: 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/dong.

[72] M. Carbone and L. Rizzo, "Dummynet revisited," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 12–20, Apr. 2010, ISSN: 0146-4833. DOI: 10.1145/1764873.1764876. [Online]. Available: http://doi.acm.org/10.1145/1764873.1764876.

[73] *Netperf: Network throughput and latency benchmark*, https://hewlettpackard.github.io/netperf/.

[74] *Iperf: Bandwidth measurement tool*, http://software.es.net/iperf/.

[75] A. Feldmann *et al.*, "The lockdown effect: Implications of the covid-19 pandemic on internet traffic," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20, Virtual Event, USA: Association for Computing Machinery, 2020, 1–18, ISBN: 9781450381383. DOI: 10.1145/3419394.3423658. [Online]. Available: https://doi.org/10.1145/3419394.3423658.

[76] S. McQuistin and C. S. Perkins, "Is explicit congestion notification usable with udp?" In *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15, Tokyo, Japan: ACM, 2015, pp. 63–69, ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815716. [Online]. Available: http://doi.acm.org/10.1145/2815675.2815716.

[77] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, Colorado: ACM, 2013, 85:1–85:10, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503245. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503245.

[78] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning, Data Mining, Inference, and Prediction*. Springer-Verlag New York, 2009.

[79] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: A review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[80] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and Information Systems*, vol. 7, no. 3, pp. 358–386, 2005, ISSN: 0219-3116. DOI: 10.1007/s10115-004-0154-9. [Online]. Available: https://doi.org/10.1007/s10115-004-0154-9.

[81] M. Christ, A. W. Kempa-Liehr, and M. Feindt, "Distributed and parallel time series feature extraction for industrial big data applications," *CoRR*, vol. abs/1610.07717, 2016. arXiv: 1610.07717. [Online]. Available: http://arxiv.org/abs/1610.07717.

[82] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah, "Time-series clustering – a decade review," *Information Systems*, vol. 53, pp. 16 –38, 2015, ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2015.04.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437915000733.

[83] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, ser. AAAIWS'94, Seattle, WA: AAAI Press, 1994, pp. 359–370. [Online]. Available: http://dl.acm.org/citation.cfm?id=3000850.3000887.

[84] C.-J. Hsu, K.-S. Huang, C.-B. Yang, and Y.-P. Guo, "Flexible dynamic time warping for time series classification," *Procedia Computer Science*, vol. 51, pp. 2838 –2842, 2015, International Conference On Computational Science, ICCS 2015, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.05.444. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915012521.

[85] R. C. Holte, "Very simple classification rules perform well on most commonly used datasets," *Mach. Learn.*, vol. 11, no. 1, 63–90, 1993, ISSN: 0885-6125. DOI: 10.1023/A:1022631118932. [Online]. Available: https://doi.org/10.1023/A:1022631118932.

[86] W. Iba and P. Langley, "Induction of one-level decision trees," in *Machine Learning Proceedings 1992*, D. Sleeman and P. Edwards, Eds., San Francisco (CA): Morgan Kaufmann, 1992, pp. 233–240, ISBN: 978-1-55860-247-2. DOI: https://doi.org/10.1016/B978-1-55860-247-2.50035-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558602472500358.

[87] *Nuttcp: Network performance measurement tool*, https://nuttcp.org/.

[88] *Iperf new feature (pr#1178): Reliable receiving data on the server (not merged)*, https://github.com/esnet/iperf/pull/1178.

[89] *Iperf new feature (pr#1179): Partial file transfer on the client and appending data to the file on the server (not merged)*, https://github.com/esnet/iperf/pull/1179.