

Enhancing User Experience by Extracting Application Intelligence from Network Traffic

Author:

Madanapalli, Sharat

Publication Date:

2022

DOI:

<https://doi.org/10.26190/unsworks/24314>

License:

<https://creativecommons.org/licenses/by/4.0/>

Link to license to see what you are allowed to do with this resource.

Downloaded from <http://hdl.handle.net/1959.4/100607> in <https://unsworks.unsw.edu.au> on 2024-04-25

Enhancing User Experience by Extracting Application Intelligence from Network Traffic

Sharat Chandra Madanapalli

A dissertation submitted in fulfillment
of the requirements for the degree of

Doctor of Philosophy



School of Electrical Engineering and Telecommunications

Faculty of Engineering

The University of New South Wales

August 2022

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract	Declarations	Inclusion of Publications Statement	Corrected Thesis and Responses
---------------------------	--------------	-------------------------------------	--------------------------------

ORIGINALITY STATEMENT

☒ I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

COPYRIGHT STATEMENT

☒ I hereby grant the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).

For any substantial portions of copyright material used in this thesis, written permission for use has been obtained, or the copyright material is removed from the final public version of the thesis.

AUTHENTICITY STATEMENT

☒ I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract	Declarations	Inclusion of Publications Statement	Corrected Thesis and Responses
---------------------------	--------------	-------------------------------------	--------------------------------

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in the candidate's thesis in lieu of a Chapter provided:

- The candidate contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. they were responsible primarily for the planning, execution and preparation of the work for publication.
- The candidate has obtained approval to include the publication in their thesis in lieu of a Chapter from their Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

☒ The candidate has declared that **some of the work described in their thesis has been published and has been documented in the relevant Chapters with acknowledgement**.

A short statement on where this work appears in the thesis and how this work is acknowledged within chapter/s:

In total, I have used the content of seven papers in my thesis. The publications are mapped to four chapters as described below.

Chapter 3 presents the paper titled "FlowFormers: Transformer-based Models for Real-time Network Flow Classification" accepted into IEEE MSN 2021.


Chapter 4 consists of two papers: "Inferring Netflix User Experience from Broadband Network Measurement" published in IEEE/IFIP TMA 2019 and "ReCLive: Real-Time Classification and QoE Inference of Live Video Streaming Services" published in IEEE IWQoS 2021.

Chapter 5 presents a recent paper titled "Know Thy Lag: In-network Gameplay Detection and Latency Measurements" presented in PAM 2022.

Chapter 6 majorly consists of two papers: "Assisting Delay and Bandwidth-sensitive Applications in a Self-Driving Network" presented in SIGCOMM NetAI Workshop 2019 and "AutoQoS: Automatic, Application-aware QoS Configurations" -- paper in the process of resubmission. It also briefly mentions the paper "OpenTD: Open Traffic Differentiation in a Post-neutral World" in ACM SOSR 2021.

The mapping described above is also written in the Introduction chapter of the thesis and appropriate acknowledgements have been given to the co-authors.

Candidate's Declaration



I declare that I have complied with the Thesis Examination Procedure.

Abstract

Internet Service Providers (ISPs) continue to get complaints from users on poor experience for diverse Internet applications ranging from video streaming and gaming to social media and teleconferencing. Identifying and rectifying the root cause of these experience events requires the ISP to know more than just coarse-grained measures like link utilizations and packet losses. Application classification and experience measurement using traditional deep packet inspection (DPI) techniques is starting to fail with the increasing adoption of traffic encryption and is not cost-effective with the explosive growth in traffic rates. This thesis leverages the emerging paradigms of machine learning and programmable networks to design and develop systems that can deliver application-level intelligence to ISPs at scale, cost, and accuracy that has hitherto not been achieved before.

This thesis makes four new contributions. Our first contribution develops a novel transformer-based neural network model that classifies applications based on their traffic shape, agnostic to encryption. We show that this approach has over 97% f1-score for diverse application classes such as video streaming and gaming. Our second contribution builds and validates algorithmic and machine learning models to estimate user experience metrics for on-demand and live video streaming applications such as bitrate, resolution, buffer states, and stalls. For our third contribution, we analyse ten popular latency-sensitive online multiplayer games and develop data structures and algorithms to rapidly and accurately detect each game using automatically generated signatures. By combining this with active latency measurement and geolocation analysis of the game servers, we help ISPs determine better routing paths to reduce game latency. Our fourth and final contribution develops a prototype of a self-driving network that autonomously intervenes just-in-time to alleviate the suffering of applications that are being impacted by transient congestion. We design and build a complete system that extracts application-aware network telemetry from programmable switches and dynamically adapts the QoS policies to manage the bottleneck resources in an application-fair manner. We show that it outperforms known queue management techniques in various traffic scenarios. Taken together, our contributions allow ISPs to measure and tune their networks in an application-aware manner to offer their users the best possible experience.

List of Publications

During the course of this thesis, we published several peer-reviewed papers focusing on specific aspects of the complete system. They are listed below for reference (in the order of chapters in this thesis).

1. **S. C. Madanapalli**, R. Babariya, H. Kumar, and V. Sivaraman, “FlowFormers: Transformer-based Models for Real-time Network Flow Classification”, *IEEE Conference on Mobility, Sensing and Networking (IEEE MSN)*, Exeter, UK, December 2021.
2. **S. C. Madanapalli**, H. Habibi Gharakheili, and V. Sivaraman, “Inferring Netflix User Experience from Broadband Network Measurement”, *IEEE/IFIP Network Traffic Measurement and Analysis Conference (TMA)*, Paris, France, June 2019
3. **S. C. Madanapalli**, A. Mathai, H. Habibi Gharakheili, and V. Sivaraman, “ReCLive: Real-Time Classification and QoE Inference of Live Video Streaming Services”, *IEEE/ACM International Symposium on Quality of Service (IWQoS)*, Virtual Conference, June 2021
4. **S. C. Madanapalli**, H. Habibi Gharakheili, and V. Sivaraman, “Know Thy Lag: In-network Gameplay Detection and Latency Measurements”, *Passive and Active Measurements (PAM)*, Virtual Conference, March 2022
5. **S. C. Madanapalli**, A.G. Alcoz, A. Dietmuller, H. Habibi Gharakheili, and L. Vanbever, “AutoQoS: Automatic, Application-aware QoS Configurations”, **revised draft ready** to be submitted to *IEEE TNSM* journal
6. **S. C. Madanapalli**, H. Habibi Gharakheili, and V. Sivaraman, “Assisting Delay and Bandwidth-sensitive Applications in a Self-Driving Network”, *SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, Beijing, China, August 2019
7. V. Sivaraman, **S. C. Madanapalli**, H. Kumar, and H. Habibi Gharakheili, “OpenTD: Open Traffic Differentiation in a Post-neutral World”, *ACM Symposium on SDN Research (SOSR)*, San Jose, USA, April 2019

Acknowledgment

I'm immensely grateful to my supervisor Prof. Vijay Sivaraman who inspired me to pursue practical research, supported me throughout the journey, and struck a perfect balance between guiding me and giving me enough freedom to explore the field. His passion to use scientific research to help solve imminent problems the industry is grappling with today, drove me to conduct my research in a way that is deployable and useful. I feel very grateful to be supervised by a mentor who continues to inspire me in several ways.

I'm also very thankful to my co-supervisor Dr. Hassan Habibi Gharakheili who guided me at every step of my research. He always helped me break down and work through complex problems. His academic expertise was immensely helpful in presenting my research and writing papers. On many occasions, he went out of his way to help me edit my papers even when swamped with other work. I have learned good research practices, critical thinking, and the importance of humility from him, and I will always be grateful for that.

I'm grateful to my collaborators who have significantly contributed to this dissertation. In particular, I'd like to thank Prof. Laurent Vanbever from ETH Zurich for providing me an opportunity to collaborate with his team and for inspiring me to pursue hard problems in research. In addition, I'd like to thank Alex Mathai and Rushi Babariya who went beyond their scope of undergraduate thesis and collaborated on a couple of papers presented in this dissertation.

I express my gratitude to Canopus Networks which has funded my Ph.D. and given me an industrial perspective on research. I thoroughly enjoyed and learnt from the conversations over these couple of years with the core tech team: Maheesha, Himal, Craig, and Tara. They have helped me learn good practices in software development required to design and develop systems that scale to the rates of terabits per second.

I have also had many engaging conversations with my research group members: Ayyoob, Arunan, and Minzhao. I have learned a lot from Ayyoob ranging from research to life skills to good software practices. I had a great experience collaborating with Arunan and Minzhao on research projects. I'm grateful to have done my Ph.D. while being part of this amazing group.

Pursuing a Ph.D. can be challenging sometimes. Luckily, I had a great bunch of friends who supported me in many ways throughout the journey. I am so grateful to have had such a good time interacting with them and making so many wonderful memories. A huge shout out to all my friends, both new and old, who have stayed in touch with me throughout these years.

I'm heavily indebted to my loving family without whom I would not have been the person I am today. My father taught me the value of having a vision, and working hard towards it while being patient about the results. My mother has always believed in me and has been showering infinite and unconditional love. My brother has taught me to live my life to the fullest by being in the present. My mother always used to say, "You've got two options: a comfortable boat in which most of the people are sailing toward a known destination, and another one that may have a bumpy ride as it is traversing the unexplored waters but will lead you to treasure." My Ph.D. was a little bumpy but has left me a treasure full of learnings to begin the next chapter of my life. Thank you, family!

Contents

Abstract	i
List of Publications	ii
Acknowledgment	iii
List of Figures	viii
List of Tables	x
Acronyms	xi
1 Introduction	1
1.1 Thesis Contributions	4
1.2 Thesis Organization	6
2 Literature Review	8
2.1 Network Measurements	10
2.1.1 Traditional Network Measurements	12
2.1.2 Software-Defined Network Measurements	13
2.1.3 Programmable Network Measurements	14
2.2 Analysis and Inference	15
2.2.1 Network Traffic Classification	16
2.2.2 Application QoS/QoE Prediction	18
2.3 Control and Adaptation	19
2.3.1 Scheduling Primitives	22
2.3.2 Active Queue Management	23
2.3.3 QoS Control Frameworks	24
2.4 Summary	25
3 Application Identification via Encrypted Network Traffic Classification	27

3.1	Introduction	28
3.2	FlowPrint: Capturing Flow Behaviour	30
3.3	FlowFormers: Transformer-based Classifiers	33
3.3.1	Objective & Dataset	33
3.3.2	Vanilla DL Models	34
3.3.3	FlowFormers	37
3.4	Training and Evaluation	40
3.4.1	Training	40
3.4.2	Model Evaluation	41
3.4.3	FlowPrint Evaluation	44
3.5	Conclusion	46
4	Monitoring Video Streaming QoE	47
4.1	Introduction	48
4.2	Video Streaming Characteristics	49
4.2.1	<i>VideoMon</i> : Data collection tool	50
4.2.2	On-Demand Video Streaming Characteristics: Netflix	52
4.2.3	Live Video Streaming Characteristics: Twitch and YouTube	56
4.3	Inferring Video Streaming Performance	58
4.3.1	On-demand streaming performance	58
4.3.2	Live streaming performance	65
4.4	Conclusion	73
5	In-Network Game Detection and Latency Measurements	74
5.1	Introduction	75
5.2	Game Detection	77
5.2.1	Anatomy of Multiplayer Games	78
5.2.2	Signature Generation	80
5.2.3	Game Classifier	83
5.2.4	Evaluation	84
5.2.5	Field Deployment and Insights	85
5.3	Mapping Game Server Locations and Latencies	87
5.3.1	Active Measurements: Geolookup and Latency	87

5.3.2	Passive Measurements: Continuous latencies	88
5.3.3	Mapping Game Servers from the University	90
5.3.4	Comparing Gaming Latencies from Multiple ISPs	92
5.4	Conclusion	93
6	Data-driven Management of Application Performance	94
6.1	Introduction	96
6.2	AppAssist: Self-driving network prototype	97
6.2.1	System Architecture and Design	98
6.2.2	Assisting Sensitive Applications	102
6.2.3	Summary	109
6.3	AutoQoS: Application-aware automatic QoS configuration	109
6.3.1	Overview	112
6.3.2	System Design	114
6.3.3	Implementation: Application Profiles and Telemetry	119
6.3.4	Implementation: Software and Hardware Testbed	124
6.3.5	Evaluation	126
6.3.6	Discussion	134
6.4	Conclusion	135
7	Conclusions and Future Work	136
7.1	Conclusions	136
7.2	Future Work	138
	References	140

List of Figures

2.1	System Overview for monitoring and management of application performance.	9
3.1	FlowPrint Datastructure and Algorithm	30
3.2	FlowPrint Examples (only Bytes shown)	31
3.3	CNN Architecture with FlowPrint input	35
3.4	LSTM Architecture with FlowPrint input	36
3.5	Transformer-based Architecture with FlowPrint input	38
3.6	Type Classification Results.	42
3.7	Provider Classification Results.	43
3.8	FlowPrint Bin Results.	44
3.9	FlowPrint Duration Results.	46
4.1	<i>VideoMon</i> Tool Architecture.	49
4.2	Network profile of flows in a typical Netflix video stream.	52
4.3	Client metrics of a Netflix stream.	54
4.4	Correlation of network activity and client behavior.	54
4.5	Download activity and its auto-correlation of Twitch streaming: (a) Live, and (b) VoD.	56
4.6	Performance of phase classification: (a) confusion matrix, and (b) CCDF of confidence-level.	60
4.7	Inferring user experience considering throughput.	63
4.8	Detecting video quality degradation for users.	64
4.9	Chunk size versus resolution for Twitch (left), and YouTube (right).	69
4.10	Time-trace of buffer health value: ground-truth predicted, for a sample Twitch stream.	71

5.1	An illustrative example of signature generation using Fortnite traffic traces .	82
5.2	Signature of three representative game titles	82
5.3	The structure of our classifier, illustrating a progressive classification of a flow.	83
5.4	Dynamics of daily game-play hours across ten titles during field trial.	86
5.5	Distribution of game-play session duration across the ten titles.	86
5.6	Estimating TCP RTT.	88
5.7	Sankey diagram depicting game sessions, countries, and latency bands. . . .	89
5.8	Latency distribution of game servers from the campus field trial.	90
5.9	Latency per IP prefix of the League of Legends servers.	91
5.10	Measured latency across ISPs to popular external (outside country) game server subnets of Genshin Impact (left) and CS:GO (right).	92
6.1	System architecture of assisting applications.	97
6.2	Example of performance states transition for a video streaming application.	98
6.3	<i>AppAssist</i> data collection tool.	99
6.4	Buffer-based state machine for video streaming.	102
6.5	Latency-based state machine for online gaming.	104
6.6	Performance of sensitive applications without network assistance.	106
6.7	Performance of sensitive applications with network assistance.	107
6.8	AutoQoS System Overview.	113
6.9	Performance Curves of different classes of applications.	120
6.10	<i>AutoQoS</i> Software Testbed.	124
6.11	Comparison with different schemes across scenarios: Min-utilities (top) and Jain's fairness index (bottom)	129
6.12	Comparison with fixed optimum before and after tuning.	131
6.13	Sensitivity to Mapping Functions.	132
6.14	<i>AutoQoS</i> Hardware Benchmarks.	133

List of Tables

3.1	Classification Dataset	34
3.2	Dataset split for type classification	41
4.1	VideoMon Dataset	52
4.2	Fetch mechanisms of Twitch and YouTube video streaming.	57
4.3	Dataset resolution distribution.	67
4.4	Resolution prediction accuracy.	69
4.5	Buffer Stall Prediction Results.	73
5.1	List of games.	77
5.2	Fortnite Services, their name prefixes (suffix= ol.epicgames.com) and purpose.	79
5.3	Summary of detected game-play sessions in our field trial.	87
6.1	Experimental Scenarios	128

Acronyms

3GPP	3rd Generation Partnership Project
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AS	Autonomous System
BYOD	Bring Your Own Device
BYOT	Bring Your Own Technology
CCDF	Complementary Cumulative Distribution Function
CPU	Central Processing Unit
DDoS	Distributed Denial-of-Service
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
DoS	Denial-of-Service
DPDK	Data Plane Development Kit
EM	Expectation-Maximization
FTP	File Transfer Protocol
Gbps	Gigabits per second
HC	Hierarchical Clustering
HNB	Hidden Naive Bayes
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol

ID	Identifier
IDS	Intrusion Detection System
IMAP	Internet Message Access Protocol
IoE	Internet of Everything
IoT	Internet of Thing
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISP	Internet Service Provider
IT	Information Technology
LDAP	Lightweight Directory Access Protocol
Mbps	Megabits per second
ML	Machine Learning
MLP	Multilayer Perceptron
NAT	Network Address Translation
NFV	Network Function Virtualization
NGFW	Next Generation Firewall
NTP	Network Time Protocol
P2P	Peer-to-Peer
PC	Personal Computer
PCA	Principal Component Analysis
PCAP	Packet Capture
PN	Programmable Networks
PPS	Packets Per Second
QUIC	Quick UDP Internet Connection
RAM	Random Access Memory
RBF	Radial Basis Function
RF	Random Forest
SDN	Software Defined Networking

SFTP	Secure File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SRAM	Static Random-Access Memory
SSDP	Simple Service Discovery Protocol
SSE	Sum of Squared Errors
SSH	Secure Shell
SSL	Secure Sockets Layer
SVM	Support Vector Machine
TCP	Transmission Control Protocol
TN	True Negative
TP	True Positive
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VNF	Virtual Network Function
VPN	Virtual Private Network

Chapter 1

Introduction

Contents

1.1	Thesis Contributions	4
1.2	Thesis Organization	6

The Internet is rapidly evolving to support diverse applications such as video streaming, online gaming, teleconferencing, and social media. With the global COVID19 pandemic confining people to their homes, the Internet has seen an explosive growth in usage (average household consuming 650-750 Gigabytes per month in 2021 [1]) and engagement (average household spending 4.2 hours per day on popular apps [1]). The Internet is increasingly being used for critical purposes like daily work and education beyond entertainment. Further, Cisco's report [2] predicts that there will be 5.3 billion Internet users (66% of the global population) by 2023, up from 3.9 billion (51% of the global population) in 2018. The increase in the number of users and demand for the Internet, especially in the last couple of years, has subsequently resulted in at least 20% growth in peak rates across Internet Exchange Points (IXPs) and Tier 1 Internet Service Providers (ISPs) around the world. Several networks were not provisioned to handle such growth and encountered issues while running "hot", close to peak capacity [3].

Given the growth and dependence on the Internet, users want a good experience with their applications. Content providers have led efforts in improving the application performance by employing caches [4], developing novel congestion control algorithms [5] and

transport-layer protocols [6]. However, they do not control the last mile network which is typically the point of congestion [7] and can cause the poor experience. Therefore, it is paramount for network operators such as ISPs to ensure that their networks are meeting the diverse application requirements. Applications like video streaming (that account for around 60% of Internet traffic by volume [1]) require sustained high bandwidth to stream videos in 4K resolutions while applications like online gaming and teleconferencing require sustained low latencies to the servers to ensure a real-time communication. Further, emerging applications like Cloud Gaming (and Augmented Reality/Virtual Reality apps) require both high bandwidth and low latencies. The network operators are now realizing that networks designed to support just bandwidth/usage are no longer serving the requirements of all the applications. More importantly, using “speed” as a measure of network performance is no longer sufficient and operators need new ways of measuring and tuning their network’s quality for each application.

Traditional network monitoring solutions have relied on counting packet and byte rates of traffic at various aggregation levels ranging from a switch port to a VLAN using legacy protocols like SNMP [8]. These measures are coarse, aggregate, and give a very high-level view of network operations and only from a usage point of view. To gather finer-grained data, proposals like NetFlow [9] and sFlow [10] emerged that could count and export per-flow records by maintaining a flow cache. While they improve on the granularity, they still are not sufficient to measure application performance on the network. To do so, the network operators need to identify the application traffic i.e. assign a network traffic flow to an application type (e.g. video streaming), measure the application’s performance (e.g. able to play 4K videos without buffering), and subsequently take actions to manage the application performance by tuning their networks.

In-network monitoring and management of application performance and associated user experience, while desirable, is a non-trivial task. On the one hand, there are proposals that require communication between the network and the applications directly via APIs [11–13]. These approaches are hard to deploy as they (a) require collaboration between networks and application developers and (b) are difficult to scale with several thousand instances of applications. On the other hand, presently deployed monitoring solutions, collect sampled and coarse granular data (at a per-flow level) which are not sufficient to estimate

application-level performance. Implementing an in-network application performance monitoring and management solution has the following challenges:

1. Widely employed encryption technologies such as TLS and AES make it difficult to detect the application let alone monitor its performance;
2. Each application has a different performance criteria (video streaming requires high resolution and no buffering, games require low latency, etc.) and hence requires different measurements;
3. Customizing network monitoring for each application introduces complexity, comes at a higher cost (both computationally and economically), and is challenging to scale to the growing volumes of internet traffic;
4. Managing performance of multiple applications whose traffic mix changes dynamically requires operators to continuously configure and tune their networks which is not practical.

In this thesis, we present our efforts in addressing the existing gap in the space of application-aware network monitoring and management. Our thesis leverages advancements in Machine Learning and Programmable networks to tackle the aforementioned challenges by:

1. Performing encrypted traffic classification by employing modern machine learning and statistical algorithms by capturing traffic shape in a compact data structure;
2. Studying the behaviour of bandwidth-sensitive (live and on-demand video streaming) and latency-sensitive (multiplayer gaming) applications to design appropriate network telemetry functions (NTFs) that can accurately extract the required minimal set of features that can estimate the associated user experience;
3. Developing systems that leverage technologies like fast packet I/O (*e.g.*, DPDK [14]) and modern programmable switches (*e.g.*, Tofino [15]) to execute the network telemetry at scale and low costs;

4. Designing automatic control loops and frameworks that can manage the application performance at the bottleneck links via dynamic resource sharing according to either ISP set policies or application performance estimations.

1.1 Thesis Contributions

In this thesis, we make significant contributions to the field of application-aware network monitoring and management starting with application classification as the first step. Then we focus on measuring application performance and associated user experience of popular applications starting with video streaming. We subsequently shift our focus to detecting and monitoring the performance of latency-sensitive gaming applications. Finally, we develop methods to proactively measure and improve the performance of applications contending on congested links. The four contributions are enumerated in detail below:

1. First, we develop a deep-learning-based network traffic classification system consisting of two components (a) a novel network behaviour representation (called FlowPrint) that extracts per-flow time-series byte and packet-length patterns, agnostic to packet content and (b) use attention-based Transformer encoders (called FlowFormers) to enhance FlowPrint representation and classify internet traffic to identify application type and provider. FlowPrint extraction is real-time, fine-grained, and amenable for implementation at Terabit speeds in modern P4-programmable switches. FlowFormers leverage transformer encoders to outperform conventional DL models. Implementation and evaluation of FlowPrint and FlowFormers on live university network traffic, yields a 95% f1-score to classify popular application types within the first 10 seconds, going up to 97% within the first 30 seconds, and yields a 95+% f1-score to identify providers within video and teleconferencing traffic flows.
2. Second, we develop a system to monitor the user experience of both on-demand and live video streaming applications. We develop a measurement tool for collecting network flow activity and video client playback metrics and deploy it in home networks and synthetic network conditions to collect over 500 hours of video playback data. We, then, analyse the behavioural profile of videos from Netflix, Twitch, and

YouTube showing how on-demand and live streaming have different patterns of network activity. We develop methods for the ISP to infer Netflix (on-demand video) user experience in terms of buffer fill-time, video bitrate, and throughput, and detect playback buffer depletion and quality degradation events. For live video streaming, we develop a method that estimates QoE metrics in terms of resolution and buffer stall events with overall accuracies of 93% and 90%, respectively.

3. Third, we develop methods that give ISPs visibility into online gaming and associated server latency. We analyse packet traces of ten popular games and develop a method to automatically generate signatures and accurately detect game sessions by extracting key attributes from network traffic. Field deployment in a university campus network identifies 31k game sessions representing 9,000 gaming hours over a month. We perform BGP route and Geolocation lookups, coupled with active ICMP and TCP latency measurements, to map the AS-path and latency to the 4,500+ game servers identified. We show that the game servers span 31 Autonomous Systems, distributed across 14 countries and 165 routing prefixes, and routing decisions can significantly impact latencies for gamers in the same city. Methods proposed in this chapter give ISPs much-needed visibility into gaming so they can optimize their peering relationships and routing paths to better serve their gaming customers.
4. Fourth, we propose a self-driving network architecture (called *AppAssist*) that directly measures, optimizes, and dynamically controls application performance. We develop a method to measure and model application state in real-time using network behaviour data. We apply our framework to two representative applications, video streaming, and gaming, and show how the network can detect application deterioration in terms of playback buffers and ping latency respectively, and apply remedial action to improve application performance without requiring any explicit signalling. Building upon it, we design and develop a complete system (called *AutoQoS*) that extracts application-aware network telemetry from programmable switches and dynamically adapts the QoS policies to manage the bottleneck resources in an application-fair manner. We implement *AutoQoS* in both software and hardware testbeds. We show that it outperforms known queue management techniques in various traffic scenarios and can closely approximate an optimal static QoS configuration.

Taken together, our contributions help ISPs to detect applications, monitor the performance of QoE-sensitive applications (such as video streaming and gaming), and help alleviate congestion-induced QoE degradation at the bottleneck links by automatically distributing the queue resources to enhance user experience. We extract application intelligence from the network traffic and enable ISPs to measure and tune their networks in an application-aware manner to offer a good experience to their users. We obtained ethics clearance (HC16712) for work in this thesis which uses data from our university network – this clearance approves the analysis of campus network traffic data to derive insights into aggregate video streaming and gaming behaviours without identifying the users behind the client ip addresses.

1.2 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 surveys related literature to give a background on network measurements, inferencing methods, and adaptation strategies used to monitor and manage application performance on the Internet. It reviews current state-of-the-art tools and techniques in the field of in-network Internet application performance management and highlights the potential of emerging technologies including programmable networks and machine learning to address the identified gaps.

Chapter 3, presented in [16], elaborates on encrypted traffic classification methods using rich yet compact network telemetry and state-of-the-art ML model architectures.

Chapter 4, presented in [17, 18], describes the design and implementation of a real-time video performance monitoring system that consists of statistical and ML-based models to infer on-demand and live video streaming QoE in terms of resolution and buffer health using sophisticated network measurements like chunk metadata extraction.

Chapter 5, presented in [19], develops a deterministic and fast game detection system using automatically generated gameplay signatures derived from labelled packet traces and performs passive and active latency measurements to map out the gaming servers and identify high latency paths within ISP networks that can be optimized using better peering

relationships.

Chapter 6, presented in [20, 21], begins by describing a prototype self-driving network that can dynamically assist suffering applications and thereafter develops a robust system that leverages application intelligence extracted from programmable networks to automatically configure QoS policies and optimize the performance of diverse classes of applications in a dynamically varying traffic mix.

Chapter 7 concludes this thesis and discusses potential future directions.

Chapter 2

Literature Review

Contents

2.1	Network Measurements	10
2.1.1	Traditional Network Measurements	12
2.1.2	Software-Defined Network Measurements	13
2.1.3	Programmable Network Measurements	14
2.2	Analysis and Inference	15
2.2.1	Network Traffic Classification	16
2.2.2	Application QoS/QoE Prediction	18
2.3	Control and Adaptation	19
2.3.1	Scheduling Primitives	22
2.3.2	Active Queue Management	23
2.3.3	QoS Control Frameworks	24
2.4	Summary	25

In this chapter, we survey the existing work done by academia and industry in the field of in-network monitoring and management of Internet performance. We begin by giving an overview that decomposes the work into three parts: measurement, inference, and control. In each part, we show the evolution of methods and technologies in recent decades and highlight the current state-of-the-art.

Figure 2.1 shows an overview of the major components required to run large networks that can better serve applications. To do so, ISPs need to:

1. *measure* and collect metrics in-network (from packets) indicative of application per-

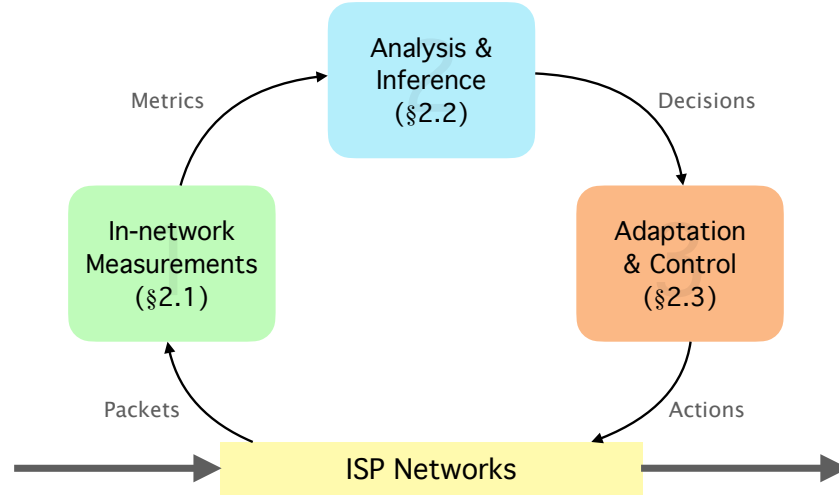


Figure 2.1: System Overview for monitoring and management of application performance.

formance,

2. *analyse* the measurements and make inferences/decisions using data-driven models,
3. *adapt* the network configuration to make it more performant using the inferences made from the measurements

These measurement and inference components constitute network monitoring and enhance visibility into the network. The adaptation component enables ISPs to take action to manage the network to improve the application performance. Monitoring and management, when coupled together, yield the best results in forming a self-driving network [22]. However, the ISPs may adopt an incrementally deployable approach in which monitoring components are deployed first to enable visibility into the network. Adding on to the monitoring systems, ISPs can deploy management components to take actions either at a short time scale (*e.g.*, seconds to minutes) such as dynamic resource distribution [20, 21] or a long time scale (*e.g.*, weeks to months) such as provisioning capacity or performing better peering.

Prior work has tackled each of these aspects and has made significant progress over the past few decades. We now focus on each of these three pillars and highlight their related work.

2.1 Network Measurements

Overview. Network measurements have been done for decades not only to monitor, debug, and manage networks but also for securing and optimizing them. Early “active measurement” tools like ping [23] and traceroute [24] aided in reachability measurements while tools like tcpdump [25] aided in passively capturing packets. Then came tools like SNMP [8, 26] and NetFlow which primarily aided in network management. With the advent of Software-Defined Networking (SDN) that decoupled the network control plane and data plane, many customized measurement tools started to be developed by both industry and academia such as OpenConfig [27], OpenNetMon [28] and more [29]. Recently, with the introduction of programmable data planes [30], very accurate and fine-grained in-band telemetry tools have started to evolve which can perform per-packet measurements at terabits per second.

We classify the related work in network measurement by considering its prominent aspects:

1. *Who measures the network?*

The network can be measured by applications themselves to run in a fair and efficient manner, or the operators to manage the network well or other entities such as content providers and end-users. Modern applications (*e.g.*, video streaming [31–33]) and protocols (*e.g.*, TCP) continuously measure the network conditions (such as available bandwidth) and adapt their application logic (*e.g.*, bitrate adaptation [34] or TCP back-off [35]) to enhance application performance leading to a better user experience. Network operators, on the other hand, primarily measure the network elements for aggregated statistics such as port rates and traffic volumes to help them with network management and operations [8, 26]. Further, industrial, government and research organizations also perform measurements on a global scale to understand the state of the Internet, connectivity issues, and track Internet evolution [36]. Finally, some measurements such as latency and speed tests [37] are also triggered by end-user to measure their network quality.

2. *What type of network measurement?*

There are predominantly three types of measurements: Active, Passive and Hybrid (active and passive) [38]. In active measurements, packets are injected into the network to *measure* its properties such as latency, throughput, loss etc. Ping and speed tests are examples of active network measurements. In passive measurements, already existing traffic is monitored to extract measures of interest. For instance, one can passively estimate latency to web servers by analysing the three-way TCP handshake which occurs to establish an HTTP(S) connection. Hybrid measurements include both active and passive methods *i.e.*, packets are injected into the network and also passively measured (say in the core) to estimate network metrics.

3. *What objectives can be addressed by measuring the network?*

Network measurements are often done to dimension and tune the network to give users the best possible experience at lowest cost to the operator. In addition, they also serve some specific use-cases such as network security [39], IoT device detection [40], accounting and legal interception [41]. Traffic monitoring, especially heavy-hitter detection [42], is also a common use case for network-wide measurements. Modern programmable networks also support fine-grained in-band telemetry [43] to analyse network performance to the granularity of queues in high-speed data center switches.

4. *What is the granularity of network measurement?*

Network measurements can be taken in varying granularities ranging from most-granular raw packet captures (using tcpdump [25]) to very aggregated per-device or per-port traffic rate counts (using SNMP [8]). Intermediate data commonly involves flow level logs exported using NetFlow [9] or HTTP logs typically collected at the server endpoint to debug service performance [44].

5. *How frequently is the network measured?*

Network measurement can be ad-hoc or taken continuously. Often network operators take coarse grained measures such as link utilizations continuously but rely on finer-grained information only when certain incidents occurs. Another form of infrequent measurement is collecting packet captures (aka pcaps) of events to be analysed at a later point in time (post-facto) if needed.

In this section, we chronologically review the domain of network measurements as it applies to the context of this thesis *i.e.*, we look at measurements primarily done in operator networks with an objective to manage Internet performance (more specifically application performance and the associated user experience). While we describe the related work we also discuss the measurement type, granularity (per-packet, per-flow, app-level or device-level) and frequency (ad-hoc or continuous). We begin by describing traditional network measurements which started in 1990s-2000s, followed by measurements made after the advent of SDN (after 2008) and finally discuss modern measurement tools and methods fueled by recent (since 2015) programmable data plane technology.

2.1.1 Traditional Network Measurements

Traditional measurements majorly focused on relatively simple performance metrics like connectivity (*is the network reachable?*), latency (*how far apart are two endpoints?*), bandwidth (*what is the available capacity for data transfer between two end-points?*) [45, 46]. Active measurement tools like Ping [23] are typically used to check connectivity and latency. Traceroute [24] was later developed to provide more fine-grained path information consisting of hops in between the endpoints. Tools like iperf [47] are used to measure bandwidth between two endpoints. In addition to these active measurement tools, passive tools such as tcpdump [25, 48] was developed to create packet captures to help operators debug protocol-based issues.

In addition to running tools on end-hosts such as described above, operators also collected information such as link utilizations from network elements such as switches and routers via port traffic counters. Simple Network Management Protocol (SNMP [8, 26] was adopted as a standard protocol through which these measurements could be taken. As switching chipsets started to mature, technologies like NetFlow [9], IPFIX [49] and sFlow [10] started to emerge as data exporting formats. NetFlow [9] and IPFIX [49] use software agents on switches to export traffic statistics at a flow level. sFlow [10] on the other hand, uses packet sampling technique to offer visibility into the network traffic. Due to limited compute and memory resources in the switches, these tools collect only sampled and aggregated information which is not enough for use cases like application performance

and user experience monitoring.

2.1.2 Software-Defined Network Measurements

Software-defined networking was a paradigm of networking which decoupled the control plane from the data plane. OpenFlow [50] introduced a standard way for disaggregated network control planes to talk to the data planes. Its monitoring features included APIs to fetch statistics such as byte and packet counters at various granularities from ports to five tuple flows. It introduced a flexible match-action paradigm in which packets could be matched on their header fields and then statistics can be exported for that matched flow. Many tools were developed leveraging SDN's flexibility to measure not only at a finer granularity but also targeted to use cases beyond simple network management [29].

OpenNetMon [28] was one of the earliest works which leveraged OpenFlow to design and build a monitoring system. Since then various SDN-based measurement methods have been proposed in both academia and industry for traditional network performance measurements such as available bandwidth [51], packet loss [52], latency [53] big-data based traffic monitoring [54] and specific use cases such as network security [39], heavy hitter detection [42, 55], path tracing [56], long flow monitoring [57], trouble shooting [58], etc. Many of the SDN based monitoring methods are passive with some like path tracing methods adopting a hybrid approach.

OpenConfig [27] standardized its APIs following the SDN paradigm and introduced Open Streaming telemetry [59]. In OpenConfig streaming telemetry is developing vendor agnostic measurement methods which address the problems faced by SNMP including scalability and complexity. The streaming telemetry methods advocate for continuous "push-based" telemetry as opposed to the traditional ad-hoc and pull-based telemetry. The SDN paradigm had created a huge shift towards making networks more open, measurable, and flexible to support multiple measurement use-cases including performance.

Along with SDN came Network Function Virtualization (NFV) in which network functions such as NATs, firewalls etc. shifted to using commodity servers. Intel's DPDK [14] is a key technology enabling fast packet processing using kernel bypass techniques. Sophis-

ticated packet processing logic can be written to analyse packet and flow-level attributes of the traffic. Work in [60] develops a flow monitoring solution to get accurate flow-level statistics using DPDK at rates of 10s of gigabits per second.

2.1.3 Programmable Network Measurements

About half a decade later, high-speed protocol independent programmable switching technology was introduced [30]. It allows the network switch to be programmed using a higher level programmable language such as P4. It enabled very sophisticated measurements such as in-band per-packet telemetry [43]. Further, all these measurements can be taken at a line-rate of terabits per second. While one cannot program arbitrarily complex telemetry functions, it gives enough flexibility to go beyond packet and byte counters which we've leveraged in our thesis to extract rich information to classify traffic and measure application performance. Our prior work in [57] uses OpenFlow and DPDK to monitor long flows in university traffic.

Traffic monitoring tools built on programmable data planes included use cases like heavy hitter detection, flow monitoring, sketch-based traffic matrix monitoring, in-band network telemetry, query-driven telemetry among others. Work in [61–63] use hash tables in the data plane to detect the heaviest flows of the network which typically cause congestion and performance issues. Several researchers have proposed flow monitoring solutions which leverage the flexible parsing and register/counter based memory features to accurately measure flow level statistics at scale. Turboflow [64] leverages both switch compute and CPU compute and memory to extract accurate flow level statistics by collecting micro flows on the switch and aggregating them using CPU. FlowStalker [65] maintains statistics at the switches and uses crawler packets to collect them network wide and aggregates at a collection point. Flow monitoring processes a lot of data to produce accurate records. Sketch based measurements offer a way to approximately measure traffic properties based on summaries collected of the current traffic state. ElasticSketch [66], Interleaved Sketch [67] and FCM [68] are some examples that leverage sketch data structure to approximately measure traffic statistics at scale.

Two new telemetry paradigms have emerged with the advent of programmable data planes: In-band telemetry [43] and query-driven telemetry [69]. In-band telemetry refers to taking measurements alongside the packets traversing the network. This is a hybrid measurement in which a packet/flow is passively monitored with active information about queueing and path information. It has been used to detect micro-bursts in the datacenter [70] and diagnosing latency spikes in web performance [71]. Work in [69, 72] performs query driven telemetry in which queries are compiled down to the P4 dataplane to extract very specific information about the network traffic.

Our Work

This thesis leverages advances in network measurement described above to collect metrics that aid in application classification and performance estimation. We leverage NFVs and sophisticated telemetry function support in DPDK to extract flow behaviour profiles in Chapter 3 (specifically in section §3.2) and fine-grained and chunk-based telemetry to extract user experience metrics such as resolution and buffer stalls for video streaming applications in Chapter 4 (specifically in sub-section §4.2.1). Further, in Chapter 6 we leverage P4 dataplanes to extract sophisticated and specific metrics from the traffic which are indicative of application performance (§6.3.3).

2.2 Analysis and Inference

The measurements taken from the network need to be analysed to produce some actionable outcomes. Network traffic analysis has been done in the contexts of network security [39] (detecting attacks using analysis and anomaly detection models), IoT device detection [40, 73] (detecting which devices are present on a network), application classification (detecting which application is using the network) [74], website fingerprinting[75] (detecting websites being accessed on network), also application performance (QoS/QoE) (inferring how various applications are performing on the network). While prior inferencing approaches relied on statistical models, machine learning is being increasingly applied [76] in various network traffic analytics use cases. In this thesis, we focus on network analysis and inferencing done

specifically to classify applications and measure their user experience.

We classify the related work by considering the following major aspects of network traffic analysis:

1. *What is the granularity of analysis?*

Network traffic can be analysed at packet-level, flow-level, or anything intermediate (such as HTTP logs [77]). Note that while measurement and data collection can be at a packet level (using pcaps), the analysis can be a flow-level.

2. *What kind of inferencing models are developed?*

Prior analytical models relied majorly on empirically tuned statistical models [78, 79]. However, more recently, machine-learning based approaches [80] have shown significant advantages in developing models that can make smart inferences (*e.g.*, network security [39], IoT device detection [40] etc.)

3. *When are the measurements analysed?*

Network measurements can be analysed either in real-time or post-facto. While most of the methods perform post-facto analysis using publicly available traces [81], recently researchers have begun developing real-time analysis systems [18, 82–84].

We now use these *lenses* and review literature specifically in the contexts of application detection/traffic classification and application QoS/QoE estimation.

2.2.1 Network Traffic Classification

Network Traffic Classification (NTC) has been extensively studied by many researchers over several decades [74, 78, 79, 85]. Early work surveyed in [78, 79] typically used attributes like port number etc. and clear-text information in the packets to identify different types of application like e-mail, web browsing, FTP etc. by essentially detecting the protocol they were using. Over the recent years, as encryption is becoming increasingly adopted, it is getting challenging to detect applications just based on protocols. For instance, video streaming, web browsing and file transfers all happen over HTTPS which is not only

encrypted *i.e.*, no clear text information such urls accessed etc., but also uses the same port 443 on the server side (with a dynamic port on client side).

More recently, researchers have begun the use of machine learning/deep learning models for classification of network traffic [80]. Recent work for encrypted NTC using deep learning can be categorized into *packet-based*, *flow content-based* and *flow time-series-based*. Work in [86] and [87] classifies applications using 1-D CNNs and Stacked Auto Encoders (SAE) on byte sequences extracted using packet headers and/or payloads. *Flow content-based* approaches use RNN/LSTM models in addition to CNN and SAE with features collected over multiple packets within a flow, like session bytes (concatenated packet payloads) [88, 89], packet inter-arrival times [90], and packet lengths to identify internet apps.

Most of the prior work in *packet-based* and *flow content-based* categories was evaluated using a public dataset [91] which contains many un-encrypted protocols such as SMTP, POP3 (email label) and SFTP and FTP (file transfer label) which are easily distinguishable using just the packet headers and/or first few payload bytes. Further, to classify encrypted applications/services based on HTTPS, DL-models often fit to features in TLS handshake such as cipher info and server name indication field (SNI) (e.g. youtube.com) without which the model accuracy drops significantly [89]. Thus, such approaches are either outdated, due to the increasing use of encryption, or are susceptible to failure with upcoming protocols like TLS 1.3 wherein even the handshake is completely encrypted. Our work doesn't consider packet payloads as input but instead relies on flow's traffic shape and behaviour characteristics which are robust even with TLS 1.3 encrypted traffic.

Flow time-series-based approaches [40, 75, 84, 92, 93] rely purely on time-series features such as packet/PDU (protocol data unit) lengths [92], [75], [40] of a flow over time, inter-arrival times [40] and/or statistical features like transfer rates (bps/pps), burstiness, idle-time etc., derived from downstream traffic [84]. Work in [84] was limited to Video vs. Download classification. Authors in [93] use all of the above features *i.e.* time-series packet lengths, IATs along with TLS handshake bytes (excluding SNI and cipher info) and summarized flow statistics. Their dataset contains only TLS/QUIC-based flows and it relies on statistical flow information which is only available at the end of the flow. In other words, it performs a post-hoc classification of the flow.

2.2.2 Application QoS/QoE Prediction

Network traffic analysis is also performed to infer QoS/QoE that a network provides to an application [94]. Some researchers have built analytical models which map QoS metrics to QoE of applications. Prior work analysed network traffic to infer behaviour and QoE of applications like web browsing [95], video streaming [82, 96–98], video conferencing [99–101]. Further, the analysis was not restricted to desktop-based or browser based applications but also was extended to mobile applications [102]. Of all applications, video streaming was studied by many researchers due to its popularity and increasing user engagement. As we tackle video streaming QoE prediction in this thesis, we now focus on network analysis literature in the context of video streaming.

Video Streaming QoE Prediction

Existing approaches for estimating video users' experience are either statistical modelling-based or machine learning-based. Recent attempts such as eMIMIC [97] and BUFFEST [103] employ statistical models, using packet traces and HTTP requests respectively, to quantify users quality of experience (QoE). Machine Learning (ML) approaches [104–106] used recently, attempt at measuring QoE by predicting categorical estimates of experience metrics like *low*, *med*, *high* bitrates [105], [106] or *low*, *high* probability of bitrate switches and rebuffering [105],[106]. To collect the ground-truth, both [104] and [106] used client-side instrumentation on mobile devices, and [105] used HTTP logs and metrics exported to content provider. However, all of them used packet traces to derive fine-grained attributes such as RTT, packet losses, ACKs, retransmissions which are expensive to compute. In a parallel work [107], authors attempt to classify buffer states (*i.e.*, filling, maintaining, depleting) for YouTube videos using attributes derived from aggregate network profile. In [83], authors use low-level packet features to detect startup delay and re-buffering events in real-time for YouTube traffic but do not report the quality of video playback or its variation.

Many researchers [82, 97, 98, 108] have studied QoE metrics for video streaming services across providers such as YouTube, Netflix, Facebook, Bilibili and Amazon, particularly

focusing on VoD. Among existing works, only [98] studied QoE for live streaming services (Twitch) by estimating only the resolution metric. Further, prior works predominantly performed post-facto analysis of video streaming QoE using features extracted from pcap traces [82, 97, 108], or CDN logs [77, 109]. Authors of [98], however, evaluated their online methods via deployment in home networks.

Our Work

In this thesis, we use both machine-learning based and statistical models to classify application type (video streaming vs. conferencing vs. downloads etc.) in §3.3, detect gaming applications in §5.2 and also to infer application QoE metrics (like resolution, buffer states etc.) in §4.3. The models make inferences based on analysis of raw network traffic by deriving high-level metrics (such as *chunk* for video streaming) from packets of network flows.

2.3 Control and Adaptation

Measurements and inferences can help operators take better control of their networks. Operators can take actions on infrastructure such as network planning, better peering and route selection. They can additionally take actions to actively manage traffic such as fix transient congestion, dynamic prioritization of traffic, selective routing of traffic onto special low latency paths, traffic shaping etc.

While protocols such as TCP adapt to network conditions by using signals like available throughput, loss etc., we are looking at how the network can enforce control over the traffic to help the mix of applications achieve a better performance. There have been studies showing how just using TCP congestion control schemes is not enough to manage network resources as they are not interoperable with each other[110] and that active network intervention (by separating congestion control schemes into their own queues [111]) can improve their interoperability and performance. Modern data center protocols are also starting to leverage in-network control primitives such as priority queues to improve metrics such as

flow completion times [112].

Currently, network operators such as ISPs enforce control in security related contexts such as blocking malicious traffic using firewalls and IDS systems or blocking access to certain websites / services as done by the great firewall of China [113]. Actions such as active traffic management, beyond dropping/blocking or zero-rating [114] traffic to improve user experience, have seldom seen adoption in operator networks at the last mile bottleneck [7]. However, we believe that with the emergence of programmable networks and machine learning models being able to provide operators with rich metrics about user experience at scale, operators can now exercise different control schemes to ensure a good user experience.

We study the related work in the field of network control considering the following aspects:

1. *Where is the control being enforced?*

Network control can be enforced at any point between the user and the service. Closer to the user at the home gateway, network control is exercised to enforce parental controls, per device fairness, bandwidth limits and active queue management systems. On the other end, application services running in data centers use network control to enforce geo-blocking, load balancing etc. The network operator enforces control such as bandwidth shaping, queue management, legal interception etc. to manage internet traffic. In addition to these points of control, government organizations can control the network to apply country-wide policies.

2. *Who has access to APIs to control the network?*

Often the control APIs are exercised by the entity owning the point of control *i.e.*, users control their home gateway and operators control the broadband network gateway (BNG) to do the shaping of all their subscriber traffic. However, some proposals [11–13] have proposed APIs that operators can expose to users and content providers to take actions within their network (with some economic pricing models). Such cross-domain network control is starting to see some deployment in modern networks with users given some control over their network [115].

3. *What granularity of traffic is controlled?*

Internet traffic can be controlled at various levels of granularity. For instance, bottleneck buffer management using active queue management techniques such as CoDEL[116] operate on a per-packet level to manage buffer bloat on congested links, whereas a routing/peering action is done on a per prefix/AS level while traffic shaping is done at BNGs at a per-subscriber level. Firewalls typically block traffic on a per-connection/flow level. Traffic prioritization/zero-rating[114] is often done on a per protocol/application level.

4. *What is the timescale of control?*

Network control configuration can be enforced once, updated periodically at scales of weeks to months or can also be real-time requiring dynamic inputs. Configuring active queue management algorithms[117] such as Codel, PIE [116, 118, 119] or Cake[120], is a one time action which can help to fix bufferbloat-induced latency problems [121]. Network capacity expansion, BGP peering and routing optimizations are often done at timescales of weeks to months. Further, network firewalls also typically receive signature updates to detect new malware etc. in similar timescales. Any API driven control scheme, as presented in [11–13], require real-time control of the network and work using APIs called by either end-users or content providers to manage their traffic. We note that while all these systems interact with packets in real-time basis, what we considered is the timescale of intervention by an operator to configure/update these control schemes.

We now look at prior work focusing on network control frameworks and technologies designed to improve application QoS/QoE and subsequently enhance user experience. To manage QoS/QoE in a traffic mix, the general requirements are (1) classification (into classes of traffic based on their requirements), (2) marking (to indicate to the network what is required) and (3) scheduling (schedule applications' traffic to satisfy their requirement). We begin by looking at scheduling algorithms that are typically adopted to manage traffic followed by frameworks which build on top of these to share the resources amongst the traffic classes.

2.3.1 Scheduling Primitives

Traffic scheduling has been studied for decades and has been surveyed by many researchers [122, 123] and more recently researches have proposed abstractions such as PIFO [124], PIEO [125] and AIFO [126] to implement multiple scheduling algorithms at line rate in programmable switches. We now discuss most widely used scheduling algorithms to manage traffic with different requirements.

First-In First-Out (FIFO): The most basic scheduling primitive is a Drop-tail First-In First Out Queue. In this scheme, packets are enqueued into the buffer from the tail-end and are dequeued from the head. If the buffer gets full, incoming packets are dropped *i.e.*, not enqueued into the buffer. The exact implementation commonly used in linux kernels can be found here [127]. Protocols like TCP were designed assuming the dynamics of a FIFO scheme and react to congestion (*i.e.*, filled buffers) by measuring the drops/increase in latency and then reduce their sending rates to recover from the congestion.

Priority Queueing: In a priority queueing scheduler, there are multiple queues each serving traffic of certain priority. The packets are enqueued into a queue depending on their priority and are dequeued from the queues from the highest to the lowest priority. By its design, priority queueing stalls traffic of lower priority queue if a higher priority queue has packets to dequeue. This is known as starvation and is a common drawback noted in priority queueing. Often, the traffic prioritized is either known to be constant bitrate encoded or very short flows that finish within a few RTTs.

Unsurprisingly, this primitive is often used to serve latency critical applications such as VoIP and real-time communications. More recently, researchers have proposed leveraging priority queueing to minimize flow completion times in data center networks [112].

Weighted Fair Queueing (WFQ): WFQ schedulers also have multiple queues into which traffic is enqueued. At the dequeue however, each queue is given a fixed amount of available capacity which is configured by its weight. It is an extension of fair queueing in which each queue is given an equal amount of capacity (if it contains traffic). WFQ doesn't suffer from the starvation problem that exists in priority queueing.

WFQ schedulers are used to distribute available capacity (bandwidth) among classes of traffic. They are widely used for bandwidth-based Network Utility Maximization (NUM) [128]. Some researchers have proposed combining priority queue (called a low latency queue – LLQ) and WFQ to create a scheduler for real-time communications [129].

Programmable Scheduling Primitives More recently, researchers have designed and developed abstractions that can be used to specify a wide variety of scheduling algorithms on the same hardware making the scheduler programmable. PIFO [124] uses push-in first-out primitive which allows inserting packets into a random position of a queue to implement scheduling algorithms such as priority, weighted and deadline aware scheduling. PIEO [125] extends PIFO by making the abstraction push-in extract-out and can also implement non-work conserving disciplines. The recent work AIFO [126] uses intelligence at enqueue to implement PIFO using a single queue. Work in [130] approximates per flow fair queueing using rotating strict priority queues. Subsequently, the authors designed an improved systems using calendar queues to implement more complex and non-work conserving scheduling disciplines [131].

2.3.2 Active Queue Management

Active queue management techniques have been developed in parallel with the congestion control algorithms to combat buffer bloat [121] and help protocols like TCP achieve high network utilization with low latencies [132]. Notable techniques in the past which saw widespread deployment were Random Early Detection (RED) and its variant Weighted RED (WRED). In both schemes, the scheduler drops a packet randomly (depending on its drop probability) before the queue gets full. This triggers the loss-based TCP congestion control algorithm to reduce its rate. Doing so, the schemes aimed at keeping the buffer low. However, network operators found it challenging to tune the parameters.

Since then, new AQM techniques have been developed starting from Controlled Delay (CoDEL [133]) which explicitly tries to minimize the queueing delay and keeping it close to the target value. It was shown to reduce bufferbloat and offer lower latencies and high utilizations. FQ-CoDEL was subsequently developed [116] and added to the linux imple-

mentation [134] which added flow-fair scheduling on top of CoDEL. Other notable AQM implementations include PIE [118] (and correspondingly FQ-PIE [135]) which use proportional integral controller to manage queue occupancy. More recently CAKE (Common applications kept enhanced) [120] which uses heuristics to improve on top of FQ-CoDEL and FQ-PIE. Most of the vanilla AQM algorithms assume a drop-tail FIFO queue as the scheduling discipline. However, their flow queue (FQ) variant use multiple drop-tail queues (typically 1024) configured to dequeue in a round-robin fashion to ensure a flow-fair behaviour.

AQM techniques have shown to help to reduce congestion-induced latency in operator networks [136] and since recently are readily available in router and DOCSIS implementations [137].

2.3.3 QoS Control Frameworks

Traditional Internet treats every packet equally and has a best effort model to serve the traffic. However, to ensure a good QoS, the Internet evolved to support different requirements. The most prominent proposals to improve Internet QoS were Differentiated Services (DiffServ) [138] and Integrated Services (IntServ) [139]. DiffServ allowed applications to mark their traffic belonging to certain traffic class (*e.g.*, low delay, high bandwidth etc.) and required the routers in the core network to serve the requirements. DiffServ was a mechanism to give relative QoS. Intserv on the other hand was a framework designed to give absolute or guaranteed QoS to applications. It required network-wide resource reservation scheme which would pre-allocate the required resources to serve the application traffic. While IntServ was not scalable due to network-wide resource reservation, DiffServ didn't see much deployment due to challenges in tuning the required parameters such as rates and drop probabilities [140] at the routers.

Software-Defined Networking enabled development of new QoS control frameworks [141]. With its decoupled control and data plane, researchers proposed frameworks to reserve resources by making requests to the control plane which maintains network-wide state[142]. Further, updated versions of the OpenFlow [50] also supported traffic man-

agement primitives such as meters and integration with hardware queues. The paradigm of match-action made it possible to selectively manage traffic up to a per-flow level[143]. These new features of SDN were used for QoS improvement of multimedia applications [143, 144], of real-time applications [145] and to perform inter-domain routing [146] to improve QoS among other use cases (as presented in the survey[141]).

Another parallel stream of work in network resource distribution (to control QoS) uses Network Utility Maximization (NUM) approach. NUM frameworks use utility functions to distribute bottleneck resource (predominantly bandwidth) while maximizing a service level objective. Work in [128] allows data center operators to configure an objective that gets applied throughout the network and distributes bandwidth accordingly. Work in [21] applies a similar concept to ISP networks which can use utility curves to differentiate their service and offer various plans to their customers. There is a large body of literature on NUM-based resource distribution (refer to this survey [147]). They are primarily deployed to achieve objectives such as fairness or weighted fairness (in terms of throughput/rates) across tenants, hosts, or flows. These control frameworks rely on operator-level policies and do not often consider application performance as an objective directly.

Our Work

In our work (presented in Chapter 6), we leverage the rich metrics extracted from networks that are indicative of user experience to drive the network control decisions. We leverage widely deployed scheduling algorithms such as priority queueing (§6.2.2) and weighted-fair queueing (§6.2.2) as our traffic management primitives. However, we dynamically adapt the queue configuration and the traffic going to the queues using a custom designed control frameworks (§6.2.2 and §6.3.2) that can manage and enhance application QoE.

2.4 Summary

Prior work has studied each aspect of network management individually: network measurement, analysis and control. In network measurement, researchers have studied active

and passive measurements being done at different granularity but haven't yet addressed the gap of how to use these measurements to improve end user experience. In network traffic analysis, researchers have analysed traffic at a packet-level and flow-level to identify applications, detect IoT devices, infer application QoE etc. However, very few researchers have tackled the problem of analysis and inferencing at scale and in real time. In the field of network control, various packet scheduling and queue management techniques have been proposed but they do not make use of the application specific measurements and inferences that can be made from the network using modern programmable data planes and machine learning models to better guide the scheduling decisions.

This thesis primarily addresses the gap of bringing these pieces together to enhance end-user experience by extracting application level intelligence from the network behaviour. In doing so, it builds upon several state-of-the-art methods and proposes algorithms that can measure and monitor applications' performance with high accuracy and designs scalable systems that can execute every function (measurement, analysis and control) in real-time while making sure they feed into each other forming a closed loop.

Chapter 3

Application Identification via Encrypted Network Traffic Classification

Contents

3.1	Introduction	28
3.2	FlowPrint: Capturing Flow Behaviour	30
3.3	FlowFormers: Transformer-based Classifiers	33
3.3.1	Objective & Dataset	33
3.3.2	Vanilla DL Models	34
3.3.3	FlowFormers	37
3.4	Training and Evaluation	40
3.4.1	Training	40
3.4.2	Model Evaluation	41
3.4.3	FlowPrint Evaluation	44
3.5	Conclusion	46

Internet Service Providers (ISPs) often perform network traffic classification (NTC) to dimension network bandwidth, forecast future demand, assure the quality of experience to users, and protect against network attacks. With the rapid growth in data rates and traffic encryption, classification has to increasingly rely on stochastic behavioural patterns inferred using deep learning (DL) techniques. To do so, two key challenges pertain to (a) high-speed and fine-grained feature extraction, and (b) efficient learning of behavioural traffic patterns

by DL models. To overcome these challenges, we propose a novel network behaviour representation called FlowPrint that extracts per-flow time-series byte and packet-length patterns, agnostic to packet content. FlowPrint extraction is real-time, fine-grained, and amenable for implementation at Terabit speeds in modern P4-programmable switches. We then develop FlowFormers, which use attention-based Transformer encoders to enhance FlowPrint representation and thereby outperform conventional DL models on NTC tasks such as application type and provider classification. Lastly, we implement and evaluate FlowPrint and FlowFormers on live university network traffic, and achieve a 95% f1-score to classify popular application types within the first 10 seconds, going up to 97% within the first 30 seconds and achieve a 95+% f1-score to identify providers within video and conferencing traffic flows.

3.1 Introduction

Network traffic classification (NTC) is widely used by network operators for tasks including network dimensioning, capacity planning and forecasting, Quality of Experience (QoE) assurance, and network security monitoring. However, traditional classification methods based on deep packet inspection (DPI) are starting to fail as network traffic gets increasingly encrypted. Many web applications now use HTTPS (i.e. HTTP with TLS encryption) and browsers like Google Chrome now use HTTPS by default[148]. Applications like video streaming (live/on-demand) have migrated to use protocols like DASH and HLS on top of HTTPS. Non-HTTP applications which are predominately UDP-based real-time applications like Conferencing and Gameplay also use various encryption protocols like AES and Wireguard to protect the privacy of their users. With emerging protocols like TLS 1.3 encrypting server names, and HTTP/2 and QUIC enforcing encryption by default, NTC is bound to get even more challenging.

In recent years researchers have proposed to use Machine Learning (ML) and Deep Learning (DL) based models to perform various NTC tasks such as IoT device classification, network security, and service/application classification, ranging from coarse grain application type (e.g. video streaming, conferencing, downloads, gaming) to specific appli-

cation providers (e.g. Netflix, YouTube, Zoom, Skype, Fortnite). However, many of these existing approaches train ML/DL models on byte sequences from the first few packets of the flow. While the approach of feeding in raw bytes to a DL model is appealing due to automatic feature extraction capabilities, it usually ends up learning patterns such as protocol headers in un-encrypted applications and server name in TLS based applications. Such models have failed to perform well in the absence of such attributes [89], for example in TLS 1.3 that encrypts the entire handshake thereby obfuscating the server name.

Our work takes an alternative approach by building a time-series behavioural profile (a.k.a. traffic shape) of the network flow, and using that to classify network traffic at both application type and provider level. Firstly, we develop a method to extract flow traffic shape attributes (aka FlowPrint) at high-speed and in real-time (§3.2). FlowPrint’s data representation format keeps track of packet and byte counts in different packet-length bins without capturing any raw byte sequences, and provides a richer set of attributes than the simplistic byte and packet counters. It also operates in real-time, unlike other approaches e.g. [93] that perform post-facto analysis on packet captures. We show that FlowPrint is amenable for implementation in modern programmable hardware switches operating at multi-Terabit scale, and is hence suitable for deployment in large Tier-1 ISP networks.

We then design and develop FlowFormers: DL architectures that introduce attention-based transformer encoder [149] to the traditional Convolutional Neural Network (CNN) and Long Short Term Memory (LSTM) networks (§3.3). Transformer encoder greatly improves the performance as it allows the models to give attention to the relevant parts of the input vector in context of the NTC task. In other words, transformer encoder enhances our FlowPrint data prior to being fed to CNN and LSTM.

In §3.4, we evaluate both FlowPrint and FlowFormers on a real-world dataset obtained from our university campus traffic. We evaluate the data representation and the models on NTC tasks identifying (1) Application type (e.g. Video vs. Conferencing vs. Download etc.), (2) Video provider (e.g Netflix vs. YouTube vs. Disney etc.) and (3) Conferencing provider (Zoom vs. MS Teams vs. Discord etc.). We show that using FlowPrint collected just for the first 10 seconds of a flow yields a 95+% f1 score to identify 5 types of applications. We further show that applying transformers increases the accuracy of both CNNs

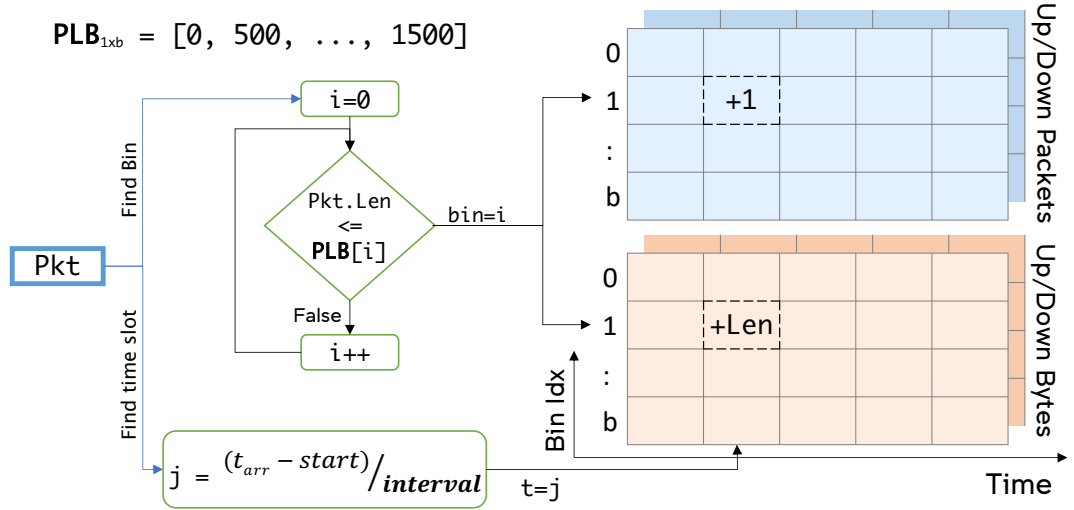


Figure 3.1: FlowPrint Datastructure and Algorithm

and LSTMs consistently across all the tasks. We demonstrate that the use of a transformer encoder together with LSTM (TE-LSTM) performs the NTC tasks with f1 scores 97.15%, 95.68% and 94.92% respectively.

3.2 FlowPrint: Capturing Flow Behaviour

FlowPrint is a data format built using counters to capture the traffic shape and behavioural profile of network flows. The data captured in FlowPrint doesn't include header/payload contents of packets and hence is protocol-agnostic and doesn't rely on clear-text indicators like SNI. It aims to support wide range NTC tasks which rely on activity profile such as application type identification (e.g. Video vs. Conferencing vs. Download), application service detection (e.g. Netflix, Zoom etc.), Device Identification (IoT sensors, smart gadgets etc.) etc. FlowPrint has been designed to be implementable not just in software, but also in modern P4 programmable network switches like Intel Tofino[15] that operates at several terabit per sec.

FlowPrint consists of four 2-D arrays: upPackets, downPackets, upBytes and downBytes. Each array consists of two dimensions: (length bins, time slots). As shown in Fig. 3.1, an incoming packet is placed into appropriate bin i based on its length. A list of packet length boundaries (PLB) creates b discrete length bins (on the y-axis). On the x-axis, the packet is placed into the time slot j in which it arrives relative to the flow start

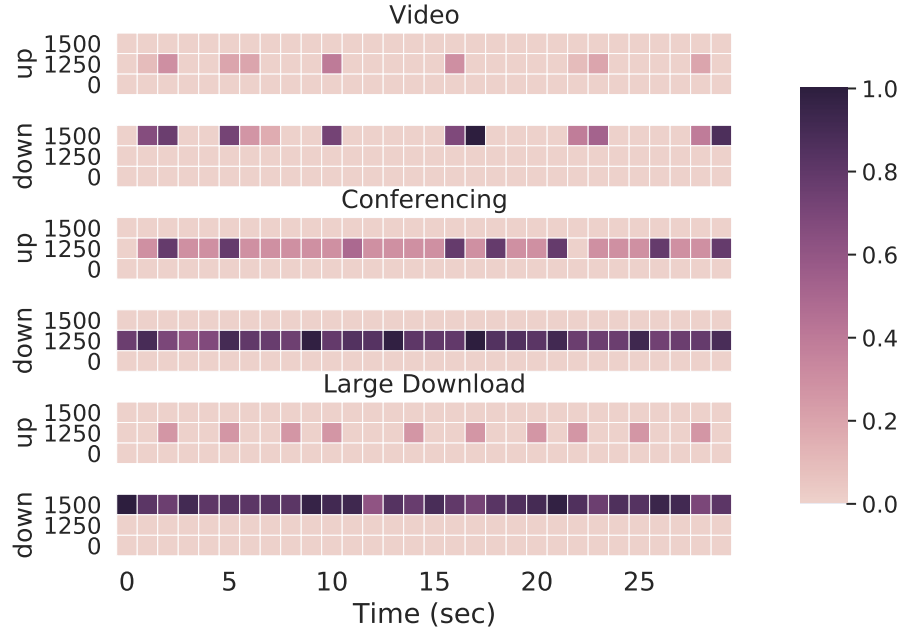


Figure 3.2: FlowPrint Examples (only Bytes shown)

– duration of the time slot is an input parameter called *interval*. Assuming its an upload packet, the cell (i,j) in `upPackets` array is incremented by 1 and the cell (i,j) in `upBytes` array incremented by the payload length of the packet (refer to Fig. 3.1). Thus, cell (i,j) of `upPackets` would contain the sum of all packets that arrive in time slot j with lengths between $PLB[i-1]$ and $PLB[i]$. The process remains similar in the other direction – down arrays are shown stacked in dark shade.

The choice of *interval* and *PLB* determines FlowPrint’s granularity and size. One may choose to have a small *interval* say 100ms and have 3 packet length boundaries or a large *interval* say 1 sec have 15 packet length bounds (in steps of 100Bytes). Such a choice needs to be made depending on the NTC task and compute/memory resources. We explore some of the trade-offs of FlowPrint configurations in §3.4.3.

Fig. 3.2 shows FlowPrint examples collected in our dataset. They show normalized `upBytes` and `downBytes` of 3 application types: Video, Conferencing and Large Download. The parameters used for the example are: *interval* = 1sec and $PLB = [0, 1250, 1500]$ – intuitively these length boundaries attempt to form 3 logical bins: ACKs, MTU-sized packets and packets in between. One can observe that FlowPrint clearly demarcates the behavioural profile of the flows. The video flow on top shows periodic activity – there

are media requests going in the up direction with payload length between 0 and 1250 and correspondingly media segments are being sent by the server using MTU-sized packets that fall in the bin (1250,1500]. Conferencing on the other hand is continuously active in the mid-bin (0,1250) in both upload and download direction with down being more active due to video transfer as opposed to audio transfer in the upload. A large download transferred typically using HTTP-chunked encoding involves the client requesting chunks of the file to the server which responds continuously with MTU-sized packets (in highest bin) until file downloads. Thus, this example illustrates the ability of FlowPrint to capture the traffic shape that can create markedly different patterns to identify application types.

We note that by a flow, we mean a set of packets identified using a `flow_key` constructed out of packet headers. Often, a 5-tuple consisting of `srcip`, `dstip`, `srcport`, `dstport` and `protocol` is used to form a `flow_key` to identify network flows at the transport level i.e. TCP connections and UDP streams. While our work also uses 5-tuple `flow_key`, FlowPrint is not inherently constrained by it. One may even use a 2-tuple (`srcip` and `dstip`) to construct a `flow_key` to identify all the traffic between a server and a client as a flow.

FlowPrint is amenable to implementation in high-speed P4 programmable switches[15]. A flow can be identified using its `flow_key` as match in a table, and sets of 4 registers can keep a track of up/down byte and packet counters. A controller can periodically poll the registers to get time-series of the counters at the defined *interval*. Once classified, the registers can be reused for a new flow.

FlowPrint fundamentally consists of four 2-D arrays. However, it can be extended by deriving two additional arrays: `upPacketLength` and `downPacketLength` by dividing the Bytes arrays by the Packets arrays in each direction. For instance, the `upPacketLength[i,j]` will contain the average packet lengths of packets which arrived in time slot `j` and are in the length bin `i`. These arrays can give precise time-series packet length measurements across the length bins. It is specifically useful to identify providers (e.g. Netflix, Disney) within a particular application type (video) as the overall shape remains very similar. In summary, the FlowPrint data structure has six 2-D arrays – collected over two directions and three counter types (packets, bytes, lengths) with each array of dimensions (numbins, time slots).

3.3 FlowFormers: Transformer-based Classifiers

In the section, we develop transformer-based DL models which efficiently learn features from FlowPrint to perform NTC tasks. We first explain the specific NTC tasks we consider in our work and our dataset. We then provide a background on 1-D CNN and LSTM models which are commonly used DL model architectures for NTC tasks. We also explain the process to convert the FlowPrint arrays into suitable input formats for these models. Finally, we present a brief overview of transformer encoder and develop FlowFormer models by introducing transformer encoders to the CNN and LSTM architectures.

3.3.1 Objective & Dataset

In our work, we tackle two specific NTC tasks: (a) Application Type Classification: Identify the type of an application (e.g. Video vs. Conference vs. Download etc.) and (b) Application Provider Classification: Identify the provider of the application/service (e.g. Netflix vs. YouTube or Zoom vs. Microsoft Teams etc.). These tasks are typically performed today in the industry using traditional DPI solutions but however rely on information like DNS, SNI or IP-block/AS based mapping. Due to increasing encryption adoption these solutions may no longer work and hence we instead take an alternative approach focusing on using traffic behavioral profile captured by FlowPrint.

Application Type Classification: The task identifies 5 common application types: Video streaming, Live video streaming, Conferencing, Gameplay and Downloads. An ML model will be trained to classify a flow into one of these 5 classes. Each type contains flows from different providers to make it diverse and not limited to provider-specific patterns. For instance, the Gameplay class has examples from the top 10 games active in our university network. For large downloads, while one may consider traffic from different sources, we chose Gaming Downloads/Updates from providers like Steam, Origin, Xbox and Playstation since they tend to be consistently large in size as opposed to downloads from other providers like Dropbox etc. which may contain smaller, say PDF, files. We note that Live video (video broadcasted live for example on platforms like Twitch etc.) has been intentionally separated from video on-demand to create a challenging task for the models.

Table 3.1: Classification Dataset

Task	# Flows per class	Classes
Application Type	40,000	Video, Live Video, Gameplay, Conferencing and Downloads
Video Provider	30,000	Netflix, YouTube, DisneyPlus and PrimeVideo
Conference Provider	40,000	Zoom, Microsoft Teams, Whatsapp and Discord

Application Provider Classification: This task identifies the provider within each type of application. We choose two popular types: Video streaming and Conferencing (and correspondingly train separate models). The objective is to detect the provider serving that content type. For Video, we detect if it is one of Netflix, YouTube, DisneyPlus or PrimeVideo (top providers used in our university). For conferencing, we detect if it is one of Zoom, Microsoft Teams, WhatsApp or Discord – two popular video conferencing platforms and two popular audio conferencing platforms.

Dataset: To perform the tasks above, we need a labelled FlowPrint dataset. We obtained the labels from a third-party commercial DPI which associates both application type and provider to each flow being collected. So, every data record is a three tuple $\langle FlowPrint, Type, Provider \rangle$. The FlowPrint arrays are recorded for 30 seconds at an *interval* of 0.5sec and with 3 bins ($PLB = [0, 1250, 1500]$). The data is filtered, pre-processed and labelled appropriately per task before feeding it to ML models. For instance, for the application type classification task, we filter the top providers of each class and just associate the type as the final label. So, Video class, for example, has records from top providers (e.g. Netflix, Disney etc.) but just with the label “Video” after the pre-processing. Table 3.1 shows number of flows (approx.) that have been used for each task.

3.3.2 Vanilla DL Models

We now present a brief overview of CNN and LSTM models extensively used for NTC tasks and how FlowPrint can be fed into each model. DL models, as opposed to traditional ML models, are suitable since they automatically extract task-specific features from FlowPrint.

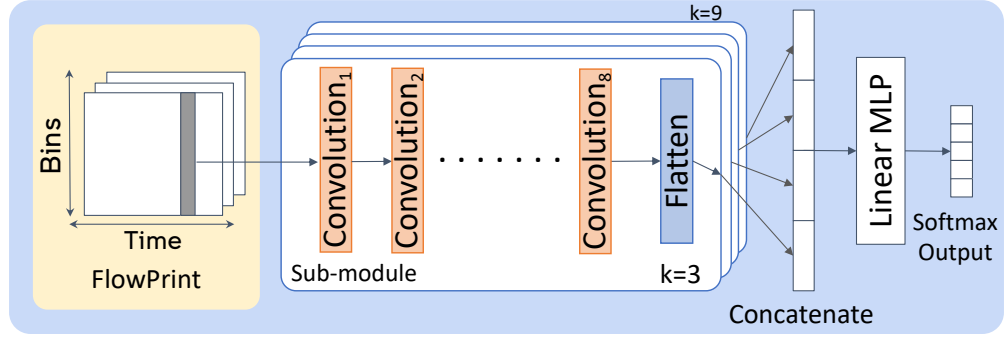


Figure 3.3: CNN Architecture with FlowPrint input

1-D CNN. It is a type of neural network widely used in the domain of computer vision to perform tasks like image classification, object detection, segmentation and since recently are also being used in time-series classification tasks. Traditional CNNs (2-D CNNs) are inspired from visual circuitry in brains wherein a series of filters (also called as kernels) stride over an channelled (RGB) image along both height and width collecting patterns of interest for the task. However, 1-D CNN (where filters stride over 1 dimension of image) have been shown to be more effective for time-series classification objectives such as NTC tasks. Further, CNN’s fast execution speed and spatial invariance makes them particularly suitable for NTC tasks [88].

FlowPrint needs no further processing to pass as an input to CNN (we omit 1-D for brevity) as it can be viewed as a colored image. Just as a regular image has height, width and 3 color channels (RGB), FlowPrint has bins (height), time slots (width) and, direction and counter types together forming six channels – upPackets, downPackets, upBytes, downBytes, upPacketLengths and downPacketLengths. Thus, FlowPrint is equivalent to a 6 channelled image of shape (numbins, timeslots, 6).

The CNN architecture (shown in Fig. 3.3) used in our work has 4 sub-modules each using a particular kernel size to perform multiple sequential convolutions on the FlowPrint image. The 4 kernel sizes used in our work are 3,5,7 and 9 along the time slot axis i.e. their field of view includes all bins, all channels and time slots equal to their kernel size. Using multiple sequential convolutions helps build features in a hierarchical way, summarizing to the most important features at the last convolutional layer. We use 8 layers as we found that results show marginal improvements on increasing the number of layers any further.

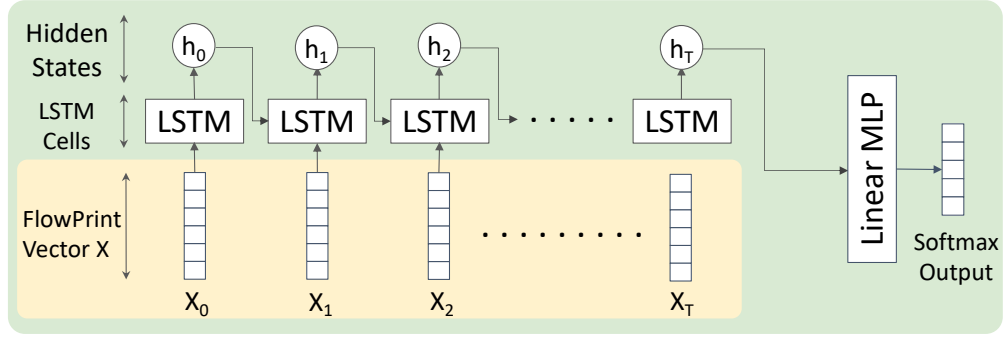


Figure 3.4: LSTM Architecture with FlowPrint input

The output from last layer of each module is flattened to a 32-dimensional vector using a dense layer, which is concatenated with the outputs of other modules. The concatenated output (32x4) is then passed to linear MLP (2 dense layers with 100 and 80 neurons) and then a softmax layer that outputs a probability distribution over the classes of the NTC task.

LSTM. It is type of Recurrent neural network (RNN) widely used in tasks such as time series classification, sequence generation etc since they are designed to extract time-dependent features out of the raw input. LSTM processes a given sequence one time step at a time while remembering context from previous time steps by using hidden states and cell state that effectively mimic the concept of memory. After processing the entire input, it produces a condensed vector consisting of features extracted to perform the given task. Due to this, LSTMs have been used to perform NTC tasks[90, 93] in addition to CNNs.

Our FlowPrint arrays need to be reshaped to be fed into an LSTM model. We convert FlowPrint into a time-series vector $X = [X_0, X_1, X_2, \dots, X_T]$ where each X_t is a $3 * 2 * b$ dimensional vector consisting of values collected in time slot t , from 3 counters types (i.e. bytes, packets and packet lengths) collected in 2 directions (i.e up and down) and for b packet length bins i.e. all counters collected in time t .

The architecture used in our work (shown in Fig. 3.4) has one LSTM layer (of 80 neurons) which sequentially processes the input X while keeping a hidden state $h(t)$ and a cell state $c(t)$ (cell state omitted in the figure). At each time step t , the LSTM is fed X_t , and $h(t-1)$ and $c(t-1)$ from previous time steps, to produce new $h(t)$ and $c(t)$. The final hidden state $h(T)$ is then fed to a linear MLP and a softmax layer to generate a probability

distribution over the classification labels.

3.3.3 FlowFormers

In order to improve the performance of CNN and LSTM based models on NTC tasks, we propose the use of Transformer Encoders on the input prior to feeding it into the vanilla model architectures. We first present a brief overview of Transformers, with a particular focus on its encoder which uses attention mechanism to enhance input features. We then develop two models by extending previous DL models using Transformer Encoders: TE-CNN and TE-LSTM respectively. We refer to TE-CNN and TE-LSTM as FlowFormers as they enhance **FlowPrint** features using **Transformers** to perform the NTC tasks.

Overview. Transformers[149] have become very popular in the field of NLP to perform tasks like text classification, text summarization, translation etc. A Transformer model has two parts an encoder and a decoder. The encoder extracts features from an input sequence and the decoder decodes according to the objective. For example, in task of German to English translation, the encoder will extract features from the German sentence and the decoder will decode them to generate the translated English sentence. For tasks like sentence classification only the feature extraction is required so the decoder part of the transformer is not used. Transformer encoder Models like BERT [150] are very effective in text classification tasks. Drawing inspiration from them, we develop a transformer encoder suited for NTC tasks.

Self-Attention. Transformer was able to outperform prior approaches in NLP due to one key innovation: Self-Attention. Prior to this, in NLP tasks, typically each word in a sentence was represented using a encoding vector independent of the context in which the word was used. For example, the word "Apple" was assigned same vector while it can refer to a fruit or the company depending on the context. A transformer encoder, on the other hand, uses a self attention mechanism in which other words in the sentence are considered to enhance the encoding of a particular word. For example while encoding this sentence, "As soon as the monkey sat on the branch it broke." Attention mechanism helps the transformer encoder to associate "it" with the branch, which is otherwise a non-trivial

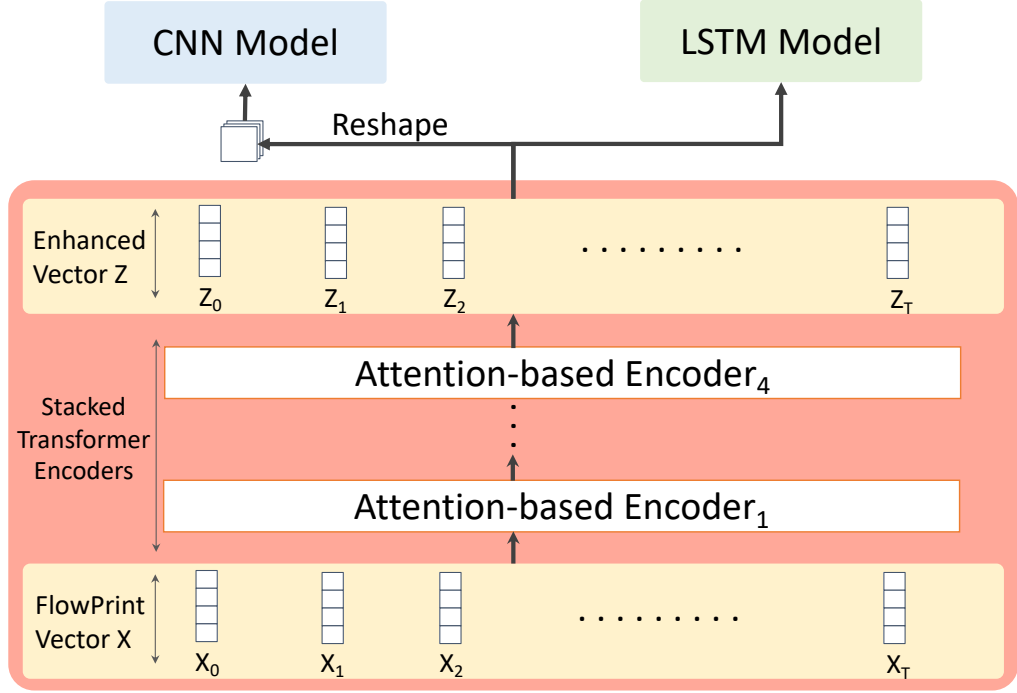


Figure 3.5: Transformer-based Architecture with FlowPrint input

task.

Concretely, self attention works by assigning an importance score to all input vectors for each output vector. The encoder takes in a sequence X_0, X_1, \dots, X_T where each X_t is a k dimensional input vector representing t -th word in the sentence. It outputs a sequence Z_0, Z_1, \dots, Z_T where each Z_t is the enhanced encoding of the t -th word. For each Z_t , it learns the importance score c_t ($0 \leq c_t \leq 1$) to give to each input X_t , and then constructs Z_t as follows:

$$Z_t = \sum_{t=0}^T c_t \cdot X_t, \text{ where } \sum_{t=0}^T c_t = 1$$

This is just an intuitive overview of attention, the exact implementation details are described in [149].

Transformers for NTC. Similar to enhancing a word encoding, transformers can be used to enhance the time-series counters collected in FlowPrint. We implement this idea by developing an architecture for FlowFormers (shown in Fig. 3.5). FlowFormers TE-CNN and TE-LSTM are CNN and LSTM extended with Transformer Encoders. FlowPrint

is first encoded by a Transformer Encoder before being fed into the CNN and LSTM architectures. In our work, we use 4 stacked transform encoders each with 6 attention heads. Each transformer encoder is designed exactly as in [149] with the dimensions of key, value and query set at 64.

The input format to the transformer encoder model is time-series vector X exactly similar to the input of LSTM. The input is passed through multiple stacked encoders which enhance the input with attention at each level. We empirically found that using 4 stacked encoders gives us the best results. The output of the final encoder is the enhanced vector Z exactly of the same dimensions as X . Now, instead of using raw FlowPrint X as input, we use enhanced version Z as an input to both models.

For TE-LSTM, the vector Z is directly fed into the LSTM model with no modification. For TE-CNN however, the vector Z is first converted into a 6-channel image (essentially, the reverse of the process of converting 6-channel image into input X described for LSTM). The image formatted input is then fed into the CNN model. We would like to highlight that since the input X and output Z are of exact same dimensions, the transformer encoder component is "pluggable" into the existing architectures requiring no modification to them.

Like most DL-models, the learning process even with transformer encoders is end-to-end i.e. all the model parameters including attention weights are learned by using stochastic gradient descent (SGD) and reducing the error of classification. Intuitively, in the case of TE-CNN, the CNN architecture updates the encoder weights to be more suitable to extract features using visual filters while in case of TE-LSTM, the LSTM updates the encoder weights to pick out time-series features. Irrespective of the vanilla model architecture used on top, transformer encoder is capable of enhancing the input such that it is amenable to how vanilla model works. This in turn makes the entire model (TE + vanilla model) learn and perform better compared to just the vanilla model architectures across the range of NTC tasks as shown in the evaluations next.

3.4 Training and Evaluation

Having explained the architectures of our DL-models, we now describe the training process followed by an evaluation of both FlowPrint and the models. As described in §3.3.1, we train our models for 2 tasks: (a) application type classification, (b) provider detection for video and conference application types. The dataset contains FlowPrint arrays labelled with both type and provider collected with the configuration mentioned in §3.3.1.

In addition to evaluating prediction performance of the models on the tasks above, we also evaluate the impact of parameters of FlowPrint. In particular, we evaluate the models' performance in various binning configurations and also with FlowPrint collected for a smaller duration i.e. 10sec and 20sec. For all these configurations, the training process remains the same as explained next.

3.4.1 Training

For each NTC task, the data is divided into train, validation and testing tests in the ratios 60%, 15% and 25% respectively. The data contains approximately equal examples from each class (for each task). All the DL models are trained for 15 epochs where in each epoch the entire dataset is fed to the model in batches of 64 at a time. Cross-entropy loss is calculated for each batch and then model parameters are learned through back-propagation using the standard Adam optimizer with an empirically tuned learning rate of $1e-4$. After each epoch, the model is tested on the validation data and if the results on the validation data begin to drop, the training process is halted. This makes sure that the model is not over-fitting to the data it is being trained on, commonly known as early stopping in DL literature. These training parameters (and models' hyper-parameters) can be tuned specifically to perform slightly better. However, our aim in this chapter is to evaluate efficacies of different model architectures on FlowPrint as opposed to investigating specific tuning parameters for the models for each task. Hence, we keep the training process simple and consistent across all models and tasks to perform a fair comparison.

3.4.2 Model Evaluation

We evaluate the vanilla models (CNN and LSTM) and FlowFormers (TE-CNN and TE-LSTM) on application type classification and provider classification tasks using FlowPrint input configured with 3 bins (0,1250,1500) and collected for 30 seconds (at 0.5 sec *interval* i.e. 60 time slots). We consider the commonly used metric f1-score (harmonic mean of precision and recall) as a measure of model performance across the tasks. We note that in our evaluation we observed that the overall precision and recall of the models was very close to the f1-score (varying in the third decimal place) and hence we show only f1-score for brevity.

Type Classification. For application type classification task, we consider the following labels: Video, Live Video, Conferencing, Gameplay, Download. We divide the dataset into 2 mutually exclusive sets based on application providers: set A and B (as shown in Table 3.2). We train the model on 75% data (60% train and 15% validation) of set A, and perform two evaluations: 1) test on 25% of data in set A and 2) On all the data in set B. We note that the class “Live Video” has been excluded in this set as it contained only two providers.

The evaluation on set A (shown in Fig. 3.6-top), compares weighted and per-class f1 scores of both vanilla models (CNN, LSTM) and FlowFormers (TE-CNN and TE-LSTM). Firstly, all models have a weighted average f1-score of at least 92% indicating the effectiveness of FlowPrint to capture the traffic shape and distinguish application types. Secondly, FlowFormers consistently outperform vanilla models (by 2-6%) showing the impact

Table 3.2: Dataset split for type classification

App Type	Set-A Providers	Set-B Providers
Video	Netflix, Youtube, Disney	AmazonPrime, Facebook
Conferencing	MS teams, Zoom, Discord	Skype, Whatsapp , Hangout
Gameplay	Genshin Impact,LoL, CoD,WOW, CS:GO,	CoD: Black Ops Cold War, Fortnite, Overwatch, Halo Reach, Battlefront II, Hearthstone
Downloads	Steam , XboxLive	Playstation, Oculus, Origin
Live Video	Twitch , Seven Live	—————

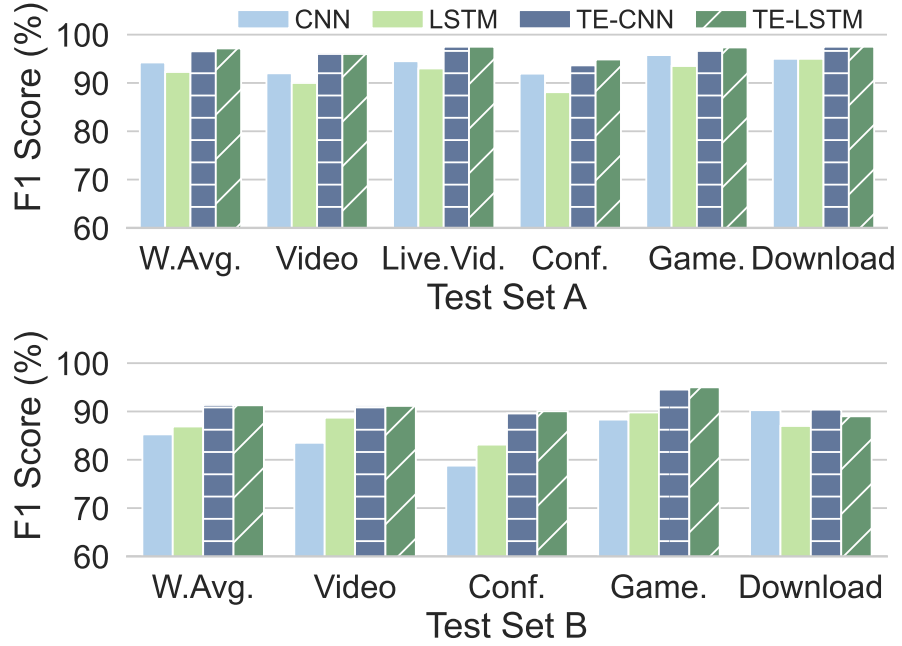


Figure 3.6: Type Classification Results.

of transformer encoders.

The evaluation on set B (shown in Fig. 3.6-bottom) tests the ability of models to learn provider-agnostic patterns to detect the application type since they were never shown examples from set B’s providers. While the performance drops across models as expected, we observe that FlowFormers outperform vanilla models by a huge margin (6-11%). This clearly depicts that FlowFormers can generalize better than vanilla DL models due to attention-based encoders enhancing the FlowPrint input.

Provider Classification. For application provider classification, we aim to classify top providers amongst 2 application types: Video and Conferencing, i.e. classify amongst Netflix, YouTube, Disney and AmazonPrime for Video and Microsoft Teams, Zoom, Discord and WhatsApp for Conferencing. This task is inherently more challenging since all the providers belong to the same application type and hence largely have the same traffic shape. The models need to pick up on intricate patterns and dependencies such as packet length distribution and periodicity.

For video provider classification (shown in Fig. 3.7-top), we observe that FlowFormers evidently perform better than the vanilla models with a 12% gain in the weighted average

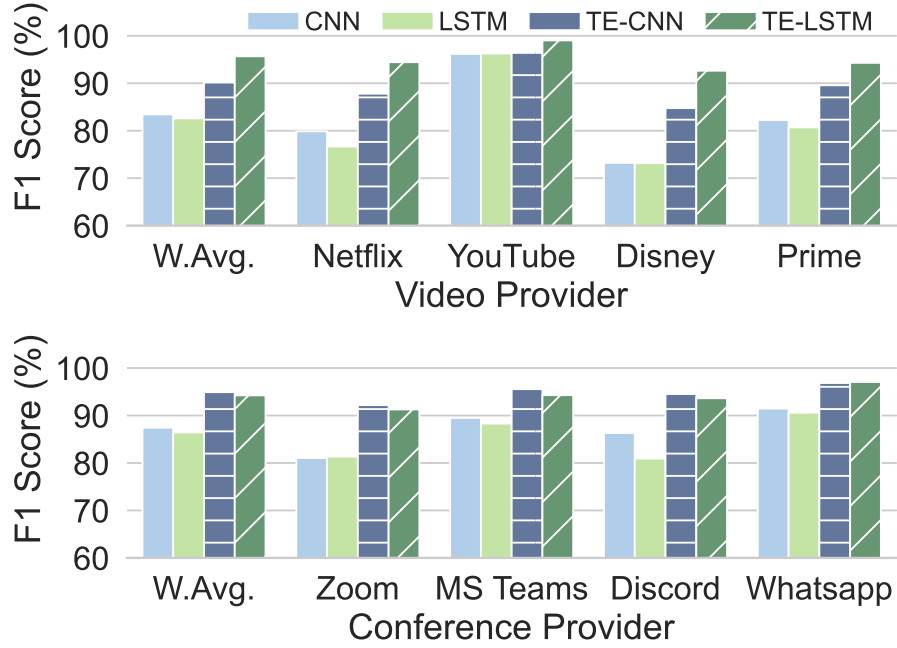


Figure 3.7: Provider Classification Results.

(e.g. TE-LSTM vs LSTM). We believe TE-LSTM outperforms other models since it can better pick up the periodic patterns (transfer of media followed by no activity shown in 3.2) that exist in the video applications. For instance, we observe (in our dataset) that YouTube transfers media every 2-5 seconds, whereas Netflix transfers it every 16 seconds. Transformers enrich FlowPrint by learning to augment this information and thus improving the classification accuracies.

Similarly in conference provider classification (shown in Fig. 3.7-bottom), FlowFormers outperform the vanilla models by 7% on an average (TE-CNN vs. CNN). We note that for this task, TE-CNN performs slightly better than TE-LSTM since this task predominantly relies on packet length distributions which tend to be different for the providers of conferencing applications rather than periodic patterns observed in video applications.

To summarize, FlowFormers are able to learn complex patterns beyond just the traffic shape, to outperform vanilla models in the challenging tasks of video provider and conference provider classification.

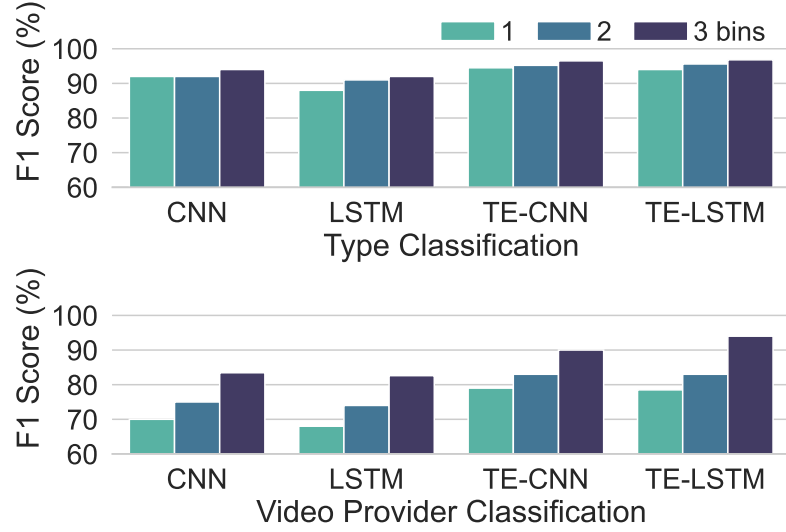


Figure 3.8: FlowPrint Bin Results.

3.4.3 FlowPrint Evaluation

We now evaluate the performance of FlowPrint by varying number of bins and length of data. We show that FlowPrint’s binning is a key factor that increases the performance across models, especially in the challenging task of provider classification.

Bin analysis. In previous evaluations, each FlowPrint sample had 3 bins ($PLB = [0, 1250, 1500]$). Now, we evaluate the impact of reducing bins to 2 ($PLB = [1250, 1500]$) and 1 ($PLB = [1500]$) on the performance of the models. We note that to reduce to 2 bins we have two choices, either (a) merge bin 2 and bin 3 or (b) merge bin 1 and bin 2 in the original 3 bin configuration. We chose to merge bin 1 and bin 2, since that was giving a better performance. So, in other words, the 2-bin configuration tracks the counters in less-than-MTU ($0 \leq pkt.len \leq 1250$) and close-to-MTU bins (≥ 1250). We additionally note that 1 bin, essentially means that there is no binning at all i.e. FlowPrint with 1 bin tracks the total byte and packet counts of the flow without any packet length based separation.

We re-train and evaluate every model on each of the 3 bin configurations for the tasks Application Type Classification and Video Provider Classification (weighted average f1 scores shown in Fig. 3.8). We observe that the f1 scores across the models and tasks generally improve with more bins. However, the performance improvement also depends

on the task complexity. For type classification task, the models improve by less than 2% per addition of a band (the difference is further insignificant for FlowFormers). For video provider classification however, the performance increment is evidently drastic since the task is more challenging and requires fine grained data with binning. On the other hand, Conference Provider Classification (not shown in the figure), has little to no impact on f1 scores by reducing bands as almost all packets exchanged within the one bin ($0 \leq pkt.len \leq 1250$).

Thus, the configuration of FlowPrint can be decided depending upon the NTC task at hand. Higher number of bins would imply higher memory footprint which is especially expensive in programmable switches which have very limited memory. So, this evaluation helps to navigate the bin vs. memory tradeoff to configure FlowPrint parameters and achieve a particular target accuracy for an NTC task.

Time Period Analysis. We now evaluate the impact of the time period for which FlowPrint is collected on each task. We re-train and evaluate FlowFormers (vanilla models omitted for brevity) on FlowPrint collected for 10sec, 20sec and 30sec (the max configuration). The Fig. 3.9 shows the weighted average f1-score of TE-LSTM (top) and TE-CNN across the tasks (x-axis). We note that both models are able to classify application types with about 95 % f1 score with just 10 seconds of data while going up to 97% with 30 seconds. Similarly, the conferencing provider classification results do not vary by much with increasing time as a conference call tends to exhibit similar behaviour over the given time range. However, for video provider classification task, we can observe a significant gain by using FlowPrint collected for a longer duration. This is due to the periodic nature of the flows which repeats at a longer interval (e.g. 16 seconds for Netflix).

Thus, the parameters of FlowPrint i.e. *numbins*, *time duration*, *interval* etc. can be configured depending upon the NTC task, the available compute/memory resources and required performance in terms of classification speed and overall accuracy.

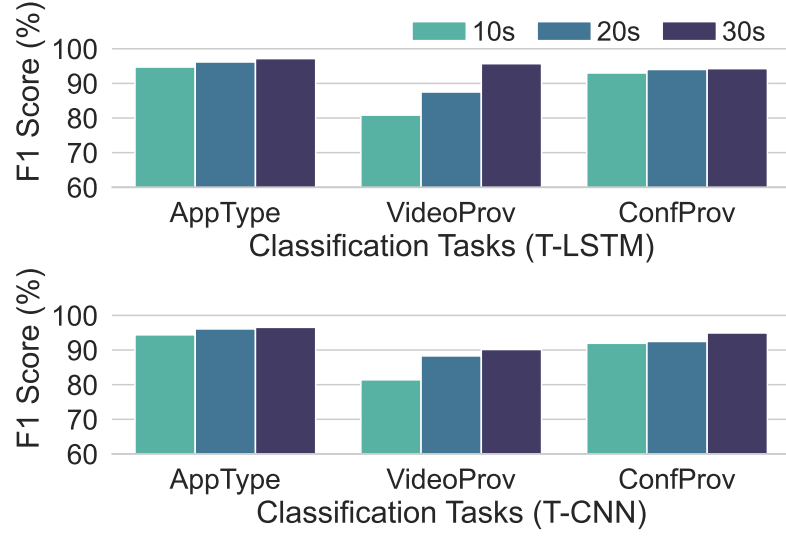


Figure 3.9: FlowPrint Duration Results.

3.5 Conclusion

With diverse applications being used on the internet, *Network Traffic Classification* is becoming important but increasingly challenging due to encryption. Existing approaches either rely on non-encrypted content of traffic or perform post-facto classification of flows. Our work has developed methods for accurately classifying traffic in real-time and at scale by using only the behavioural patterns agnostic to flow content. To this end, we design *FlowPrint*, a data-structure to efficiently capture traffic behaviour that is amenable to implementation in high-speed programmable switches. We further propose the use of transformer-encoders (*FlowFormers*) to outperform existing DL models. Our evaluations show that the combination of *FlowPrint* and *FlowFormers* can classify application type and providers at scale with high accuracies and in real-time.

Chapter 4

Monitoring Video Streaming QoE

Contents

4.1	Introduction	48
4.2	Video Streaming Characteristics	49
4.2.1	<i>VideoMon</i> : Data collection tool	50
4.2.2	On-Demand Video Streaming Characteristics: Netflix	52
4.2.3	Live Video Streaming Characteristics: Twitch and YouTube	56
4.3	Inferring Video Streaming Performance	58
4.3.1	On-demand streaming performance	58
4.3.2	Live streaming performance	65
4.4	Conclusion	73

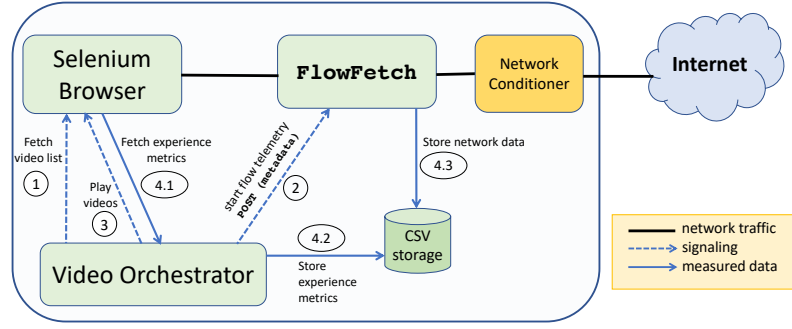
Having developed a robust classification system in the previous chapter, we now shift our focus on monitoring one of the classes of traffic: video streaming. Streaming video is one of the most engaging application that continues to grow, accounting for about 54% of traffic on the Internet according to the Sandvine 2021 report [1]. There are two types of video streaming: on-demand streaming and live streaming. Video on-demand (VoD) streaming consists of users watching content such as movies and TV shows on-demand. Netflix[31] is the most popular on-demand video streaming platform with over 222 million subscribers around the world [151]. On the other hand, in live video streaming, users watch live content that is broadcasted such as a sporting event or gaming streams. YouTube[32] since 2017 allows the larger public to do live streaming, and is widely used for concerts, sporting events, and video games. Twitch (acquired by Amazon) [33] is a popular platform for live streaming video games from individual gamers as well as from tournaments. In this chapter,

we analyse the behaviour of both on-demand and live video streaming applications (Netflix, Twitch and YouTube Live) and develop ML-based and statistical models to estimate video QoE metrics in terms of bitrate, resolution and buffer state with high accuracy.

4.1 Introduction

Given the high traffic volumes and popularity of video streaming platforms, network operators are keen to provide a good experience to their subscribers. However, they currently lack fine-grained visibility into per-stream QoE as existing methods often rely on collecting packet traces and/or HTTP logs which are infeasible from an operator’s vantage point. Further, ensuring good QoE for live video streams is challenging, since clients per-force have small playback buffers (a few seconds at most) to maintain a low latency as the content is being consumed while it is being produced. Even short time-scale network congestion can cause buffer underflow leading to a video stall, causing user frustration. Indeed, consumer tolerance is much lower for live than for on-demand video [152], since they may be paying specifically to watch that event as it happens, and might additionally be missing the moments of climax that their social circle is enjoying and commenting on.

In this chapter, we develop a real-time system to monitor video streaming performance in operator networks. We begin by analysing thousands of traffic traces (annotated with video performance metrics) across video providers such as Netflix (on-demand), YouTube and Twitch (live) to understand their network activity patterns and their correlation with their performance. We then develop network telemetry methods to extract the right granularity of metrics (using flow-level and chunk-level metadata) that are indicative of video performance. Finally, we build statistical and machine-learning-based models to infer video streaming performance in terms of buffer-state, resolution and bitrates. Our models accurately classify buffer-states of with 93% accuracy, predict resolution bins (*e.g.*, LD, SD, HD) with 90+% accuracy and predict buffer stalls with about 90% precision. Our system works at scale and in real-time and was deployed in the field at an ISP serving over 7000 home subscribers. The deployment insights are not part of the scope of this thesis but can be found in our paper [153].

Figure 4.1: *VideoMon* Tool Architecture.

4.2 Video Streaming Characteristics

Video on-demand (VoD) streaming uses HTTP Adaptive Streaming (HAS) technology in which the video client requests media segments (video/audio) from a server. The server hosts a collection of videos (movies and TV shows) which are available in multiple resolutions. Depending upon the network conditions, the client uses adaptive bitrate (ABR) algorithms (which account for buffer health and network conditions) to fetch segments of appropriate resolution/bitrate of the stream. The client fetches multiple segments in the beginning to fill up the large buffer and thereafter tops it up as the playback continues.

In contrast, Live video streaming uses HTTP Live Streaming (HLS) technology to deliver video content which is simultaneously recorded and broadcasted in real-time. The content uploaded by the streamer sequentially passes through ingestion, transcoding, and a delivery service of a content provider before reaching the viewers ([154–156]). More recently, content providers have started to offer “ultra-low” latency live streaming using CMAF [157] containers in which each segment is divided into small chunks (*e.g.*, containing a few frames), the player renders the segment even if its not fully downloaded. Thus, a live streaming client maintains a short buffer of content so as to keep the latency between the streamer and the viewer to a minimum. This increases the likelihood of buffer underflow as network conditions vary, making live videos more prone to QoE impairments such as resolution drop and video stall ([77, 109]).

4.2.1 *VideoMon*: Data collection tool

To construct an accurate profile of video streams, we have developed a tool – *VideoMon* – which automatically plays videos, measures their network activity profile along with client playback metrics, and stores measured records into a pair of CSV files.

VideoMon has three main components, each packaged into a separate docker container: a custom-built network measurement app called *FlowFetch*, a *selenium* browser instance, and a *video orchestrator* application which signals the browser to play videos. There is also an optional network conditioner which uses `tc` linux tool to shape traffic by synthetically changing network conditions in software. Containerizing applications eases deployment of the FlixMon. A shared virtual network interface among the containers ensures that packets flowing through *FlowFetch* originate solely from the browser, eliminating other traffic on the machine where FlixMon runs.

FlowFetch is a tool that we built in *Golang* to record flow-level activity by capturing packets from a network interface. By a flow, we mean a transport-level TCP connection or UDP stream identified by a unique 5-tuple consisting of *source IP*, *source port*, *destination IP*, *destination port* and *protocol*. We believe that collecting per-flow measurements makes our method more practical in real environments (compared to capturing packet traces), thus enabling operators to measure at scale in real-time. For a TCP/UDP flow, the tool records (at a configurable granularity) cumulative byte and packet counts into a CSV. *FlowFetch* is also able to filter flows of interest belonging to certain providers (*e.g.*, Netflix, Twitch and YouTube) by either filtering on server names from SNI/DNS metadata or filtering on flows classified from FlowFormers (presented in previous chapter).

In *FlowFetch*, multiple fully programmable telemetry functions can be associated with a flow. Two functions used in this chapter are (1) ***100ms packet and byte counters*** and (2) ***chunk telemetry***. The first function exports the number of packets and bytes observed on the flow every 100ms. The second function builds upon the chunk detection algorithm proposed in [82] and exports information like *chunkSize* (in bytes and packets) and timestamps such as *chunkRequest*, *chunkBegin* and *chunkEnd* corresponding to the media chunks. We note that the first function is simple and scales better (easier to translate

to programmable hardware primitives) in comparison to second function that requires bidirectional state and tracking multiple timestamps.

We further note that the metrics extracted for video performance estimation is different from metrics extracted for classification (in the previous chapter). FlowPrint’s objective is to capture shape of traffic to classify applications while video performance estimation often requires specific telemetry (*e.g.*, *chunk telemetry*) as per the application (*e.g.*, live video streaming). We demonstrate how the specific network telemetry helps in estimating video performance in §4.3.

For the *video orchestrator*, we have used Selenium client library in Python to interact with a remote Selenium browser instance (*i.e.*, server) for loading and playing Netflix videos. At the beginning of each measurement session, a browser instance (*i.e.*, Firefox or Chrome) is spawned with no cache or cookies saved which loads the Netflix web-page and logs in to the user account by entering credentials (shown by step ① in Fig. 4.1). The tool can be configured in either of two ways to generate a video list: (a) from a fixed set of video links specified in a config file, or (b) by fetching the URLs of recommended videos on the homepage that are updated regularly. Given the list, *VideoMon* starts playing videos sequentially. Prior to playback of each video, the player module signals the *FlowFetch* to start measuring network activity (shown by step ② in Fig. 4.1). Then, the orchestrator signals the browser to load the video and collects the playback metrics (shown by step ③ and ④.1 respectively in Fig. 4.1) – video players offer a hidden menu that can be enabled to track streaming quality stats and diagnose any potential issues. The real-time stats (refresh every second) for audio and video media include the buffering/playing bitrates, buffer health (in seconds and bytes), and the CDN from which the stream is sourced. Additionally, position and duration of the playback, frame statistics (*e.g.*, frame rate and frame drops), and throughput are also provided. The orchestrator stores client playback metrics (every second) into a CSV file (step ④.2) that exists in storage – a shared volume among the orchestrator and Flowfetch containers. Simultaneously, Flowfetch stores the network activity into another co-located CSV (step ④.3).

The dataset of video streams collected from *VideoMon* consists of two CSVs for each video stream being played for at least 5 minutes. One containing the client-side metrics

Table 4.1: VideoMon Dataset

Provider	Number of streams
Netflix	2639
Twitch Live	2587
YouTube Live	1430

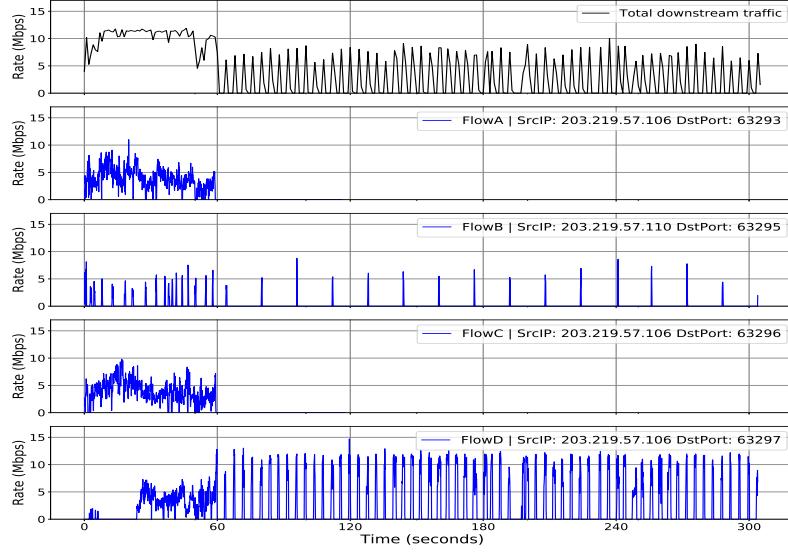


Figure 4.2: Network profile of flows in a typical Netflix video stream.

(indicating session quality stats) and one containing the network telemetry. In this chapter, we consider three video streaming providers: Netflix (on-demand streaming), YouTube and Twitch (live streaming). Tab. 4.1 shows the distribution of the dataset across providers. In total, we have collected over 500 hours of video playback in different network conditions (varying the bandwidth from 100kbps to 20Mbps) imposed by the conditioner module within *VideoMon*.

4.2.2 On-Demand Video Streaming Characteristics: Netflix

We now present our analysis into the dataset we collected by looking at a few exemplary video streams across providers. We begin by analysing Netflix video streams followed by the analysis of live video streams from YouTube and Twitch.

Fig. 4.2 illustrates a time-trace of network activity measured for a representative Netflix video stream played for 5 minutes with no interruption. The top subplot shows in black lines the total downstream traffic profile for this stream, and the four subplots below in

blue lines show downstream traffic profile of each TCP flow associated with this stream. We observe that the Netflix client established four parallel TCP flows to start the video, three of them come from Netflix server 203.219.57.106 and one from 203.219.57.110. All four TCP flows actively transferred content for first 60 seconds. Thereafter, two flows (A,C) became inactive (*i.e.*, idle) for a minute before being terminated by the client (*i.e.*, TCP FIN). It is seen that the remaining two active flows (B,D) changed their pattern of activity – *FlowB* has small spikes occurring every 16 seconds and *flowD* has large spikes occurring every 4 seconds.

Let us correlate this with metrics offered by the Netflix client application for the same video stream shown in Fig. 4.3. We show in Fig. 4.3(a) and 4.3(b) the buffer health of audio and video respectively which is measured in terms of: (a) volume in bytes (shown by solid blue lines and left y-axis) and (b) duration in seconds (shown by dashed red lines and right y-axis). We observe that the buffer health in seconds for both audio and video ramps up during the first 60 seconds of playback, till it reaches to a saturation level at 240 seconds of buffered content – thereafter, this level is consistently maintained by periodic filling. Note that the audio and video buffers are replenished every 16 and 4 seconds respectively, suggesting a direct contribution from the periodic spikes in network activity (observed in *FlowB* and *FlowD*).

Netflix client interface reports a metric called “throughput” which is an estimate of bandwidth available for the video stream. Fig. 4.3(c) shows the throughput (in Mbps, solid blue lines, on the left y-axis) and the buffering-bitrate of video (in Kbps, dashed red lines, on the right y-axis). We observe that the video starts at a low-quality bitrate 950Kbps, switches to higher bitrate 1330Kbps after 2 seconds, and jumps to its highest bitrate 2050Kbps after a second. Note that it stays at this highest bitrate for the rest of video playback even though far more bandwidth is available. Additionally, we note in Fig. 4.3(b) that the video buffer health in volume is variable while the buffer in seconds and the buffering bitrate are both consistent. This is because of variable bitrate encoding used by Netflix to process the videos where each video chunk is different in size depending on scene complexity. In contrast, buffer health volume for audio in Fig. 4.3(a) stays at 3MB with periodic bumps to 3.2MB – this indicates a constant bitrate encoding used for audio content and bumps occur when a new audio chunk is downloaded and an old one is

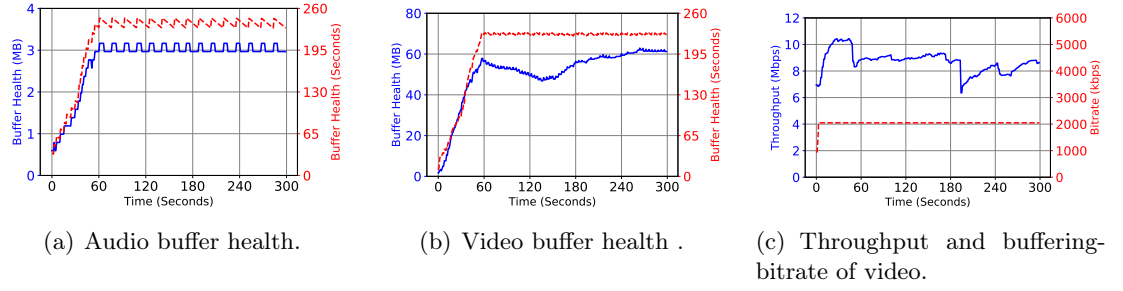


Figure 4.3: Client metrics of a Netflix stream.

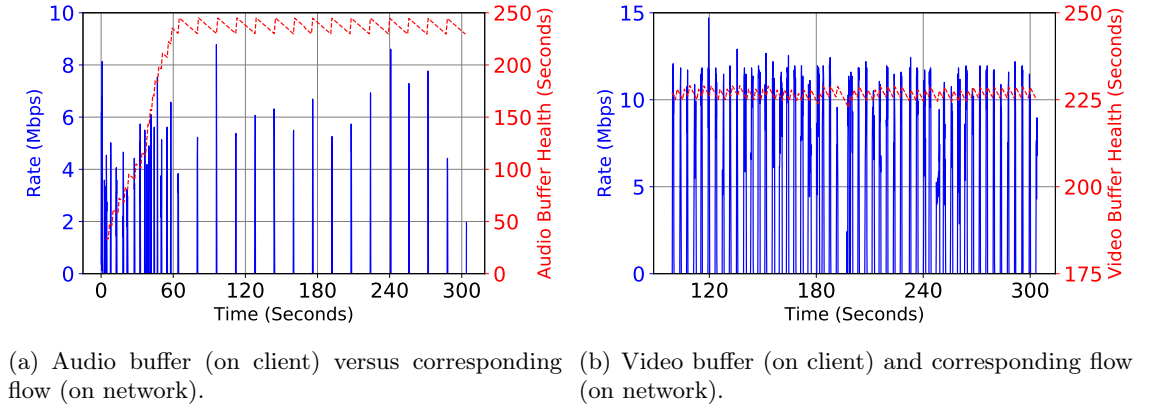


Figure 4.4: Correlation of network activity and client behavior.

discarded from the buffer. For audio, we observed (not shown in the Fig. 4.3(c)) a constant bitrate of 96Kbps throughout the playback.

Having analysed streaming behaviour on network and client individually, we analyse their correlation. We observed two distinct phases of video streaming: (a) the first 60 seconds of *buffering*, (b) followed by *stable* buffer maintenance. In the buffering phase, the client aggressively transferred contents at a maximum rate possible using four concurrent flows and then in the stable phase it transferred chunks of data periodically to replenish the buffer using only two flows.

Of the two flows active in stable phase, *FlowB* (with a spike periodicity of 16 seconds) displays a strong correlation between the spikes of its network activity and the replenishing audio buffer levels on the client, as shown in Fig. 4.4(a). This suggests that the TCP flow was used to transfer audio content right from the beginning of the stream. Isolating content chunks of this flow, we found that the average chunk size was 213KB with a

standard deviation of 3KB (1.4%). Every chunk transfer corresponds to an increase of 16 seconds in the client buffer level. Considering the fact that each chunk transferred 16 seconds (indicated by both periodicity and increase in buffer level) of audio and the buffering bitrate of audio was 96Kbps, the size of audio chunk is expected to be 192KB which is very close to our computed chunk size of 213KB which includes the packet headers. Additionally, we note that for this specific flow, the server IP address differs from other flows (as shown in Fig. 4.2) and the Netflix client statistics also indicate that audio comes from a different CDN endpoint.

Further, *FlowD* (with a spike periodicity of 4 seconds) during the stable phase, displays a similar correlation between its network activity and the client buffer health of video, as shown in Fig. 4.4(b). The chunks of this flow have an average size of 1.15MB and a standard deviation of 312KB (27%). With each chunk constituting 4 seconds of video content and the video bitrate on client measured as 2050Kbps, the actual chunk size is expected to be 1.00MB which is close to the computed average chunk size while accounting for packet headers. Additionally, a high deviation in video chunks size also suggests that video is encoded using variable bitrate (in contrast, audio has a constant bitrate).

Trickplay: Having understood the streaming behavior during a normal playback (with no interruption), let us now analyze the behavior of Netflix streams during trickplay events. Trickplay occurs when the user watching the video decides to play another segment far from current seek position by performing actions such as fast-forward, or rewind. A trickplay is performed either within the buffered content (*e.g.*, forward 10 seconds to skip a scene) or outside the buffered content (*e.g.*, random seek to unbuffered point). In the former case (within buffer), our observations show that the Netflix client uses existing TCP flows to fetch the additional content filling up the buffer up to 240 seconds. However, in the latter case, the client discards the current buffer and existing flows, and starts a new set of flows to fetch content from the point of trickplay. This means that trickplay outside the buffer is very similar to the start of a new video stream, making it difficult to determine whether the client has started a new video (say next episode in a series) or has performed a trickplay. For this reason, we consider a trickplay event equivalent to start of a new video stream and compute our experience metrics accordingly. Additionally, we note that for a stream in the stable phase, trickplay results in transitioning back to the buffering phase until the

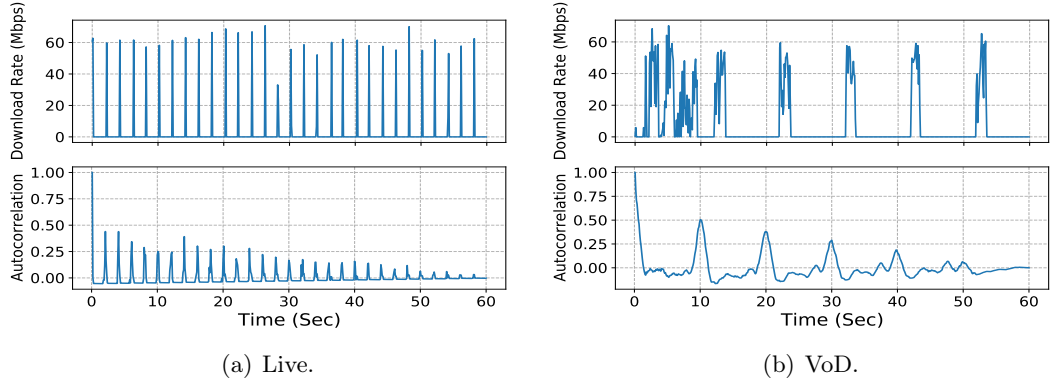


Figure 4.5: Download activity and its auto-correlation of Twitch streaming: (a) Live, and (b) VoD.

buffer is replenished. In §4.3.1, we will distinguish trickplay from network congestion that can cause a stream to transition into the buffering phase.

4.2.3 Live Video Streaming Characteristics: Twitch and YouTube

Fig. 4.5 shows the client’s network behavior (download rate collected at 100 ms granularity) of live and VoD streams (both from Twitch). It can be clearly seen how the two time-trace profiles differ. The live streaming client downloads video segments every two seconds. In contrast, the VoD client begins by downloading multiple segments to fill up a long buffer and then fetches subsequent segments every ten seconds (very similar behaviour to Netflix as shown above).

To better understand the delivery mechanism of live videos, we collected the playback data from the video client such as latency modes, buffer sizes and resolutions (using browser automation tool described in §4.2.1). We also used the network debugging tools available in Google Chrome [158] browser and Wireshark[48] (configured to decrypt SSL) to gain insights into protocols being used, patterns of the requests made for content and manifest files, their periodicity, and the available latency modes as shown in Table 4.2.

Twitch: The Twitch VoD client uses HTTP/2 and fetches combined audio and video media segments (with extension `.ts`) from a server with the SNI `vod-secure.twitch.com`. Twitch live, however, uses HTTP/1.1, and on the same TCP flow fetches separate audio

Table 4.2: Fetch mechanisms of Twitch and YouTube video streaming.

Provider	Type	Protocol	Request for Manifest	Frequency	Latency modes
Twitch	VoD	HTTP/2	Once	10s	-
	Live	HTTP/1.1	Periodic (different flow)	2/4s	Low, Normal
YouTube	VoD	HTTP/2 + QUIC	Once	5-10s	-
	Live	HTTP/2 + QUIC	Manifestless	1/2/5s	Ultra Low, Low

and video segments from a server endpoint with the SNI *video-edge*.abs.hls.ttv.net*. This obfuscated URL pattern is indicative of CDNs and edge compute usage. Additionally, it requests manifest updates from a different server endpoint with name prefix “*video-weaver*” which also seems to be distributed using CDNs. The periodicity of segment requests is around 10 seconds for VoD and 2 seconds for live streams [154], corroborating our earlier observation in this section. Additionally, Twitch offers two modes of latency, *i.e.*, Low and Normal. We note that major differences between these modes include a) client buffer capacity – it is higher (*i.e.*, 6-8 seconds) for Normal when compared to Low (*i.e.*, 2-4 seconds) and b) use of CMAF media containers – it is predominant in Low but rarely used in Normal.

YouTube: YouTube primarily uses HTTP/2 over QUIC [159] for both VoD and live streams, fetching audio and video segments separately on multiple flows (usually two in case of QUIC). These flows are established to the server endpoint with name matching pattern “**.googlevideo.com*”. If QUIC protocol is disabled or not supported by the browser (*e.g.*, Firefox, Edge, or Safari), YouTube uses HTTP/1.1 and multiple TCP flows to fetch the video content. YouTube live operates in *manifestless* mode (as indicated by the client playback statistics) and thus manifest files are not transferred on the network. In case of VoD, after filling up the initial buffer, the client typically tops it up at a periodicity of 5-10 seconds. We observed that the buffer size and periodicity can vary depending on resolution selected and network conditions. In case of live streaming, however, the buffer health and periodicity of content fetch will depend on the latency mode of the video. There are three modes of latency for YouTube live including Ultra Low (buffer health: 2-5 sec, periodicity: 1 sec, uses CMAF media containers), Low (buffer health: 8-12 sec, periodicity: 2 sec), and Normal (buffer health: 30 sec, periodicity: 5 sec). We found that live streaming in normal latency mode displays the same network behavior as VoD, and hence is excluded from our

study – this mode of streaming is not as sensitive as the other two modes.

The analysis of live video streaming across Twitch and YouTube indicated that (a) there’s very little buffering of the video stream (no separate buffering and stable phase like with Netflix), (b) each media segment was delivered periodically with the lowest transcoding time possible indicating that each segment is tracked on network (using chunk telemetry described in 4.2.1) can indicate buffer stability and resolution of the video. In the next section, we look at estimating the video performance metrics from network telemetry in detail.

4.3 Inferring Video Streaming Performance

In the last section, we looked at video streaming behaviour and correlated it with network activity. Now, we will develop algorithms to infer video streaming performance from the network activity.

A video streaming client tries to load its buffers quickly and play the highest quality under good network conditions. When the network deteriorates however, the client needs to adjust the quality of playback using adaptive bitrate algorithms [34, 160]. It assesses the network conditions (typically using estimates of available bandwidth) and switches to lower bitrates. If the network conditions do not improve over time, it may result in a buffer depletion and eventually lead to stall. Our goal in this section is to predict the changes in video behaviour when network conditions worsen and estimate metrics that can quantify video performance. We begin by developing performance prediction algorithms for on-demand streaming followed by live video streaming.

4.3.1 On-demand streaming performance

For on-demand video streaming, we first build a classifier to detect buffer-phase of the Netflix video stream. Subsequently, we identify 3 key metrics that estimate video performance and describe methods to compute them. Finally, we show examples of detecting on-demand performance degradation. For all the tasks, we rely only on the per-flow byte

counter telemetry collected every 100ms as described in 4.2.1.

Buffer-phase classification

We observed in §4.2.2 that on-demand streaming has two distinct phases: buffering and stable. In buffering phase, the client continuously fetches segments and in stable phase it periodically fetches segments. A client will go into buffering phase only in a few situations: (a) at the beginning of the stream to fill up the buffer, (b) when the network conditions deteriorate and client starts losing the buffer i.e. playback is faster than buffer fill-up, and (c) when the user performs trickplay i.e. fast forward/backward to an unbuffered point in the video. Thus, knowledge about the buffer phase is useful to model the video behaviour and infer its performance. To do so, we build machine learning-based model to classify the phase (*i.e.*, buffering or stable) of a video streaming playback by using several waveform attributes (explained next).

Data Labeling: Each video streaming instance in our Netflix dataset is broken into separate windows of each 1-minute duration. We label a window of individual TCP flows associated with a stream using the client buffer health (in seconds) of that stream. For each window, we consider three measures namely the average, the first, and the last value of buffer health in that window. If both the average and last buffer values are greater than 220 seconds, then we label it as “stable”. If both the average and the last buffer values are less than 220 seconds but greater than the first buffer value, then we label the window as “buffering”. Otherwise (*e.g.*, transition between phases), we discard the window and do not use for training of our model.

Attributes: For each flow active during a window, we compute two sets of attributes. Our first set of attributes include: (a) **totalVolume** – relatively high during buffering phase; (b) **burstiness** (*i.e.*, μ/σ) of flow rate – captures the spike patterns (high during stable phase); (c) **zeroFrac**, fraction of time the flow is idle (*i.e.*, transferred zero bytes) – this attribute is expected to be smaller in the buffering phase; (d) **zeroCross**, count of zero crossing in the zero-mean flow profile (*i.e.*, $[x-\mu]$) – this attribute is expected to be high in the buffering phase due to high activity of flows; and (e) **maxZeroRun**, maximum

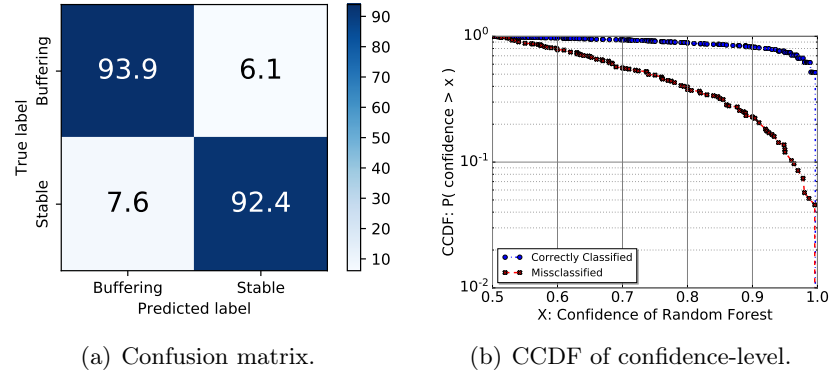


Figure 4.6: Performance of phase classification: (a) confusion matrix, and (b) CCDF of confidence-level.

duration of being continuously idle – this attribute is relatively higher for certain flows (*e.g.*, aging out or waiting for next transfer) in the buffering phase.

Our next set of attributes are computed by isolating chunks of transfers from the flow profile. Each chunk in a flow is isolated by three successive data points of zero (*i.e.*, 300 ms idle after a transfer). Our five attributes computed from chunks are: (f) **chunksCount**; (g,h) **average** and **standard-deviation** of chunk sizes; (i,j) **average** and **mode** of chunks inter-arrival time. In the buffering phase, the flow would have less chunks, lower inter-chunk time, and higher volume in each chunk compared to the stable phase. In total, for each flow in a window, we have 10 attributes computed (considering just the waveform profile, independent of available bandwidth) for each training instance (*i.e.*, 1-min window of a TCP flow).

Classification Results: We used the RandomForest ML algorithm available in Python scikit-learn library. We configured our model to use 100 estimators which are used to predict the output along with a confidence-level of the model. We split our labeled data of 12,340 instances into training (80%) and testing (20%) sets. We evaluated the performance of our classifier using the testing set and obtained a total accuracy of 93.15%, precision of 94.5% and recall of 92.5%. We show in Fig. 4.6(a) the confusion matrix of our classifier. It is seen that 93.9% of buffering and 92.4% of stable instances are correctly classified. Fig. 4.6(b) illustrates the CCDF of the model confidence for both correctly and incorrectly classified instances. The average confidence of our model is greater 94% for correct clas-

sification while it is less than 75% for incorrect classification – setting a threshold of 80% on the confidence-level would improve the performance of our classification.

Use of Classification: For each TCP flow associated with a streaming session, we call our trained model to predict its phase of video playback. As explained earlier, multiple flows are expected especially at the beginning of a stream. We perform majority voting on outputs of the classifier for individual flows to determine phase of the video stream. In case we have a tie, we pick the phase with maximum sum confidence of the model. In addition to the classification output, the count of flows in the stable phase (*i.e.*, two flows) can be used to check (validate) the phase detection. This cross-check method also helps detect the presence of concurrent video streams for a household to discount them out of the analysis – having more than two Netflix flows for a household IP address, while the model indicates the stable phase (with a high confidence), likely suggests parallel playback streams.

Computing Performance Metrics

We now identify three key metrics that together help us infer Netflix performance. The first two are metrics directly related to performance, and the third one is used to deduce events affecting performance.

1) Buffer Fill-Time: As explained in §4.2.2 (by Fig. 4.3(a) and 4.3(b)), Netflix streams tend to fill up to 240 seconds worth of audio and video to enter into the stable phase – a shorter buffer fill-time implies a better network condition and hence a good user experience. Once the stream starts its stable phase, we begin by measuring *bufferingStartTime* when the first TCP flow of the stream was established. We then identify *bufferingOnly* flows – those that were active only during the buffering phase, go inactive upon the completion of buffering, and are terminated after one minute of inactivity (*FlowA* and *FlowC* shown in Fig. 4.2). We, next, compute *bufferingEndTime* as the latest time when any *bufferingOnly* flow was last seen active (ignoring activity during connection termination (*e.g.*, TCP FIN)). Lastly, the buffer fill-time is obtained by subtracting *bufferingEndTime* and *bufferingStartTime*.

Fill-Time Results: To quantify the accuracy of computing buffer fill-time, we use our client data of video buffer health (in seconds) as ground-truth. Results show that our method achieved 10% relative error for 75% of streams in our dataset – the average error for all streams was 20%. We observe that in some cases a TCP flow starts in the buffering phase and (unexpectedly) continues carrying traffic in the stable phase for some time after which it goes idle and terminates. This will result in our predictions of buffer fill-time to be larger than its true value thereby underestimating the user experience.

2) Bitrate: A video playing at a higher bitrate brings a better experience to the user. We estimate the average bitrate of Netflix streams using the following heuristics. During the stable phase, Netflix replaces the playback buffer by periodically fetching the video and audio chunks. This means that over a sufficiently large window (say, 30 seconds), the total volume transferred on the network would be equal to the playback buffer of the window size (*i.e.*, 30 seconds) since the client tends to maintain the buffer at a constant value (*i.e.*, 240 seconds). Therefore, the average bitrate of the stable stream is computed by dividing the volume transferred over the window by the window length. During the buffering phase, Netflix client downloads data for the buffer-fill-time and an additional 240 seconds (*i.e.*, the level maintained during the stable phase). Thus, the average bitrate of the buffering stream is computed by dividing total volume downloaded by sum of buffer fill-time and 240 seconds.

By tracking the average bitrate, we are able to determine the bitrate switches (*i.e.*, rising or falling bitrate) in the stable phase. As discussed earlier, there are a range of bitrates available for each video. For example, title “Eternal Love” was sequentially played at 490, 750, 1100, 1620, 2370, and 3480Kbps during a session in our dataset. We note that Netflix makes bitrates available in a non-linear fashion – bitrate values step up/down by a factor of ~ 1.5 to their next/previous level (*e.g.*, 490×1.5 approximately indicates the next bitrate level 750). We use this pattern to detect a bitrate switch if the measured average bitrate changes by a factor of 1.5 or more.

Bitrate Results: We evaluated the accuracy of our bitrate estimation using the client data as ground-truth. For the average bitrate in buffering phase, our estimation resulted in a mean absolute error of 158Kbps and an average relative error of 10%. The estimation

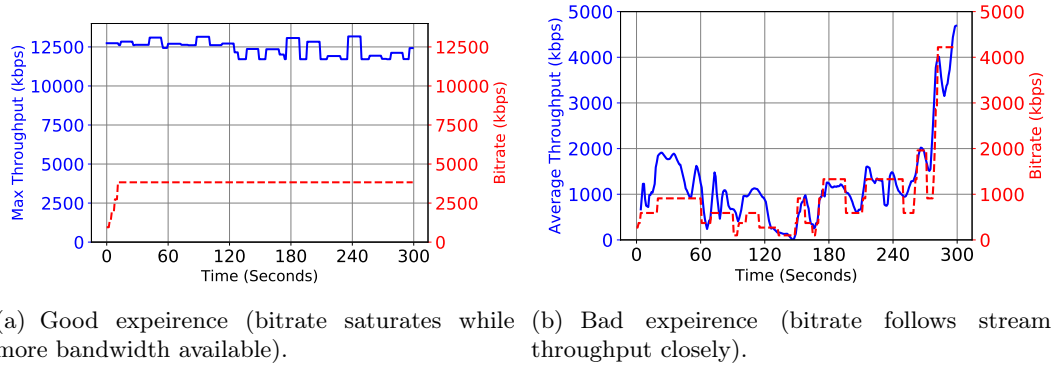


Figure 4.7: Inferring user experience considering throughput.

errors for average bitrate in stable phase, were 297Kbps and 18% respectively. These errors arise mainly due to the fact that Netflix client seems to report an average bitrate of the movie but due to variable bitrate encoding, each scene is transferred in different sizes of chunks, hence a slightly different bitrate is measured on the network. Nonetheless, we note that detection of bitrate switch events will be accurate since the average bitrate would change by more than a factor of 1.5 in case of bitrate upgrade/downgrade.

3) Throughput: We use the aggregate throughput measurements of the stream (obtained by summing up the throughput of all TCP flows involved) to detect experience events listed below. To do so, we derive two signals over a sliding window (say, 5 seconds) of the aggregate throughput: (a) max throughput, and (b) average throughput – note that the flow throughput is measured every 100ms.

Max bitrate playback. For a video stream, if the gap between the max throughput and the computed average bitrate is significantly high (say, twice the bitrate being played), then it implies that the client is not using the available bandwidth as it is currently playing at its maximum possible bitrate, as shown in Fig. 4.7(a), indicating a good experience.

Bitrate variations during buffering. If the max throughput measured is relatively close to the bitrate ranges of Netflix (up to 5000 kbps) and is highly varying, it indicates possible bitrate switching events. In this case, the actual bitrate strongly correlates with the average throughput signal, as shown in Fig. 4.7(b). The fluctuating average throughput with high standard deviation (*i.e.*, $\geq 20\%$) causes the stream to switch bitrates and becomes unstable, indicating a bad experience.

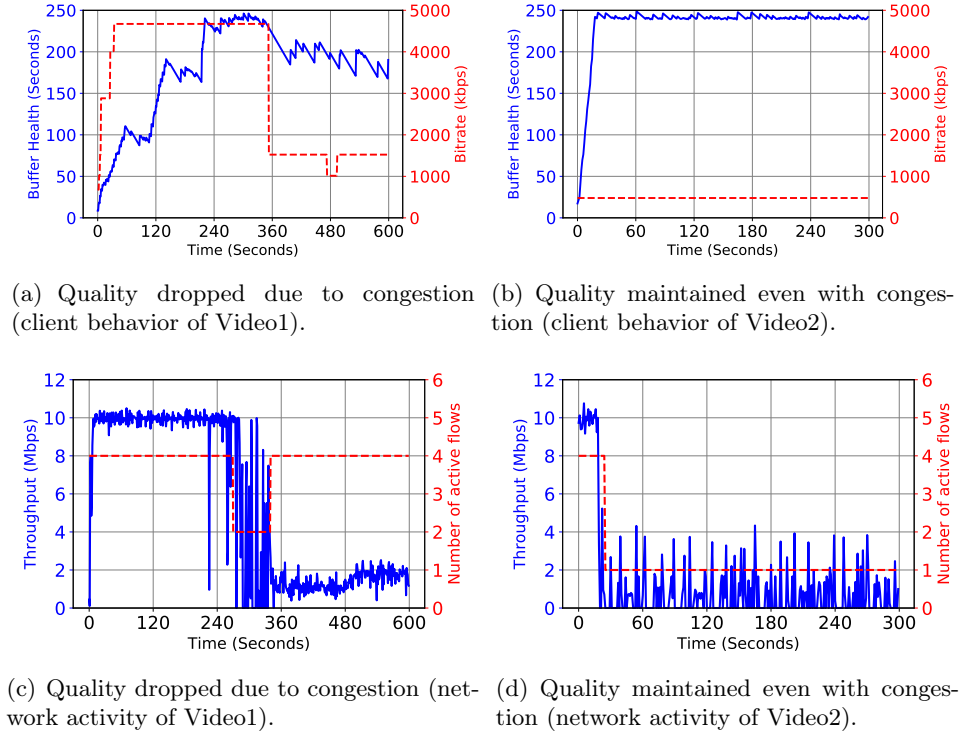


Figure 4.8: Detecting video quality degradation for users.

Detecting Buffer Depletion and Quality Degradation

We now detect bad experiences in terms of buffer health and video quality using the metrics described above. To illustrate our method, we conducted an experiment in our lab whereby the available network bandwidth was capped at 10 Mbps. We first played a Netflix video on a machine, and one minute after the video went into the stable phase (*i.e.*, 240 seconds of buffer filled on client) we introduced UDP downstream traffic (*i.e.*, CBR at 8Mbps using **iperf** tool) to congest the link. For videos, we chose two Netflix movies – Season 3 Episode 2 of “Deadly 60” with high quality bitrate available up to 4672Kbps (Video1), and Season 1 Episode 1 of “How I Met Your Mother” with a maximum bitrate of 478Kbps (Video2). Fig. 4.8 shows client behavior (top plots) and network activity (bottom plots) for the two videos.

Considering Fig. 4.8(a) for Video1, it is seen that the stream started at 679Kbps bitrate (dashed red lines), quickly switched up, and reached to the highest possible value 4672Kbps in 30 seconds. It continued to play at this bitrate and entered into the stable phase (at second 270) where only two flows remained active, as shown in Fig. 4.8(c), and the buffer

health (solid blue lines) reached to its peak value of 240 seconds. Upon commencement of congestion (at second 340), we observe that the buffer started depleting followed by a bitrate drop to 1523Kbps. Moving to the network activity in Fig. 4.8(c), we observe that two new flows spawned, the stream went to the buffering phase, and the network throughput fell below 2Mbps. The change of phase, combined with a drop in throughput, indicates that the client experiences a buffer depletion – a bad experience. Using our method, we detected a phase transition (into buffering) at second 360 and deduced bitrate from the average throughput (as explained earlier in Fig. 4.7(b)), ranging from 900Kbps to 2160Kbps. This estimate shows a significant drop (*i.e.*, more than a factor of 1.5) from the previously measured average stable bitrate (*i.e.*, 3955Kbps). Additionally during the second buffering phase, we observe a varying average throughput with the mean 1.48Mbps and the standard-deviation 512Kbps (*i.e.*, 35% of mean) indicating a fluctuating bitrate on the client. We note that although a transition from stable to buffering can result from a trickplay (discussed in §4.2.2) we do not detect a bad experience since no change in max throughput is observed.

4.3.2 Live streaming performance

The QoE of a live video stream can be captured by two major metrics, namely, video quality and buffer stalls. Video quality can be measured using: (a) resolution of the video, (b) bitrate (number of bits transferred per sec), and (c) more complex perceptual metrics like MOSS [161] and VMAF [162, 163]. In this subsection, we develop a method to estimate the resolution of the playback video since the ground-truth data is available across the three providers. Also, resolution is typically reported (or available to select) in popular live streaming services. In addition to video resolution, we devise a method to detect the presence of buffer stalls which are more likely to occur in case of live streaming (compared to VoD), since a smaller buffer size is maintained on the client to reduce the latency between the producer and the viewer. In what follows, we present our analysis of data collected from the network consisting of audio/video segments versus metrics recorded on the client. Subsequently, we develop methods that estimate video resolution and detect buffer stalls.

Network-Level Measurement

To estimate QoE metrics for the live stream, we need to estimate the size of the media segments being fetched from the packets traversing on the network flow. The data downloaded between two consecutive requests is a good estimate of the size of media segments. We refer to this estimate as a *chunk*. Hence, we use the term *segment* for a unit of media requested by the player, while *chunk* denotes a corresponding unit of data observed on the network (demarcated by the request packets). We build upon existing network chunk-detection algorithms [82, 97] to isolate the video chunks fetched by the live player. In short, the algorithm identifies the start of a chunk by an upstream request packet, and aggregates all subsequent downstream packets to “form” the chunk. For each chunk, we extract the following features: *requestTime*, *i.e.*, the timestamp of the request packet, *requestPacketLength*, *chunkStartTime* and *chunkEndTime*, *i.e.*, timestamps of the first and the last downstream packets following the request (subtracting these two timestamps gives *chunkDownloadTime*), and lastly *chunkPackets* and *chunkBytes*, *i.e.*, total count and volume of downstream packets corresponding to the chunk.

During the playback of a live video stream, the *chunk telemetry function* operates on a per-flow basis in our FlowFetch tool, and exports the features mentioned above for every chunk observed on five-tuple flow(s) carrying the video. We note that chunk telemetry is more expensive (in both compute and memory) than simply exporting 100ms counters as done for Netflix. However, since the volume for Netflix is very high we rely on simpler function whereas live video is comparatively low and hence we can execute these functions in operator networks.

As earlier mentioned in §4.2.1 we collect resolution and buffer health metrics reported by the video client. In what follows, we correlate and analyze the chunk data obtained from the network and client metrics to train our models for estimating resolution and detecting the presence of buffer stalls.

Table 4.3: Dataset resolution distribution.

Provider	LD	SD	HD	SOURCE
Twitch	17%	32%	34%	17%
YouTube	36%	36%	28%	-

Estimating Resolution

The resolution of a live video stream indicates the frame size of video playback – it may also sometimes indicate the rate of frames being played. For example, a resolution of 720p60 means the frame size is 1280×720 pixels while playing 60 frames per sec. For a given fixed duration video segment, the video segment size (and hence our corresponding chunk estimate) usually increases in higher resolutions as more bits need to be packed into the segment.

We analysed the live video streams played using our tool for both content providers to better understand the distribution of video segment sizes across various resolutions. We also consider four bins of resolution, namely Low Definition (LD), Standard Definition (SD), High Definition (HD), and Source (originally uploaded video with no compression, only available in Twitch) – Table 4.3 shows the distribution of streams across these bins. The bins are mapped as follows, anything less than 360p is LD, 360p and 480p belong to SD, 720p and beyond belongs to HD. If the client tags a resolution (usually 720p or 1080p) as Source, it is binned into Source. Such binning serves two purposes: (a) it accounts for a similar visual experience for a user in neighbouring resolutions, and (b) it provides a consistent way to analyze across providers. Fig. 4.9 shows the distribution of chunk sizes versus resolutions, and will be further explained next. We estimate the resolution in two steps: (a) first, separating chunks corresponding to video segments, and (b) next developing an ML-based model to map the chunk size to resolution.

Separation of video chunks Network flows corresponding to a live stream can carry chunks of data that correspond to any of video segments, audio segments, or manifest files, and hence the video component needs to be separated out to estimate its resolution. Moreover, our telemetry engine also picks up some other small stray chunks which are not actual HTTP GET responses. We employ a simple method to separate the stray chunks

by ignoring a chunk less than a threshold (say, 10KB) – both audio and video segments are larger than 10KB across content providers. However, the method to isolate video segments depends on the provider – it can be developed by analysing a few examples of streaming sessions and/or by decrypting SSL connections and analysing the request URLs.

Twitch usually streams both audio and video segments on the same 5-tuple flow for live video streaming, and manifest files are fetched in a separate flow. We observed that audio is encoded in a fixed bitrate, and thus its chunk size is consistent (≈ 35 KB). Further, Twitch video chunks of the lowest available bitrate (160p) have a mean of 76 KB. Thus, video chunk identification is fairly simple for Twitch live streams, *i.e.*, all chunks more than 40 KB in size.

YouTube live usually uses multiple TCP/QUIC flows to stream the content consisting of audio and video segments – Youtube operates manifest-less. As indicated in Table 4.2, Youtube live operates in two modes, *i.e.*, Low Latency (LL) with 2 sec periodicity of content fetch, and Ultra Low Latency (ULL) with 1 sec periodicity. We found that the audio segments have a fixed bitrate (*i.e.*, size per second is relatively constant) regardless of the latency mode – audio chunk size of 28 – 34 KB for the ULL mode, and 56 – 68 KB for the LL mode. However, separating the video chunks is still nontrivial as video chunks of 144p and 240p sometimes tend to be smaller in size than the audio chunks.

To separate the audio chunks, authors of [82] used the *requestPacketLength* as they observed that the audio segment requests were always smaller than the video requests. We used this method for TCP flows but found it inaccurate in UDP QUIC flows as the audio segment requests are sometimes larger than video segment requests due to header compression. Further, QUIC flows pose additional challenges, especially for live video streams. Because of bi-directional stream support available in HTTP/2 + QUIC, a request for a media segment can be sent before the previous segment completely downloads. Since our chunk telemetry function relies on the request packets to mark the start of chunks, the *chunk* sizes computed by the network telemetry function differ from the actual size of media *segments*. For this reason, we cannot accurately capture individual media segments for YouTube videos delivered over QUIC flows. Thus, performance inferencing for YouTube QUIC video streams is beyond the scope of this chapter.

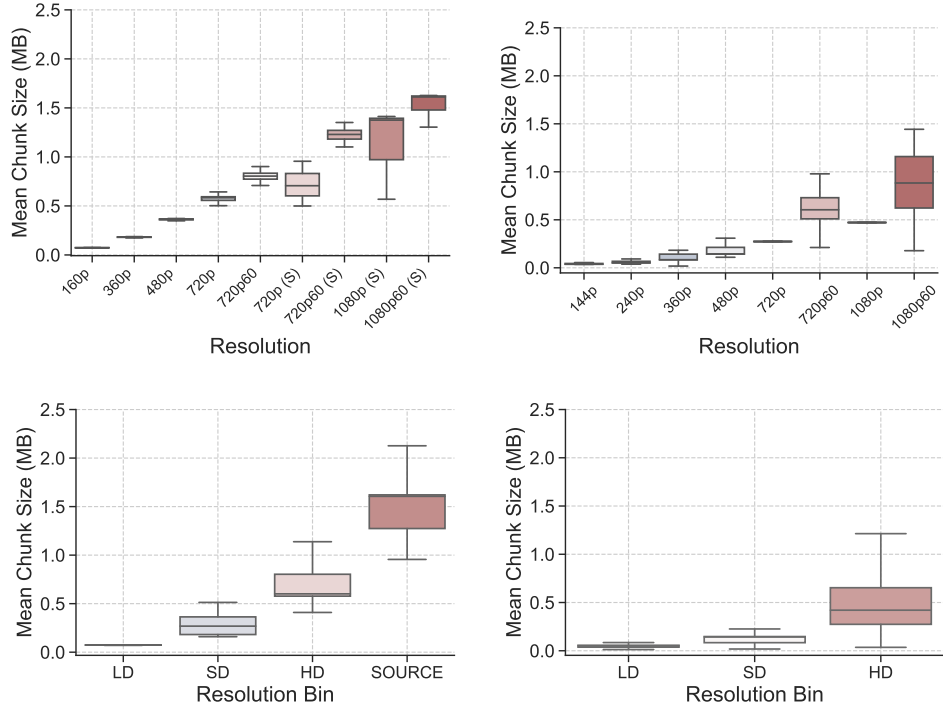


Figure 4.9: Chunk size versus resolution for Twitch (left), and YouTube (right).

Table 4.4: Resolution prediction accuracy.

Provider	Resolution	Resolution bin
Twitch	90.64%	97.62%
YouTube	75.17%	90.08%

Analysis and inference After identifying the chunks corresponding to the video segments for each provider, we now look at the distribution of chunk sizes across various resolutions at which the video is played. Fig. 4.9 shows box plots of mean (video) chunk size in MB versus the resolution (*i.e.*, actual value or binned value) in categorical values. Note that the mean chunk size is computed for individual video streams of duration 2-5 minutes. Further, the label (*S*) on the X-axis indicates a Source resolution.

Looking at Fig. 4.9, we make the following observations: (a) video chunk size increases with resolution across both the providers; (b) chunk sizes are less spread in lower resolutions; and (c) chunk sizes of various transcoded resolutions (*i.e.*, not the source resolution) do not overlap much with each other for Twitch. However, the overlap of neighboring resolutions becomes more evident in YouTube streams. Such overlaps make it challenging to estimate the resolution.

We use the Random Forest algorithm for mapping chunk sizes to the resolution of playback as it creates overlapping decision boundaries using multiple trees and then uses majority voting to estimate the best possible resolution by learning the distribution from the training data. Using the mean chunk size as an input feature, we trained two models, *i.e.*, one estimating the exact resolution and the other estimating the resolution bin. We perform 5-fold cross-validation on the dataset with 80-20 train-test split, and our results are shown in Table 4.4.

Predicting Buffer Stalls

Buffer stalls occur when the playback buffer is emptied out because the video segments cannot be fetched in time. This QoE metric is vital for live streams, which typically maintain short buffers (4 seconds for Twitch LL and Youtube ULL modes). Even for a few seconds, network instability can cause the live buffer to deplete, leading to a stall causing viewer frustration.

To better understand the live buffering mechanism across the three providers, we collect data for live video streams ($\approx 5min$ per session) while using the network conditioner component of our tool to impose synthetic bandwidth caps. We created a commonly occurring situation in a household wherein cross-traffic (browsing/e-mail etc.) is introduced for a few seconds while a live stream is going on. To do so, the tool starts with a cap of 10 Mbps (typical household bandwidth) and then, after every 30 seconds, caps the download/upload bandwidth at a random value (between 100 Kbps to 2 Mbps) for a duration of 10 seconds (mimicking the congestion due to cross traffic). Live videos being played in the browser are accordingly affected by these bandwidth switches. We found that if videos are played at *Auto* resolution, then the clients avoid stalls most of the time by switching to lower resolutions. Therefore, we forced the video streams to play at one of the HD resolutions (1080p or 720p) to gather data of buffer stall events. In total, we collected more than 250 video streams across the three providers. On average, 15% and 6% of the playback time were spent in the stall state for Twitch and YouTube.

Fig. 4.10 shows the dynamics of buffer health for a representative stream in our dataset.

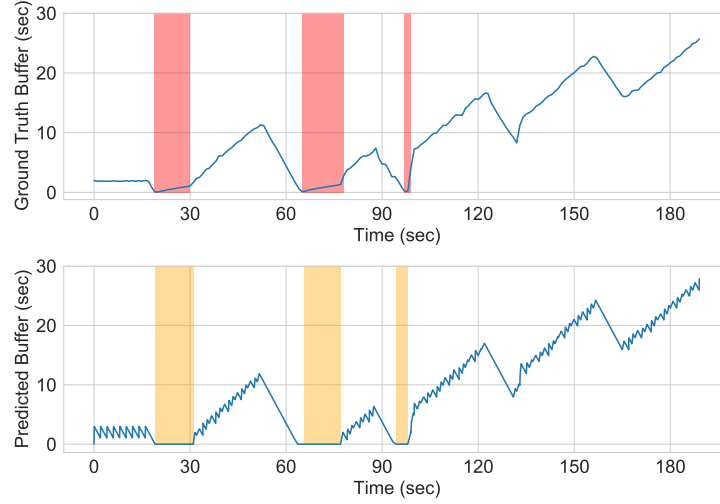


Figure 4.10: Time-trace of buffer health value: ground-truth predicted, for a sample Twitch stream.

Algorithm 1: Predict Buffer Stall.

Parameters: Buf_{min}, Seg_{dur}

Data: *Chunks* detected on network $\{c_1, c_2, \dots, c_n\}$

Output: Estimated buffer health

```

1  $b \leftarrow 0$  (tracks current buffer)
2  $t \leftarrow 0$  (tracks endTime of last chunk)
3 for each network chunk  $c$  do
4    $b += Seg_{dur}$ 
5   if  $b \leq Buf_{min}$  then
6      $t = c.EndTime$ 
7     continue
8    $b -= c.EndTime - t$  if  $b \leq 0$  then
9      $b = 0$ 
10   $t = c.EndTime$ 

```

We observe that this low latency Twitch video starts with 2 seconds of the buffer. It soon encounters the first stall (highlighted by the red bar) around second 25 due to network congestion caused by cross traffic. Following that, the stream linearly increases its buffer to 10 seconds but experiences stalls a couple more times until the second 100. After this point, the buffer value increases to more than 20 seconds. It can be seen that a stall event not only deteriorates user experience but also increases the latency of the live stream as the user is watching content that was recorded at least 20 seconds ago – defeating the purpose of live streaming.

To predict such stalls, our buffer estimator algorithm takes two parameters as input. The first is Seg_{dur} ; live video streams typically encode content into video segments of

fixed duration. This duration depends on the playback mode - for instance, YouTube ULL streams have $Seg_{dur} = 1$ sec, while YouTube LL streams have $Seg_{dur} = 2$ sec. We automated this estimation by equating Seg_{dur} to be the median inter-request time (IRT) of video segments in the first window of n seconds (empirically configured to be 20 sec). The second parameter is Buf_{min} ; a client typically fetches few video segments (at least one) until a minimum buffer is filled before it begins playback. In the case of Twitch, playback begins after the first segment finishes downloading - hence, $Buf_{min} = 2$ sec (one segment long). However, in the case of YouTube, Buf_{min} seemed to vary between 2-10 seconds. Thus, we conservatively choose the mean value in our dataset - 3 sec for ULL streams and 6 sec for LL streams.

Using the parameters above and the isolated video chunks mentioned above, the buffer estimation algorithm (Algorithm 1) works as follows. At the beginning of a stream, its buffer is initialized at zero and increases by steps of Seg_{dur} at the end of every chunk observed on the network, until it reaches Buf_{min} (Algorithm 1, Lines: 4-7). For every subsequent video chunk, the buffer value is adjusted by: (a) adding Seg_{dur} and (b) subtracting the time elapsed in the playback since its previous chunk (Algorithm 1, Lines: 4,8).

Our algorithm predicts the current buffer value (in seconds) of the client video player for both providers. To quantify the accuracy of predicting buffer stalls (buffer value = 0), we first divide a given video stream into 5-sec windows and assign a boolean value (true when there was a stall and false otherwise) to each window. The ground truth of buffer stalls comes from the playback metrics collected by our tool described in §4.2.1. Table 4.5 summarizes the performance of predicting buffer stalls across all playback windows. Among all the windows across the video playback, we computed the accuracy, precision and false positive rate of our algorithm across providers as shown in Table 4.5. Overall, our algorithm yields about 90% accuracy in predicting the presence of a buffer stall in a 5-sec window. Note that our method tends to underestimate the buffer health in YouTube videos (false positive rate 14.2%) since we choose a conservative Buf_{min} value, leading us to predict stalls even when the buffer value is small but non-zero. We found that in more than 50% of the false-positives, our algorithm underestimates the buffer value by at most 2.3 sec (\approx the duration of a segment).

Table 4.5: Buffer Stall Prediction Results.

	Accuracy	Precision	FP Rate
Twitch	90.1%	90%	10%
YouTube	89%	88%	14%

4.4 Conclusion

Video streaming is a rapidly growing Internet application. ISPs today lack tools to infer its performance metrics in their network as existing DPI-based solutions fall short due to encryption and scale. In this chapter, we built a tool to collect a dataset on playback metrics and network measurement of video streams from Netflix, Twitch and YouTube, and subsequently analysed it to understand the correlation between video behaviour (on-demand and live) and its network activity. Subsequently, we developed algorithms and ML based models to infer video performance including buffering phase detection, bitrate estimation and quality degradation detection for on-demand video streams and prediction of resolution and buffer stall events for the live streams using chunk attributes extracted from the network flows. Our method enables ISPs to better understand the experience of video streaming services purely using the behavioural profile of network flows, and subsequently enables them to take corrective actions to help improve user experience.

Chapter 5

In-Network Game Detection and Latency Measurements

Contents

5.1	Introduction	75
5.2	Game Detection	77
5.2.1	Anatomy of Multiplayer Games	78
5.2.2	Signature Generation	80
5.2.3	Game Classifier	83
5.2.4	Evaluation	84
5.2.5	Field Deployment and Insights	85
5.3	Mapping Game Server Locations and Latencies	87
5.3.1	Active Measurements: Geolookup and Latency	87
5.3.2	Passive Measurements: Continuous latencies	88
5.3.3	Mapping Game Servers from the University	90
5.3.4	Comparing Gaming Latencies from Multiple ISPs	92
5.4	Conclusion	93

In this chapter, we shift our focus to understand another class of highly engaging applications: online multiplayer gaming. Most online games require only a few hundred kbps of bandwidth, but are very sensitive to latency. Internet Service Providers (ISPs) keen to reduce “lag” by tuning their peering relationships and routing paths to game servers are hamstrung by lack of visibility on: (a) gaming patterns, which can change day-to-day as games rise and fall in popularity; and (b) locations of gaming servers, which can change

from hour-to-hour across countries and cloud providers depending on player locations and matchmaking.

In this chapter, we develop methods that give ISPs visibility into online gaming activity and associated server latency. As our first contribution, we analyze packet traces of ten popular games and develop a method to automatically generate signatures and accurately detect game sessions by extracting key attributes from network traffic. Field deployment in a university campus identifies 31k game sessions representing 9,000 gaming hours over a month. As the second contribution, we perform BGP route and Geolocation lookups, coupled with active ICMP and TCP latency measurements, to map the AS-path and latency to the 4,500+ game servers identified. We show that the game servers span 31 Autonomous Systems, distributed across 14 countries and 165 routing prefixes, and routing decisions can significantly impact latencies for gamers in the same city. Our study gives ISPs much-needed visibility so they can optimize their peering relationships and routing paths to better serve their gaming customers.

5.1 Introduction

Online gaming is experiencing explosive growth: 2.9 billion players collectively contributed \$178 billion to global revenues in 2020, representing a 23% growth over the year before [164]. Popular online games like Fortnite, Call-of-Duty, League of Legends and Counter-Strike account for hundreds of millions of online players. Interestingly, most of these games are free-to-play, and generate their whopping revenues from in-game purchases (in-game currency, emotes, skins, stickers, weapons, backblings, battle passes, and other such trinkets). Game publishers and platforms are therefore strongly motivated to give gamers the best possible experience to keep them engaged, and thus deploy their game servers on cloud platforms across multiple countries in an effort to minimize network latency for users.

Network latency (aka “lag”) is indeed one of the largest sources of frustration for online gamers. A typical shooting game requires no more than a few hundred kbps of bandwidth, so a higher speed broadband connection does not by itself have a material impact on

gaming experience. By contrast, a 100 ms higher latency can severely handicap the gamer [165], since their gunshots will be slower to take effect, and their movements lag behind others in the game. A whole industry of “game acceleration” is dedicated to address the latency issue, ranging from gaming VPNs/overlays (*e.g.*, WTFast[166] and ExitLag[167]) to gaming CDNs (*e.g.*, SubSpace[168]); indeed, one innovative eSport hosting company (OneQode[169]) has even gone to the extent of locating its servers in the island of Guam to provide equidistant latency to several Asian countries.

Internet Service Providers (ISPs), who have hitherto marketed their broadband offering based purely on speed, are now realizing that they are blind to latency. This is hurting their bottom line, since gamers are vocal in online forums comparing gaming latencies across ISPs, and quick to churn to get any latency advantage. With new game titles and seasons launching every week, and their popularity waxing and waning faster than the phases of the moon, ISPs are struggling to stay ahead to keep gamers happy, and consequently bearing reputational and financial damage.

ISPs have almost no tools today to give them visibility into gaming latencies. Traditional Deep Packet Inspection (DPI) appliances target a wide range of applications spanning streaming, social media, and downloads, and have evolved to largely rely on hostnames found in DNS records and/or the TLS security certificates of a TCP connection. Tracking modern games requires specialized machinery that can track UDP flows with no associated DNS or SNI signaling by matching on multiple flow attributes in a stateful manner. Further, game developers and publishers use different cloud operators in various countries to host their game servers, and use dynamic algorithms for game server selection depending on the availability of players and match making. These factors have made it very challenging for ISPs to get visibility into game play behaviors, limiting their ability to tune their networks to improve gaming latencies.

In this chapter, we develop a method to detect games, measure gaming latencies, and relate them to routing paths. Our first contribution in §5.2 analyzes ten popular games spanning genres, developers, and distributors. We identify key game-specific attributes from network traffic to automatically construct game signatures, and consolidate these into an efficient classification model that can identify gaming sessions with 99% accuracy

within first few packets from commencement. Deployment of our classifier in a University network over a month identified 31k game sessions spanning 9,000 gaming hours, and we highlight interesting patterns of game popularity and engagement in terms of session lengths.

Our second contribution in §5.3 uses the servers identified using our classifier from the previous contribution to measure game servers location and latencies. We perform BGP route and Geolocation lookups, coupled with active ICMP and TCP latency measurements, to map the AS-path and latency to the 4,500+ game servers identified. We illustrate the spread of game servers across 31 ASes, 14 countries, and 165 routing prefixes, and the resulting impact on latency for each game title. We further show that different ISPs serving gamers in the same city can offer radically different gaming latency, influenced by their peering relationships and path selection preferences. Our study gives ISPs much-needed visibility into gaming behaviors and game server locations so they better optimize their networks to improve gaming latencies.

Table 5.1: List of games.

Game	Genre	Developer	Distributor/Publisher
Fortnite	Shooter	Epic Games	Epic Games
Call of Duty: Modern Warfare (CoD:MW)	Shooter	Infinity Ward	Blizzard Entertainment
World of Warcraft (WoW)	RTS	Blizzard Entertainment	Blizzard Entertainment
League of Legends (LoL)	MOBA	Riot Games	Riot Games
Counter Strike: Global Offensive (CS:GO)	Shooter	Valve Corp.	Steam
FIFA 20/21	Sports	Electronic Arts	Origin
Rocket League	Sports	Psyonix	Steam
Hearthstone	Card game	Blizzard Entertainment	Blizzard Entertainment
Escape From Tarkov	Shooter Survival	Battlestate Games	Battlestate Games
Genshin Impact	Action RPG	miHoYo	miHoYo

5.2 Game Detection

In this section, we begin by illustrating the network behavior of a representative online game (§5.2.1), followed by developing: (i) a method to automatically generate signatures of gaming flows (§5.2.2), and (ii) a deterministic classifier that combines the signatures to passively detect games using in-network attributes (§5.2.3). The classifier is evaluated (§5.2.3) and deployed (§5.2.4) to observe the gaming patterns in our university network.

We first collected and analyzed hundreds of *pcap* traces by playing ten popular online games (shown in Table 5.1) that represent a good mix across genres (*e.g.*, Shooting, Strategy, Sport), multiplayer modes (*e.g.*, Battle-Royale, Co-Operative, Player-vs-Player), and developers/distributors. These traces (labeled lab data) were collected by playing games on a desktop computer in our university research lab. Next, we collected over 1000 hours of game-play packet traces selected from a full mirror (both inbound and outbound) of our university campus Internet traffic (on a 10 Gbps interface) to our data collection system from the campus border router. Selected *pcaps* (labeled field data) were recorded by filtering the IP address of the game servers (to which our lab computer connected while playing). This helped us collect all game-play traffic to those “known” servers when someone on our campus played any game. To gather and analyse this data, ethics clearance (HC16712) has been obtained from UNSW Human Research Ethics Advisory Panel. This clearance approves the analysis of campus network traffic data to derive insights into aggregate video streaming and gaming behaviours without identifying the users behind the client ip addresses.

5.2.1 Anatomy of Multiplayer Games

Let us start with an illustrative example from a popular online game: Fortnite. It is a third person shooter (TPS) game developed by Epic Games which has risen in popularity with a game mode called Battle Royale wherein 100 players fight each other to be the last one standing. Fortnite is played by over 350 million players around the world [170]. In what follows, we outline the anatomy of a Fortnite game session by manually analyzing a packet capture (*pcap*) trace from our labeled lab data.

Gamer Interaction: A gamer first logs in to the Epic Games launcher and starts the Fortnite game client. The game starts in a lobby where users have access to their social network, collectibles, player stats, and game settings. When the user decides to play, the client contacts Fortnite’s matchmaking server that groups players waiting in a queue and assigns a server on which the online game runs. Subsequently, the match starts, and its duration depends on how long the player lasts in the battle royale – the last one/team standing wins among 100 players. After the game, the user returns to the lobby area,

where they may choose to start another game.

Table 5.2: Fortnite Services, their name prefixes (suffix=`ol.epicgames.com`) and purpose.

Service	Domain Name Prefix	Purpose
Launcher	<code>launcher-public-service-prod06</code>	Epic games launcher for login
Waiting Room	<code>fortnitewaitingroom-public-service-prod</code>	The user decides the game mode
Party	<code>party-service-prod</code>	Lobby area to invite friends to play
Social Network	<code>friends-public-service-prod</code>	In-game social network
Matchmaking	<code>fortnite-matchmaking-public-service</code>	Creates matches among waiting players
Anti-cheat	<code>hydra.anticheat.com</code>	Third-party anti-cheat service
Data reporting	<code>data-router</code>	Anonymous stats reporting

Network Behavior: From the *pcap* trace, we observe that the client communicates with various service endpoints (which can be identified by their unique domain name) for joining the lobby, matchmaking and social networking (as shown in Table 5.2). These communications occur over encrypted TLS connections and constitute “foreplay” before game-play begins. Once the game starts, the actual game-play traffic is exchanged over a UDP stream between the client and a game server (which is usually different from the foreplay endpoints). However, the IP address of the gaming server is not resolved by DNS lookup – we, therefore, believe the server IP address is exchanged over the encrypted connection during the matchmaking process. The lack of the server identity/name (common across other game titles) makes the game-flow detection challenging. We note that the game server and other servers may or may not be co-located – *e.g.*, the game server may be very close to the user, but the matchmaking server could be operating from a different cloud in a different country.

The Fortnite game-play stream (identified using a five-tuple: SrcIP, DstIP, SrcPort, DstPort and Protocol) has a packet rate of 30-40 pkt/sec upstream and about 30 pkt/sec downstream throughout the game – fluctuations depend on player actions. However, this profile of flow rate (as used in some prior works to classify applications [84]) is insufficient to detect the game since we observed a similar pattern in other games. Most gameplay flows have the following traffic characteristics: they operate in the packet rate range of 20-130 packets per second – games with competitive genres like Shooters and Strategy have higher tick-rates while games which are turn-based like Hearthstone have low packet rates. Further, gameplay streams contain packets of varying sizes depending on type of

game and the messages exchanged between the client and server *i.e.*, during an intense fight the packet sizes increase for certain games (like Call of Duty). The packet lengths however rarely are close to MTU. The similarities in flow level behaviour make it difficult to distinguish games using traffic classification models like FlowFormers presented earlier.

That being said, gameplay streams do contain some idiosyncratic characteristics. For example, in Fortnite, the client connects to port 9017 on the server-side in our example trace; it starts with a few packets of payload size of 29 bytes; the first upstream packet contains 28 trailing `0x00`s; etc. These features, albeit simple, seem to be unique to Fortnite. The other competitive games we analyzed displayed similar patterns of user activity and interaction including contacting various services and having idiosyncratic patterns in the first few packets. We next describe methods to analyze multiple gaming flows to extract such signatures automatically.

5.2.2 Signature Generation

As briefly mentioned above, game-play servers typically lack DNS records, and the flow rate profile is quite similar across games. Therefore, identifying the game-play flows (among a mix of traffic) becomes challenging and requires us to inspect packets of flows for patterns. While signatures can be generated manually by playing the game to collect packet traces, we develop a method to automatically extract signatures from a collection of flows associated with game servers captured in our field dataset.

Dataset: From the lab and field packet traces (described above), we obtained over 20,000 labeled flows, with each game at least having 500 flows. We filtered and cleaned the field traces to remove non-game-play flows using simple heuristics such as flow duration (games typically tend to last for more than a minute at the very least) and protocols (excluding ICMP traffic). A flow record in our dataset contains: (i) game name, (ii) transport-layer protocol (UDP/TCP), (iii) server-side port number – *e.g.*, 9017 for the Fortnite example considered in §5.2.1, (iv) packet size (in bytes) arrays of upstream and downstream directions each for five initial packets – *e.g.*, up:[29,29,50,314,78] and down:[29,29,116,114,114], and (v) payload byte (in hex strings) arrays of upstream and

downstream directions each for five initial packets – *e.g.*, [“17aabb...”, “28a004...”]. We note that while client-side port numbers can be useful, they are often obfuscated due to the presence of NAT and hence are not considered in this study. Further, we extract packet-level attributes from just the first five upstream and five downstream packets as they are enough to capture game-specific handshakes.

To extract game signatures from our dataset, we focused on extracting specific patterns, which could be a *static* value (consistent across all flows of a game title) or a range of *dynamic* values. To illustrate, Fortnite¹ comes with the following specific signatures: the server UDP port number is a *dynamic* value between 9000 and 9100; 1st upstream and downstream packets have a *static* size of 29 bytes ($u_0_len = d_0_len = 29$)²; second to tenth byte of 1st upstream packet are 0x00. ($u_0_b_1 = \dots = u_0_b_9 = 0x00$)³

Static Signatures: We extract static signatures from *protocol*, *packet size* and *payload byte content* specific to each game title by checking if an attribute has the same value for more than α fraction of the flows. If so, the attribute and its value are added to that specific game’s signature (*e.g.*, “ $u_0_len = 29$ ” or “ $u_0_b_9 = 0x00$ ”). Note that if α is set to a small value (say, 0.5), the game’s signature becomes richer (containing more attributes to match) and more specific to that game. A rich signature demands more stringent requirements from a flow (*i.e.*, higher chance of rejecting a flow with minor deviation from expected attributes – resulting in false-negatives). Setting α to a value close to 1 makes the signature fairly generic, which would imply a chance of overlap with other games – resulting in false positives. We empirically tuned it at 0.90 to strike a balance and detect the games accurately. In addition, we use another parameter k to specify the depth of packet payload (in number of bytes) to be analyzed. We found that most of the static payload byte values can be captured by looking at just the first 10 bytes of each packet, meaning $k = 10$.

Dynamic Signatures: We extract dynamic signatures for *server-side port numbers* as they often do not have a fixed value but lie in a specific range of possible values (configured

¹A snippet of our signatures for three representative games is shown in Fig. 5.2)

²“ d_0_len ”: first letter denotes the direction (“ d ” for downstream and “ u ” for upstream), second letter (“0”) denotes the packet index, and third letter (“ len ”) denotes the packet size.

³“ $u_0_b_9$ ”: the letters “ u ” and “0” are same as above while third letter (“ b ”) denotes byte, and fourth letter (“9”) denotes the byte index.

Game	Protocol	Server Port	Up Pkt sizes	Down Pkt sizes	Up Payloads	Down Payloads
Fortnite	17	9017	[29, 29, 45, 62, 80]	[29, 29, 45, 62, 80]	[0x170000, ...]	[0xd7bf45e8, ...]
Fortnite	17	9002	[29, 29, 35, 43, 51]	[29, 29, 45, 62, 80]	[0x170000, ...]	[0x570000c0, ...]
Fortnite	17	9035	[29, 29, 37, 89, 74]	[29, 29, 45, 62, 80]	[0x170000, ...]	[0x07e86474, ...]
Fortnite	17	9067	[29, 29, 41, 39, 72]	[29, 29, 45, 62, 80]	[0x160000, ...]	[0xf0c476e6, ...]

Figure 5.1: An illustrative example of signature generation using Fortnite traffic traces

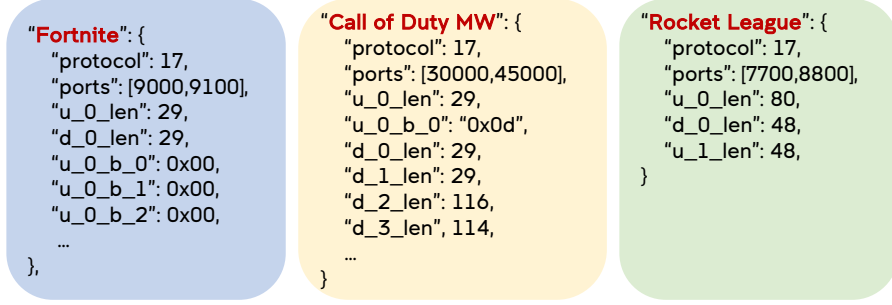


Figure 5.2: Signature of three representative game titles

by their developers). Since we collected a rich set of flows in the field dataset, we use the *min* and *max* of the port numbers to identify an expected range. We further expand the range by rounding the *min* and *max* to the nearest 100 to capture those port numbers that might have missed out in our traces. Doing so gives us a signature like *port* = [9000 – 9100] for Fortnite.

Example: Fortnite Game Signature Generation As shown in Fig. 5.1 above, each row corresponds to attributes extracted from the first few packets of Fortnite gaming flows from our dataset. The attributes include protocol, transport layer port numbers, packet sizes and payload bytes. In one flow (identified by the standard five-tuple), protocol and server port remain the same but the packet sizes and content vary as more packets arrive. For this illustration, the table shows 5 packet sizes in each direction and (stripped) payload content of the first packet. Some attribute values (shown in red) are fixed/constant across all the flows (called *static signatures*) and other (shown in green) fall within a close range of values (called *dynamic signatures*). These signatures are same across the flows implying that they can detect a Fortnite game session.

Fig. 5.2 shows example signatures generated from our dataset. We can see that while all attributes have a key and a value, only *ports* has a range since it is a dynamic signature. We note that the complexity of signatures varies: some are primarily based on packet size

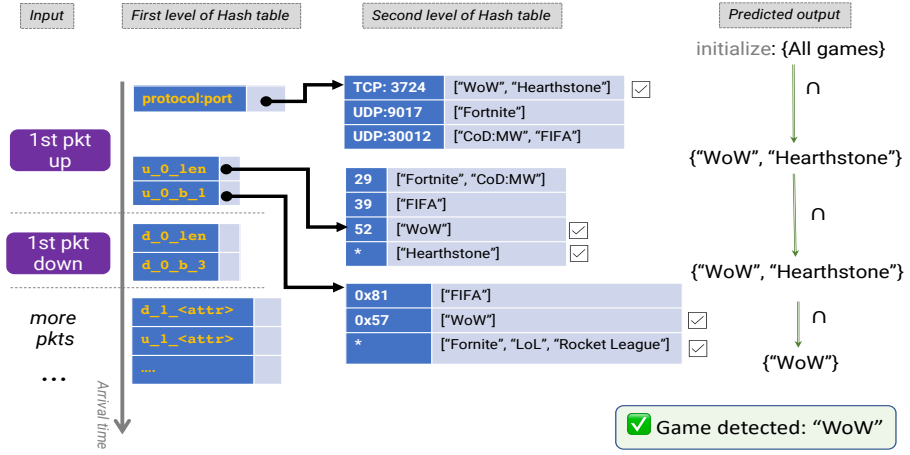


Figure 5.3: The structure of our classifier, illustrating a progressive classification of a flow.

(Rocket League) while others require payload bytes too (Fortnite and Call of Duty MW); some are based on attributes of first two packets (Fortnite and Rocket League) while others require more data (Call of Duty MW). These signatures need to be combined to predict the actual game being played as they may have some common attributes for *e.g.*, both Fortnite and Call of Duty MW have the first upload packet length as 29 and thus require further inspection to classify the game. Therefore we need a classifier model takes into account all attributes and looks at the minimum number of packets to rapidly detect the game.

5.2.3 Game Classifier

We employ a *two-level hash table* (Fig. 5.3) that is constructed by combining all the game signatures extracted above, enabling us to rapidly detect game-play flows (and dismiss undesired traffic). The first level contains the packet attributes (*e.g.*, `u_0_len`, `u_0_b_0`) as keys. The second level contains the possible values of the attribute as key, and possible game titles that have the same value as the entry of the hash table.

Flow of Events: Given the pre-populated hash table, we demonstrate our classification algorithm for an illustrative example in Fig. 5.3. We initialize the predicted output by the set of all possible games in our dataset (shown on the right side). For each incoming packet of a given flow (shown on the left side), the attributes are extracted and looked up in the hash table. For each attribute, a set of possible game classes is inferred. For

an illustrative WoW (World of Warcraft) flow, upon arrival of the first packet, the protocol and port are identified as TCP:3724. Looking them up in the hash table followed by intersection with `{all games}` gives us the set `{‘WoW’, ‘Hearthstone’}` as output. We then proceed by looking up the packet size of 52 bytes. While 52 only yields WoW in our hash table, keep in mind that Hearthstone corresponds to a wildcard (`*` : indicating that attribute values were not static) meaning that the size of the first upstream packet in Hearthstone can be anything (including 52) and hence no change in the output game set. Upon extracting the second byte of the first upstream packet (`u_0_b_1`) we narrow it down to WoW. When the set of games reduces to one game, we declare it as classified. Thus, the classifier rapidly eliminated other possibilities and detected a WoW game-play flow by analyzing the protocol, port, packet size, and the first few bytes of the upload packet. Note that packets’ inter-arrival time in a game-play flow is in the order of milliseconds, giving sufficient room for hash table lookups (in the order of microseconds) in between packets.

We intentionally employ an algorithmic model rather than a machine learning model since the latter requires all the input attributes to be collected, stored and processed in memory to make a classification decision, which is more expensive in memory and compute. Our classifier model detects the game or rejects non-gaming flows progressively on a per-packet basis, without necessarily requiring the attributes of all ten initial packets. Whenever the possible games reduce to an empty set, we do not process packets of that flow further by classifying it as a non-gaming flow. This helps us quickly eliminate flows (often on the first packet) that do not form a part of our game set. For example, none of the games use HTTP(S), so a majority of the traffic using TCP:80 or TCP:443 is eliminated straight away and is never detected as a game. This avoids unnecessary per-flow state maintenance (no state is maintained for flows rejected on the first packet) and helps our detection method scale.

5.2.4 Evaluation

Our model (signatures and classifier algorithm) achieves an overall accuracy of **99.6%** (with a precision of 100% and a false negative rate of 0.36%) when it is applied to our field

dataset. We used the metrics precision to signify that flows that were identified as games were indeed classified with the right title. The metric false negative rate indicates that a small fraction of gameplay traffic wasn't detected as gameplay. We found that flows of nine game titles receive a perfect accuracy 100%, while 4.5% of WoW flows are not detected as a game flow. Note that our game-specific signatures are generated based on traffic patterns found in $\alpha = 0.90$ fraction of labeled game flows; hence a minority of flows that do not conform to those signatures will not be detected as gaming flows. Our model may miss some game flows but indeed detects games correctly and confidently. We observe that the model is able to detect all games in our dataset within the first two packets (first upload and first download) as the signatures across the ten games are fairly unique, resulting in a rapid detection.

5.2.5 Field Deployment and Insights

The game detection system was deployed in our university campus network (with users from offices and student dormitories) during the month of Sep 2021 to obtain insights into the game playing patterns, as well as to determine corresponding gaming servers that clients connect to and their latency from our campus (discussed in §5.3). Our classifier (loaded with the signatures) is implemented as a DPDK[14] application running on a server which receives campus traffic mirror from optical taps (observed total traffic peak: 8Gbps). To reduce the rate of false positives in the wild *i.e.*, not detect non-gaming traffic as games, we made our algorithm more conservative to analyze all attributes of the initial ten packets of each flow before classifying the flow. Also, we monitored the activity of the flow for the first minute of its lifetime, ensuring packet rates match the expected rate of gaming flows (typically less than 100 pkts/sec).

The system detected over 31k game-play sessions, constituting nearly 9000 hours worth of game-play across the ten titles. We found that the top three games by the number of gaming sessions were CoD:MW (9545), Fortnite (7930), and League of Legends (6290). Interestingly, LoL dominated by the total number of gaming hours – LoL was played for 2611 hours, followed by CoD:MW for 1575 hours and Fortnite for 1562 hours. This highlights the games with which gamers generally engage most.

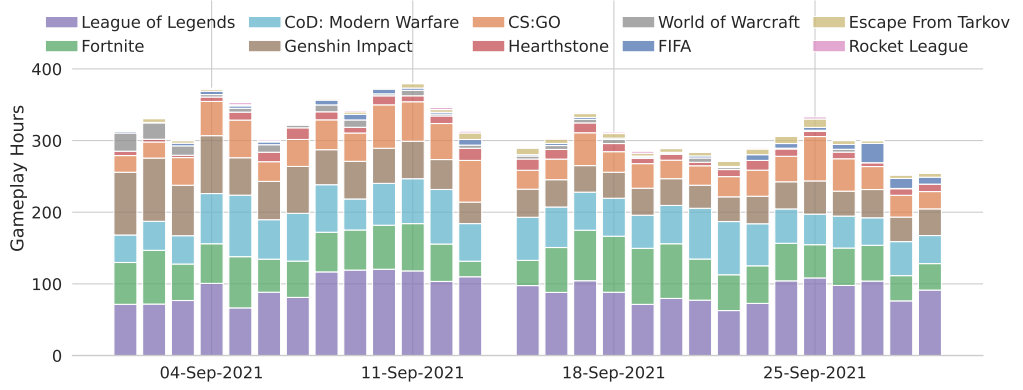


Figure 5.4: Dynamics of daily game-play hours across ten titles during field trial.

Fig. 5.4 shows the dynamics of daily game-play hours across the ten titles. Unfortunately there was a power outage in our lab on 14 Sep, causing data to be missed for that day. We make a couple of observations: (a) there is a slight decreasing trend of daily gaming hours during this period (more gaming hours in the first half than the second half) due to academic term starting on 13-Sep following a study break; and (b) gaming hours fluctuate across game titles – as an example, Genshin Impact (shown in brown) was more popular early in September (≈ 87 hours daily), but then trended down to less than half that (≈ 37 hours daily) towards the end of the month; Fortnite (shown in green) was played for 475 hours in the third week when Chapter 2 Season 8 was released, but this dropped to 325 hours in the fourth week once the excitement wore off – such ebb and flow is the norm in gaming [171], requiring ISPs to have constant visibility so they can tune their networks accordingly.

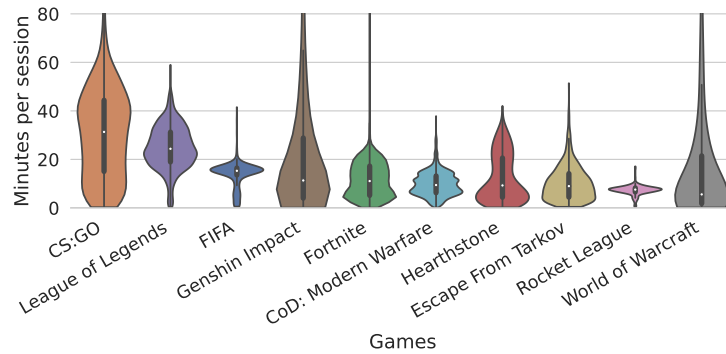


Figure 5.5: Distribution of game-play session duration across the ten titles.

Fig. 5.5 shows the distribution of game-play session duration across the ten titles. We observe a few patterns of user engagement with various games: Several CS:GO, Genshin Impact, and WoW gamers spend more than an hour in each gaming session, with CS:GO

being the most engaging game with median duration of 32 minutes. Rocket League is played for a relatively fixed duration of 10 minutes. Further, the impact of game modes is pronounced in games like CoD:MW with three bumps on its corresponding curve, highlighting three clusters of game modes, namely 5v5, GroundWar, and BattleRoyale offered by this game title.

Lastly, we analyzed short game flows (with duration less than 2 min), which can indicate game abandonment. While only 3.5% of the flows with local servers (within Australia) were short, it quadruples to more than 12% when the game is played on remote servers. Though correlation should not be interpreted as causation, it does indicate that gamers tend to abandon games more often when the latency to the server is high. The next section draws insights into game server locations and latencies.

Table 5.3: Summary of detected game-play sessions in our field trial.

# Game Session	# Game Hour	# Game Server	# IP Prefix	# AS	# Country
31673	8956	4523	165	31	14

5.3 Mapping Game Server Locations and Latencies

Having measured gaming behaviors in the University campus over a one-month period, we now shift focus to the game servers, including their location and latency. For each of the game-play servers, we obtain their geo-location, BGP routing prefix, and the AS number from public data sources. Additionally, we perform active ICMP/TCP-based latency measurements to the detected game servers. This covers over 31k gaming sessions played against 4,500 unique game servers, spread across 14 countries and 165 routing prefixes and 31 ASes, as shown in Table 5.3.

5.3.1 Active Measurements: Geolookup and Latency

We employed an IP Geolocation service [172] to tag the location of every server IP address. We also used the online Looking Glass tool exposed by the University’s ISP, that offers **ping**, **traceroute**, and BGP queries to obtain routing prefix (*i.e.*, the subnet of the server IP address) and its AS path. Furthermore, we estimated the latency (we will use latency

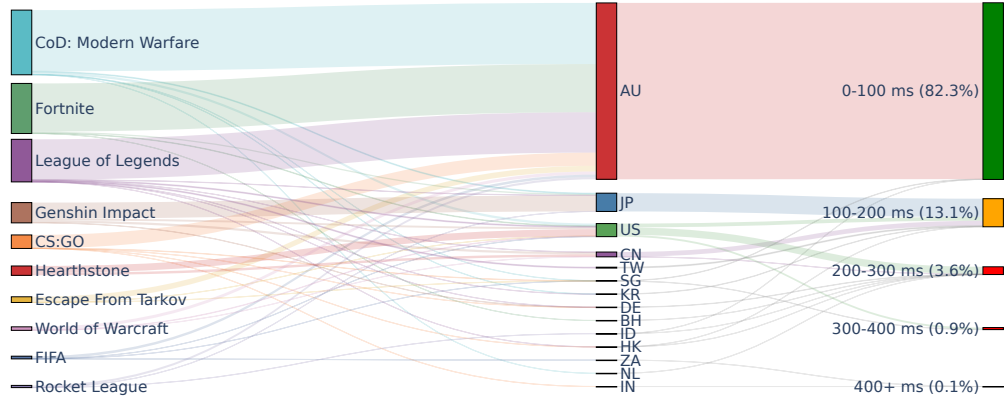


Figure 5.7: Sankey diagram depicting game sessions, countries, and latency bands.

with the latest one corresponding to $\text{seq } n (+1)$. The server chooses to acknowledge them both together and sends an data packet with ACK number set to $n + 1$. Now, the monitor dequeues the two tuples until it finds one whose value is equal to the ack number (which is packet two). Hence, it computes the RTT between the monitoring point and server by subtracting the timestamp of ack with timestamp in the two-tuple enqueued. The server also send data along with ack ($n + 1$) with a sequence number m . The monitor uses another queue to store two-tuples in this direction and estimates the RTT between monitor and client in a similar way. Thus, by adding up both the RTT, we can compute the RTT between a gaming client and gaming server for the games running on TCP.

We note that in case of packet losses and retransmissions, a two-tuple with the same sequence maybe enqueued twice (i.e. when loss happens after the monitoring point). To account for such cases, the monitor dequeues the packets until the point when the head of the queue has a greater sequence number than the ack number in the other direction.

We further note that we could not passively measure latencies for UDP games since the UDP protocol does not have sequence numbers and acking mechanisms. In addition, most of the games are asymmetric – upload stream is independent of download stream and hence correlating both directions is non-trivial.

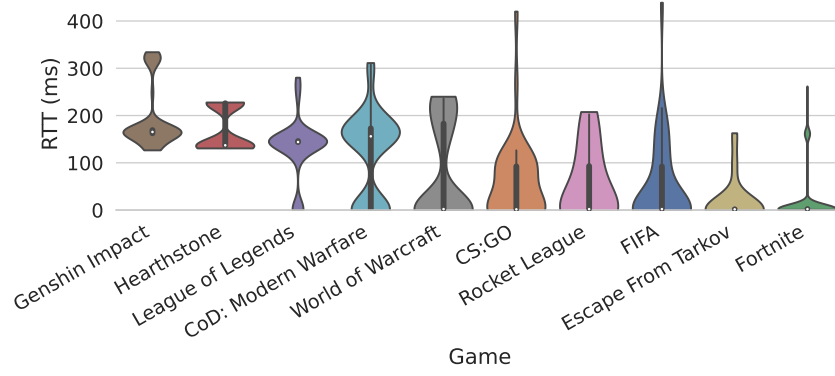


Figure 5.8: Latency distribution of game servers from the campus field trial.

5.3.3 Mapping Game Servers from the University

A high level view of sessions of each of the ten game titles as they map to servers in various countries and at different latency bands is shown in Fig. 5.7. Most countries map to a single latency band (needless to say Australia (AU) is the home country), though some countries (like US) map to multiple latency bands, due to disparities in routing paths to multiple ASes in the same country, or to different subnets within the same AS. Specifically, 82.3% of the game-play sessions connected to servers within Australia with fairly low latency of 2-20ms, 13.1% of the sessions experienced 100-200ms, 3.6% had 200-300ms, and 1% had latency of 300+ ms.

Our measurements clearly reveal that game providers often use multiple CDNs (each identified by a unique AS number) to host their game servers – for example, while Fortnite largely connects to Amazon cloud locally, some sessions connected to Google cloud in another country. There are several reasons why a gamer’s session may be hosted at a server with high latency: (a) no nearer server availability; (b) there may not be enough local players available, and the player is therefore matched with players in other geographies; or (c) the player deciding to team with friends in another country, and the server is chosen in proximity to the majority of players.

To get a better understanding of gaming latency per title, we plot in Fig. 5.8 the latency distribution across the ten games. Fortnite and Escape from Tarkov predominantly use local servers (50ms or lower); League of Legends and CoD:MW use only a small number of local servers; while Hearthstone and Genshin Impact do not have any servers operational

in the local country (the closest ones being 100+ ms away). It is also interesting to see that servers are clustered for some games (*e.g.*, Hearthstone, Genshin Impact, WoW), highlighting servers co-located in the same CDN. Curiously, though WoW and Hearthstone are from the same publisher (and share the AS owned by Blizzard), only WoW uses local in-country servers.

To highlight the deeper dynamics of latency, we focus on League of Legends (LoL) and show in Fig. 5.9 the distribution of latency across various server prefixes, color-coded by their country of residence. The game connected to 293 servers located in 8 countries spanning 22 routing prefixes. We observe that it has only one routing prefix locally (P1) that offers a very low latency of under 5ms. Across other prefixes, we make a couple of observations. First, prefixes (P3, P4, P9) and (P13, P15), while located in China, belong to two different ASes and hence give very different latencies. In fact, P13 is geographically closer to P3 but the latter is one AS hop away while P13 is 3 AS hops away, leading to a latency differential of about 100ms. Second, prefixes (P5, P6, P16) belong to the same AS and are located in USA. They are all one AS away from the source but P16 has a 120ms higher latency, illustrating that routing paths can vary for different subnets even within the same AS (in this case owned by Riot Games, the publisher of LoL). Further, counter-intuitively, prefix P16 is geographically closer (to game client) than P5 and P6. This analysis can help ISPs identify game server locations and routing prefixes so they can tune their peering relationships and path selections to improve latency for their gamers.

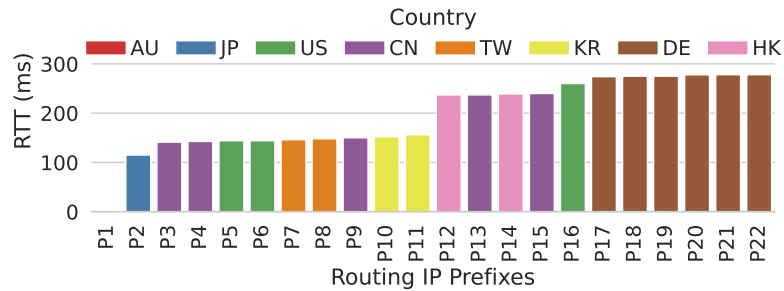


Figure 5.9: Latency per IP prefix of the League of Legends servers.

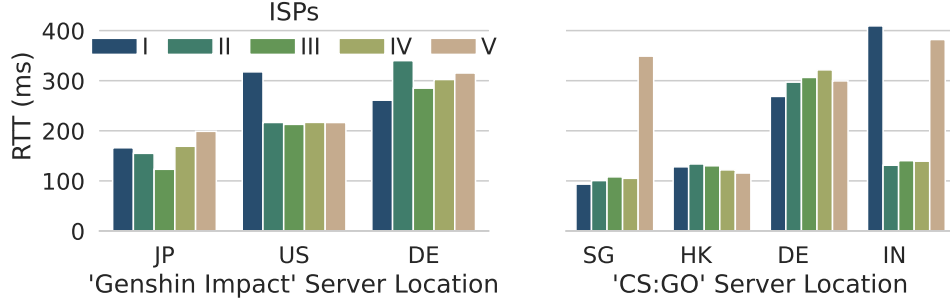


Figure 5.10: Measured latency across ISPs to popular external (outside country) game server subnets of Genshin Impact (left) and CS:GO (right).

5.3.4 Comparing Gaming Latencies from Multiple ISPs

To better illustrate the impact of peering relationships and routing paths on latency, we performed active latency measurements (using an automated script) from several volunteers' home broadband connections in our local city to the game servers discovered in §5.2. The volunteers were spread across four residential ISPs (numbered *II-V*, with *ISP-I* representing the University), and we found that the average latency to game servers outside the country varied significantly across these ISPs, as illustrated in Fig. 5.10 for two representative games namely Genshin Impact and CS:GO.

Genshin Impact has no local servers, and a majority of its servers are in Japan (JP). It can be seen that *ISP-III* offers the lowest latency of 119ms while the latency is much higher (at around 198ms) with *ISP-V*. The USA serves the next higher number of sessions of Genshin Impact, and in this case ISPs *II-V* provide a latency of around 200ms, while the University's *ISP-I* has 300+ ms latency. For Denmark (DE), *ISP II* provides the highest latency at 322ms. Overall, a Genshin Impact gamer would get a better experience if they were with *ISP-III*. However, any ISP with this visibility into game server locations can optimise their routing paths to improve the gamer experience.

The difference across ISPs for CS:GO is even more stark, as shown in the right side of Fig. 5.10. In this case *ISP-V* offers a significantly worse latency to CS:GO servers in Singapore (SG) and India (IN). Given that CS:GO is a tournament-grade first-person shooting game, the latency handicap induced by *ISP-V* will be unacceptable to gamers, and likely to lead to complaints and churn. The situation is very avoidable – indeed we

have reached out to this ISP, urging them to look into their peering relationships and routing path selections to address this issue.

5.4 Conclusion

The gaming industry is experiencing explosive growth, and ISPs are keen to offer a better gaming experience to their subscribers. However, they are hamstrung by the lack of visibility into gaming patterns, servers, and latencies. We collected and analyzed packet traces from ten popular games across various genres, extracted packet attributes, and developed a deterministic model to identify games based on automatically generated game-specific signatures. We deployed our system on live traffic of a university network, and over a 1-month period detected 31k game sessions to gain insights into game popularity and gaming engagement. We then related game latencies to routing paths by performing BGP/Geo lookups and active latency measurements to the 4,500+ game servers identified. We illustrated how the spread of games servers across ASes and countries impacts latency. Finally, we showed that ISPs serving gamers in the same city have varying latencies to these game servers, influenced by their peering relationships. Our study gives ISPs much-needed visibility so they can optimize their peering relationships and routing paths to better serve their gaming customers.

Chapter 6

Data-driven Management of Application Performance

Contents

6.1	Introduction	96
6.2	AppAssist: Self-driving network prototype	97
6.2.1	System Architecture and Design	98
6.2.2	Assisting Sensitive Applications	102
6.2.3	Summary	109
6.3	AutoQoS: Application-aware automatic QoS configuration	109
6.3.1	Overview	112
6.3.2	System Design	114
6.3.3	Implementation: Application Profiles and Telemetry	119
6.3.4	Implementation: Software and Hardware Testbed	124
6.3.5	Evaluation	126
6.3.6	Discussion	134
6.4	Conclusion	135

In the previous chapters, we have tackled application detection and performance monitoring looking specifically at two classes of traffic: video streaming and gaming. These applications have different requirements from the network: video streaming being bandwidth sensitive and gaming being latency sensitive. Different network impairments can impact their performance and can result in a poor user experience. Examples include being far from a WiFi router affecting WiFi signal strength, inefficient routing leading to

high latencies and multiple applications contending on a bottleneck link thereby adversely impacting the performance of each other.

This chapter focuses on bottleneck links within operator networks that cause a poor user experience since they are typically within the control of the operator and can be actioned upon. We have worked on a few frameworks that assist network operators in improving the application performance:

- Firstly, operators can formally specify bandwidth allocation amongst traffic classes that suits most common scenarios and offer differentiated plans [21]. This framework optimizes bandwidth allocation using formally specified utility curves without the need of application performance measurement.
- Secondly, they can use application performance monitoring systems (that we've shown in the past few chapters) and prioritize traffic depending upon their performance (*i.e.*, selectively prioritize "suffering" applications) [20]. This approach leverages application level QoE metrics and uses simple prioritization primitive to prioritize (and de-prioritize) applications according to their performance.
- Finally, we also propose an AutoQoS framework which takes specialized network telemetry and dynamically adapts QoS configurations on the bottleneck link to share the resource in an application fair manner. This is the most complete framework as it combines techniques from the first two frameworks by incorporating sophisticated telemetry and application-specific profiles (instead of operator specified functions) to measure application performance and uses dynamic resource scheduling (instead of simply prioritizing one class over another).

Depending on the requirements of operators, for instance, offering temporary application boost plans etc. and the feasibility of the deployment, for instance, picking from the available queueing primitives, a framework can be picked and employed.

6.1 Introduction

User-perceived application experience is of paramount importance in broadband as well as cellular networks, be it for video streaming, teleconferencing, gaming, or web-browsing. The best-effort delivery model of the Internet makes it challenging for Application/Content Providers to maintain user experience, requiring them to implement complex methods such as buffering, rate adaptation, dynamic CDN selection, and error-correction to combat unpredictable network conditions. Network operators, also eager to provide better user experience over their congested networks, often employ middle-boxes to classify network traffic and apply prioritization policies. However, these policies tend to be static and applied on a per-traffic-class basis, with the benefits to individual applications being unclear, while also potentially being wasteful in resources.

In this chapter, we tackle the problem of active management of application QoE, especially focusing on aggregate QoE of applications sharing a common bottleneck link (typically in the last mile network [7]). We begin by proposing a self-driving network prototype called *AppAssist* which uses simple telemetry functions to map out application behavioural states and selectively prioritizes applications entering into a “bad” state. This prototype gives an overview of a self-driving network designed to monitor and act upon application performance. Subsequently, we then present a complete system called *AutoQoS* which can leverage programmable networks to measure, monitor and manage application performance and automatically update QoS parameters such as weights of a weighted-fair-queueing scheduler to distribute the bottleneck resources in a fair manner. While we have also worked on the first proposal of an open, formal and rigorous bandwidth allocation framework in one of our papers [21], we do not include it in the scope of this thesis since it doesn’t take in application performance metrics as input and instead relies on a policies given by operators which is not always feasible. However, both *AppAssist* and *AutoQoS* take application performance (as studied in previous chapters) under consideration while making decisions to manage bottleneck resources and improve user experience.

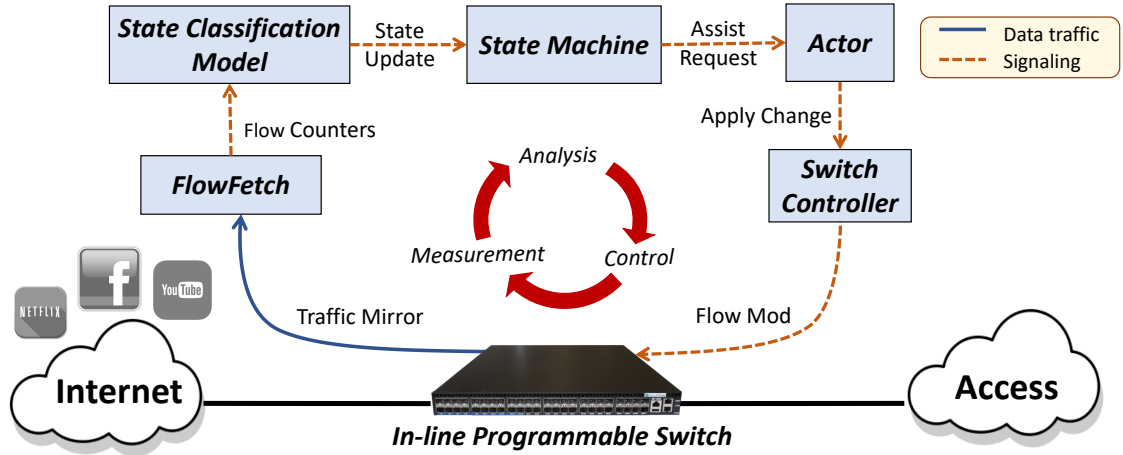


Figure 6.1: System architecture of assisting applications.

6.2 AppAssist: Self-driving network prototype

It is envisaged that Self-Driving Networks of the future will be able to address the problem of active QoE management through a combination of continuous network measurement, automated inferencing of application performance, and programmatic control to protect user experience. This chapter works towards this goal, leveraging recent developments in programmable networks and machine learning. Our aim is to show that the network need not be manually pre-configured for resource sharing amongst applications; instead, it can autonomously deduce application experience at run-time, and provide assistance as and when required to specific traffic streams, thereby restoring user experience in a self-driving manner (aka without any explicit signalling).

We begin by outlining the architecture of our system that uses a trained machine to dynamically deduce the application state and apply corrective actions when application performance deteriorates to an unacceptable state (§6.2.1). We then prototype our system and apply our state inference methods to two applications, namely Netflix video streaming (that is sensitive to network bandwidth) and Gaming (that is sensitive to network latency), and show that network assistance can protect application experience in a timely manner in the face of changing traffic conditions, without requiring any explicit signalling (§6.2.2). Our work paves the way towards a network that can self-manage user experience without human configuration.

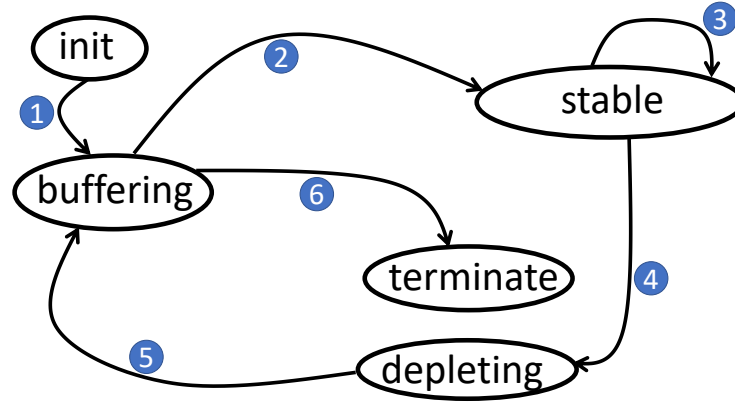


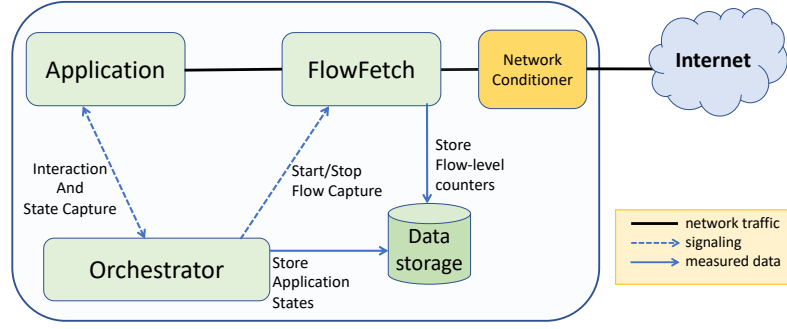
Figure 6.2: Example of performance states transition for a video streaming application.

6.2.1 System Architecture and Design

To realize a self-driven network assistance, three tasks are needed to be performed automatically and sequentially: (a) “measurement”, (b) “analysis and inference”, and (c) “control”, as shown by a closed-loop in Fig. 6.1.

In our architecture, a programmable switch is placed in-line on the link between the access network and the Internet. In a typical ISP network, this link is the bottleneck (and hence the right place to do traffic shaping) as it multiplexes subscribers to a limited back haul capacity. First, traffic of a desired application (*e.g.*, video streaming) is mirrored to *FlowFetch* module which exports flow-level counters (measurement) to a classifier model. Next, the network telemetry data is used by a classification model to determine the current state of application (analysis and inference) and feed the state-machine module. If a critical event of the application behaviour (*e.g.*, video re-buffering) is detected by the state-machine an assist request is sent to *actor* module. Lastly, the actor requests changes (*e.g.*, queue provision) to the switch controller which in turn sends *FlowMod* messages to the switch, executing the corresponding action.

In order to automatically infer the performance (*e.g.*, quality-of-experience) of an application, we model its network behaviour using a state machine. Every application begins in state “*start*”, when its first packet is seen on the network. Subsequently, it transitions into different states depending on the type of application. We illustrate in Fig. 6.2 an example of performance state-machine for a video streaming application as a sequence of states: *init* → *buffering* → *stable* → *stable* → *depleting* → *terminate*. Depending upon the policies of

Figure 6.3: *AppAssist* data collection tool.

network operator for video streaming, a required action can be taken automatically at any of these states (*e.g.*, when it is found at depleting state, a minimum amount of bandwidth is provisioned to corresponding flows, till the application comes back to its stable state)

Data Collection

To realize such a system architecture, we first need to acquire network activity data for the applications of interest, labelled by their behavioural states. This enables the network operator to train classifiers and build state machines which can infer application behaviour without the need of any explicit signals from either the application provider or client. We have developed a tool for generating application dataset – the high-level architecture of our tool is shown in Fig. 6.3. It consists of three main components namely *Orchestrator*, *Application* player, and *FlowFetch*. An implementation of this tool was presented in §4.2.1 to collect video streaming datasets. The orchestrator performs two tasks: (a) initiates and runs the application instance, and keeps track of its behavioural state, and (b) signals to the FlowFetch for recording the corresponding network activities (*i.e.*, time-trace of flow counters). The optional network conditioner module can be used to impose (synthetic) network conditions such as limited bandwidth or extra delays to capture various behaviours of the application.

Labelling Application States: As mentioned earlier, important application states are needed to be labelled since they help the state machine determine when a network assist is required. For example, stall/buffer-depletion, high latency, and lag/jitter are crucial states for video streaming, online gaming, and teleconferencing applications, respectively.

Having identified the important behavioural states of an application, the orchestrator is configured to detect and label these states. In prior research, authors have used GUI interaction tools [104], javascript APIs [105] and web automation tools (*e.g.*, Selenium library) [17, 18, 97] to automatically interact with the application and capture its behaviour.

Measuring Network Activity: The network activity of applications can be measured in several ways ranging from a basic packet capture (expensive recording and processing) to proprietary HTTP loggers combined with proxies (limited scalability). We propose a method that strikes a balance by capturing flow-level activity at configurable granularity using conditional counters. This method stores less data due to aggregation on a per-flow basis, and can be deployed using hardware accelerators like DPDK or be implemented in the data-plane using P4 [30].

Our FlowFetch tool (also described in §4.2.1) records flow-level activity by capturing packets from a network interface. The output records will form the training dataset. We added a telemetry function primitive called as conditional counters in which each flow (*i.e.*, 5-tuple) has a set of conditional counters associated with it – if an arriving packet satisfies the condition, then the corresponding counter increments by a defined value. For example, a counter to track the number of outgoing packets greater than a volume-threshold (important to identify video-streaming experience [107]). Similarly, other basic counters (without any explicit condition) to track volume of a flow can be defined. The set of counters are exported at a configurable granularity (*e.g.*, every 100ms) – it depends on the complexity of application behaviour.

State Classification and State Machine

The training set consisting of multiple labelled application runs is used to train and generate a model which will classify the application state given its network activity patterns. Certain states can be identified from prior knowledge of application (*e.g.*, video streaming always starts in buffering state). For other states which require pattern recognition on the network activity, it requires to extract important traffic attributes computed over a time window (say 10 seconds) and build an ML-based classifier (as presented in chapter 3 and 44). Thus,

the State Classifier is composed of rule-based and/or ML-based models which together classify application's current state that is passed as an update to the state-machine, shown in Fig. 6.1.

State Machine Generation: The state machine of the application is generated using the behavioural state labels available in the dataset along with corresponding transitions. We note that all possible transitions may not occur for an application during data collection, and hence we need to edit the state machine manually.

Experience-Critical Events Annotation: The state machine that models application behaviour needs to be annotated with Experience-Critical (EC) events that require assistance from the network. When such events occur within the state machine, a notification is sent out to the *Actor* module (in Fig. 6.1). There might be multiple types of EC events. For instance, a transition to “bad” state (*e.g.*, buffer depletion for video streaming) or spending long time in a certain state (*e.g.*, prolonged buffering) indicate QoE impairments, and thus are considered as EC events.

Actor: Enhancing Experience

Upon receiving assist requests from the State Machine, the Actor is responsible for enhancing the performance of the application via interaction with the Switch Controller. Typically the application's poor performance can be alleviated by prioritizing its traffic over others in a congested scenario. This can be done in multiple ways including but not limited to: (a) strict priority queues where priority levels are assigned depending on the severity of the assist requests, (b) weighted queues where more bandwidth is provisioned to applications in need, or (c) use packet coloring and assigning different drop probabilities to different colors, *e.g.*, a two-rate three-color WRED mechanism [heinanen1999two]. Assisting methods are confined by the capability of the programmable switching hardware and the APIs it exposes. Nonetheless, the actor needs to request the switch controller to map the flow(s) of the application to the prioritizing primitive (changing queues or coloring using meters, etc.).

Note that the assisted application needs to be de-assisted after certain time for two

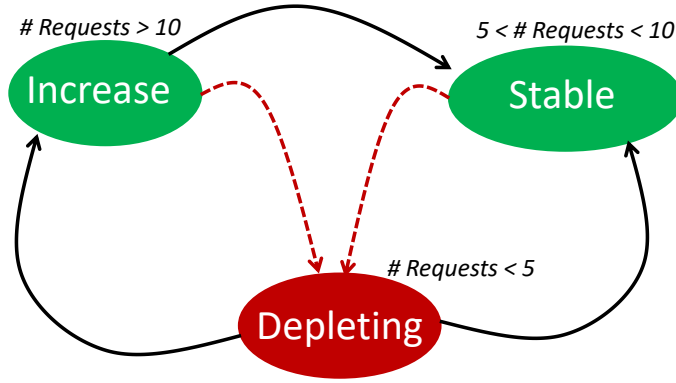


Figure 6.4: Buffer-based state machine for video streaming.

reasons: (a) to make room for other applications in need (to be prioritized), and (b) the performance (QoE) of the assisted application has already improved. However, doing so might cause the application to suffer again and thus results in performance oscillation (*i.e.*, a loop between assistance and de-assistance). To overcome this, we propose that the de-assisting policy could be defined by the network operators using the network load (*i.e.*, link utilization). A primitive policy could be to de-assist an application when the total link utilization is below a threshold, say, 70%. This would ensure that the de-assisted application has enough resources to (at least) maintain the experience, if not improve it. These policies could be further matured depending on the number and type of applications supported and also various priority levels defined by the operator.

6.2.2 Assisting Sensitive Applications

We now implement our framework and assist two applications, namely, Netflix (representative of bandwidth sensitive video streaming) and ping (representative of latency sensitive online gaming). Although ping is relatively simple when compared to actual gaming applications, we note that the requirement of the application still remains the same, *i.e.*, low latency. In what follows next, we describe our measurements, state classification models of the applications behavior, and subsequently elaborate on assistance methodology which enhances the user experience.

Dataset and State Classification

Dataset: We use our data collection tool, shown in Fig. 6.3, to orchestrate sessions of Netflix video streaming and ping as follows. For Netflix, we use a web client on a chrome browser (*i.e.*, the Application block in Fig. 6.3) which is controlled by a python script (*i.e.*, the Orchestrator) using Selenium web automation library. The network data is collected using our FlowFetch tool described in §6.2.1. The orchestrator also captures user experience by enabling a menu that offers multiple video playback metrics. We detect bad experience in terms of buffer depletion which often also leads to bitrate degradation as the video client adapts to poor network conditions. Prior studies [97, 105, 106] have found that chunks transfer in a flow starts by an upstream request packet of large size (other small upstream packets are generally ACKs for the contents received). To capture such transfers, we employ three conditional counters: “ByteCount” transferred both downstream and upstream, “PacketCount” both downstream and upstream, and “RequestCount” for upstream packets greater than a threshold (say, 500 Bytes). We collected these flow counters every 100ms, over 6 hours worth of Netflix video playback.

For gaming (represented by ping), the experience metric, latency, is measured both at the client-end and in the network using the FlowFetch. On the client, we have built a python wrapper which reads the output of the ping utility. On the network, the FlowFetch keeps track of the ICMPv4 flow using the 4-tuple sourceIP, destIP, Protocol and ICMP ID. It calculates the latency by subtracting the timestamp in request and response packets. We note that the latency measured from network is slightly lower than measured on client as it does not include the latency in the access network.

Classifying Buffer-State for Video Streaming: In our dataset, we have observed that Netflix client: (a) in buffer-stable state, it requests one video chunk every 4 seconds and an audio chunk every 16 seconds, (b) in buffer-increase state, it requests contents at a rate faster than playback, and (c) in buffer-depleting state, it requests less number of chunks than being played. Given this knowledge of Netflix streaming, we devise a decision tree-based classifier for the count of requests over a window of 20 seconds. To maintain the buffer level over this window, the Netflix client should ideally request for 7 chunks, *i.e.*, 5 video chunks (of 4 second duration) and 2 audio chunks (of 16 second duration).

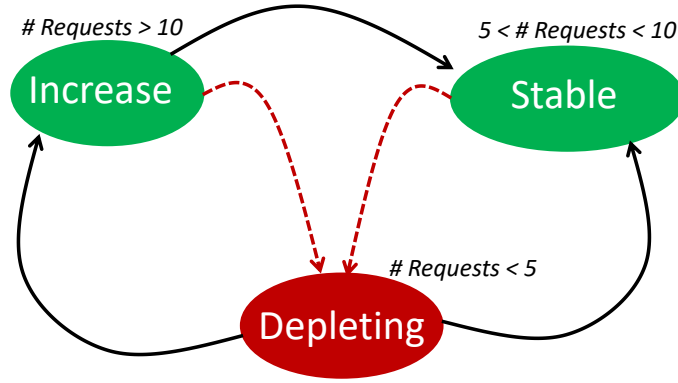


Figure 6.5: Latency-based state machine for online gaming.

Thus, this naturally indicates a threshold to detect buffer increase (>7 chunk requests) and buffer depletion (<7 chunk requests). However, in practice, deviations from ideal behaviour are observed – we, therefore, built our decision tree by slightly broadening the threshold values as depicted in Fig. 6.4.

We acknowledge that it is needed to have a combination of counters and statistical techniques to isolate chunk data and extract features to predict bitrate switches and buffer stalls, as showed in chapter 4, to ideally capture the experience of video streaming. However, the scope of this chapter is limited to develop a framework for automatic assistance of applications by acting upon triggers detected by real-time network measurement. Hence, we use simple request counters based state to demonstrate the self-driving application assisting framework which can additionally incorporate any number states reported by sophisticated models and assist the applications when experience-critical events are detected.

Classifying Latency-State for Gaming: In multiplayer online gaming applications, an important experience metric is latency which represents the end-to-end delay from the gaming client to either the servers or other clients (*i.e.*, peers). The latency (also referred to as “lag”, “ping rate”, or simply “ping”), arises by the distance between end-hosts (static), and congestion in the network (dynamic) which causes packets to wait in queues. Our solution attempts to alleviate the gaming performance by reducing the delay caused in congested networks. Although the latency requirements differ depending on the type of game being played, typically at least a latency of under 100ms is desired to have a smooth experience [Latency] – although top gamers prefer a latency of up to 50ms. Using the latency measurements, we define three states of gaming, *i.e.*, “good” (0-50ms), “medium”

(50-100ms) and “bad” (>100 ms), as depicted in Fig. 6.5 – these latency ranges were reported by players of various popular gaming applications such as Fortnite, Apex Legends and CS:GO. Any transition into the bad state triggers a notification requesting an assist to the actor.

Performance Evaluation

With state machines and classification models built, we now demonstrate the efficacy of our framework by implementing the end-to-end system from measurement to action in a self-driving network. Our lab setup consists of a host on the access network running Ubuntu 16.04 with a quad-core i5 CPU and 4 GB of RAM. The access network is connected to the Internet via an inline SDN enabled switch (*i.e.*, Noviflow model 2116). On the switch, we have capped the maximum bandwidth of the ports at 10Mbps. We have pre-configured three queues (*i.e.*, A, B, and C) on two ports (*i.e.*, P1: upstream to the Internet and P2: downstream to the access) which are used to shape the traffic, assisting sensitive applications. Queue A, is the lowest-priority default queue for all traffic and is unbounded (though maximum is still 10Mbps). Queue B has medium priority and Queue C has the highest priority. This means that packets of the queue C are served first, followed by the queue B, and then the queue A.

We acknowledge that the queueing primitives we use in this prototype are limited to just prioritization. In the next section, we describe a more robust queueing control system that dynamically distributes weights of a weighted fair queueing (WFQ) scheduler depending on the application performance.

We now set up a scenario with three applications – Netflix client on Chrome browser representing video streaming application, **ping** utility representing gaming, and **iperf** to create cross-traffic on the link. First, we use the applications without any assistance wherein all network traffic is served by one queue without prioritizing any traffic (*i.e.*, best-effort) – performance of applications is shown in Fig. 6.6.

The flow of events is as follows. At $t=0$, we start a ping to 8.8.8.8 – this traffic persists during the entire experiment (400 seconds). At $t=10$, we launch the chrome browser and

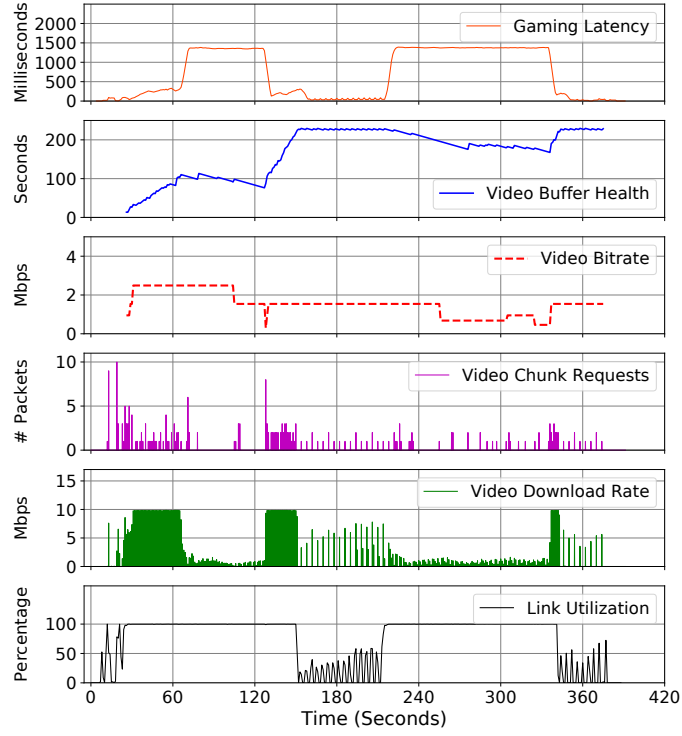


Figure 6.6: Performance of sensitive applications without network assistance.

log in to Netflix. We observe that ping latency (shown by solid orange lines), which is initially at around 2ms, starts increasing to 100ms once the user logs into Netflix. The user, loads a Netflix movie (“Pacific Rim”) and starts playing it at $t = 30$. From this point onward, we observe that the ping latency rises up to 300ms, and Netflix requests chunks and transfers contents at its peak rates (purple lines) – the link utilization hits 100%, as shown by solid black lines in the bottom plot. On the Netflix client, we see that the buffer-health is increasing slowly (solid blue lines), and the client selects the highest available bitrate of 2560 kbps (dashed red lines).

At $t = 70$, we initiate a downstream flow of UDP traffic with a max rate of 9 Mbps using the iperf tool to create congestion. We immediately notice that both sensitive applications start to suffer with the link utilization remains at 100%. The buffer level on the client starts depleting from 110 to 100 after which the Netflix client switches to a lower video bitrate. The video client does not request enough chunks as shown by a gap in the purple curve. It only starts sending out requests again at around $t = 100$, when the video bitrate dropped. The ping suffers even more and the latency reaches to 1300-1400 ms. Once the download finishes at $t=130$, we notice that the video starts to ramp up its buffers, but at a lower

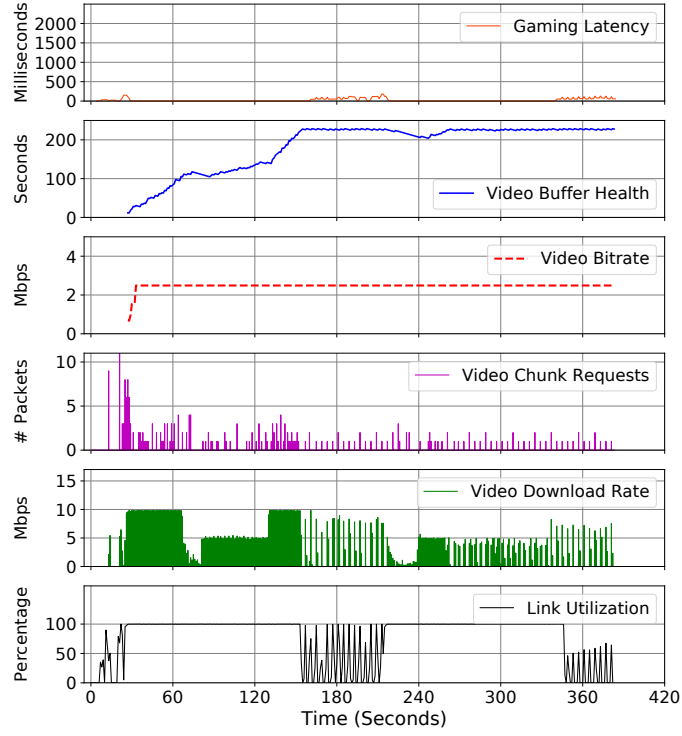


Figure 6.7: Performance of sensitive applications with network assistance.

bitrate (because it just detected poor network conditions) and reaches the stable buffer value of 4-minute at around $t = 140$. The ping also displays a better performance with the latency between 300-400ms (during video buffering), but it gets even better dropping to 100 ms when the video enters into its stable state.

At $t = 220$, we initiate another UDP traffic stream which makes the applications suffer again. This time, we notice that video transitions into buffer-depleting state from buffer-stable state. Again we observe gaps in video chunk requests, clearly indicating decrease in buffer, and subsequently the video download rate falls below 2 Mbps. Ping reacts similarly by reporting the latency of over a second. Upon completion of the download, we note that both sensitive applications display an acceptable performance.

For our second scenario, we demonstrate the automatic assistance from a self-driving network which continuously monitors the applications states and intervenes whenever needed. In our prototype, we allocate the highest priority queue C to gaming applications, which will ensure reduction in latencies. The video streaming flows, when require assistance, are served by the queue B. Note that we configured the max-rate on the queue B at 4 Mbps – when exceeded, the priority of exceeded packets becomes equal to of the

queue A. The need for such a mechanism is due to the elastic nature of video streaming application, it will take up as much bandwidth as available. In other words, it will throttle the default traffic to almost 0.

With these settings, we notice a significant improvement in the experience of both sensitive applications as shown in Fig. 6.7. As described earlier, we start with only ping where it reports a very low latency (*i.e.*, $< 5\text{ms}$). Logging into Netflix at $t = 20$ causes ping latency to go beyond 100ms. First, the classifier finds the gaming application in the medium state (a transition from the good state) which results in a request for assistance. The actor elevates the ping experience by shifting its flow to the queue C. Following this action, we observe that the ping latency immediately drops back to around 2ms. Meanwhile, the video stream starts and is detected to be in the buffer-increase state, given the large number of chunk requests. At $t = 70$, when the UDP iperf traffic (*i.e.*, download) is introduced, we note that the buffer depletes and no chunk requests are sent for a few seconds. Our classifier now detects the video state at buffer-depleting which initiates an assist request. Within a few seconds, all flows corresponding to the video stream are pushed to the queue B. Upon assisting the video, we observe that buffer starts to rise again. Note that the buffer rises slower this time because Netflix application gets about 4-5 Mbps due to the queue configuration. Nonetheless, this ensures that the video performs better without heavily throttling the download on the default queue. When the download stops, the buffer steeply rises till it enters into the stable state. At this point, latency values go up to 100ms. This happens due to de-assist policy which pushes back the applications' traffic to the default queue as the link utilization falls below the 70% threshold (for video) and 40% threshold (for gaming) respectively.

At $t = 220$, the iperf generates traffic again. As soon as the ping values go above 100ms, the ping flow is assisted, and thus its performance is improved. Similarly, the video application is re-assisted as it is found in the buffer-depleting state. This time we note that the video buffer fills up very quickly, taking the application back to its stable state. Note that the video stream is not de-assisted since the iperf traffic is still present (*i.e.*, high link utilization), and the video download rate is capped at around 4-5 Mbps. Once the download traffic subsides (and thus the link utilization drops), both video stream and ping traffic are pushed back to the default queue A.

6.2.3 Summary

Packet networks are agnostic to applications, which have served to keep the Internet infrastructure simple and scalable over the past several decades. However, the best-effort model is now seen as an inhibitor to meeting user experience expectations for the diverse applications such as streaming video, gaming, browsing, and social media. Current methods for prioritization of certain application types are static, and do not react to changes in network conditions or user experience. Sensitive applications need dynamic prioritization from a self-driving network that reacts to changes in user experience.

In this subsection, we have proposed an architecture (called *AppAssist*) for continuous monitoring and dynamic control over the performance of sensitive applications. We have developed data-driven models for the behavioural state of applications in real-time. Lastly, we showed how our scheme is able to detect performance deterioration and take remedial action for two popular sensitive applications, video streaming and gaming. In the next subsection, we describe an improved and scalable self-driving network which leverages programmable networks and dynamically allocates resources across traffic classes in an application fair manner.

6.3 AutoQoS: Application-aware automatic QoS configuration

Few ISPs nowadays actively aim to improve the QoE of their customer traffic. Clearly, this is not due to a lack of tools: nearly all modern network devices support advanced QoS policies including traffic classifiers, policers, shapers, queuing and scheduling policies, together with active queue management disciplines. The problem lies instead in *how* to use these tools to *consistently* improve the QoE of an always-varying traffic mix: an optimal configuration today might end up detrimental tomorrow.

In this subsection, we present *AutoQoS*, a system that dynamically adapts the QoS policies running on a switch so as to optimize application performance. To do so, *AutoQoS* leverages modern programmable networks to extract rich telemetry from the data plane

that is indicative of application performance and dynamically adapts the parameters of QoS schedulers so as to manage resources in an application-fair manner. We implement *AutoQoS* in both software and hardware testbeds. Our evaluations show that *AutoQoS* closely approximates optimally-tuned QoS configurations and outperforms static QoS policies and active queue management disciplines in various traffic scenarios.

A modern Internet service provider (ISP) serves a diverse range of customers who run multiple applications. Due to the recent pandemic confining users to their households, home networks have seen their peak rates climb up and users have experiencing sustained congestion[3]. In such scenarios, ISPs often receive complaints from their users like: “there’s a high delay on conferencing calls”, “web pages load slowly” or “my game download takes forever”. At its core, the problem is that multiple applications with different behaviours and requirements struggle for bottleneck resources, where some applications suffer more than others.

To manage the congestion and combat these issues, ISPs can potentially take the following approaches: (1) active queue management (AQM) techniques, and (2) QoS policies. AQM is a queue management discipline that aims at reducing standing queue lengths and queuing delay [175]. Since AQM is relatively easy to deploy, ISPs have rolled it out across the edge network (where traffic gets queued predominantly [176]). AQM is neither aware of applications nor does it actively distribute resources among them. Thus, congestion may still impact some applications more than others. Indeed, ISPs have observed fewer “delay”-related issues [136]. However, customers can still complain about “bandwidth”-related issues, for instance, “my game download still takes ages!”.

Therefore, in addition to AQM, ISPs now require employing QoS policies to balance application needs. First, they need to broadly classify application into categories like conferencing calls, video, web browsing, and large downloads/file transfers using existing classification tools such as DPI [177]. Simply prioritizing one class over another is not a desired solution as it can starve the least-prioritized application(s) during congestion (a shortcoming of the prototype presented in the previous section). Thus, ISPs need to attempt to allocate different bandwidth fractions to each application class. This is easier said than done, as it is far from trivial to estimate the optimal distribution of resources.

While operators can rely on historical traffic data, picking an optimal distribution that suits all scenarios is difficult.

Different traffic mixes require different optimal static QoS configurations. Not only is the traffic different during the day and night (e.g., switching between conferencing applications and entertainment applications), but may also be impacted by irregular events. For instance, networks were clogged up when a popular show was launched on Netflix[178]. Similar occurrences also prevail when a popular game (such as Fortnite) launches a new update requiring users to download 10s of gigabits of files [179]. This means that the network operator would need to tune the distribution frequently.

In summary, to tackle the high traffic demand and consequent congestion, deploying AQM is insufficient and configuring QoS (finding the optimal resource allocation) is hard for dynamic traffic distributions. We require an equally dynamic system that is aware of application performance and can automatically adapt QoS parameters accordingly. While there exist several tools to classify the application traffic (e.g., modern DPI appliances [177, 180]) and widely developed packet schedulers supporting schemes like priority queuing and weighted fair queuing [176], there is a lack of: (a) in-network, per-class measurements that can reliably indicate application performance; and (b) a feedback loop that can automatically adapt QoS parameters (i.e., the resource distributions between classes) based on these measurements.

To this end, we build *AutoQoS*, which measures appropriate per-class metrics in-network that approximately indicate application performance and automatically reconfigures QoS parameters to provide a fair application performance during congestion. It is essentially solving the bottleneck resource distribution problem using a control loop consisting of:

- **Sensing:** Leverage modern programmable networks to extract sophisticated metrics (beyond queuing delay) indicative of application performance.
- **Decision:** Translate the network metrics into a normalized score which indicates relative application performance. Then, decide how to distribute the resources to provide fair performance across applications.

- **Actuation:** Enforce decisions by changing the appropriate QoS parameters to adapt to the current application traffic mix.

While methods and tools exist in prior work to tackle individual aspects separately, we combine them and build a dynamic re-configuration QoS system to better manage application performance in varied congestion scenarios. We implement *AutoQoS* on both software and hardware test beds running real application traffic. We evaluate *AutoQoS* and show that:

- It outperforms widely deployed FIFO, FQ_Codel (an AQM approach [181]) and priority based schemes in different traffic scenarios;
- By dynamically adapting the weights, it performs very close to an optimal fixed QoS policy;
- It allows for deviation in the normalizing functions *i.e.*, the functions need not precisely map to scores; they can be approximate; and,
- It is implementable in programmable hardware today.

6.3.1 Overview

In this section, we provide an overview of how *AutoQoS* distributes bottleneck resources in an application-aware manner and automatically configures QoS parameters to ensure a fair performance. *AutoQoS* has five major components: (a) traffic classification identifying broad classes of applications, (b) weighted queues for each class, (c) per-class telemetry logic that extracts relevant metrics, (d) mapping functions that map metrics to performance scores and finally (e) online adaptation logic that redistributes the weight.

Example. Consider a small network serving a few households, each of which predominantly uses a different kind of application: household 1 uses VoIP like apps and web browsing, household 2 does a lot of web browsing, and household 3 consists of a gamer downloading a huge update. We begin by briefly describing the application behaviour, followed by a walk-through of the system.

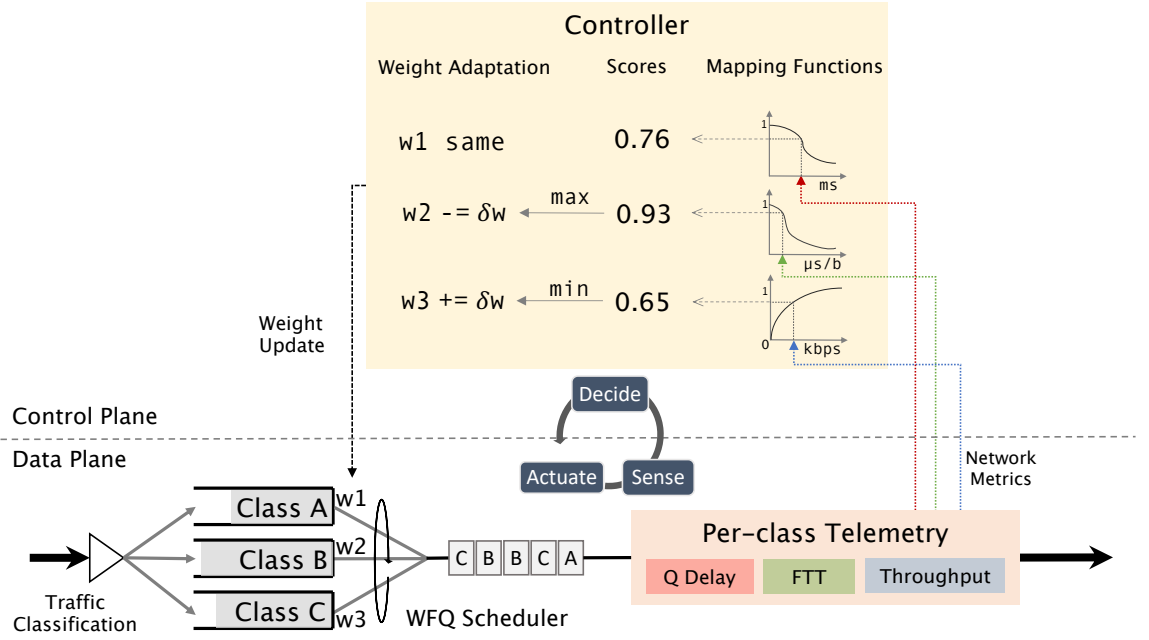


Figure 6.8: AutoQoS System Overview.

Applications. Our example traffic can be divided into three classes. VoIP applications typically use UDP and send CBR-like traffic requiring low end-to-end delay to avoid impairments like lag or echoes. They fall under the category “delay-sensitive” applications (class A in the following). The other two applications, web browsing and file transfers both predominantly use TCP at the transport layer, yet their behaviours differ significantly. Web browsing is a “short-and-bursty” application (class B in the following) and uses multiple TCP flows to load many small files (*e.g.*, CSS, Javascript, Images). It requires the objects to be transferred quickly, resulting in a responsive page load. In contrast, download clients use few flows to fetch large files at a maximum rate from the servers, thus requiring high throughput. This is a “throughput-intensive” application (class C in the following).

Congestion Behaviour. During congestion (assuming a drop-tail FIFO queue), these applications compete for resources at the bottleneck queue. Depending on the traffic distribution, one application may suffer more than the other. For instance, due to the presence of both browsing and download flows creating standing queues (assuming TCP Cubic like behaviour), the VoIP calls face an extra delay. Further, since two households (1 and 2) are doing browsing it may dominate the download initiated by household 3. To tackle such a scenario, we need a system to measure the appropriate metrics from

network which can indicate the performance levels of each of these applications and adapt the resource distribution accordingly.

Inputs. *AutoQoS* assumes the incoming traffic is classified into broad classes introduced above. For each class, *AutoQoS* receives as input a telemetry function that exports relevant metrics and a mapping function that maps metrics to a normalized score. Furthermore, the control plane requires two parameters: (a) *epoch* and (b) δw . *AutoQoS* adapts the weights of the per-class queues by δw every *epoch*. In this example, we use *AutoQoS*' default values of $\delta w = 5\%$ and one second *epoch* i.e., every second the weight is adjusted by 5%.

Data plane. Each class is sent into a separate queue, and the scheduler dequeues according to the weight of the class-level queues. After dequeuing, *AutoQoS* executes telemetry functions (system input) on a programmable data plane on the traffic of each class resulting in different network metrics being exported to the controller. For instance, we measure queueing delay experienced by the delay-sensitive traffic while we measure *Flowlet transfer time* (FTT) for bursty traffic that measures how quickly flowlets (approximately mapping to web objects) are transferred on the network.

Control plane. The metrics are received by the controller, which first maps the metric values (measuring different values) to a normalized score using mapping functions (also system input). The controller combines all the scores and compares them to realize that class C (gaming downloads in the example) performs the worst and class B (web browsing) performs best. It adapts the queue weights by transferring a weight quanta (δw) from class B to class C. These weight updates are then pushed to the scheduler in the data plane, resulting in more resources being given to the gaming downloads. This process (sensing→deciding→actuation) repeats every *epoch* to approximately converge at a max-min fairness across applications.

6.3.2 System Design

In this section, we describe the design of *AutoQoS*, including traffic classification, data plane telemetry and the control plane adaptation. We discuss requirements of each component

followed by key insights that lead to our design.

Traffic Classification

Incoming traffic needs to be grouped into a class depending on the application type and their network requirements. This enables the network to monitor and manage the classes of traffic individually. An exemplary set of classes used in this chapter consists of delay-sensitive class (VoIP traffic), short and bursty class (web browsing), and throughput-intensive class (file transfers). Note that a particular application type (*e.g.*, VoIP) implemented by different developers (*e.g.*, Whatsapp, Messenger, and Skype) can be grouped into the same class. While the classifier design is beyond the scope of this chapter, we recommend the use of modern DPI appliances [177, 180] or our proposal FlowFormers (chapter 3) to accurately detect a wide range of applications on the Internet. They can be used (beyond accounting or zero-rating [114]) to manage application requirements using our framework.

Data Plane: Telemetry

Post-classification the application traffic needs to be monitored to know if and by how much they are suffering at the bottleneck. To do so, we need metrics that are computable in-network and can closely approximate the performance of the application. For instance, while queueing delay may approximate the performance of delay-sensitive applications like VoIP, online multiplayer gaming, it may not be sufficient to measure web page loading times. A web page loads by fetching multiple small web objects, and the quicker they are fetched, the lower time it will take for the page to load. While a low queueing delay is required for browsing too, it does not directly translate into the application’s performance. Instead, measuring “flowlet transfer times” (as we show in the next section) is a better metric that incorporates both delay and available capacity and measures the time it takes for flowlets to transfer (which actually carry the web page content).

We leverage the flexibility of modern programmable data planes to collect per-class telemetry. The system first matches on the traffic class (can be present in a network header

field) and then executes appropriate telemetry functions written in the P4 language [30]. The telemetry functions can range in complexity from simply exporting metadata about queues to using stateful resources to execute per-flow measurements (as we show in the next section). The emerging programmable switches, being limited in compute and memory resources, cannot implement any arbitrarily complex functions but offer enough flexibility to explore metrics beyond basic queue stats. The collected metrics can be exported to a control plane either via special telemetry packets (e.g. INT[43]) or using DMA channels which then use the per-class telemetry to decide on the resource distribution.

Control Plane: Performance Mapping

The metrics being measured at the data plane now need to be mapped to a normalized performance score. Let us say the metrics are a queueing delay of 50ms or an FTT of $1 \mu s/byte$. What does this value mean to an application? Can the VoIP call work with such delays, or are the web pages loading "fast-enough"? To answer these questions, there must be a function that maps the measurement to a normalized performance score, say between 0 and 1. While metric indicates "what to measure?", the mapping function lets the network know at what "level" the application is performing. For instance, a VoIP call's queueing delay can be mapped using a simple thresholding function like: $delay < 20ms \implies 1$, $20 < delay < 50ms \implies 0.7$ and $delay > 50ms \implies 0.3$. With such a mapping function, the network can now measure and compare the performance of competing applications. While they might not accurately represent the exact end-user perceived performance, they serve as a "good-enough" proxy for the network to distribute resources at the bottleneck.

We group both the metric and mapping function together to make the *AppNet* profile of an application class. *AutoQoS* takes multiple *AppNet* profiles as input. They bridge the gap between what can be measured in network and what performance the application gets. In other words, an *AppNet* profile specifies what to measure (using a language like p4) and how to map it to a normalized performance score (using a mathematical function). *AppNet* profiles are dependent on the applications. However, applications in a particular category tend to have very similar metrics and performance functions. For instance, VoIP and online gaming fall under the "delay-sensitive" category and require low and consistent

delays. So *AppNet* profiles can be defined on a coarser per-application category level. A control plane can convert the metrics to performance scores with these profiles.

Control Plane: Resource Distribution

With the metrics and performance scores streaming in, the network control plane now needs to act and manage the performance of applications at the bottleneck link. The main challenge here is the heterogeneity of applications at the last-mile network. Modern applications use different transport level protocols (UDP/TCP), have different CCAs, and connect to servers with varying RTTs [176]. A control plane should ideally help alleviate the degradation in performance experienced by different applications by distributing network resources efficiently.

Overview. Fig. 6.8 shows a high-level view of AutoQoS’s control plane. It receives the mapping functions as an input at startup along with the parameters *epoch* and *weight quanta* δw . Every *epoch*, the control plane uses mapping functions to map raw metrics to performance scores of all applications. The controller then picks the worst and the best performing applications by comparing the scores. It then redistributes the weight by taking a *quanta* (δw) of weight from best and gives it to the worst-performing application queue. In this way, the control plane is able to measure the application performance and react to make resource distribution application-fair.

Queueing Primitive. We choose weighted queueing as a mechanism to distribute network resources among applications. We use one queue per application category (more scalable compared to per-flow queues) to which a weight is assigned. Since flows within the same application category tend to behave similarly, their transport level CCA’s are also much more effective to operate within the given fraction of the total resource [111]. Using weighted queueing has several advantages: (a) it uses a work-conserving scheduler *i.e.*, if a certain queue is not active, the resource is distributed to other queues, (b) it directly indicates the proportion of resource given to an application type – which means increasing it results in better performance, (c) it does not create starvation (a common issue with priority queues) and (d) it is a commonly implemented scheduler from home routers to

high-end programmable router chipsets. With weighted queueing primitive, the control plane needs to decide: (a) *what weights to give to different applications?* and (b) *when and how frequently should the weights be changed?*. Answers to both questions decide the optimization strategy to achieve best effort fairness across application categories.

Last-mile Congestion Patterns. To answer these questions, let us understand the type of congestion and its impact that is prevalent in last-mile networks. Firstly, as opposed to micro-bursts in data centers, internet users perceive application degradation caused by persistent congestion (in the order of seconds to minutes) [7]. Secondly, the congestion build-up also happens slowly over time, typically during evenings [176]. Therefore, ISPs need to ensure a good experience for their customers not at sub-second levels but in seconds to minutes when the network is bottlenecked. This means that the control plane need not necessarily adapt every few milliseconds but can operate at the granularities of 100's of milliseconds to seconds.

Optimization Strategy. AutoQoS controller employs a greedy strategy to distribute resources based on the performance scores. It follows a "robinhood-like" approach by taking resources (weight) from the best-performing application and giving them to the worst-performing application. It continuously receives metrics, measures performance, and redistributes weights to ensure fair performance across the applications. Developing a more complex strategy (for instance, a gradient descent algorithm) requires the control plane to know beforehand what impact a certain weight increment has on the application. Given that the application performance is determined from metrics of different dimensions (*e.g.*, delay, FTTs) and the traffic scenario is dynamic in nature, knowing the relation between weight and performance is non-trivial. The greedy approach does not assume any such relation except that it expects the performance of the application class to increase with an increase in weight. This simple naive approach has other advantages like: (a) it doesn't enforce any constraints on performance curve (differentiability, convexity) since it just relies on one point in the curve at a time – also perceived application performance often doesn't adhere to such constraints anyway [182] (b) it continuously measures and changes the weights being adaptive to different kinds of congestion scenarios and (c) it is simple to implement, test and verify.

Challenges of greedy optimization. There are a couple of challenges in using the greedy-approach for the control plane: (1) it may not achieve optimum allocation "fast-enough" (compared to gradient-based techniques) *i.e.*, takes a few *epochs* to converge and (2) it may suffer from oscillations *i.e.*, the *quanta* of weight may be allocated back and forth between two classes. For the first challenge, since the users perceive persistent congestion effects in the order of seconds to minutes, we choose to keep a simple model which might take a few seconds to hit the optimum in favor of avoiding assumptions about the traffic distribution or curve nature (required for gradient-based techniques). For the second challenge, we use a simple heuristic: if the delta of performance score between the highest and lowest class is less than a threshold α (empirically tuned to be 0.05), the control plane does not redistribute the weight. This prevents oscillations and also keeps controller intervention to a minimum.

In evaluation, we show the impact of the inputs to the control plane both by varying the parameters and the mapping functions. We show that *AutoQoS* control plane can provide approximate fairness and isolation across classes while being simple and scalable to implement and deploy.

6.3.3 Implementation: Application Profiles and Telemetry

Having discussed the design of the framework, we describe the application classes, their requirements, and corresponding *AppNet* profiles used in our work.

Our work considers three application categories used on the internet that have diverse requirements from the network: "Delay sensitive", "Short and bursty", and "Throughput intensive" applications. For each class, we answer the questions: (1) how to translate an application's requirements to in-network metrics? (2) how to collect the metrics using modern programmable data planes? and (3) how to map the metrics to a normalizing performance score?

We note that in the absence of developer-provided "AppNet" profiles, we use exemplary utility and QoE curves that depend on different metrics published in the literature. Further, while we use only three classes to demonstrate our system, the process is generally

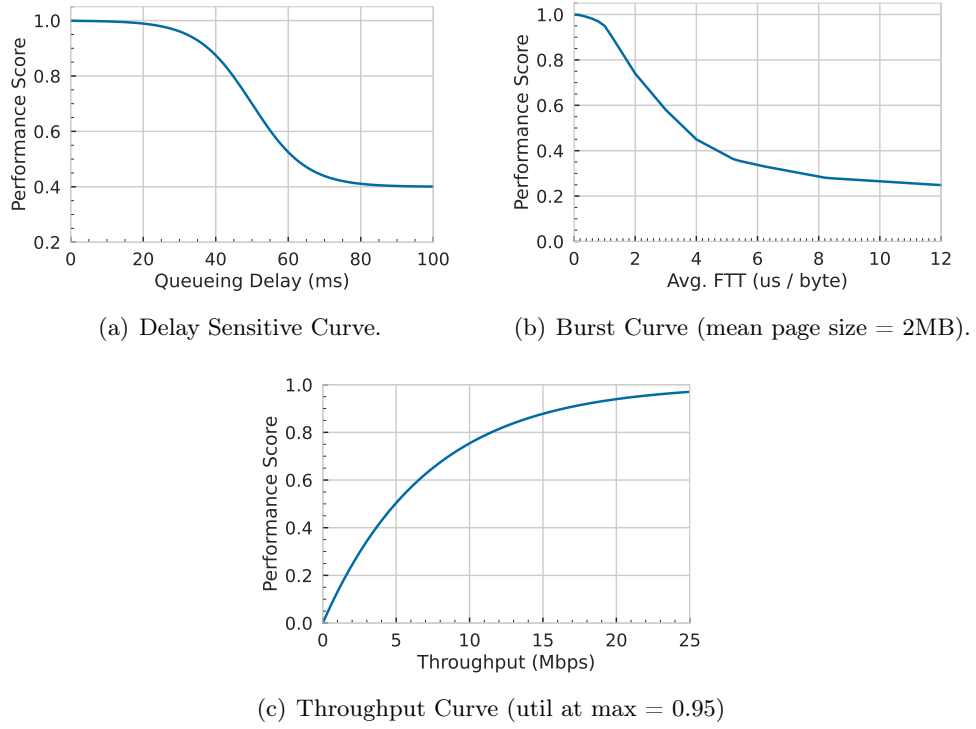


Figure 6.9: Performance Curves of different classes of applications.

applicable to other classes of applications as well (*e.g.*, video streaming, cloud gaming).

Class A: Delay-sensitive

These applications are sensitive to delays between the server and client. Examples include VoIP, Conferencing, and Multiplayer Gaming applications. During congestion, the packets of these applications experience a higher delay (often higher than the RTT) as a result of being queued at the bottleneck (assuming a drop-tail queue). The increment in delay degrades the user-perceived performance. For instance, in a multiplayer game, the user receives a delayed response on the VoIP call or their actions not showing real-impact (*e.g.*, resulting in a kill).

Unsurprisingly, the metric to measure for these applications is the queueing delay experienced by their packets. It can be measured in-kernel (if running on a linux based network element) by keeping track of enqueue and dequeue times (like FQ-Codel). Modern programmable dataplanes have this information available via packet metadata at the egress pipeline. Further, this can also be obtained via link tapping the bottleneck network ele-

ment even if it is not programmable (as done in [183]). In our testbed, we use link tapping in software testbed and packet metadata in programmable switches to obtain the queueing delay metric.

Once measured, the value needs to be mapped to a performance score. We adapt the delay utility curve of a conversational VoIP call presented in [184] which maps end-to-end delay to a utility value. We subtract the baseline one-way delay (from server to client) to convert the x-axis of the curve to represent just the queueing delay (Fig. 6.9(a)).

Class B: Short-and-bursty

These applications are interactive and require quick response times. Web browsing is a popular application in this class that fetches small objects (CSS, javascript) from web servers to load the content of a web page. During congestion, the web page loads slowly, as it takes more time to transfer the objects with standing queues at the bottleneck. Additionally, since they occur in short bursts, they are not active until the time required to obtain a fair share from competing heavy flows for example. Further, since webpages inherently have a dependency structure *i.e.*, html loads first, then loads javascript that in turn is executed to fetch more resources. Even in the beginning, short-term congestion can cause the entire page to load slowly.

Flowlet Transfer Times. Queueing delays, while need to be relatively low for a good browsing performance, do not capture the complete picture. Since web pages transfer web objects from the server, directly measuring their transfer times serves as a better proxy. Since modern browsers use long-lived connections to web servers and transfer multiple objects within a transport connection, we need to measure the time it takes to transfer *flowlets* as opposed to the entire flow. *Flowlets* are smaller chunks of data transferred within a flow separated by at least the timeout of an RTT. They have been studied in the context of load distribution and path selection in data center traffic scheduling[185]. We found that identification of flowlets and measuring their transfer times (normalized to $\mu s/byte$) correlates very well with the actual web page load times *i.e.*, a lower FTT resulted in lower page loads time and vice-versa.

FTT can be measured on the data plane using 3 registers per flow tracking *flowletStart*, *flowletEnd* and *flowletSize*. Whenever a packet arrives that is more than RTT apart, the data plane exports the previous flowlet information to the controller and reuses the register to track the new ones. Similar to other per-flow counting measures, we make use of an array of registers where the index is computed using a hash of the flow's five-tuple. There can be some collisions. However, since the controller takes an average for the entire class, they have minimal impact on the decision.

To map the measured FTT to a performance score, we use the Page Load Time vs. QoE curve presented in [182]. The curve shows a user QoE score from 1 to 5 when a group of subjects were shown a web page loaded under certain conditions. The curve has 3 phases, the first phase where the load time is within 2 seconds, the user score is almost close to 5, between 2 to 6 seconds is where the user is very sensitive and the QoE score quickly drops and beyond six the score remains low and drops very slowly. We can't use that curve directly since the measuring *Page Load time* is very challenging in-network since we cannot identify the start and end of a web page reliably. Instead, we use the metric FTT measured in $\mu\text{s}/\text{byte}$ and map it to a page load time assuming an average web page size (given as parametric input to the curve). So, for example, with a mean page size of 1MB, an FTT of $2 \mu\text{s}/\text{byte}$ leads to a 2 sec load time. We then map $2 \mu\text{s}/\text{byte}$ to the QoE value in the curve for 2 sec load time. Since FTT correlates well with the web page load times, the linear mapping is close to reality. Further, we show (in the next section) that the overall system can tolerate inaccuracies and does not expect a precise performance function which is difficult to obtain in the real world. The mapping function from FTTs to browsing performance is shown in Fig. 6.9(b).

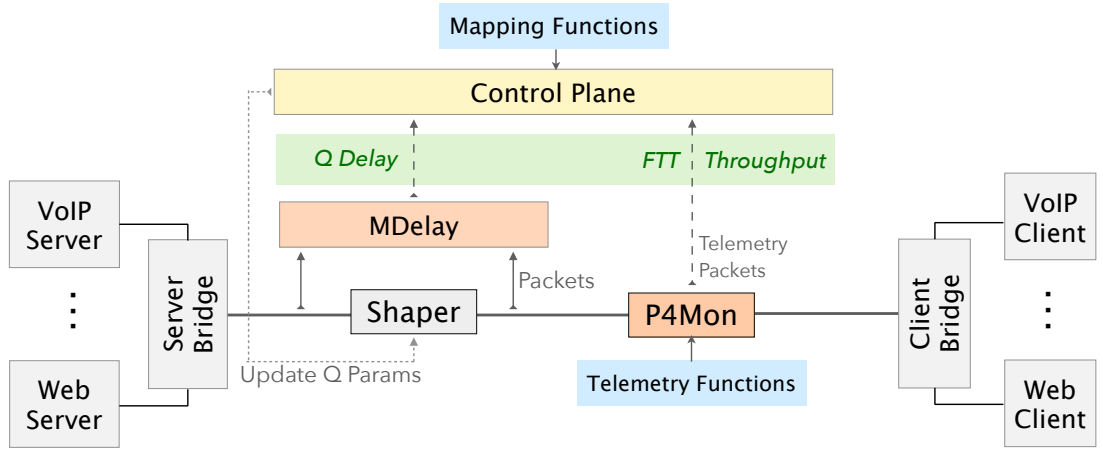
Class C: Throughput Intensive

Throughput-intensive applications typically transfer large files between a server and client. Common examples include gaming downloads, software updates, and cloud backup/restore. During congestion, their speeds drop, resulting in the file transfers/game updates taking longer to finish. Often though, these classes of applications cause congestion since they are elastic in nature *i.e.*, they expand and try to use as much network capacity as possible

(unless the sender is limited, which is rare). To transfer data at highest possible speeds, these applications often use multiple flows established to the server in parallel (2 - 8). We note that using multiple flows takes advantage of per-flow fair systems (such as TCP's congestion control or flow-fair AQM techniques) by grabbing more than fair-share at the application level.

We leverage a programmable data plane to combat such applications to measure the real-time throughput of a client's download traffic independent of the number of flows used. It multiple ways depending on the scale of deployment. One can use an array of registers tracking byte counts of download traffic (indexed by hashing the client IP) and export them to the control plane every second. This will indicate the number of active download clients and their throughput in the last second. Another approach could be used if the client cardinality is very high (*e.g.*, millions of throughput intensive clients), an approximate cardinality can be estimated using linear counting, hyperloglog etc. [186] and the overall throughput of the application class (can be measured using a single counter) can be divided by the estimated cardinality. We used the first approach in our work as the number of clients was not so high.

The utility of a throughput intensive elastic application was proposed decades ago to be an exponential curve with diminishing utility as capacity increases in [187]. We adapt that utility curve by using average per-client throughput as the measurement on x-axis (as shown in Fig. 6.9(c)). While applications of class A and B have some bound to measure the best performance (*i.e.*, queueing delay of 0ms or web-page load time within 1sec can be 1), throughput intensive applications can grab as much as the bottleneck link supports. In practice, however, they are limited on a per-subscriber basis to a "plan" speed such as 25Mbps, 50Mbps or 100Mbps. Therefore, we use per-client throughput which means that a subscribers download application will get a utility close to one when they are getting close to their plan speed (independent of the number of flows used). The operator can configure two parameters: (a) max speed and (b) utility at max speed (typically lies in the range of 0.90-0.99) to configure the slope of the curve (the higher it is more quickly it gets to the max utility). We vary it to a certain degree to show that our framework is tolerable to minor variations in the curve parameters. We note that in this exemplary set of classes, class C can be used as a default class in the absence of a classification since it by default

Figure 6.10: *AutoQoS* Software Testbed.

maps performance to the plan speed that they paid for.

In summary, application developers, standardizing bodies and network researchers can come up with *AppNet* profiles by understanding the application requirements, specifying the metrics to be measured and the mapping function which can collectively make the network "application-aware" and make it respond to congestion in a better way.

6.3.4 Implementation: Software and Hardware Testbed

We now describe both its software and hardware implementation (using Barefoot Tofino) of AutoQoS.

Software Testbed

Our software testbed (shown in Fig. 6.10) consists of 3 major components: Applications (servers and clients), Network Dataplane (Shaper + P4 software switch bmv2[188]) and *AutoQoS* Control Plane. The applications communicate over a link that is shaped using the Shaper and monitored using the P4 data plane and control plane. Since, shaping occurs outside the p4 dataplane (using *tc*[189]), we have an additional component to measure the queueing delay called *MDelay*. The network metrics (shown in green) are exported to the control plane from the programmable data plane (and *MDelay*) configured with the telemetry functions. The control plane ingests the metrics and updates the queueing parameters (i.e. weights of the individual queues) according to congestion scenario and the

specified mapping functions.

The applications in our testbed include VoIP, Web Browsing and File Transfers. VoIP server and client contain wrapper code that runs the open-source PJSIP/PJSUA[190] application in which an audio file is sent from the server to the client using the SIP protocol. At the end of a call, the application reports call QoS stats in terms of RTT (avg/min/max), Jitter and Loss which can be used to assess the VoIP performance. For Web Browsing server-side, we used WebPageReplay[191] a web page caching server which can cache web page objects in the record mode and serve them in replay mode. We used Alexa top 20 webpages[192] to build the cache which includes web pages from search engines, wiki pages, e-commerce and social media websites. On the client-side, we used Google Chrome browser[158] configured to fetch web pages from the web server. Web page load time is recorded using javascript APIs as an indicator of web performance. For file transfers, we use iperf3[193] and both server and client which can use multiple TCP flows to emulate a large download and report both instantaneous and average speed as a measure of performance. All application servers and clients are containerized using Docker[194] to ensure ease of deployment and repeatable tests as they package the application and its software dependencies in one executable container. Additionally, we have written convenience scripts to run experiments using a YAML specification which can orchestrate and schedule applications for a wide variety of experimental scenarios.

The network elements connecting the application servers and clients in the testbed include bridges using OpenVSwitch [195] and the bmv2 P4 programmable software switch[188]. The *ServerBridge* and *ClientBridge* act as an aggregation element to send all traffic via one link. A link itself is created using Linux veth port pairs (an approach Mininet[196] also uses to create links in its emulated network). The Shaper component uses the linux tool *tc* to create a wide range of queueing configurations including FIFO, FQ_Codel[181] and HTB[197] based weighted queueing. We note that the interface that Shaper controls is ensured to be the only bottleneck in the whole network. The P4Mon switch (connected to the other end of veth Pair from the shaped interface) runs the P4 program which measures various metrics according to the application type and sends it to the control plane (using special packets sent via special *cpu-port* on the switch).

The control plane is developed from scratch in Python. It takes the *AppNet* profiles as input and collects network metrics by capturing the telemetry packets sent by the data plane. In addition, it gets the queueing delay metric from *MDelay* using a REST API. The network metrics are collected continuously and are mapped to application performance using the respective mapping functions. When an application seems to suffer more than the other, the control plane adjusts the queueing parameters (weight for the appropriate queue in *tc*) to keep the performance fair across applications. It uses the algorithm mentioned in §6.3.2 to adjust the weights and dynamically adapt to congestion at the bottleneck.

Hardware Testbed

The hardware testbed is essentially the same as the software one except that it substitutes the network data plane with Intel Tofino[15], the Server and Client Bridges exit at a 10G ethernet NIC and the links are copper wires. Since, the hardware switch offers shaping support in its traffic manager, a bottleneck can be created between the ingress and egress pipeline. We have adapted the P4 dataplane program to fit into the resource-constrained hardware. In addition to measuring FTTs and Throughput via P4 program, Tofino also supports the measurement of queueing delay "out-of-the-box" so *MDelay* is not required.

The control plane had to be modified to use the APIs of the programmable switch to configure queues and the parameters. The rest of the operations, using *AppNet* profiles, metric collection and optimization, remain the same.

The hardware implementation demonstrates that the metrics can be collected at scale (also as shown by other prior work) which implies that congestion can be managed using our framework at Access Aggregation Networks that operate at such high scales.

6.3.5 Evaluation

In this subsection, we evaluate the efficacy of our system by studying its behaviour in different scenarios created in our software and hardware test beds. In what follows, we first describe the experimental setup and introduce the algorithms we compare against,

followed by a detailed comparison and sensitivity analysis for both the tunable parameters and the input curves. In particular, we answer the following questions:

1. How does *AutoQoS* compare to other existing schemes?
2. How close is *AutoQoS* to a per-scenario optimal fixed weight distribution?
3. How sensitive is *AutoQoS* to mapping functions?

Experimental Setup

We run different traffic scenarios (as listed in Table 6.1) on our testbeds to evaluate Auto-QoS by varying the following parameters: Number of calls for VoIP flows, Webpage type: small (<1MB) and large (>1MB) and number of browsers for web browsing traffic and number of TCP flows (N_{flows}) for file transfers/downloads. VoIP and Download flows are started at the beginning of the scenario and last until the end of experiment – each run lasts for a minute. In the case of web browsing, we have two access patterns of web browsing: (a) Sequential in which a fixed number of web pages are fetched sequentially with a wait of 1 second between them and it stops once the number of web pages are fetched and (b) Loop in which a set of web pages are loaded in a loop (without any wait time) throughout the experiment.

Scenarios A and B are relatively “light” congestion scenarios where in VoIP calls run in parallel with browsing (A) and 1 TCP download flow (B). Scenario C involves fetching all the Alexa top 20 webpages sequentially with 2 TCP download flows in the parallel. Scenario D1 involves fetching large webpages (>1MB) continuously and 2 TCP download flows with 4 VoIP calls. This introduces a mix of continuously active short TCP flows (predominantly in slow-start phase) and long TCP flows (from file transfer). We tweak D1 to create D2 and D3 where in one application dominates the mix and thereby the traffic mix at congestion varies: D2 being “download-heavy ” and D3 being “browsing-heavy”.

Table 6.1: Experimental Scenarios

Scenario	# VoIP calls	# Web Browsers	Page Size (Access Pattern)	# Download Flows
A	8	1	Small (sequential)	0
B	8	-	-	1
C	-	1	All (sequential)	2
D1	4	1	Large (loop)	2
D2	4	1	Large (loop)	16
D3	4	2	Large (loop)	1

Comparison with existing schemes

We run all the scenarios described above across 4 scheduling schemes at the bottleneck queue: (1) FIFO, (2) FQ_Codel, (3) Priority and (4) AutoQoS. The first three schemes are available in standard linux tool called as *tc* which can enable any of the queueing discipline with configurable parameters. For *AutoQoS*, we use *tc* to create a weighted queueing scheme using Hierarchical Token Bucket (HTB) queueing classes.

Configuration. FIFO queue is configured with a maximum length of 1 BDP. FQ_Codel is a “no-knobs” queueing discipline and comes with standard configuration in which it uses 1024 hash buckets to distribute 5-tuple flows and has a limit of 5ms to control the queueing delay. We use 3 priority queues in the priority order: VoIP > Web Browsing > Download (similar to configurations are typically used in ISP setting wherein VoIP is given the highest priority and large file transfers are given the last priority [176] – we additionally give web browsing a medium priority for quick load times). For AutoQoS, we use the *AppNet* profiles described in §6.3.3 and the parameters $epoch = 1sec$ and $\delta w = 5\%$. *i.e.*, the control plane updates weights every second with a 5% transfer from the best performing class to worst performing class. We will evaluate the impact of these parameters in the next few subsections.

Criteria. For all the scenarios, we use both min-utility and jain fairness index to com-

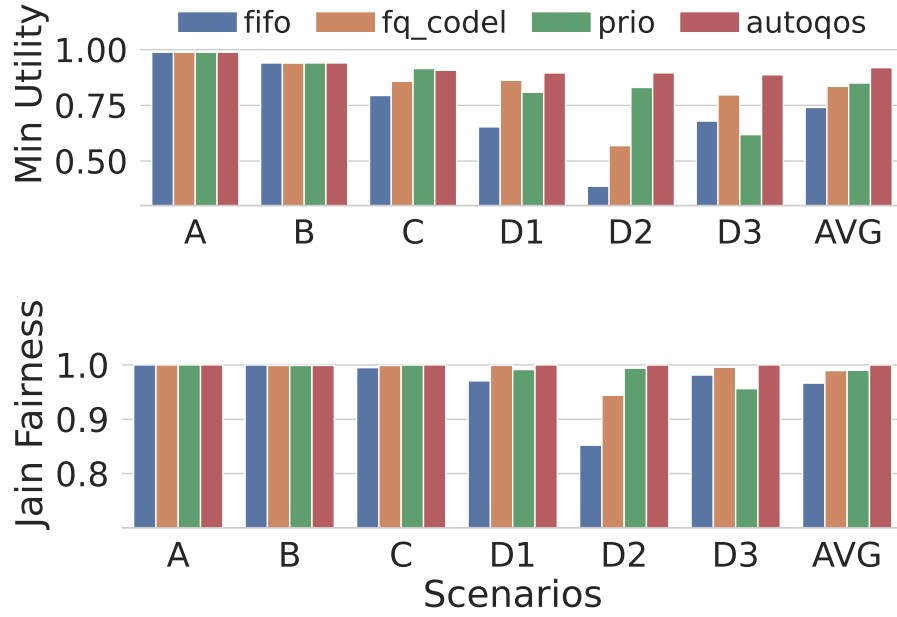


Figure 6.11: Comparison with different schemes across scenarios: Min-utilities (top) and Jain's fairness index (bottom)

pare the performance of the schemes. Since our control scheme aims for min-max fairness, we score the output of the experiment by calculating the utility of each application (as dictated by their curves) and then computing the minimum across all active applications. We also compute Jain's Fairness Index ([198]) using the application utilities (as opposed to transport layer throughput as used in prior work). We note that the scores are computed from the application reported metrics (*e.g.*, the actual web page load time) as opposed to network metrics used in the framework (*e.g.*, flowlet transfer times) to give an accurate performance depiction across the schemes.

Results. We observe that during “light” congestion scenarios (A and B), *AutoQoS* performs similar to other schemes. This is broadly because the queue is not congested and the competition is not high amongst the applications. However, we start observing differences with remaining scenarios. In scenario C, FIFO performs the worst by taking the most amount of time to load web pages and PRIO performs the best. In this “static” scenario, since the volume of webpages transferred is constant, the rest of the capacity is being used by the download flows until the end of the experiment. Therefore, the download rates were very close to each other across all schemes as the volume downloaded was very similar and thus the minimum utility was that of the browsing application. Here,

theoretically, priority queueing is the optimal discipline as it results in fastest browsing time (given fixed number of web pages). However, we note that *AutoQoS* is not far behind from PRIO queueing.

In the rest of the (more dynamic) schemes D1-3, we can observe *AutoQoS* outperforming other schemes. We see in scenario D1 where large webpages are fetched continuously (no longer fixed volume of web content) and a 2 flow download is running in parallel, FQ_Codel performs worse than FIFO. This indicates that network-supported "flow-fair" scheduling might not be the optimal scheduling scheme when it comes to application performance. In D2 where the number of download flows are increased, we observe that the web browsing performance suffers in both FIFO and FQ_Codel since they are not able to provide application isolation. An file application can "cheat" by using more TCP flows to gain more bandwidth since these schemes support flow-fair behaviour. *AutoQoS*, on the other hand, uses per-app queues to isolate the traffic and measures the performance of the download independent of the number of TCP flows between the endpoints (as described by the *AppNet* profile) and hence provides isolated and fair performance across classes. Since, the browsing traffic is no longer "static", priority queueing no longer is the best scheme and in D3 turns out to be the worst scheme as it blindly prioritizes heavy browsing traffic over large downloads (which suffer a lot).

Overall, *AutoQoS* performs well tackling different kinds of congestion scenarios and outperforms widely-deployed scheduling schemes. By collecting the right metrics that indicate app-level performance and dynamically distributing the resources (using weights), *AutoQoS* is able to achieve isolation and fairness at the application level.

Comparison with Fixed Optimum

Having compared *AutoQoS* to existing schemes, we now ask the question: *How close is AutoQoS to an optimal weight distribution set for a scenario?* For this, we consider the dynamic scenarios (D1-3) where *AutoQoS*'s impact is observed. For each of these scenarios, we find a constant weight distribution that can give the best min-max fairness by performing a grid-search of all the weight combinations. We call this configuration

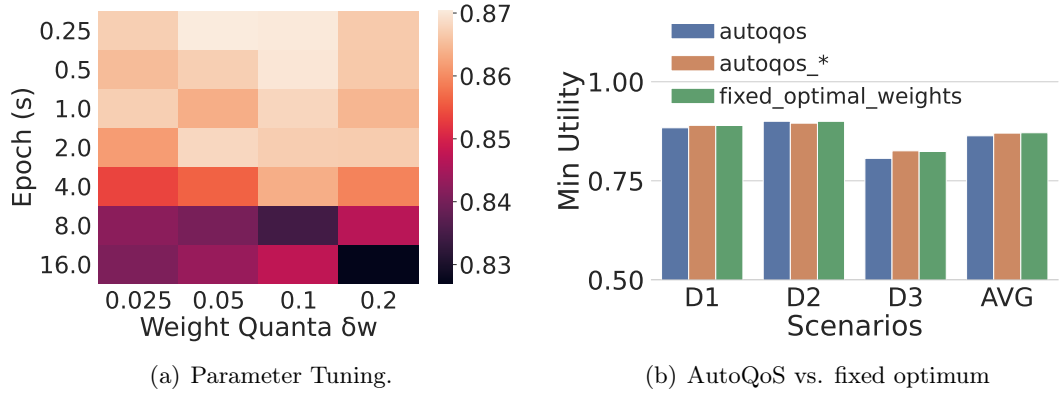


Figure 6.12: Comparison with fixed optimum before and after tuning.

"fixed-optimal-weight". As shown in Fig. 6.12(b), we see that the fixed-optimal-weight slightly outperforms *AutoQoS* in each of the scenarios. This is expected as the scenario was run multiple times to figure out the optimal weights while *AutoQoS* has no prior knowledge of the scenario and reacts dynamically. We note that this is not necessarily the absolute optimal scheduling policy since in such a dynamic scenario, with real applications, finding the optimal scheduling policy is itself a non-trivial task. We thus use the "fixed-optimal-weight" as an upper bound to compare.

Having found the upper-bound, we now tune *AutoQoS*'s parameters *epoch* and δw to improve its performance. We do a grid search (as shown in Fig. 6.12(a)) by varying the *epoch* from 250ms to 16s and *quanta* from 0.025 to 0.2. We found that an *epoch* of less than 2 seconds generally yields good result however, a faster reaction time of 250ms yields the best result since *AutoQoS* has more opportunities to redistribute the weights in real-time. The *quanta* parameter needs to be between 0.05 and 0.1 and being lower or higher than this leads to poor performance. A lower quanta made the reactive loop slow since it took multiple epochs to increment the weight and a higher quanta results in oscillations. We (unsurprisingly) found that *AutoQoS* tuned with *epoch* = 250ms and *quanta* = 0.05 (named *autoqos_**) worked the better than the default configuration and is very close to the "fixed-optimal-weight" while not requiring any prior knowledge. We note that *AutoQoS** slightly outperformed "fixed-optimal-weight" in scenario D3 since it could dynamically adapt the weights and reach a better performance than fixed weights.

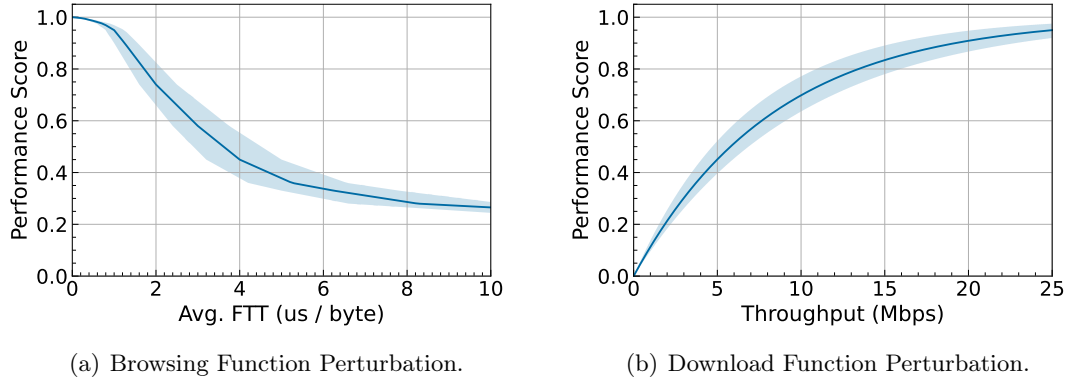


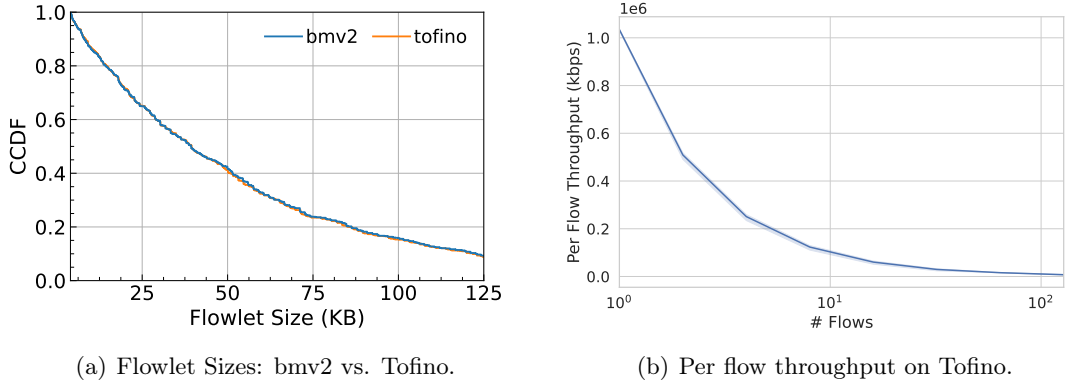
Figure 6.13: Sensitivity to Mapping Functions.

Mapping Function Sensitivity

In addition to evaluating the impact of *AutoQoS*'s parameters, we now study the impact of variation in the performance mapping functions. In particular, we answer the question: *Do the mapping functions have to be very precise? If not, how much variation can AutoQoS tolerate without affecting the overall performance?*

To study the impact of curve perturbation, we created two additional scenarios using web browsing and download traffic: scenario 1 in which we use one browser to fetch all Alexa top 20 webpages in a loop with 2 flows in the background and scenario 2 in which we double the browsers and quadruple the download flows to create a heavy congested scenario. Then, for each scenario, we first keep one curve constant, for example download curve, and vary the other curve, say browsing curve. And then repeat the experiments vice-versa. We study the impact of variation in the curve in both scenarios and take an average score.

We observed in both cases that the curve even when varied up to 10% in area (as shown in Fig. 6.13) creates only 1% deviation in the final min utilities of the applications. We observe that *AutoQoS* is tolerant to imprecise curves. For instance, even though the actual mean web page size (an input to define the browsing curve) was 2.05MB for the experiment, perturbing the curve by using values from 1.6 to 2.4 MB (+20%) had a very small impact on the final performance. This is because of two major reasons: *AutoQoS* employs weighted queuing which is work-conserving *i.e.*, any resource not used by a class

Figure 6.14: *AutoQoS* Hardware Benchmarks.

is automatically given to another class even at smaller timescales and secondly, *AutoQoS* is approximate in measuring the performance as the network metrics act as a good proxy but are not precise themselves. Thus, due to the approximations *AutoQoS* makes it can tolerate some amount of perturbation. Upon further perturbing the curve (more than 10%), *AutoQoS*'s operation results in gradual decrease of performance of the class being effected *i.e.*, *AutoQoS* deteriorates slowly and continuously.

Hardware Micro-benchmark

We now evaluate the hardware implementation of our framework on Tofino, in particular focusing on accuracy and scale of measurements. Since Tofino has several resource constraints one needs to make some approximations. For example, while we could access, store (in registers) and perform arithmetic on usec timestamps in bmv2, we could not do the same on Tofino (especially for flowlet computation). However, we note that even with approximations (storing only 32bit timestamps), our implementation produces a very similar flowlet size distribution for the web browsing traffic as shown in 6.14(a). While Tofino's scale has been evaluated in prior work[68, 199, 200], we also show in our work (refer to 6.14(b)) that we were able to measure per-flow throughput accurately to 100s of concurrent large TCP connections using Tofino. We see that with on a 1Gbps bottleneck link, the measurements accurately reflect the per-flow throughput as we keep doubling the flows from 1 to 128 (maximum supported by iperf tool). One may use additional data structures like bloom filters etc. to implement loglog counting method to estimate the

average throughput of 1000s of concurrent flows. It is left out of the scope of this work.

6.3.6 Discussion

We discuss some key aspects and limitations of *AutoQoS* and set some directions for future work.

AppNet Profile Dependency *AutoQoS* depends on the input profiles containing the telemetry logic and the mapping function to measure and score the performance of an application class. We argue that application developers are the ideal source of such profiles. However, academia can also contribute in coming up with the profiles by testing applications under different conditions[101] and/or giving scores based on subjective experiments (as done in [182]). *AutoQoS* currently cannot auto generate such profiles but it is an interesting direction for future work.

Adding new classes While we described three classes of traffic, *AutoQoS* is not limited by it. One can add a new class to the system by first, separating the class from rest of the traffic (using traffic classification tools), second, designing the right metric to extract and writing data plane code to export network telemetry for it (can also re-use functions if suitable) and third, specifying a mapping function to map to application performance. We note that the metrics exported from data plane need not map directly to performance but some higher level inferencing can be done on top of them. For instance, as shown in [17, 18, 82] one can use ML models on specific metadata extracted from network to infer properties like resolution and re-buffering ratio which can be used to estimate a performance score. In this chapter, we wanted to highlight the design of the system via a few exemplary classes and hence didn't study more applications. We encourage researchers to study different applications and their requirements and test their proposals using our testbed.

Programmable Hardware Constraints. Programmable switches while are resource constrained and cannot execute complex functions. We acknowledge that the number of classes supported by the system can be limited by number of telemetry functions that can run together on the hardware. While we use programmable data planes in our work and borrow its limitations, the framework is not constrained by it: one can also use more

flexible frameworks like DPDK [14] to extract more sophisticated telemetry.

Operational Context We designed *AutoQoS* to support deployment in ISP edge networks where bottleneck is predominant [176]. However, it can be deployed at any other bottleneck, which consists of applications with different requirements. The current limitation is that home routers are not programmable nor very powerful to extract telemetry, but we hope future advancements make it possible.

6.4 Conclusion

In conclusion, increasing internet usage, fueled by the pandemic, is causing online applications to suffer due to persistent congestion prevalent in last mile bottlenecks. Network operators find it difficult to manage bottlenecks as AQM policies only improve delay and QoS policies are hard to tune with dynamic traffic scenarios. In this chapter, we first presented our self-driving network prototype called *AppAssist* which can dynamically assist suffering applications using priority queues. We showed that a control loop is indeed possible if appropriate telemetry functions can be executed that can indicate a drop in performance. We then built upon the ideas in that framework by developing a system called *AutoQoS* which helps network operators offer better application performance by automatically re-configuring the weights to each application class. It does so by leveraging programmable networks to extract in-network metrics that are more indicative of application performance, maps them to normalized performance scores and then redistributes the resources to ensure a fair and performant network on the application level. Our framework is extendable to support a variety of applications and can adapt to different traffic distributions automatically. Our evaluations showed that *AutoQoS* outperforms widely deployed static QoS policies and AQM methods in various traffic scenarios, performs very close to an optimally tuned QoS configuration and can be implementable at scale in programmable hardware.

Chapter 7

Conclusions and Future Work

Contents

7.1	Conclusions	136
7.2	Future Work	138

7.1 Conclusions

Internet Service Providers (ISPs) have been struggling to offer good QoE for diverse applications ranging from video streaming and gaming to social media and teleconferencing. To ensure a good user experience, ISPs have to understand and manage the performance of the applications traversing their network. ISPs need to measure more than just coarse-metrics link utilization and packet loss. Existing application classification and simplistic QoE monitoring technologies using traditional DPIs are not only starting to fail with traffic encryption but also are prohibitively expensive with exploding traffic rates of the Internet.

In this thesis, we leveraged emerging programmable networks to extract fine-grained and specific telemetry and developed machine learning and statistical models that can classify applications, monitor their performance, and tune the network to enhance user experience. We built complete systems that can extract application-level intelligence from encrypted network traffic and provide enhanced visibility to ISPs at scale, with high accuracy, and at low costs that the existing solutions are not able to achieve.

The key contributions presented in this thesis toward application-aware monitoring and management of operator networks are briefly summarized below.

- As a first step, we developed a novel data format and telemetry algorithm to capture the behaviour of encrypted network flows. We then developed transformer-based deep learning models which use self-attention to efficiently learn intricate patterns in the traffic shape and classify application types and providers with f1 scores of 97% and 95% respectively.
- We then focused on measuring the user experience of highly engaging applications starting with video streaming. We developed a tool to collect client-side metrics and network activity of over 500 hours of both on-demand and live video streaming applications such as Netflix, Twitch, and YouTube. We analysed the data to highlight the key characteristics of video streaming and designed telemetry functions to extract flow-level and chunk-level data. Lastly, we developed machine learning-based and statistical models to predict video streaming QoE metrics such as bitrate, buffering states, resolution, and buffer stalls with 90+% overall accuracy.
- We shifted our focus to study latency-sensitive online multiplayer gaming applications which are highly engaging and of high economic value. We collected and analysed packet traces of ten popular online games to develop a classifier that quickly detects a game from automatically generated signatures. We deployed this system in our university traffic and found over 31,000 gaming sessions representing 9,000 gaming hours over a month. We then performed latency measurements and BGP/Geo-IP lookups to the 4,500+ gaming servers (spanning 14 countries and 165 routing prefixes) and showed that routing and peering decisions can significantly impact gaming latencies.
- Finally we developed systems to improve the QoE of applications contending at congested links. We developed a self-driving network prototype that continuously measures application states and automatically intervenes to assist “suffering” applications that are being impacted by transient congestion. Subsequently, we designed and built a complete system that extracts application-aware network telemetry from programmable switches and dynamically adapts the QoS policies to manage the bot-

tleneck resources in an application-fair manner. We showed that the system outperforms known queue management techniques in various traffic scenarios and closely approximates optimally tuned configurations for each scenario.

We believe that our contributions presented in this thesis provide a new framework for ISPs to effectively monitor and manage their network with enhanced application awareness. The contributions taken together can classify applications, measure their QoE metrics, and improve the QoE during congestion in an application-fair manner.

7.2 Future Work

We note that our methods, designs, and prototypes can be further improved and extended to address their limitations or widen their applicability and scope. Some directions for future work are highlighted below.

- In Chapter 3, we classified encrypted network traffic using the FlowPrint data structure and transformer-based deep learning models. Currently, it takes multiple registers in a memory-limited programmable switch to collect FlowPrint for each flow. Optimizations to compress FlowPrint data structure while tracking the traffic shape (even if approximate) can make it more scalable. Further, we currently used a fixed time window to classify flows – a dynamic classification system that takes the minimum amount of time to detect a traffic class (while taking more time until the confidence improves) can be developed.
- Chapter 4 executed network telemetry functions to extract fine-grained flow-level telemetry and sophisticated chunk-level telemetry to estimate various video streaming QoE metrics. While we implemented the telemetry using DPDK, we believe it can be implemented at a larger scale in programmable switches by combining approximate data structures like sketches and inactivity timeouts (to export the chunk). Further, our existing methods export the QoE metrics every window of 30 seconds leading to multiple metrics exported across thousands of concurrent video streams running for hours. Big data analysis techniques can be used to process these metrics and help ISPs zero in on the streams, users or sub-networks which have consistent poor QoE.

- The game classification system described in Chapter 5 rapidly detects online games using packet-byte and length-based signatures. While this chapter studied ten popular games, an evaluation of the proposed method on a wider set of games can be done. If conflicts arise amongst games, the classifier may require richer signatures extracted from more packets and/or deeper payload contents of individual packets. Another avenue for future work is the analysis of public peering datasets to offer low-latency peering recommendations within cost budgets to ISPs to improve their gaming latencies.
- The self-driving network prototype to assist suffering applications presented in Chapter 6 was quite limited in scope both in terms of applications and the action primitives used. While we improved it further and proposed *AutoQoS* (§6.3), it still has a few limitations which can be tackled with emerging technologies. For instance, *AutoQoS* relies on programmable switches to export per-class application-specific telemetry at scale. However, the compute and memory constraints limit the flexibility of telemetry and hence we cannot measure arbitrarily complex metrics. Further, the system cannot be currently implemented in small-scale WiFi router chipsets due to the lack of P4 programmability support. We believe newer technologies such as Tofino 2 [201] and p4-based routers can help alleviate some of these issues.
- This thesis limits its scope to the operational context of ISP networks and focuses on developing systems that can infer and improve QoE of sensitive applications. A subsequent stream of work that can build on top of it could be to correlate the QoE metrics and other network QoS datasets to diagnose network issues and pinpoint the cause of poor experience and subsequently recommend corrective actions that can be taken to improve the user experience. In addition to that, applications with increasing usage such as teleconferencing and cloud gaming can be studied in a similar fashion to infer and improve their QoE.

In addition to the future directions listed above, we hope researchers can build upon these contributions to explore further aspects of application-aware network monitoring to help network operators effectively manage and tune their networks for modern Internet applications.

References

- [1] Sandvine, *Global internet phenomenon report*, <https://www.sandvine.com/phenomena>, Accessed: 2022-02-28, 2022.
- [2] Cisco, *Cisco Annual Internet Report*, <https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/index.html>, Accessed: 2022-02-28, 2022.
- [3] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poesse, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohlfeld, and G. Smaragdakis, “A year in lockdown: How the waves of covid-19 impact internet traffic”, *Commun. ACM*, vol. 64, no. 7, pp. 101–108, Jun. 2021, ISSN: 0001-0782. DOI: 10.1145/3465212. [Online]. Available: <https://doi.org/10.1145/3465212>.
- [4] *Netflix | Open Connect*, [Online; accessed 4. Apr. 2022], Feb. 2022. [Online]. Available: <https://openconnect.netflix.com/en>.
- [5] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time”, *Queue*, vol. 14, no. 5, pp. 20–53, 2016.
- [6] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000. [Online]. Available: <https://www.rfc-editor.org/info/rfc9000>.
- [7] R. Fontugne, A. Shah, and K. Cho, “Persistent last-mile congestion: Not so uncommon”, in *Proceedings of the ACM Internet Measurement Conference*, 2020, pp. 420–427.
- [8] *RFC 1157 - Simple Network Management Protocol (SNMP)*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc1157>.
- [9] *RFC 3954 - Cisco Systems NetFlow Services Export Version 9*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc3954>.
- [10] *sFlow.org - Making the Network Visible*, [Online; accessed 28. Mar. 2022], Mar. 2022. [Online]. Available: <https://sflow.org>.
- [11] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, “Participatory networking: An api for application control of sdns”, *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 327–338, 2013.

-
- [12] J. Jiang, X. Liu, V. Sekar, I. Stoica, and H. Zhang, “Eona: Experience-oriented network architecture”, in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014, pp. 1–7.
 - [13] H. H. Gharakheili, V. Sivaraman, A. Vishwanath, L. Exton, J. Matthews, and C. Russell, “Broadband fast-lanes with two-sided control: Design, evaluation, and economics”, in *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, IEEE, 2015, pp. 195–200.
 - [14] Intel, *Data plane development kit (dpdk)*, 2021. [Online]. Available: <https://www.dpdk.org/>.
 - [15] Intel, *Intel tofino programmable chipset series*, 2021. [Online]. Available: <https://www.intel.com.au/content/www/au/en/products/network-io/programmable-ethernet-switch/tofino-series.html>.
 - [16] R. Babaria, S. Madanapalli, H. Kumar, and V. Sivaraman, “FlowFormers: Transformer-based Models for Real-time Network Flow Classification”, *IEEE MSN’21: The 17th International Conference on Mobility, Sensing and Networking*, Dec. 2021.
 - [17] S. C. Madanapalli, H. H. Gharakheili, and V. Sivaraman, “Inferring netflix user experience from broadband network measurement”, in *2019 Network Traffic Measurement and Analysis Conference (TMA)*, IEEE, 2019, pp. 41–48.
 - [18] S. C. Madanapalli, A. Mathai, H. H. Gharakheili, and V. Sivaraman, “Reclive: Real-time classification and qoe inference of live video streaming services”, in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQoS)*, IEEE, 2021, pp. 1–7.
 - [19] S. C. Madanapalli, H. H. Gharakheili, and V. Sivaraman, “Know thy lag: In-network game detection and latency measurement”, in *International Conference on Passive and Active Network Measurement*, Springer, 2022, pp. 395–410.
 - [20] S. C. Madanapalli, H. H. Gharakheili, and V. Sivaraman, “Assisting delay and bandwidth sensitive applications in a self-driving network”, in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, 2019, pp. 64–69.
 - [21] V. Sivaraman, S. C. Madanapalli, H. Kumar, and H. H. Gharakheili, “Opentd: Open traffic differentiation in a post-neutral world”, in *Proceedings of the 2019 ACM Symposium on SDN Research*, 2019, pp. 119–126.
 - [22] *Workshop on Self-Driving Networks*, [Online; accessed 8. Apr. 2022], Apr. 2018. [Online]. Available: <https://nsf-srn-2018.cs.princeton.edu/nsf-srn-report.pdf>.
 - [23] *ping(8) - Linux man page*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://linux.die.net/man/8/ping>.
 - [24] *traceroute(8) - Linux man page*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://linux.die.net/man/8/traceroute>.
 - [25] *TCPDUMP & LIBPCAP*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://www.tcpdump.org>.

- [26] *RFC 3413 - Simple Network Management Protocol (SNMP) Applications*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc3413>.
- [27] A. Shaikh and J. George, *Sdn in the management plane: Openconfig and streaming telemetry*, 2015. [Online]. Available: https://archive.nanog.org/sites/default/files//meetings/NANOG64/1011/20150604_George_Sdn_In_The_v1.pdf.
- [28] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, “OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks”, in *2014 IEEE Network Operations and Management Symposium (NOMS)*, IEEE, May 2014, pp. 1–8, ISBN: 978-1-4799-0913-1. DOI: 10.1109/NOMS.2014.6838228.
- [29] P.-W. Tsai, C.-W. Tsai, C.-W. Hsu, and C.-S. Yang, “Network Monitoring in Software-Defined Networking: A Review”, *IEEE Systems Journal*, vol. 12, no. 4, pp. 3958–3969, Feb. 2018, ISSN: 1937-9234. DOI: 10.1109/JSYST.2018.2798060.
- [30] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, “P4: Programming protocol-independent packet processors”, *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
- [31] Netflix, *Netflix: Watch tv shows and movies online*, <https://www.netflix.com>, Accessed: 2022-03-20, 2022.
- [32] YouTube, *Youtube*, <https://www.youtube.com>, Accessed: 2022-03-20, 2022.
- [33] Twitch, *Twitch*, <https://www.twitch.tv>, Accessed: 2022-03-20, 2022.
- [34] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein, “Learning in situ: A randomized experiment in video streaming”, in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 495–511.
- [35] *RFC 5681 - TCP Congestion Control*, [Online; accessed 28. Mar. 2022], Mar. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc5681>.
- [36] CAIDA, [Online; accessed 28. Mar. 2022], Mar. 2022. [Online]. Available: <https://www.caida.org>.
- [37] N. Feamster and J. Livingood, “Measuring internet speed: Current challenges and future recommendations”, *Communications of the ACM*, vol. 63, no. 12, pp. 72–80, 2020.
- [38] *RFC 7799 - Active and Passive Metrics and Methods (with Hybrid Types In-Between)*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc7799>.
- [39] H. Zhang, Z. Cai, Q. Liu, Q. Xiao, Y. Li, and C. F. Cheang, “A Survey on Security-Aware Measurement in SDN”, *Security and Communication Networks*, vol. 2018, p. 2459154, Apr. 2018, ISSN: 1939-0114. DOI: 10.1155/2018/2459154.

-
- [40] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret, “Network traffic classifier with convolutional and recurrent neural networks for internet of things”, *IEEE Access*, vol. 5, pp. 18 042–18 050, 2017. DOI: 10.1109/ACCESS.2017.2747560.
 - [41] G. V. Lioudakis, F. Gaudino, E. Boschi, G. Bianchi, D. I. Kaklamani, and I. S. Venieris, “Legislation-aware privacy protection in passive network monitoring”, in *Information Communication Technology Law, Protection and Access Rights: Global Approaches and Issues*, IGI Global, 2010, pp. 363–383.
 - [42] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, “Designing heavy-hitter detection algorithms for programmable switches”, *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1172–1185, 2020.
 - [43] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, “In-band network telemetry: A survey”, *Computer Networks*, vol. 186, p. 107 763, 2021.
 - [44] B. Krishnamurthy and C. E. Wills, “Analyzing factors that influence end-to-end web performance”, *Computer Networks*, vol. 33, no. 1-6, pp. 17–32, 2000.
 - [45] C. So-In, “A survey of network traffic monitoring and analysis tools”, *Cse 576m computer system analysis project, Washington University in St. Louis*, 2009.
 - [46] K. Ervasti, *A survey on network measurement: Concepts, techniques, and tools*, 2016.
 - [47] V. Gueant, *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*, [Online; accessed 28. Mar. 2022], Mar. 2022. [Online]. Available: <https://iperf.fr>.
 - [48] *Wireshark · Go Deep*. [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://www.wireshark.org>.
 - [49] *RFC 7011 - Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*, [Online; accessed 23. Mar. 2022], Mar. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc7011>.
 - [50] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: Enabling innovation in campus networks”, *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.
 - [51] P. Megyesi, A. Botta, G. Aceto, A. Pescapé, and S. Molnár, “Challenges and solution for measuring available bandwidth in software defined networks”, *Computer Communications*, vol. 99, pp. 48–61, Feb. 2017, ISSN: 0140-3664. DOI: 10.1016/j.comcom.2016.12.004.
 - [52] X. Zhang, Y. Wang, J. Zhang, L. Wang, and Y. Zhao, “A two-way link loss measurement approach for software-defined networks”, in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, IEEE, Jun. 2017, pp. 1–10. DOI: 10.1109/IWQoS.2017.7969164.
 - [53] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha, “Software-defined latency monitoring in data center networks”, in *International Conference on Passive and Active Network Measurement*, Springer, 2015, pp. 360–372.

-
- [54] W. Queiroz, M. A. Capretz, and M. Dantas, “An approach for sdn traffic monitoring based on big data techniques”, *Journal of Network and Computer Applications*, vol. 131, pp. 28–39, 2019.
 - [55] A. AlGhadhban and B. Shihada, “FLight: A Fast and Lightweight Elephant-Flow Detection Mechanism”, in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, Jul. 2018, pp. 1537–1538. DOI: 10.1109/ICDCS.2018.00161.
 - [56] S. Wang, J. Zhang, T. Huang, J. Liu, Y.-j. Liu, and F. R. Yu, “Flowtrace: Measuring round-trip time and tracing path in software-defined networking with low communication overhead”, *Frontiers of Information Technology & Electronic Engineering*, vol. 18, no. 2, pp. 206–219, 2017.
 - [57] S. C. Madanapalli, M. Lyu, H. Kumar, H. H. Gharakheili, and V. Sivaraman, “Real-time detection, isolation and monitoring of elephant flows using commodity sdn system”, in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2018, pp. 1–5.
 - [58] Y. Zhao, P. Zhang, and Y. Jin, “Netography: Troubleshoot your network with packet behavior in SDN”, in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, IEEE, Apr. 2016, pp. 878–882, ISBN: 978-1-5090-0223-8. DOI: 10.1109/NOMS.2016.7502919.
 - [59] *OpenConfig - Streaming Telemetry*, [Online; accessed 23. Mar. 2022], Aug. 2019. [Online]. Available: <https://www.openconfig.net/projects/telemetry>.
 - [60] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, “Flowwatcher-dpdk: Lightweight line-rate flow-level monitoring in software”, *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 1143–1156, 2019.
 - [61] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-Hitter Detection Entirely in the Data Plane”, in *SOSR '17: Proceedings of the Symposium on SDN Research*, New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 164–176, ISBN: 978-1-45034947-5. DOI: 10.1145/3050220.3063772.
 - [62] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, “Network-wide heavy hitter detection with commodity switches”, in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.
 - [63] M. V. B. da Silva, A. S. Jacobs, R. J. Pfitscher, and L. Z. Granville, “IDEAFIX: Identifying Elephant Flows in P4-Based IXP Networks”, in *2018 IEEE Global Communications Conference (GLOBECOM)*, IEEE, Dec. 2018, pp. 1–6. DOI: 10.1109/GLOCOM.2018.8647685.
 - [64] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, “Turboflow: information rich flow record generation on commodity switches”, in *EuroSys '18: Proceedings of the Thirteenth EuroSys Conference*, New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–16, ISBN: 978-1-45035584-1. DOI: 10.1145/3190508.3190558.
 - [65] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho, “FlowStalker: Comprehensive Traffic Flow Monitoring on the Data Plane using P4”, in *ICC 2019 - 2019 IEEE*

- International Conference on Communications (ICC)*, IEEE, May 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761197.
- [66] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic sketch: Adaptive and fast network-wide measurements”, in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.
 - [67] Z. Hang, M. Wen, Y. Shi, and C. Zhang, “Interleaved Sketch: Toward Consistent Network Telemetry for Commodity Programmable Switches”, *IEEE Access*, vol. 7, pp. 146 745–146 758, Oct. 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2946704.
 - [68] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, “Fcm-sketch: Generic network measurements with data plane support”, in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 78–92.
 - [69] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, “Sonata: Query-driven streaming network telemetry”, in *Proceedings of the 2018 conference of the ACM special interest group on data communication*, 2018, pp. 357–371.
 - [70] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich, “Catching the microburst culprits with snappy”, in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, 2018, pp. 22–28.
 - [71] F. Cugini, P. Gunning, F. Paolucci, P. Castoldi, and A. Lord, “P4 In-Band Telemetry (INT) for Latency-Aware VNF in Metro Networks”, in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*, IEEE, Mar. 2019, pp. 1–3. [Online]. Available: <https://ieeexplore.ieee.org/document/8696518>.
 - [72] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, “Scaling hardware accelerated network monitoring to concurrent and dynamic queries with starflow”, in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 823–835.
 - [73] H. Tahaei, F. Afifi, A. Asemi, F. Zaki, and N. B. Anuar, “The rise of traffic classification in IoT networks: A survey”, *Journal of Network and Computer Applications*, vol. 154, p. 102 538, Mar. 2020, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2020.102538.
 - [74] P. Velan, M. Čermák, P. Čeleda, and M. Drašar, “A survey of methods for encrypted traffic classification and analysis”, *International Journal of Network Management*, vol. 25, no. 5, pp. 355–374, Sep. 2015, ISSN: 1055-7148. DOI: 10.1002/nem.1901.
 - [75] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, “Automated website fingerprinting through deep learning”, *arXiv preprint arXiv:1708.06376*, 2017.
 - [76] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities”, *Journal of Internet*

-
- Services and Applications*, vol. 9, no. 1, pp. 1–99, Dec. 2018, ISSN: 1869-0238. DOI: 10.1186/s13174-018-0087-2.
- [77] A. Ahmed, Z. Shafiq, H. Bedi, and A. Khakpour, “Suffering from buffering? detecting qoe impairments in live video streams”, in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, IEEE, 2017, pp. 1–10.
 - [78] Y. Xue, D. Wang, and L. Zhang, “Traffic classification: Issues and challenges”, in *2013 International Conference on Computing, Networking and Communications (ICNC)*, IEEE, Jan. 2013, pp. 545–549. DOI: 10.1109/ICCNC.2013.6504144.
 - [79] M. Finsterbusch, C. Richter, E. Rocha, J.-A. Muller, and K. Hanssgen, “A Survey of Payload-Based Traffic Classification Approaches”, *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 1135–1156, Oct. 2013, ISSN: 1553-877X. DOI: 10.1109/SURV.2013.100613.00161.
 - [80] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, “Towards the deployment of machine learning solutions in network traffic classification: A systematic survey”, *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2018.
 - [81] *The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019)*, [Online; accessed 8. Apr. 2022], Dec. 2019. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset.
 - [82] C. Gutterman, K. Guo, S. Arora, X. Wang, L. Wu, E. Katz-Bassett, and G. Zussman, “Requet: Real-time qoe detection for encrypted youtube traffic”, in *Proceedings of the 10th ACM Multimedia Systems Conference*, 2019, pp. 48–59.
 - [83] M. H. Mazhar *et al.*, “Real-time video quality of experience monitoring for https and quic”, in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, IEEE, 2018, pp. 1331–1339.
 - [84] H. Habibi Gharakheili, M. Lyu, Y. Wang, H. Kumar, and V. Sivaraman, “iTeleScope: Softwarized Network Middle-box for Real-Time Video Telemetry and Classification”, *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 1071–1085, 2019.
 - [85] Y. Dhote, S. Agrawal, and A. J. Deen, “A Survey on Feature Selection Techniques for Internet Traffic Classification”, in *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, IEEE, Dec. 2015, pp. 1375–1380. DOI: 10.1109/CICN.2015.267.
 - [86] M. Lotfollahi, M. Jafari Siavoshani, R. Shirali Hossein Zade, and M. Saberian, “Deep packet: A novel approach for encrypted traffic classification using deep learning”, *Soft Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.
 - [87] P. Wang, F. Ye, X. Chen, and Y. Qian, “Datanet: Deep learning based encrypted network traffic classification in sdn home gateway”, *IEEE Access*, vol. 6, pp. 55 380–55 391, 2018.
 - [88] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang, “End-to-end encrypted traffic classification with one-dimensional convolution neural networks”, in *2017 IEEE in-*

- ternational conference on intelligence and security informatics (ISI)*, IEEE, 2017, pp. 43–48.
- [89] S. Rezaei, B. Kroencke, and X. Liu, “Large-scale mobile app identification using deep learning”, *IEEE Access*, vol. 8, pp. 348–362, 2019.
 - [90] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, “Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges”, *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 445–458, 2019.
 - [91] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, “Characterization of encrypted and vpn traffic using time-related”, in *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*, 2016, pp. 407–414.
 - [92] Z. Chen, G. Cheng, B. Jiang, S. Tang, S. Guo, and Y. Zhou, “Length matters: Fast internet encrypted traffic service classification based on multi-pdu lengths”, in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, IEEE, 2020, pp. 531–538.
 - [93] I. Akbari, M. A. Salahuddin, L. Ven, N. Limam, R. Boutaba, B. Mathieu, S. Moteau, and S. Tuffin, “A look behind the curtain: Traffic classification in an increasingly encrypted web”, *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 1, Feb. 2021. DOI: 10.1145/3447382. [Online]. Available: <https://doi.org/10.1145/3447382>.
 - [94] M. Alreshoodi and J. Woods, “Survey on qoe-qos correlation models for multimedia services”, *arXiv preprint arXiv:1306.0221*, 2013.
 - [95] A. Huet, A. Saverimoutou, Z. B. Houidi, H. Shi, S. Cai, J. Xu, B. Mathieu, and D. Rossi, “Revealing QoE of Web Users from Encrypted Network Traffic”, in *2020 IFIP Networking Conference (Networking)*, IEEE, Jun. 2020, pp. 28–36. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9142810>.
 - [96] N. Barman and M. G. Martini, “QoE Modeling for HTTP Adaptive Video Streaming: A Survey and Open Challenges”, *IEEE Access*, vol. 7, pp. 30 831–30 859, Mar. 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2901778.
 - [97] T. Mangla, E. Halepovic, M. Ammar, and E. Zegura, “eMIMIC: estimating http-based video QoE metrics from encrypted network traffic”, in *Proc. IEEE/IFIP TMA*, Vienna, Austria 2018.
 - [98] F. Bronzino, P. Schmitt, S. Ayoubi, G. Martins, R. Teixeira, and N. Feamster, “Inferring streaming video quality from encrypted traffic: Practical models and deployment experience”, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–25, 2019.
 - [99] T. Hoßfeld and A. Binzenhöfer, “Analysis of Skype VoIP traffic in UMTS: End-to-end QoS and QoE measurements”, *Computer Networks*, vol. 52, no. 3, pp. 650–666, Feb. 2008, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2007.10.008.
 - [100] G. Carofiglio, G. Grassi, E. Loparco, L. Muscariello, M. Papalini, and J. Samain, “Characterizing the relationship between application qoe and network qos for real-

- time services”, in *Proceedings of the ACM SIGCOMM 2021 Workshop on Network-Application Integration*, 2021, pp. 20–25.
- [101] K. MacMillan, T. Mangla, J. Saxon, and N. Feamster, “Measuring the performance and network utilization of popular video conferencing applications”, *arXiv preprint arXiv:2105.13478*, 2021.
 - [102] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, “QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis”, in *IMC '14: Proceedings of the 2014 Conference on Internet Measurement Conference*, New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 151–164, ISBN: 978-1-45033213-2. DOI: 10.1145/2663716.2663726.
 - [103] V. Krishnamoorthi, N. Carlsson, E. Halepovic, and E. Petajan, “Buffest: Predicting buffer conditions and real-time requirements of http (s) adaptive streaming clients”, in *Proc. ACM MMSys*, Taipei, Taiwan, Jun. 2017.
 - [104] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan, “Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements”, in *Proc. ACM HotMobile*, Santa Barbara, CA, USA, Feb. 2014.
 - [105] G. Dimopoulos, I. Leontiadis, P. Barlet-Ros, and K. Papagiannaki, “Measuring video qoe from encrypted traffic”, in *Proc. ACM IMC*, Santa Monica, CA, USA, Mar. 2016.
 - [106] I. Orsolic, D. Pevec, M. Suznjevic, and L. Skorin-Kapov, “A machine learning approach to classifying youtube qoe based on encrypted network traffic”, *Springer, Multimedia tools and applications*, vol. 76, no. 21, pp. 22 267–22 301, 2017.
 - [107] D. Tsilimantos, T. Karagkioules, and S. Valentin, “Classifying flows and buffer state for youtube’s http adaptive streaming service in mobile networks”, in *Proceedings of the 9th ACM Multimedia Systems Conference*, 2018, pp. 138–149.
 - [108] M. Shen, J. Zhang, K. Xu, L. Zhu, J. Liu, and X. Du, “DeepQoE: Real-time Measurement of Video QoE from Encrypted Traffic with Deep Learning”, in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, IEEE, Jun. 2020, pp. 1–10. DOI: 10.1109/IWQoS49365.2020.9212897.
 - [109] T. Guarnieri, I. Drago, A. B. Vieira, I. Cunha, and J. Almeida, “Characterizing qoe in large-scale live streaming”, in *GLOBECOM 2017-2017 IEEE Global Communications Conference*, IEEE, 2017, pp. 1–7.
 - [110] R. Ware, M. K. Mukerjee, S. Seshan, and J. Sherry, “Modeling bbr’s interactions with loss-based congestion control”, in *Proceedings of the internet measurement conference*, 2019, pp. 137–143.
 - [111] B. Turkovic and F. Kuipers, “P4air: Increasing fairness among competing congestion control algorithms”, in *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, IEEE, 2020, pp. 1–12.
 - [112] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, “Homa: A receiver-driven low-latency transport protocol using network priorities”, in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 221–235.

- [113] E. C. Economy, “The great firewall of China: Xi Jinping’s internet shutdown”, *the Guardian*, Jul. 2021. [Online]. Available: <https://www.theguardian.com/news/2018/jun/29/the-great-firewall-of-china-xi-jinpings-internet-shutdown>.
- [114] A. M. Kakhki, F. Li, D. Choffnes, E. Katz-Bassett, and A. Mislove, “Bingeon under the microscope: Understanding t-mobiles zero-rating implementation”, in *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*, 2016, pp. 43–48.
- [115] S. Optus, *Optus Living Network: Optus*, [Online; accessed 1. Apr. 2022], Apr. 2022. [Online]. Available: <https://www.optus.com.au/living-network>.
- [116] *RFC 8290 - The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*, [Online; accessed 2. Apr. 2022], Apr. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc8290>.
- [117] *RFC 7567 - IETF Recommendations Regarding Active Queue Management*, [Online; accessed 2. Apr. 2022], Apr. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc7567>.
- [118] R. Pan, P. Natarajan, F. Baker, and G. White, *Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem*, RFC 8033, Feb. 2017. DOI: 10.17487/RFC8033. [Online]. Available: <https://www.rfc-editor.org/info/rfc8033>.
- [119] N. Khademi, D. Ros, and M. Welzl, “The new aqm kids on the block: An experimental evaluation of codel and pie”, in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2014, pp. 85–90.
- [120] T. Høiland-Jørgensen, D. Täht, and J. Morton, “Piece of cake: A comprehensive queue management solution for home gateways”, in *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, IEEE, 2018, pp. 37–42.
- [121] *Bufferbloat.net*, [Online; accessed 2. Apr. 2022], Jan. 2022. [Online]. Available: <https://www.bufferbloat.net/projects>.
- [122] R. Jain, “A survey of scheduling methods”, *Nokia Research Center*, 1997.
- [123] D. Stiliadis and A. Varma, “Efficient fair queueing algorithms for packet-switched networks”, *IEEE/ACM transactions on Networking*, vol. 6, no. 2, pp. 175–185, 1998.
- [124] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, “Programmable packet scheduling at line rate”, in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 44–57.
- [125] V. Shrivastav, “Fast, scalable, and programmable packet scheduler in hardware”, in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 367–379.
- [126] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, “Programmable packet scheduling with a single queue”, in *SIGCOMM ’21: Proceedings*

- of the 2021 ACM SIGCOMM 2021 Conference, New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 179–193, ISBN: 978-1-45038383-7. DOI: 10.1145/3452296.3472887.
- [127] *tc-bfifo(8) - Linux manual page*, [Online; accessed 2. Apr. 2022], Mar. 2022. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc-bfifo.8.html>.
 - [128] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, “Num-fabric: Fast and flexible bandwidth allocation in datacenters”, in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 188–201.
 - [129] B. Dekeris, T. Adomkus, and A. Budnikas, “Analysis of qos assurance using weighted fair queueing (wqf) scheduling discipline with low latency queue (llq)”, in *28th International Conference on Information Technology Interfaces, 2006.*, IEEE, 2006, pp. 507–512.
 - [130] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, “Approximating fair queueing on reconfigurable switches”, in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 1–16.
 - [131] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, “Programmable calendar queues for high-speed packet scheduling”, in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 685–699.
 - [132] R. Adams, “Active Queue Management: A Survey”, *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1425–1476, Oct. 2012, ISSN: 1553-877X. DOI: 10.1109/SURV.2012.082212.00018.
 - [133] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, “Controlled delay active queue management”, *RFC 8289*, vol. 1, pp. 1–25, 2018.
 - [134] *tc-fq_codel(8) - Linux manual page*, [Online; accessed 2. Apr. 2022], Mar. 2022. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-fq_codel.8.html.
 - [135] *tc-fq_pie(8) - Linux manual page*, [Online; accessed 2. Apr. 2022], Mar. 2022. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-fq_pie.8.html.
 - [136] A. Flickinger, C. Klatsky, A. Ledesma, J. Livingood, and S. Ozer, “Improving latency with active queue management (aqm) during covid-19”, *arXiv preprint arXiv:2107.13968*, 2021.
 - [137] *What Can I Do About Bufferbloat? - Bufferbloat.net*, [Online; accessed 2. Apr. 2022], Jan. 2022. [Online]. Available: https://www.bufferbloat.net/projects/bloat/wiki/What_can_I_do_about_Bufferbloat.
 - [138] D. L. Black, Z. Wang, M. A. Carlson, W. Weiss, E. B. Davies, and S. L. Blake, *An Architecture for Differentiated Services*, RFC 2475, Dec. 1998. DOI: 10.17487/RFC2475. [Online]. Available: <https://www.rfc-editor.org/info/rfc2475>.
 - [139] R. T. Braden, D. D. D. Clark, and S. Shenker, *Integrated Services in the Internet Architecture: an Overview*, RFC 1633, Jun. 1994. DOI: 10.17487/RFC1633. [Online]. Available: <https://www.rfc-editor.org/info/rfc1633>.

-
- [140] Y. Li, H. Xie, J. C. Lui, and K. L. Calvert, “Quantifying deployability and evolvability of future internet architectures via economic models”, *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 1995–2008, 2020.
 - [141] M. Karakus and A. Durresi, “Quality of Service (QoS) in Software Defined Networking (SDN): A survey”, *Journal of Network and Computer Applications*, vol. 80, pp. 200–218, Feb. 2017, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.12.019.
 - [142] J. Yan, H. Zhang, Q. Shuai, B. Liu, and X. Guo, “HiQoS: An SDN-based multipath QoS solution”, *China Communications*, vol. 12, no. 5, pp. 123–133, Jun. 2015, ISSN: 1673-5447. DOI: 10.1109/CC.2015.7112035.
 - [143] R. F. Diorio and V. S. Timóteo, “Per-Flow Routing with QoS Support to Enhance Multimedia Delivery in OpenFlow SDN”, in *Webmedia '16: Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*, New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 167–174, ISBN: 978-1-45034512-5. DOI: 10.1145/2976796.2976844.
 - [144] S. Tomovic, N. Prasad, and I. Radusinovic, “SDN control framework for QoS provisioning”, in *2014 22nd Telecommunications Forum Telfor (TELFOR)*, IEEE, Nov. 2014, pp. 111–114. DOI: 10.1109/TELFOR.2014.7034369.
 - [145] S. Gorlatch and T. Humernbrum, “Enabling high-level QoS metrics for interactive online applications using SDN”, in *2015 International Conference on Computing, Networking and Communications (ICNC)*, IEEE, Feb. 2015, pp. 707–711, ISBN: 978-1-4799-6959-3. DOI: 10.1109/ICCNC.2015.7069432.
 - [146] V. Kotronis, R. Klöti, M. Rost, P. Georgopoulos, B. Ager, S. Schmid, and X. Dimitropoulos, “Stitching Inter-Domain Paths over IXPs”, in *SOSR '16: Proceedings of the Symposium on SDN Research*, New York, NY, USA: Association for Computing Machinery, Mar. 2016, pp. 1–12, ISBN: 978-1-45034211-7. DOI: 10.1145/2890955.2890960.
 - [147] Q.-V. Pham and W.-J. Hwang, “Network utility maximization-based congestion control over wireless networks: A survey and potential directives”, *IEEE Communications Surveys Tutorials*, vol. 19, no. 2, pp. 1173–1200, 2017. DOI: 10.1109/COMST.2016.2619485.
 - [148] Chrome Team, *A safer default for navigation: HTTPS*, <https://bit.ly/2V1KWrS>, 2021.
 - [149] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need”, *Advances in neural information processing systems*, vol. 30, 2017.
 - [150] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding”, *arXiv preprint arXiv:1810.04805*, 2018.
 - [151] F. O. Reports, *Netflix Subscriber Count Grows to 222M*, <https://frontofficesports.com/netflix-subscriber-count-grows-to-222m>, Accessed: 2022-03-20, 2022.

-
- [152] The Interactive Advertising Bureau (IAB), *Live Video Streaming – A Global Perspective*, <https://bit.ly/2QuzcJI>, Jun. 2018.
 - [153] S. C. Madanapalli, A. Mathai, H. H. Gharakheili, and V. Sivaraman, “Modeling Live Video Streaming: Real-Time Classification, QoE Inference, and Field Evaluation”, *ArXiv e-prints*, Dec. 2021. DOI: 10.48550/arXiv.2112.02637. eprint: 2112.02637.
 - [154] *Twitch Engineering: An Introduction and Overview*, <http://bit.ly/2sb86hv>, 2018.
 - [155] K. Pires and G. Simon, “Youtube live and twitch: A tour of user-generated live streaming systems”, in *Proc. ACM MMSys*, ACM, 2015, pp. 225–230.
 - [156] C. Zhang and J. Liu, “On Crowdsourced Interactive Live Streaming: A Twitch.Tv-Based Measurement Study”, in *Proc. ACM NOSSDAV*, Portland, USA, 2015.
 - [157] Will Law, *Ultra-Low-Latency Streaming Using Chunked-Encoded and Chunked-Transferred CMAF*, <https://www.akamai.com/us/en/multimedia/documents/white-paper/low-latency-streaming-cmaf-whitepaper.pdf>, Oct. 2018.
 - [158] Google, *Google chrome*, Jan. 2022. [Online]. Available: <https://www.google.com/chrome/>.
 - [159] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, and J. Iyengar, “The quic transport protocol: Design and internet-scale deployment”, in *Proc. ACM SIGCOMM*, Los Angeles, USA, Aug. 2017.
 - [160] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, “A buffer-based approach to rate adaptation: Evidence from a large video streaming service”, *ACM SIGCOMM CCR*, vol. 44, no. 4, pp. 187–198, 2015.
 - [161] S. Winkler and P. Mohandas, “The evolution of video quality measurement: From psnr to hybrid metrics”, *IEEE Transactions on Broadcasting*, vol. 54, no. 3, pp. 660–668, 2008.
 - [162] N. Barman, S. Schmidt, S. Zadtootaghaj, M. G. Martini, and S. Möller, “An evaluation of video quality assessment metrics for passive gaming video streaming”, in *Proc. ACM Packet Video Workshop*, Amsterdam, Netherlands, Jun. 2018.
 - [163] Z. Li, C. Bampis, J. Novak, A. Aaron, K. Swanson, A. Moorthy, and J. De Cock, *Vmaf: The journey continues*, <http://bit.ly/2Nad05K>, 2018.
 - [164] *Global Games Market to Generate 175.8 Billion in 2021*, 2021. [Online]. Available: <https://newzoo.com/insights/articles/global-games-market-to-generate-175-8-billion-in-2021-despite-a-slight-decline-the-market-is-on-track-to-surpass-200-billion-in-2023/>.
 - [165] J. Spjut, B. Boudaoud, K. Binaee, J. Kim, A. Majercik, M. McGuire, D. Luebke, and J. Kim, “Latency of 30 Ms Benefits First Person Targeting Tasks More Than Refresh Rate Above 60 Hz”, in *Proc. ACM SIGGRAPH Asia 2019 Technical Briefs*, Brisbane, QLD, Australia, 2019, pp. 110–113, ISBN: 9781450369459.
 - [166] *Wtfast*, Oct. 2021. [Online]. Available: <https://www.wtfast.com/en/>.

-
- [167] *Exitlag*, Oct. 2021. [Online]. Available: <https://www.exitlag.com/en/>.
 - [168] *Subspace: Dedicated network for real-time applications*, Oct. 2021. [Online]. Available: <https://subspace.com/>.
 - [169] *Oneqode: The gaming infrastructure company*, Oct. 2021. [Online]. Available: <https://www.oneqode.com/>.
 - [170] *Here's how many people play Fortnite*, 2021. [Online]. Available: <https://www.gamesradar.com/au/how-many-people-play-fortnite/>.
 - [171] M. Quwaider, A. Alabed, and R. Duwairi, "The impact of video games on the players behaviors: A survey", *Proc. of 10th International Conference ANT*, vol. 151, pp. 575–582, 2019.
 - [172] B. Dowling, *The Trusted Source for IP Address Data*, 2021. [Online]. Available: <https://ipinfo.io/>.
 - [173] S. Sanfilippo, *Active network security tool*, 2021. [Online]. Available: <http://www.hpiping.org/>.
 - [174] D. Schweikert, *Fping homepage*. [Online]. Available: <https://fping.org/>.
 - [175] F. Baker and G. Fairhurst, *IETF Recommendations Regarding Active Queue Management*, RFC 7567, Jul. 2015. DOI: 10.17487/RFC7567. [Online]. Available: <https://www.rfc-editor.org/info/rfc7567>.
 - [176] R. Kundel, J. Wallerich, W. Maas, L. Nobach, B. Koldehofe, and R. Steinmetz, "Queueing at the telco service edge: Requirements, challenges and opportunities", in *Workshop on Buffer Sizing. Stanford, US*, 2019.
 - [177] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, "Ndpi: Open-source high-speed deep packet inspection", in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, IEEE, 2014, pp. 617–622.
 - [178] D. Gupta, *Squid game clogs up the korean network and providers sue netflix*, Oct. 2021. [Online]. Available: <https://techunwrapped.com/squid-game-clogs-up-the-korean-network-and-providers-sue-netflix/>.
 - [179] N. Statt, *Fortnite on pc is now over 60gb smaller, thanks to epic optimization*, Oct. 2021. [Online]. Available: <https://www.theverge.com/2020/10/21/21526916/fortnite-pc-file-size-60gb-smaller-epic-games-optimization>.
 - [180] Allot, *Data traffic classification*, Jan. 2022. [Online]. Available: <https://www.allot.com/traffic-identification/>.
 - [181] T. Høiland-Jørgensen, P. McKenney, dave.taht@gmail.com, J. Gettys, and E. Dumas, *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*, RFC 8290, Jan. 2018. DOI: 10.17487/RFC8290. [Online]. Available: <https://www.rfc-editor.org/info/rfc8290>.
 - [182] X. Zhang, S. Sen, D. Kurniawan, H. Gunawi, and J. Jiang, "E2e: Embracing user heterogeneity to improve quality of experience on the web", in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 289–302.

- [183] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, “Fine-grained queue measurement in the data plane”, in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 15–29.
- [184] M. Mu, A. Mauthe, and F. Garcia, “A utility-based qos model for emerging multimedia applications”, in *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, IEEE, 2008, pp. 521–528.
- [185] S. Sinha, S. Kandula, and D. Katabi, “Harnessing tcp’s burstiness with flowlet switching”, in *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, Citeseer, 2004.
- [186] D. Ding, M. Savi, G. Antichi, and D. Siracusa, “An incrementally-deployable p4-enabled architecture for network-wide heavy-hitter detection”, *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 75–88, 2020.
- [187] S. Shenker, “Fundamental design issues for the future internet”, *IEEE Journal on selected areas in communications*, vol. 13, no. 7, pp. 1176–1188, 1995.
- [188] P4Lang, *Bmv2: The reference p4 software switch*, Jan. 2022. [Online]. Available: <https://github.com/p4lang/behavioral-model>.
- [189] Linux, *Traffic control utility*, 2021. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc.8.html>.
- [190] PJSIP, *Pjsip - open source sip, media, and nat traversal library*, Jan. 2022. [Online]. Available: <https://www.pjsip.org/>.
- [191] C. Project, *Web page replay go*, Jun. 2021. [Online]. Available: https://github.com/catapult-project/catapult/tree/master/web_page_replay_go.
- [192] Alexa, *Alexa - top sites*, Jun. 2021. [Online]. Available: <https://www.alexa.com/topsites/>.
- [193] J. Dugan, S. Elliott, B. Mah, J. Poskanzer, and K. Prabhu, *Iperf3—perform network throughput tests*, 2019.
- [194] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment”, *Linux journal*, vol. 2014, no. 239, p. 2, 2014.
- [195] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, “The design and implementation of open vswitch”, in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 117–130.
- [196] K. Kaur, J. Singh, and N. S. Ghumman, “Mininet as software defined networking testing platform”, in *International Conference on Communication, Computing & Systems (ICCCS)*, 2014, pp. 139–42.
- [197] D. G. Balan and D. A. Potorac, “Linux htb queuing discipline implementations”, in *2009 First International Conference on Networked Digital Technologies*, IEEE, 2009, pp. 122–126.

- [198] R. Jain, A. Duresi, and G. Babic, “Throughput fairness index: An explanation”, in *ATM Forum contribution*, vol. 99, 1999.
- [199] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, “Measuring tcp round-trip time in the data plane”, in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 35–41.
- [200] M. Ghasemi, T. Benson, and J. Rexford, “Dapper: Data plane performance diagnosis of tcp”, in *Proceedings of the Symposium on SDN Research*, 2017, pp. 61–74.
- [201] *Intel Tofino 2 P4 Programmability with More Bandwidth*, [Online; accessed 7. Apr. 2022], Apr. 2022. [Online]. Available: <https://www.intel.com.au/content/www/au/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html>.