
Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, electronics and Communications Engineering
($35^{th}$cycle)

# Machine learning for Quality of Experience in real-time applications

**Dena Markudova**
******

**Supervisor:**
Prof. Michela Meo, Ph.D

**Doctoral Examination Committee:**
Prof. Tobias Hoßfeld, Ph.D, University of Würzburg, Germany
Andra Lutu, Ph.D, Telefónica Research, Spain

Politecnico di Torino
2023

# Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Dena Markudova
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

# Acknowledgements

First and foremost, I'd like to thank my mentor Michela Meo, who, throughout the years, became a friend and confidante. Her deep technical knowledge and great empathetic character coexist in perfect harmony, making her a dream mentor. A big thank you to Marco Mellia, who introduced me to the world of research and believed in me. If it weren't for his gentle persuasive skills, I wouldn't be here. To Gianluca, with whom I shared the better part of this journey. I couldn't have wished for a better colleague. To Martino, to whom I own all my knowledge on networking, research and writing papers. Cannot thank you enough. To the rest of the ML4QoE team: Maurizio and Paolo. Thank you for your immense support and guidance. To my reviewers, Tobias Hoßfeld and Andra Lutu. Thank you for reading this thesis and for your valuable feedback.

To the labmates that became so much more - Francesca and Ale. Thank you for your everlasting support and all the moments spent together. To all the SmartDaters in the office with whom I crossed paths: AleCioc, Sara, Nik, Luca G, Silvia, Phil, Matteo, Luca V, Danilo. Thank you for making a great working atmosphere and filling the days with joy and laughter. To Sata, for his invaluable support during our amazing internship at Netflix. To my Italian family, Ximena, Cantius, Matteo, Vittoria, Sergio and Gabriele, for making life colourful. To my best friends from Macedonia, Sara, Jana and Marija, for making all the days feel lighter. To my mom for always believing in my academic pursuits, even when I myself didn't. To my dad for encouraging my engineering mind all my life. To my sister for always being a happy and reassuring presence. To my nephew, whose presence always grounds me and reminds me of what's important in life. And most importantly, to my partner Marko, who is my greatest supporter and the wind in my back.

This work was sponsored by the *SmartData@PoliTO center on Big Data and Data Science* and *Cisco Systems Inc*.

# Abstract

Internet real-time communication (RTC) platforms include any application that streams audio and video in real-time, such as video-conferencing applications (VCAs) and cloud gaming. These applications have seen a considerable surge in popularity in recent years, especially during and after the COVID-19 pandemic. VCAs are the main enabler of remote working, which has become the de-facto standard way of work for many companies. Nowadays there are countless proprietary VCA applications on the market. In this context, it is becoming increasingly important to maximize the Quality of Experience (QoE) of users of RTC applications.

In this thesis, we explore ways to monitor and control the network, in order to improve the QoE of RTC applications, using Machine Learning (ML) techniques.

We first give a vast overview of the most popular video-conferencing applications on the market today (Chapter 3), how they operate, what protocols they use and how we can distinguish them in a traffic mix.

Second, we propose a comprehensive application-level observability system for the network layer, that leverages Machine Learning. It contains two ML classifiers, trained offline, that predict traffic characteristics, such as the RTC application being used and the type of media traffic exchanged (e.g. audio, video, screen sharing etc.). In the event of worsening network conditions, the network control can apply select management techniques, such as bandwidth allocation or path selection, to improve the conditions. The system inspects the packet headers in a traffic mix and makes decisions based on packet statistics collected over time bins before or during the video call. We prototype the classifiers in Chapters 6 and 7 and show their effectiveness in providing application-level details for video-conferencing applications.

We also release the module used to systematically calculate network traffic features from raw packets as a standalone software. It can be used for analysis of

RTC applications and ML feature construction for different kinds of networking tasks (Chapter 5).

Finally, we propose a network control system for the application layer: a congestion control algorithm for real-time applications, that uses Reinforcement Learning (Chapter 8). In a scenario with an RTP sender and receiver, it gains information about the network conditions at the receiver-side, such as receiving rate, one-way delay and loss ratio and predicts the available bandwidth in the next time bin. Using this information, it controls the sending rate of the sender, thereby mitigating network congestion. This algorithm is envisioned to work at the client side, in the RTC application.

To sum up, this thesis explores the use of Machine Learning to improve networking: we use ML classification algorithms to differentiate media types in real-time traffic. We apply Natural Language Processing techniques to classify video-conferencing applications based on domain names and we employ Reinforcement Learning (RL) for Rate adaptation of RTC on the application layer. Results show that these techniques are promising for real-time traffic monitoring, visibility and management.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Nowadays, life without the Internet is unimaginable. The Internet is ubiquitous, it has spread to a large part of the world, reaching billions of users, with more and more services on the rise as the available bandwidth grows [1]. One service that has seen a significant surge of popularity, are real-time communication (RTC) applications. RTC applications encompass applications that stream video and audio in real time, such as video-conferencing applications (VCAs) and cloud gaming. VCAs allow individuals and groups of people to communicate on video calls. We especially saw their importance during the COVID-19 pandemic of 2020/21, when the social distancing and lockdown measures adopted to curb the outbreak forced millions of people to communicate solely by using VCA platforms, for both work and leisure. This led to a global increase of RTC traffic by more than 200% [2, 3]. VCAs are now the main enabler of remote working, which has become the de-facto standard way of work for many companies [4]. They have also become the most popular way to reach friends and relatives, substituting the traditional voice telephony (e.g. *Facetime* vs. a phone call).

In this context, reliability of RTC applications has become critical. It has also become essential to maximize the Quality of Experience (QoE) of users of RTC applications. In this thesis, we present QoE improvement scenarios on both the network layer and the application layer. On the network layer we present solutions for traffic observability and classification in-the-wild (Chapters 5, 6 and 7). These systems can then lead to QoE improvements, through deployment scenarios such as those described in Chapter 4. On the application layer, we suggest a network control

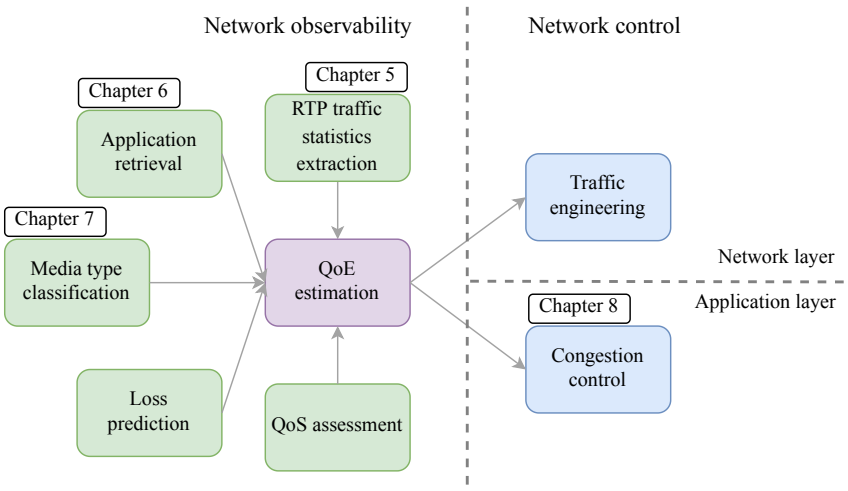



Fig. 1.1 Network observability and control system.

scheme that directly optimizes QoE metrics in terms of receiving rate, one-way delay and packet loss (Chapter 8).

This thesis has been developed as part of a 3-year project with *Cisco Systems Inc.*, who expressed the interest and need for a system that could be implemented in network equipment and offer QoE as a service. Thus, the contents of the thesis is part of a bigger, meticulous system that offers QoE-centered network observability and control for RTC applications. The system is depicted on Figure 1.1. Indeed, we envision a few software modules that work together to estimate the QoE of RTC traffic. Based on that, we can suggest improvements at both the network layer and the application layer. In this thesis, we address four of the system modules: RTP traffic statistics extraction, Application retrieval, Media type classification and Congestion control. Each of the modules is discussed in a separate chapter, as marked on Figure 1.1. We also perform some QoE estimation as part of the congestion control algorithm. In the scope of the project, the research group also developed two other modules: the QoS assessment module [5] and Loss prediction module [6, 7]. The precise QoE estimation and traffic engineering are left as future work, with some ideas and possible deployment scenarios outlined in Chapter 4.

## 1.1 Network layer observability

The network is very important for the user QoE. The quality of the user connection, network topology and network management all have a first-hand influence on user QoE [8–10]. Good network management policies can help avoid impairments, service misbehavior and consequently user churn. The first step towards effective QoE improvement is application-level visibility on the network layer. Application-level visibility can enable better capacity planning, traffic-prioritization policies and help with network troubleshooting.

Traditional approaches used to gain visibility on network traffic are port-based and Deep Packet Inspection (DPI) techniques. Port-based approaches include matching the protocol ports to distinguish the type of traffic and DPI involves matching the payload with pre-defined patterns. However, most applications today use dynamic ports, annulling port-based traffic classification. Moreover, the widespread adoption of encryption of the packet payload hinders use of DPI [11]. Most RTC applications use the RTP protocol (see Chapter 3) to encapsulate the multimedia content. RTP packets have the header in-clear and the payload encrypted. Thus it is difficult to guess application-level information on RTP traffic using traditional techniques.

For these reasons, the networking community has turned towards Machine learning (ML) to help with network traffic visibility. Machine learning can leverage packet statistics and reveal network insights and relationships that cannot be captured by simple heuristics. As an approach, ML has been revolutionary for network traffic classification and visibility [12]. The growth of ML for networking is also supported by the increasing amount of network data available. In this thesis, we apply ML techniques to RTC network traffic, with the goal of classification and observability. We shed light on RTC applications, without any cooperation from application proprietary servers, just by inspecting user traffic.

A full classification pipeline that aims to regain visibility on RTC traffic is depicted on Figure 1.2. Taking all UDP traffic, we first have to identify RTP streams (Classification (1)). Then, we establish whether the RTP streams are part of a video call in progress or another application, e.g. cloud gaming ((2)). Next, we can distinguish what is the VCA application being used - Webex, Jitsi, Zoom etc. ((3)) and finally what is the media type of the underlying stream ((4)). In this thesis, we address the classification problems (1), (3) and (4). For, (1), RTP streams can

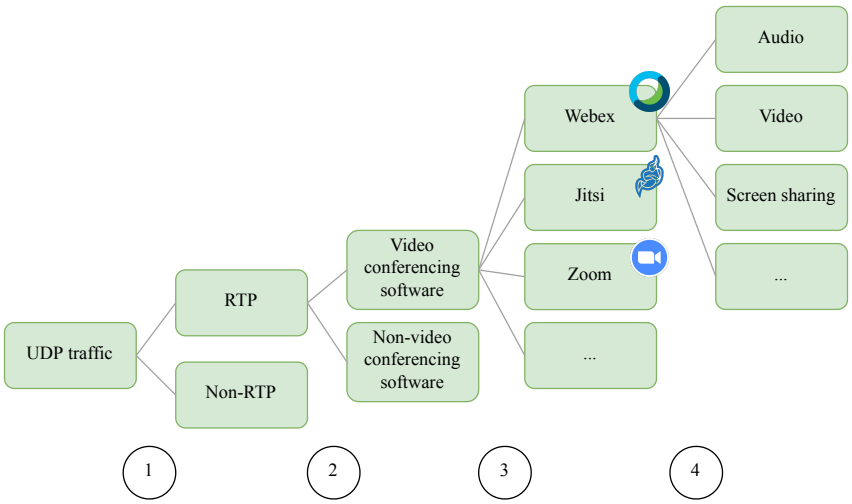



Fig. 1.2 Classification tree for network observability.

be distinguished from UDP traffic by matching the protocol headers using popular traffic analysis tools, such as *Wireshark*, with little manual assistance. Problem (3) is addressed by Chapter 6 and problem (4) by Chapter 7, both using machine learning techniques. Problem (2) is an interesting future work, however, our approach for application retrieval ((3)), works by reading domain names and supports a category "Other", so it can be applied to RTP traffic directly, skipping step (2).

## 1.2 Application layer control

RTC applications use the RTP protocol mostly over UDP, so they are not subject to TCP congestion control (CC) protocols. Instead, they implement Congestion control via Rate adaptation at the application layer, by using a feedback mechanism between the sender and receiver based on the Real-time Control Protocol (RTCP). The goal of these CC algorithms is to control the sending rate. The sending rate directly affects the packet delay, loss rate and throughput, which are the main drivers of QoE [13].

Some of the applications use proprietary CC algorithms, while all applications that run in the browser use the algorithm Google Congestion Control (GCC) [14]. GCC is a heuristic algorithm that makes decisions based on the one-way queuing delay and loss ratio at the receiver. However, in a complicated network scenario, such as wireless network links with very variable bandwidth, it is hard to optimize all network metrics with a heuristic.

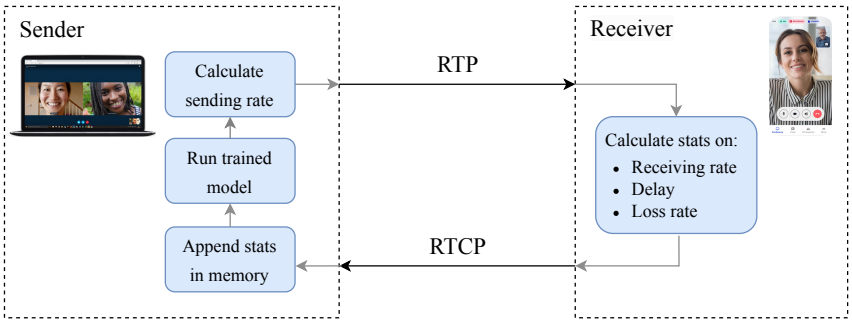



Fig. 1.3 Congestion control system.

To overcome this limitation, in Chapter 8, we propose the use of Reinforcement Learning (RL) to control the sending rate. Indeed, control problems can naturally be translated to Reinforcement learning tasks. Moreover, we can design the reward to exactly generate the network behaviour we want and the resulting model will have high capacity of understanding the complicated relationship between different network metrics, such as receiving rate, bandwidth utilization, latency and loss rate. The system infrastructure is depicted on Figure 1.3. We envision a sender, that, based on the network statistics received via the RTCP protocol, runs the trained RL model and controls the sending rate.

## 1.3 Thesis outline

In this section we outline the thesis structure and the main contributions of each chapter.

Chapter 2 outlines previous work on the problems tackled by this thesis and how they compare with our solutions. We first describe works that investigate the operation of RTC applications, then works that address RTC traffic classification and finally works on Reinforcement learning for Congestion control of RTC applications.

Chapter 3 presents a comparative study of the 13 most famous RTC applications on the market today, demystifying how they operate on the network layer, what protocols they use, what server architecture and how they can be identified in a mix of traffic. This chapter servers as a Background, to get acquainted with RTC applications and traffic, which are the topic of this thesis.

Chapter 4 describes a possible deployment scenario for the systems prototyped in this thesis, in the form of an in-network monitoring and control software, that could either run on Edge network equipment or in an SDN scenario.

Chapter 5 presents *Retina*, an open-source tool for flexible analysis of RTC traffic. Given raw traffic captures of RTC traffic, *Retina* outputs various logs and graphs, according to user instructions. One of the logs is of per-second traffic statistics, that we then use to feed the Machine Learning algorithms of Chapter 7.

Chapter 6 introduces a classifier that uses supervised machine learning to distinguish between different RTC applications, by just looking at the packet headers prior to the start of the video call. It uses Natural Language Processing (NLP) techniques to process the textual domain name data and one-vs-rest classification to identify the application.

Chapter 7 outlines another ML classifier, that uses statistical features of the traffic, produced by *Retina* (Chapter 5), to differentiate RTP streams by the type of media content they carry (audio, video, screen sharing etc.). It works with one second of traffic as input and apart from standard media, it can also discern different qualities of video and error correction streams.

Finally, Chapter 8 portrays *ReCoCo*, an algorithm for Congestion control of RTC applications that is based on Reinforcement learning. *ReCoCo* gains information about the network conditions at the receiver-side, such as receiving rate, one-way delay and loss ratio and predicts the available bandwidth in the next time bin, so that the sender can adjust their sending rate. It does so using a carefully curated reward function.

## 1.4 List of publications

In this section, we outline the papers published during the PhD, dividing them into two subsets: those relevant for the thesis and additional works.

Publications described in the thesis:

1. **A comparative study of RTC applications**, Nistico, Antonio; Markudova, Dena; Trevisan, Martino; Meo, Michela; Carofiglio, Giovanna, (2020) In: 2020 IEEE International Symposium on Multimedia (ISM) - *Chapter 3*

2. **Online Classification of RTC Traffic**, Perna, Gianluca; Markudova, Dena; Trevisan, Martino; Garza, Paolo; Meo, Michela; Munafò, Maurizio; Carofiglio, Giovanna, (2020) In: 2020 IEEE 18th Annual Consumer Communications and Networking Conference (CCNC) - *Chapter 7: conference version*

3. **What's my App?: ML-based classification of RTC applications**, Markudova, Dena; Trevisan, Martino; Garza, Paolo; Meo, Michela; Munafo, Maurizio; Carofiglio, Giovanna, (2021) In: Performance Evaluation Review - *Chapter 6*

4. **Retina: An open-source tool for flexible analysis of RTC traffic**, Perna, Gianluca; Markudova, Dena; Trevisan, Martino; Garza, Paolo; Meo, Michela; Munafò, Maurizio, (2022) In: Computer Networks 202 - *Chapter 5*

5. **Real-Time Classification of Real-Time Communications**, Perna, Gianluca; Markudova, Dena; Trevisan, Martino; Garza, Paolo; Meo, Michela; Munafo, Maurizio; Carofiglio, Giovanna, (2022) In: IEEE Transactions on Network and Service Management - *Chapter 7: journal version*

6. **ReCoCo: Reinforcement learning-based Congestion control for Real-time applications**, Markudova, Dena; Meo, Michela, (2023) Accepted. To appear in 2023 IEEE 23rd International Conference on High Performance Switching and Routing (HPSR) - *Chapter 8*

Other published works:

7. **Heterogeneous industrial vehicle usage predictions: A real case**, Markudova, Dena; Baralis, Elena; Cagliero, Luca; Mellia, Marco; Vassio, Luca;

Amparore, Elvio; Loti, Riccardo; Salvatori, Lucia, (2019) In: Workshops of the EDBT/ICDT Joint Conference, EDBT/ICDT-WS 2019

8. **Impact of Charging Infrastructure and Policies on Electric Car Sharing Systems**, Ciociola, Alessandro; Markudova, Dena; Vassio, Luca; Giordano, Danilo; Mellia, Marco; Meo, Michela, (2020) In IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)

9. **Preventive maintenance for heterogeneous industrial vehicles with incomplete usage data**, Markudova, Dena; Mishra, Sachit; Cagliero, Luca; Vassio, Luca; Mellia, Marco; Baralis, Elena; Salvatori, Lucia; Loti, Riccardo, (2021) In: Computers in Industry

10. **Where did my packet go? Real-time prediction of losses in networks**, Song, Tailai; Markudova, Dena; Perna, Gianluca; Meo, Michela, (2023) Accepted. To appear in ICC 2023-IEEE International Conference on Communications

I have had a heavy individual contribution to all the publications described in this thesis. More specifically, for papers (3) and (6), I developed the whole pipeline on my own - from data collection to algorithm application and result analysis. For paper (1), I ran the full data analysis. For papers (2), (4) and (5), I had equal contribution as the first author. We collected the data together, coded *Retina* collaboratively and split the work on the data analysis and algorithm application. I had the leading role in writing all of the papers.

## 1.5 Open code and datasets

We make all the code and datasets we use in this thesis public. Table 1.1 outlines all the sources. We believe that the networking community still lacks proper datasets, tools and simulators for benchmarking different algorithms and problems. This significantly slows down networking research, as developing significant experiments requires a large set of competences from a researcher: expertise in different programming languages and software, network protocols, simulators and data analysis. We strongly believe that open-source data and software that has a shallow learning curve can significantly accelerate networking research, especially Machine learning applied to Networking, since ML strongly depends on datasets for both development and benchmarking.

Table 1.1 All publically available data and code.

| Chapter | Open code / dataset |
|---|---|
| 3 A study of RTC applications | https://smartdata.polito.it/a-comparative-study-of-rtc-applications-the-dataset/ |
| 5 Retina | https://github.com/GianlucaPoliTo/Retina |
| 6 RTC application retrieval | https://github.com/denama/RTC_apps_classifier |
| 7 Media type retrieval | https://smartdata.polito.it/rtc-classification/ |
| 8 ReCoCo | https://github.com/denama/ReCoCo |

## 1.6 Taxonomy

Throughout the thesis, we use the general term "RTC applications" to indicate video-conferencing applications, since all our systems are deployed with data from video-conferencing applications. However, all methodologies can easily be extended to general RTC applications that use the RTP protocol for communication (such as cloud gaming platforms).

Since the topic of the thesis is improvement of user QoE, we hereby explain all necessary terms. By definition, Quality of Service (QoS) measures key network performance metrics, such as delay, jitter, throughput or packet loss. They can be measured at both network level (network QoS) or application level (application-level QoS). Quality of Experience (QoE) captures the actual user experience and is often a subjective metric. A traditional example is the 1-5 star review after a video call. While the two are closely related, they are not the same thing. A user can sometimes have a bad experience even when the QoS is satisfactory, or vice-versa. In Chapter 8,

we describe "QoE scores" related to the receiving rate, delay and loss rate. These are more specifically Key Quality Indicators (KQIs), that influence the actual QoE. For the sake of readability, in the remainder of the thesis we use the term "QoE metric" as un umbrella term for all these metrics.

# Chapter 2

# Related work

In this section we present a Literature review in regard to the operation of RTC applications, network traffic classification that includes RTC traffic and congestion control for RTC traffic. We include direct comparison of some works with work presented in different chapters of the thesis.

## 2.1 On the operation of RTC applications

The operation of RTC applications has been studied since their introduction in the early 2000s. Several works target Skype traffic, since it was the pioneer in the world of modern RTC. Bonfiglio *et al.* [15] show how Skype was using aggressive obfuscation of traffic at its early stage, mainly to avoid ISP-level throttling and propose a heuristic algorithm to identify it. They again provide a detailed analysis of Skype traffic in [16] and [17], based on traffic measurements, which leads to the same conclusions of Guha *et al.* [18]. Moreover, Ehlert *et al.* [19] develop signatures for the detection of Skype traffic, while Baset *et al.* [20] analyze its key functions: login, NAT/Firewall traversal, and media transfer. Hoßfeld *et al.* [21] provide an analysis of Skype VoIP traffic in mobile networks, focusing on Quality of Service (QoS) and QoE.

In Chapter 3, we provide an updated view of Skype traffic, after the acquisition by Microsoft. We show that it has converged to use a more standard approach based on RTP and shares its behavior with Microsoft Teams.

Telegram, known for its security features, has been object of several studies too. In particular, many works provide a forensic analysis of Telegram on different customer devices like MacOS [22], Android smartphones [23] and Android devices in general [24]. These works try to describe the artifacts generated by the Telegram application on each type of device. Studying the security features of RTC applications is out of the scope of this thesis. However, we testify that Telegram is peculiar among RTC applications in the employed network protocols as well.

Michel *et al.* [25] demystify Zoom in production traffic by digging deep into its protocol and header format. They show how to extract metrics that closely relate to the quality of a Zoom call, such as media bit rates, frame rate, and frame-level jitter.

There are fewer works that compare different RTC applications. Azfar *et al.* [26] study ten Android VoIP applications, mainly from the security point of view. Karya *et al.* [27] compare RTP traffic in Whatsapp and Skype, under mobile networks. Wuttidittachotti *et al.* [28] provide a study on the perceived QoE of three well-known VoIP applications, using Perceptual Evaluation of Speech Quality (PESQ). Xu *et al.* [29] report a measurement study of Google+, iChat, and Skype, unveiling important information about their key design choices and performance. They extend their analysis in [30], this time focusing on FaceTime, Google Plus Hangout, and Skype. Sutkino *et al.* [31] compare the instant messaging services of WhatsApp, Viber and Telegram in terms of security, speed and ease-of-use. Patel *et al.* [32] evaluate the performance of WhatsApp and Skype in terms of their data consumption, as well as quality of the VoIP calls. Their results show that WhatsApp uses less data and also provides better call quality under poor network conditions. Chang *et al.* [33], instead, provide a measurement study of Zoom, Webex and Google Meet, characterizing the user-perceived performance. They look at streaming lag in correlation to geographical placement of the application servers, bandwidth requirements and CPU and battery consumption. Their data is mostly emulated and partly real network data from a residential network.

With respect to these works, in Chapter 3, we target a wider range of RTC applications, comparing 13 leading services in the consumer and business market segments. We provide an updated overview of the technologies used at the network level, showing a large set of peculiar protocols and behaviors. Indeed, popular applications change very often over time – see the Skype case. We believe Chapter 3

provides a rich but concise summary of the currently adopted solutions, unifying and updating what researchers have discovered in previous works.

Note that the studies by Chang *et al.* [33] and Michel *et al.* [25] were published **after** the publishing of our paper described in Chapter 3. However, the novelty points we present still hold.

## 2.2 On RTC traffic classification

Network traffic classification has been extensively studied since the birth of the Internet [34]. Due to the widespread adoption of encryption and the use of proprietary protocols, traditional approaches based on mere DPI and port numbers fall short, and the current research tends to use statistical traffic features and machine learning techniques [35]. Recent efforts aim to identify the web services [36] or mobile applications [37] behind network traffic, predicting the QoE of web [38, 39], video [40] or smartphone [41] users. DNS has also been used extensively to mark traffic [42, 43].

Focusing on RTC traffic, many works propose techniques to identify it among other traffic categories. The authors of [15] use a stochastic characterization of Skype traffic to obtain an ML-based model to be used for classification. In [44], UDP flows are classified into different classes, including Skype and RTP-based traffic, using SVM models and statistical signatures of the payload. The approach proposed in [45] leverages statistical properties of RTP to differentiate between voice and data traffic. The authors of [46] propose a method to detect WebRTC sessions at run-time based on statistical pattern recognition. Finally, some approaches target signaling mechanisms of RTC applications to identify Skype traffic through in-clear headers exchanged during session setup [47].

Fewer works address the classification of media streams carried by RTP streams. Authors of [48] train machine learning classifiers to distinguish, among other classes, video and audio flows, targeting the WeChat messaging application. The approach presented in [49] identifies 20 codecs used for compression of audio, based on packet size, RTP timestamp delta, payload type and ratio between RTP timestamp delta and payload size. However, they do not use machine learning but a simple lookup table. In [50], the authors use statistics on the packet size as a distinguishing feature between audio, video streaming, browser, and chat traffic. They use interesting

features, although few and classify into broad traffic classes. As a model, they opt for an interpretable Decision tree, similar to our result in Chapter 7.

In Section 7.4.6 of Chapter 7, we investigate the use of transfer learning techniques for our classification problem. A few works already proposed their use for problems related to networking, albeit in different contexts. Authors of [51] use transfer learning in wireless networks for a caching procedure. Instead, the approach proposed in [52] used it in combination with Deep Reinforcement Learning to solve the reconfiguration problem in the context of experience-driven networking. It has also been used for QoE estimation of video streaming [53, 54]. The transfer learning technique we use, CORAL [55], has already been used in optical networks for assisted quality of transmission estimation of an optical lightpath [56]. In this thesis, we apply it to the RTC scenario, trying to align statistical features of network traffic from different applications.

The ultimate goal of our RTC traffic classification is the improvement of QoS and users' QoE. These aspects have been studied, focusing on the relationship between QoS and QoE [9, 57], targeting the WebRTC [58] and mobile [59] scenarios.

The closest work to the Media type classification presented in Chapter 7 is the approach proposed by Choudhury *et al.* [60]. There, the authors design a system to classify RTP traffic to the employed codec. They develop a similar ML pipeline, to classify audio traffic into three Variable Bit Rate (VBR) codecs, thus identifying three types of audio. Conversely, we distinguish seven classes, two of which are audio (audio and FEC audio), four are video (three video qualities and FEC video) and one is screen sharing. Another similarity is that they classify RTP streams separately by time bin, with a granularity coarser than ours – 10-20 seconds vs. 1 second. They use two types of features: statistical features of packet sizes (such as mean, standard deviation, mode, etc.) and entropy-based features (4 types of entropy calculations on the RTP payload of the packets). We follow a similar approach, using five feature groups and calculating various statistics on the distributions. Like in our system, they train offline, using 18-second streams and then the classifier is deployed in real time, over 10 seconds of stream data. They get overall 97% accuracy, similar to ours (95%). With respect to the algorithms, they opt for a 1-Nearest Neighbours, while we choose a Decision Tree.

In this thesis, we aim to give detailed visibility to RTC traffic, unveiling both RTC applications and the nature of media streams. Differently from previous works,

we classify streams into a rich set of classes including media type (audio and video), video quality and redundant data (FEC). We engineer a wider range of features and then run a thorough feature selection process. Moreover, to the best of our knowledge, we are the first ones to explicitly target real-time applications with a 1 second (or shorter) classification delay, while the past approaches base their decision on the characteristics of an entire stream, lasting 10 seconds or more.

## 2.3 On Reinforcement learning for Congestion Control of RTC applications

**Reinforcement learning in networking control tasks.** Reinforcement learning has been successfully used in many networking applications. One of the pioneers was Pensieve [61], which proposed Deep RL for ABR streaming, selecting the bitrate for future video chunks. Authors of [62] use RL for network congestion control in data centers. Numerous works have tried to improve TCP congestion control, by controlling the congestion window size with RL, as outlined in [63]. Notable examples include Aurora [64], Eagle [65] and other approaches [66–69], as well as MVFST-RL [70] for QUIC.

**Congestion control for RTC**. Congestion control in employed RTC applications today is mostly heuristic-based. There are three main algorithms standardized by the IETF RTP Media Congestion Avoidance Techniques (RMCAT) working group: Network Assisted Dynamic Adaptation (NADA) proposed by Cisco [71], Self-Clocked Rate Adaptation for Multimedia(SCReAM) proposed by Ericsson [72] and Google Congestion Control (GCC) proposed by Google [14]. They all employ delay-based mechanisms to detect congestion (NADA and SCReAM measure the one-way delay and GCC the one-way delay variation) and loss-based mechanisms as fall-back when there is buffer overflow and the delay cannot indicate the congestion state. The most employed implementation is *GCC*, used by the popular RTC application, *Google Meet*, so we use it as the baseline algorithm in Chapter 8. *GCC* suffers from a few limitations: it only responds to latency variation, so it does not note an increase in the absolute value of RTT, introducing delays in the conversation. It also becomes too conservative in the event of losses [13]. *GCC* is slow to respond to an increased bandwidth, since it increases the sending rate by just 5% every second.

This holds true also for some RTC applications that use proprietary congestion control protocols [73]. In Chapter 8, we aim to overcome these limitations by setting the reward to optimize for all three metrics: delay, losses and bandwidth utilization, at any given time, making it faster to adapt to varying network conditions.

**Reinforcement learning in RTC Congestion control.** There are three previous works that tackle exactly the problem of Rate adaptation or Congestion control for real-time communications: HRCC, CLCC and R3Net. HRCC [74] is a hybrid receiver-side scheme, that uses *GCC* as the main algorithm and at every fourth time interval tunes the bandwidth estimate with a gain coefficient generated by an RL agent. R3Net [75] is a fully RL-based solution. It uses the PPO algorithm with two Recurrent Neural Networks (RNN) as the Actor and Critic. It employs a 4-dimensional state vector, that consists of receiving rate, packet loss rate, average RTT and average packet inter-arrival time. The reward function is a very simple combination of the receiving rate, delay and loss rate in the evaluated time bin. It predicts the bandwidth every 50ms. CLCC [76] is also a fully RL-based solution. Different from other works, it uses both packet-level and frame-level statistics of traffic as states and in the reward, arguing that frame-level information is more important for RTC and doing congestion control at the application layer allows us to use it. However, this requires adding an additional exclusive channel to RTP, to send frame-level information, which is a non-trivial change to the protocol. CLCC tunes the rate by multiplying the rate in the previous time bin with a gain coefficient. Like R3Net, it uses PPO with the same RNN architecture. They run RTC in a real environment both in the training and evaluating process. They use the same QoE metrics for performance evaluation, however they add frame-level metrics as well.

Albeit having the same end goal, our work in Chapter 8 contrasts these works in a few important ways. First, we explore many different configurations of the problem formulation. We try three different algorithms, with different hyperparameters and actually find that PPO, which is used by both R3Net and CLCC is outperformed by both SAC and TD3 on the traces we use. We play with the states to see if delaying them gives better results. We meticulously design the reward function, since it is vital to the performance of the RL algorithm, outlining separate functions for all network metrics we wish to optimize, based on network standards and recommendations. We provide details on the traces in use, to better understand what kind of environments are hard for RL algorithms to solve, while previous works mention the dataset briefly,

without giving any details on the bandwidth variability. While HRCC and CLCC present one model and its general performance, we train both specific and general models, testing their performance on all traces separately to find the best trade-off between a personalized and general model.

## 2.4 Knowledge gap

Even though RTP has been studied since its introduction, there is not much literature on RTP traffic and applications as they are today. There are many papers on the operation of a specific RTC application in more detail (e.g. Skype, Zoom, Telegram), but few that compare different RTC applications. We study a wide range of modern RTC applications (13) and give the most updated technological view in the literature today. There are many works that distinguish RTP among other traffic categories, but very little that perform classification *inside* RTP traffic - classifying RTP traffic into further categories. To the best of our knowledge, we are the first ones to introduce both an RTC application classifier and a Media type classifier within RTP traffic, using packet data and statistical features. We are also the first ones to analyze RTP traffic using statistical features on a deeper level. In terms of Congestion control, there are a few papers that suggest RL for rate adaptation of RTC applications. However, the field is very young and the approaches are vague, with many unexplained design choices. We are the first ones to engineer a reward that uses separate functions for all QoE metrics we wish to improve and pioneers in using curriculum learning to optimize the order of showing environments to the RL agent.

# Chapter 3

# Operation of RTC applications

The work presented in this chapter is mostly taken from our paper *A comparative study of RTC applications*, presented at the *2020 IEEE International Symposium on Multimedia (ISM)* [77]. It serves as an introduction to Real-time Communication applications, showing which are the most popular applications on the market today, how they operate, what protocols they use, what kind of media can be exchanged and how they can be identified on the network layer in a traffic mix. Distinguishing RTP traffic in a traffic mix is a necessary first step for both application retrieval (Chapter 6) and media type classification (Chapter 7).

## 3.1 Introduction

Historically, the first proposals for real-time communication over IP networks were based on the Session Initiation Protocol (SIP) [78] for session setup and the Real Time Protocol (RTP) [79] for media stream transmission. Indeed, in the late 1990s and early 2000s, Voice-over-IP (VoIP) solutions created a market for corporate-level telephony, replacing the old public circuit-switched telephone network. Then, Skype brought the VoIP technology to individuals, allowing people to make calls via the Internet for free. Differently from previous VoIP proposals, it was based on a Peer-to-Peer (P2P) architecture and made use of encrypted and undocumented protocols [15]. Recently, dozens of new RTC applications have appeared, competing in a world where audio and video calls are part of the normal business and leisure routine. They are nowadays massively adopted and companies pay subscriptions

for premium and customized access plans. However, there is still no standard for interoperability between different applications, and even when they employ well-known protocols, the resulting mix of protocols is diverse in each application. Moreover, the vast majority of applications are closed-source and provide none or very little documentation. Even though they do this to protect the intellectual property behind the applications, it complicates the network management for Internet Service Providers (ISPs) and corporate network administrators. Prioritizing RTC traffic or blocking unauthorized applications is therefore hard, while unknown protocols might cause issues in middleboxes that rely on Deep Packet Inspection (DPI) – e.g., firewalls or NATs.

In this chapter, we study and compare different RTC applications and outline similarities and differences in the way they use the network. We collect packet traces from 13 applications, choosing them from top popular consumer solutions (Skype and Google Meet above all), to products for business communication, where Microsoft Teams and Webex Teams are notable examples. We run an extensive experimental campaign in which we capture the traffic generated by the applications using different devices and types of calls. This allows us to provide an overview of the common practices and peculiarities currently adopted by the competitors on the RTC market. Notable findings we obtain are:

- Most of the applications still use the RTP protocol for media streaming.
- They use various mixes of protocols. The most frequent ones are STUN and TURN for session setup and TLS and DTLS for control data exchange.
- We find examples of undocumented protocols used in Telegram and GoTo Meeting. Zoom uses an unknown encapsulation mechanism for RTP, while Microsoft Teams uses a non-standard, yet documented encapsulation protocol.
- Peer-to-peer communication is often exploited in calls with only two participants, when the network allows it.
- Six applications send redundant data for Forward Error Correction (FEC) or send the user's video at different qualities at the same time (Simulcast).

The collected data also allows us to characterize the traffic generated during the calls and to provide guidelines for its identification. We show that the media traffic

of some applications can be easily recognized by simply looking at the Autonomous System (AS) of the server, while others rely on large infrastructures and/or content delivery networks (CDNs). We also find that almost all of them can be identified (and blocked) using the domains the client resolves during the application execution.[1] Only for Google this is complicated, since it uses very generic domains that cannot be associated to the specific RTC service.

We make the dataset we used for the experiments public: a list of contacted ASes, domains, ports and Payload Types (PTs) per application, as well as all the collected traffic traces.[2] We believe that our data may help researchers to reproduce our results or extend them to different contexts, while also providing useful indications to network practitioners and administrators.

The remainder of the chapter is organized as follows. In Section 3.2, we provide a background on the most popular protocols used in RTC applications. Section 3.3 describes the applications under scrutiny as well as the packet traces used to analyze them. Section 3.4, illustrates the network protocols we find. Section 3.5 discusses the different design choices, while Section 3.6 provides useful guidelines for traffic identification.

## 3.2 Background

To guide the reader through the paper, in this section we provide an overview of the most common protocols that are used in RTC applications. Although this list is not meant to be exhaustive, it includes all protocols that we observe in the 13 applications under test (neglecting the undocumented solutions).

**Media streaming**. The most popular protocol for real-time media streaming is RTP [79]. Proposed in the faraway 1996, it defines a simple encapsulation mechanism in which different streams are multiplexed using a unique Synchronization source identifier (SSRC). The timestamp field reports the instant at which the content is generated and Payload Type (PT) indicates the employed video or audio codec. RTP defines a set of predefined or static PTs, and offers the possibility to define them

[1] We use the term *domain* throughout the chapter, meaning Fully Qualified Domain Name.

[2] Our dataset is available at: https://smartdata.polito.it/a-comparative-study-of-rtc-applications-the-dataset/

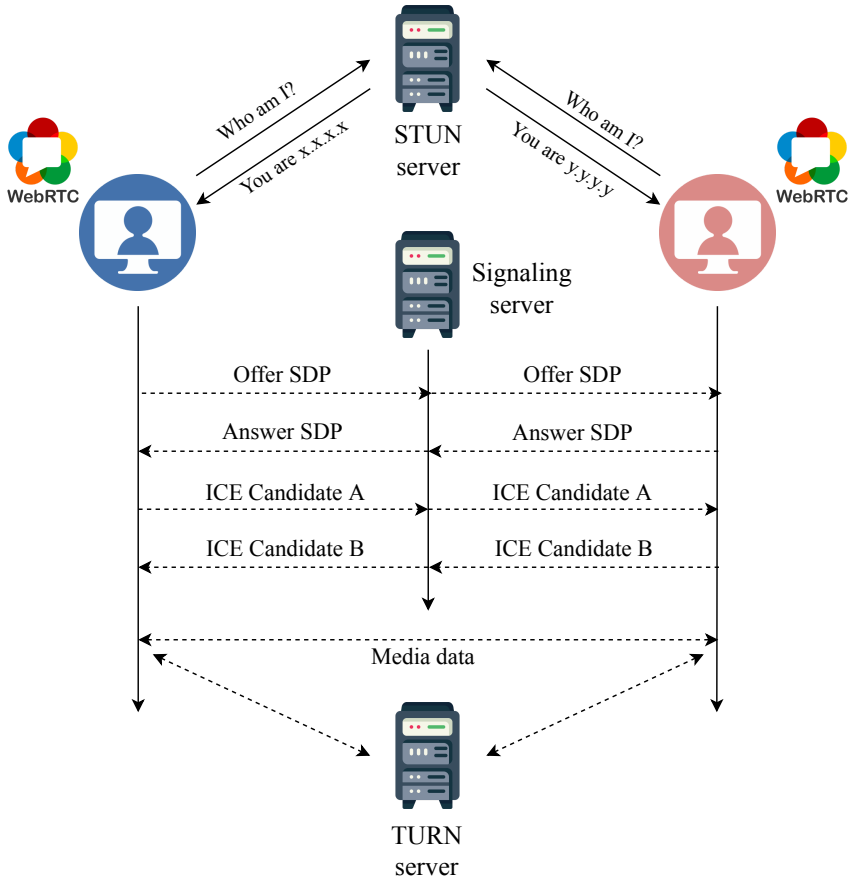



Fig. 3.1 Session setup in WebRTC.

dynamically during a session. RTP is carried over UDP or (very rarely) over TCP as a transport protocol. The support protocol RTCP is typically used beside RTP for exchange of various streaming statistics, like packet loss ratio. Secure RTP (SRTP) [80] is a variant of RTP that achieves confidentiality by encrypting the media payload while leaving all the original headers in clear. In the rest of the thesis, we use the terms *RTP* and *SRTP* interchangeably.

**Session Setup**. To establish a media session, it is necessary for the endpoints to be able to communicate with each other, especially in the case of peer-to-peer communication between participants. This is complicated by the presence of NATs, firewalls and middleboxes in general. To ensure connectivity, the applications often use the STUN protocol [81] for NAT detection and TURN [82] to relay the traffic through a server that resides on the public Internet. ICE [83] combines STUN and TURN into a single technique. The RFC 7983 [84] defines a simple mechanism for

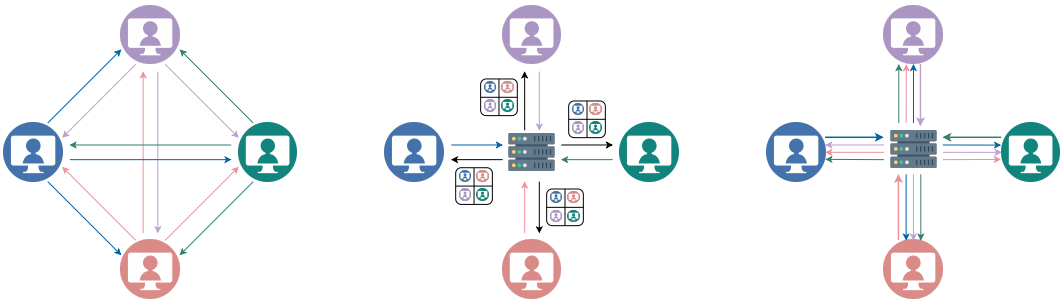



Fig. 3.2 System topologies.

multiplexing RTP, STUN and other protocols on the same UDP flow. The Session Description Protocol (SDP) [85] defines a format for negotiating the network and media characteristics of the session – e.g., the audio and video codecs. Nowadays it is (almost) always sent over encrypted channels (e.g., TLS or DTLS, see next paragraph), and as such, completely invisible to the network. The whole process is depicted on Figure 3.1.

**Additional protocols**. RTC applications use a wide range of protocols for exchanging control data, e.g., for login or chat. This typically requires confidentiality, and we observe a large prevalence of the TLS [86] and DTLS [87] protocols.

**WebRTC**. The above protocols need to be carefully coordinated to have a working RTC application. To ease the development, WebRTC [88] is a set of high-level and standard APIs that can be used in browsers and mobile applications for video and audio communication. Released in 2011, currently most browsers support WebRTC and it represents the only way for RTC applications to run via web, if we exclude application-specific plugins. WebRTC provides programming interfaces to establish media sessions, coordinating the use of the SRTP, RTCP, STUN, TURN and DTLS protocols. It also outlines a common set of media codecs and a standard congestion control algorithm (Google Congestion Control).

**System topologies.** The connection between the participants of a call can be set up in a few ways:

1. **Peer-to-Peer (P2P):** the participants connect to each other directly, without a server to relay traffic. This is sometimes used by applications when there are only two participants on a call.

2. **Multipoint Control Unit (MCU):** there is a media center, or server in the middle (an MCU), that receives multiple media streams from all parties, re-codes them and produces one video per recipient (e.g. a grid of videos of the participants or a large video of the speaker and thumbnails of the other participants).

3. **Selective Forwarding Unit (SFU):** there is a media center, which receives multiple media streams from all parties and decides which of these streams should be sent to which participant. It does not re-code video, just forwards streams. Then the endpoint is responsible to merge the incoming video streams into one.

The three system topologies are depicted on Figure 3.2. MCU and SFU topologies are obviously useful for multi-party calls. There are pros and cons to both architectures. An MCU is theoretically more efficient, because it always sends and receives one stream. However, it uses a lot of computing power and does not scale well. An SFU, on the other hand, requires a more powerful CPU on the clients, since they have to decode a large number of media streams.

## 3.3 Data collection

For the purposes of this analysis, we target 13 popular RTC applications, that we can roughly group into two categories. We first consider 9 *consumer applications*, used by people for communicating with relatives and friends and for leisure in general. The set includes Skype, historically the pioneer of VoIP, now owned by Microsoft. We also involve two competitors: Meet by Google, and Jitsi Meet, the public instance of the open-source Jitsi tool. We then consider chat and social applications that also provide the possibility of making calls: We test Whatsapp, Telegram, Facebook and Instagram. Finally, we consider FaceTime, included in all Apple products, as well as HouseParty, that suddenly became popular during 2020. In the second category we include *business platforms* for RTC, which typically provide commercial plans for enterprises. We consider Microsoft Teams and Webex Teams, that are very popular solutions used for remote working and teaching. We also place Zoom and GoTo Meeting in the business category, since they offer both a free version and premium plans for businesses. We show an overview of the tested applications in Table 3.1.

Table 3.1 Overview of the tested applications (consumer apps are on the first block, while business on the second). On the *Media* column, A=Audio, V=Video and S=Screen Sharing.

| Application | Multiparty | Desktop App | Mobile App | Browser version | Media |
|---|---|---|---|---|---|
| Skype | ✓ | ✓ | ✓ | ✓ | AVS |
| Google Meet | ✓ | | ✓ | ✓ | AVS |
| Jitsi Meet | ✓ | ✓ | ✓ | ✓ | AVS |
| WhatsApp | ✓ | | ✓ | | AV |
| Telegram | | ✓ | ✓ | | AV |
| Facebook Messenger | ✓ | ✓ | ✓ | ✓ | AV |
| Instagram Messenger | ✓ | | ✓ | | AV |
| Facetime | ✓ | ✓ | ✓ | | AV |
| HouseParty | ✓ | ✓ | ✓ | ✓ | AV |
| Microsoft Teams | ✓ | ✓ | ✓ | ✓ | AVS |
| Webex Teams | ✓ | ✓ | ✓ | ✓ | AVS |
| Zoom | ✓ | ✓ | ✓ | ✓ | AVS |
| GoTo Meeting | ✓ | ✓ | ✓ | ✓ | AVS |

All applications except Telegram allow for multiparty calls – i.e., with more than two participants. Screen sharing is available in seven of the tested applications, including all business platforms, that we report in the bottom part of the table. Most applications have a desktop/PC version. Google Meet on a PC can be used only via browser, while FaceTime only works on Mac PCs and phones. Whatsapp has a desktop app which does not support calls. All applications have a mobile client and 9 out of 13 can be used directly via browsers supporting WebRTC.

We perform several experiments to collect representative packet traces for the chosen applications. Those which provide a desktop client are installed on three Windows testing machines. For the applications that support mobile clients, we perform additional experiments using an iPhone and for a few an Android phone. We also perform some experiments with Google Chrome to check the application behavior when used via browser. However, browser versions must use the WebRTC APIs, to limit the variability in terms of protocol usage and operation. Note that

all the tested applications provide either a mobile or a desktop client, and none of them can be used *uniquely* via browser. We also perform a few tests on the operating systems Linux and MacOS.

For each application, we make several experiments under different setups. We make calls with 2 and 3 participants (when allowed). We run individual experiments with only audio enabled, with both audio and video, and, finally, using also the screen sharing functionality, when available in the application. During each experiment, a participant collects all the traffic their machine exchanges with the Internet and stores it in `pcap` format. Each call lasts no less than 5 minutes. For each setup, we make a minimum of 5 calls. As such, we collect 20-30 packet traces for each application, summing to 334 in total. From the collected traces we identify the employed protocols and study the operation mechanisms. To achieve this, we first use Tstat [89], a passive meter which extracts rich flow-level records. It provides us entries for all the observed TCP and UDP flows, and, more importantly, it shows general statistics for each RTP stream, such as the number of packets, bitrate, etc. When no known protocol is found and in general for further analysis, we manually inspect the `pcap` files.

Table 3.2 Comparison of the RTC applications under test. Under *Redundant data*, "F" stands for FEC and "S" for Simulcast. Under *DNS domains*, "B" stands for easy to block, "C" for company-specific and "S" for social networks. Under *Other*, "N" means it uses less than four server-side ports and "T" means that PTs are used in a static fashion.

| Application | Protocols | | | | Operation | | | Identification | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | RTP | STUN/TURN | DTLS | Other | P2P | Redundant Data | Other | Own AS | DNS Domains | Other |
| Skype | ✓ | ✓ | | ✓ | ✓ | F,S | | ✓ | B | N,T |
| Google Meet | ✓ | ✓ | ✓ | | | S | ✓ | ✓ | C | N,T |
| Jitsi Meet | ✓ | ✓ | ✓ | | ✓ | | | | B | |
| WhatsApp | ✓ | ✓ | | | ✓ | F | | ✓ | B | N,T |
| Telegram | | ✓ | | ✓ | ✓ | | | ✓ | B | |
| Facebook Messenger | ✓ | ✓ | ✓ | | ✓ | | | ✓ | S | T |
| Instagram Messenger | ✓ | ✓ | | | | | | ✓ | S | N,T |
| Facetime | ✓ | ✓ | | | ✓ | | ✓ | ✓ | C | N,T |
| HouseParty | ✓ | ✓ | ✓ | | | | | | B | T |
| Microsoft Teams | ✓ | ✓ | | ✓ | ✓ | F,S | | ✓ | B | N,T |
| Webex Teams | ✓ | ✓ | | | | F,S | ✓ | ✓ | B | N |
| Zoom | ✓ | | | ✓ | ✓ | F | | | B | N,T |
| GoTo Meeting | | | | ✓ | | | | | B | N |

## 3.4 Network protocols

We start our analysis studying the protocols employed in the tested applications. Looking at the Tstat log files, we find the network flows carrying the media content. This operation is simple since our test machines do not run any concurrent task when making the calls. We notice that, in all cases, the applications opt for UDP as a transport protocol. We then analyze the payload of the media flows to identify the employed protocols. Indeed, a single UDP media flow may carry different protocols multiplexed together. This is always true in WebRTC, where RTP, RTCP, STUN and DTLS are sent over the same UDP flow.

In the left-most part of Table 3.2, called "Protocols", we report the protocols we find in the captures made with desktop/mobile clients. We intentionally neglect web clients, as they solely use the standard WebRTC APIs, resulting in the protocols mentioned above. We first notice that RTP is adopted in 11 out of 13 applications. We believe that all applications encrypt the payload using SRTP, but this does not alter the network behavior. STUN is commonly used to establish the session in the applications which use RTP. We notice that applications using STUN make use of TURN to communicate in case direct connection is not possible. This is no surprise as STUN and TURN are complementary protocols, orchestrated together in the ICE mechanism (see Section 3.2). We also find that four applications use DTLS, interleaved among RTP packets. Finally, we notice five applications that employ peculiar approaches, as described in the next paragraphs.

**Skype and Microsoft Teams:** the two services from Microsoft typically employ normal RTP to stream media and STUN to establish sessions. However, we find that in some cases they use a modified version of TURN called Multiplexed TURN[3], which is an encapsulation mechanism as simple as TURN, in which the ordinary RTP header follows a few header bytes. It can be easily identified looking at the first two bytes, always assuming a value of `0xFF10`. We create a simple command-line tool [90] to strip this header so that RTP is contained directly in UDP, allowing analysis with classical packet inspectors.

**Telegram:** we do not identify any known protocol within the UDP flow used to transport media data, only STUN and TURN for session setup. Indeed, the official

[3] https://docs.microsoft.com/en-us/openspecs/office_protocols/ms-turn/65f6ef76-a79d-42a4-a43f-dac56d4a19ac

Telegram documentation reports that the media track is encrypted and sent on the network via the proprietary MTProto protocol.[4] Note that Telegram calls are not available from the web client, where a custom protocol would not work.

**Zoom:** we find that the RTP header is not directly contained in the UDP payload, but a custom encapsulation mechanism accounts for 4 Bytes. Looking at the packet size and timing we conclude that in video streams the encapsulation Bytes assume the value `0x05100100`, while in audio streams, the value `0x050f0100`. We cannot find any explanation of this mechanism on the online documentation of Zoom, but in the command-line tool we created [90], we include a feature that strips the custom header of Zoom. Notice that these considerations hold only for the desktop and mobile clients of Zoom. The web client uses the standard WebRTC APIs, although very peculiarly. Indeed, it does not open any WebRTC media stream but only creates a data channel (WebRTC Data Channel), through which the media is transferred.

**GoTo Meeting:** as declared on the official website, it employs the Audio Video Transport Protocol (AVTP) for streaming multimedia content.[5] AVTP is a protocol alternative to RTP, which is part of the IEEE standard 1722-2011 [91]. Since AVTP is designed to run directly over Ethernet, GoTo Meeting uses an undocumented 4-Bytes encapsulation mechanism, for which we observe that the third and fourth Bytes are reserved to a 16-bit increasing sequence number. As reported on the documentation, the traffic is encrypted using the Advanced Encryption Standard (AES). Again, this happens only with the desktop and mobile clients, while the web client relies on WebRTC.

## 3.5 Operation and design choices

In this section, we analyze the operation of the tested RTC applications. We aim at understanding their design choices for streaming the multimedia content as well as peculiar uses of protocols, RTP above all. We summarize our main findings in the middle columns of Table 3.2, called “Operation”.

[4] https://core.telegram.org/mtproto/description
[5] https://blog.gotomeeting.com/gotomeeting-transports-protects-data/

### 3.5.1 Peer-to-peer

When a call involves only two participants, RTC applications often try to make them communicate directly, to avoid relaying the media traffic through a server. This has immediate advantages. First, the communication latency is always lower since the packets have a shorter distance to travel. Second, the application servers do not need to take the load of forwarding the media traffic. However, peer-to-peer is not always possible, since NATs, firewalls and middleboxes may prevent internal clients from receiving incoming traffic. Moreover, it works only with two-participant calls, since, otherwise, it would result in a full mesh of media streams among all participant pairs (see Figure 3.2). In our experiments, we want to spot the use of peer-to-peer communication, and, as such, we make calls with two participants using devices on the same LAN, where direct communication is always possible. Then, looking at the IP addresses of the RTP streams, we find peer-to-peer communication. Out of the 13 RTC applications, only 5 never use peer-to-peer, as we report in the fifth column of Table 3.2, called "P2P". This is somehow expected for business applications. Indeed, Webex Teams and GoTo Meeting offer to customers to install dedicated appliances on their premises, as advertised on their respective websites. Interestingly, three consumer services also never make use of peer-to-peer, loading their servers with the traffic of all calls. These are Google Meet, Instagram Messenger and HouseParty.

### 3.5.2 Redundant streams

To tackle the network unreliability, in some applications the participants' equipment sends redundant data, which can hopefully be used at the receiver in case of packet losses or errors. This approach is called Forward Error Correction (FEC) and is typically achieved exploiting simple mathematical properties – e.g., sending parity bits for the protected packets. Some codecs are designed to support FEC natively, and the current packet embeds redundant data of the previous packet. This mechanism is called in-band FEC and is implemented, e.g., in the Opus audio codec [92]. In other cases, the sender transmits FEC data on a separate channel, resulting in an additional and independent RTP stream. This is called out-of-band FEC, and it is used to achieve strong error correction capability and flexibility. In our experiments, we aim at finding the latter cases, which result in a client sending a higher number of RTP streams than expected – e.g., two output streams when only audio is enabled

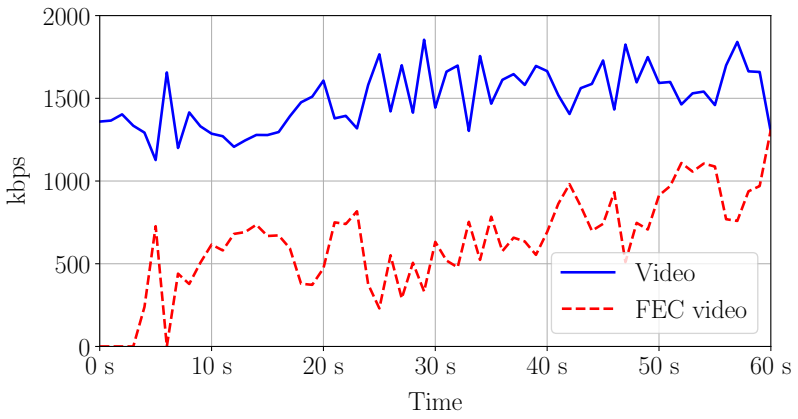



Fig. 3.3 Example of FEC in Webex Teams: a video stream and its corresponding FEC stream.

– or using multiple PTs within the same session. We find 5 services that make an evident use of out-of-band FEC, which we mark with *F* in the "Redundant data" column of Table 3.2. Skype and Microsoft Teams use video FEC with the H.264 codec, sending the data with different PT within the same RTP stream. Indeed, we observe PT 122 and 123, which indicate video and FEC video according to the online documentation.[6] Webex Teams sends audio and video FEC on separate RTP streams, in which the RTP Timestamp field is always set to 0. This can be confirmed by looking at the application logs stored on the user equipment for each call. We sketch an example video call with FEC in Figure 3.3. The figure only reports the video streams sent by a client to the relay server, and it is possible to observe how the FEC stream exhibits approximately half of the bitrate of the video stream. Similarly, WhatsApp sends two concurrent RTP streams containing video, both with low bitrate, in the order of $20-40$ kbps. They have a similar bitrate profile – i.e, they increase or decrease in bitrate simultaneously. One of the two has a lower bitrate, suggesting that it is used for FEC.[7] Finally, Zoom sends redundant audio data using the mechanism defined in the RFC 2198 [93]. For video, we observe that each stream carries a small but constant fraction of packets with a different PT, suggesting the use of a similar mechanism.

[6] https://docs.microsoft.com/en-us/openspecs/office_protocols/ms-rtp/3b8dc3c6-34b8-4827-9b38-3b00154f471c

[7] The two streams contain video since they have PT 102 and 103, respectively, and appear only in calls where video is enabled.

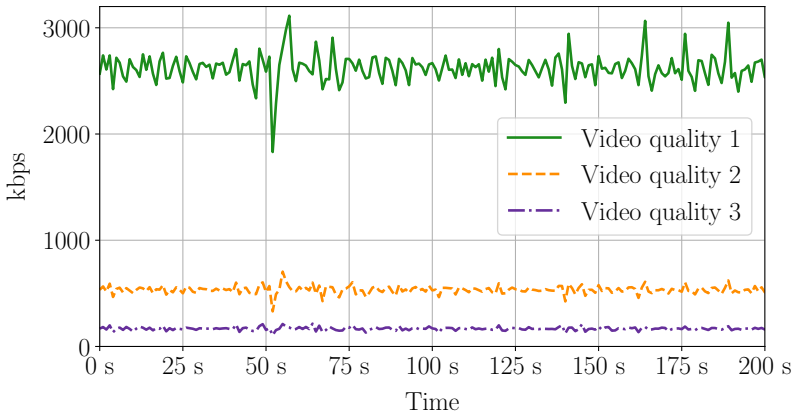



Fig. 3.4 Example of Simulcast in Google Meet: three video streams at different quality levels generated from one source.

A second use of redundant streams is the so called Simulcast technique that we indicate with *S* in the "Redundant data" column of Table 3.2. With Simulcast, the client encodes the video in different resolutions (and bitrates) and sends them as separate streams to a Selective Forwarding Unit that decides who receives which streams. This is useful in case some participants experience poor network conditions and can receive only low-bandwidth videos. We find that Google Meet uses Simulcast, and, when using the dedicated mobile application, the client (often) sends their video on three different bitrates, resulting in three separate RTP streams. This is exemplified in Figure 3.4, where we observe three video streams with different (yet constant) bitrates, that the client sends to the relay server. Then, each participant receives only one quality level, according to the server choice. We can confirm that these streams do not carry FEC but the same video at different definitions using the WebRTC debugging console of Google Chrome (at the receiver). Webex Teams also sends several streams in different qualities, mostly to account for the thumbnail videos of participants not speaking at the moment. We verify this using the logs generated by the application for every call. Microsoft Teams and Skype make use of Simulcast too, and we observe the user's video sent with up to three qualities at the same time. We exclude the possibility that that these streams contain FEC by looking at their PT.[8]

[8]See footnote 6 (Page 30).

### 3.5.3 Peculiar uses of RTP

Here we report two cases of peculiar uses of RTP. First, we notice that FaceTime uses regular RTP traffic, but employs PT numbers which are forbidden by the protocol standard [79]. In particular, we often observe $PT = 20$ which falls in the reserved range $20 - 24$. This peculiarity must be taken into account when using DPI to identify RTP traffic, if the filtering is done using allowed PTs. Indeed, a middlebox relying on the PT to make decisions on traffic would fall short for FaceTime.

We also observe that a few applications use the Contributing source (CSRC) optional header of RTP. Those are Webex Teams, Google Meet, Microsoft Teams and Skype. The objective of the CSRC is to enumerate the sources of a stream in case more than one are combined by a mixer. In Webex Teams, we notice that the CSRC uniquely identifies a participant of a call and, as such, can be used to isolate the streams of a particular user at network level.

Finally, Google Meet uses dedicated RTP streams for retransmitting lost data. We observe RTP streams that are active in short spikes and we confirm their nature using the WebRTC debugging console of Google Chrome (at the receiver), which reveals the MIME type to be `video/rtx`.

## 3.6 Identification of RTC applications

We now focus on the destination of the traffic generated by the RTC applications under test. We first investigate which Autonomous Systems they use to relay the media traffic (audio and video). Second, we discuss the domain names applications resolve via DNS during the normal execution. Third, we explore the UDP ports used during calls and provide other RTP-specific details like the usage of Payload Types. The goal of the first analysis is to show to what extent it is possible to use traffic management rules to prioritize RTC traffic. The goal of the second and third analysis is to provide useful guidelines for a network administrator willing to block specific RTC applications, because, e.g., not authorized within the enterprise.

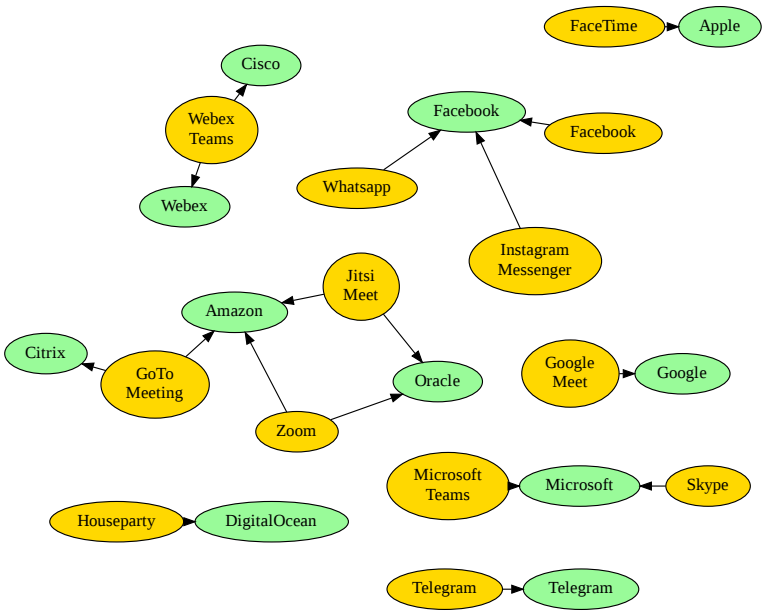



Fig. 3.5 Graph representation of the ASes (green) that RTC applications (yellow) use to relay RTC traffic.

### 3.6.1 Destination ASes

We first analyze the traffic of RTC applications in terms of destination AS. We focus solely on the media traffic, restricting our analysis to those UDP network flows carrying audio or video streams. We easily identify them for the majority of applications using RTP for media streaming. For Telegram and GoTo Meetings, which do not rely on RTP, we use a simple heuristic to find the correct flow. We then map an IP address to its corresponding AS using an updated Routing Information Base (RIB).[9]. We run the analysis only for calls with three participants, to ensure that peer-to-peer communication is not in place between two participants, which would warp our analysis. We report the results in Figure 3.5 in the form of a graph. Yellow nodes represent the 13 RTC applications, while green nodes are the ASes we find. There is an edge between an application and an AS if we observe at least one media flow between them.

The first thing we notice is that the majority of applications use ASes of their respective organizations for relaying media streams. For example, Google Meet

[9]The RIB can be found at http://www.routeviews.org/

uses the Google AS (numbered 15169), while FaceTime the Apple AS (numbered 714). Both Skype and Microsoft teams rely on the Microsoft 8075 AS. On the other hand, we find three applications that rely only on cloud providers for deploying their infrastructure. Indeed, Jitsi Meet and Zoom use both Amazon and Oracle cloud services, while HouseParty relies on the DigitalOcean cloud provider. Finally, there is Goto Meeting, which employs a hybrid approach and uses the Amazon infrastructure as well as servers located on the 16815 AS belonging to Citrix, the owner company. The applications which use ASes of their own organizations are marked with a tick in the "Own AS" column of Table 3.2.

Notice that this analysis is not exhaustive as it includes measurements collected from a single location in Italy. However, it gives useful indications for traffic management. Indeed, for many applications, it is easy to identify (and possibly prioritize) the media traffic. In other cases, they use large and shared infrastructures, requiring finer-grained identification mechanisms.

### 3.6.2 Contacted domains

In this section we discuss the use of domains that client applications contact during or before starting a call. The servers identified by such domains are used for signaling and accessory traffic – e.g., login or presence information. We provide this analysis with the goal of studying to what extent an ISP or a network administrator can block particular RTC applications (without compromising other allowed services).

Here, we divide the RTC applications into roughly three categories, that we report in the column "DNS Domains" of Table 3.2. First, there are the applications which can be reasonably blocked without impairing other services. We indicate them with "B". Second, we find applications that can be blocked but also include non-RTC functionalities. This is the case of social networks, that we indicate by "S". Finally, there are a few services whose block would prevent the operation of different services from the same company, that we indicate with "C".

In the first category we have the majority of RTC applications. This includes Skype, Jitsi Meet, Whatsapp, Telegram, HouseParty and the four business oriented services. They contact meaningful domains – e.g., `skype.com` in case of Skype, `zoom.us` for Zoom, `teams.microsoft.com` for Microsoft Teams and `wbx2.com` for Webex Teams – these can be used to totally block the application. Notice that

in such a case, no functionality would work, but we observe this set of applications only offer RTC or RTC-related features (e.g., chat). We find that applications mostly used by mobile devices have domains resolved long before the call, since they run continuously in background. Notable examples are WhatsApp and Telegram. We notice they contact only trivial domains like `whatsapp.com` or `telegram.org` which are easy to block, but would prevent also the chat functionalities of the products.

In the second category, we place the applications for which the RTC functionality is only a secondary feature of a rich service. This is the case of social networks. Instagram uses `*.instagram.com` sub-domains and Facebook uses `*.facebook.com`. This means that they are easy to identify in general, but blocking these domains would block all functionalities of the social network. Indeed, we cannot find domains related uniquely to the RTC features.

Finally, we find two applications which are particularly hard to block, as they are part of a large ecosystem of services. Google Meet, when accessed via browser, is contacted at `meet.google.com`, while via application only resolves generic Google domains. Particularly hard is the case of FaceTime, which only uses generic Apple domains, which, if blocked, would reasonably compromise the use of the iOS operating system for, e.g., software updates.

Notice that the identification of the applications by simply looking at the domain names is not an easy task. Thus, in Chapter 6, we propose a classification mechanism to distinguish RTC applications, based on machine learning.

### 3.6.3 Ports Numbers and Payload Types

We now discuss to what extent other features of the RTP protocol can be used to further understand the traffic. We investigate whether server-side port numbers can be of use to identify an application and how Payload Types can help a finer-grained classification.

Even though traffic classification using port numbers is becoming obsolete, we observe that in the case of RTC traffic, static UDP port numbers are still heavily used and they could be leveraged for effective traffic management mechanisms. Indeed, six of the tested applications always use a single UDP port, and three make use of 2, 3 or 4 unique port numbers. This makes them easily distinguishable among other

UDP traffic. These applications are marked with "N" in the last column of Table 3.2. For the remaining applications, we observe more than 4 ports in use. An interesting case is HouseParty, which uses a wide range of ports (we observed a different port for each traffic trace). Finally, we note that all applications use the allowed unprivileged UDP ports (greater than 1024).

If a network device, like a router, can identify RTP streams, then it can use the Payload Type values to distinguish different types of media, in general audio and video. The RTP protocol [79] defines a set of PTs to be allocated dynamically during the call setup phase, in addition to a set of static PTs whose usage is mandated by the RFC itself (see Section 3.2). In the RTC applications we study, we observe only usage of dynamic PTs, except for Skype, which sometimes uses static PTs for audio.[10] However, some applications use dynamic PTs in a static fashion and allow for distinction of audio, video, FEC or other types of streams, by assigning PTs to media types. From our analysis, although all applications use PTs from the dynamic range, 9 out of 13 always use the same values. Some of them are officially published, like those of Microsoft Teams[11], while others can be easily found by making calls with only audio or video enabled and observing the PTs. Applications that we find use constant PTs are marked with "T" in the last column of Table 3.2. However, even the static PTs may easily be subject to change in the future.

In conclusion, an algorithm that relies on a carefully-engineered combination of ASes, domain names, ports and payload types could lead to simple, yet effective RTC traffic management.

## 3.7 Takeaways

The purpose of this chapter is to introduce RTC applications, since they are the topic under scrutiny of the whole thesis. Here, we outlined the main technologies and protocols used by the most popular RTC applications on the market today. We hope it helped the reader get acquainted to the technologies.

We dived into the similarities and differences in the use of the network and the operation of 13 popular RTC applications. We found that most of them use the

[10] FaceTime uses the reserved PT 20 as we discuss in Section 3.4.

[11] See footnote 6 (Page 30).

RTP protocol STUN/TURN, but each has its own peculiarities. From the operation perspective, most of them use peer-to-peer communication between participants when the network allows it, and some of them use redundant streams for better QoE (simulcast) or for mitigating losses (FEC). Most of them are simple to identify, by looking at the destination Autonomous systems of the traffic, domains resolved via DNS, port numbers and Payload Types - all visible in the packet headers.

We believe the research presented in this Chapter could be useful to network administrators in improving RTC traffic management. It could also be important in corporate scenarios, in which, for instance, accredited services must be prioritized and the other segregated. In the context of this thesis, it was vital for the development of Chapters 5, 6 and 7. First, it helped us find ways to distinguish RTP traffic of various applications in a traffic mix and second, it helped us better understand the intricacies of RTC and thus engineer features that could be useful in an ML algorithm.

The reader should keep in mind that the research for this chapter was carried out in the year 2020 and so some details may be subject to change, if the applications change the way they operate. The applications and technologies are evolving fast, however, we believe that the basic premises, such as using RTP and heading towards WebRTC as a standard are here to stay for at least a decade.

# Chapter 4

# System deployment scenarios

In this chapter we describe a possible deployment of the in-network monitoring and control system described on Figure 1.1, that could make use of the classification modules described in Chapter 6 and Chapter 7, powered by the traffic statistics collected via *Retina* (Chapter 5). Indeed, distinguishing the RTC application being used (Webex, Jitsi, Zoom etc.) and the media type (audio, video, sceen sharing) at line rate could directly lead to network management policies that improve the QoE of the application users.

Knowing the underlying RTC application at the network layer allows the network to allocate appropriate resources and make decisions based on the specific application requirements (data load, bit-rate, latency, etc.). For example, we know that some applications are more bandwidth-heavy, e.g. Jitsi requires more bandwidth than Webex (Figure 7.1), Zoom requires very little bandwidth even for high-quality video [25] and Google Meet requires a large amount of bandwidth [33]. Thus, the network can decide how much bandwidth to allocate to different RTP flows if it knows what is the underlying application. Similarly, if the network is aware of the kind of streams being sent (audio, video, screen sharing), it can prioritize more important media. For example, in the event of worsening network conditions on a video call, it is more important to keep good quality of the audio, than it is to salvage the video. Indeed, it has been proven in the literature that, when presented with a good audio and several different degraded versions of video, users perceive sufficent QoE [94, 13].

When deploying Machine Learning in an operational network, there are practical issues to be taken into account, as outlined by [95]: computation and energy resources are limited, so there is a trade-off to be considered between the model accuracy and its computational complexity. Thus, for both classification tasks we choose lightweight models (see Sections 6.4.1 and 7.4.1). The system should work online, at line rate, taking packets as input, computing statistics, applying a pre-learned model and outputing a result. Moreover, since there is no performance guarantee in the wild, the system should consider fault tolerance and not degrade the performance of the network. When designing the system, we consider all these requirements.

We outline two possible deployment scenarios: (i) on edge network equipment, i.e. programmable switches or routers at the network edge and (ii) on a controller in a Software-defined network (SDN).

## 4.1 Deployment on edge network equipment

Recently, there have been various efforts to apply machine learning inference on the data plane. Emerging programmable switches, such as Intel Tofino[1], coupled with the P4 programmable match-action tables (MATs) now have the possibility to execute more complex operations and ML models directly in the network at line rate. For example, authors of [96] adopt packet classification with P4 on Tofino, using both supervised and unsupervised algorithms, among which Decision Trees, SVM and Naïve Bayes, all algorithms that we suggest for our classification problems. Decision Trees have been successfully implemented on a P4 switch by [97] as well, for classifying flows in data center networks.

Emerging data plane platforms, such as Taurus [98], have gone a step further and implemented per-packet ML inference for complex models such as Deep Neural Networks, at line rate as well. There have also been efforts to ease the implementation of ML modules on switches, such as Homunculus [99], a framework that automatically generates efficient data-plane ML pipelines for various use-cases and installs the models onto the underlying switching hardware, easing the work of network operators that are not ML experts. There are even efforts for reactive programmable switches, such as Mantis [100], that can react to current network conditions with

[1] https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

minimum latency and a lot of flexibility. Indeed, authors report a control plane that can react to changes in the network in 10s of microseconds.

In this climate, one can envision a deployment scenario for an RTC application classifier (Chapter 6) or a media type classifier (Chapter 7) on a programmable switch at the edge of the network. The network could be an enterprise network or a campus network. The switch's proximity to the end-user would enable it to take immediate actions in case of worsening network conditions, which would benefit the end user, network operator and the owner of the RTC application (e.g. Webex, Jitsi, Zoom).

A sketch of an example system for media type classification is shown on Figure 4.1. Here the idea is to do media-aware path selection. The edge switch runs the classification model and selects the path for each stream based on the media content they carry. In the example, audio packets are considered more critical and are routed to a *Golden* (reliable yet expensive) egress link, while the video is routed to a *Silver* (unreliable, yet cheap) path – e.g., a congested peering link.

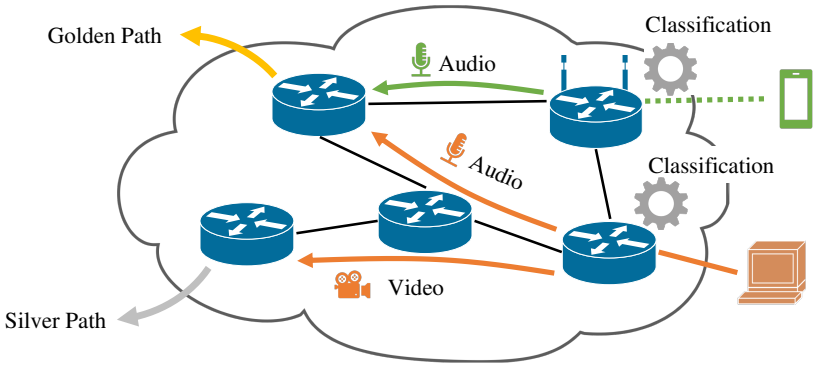



Fig. 4.1 Media-Aware Path Selection.

For the Application retrieval case, the switch would only have to observe packet headers from TLS Client Hello messages and extract the domain name, as this is the only necessary information for the classifier (see Section 6.2). That would greatly ease the switch's task.

Other implementations scenarios can include annotating packets based on their class. For example, in a Multiprotocol Label Switching (MPLS) network, the ingress node can set the label according to the classification outcome. Another approach could be to set the DSCP IP header field (DiffServ).

## 4.2 Deployment with SDN

Software-defined Networking (SDN) is a paradigm that involves decoupling the data plane (packet forwarding) from the control plane (routing). The network resources are managed by a logically centralized Controller, which has a global view of the network and can be dynamically programmed. SDN is a naturally good ground for Machine learning algorithms mainly due to two reasons. First, ML algorithms are data-driven and the SDN controller, having a global view of the network, is able to collect network data, to be used for learning, at various vantage points [101]. Second, SDN is programmatically configured and managed by the network administrators, which means network policies based on the results of ML models are easily implemented. There have been many applications of ML algorithms suggested for SDN, as explored by Xie et. al. in their survey [102]. Authors of [101] suggest application-aware traffic classification and show how it can work with SDN on an enterprise network. Authors of [103] deploy traffic management policies with SDN - they use ML to distinguish elephant from mice flows at the edge of the network and then suggest an SDN controller that implements traffic flow optimization algorithms based on the classification result.

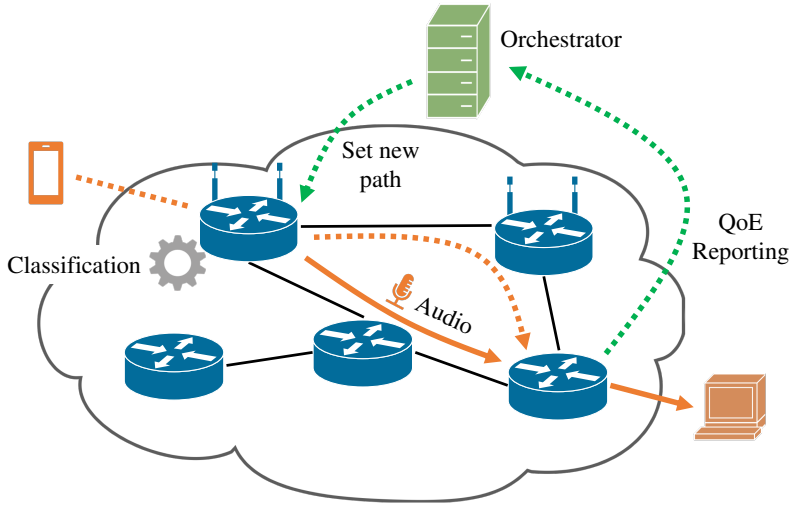



Fig. 4.2 Path Selection based on media type and QoE feedback.

The SDN controller can control the routing of traffic flows by modifying flow tables in switches. For example, it can guide switches to discard a traffic flow or route it through a specific path.

Figure 4.2 illustrates a deployment possibility of the media type classifier with SDN. Here, the classification module is a building block of a more complex RTC-aware traffic management system. A switch close to the source (left side of the image) runs the classification, while a switch close to the destination (right side of the image), reports QoE-related metrics to the SDN controller (Orchestrator). The Orchestrator can then do certain actions based on the measured QoE. Indeed, an SDN controller can control the routing of traffic flows by modifying flow tables in the switches. In Section 7.4.5, we show that the media type classifier can classify per-second traffic snapshots, but is also capable of classifying whole RTP flows with very high accuracy. Also, our Application retrieval classifier works per-RTP flow. Thus, if the SDN controller detects degradation in the measured QoE, it can select new paths for valuable RTC streams (e.g. audio), or allocate more bandwitdh to certain flows. The QoE reporting module can then be engineered in different ways, using well-known industrial standards such as Mean Opinion Score (MOS) [104, 105], using other mathematical QoS to QoE relationships [106, 107] or ML models [57, 108, 7].

## 4.3 Fault tolerance

Note that the envisioned scenarios are robust to possible flaws or delays in the underlying classification task. In both scenarios, deployment on edge network equipment and on an SDN controller, the system works by promoting streams to a better bath when it detects poor network conditions (congested link or degraded QoE).

There are generally two types of misclassification: (i) the error causes the system to respond unnecessarily – for example, we classify a video stream as an audio stream and promote it to a more reliable path. In this case, the system wastes resources unnecessarily. (ii) the error does not trigger a system response when it should have – e.g., we classify a stream that is actually an audio stream as video and do not promote it. In this case, the system would maintain the *status quo*, i.e., a "bad" QoE.

In Section 7.4, we report classification performance for the media type classifier of 96.3% and 95.3% for Webex and Jitsi, respectively. In this sense, an accuracy of 95-96% means that the system improves the QoE in 95-96% of the cases, while in 4-5% of the cases we maintain the status quo or we waste some resources. Although

undesirable, these situations do not entail severe impairment in QoE or in the whole system, provided they are sufficiently sporadic.

Moreover, in Section 7.4, we report a delay of 1 second for collecting statistics and a few milliseconds for computing features and running the classification. We believe a delay in system reaction in the order of 1 s is tolerable for video calls, since their lifetime is in the order of minutes or hours. Collecting information about a stream for 1 second allows the system to compute representative statistics about the stream, thus increasing the accuracy of the classifier. Further on in Section 7.4, we also show that it is possible to use our classifier at a reduced delay of 200ms, slightly sacrificing accuracy.

The same notions hold for the Application retrieval classifier (Chapter 6). Here we report an accuracy of 89% (Section 6.4.1). If the system classifies the incoming traffic as “Webex”, while it was “Zoom”, it may allocate more bandwidth to that flow, when it was not necessary - which is not a critical operation.

## 4.4 Takeaways

The goal of this Chapter is to give the reader a general idea of what a system that uses the approaches shown in this thesis may look like and how they can be useful in a production network. To this end, we outline two possible deployment scenarios: deployment on edge network equipment, or in an SDN scenario. The actual implementation of the whole system and its performance evaluation is out of the scope of this thesis.

# Chapter 5

# Retina: An open-source tool for flexible analysis of RTC traffic

The work presented in this chapter is mostly taken from our paper *Retina: An open-source tool for flexible analysis of RTC traffic*, published in *Computer Networks 202* [109]. *Retina* is a tool that transforms the data from packets (in *.pcap* format) to traffic statistics, that can be used as features for the Machine Learning algorithms. We use it to create features and ground truth to predict the media type of RTC traffic in Chapter 7. However, *Retina* is a versatile tool that outputs various logs and graphs and parses specific application logs. Thus, it can be used for general analysis and diagnostics of RTC traffic.

## 5.1 Introduction

*Retina* is an easy-to-use command-line tool that extracts advanced network statistics for RTC sessions found in packet captures. It also generates graphical output with various charts and visualizations of the statistics for easy analysis. *Retina* focuses on the Real-Time Protocol (RTP) [79] protocol used in most RTC applications [77], with its encrypted version SRTP (SRTP leaves packet headers unencrypted). *Retina* goes deeper than general tools in understanding RTC traffic. Starting from a capture, *Retina* searches for RTC traffic, identifies streams and outputs more than 130 statistics on packet characteristics, such as timing and size, and tracks the evolution of the stream over time bins of a chosen duration. It is highly configurable, and the user

can customize the output statistics as well as a number of other parameters. *Retina* can enrich its output by merging the information available in the RTC application logs to provide the ground truth required for many classification problems.

*Retina* is open-source and available to the research community and network practitioners.[1] We believe it can be useful for traffic monitoring, and we have successfully used it for data processing and feature extraction to feed Machine Learning (ML) algorithms in the context of RTC-aware network management.

Several tools already perform in-depth traffic analysis, and packet dissectors such as *Wireshark*[2] (and its command-line version *Tshark*) are the first resources for network troubleshooting. Flow monitoring is also commonly used to analyze traffic summaries [110], and NetFlow [111] is the de facto standard for collecting and processing flow records. Sophisticated network meters also expose application-level statistics using Deep-Packet Inspection on Layer-7 protocols. Tstat [112], for example, provides global statistics on RTP streams, while nProbe [113] offers a VoIP plugin as a closed-source commercial product. In contrast to these works, *Retina* provides comprehensive statistics both per time unit and per flow. It specializes in RTC traffic and detects numerous RTC applications, including some that modify the RTP protocol. It also offers a wide range of parameters for personalized log creation.

## 5.2 System overview

In this section, we describe *Retina*'s operation. As input, *Retina* takes one or more packet captures as well as optional configuration parameters. It processes the traffic and outputs the desired output in various forms. Figure 5.1 depicts its overall architecture. *Retina* is written in Python and depends on *Tshark* and a number of modules that can be installed via the package manager `pip`. We also provide a dockerized version to allow the use as a standalone container.[3]

[1] https://github.com/GianlucaPoliTo/Retina
[2] https://www.wireshark.org/
[3] The dockerized version is available at: https://hub.docker.com/r/gianlucapolito/retina

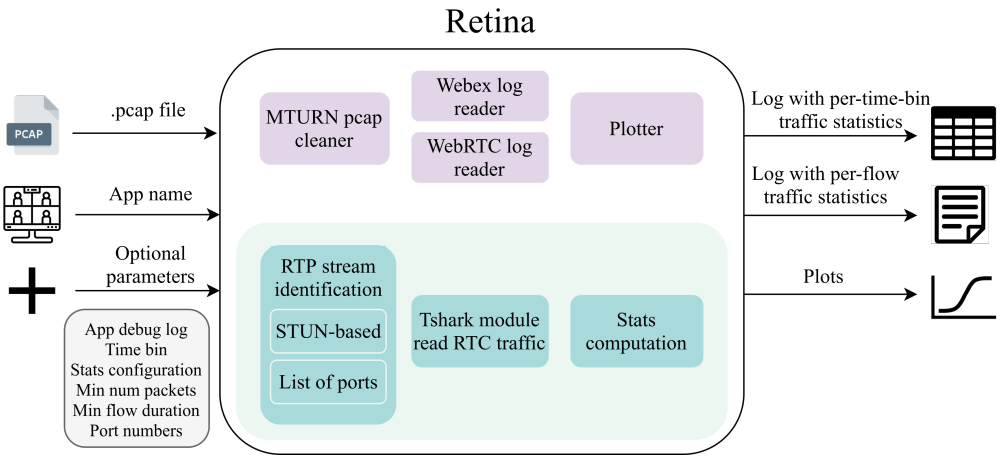



Fig. 5.1 *Retina* architecture.

### 5.2.1 Inputs and configuration

*Retina* requires the user to specify one or more captures in PCAP format, the most common format used in many traffic capture software (*Wireshark*, *TCPdump*, etc.). *Retina* can also process an entire directory by searching for all captures in it. If it finds more than one, *Retina* uses multiprocessing to process multiple files at once. The number of processes is a configurable parameter.

For some RTC applications, the user can provide application log files that *Retina* uses to calculate additional statistics and enrich the output. The application logs typically contain details about the media sessions, including the Source Identifiers of the RTP streams, the type of media (audio, video, or screen sharing), the video resolution, the number of frames per second, etc. When available, *Retina* uses this additional information to provide finer-grained per-second statistics – e.g., media type, video resolution or concealment events at the codec level. Currently, *Retina* supports log files of: (i) Cisco Webex[4], which logs second-by-second details for each RTP stream, and (ii) Google Chrome, by collecting WebRTC debugging logs with WebRTC browser-based RTC services.[5]. This way we can download logs of each application used through Google Chrome (Google Meet, Jitsi etc.).

In *Retina*, the user can customize a variety of parameters. All are optional, with carefully set default values. *Retina* has personalized features for many RTC

[4] https://www.webex.com/

[5] These logs can be obtained by creating and downloading a dump at `chrome://webrtc-internals`

applications, which can be enabled by specifying the name of the RTC application whose traffic is included in the capture as an input parameter. While it supports all applications that use RTP at their core, we have tested it extensively for Webex, Jitsi, Zoom, and Microsoft Teams. *Retina* accepts threshold parameters, such as the minimum number of packets or the minimum duration of a stream for it to be considered valid. The user can also control the statistics computed at each time bin (see Section 5.2.3) and can ask *Retina* to create (interactive) graphs. The full list of parameters can be found in the documentation, while in the rest of the chapter we will only mention the most important ones.

### 5.2.2 System core

The overall architecture of *Retina* is shown on Figure 5.1, with the middle rectangle indicating the building blocks at its core. We depict the basic functionalities in blue, at the bottom, and the optional modules in purple, at the top. We also show a sample command line at the top of Table 5.1.

The basic functionalities of *Retina* analyze the raw packets contained in the input PCAP captures and gather statistics, organized in tables per stream and per time-bin. For example, consider a PCAP capture collected at a user side, containing RTP traffic from a two-party call consisting of 4 RTP streams (outgoing and incoming audio and video). Setting a time bin duration of 1 second, *Retina* maintains a table where, for each of the 4 streams and for each second, it accumulates several statistics. Given a packet characteristic, such as packet size or interarrival time, *Retina* calculates several statistical indicators, such as mean, median, third and fourth moments, or percentiles. We report the list of packet features and available statistics in Figure 5.2, which summarizes the whole process of statistics extraction. The user can configure the duration of the time bin for this aggregation of packets, which is 1 s by default. The duration of the time bin directly affects the number of packets used to compute the statistics, and should therefore be varied judiciously. For example, in 1 s of audio, 50 packets are sent, while, in 1 s of HD video, more than 200. Clearly, if the time window is 200 ms for audio, no significant features can be computed, while this time window would be fine for video.

To identify RTP streams in traffic, *Retina* internally relies on *Tshark*, the command-line version of *Wireshark*. This step is not straightforward, as RTP packets

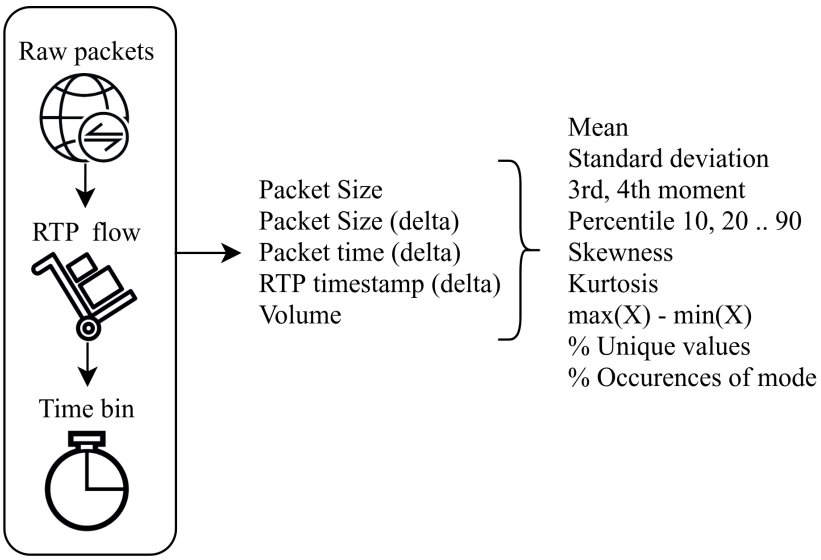



Fig. 5.2 Aggregation process and some of the statistics computed by *Retina*.

often appear in a UDP flow along with other protocols. In fact, many applications use STUN [81] to establish the media session and/or TURN [82] to relay the streams if no direct connection between peers is possible (see Section 3.2). In addition, it is common to use DTLS [87] interleaved among RTP packets to exchange control information such as encryption keys. *Retina* supports two methods for identifying RTP streams: (i) with a user-defined list of ports or (ii) by examining the STUN-initiated UDP flows. *Retina* attempts to decode the UDP payload as RTP and verifies that the protocol headers are compatible with RTP. We define an RTP stream using the combination of IP addresses and ports (the classic *tuple*) plus the RTP Synchronization Source Identifier (SSRC), which is used to multiplex multiple streams within a single UDP flow. For some RTC applications, we also use the RTP Payload Type (an RTP field that specifies the media codec). *Retina* maintains internal data structures to efficiently collect statistics for each RTP stream. This way to identify RTP streams in a traffic mix is used throughout the thesis.

*Retina* has a number of optional modules that target RTC applications, for which we have implemented special support. First, the traffic of some popular RTC applications (Zoom and Microsoft Teams) needs to be preprocessed to become standard RTP traffic. This is because they use the RTP protocol in a non-standard form. Microsoft Teams encapsulates RTP in a proprietary version of TURN called MTURN, while Zoom adds its own undocumented header. To make *Retina* work for

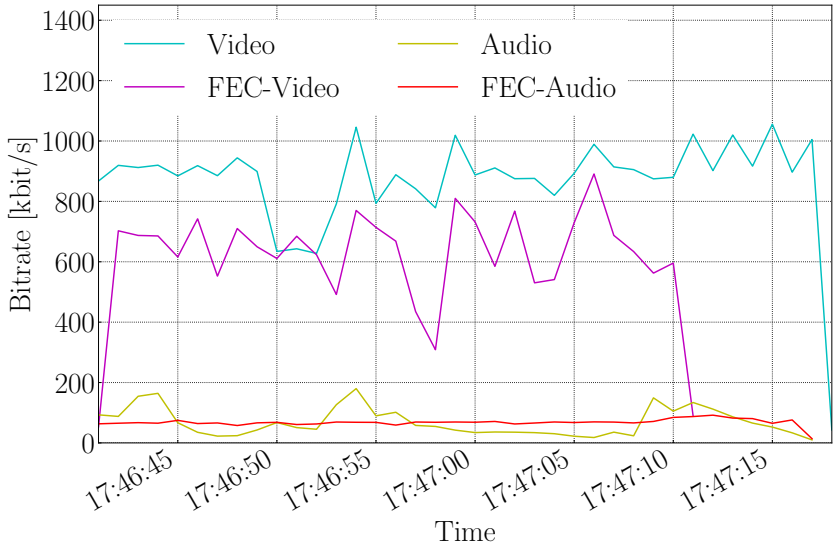



Fig. 5.3 Example plot of the stream bitrate in a call.

these RTC applications, we have created specific modules that can also be used as standalone command line tools. They can be found in a separate folder in the code repository.

Second, *Retina* can read and process the application log of (i) Webex and (ii) Google Chrome, as mentioned in Section 5.2.1. *Retina* can parse these logs and provide additional information about the RTP flows. If the application logs are available, we enrich the output logs from *Retina* with information such as the video resolution, employed codec, frames per second, jitter, codec concealment events, etc. We also provide a classification of media types into 7 classes, such as audio, FEC streams, 3 different qualities of video and screen sharing, for easier recognition. The information in the application logs is particularly useful for training supervised ML models, as it contains the necessary ground truth for many QoE-related problems, such as number of losses, smoothness of the video, concealment etc. *Retina* matches the timings of the logs with the timings of the packets exactly, so it outputs a labelled dataset.

Lastly, *Retina* includes a plotting engine based on the Matplotlib and Plotly libraries[6] to create both static and responsive graphs of all RTP streams. It draws the time-series of stream characteristics, such as bitrate or inter-arrival time, so that the user can easily get an overview of the traffic or debug an RTC application. It

[6]Matplotlib: https://matplotlib.org/, Plotly: https://plotly.com/

Command: `./Retina.py -d capture.pcap -so webex -log webex.log`

| Timestamp | Packet size (mean) | Packet size (std dev) | Bitrate (kbit/s) | Interarrival (max) | Packets/s | Frame width | Frame height | Frames/s |
|---|---|---|---|---|---|---|---|---|
| 2021-06-08 14:32:11 | 1041.84 | 66.74 | 1163.93 | 0.043 | 143 | 480 | 270 | 30 |
| 2021-06-08 14:32:12 | 1080.72 | 100.75 | 1578.86 | 0.045 | 187 | 640 | 360 | 30 |
| 2021-06-08 14:32:13 | 1023.49 | 72.21 | 1023.49 | 0.045 | 128 | 640 | 360 | 30 |
| 2021-06-08 14:32:14 | 1076.80 | 52.91 | 1362.82 | 0.043 | 162 | 640 | 360 | 30 |
| 2021-06-08 14:32:15 | 1055.50 | 52.41 | 1410.08 | 0.044 | 171 | 640 | 360 | 30 |
| 2021-06-08 14:32:16 | 1074.62 | 62.71 | 1989.73 | 0.089 | 237 | 640 | 360 | 30 |
| 2021-06-08 14:32:17 | 1055.22 | 40.09 | 2588.59 | 0.033 | 314 | 640 | 360 | 30 |
| 2021-06-08 14:32:18 | 1057.73 | 51.67 | 1479.17 | 0.040 | 179 | 640 | 360 | 30 |

Table 5.1 Example command and *Retina* log for an RTC stream. The last three columns are derived from the application logs.

also draws several histograms for each stream to show the stream-wise distribution of packet characteristics (e.g. packet size). For an example graph, see Figure 5.3. Here we show the bitrate of 4 RTP streams present in a portion of a Webex call. The plotting engine also labels the time-series with their media type (audio, video, FEC etc.), if the information is provided (e.g. through an application log file).

### 5.2.3 Outputs

*Retina* produces a CSV file for each RTP stream found in the input capture, reporting the selected statistical features for each time bin. The logs contain different columns according to user preferences and additional stream information if the RTC application log is provided. We show an example output log in Table 5.1, along with the command line used to create it. Optionally, *Retina* creates a summary log file in which it reports stream-wise statistics. The file contains the most important information for each stream – i.e., the source and destination IP addresses and ports as well as general statistics such as the number of packets, duration, etc. Having per-stream information is useful for many applications that rely the analysis of flow/stream records for e.g., traffic accounting. Additionally, *Retina* provides a rich set of graphs that describe the traffic, which are discussed in Section 5.2.2.

Finally, *Retina* also provides a dashboard for analyzing RTC traffic through an interactive interface.[7] The dashboard requires an input `.pickle` file, which can

[7]An online demonstrator of the dashboard is available at: https://share.streamlit.io/gianlucapolito/retina-dashboard/main/dashboard.py

be produced by passing one or more packet captures to *Retina* and specifying an argument for the plot. Here the user can see interactive plots of stream statistics and compare streams of interest.

## 5.3 System design assets

*Retina* is designed following principles of scalability and modularity, so that it can be easily extended. It adopts a multiprocessing architecture, so when there are multiple PCAP files to process, it uses an independent process for each of them and stores separate output log files. These files can then be merged at the end of the processing. This also increases the robustness of the tool.

*Retina* is highly modular, with separate functions organized into logical modules for all the different operations. This also allows for extensibility, as a user can write new functionalities with minimal effort. For example, it is easy to support the application log of a new RTC application (e.g. Microsoft Teams), as it is only necessary to add a parser function and call it with an argument.

*Retina* can be used to analyze any kind of RTP traffic, and it is not limited to video conference applications. For example, we have successfully used *Retina* to gain insights into the operation of cloud gaming applications running over the browser [114]. Similarly, our parser for the Chrome WebRTC log works seamlessly for any type of browser-based application.

Finally, *Retina*, as described in Section 5.2.1, is highly configurable. The user can limit the statistics to be computed (potentially speeding up the computation), the desired time aggregation, and several internal parameters - e.g., the minimum length of an RTP stream for it to be considered - which are detailed in the README file.

## 5.4 Publications enabled by the software

*Retina* was first developed at the end of 2019, and within 3 years of its existence, it has already been a valuable asset for 6 scientific publications that target RTC traffic. *Retina* sits at the core of [115] and [116], described in Chapter 7. There, we used it to engineer features and extract the ground truth for an ML classifier that

distinguishes media types. We further built on it in [117], or Chapter 6, to do data pre-processing and identify RTC streams in traffic. It served for data characterization in [77] (Chapter 3), where we analyze the operation of 13 different RTC applications. It has been also used in [7], to engineer features for an ML classifier that predicts the presence of losses in the near future. Moreover, it has successfully been employed to study cloud gaming traffic, allowing the authors to understand the networking operation behind Google Stadia, GeForce NOW and PSNow in [114].

## 5.5 Limitations and Future work

While *Retina* supports most RTC applications, it still does not support those that do not use RTP (or a modified version of it), like *GoToMeeting* or *Telegram*. Moreover, it relies on the RTP headers, so if in a future protocol version these are encrypted, the tool will need major revisions.

As future work, we aim to make *Retina* work in real-time and be able to support traffic at high speeds (e.g. 40 Gb/s links). We would also like to introduce better support for gaming traffic, cover different cloud gaming platforms, and output more gaming-specific ML features. Our research group also plans to support *Retina* in the long run and follow the future developments of the underlying protocols such as RTP, STUN, and TURN, as well as tackle new protocols from novel providers.

## 5.6 Takeaways

This chapter presented *Retina*, a flexible command-line tool for extracting advanced statistics from network traffic of RTC applications. It provided a schematic description of all its features: the inputs, the system core and the outputs with examples. It also highlighted the design strengths of *Retina*, its modularity, scalability and configurability.

The contributions of this chapter are:

- In the context of the thesis, it describes the process all RTC traffic should go through in order to use the proposed traffic classification system of Chapter 7. All RTC packets should go through *Retina* first.

- To help both the scientific community in studying RTC applications and network administrators in troubleshooting RTC traffic.
- To serve as a feature construction engine and provider of ground truth for ML-based downstream tasks related to RTC traffic visibility and QoE improvement.

We believe that the networking community still lacks modern easy-to-use tools and datasets for fast prototyping of algorithms and that can be used as benchmarks. By releasing this software, we hope to bridge a little bit of this gap.

# Chapter 6

# RTC Application Retrieval

The work presented in this Chapter is mostly taken from our paper *What's my app? ML-based classification of RTC applications*, published in *ACM SIGMETRICS Performance Evaluation Review 48* [117]. It describes an ML classifier that distinguishes between five different RTC video-conferencing applications, with the goal of observability and ultimately QoE improvement of RTC traffic.

## 6.1 Introduction

In this chapter, we propose a novel methodology to unveil the applications behind RTC traffic. We base the approach on the domain names (Fully Qualified Domain Name - FQDN) applications contact prior to set up a call, that we use as a signature to classify RTC streams in live traffic. We employ Natural Language Processing (NLP) techniques to model how these domains appear in the traffic and use them to extract meaningful features that we then feed to Machine Learning (ML) classifiers. Exploiting a large dataset that contains more than 230 packet traces, we evaluate the performance of the methodology in distinguishing the traffic between 5 RTC applications. Results show that it is possible to obtain good performance, reaching an F1 score as high as 0.89. Moreover, our approach is based solely on the domains clients resolve *prior* to start a video call, allowing the system to classify new streams with virtually zero delay and possibly adapt to the required network conditions of the application. Indeed, the system does not need any feature coming from the RTC stream itself, making it robust to, e.g., applications resorting to peer-to-peer

communication between clients or applications that modify the RTP protocol and are harder to distinguish in a mix of traffic (e.g. Zoom).

Our code and dataset are available online [1]. We hope they can be useful to other researchers to reproduce our results or use them in other approaches.

The rest of the Chapter is organized as follows: In Section 6.2, we describe the system architecture - from packet traces to extracting domains and learning. Section 6.3 describes the data we use, Section 6.4 shows the experimental results and Section 6.5 concludes the chapter.

## 6.2 System architecture

The goal is to identify the meeting software that is behind an RTC stream when observing live network traffic. In other words, whenever we find a client starting an RTC session, we want to classify it as *generated* by, e.g., application *A*. For us, an *RTC session* is a media stream of an RTC application, carried over the RTP protocol.

Rather than extracting features directly from the RTC streams or leveraging the server-side IP address, we target the domains the client resolves *prior* to start the session. The reasons we opt for this approach are two-fold. First, nowadays, large companies rely on Content Delivery Networks to serve their content and on Cloud Providers to host their infrastructure (see Chapter 3), making simple approaches based on the enumeration of the server IP addresses or ranges ineffective. Second, RTC applications are known to rely on peer-to-peer communication between participants when possible. This again makes it difficult to leverage IP address and ports numbers, typically randomized by NATs. Also, the behavioral features of the streams are subject to extreme variability due to the diverse possible network conditions, discouraging their use for classification.

As such, we feed the classifier the domains resolved by the client before starting the RTC session. In this way, we target the control traffic of the application, that is typically exchanged over the TLS protocol. More specifically, we extract the Server Name Indication (SNI) contained in the Client Hello messages, that contain the domain name of the server in-clear. That said, whenever we encounter an RTC

[1] https://github.com/denama/RTC_apps_classifier

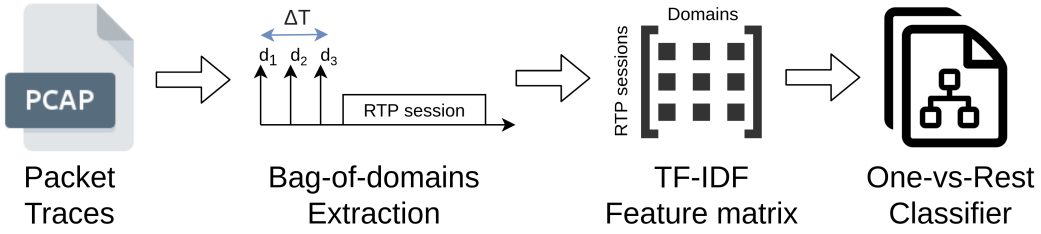



Fig. 6.1 Scheme of our training methodology.

stream in the traffic, we collect the *bag* of domains the client resolved in the previous $\Delta T$ seconds.

**Training Methodology.** We train the system on a collection of packet traces, each containing a single RTC call using a known application – i.e., the ground truth. In Figure 6.1, we show an overview of the training steps. We collect all domains contacted by the client in the $\Delta T$ seconds before the call begins. Thus, we have a number of domain names per RTC call. Then, we prepare the dataset to be used to train an ML-classifier – i.e., we *vectorize* the data. This means we turn the textual domains into numeric features to form a feature matrix. In this feature matrix, every domain name seen in all the calls is a column and every RTC call is a row. We fill the table by marking, for every RTC call (row), how many times each domain name (column) has appeared in the $\Delta T$ seconds before the call began. We call the value of each cell $v$. Obviously, this table is sparse - it has a lot of $v = 0$ values, since in one call, only few of the domains have been contacted.

Our goal is to find, for each application, the domains that are more useful for its classification, because they are used, for example, for signaling, session setup, login, etc. To this end, we rely on NLP techniques, which have been proved to be useful to find the terms that better characterize each document within a corpus. In our system, we opt for the $tf-idf$ analysis [118], which is traditionally used to describe *documents* in a collection with the most representative terms. Given a particular term and a document, the $tf-idf$ is computed as the product of the frequency of the term in the given document ($tf$) and the inverse of the frequency at which the term appears in distinct documents ($idf$). Here $tf$ estimates how well the given term describes the document and $idf$ captures the term's capacity to discriminate the document from others. In the context of our problem, we use $tf-idf$ to identify the domains that better characterize an RTC application. The intuition is that if a domain appears in the majority of the traces, it is less important than a domain appearing in only a few

of them. Using the $tf-idf$ paradigm, we process the already built feature matrix so that each cell value $v$ is replaced by the results of $tf-idf$ on the dataset.

To help the $tf-idf$ get rid of noise, we apply a threshold that each domain must reach to be included. We want to avoid very rare domains, that appear in just one or a few traces, but obtain high $tf-idf$ scores due to a high $idf$ (because they are infrequent) and a high $tf$ (because they have appeared many times in a trace). To this end, we set a threshold $MinFreq$, below which a domain is discarded. If a domain appears in a fraction of packet traces smaller than $MinFreq$, we flag it as noise and neglect it.

Once we obtain the final $tf-idf$ filtered feature matrix, we use ML classification to distinguish between the RTC applications. This institutes supervised learning, since the training set includes a ground truth, indicating for each row (a session), the employed RTC application. We use the one-vs.-rest strategy, which consists of fitting one binary classifier per class. We opt for this schema to improve robustness if classifiers are used in the wild, with potentially new unknown applications. By using one-vs.-rest, we force each classifier to focus on a single application, leaving all the rest in the *Other* class. This also improves the interpretability, since it allows us to gain knowledge about each class by looking at its specific classifier.

**RTC Stream Classification.** We use the obtained classifiers to identify the applications behind live network traffic. At classification time, we build the bag of domains whenever an RTC session begins. We then build the $tf-idf$ vector from the bag, which means, using the frequency at which domains have appeared as $tf$ and the values we computed at training time as $idf$. If a domain was not seen at training time, and, as such, we have no $idf$ value, it is discarded. On the obtained feature vector, we run the previously trained classification models. We try five different algorithms: tree-based classifiers [Decision Tree (DT) and Random Forest (RF)], k-Nearest Neighbors (kNN), which classifies points based on proximity to other data points, Support Vector Machine (SVM), which instead constructs a hyperplane in high dimensional space to separate the data points and Gaussian Naïve Bayes (GNB) as a generative probability model. We then compare their performance under different conditions. Moreover, in Section 6.4.2, we explore the impact of the system parameters ($\Delta T$ and $MinFreq$ among all) on the classification performance. The system can easily make use of parallel processing, since it works on a per-client basis.

Table 6.1 Dataset overview.

| Application | Webex Teams | Microsoft Teams | Skype | Jitsi | Zoom |
|---|---|---|---|---|---|
| No. training | 27 | 47 | 35 | 35 | 30 |
| No. testing | 12 | 17 | 11 | 12 | 13 |

## 6.3 Dataset

The dataset we use for this analysis is collected by a set of 15 volunteers, which recorded packet traces whenever they made a call during the first six months of 2020. The volunteers begin capturing the traffic on their equipment *before* starting the RTC application, since this is vital to the analysis. In total, we collect 239 traffic traces from five meeting applications: **Webex Teams, Microsoft Teams, Skype, Jitsi and Zoom**. Note that it is hard to do automatic collection of such data, because it is not possible to rely on well-known browser automation tools like *Selenium* when using native applications running on PCs. Moreover, collecting data in the wild improves the robustness of the results, since the packet traces were collected under a diverse set of operating systems, user devices, application versions, etc. Out of 239 calls, we use 174 for training and 65 for testing. We explicitly use all the traces of a single individual either for training or for testing, to avoid overfitting to specific habits of a client. Due to the nature of the problem, one call translates to one data point for the classifiers. A breakdown of the dataset is provided in Table 6.1. Since we have to resort to a non-automated data collection, we consider to have a fair amount of data in terms of number of calls. More importantly, results show that the system reaches high accuracy even with this number of data points.

We process the packet traces using Tstat [112] to obtain the bag of domains the client contacted before starting the RTC stream. For all applications, it is straightforward to identify the media streams as they all rely on RTP. Only for Zoom, we had to strip the custom encapsulation header using our simple command-line tool [90], mentioned in Chapter 3.

## 6.4 Experimental results

In this section, we discuss the system performance in terms of its ability to classify the application generating RTC streams. We assess the performance using the test set composed only of packet traces collected by individuals that do not appear in the training set. We first outline the results of the best-performing classifier and then discuss the performance under different configuration parameters, such as classification algorithms, $\Delta T$ and $MinFreq$. As a performance metric to evaluate between different sets of parameters, we use the F1 score. The F1 Score is the harmonic mean between the Precision and Recall of a class.

### 6.4.1 Best classification performance

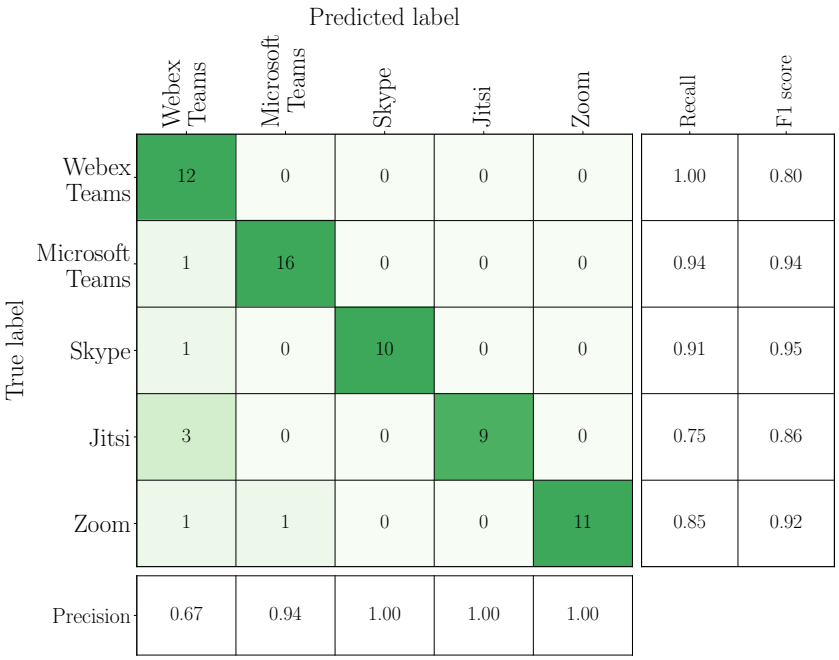



Fig. 6.2 Confusion matrix with the best parameter choice.

Figure 6.2 shows the confusion matrix obtained using the best configuration of parameters. This constitutes using a Random Forest algorithm, with $\Delta T = 25s$ and $MinFreq = 0.05$. By definition, a confusion matrix $C$ is such that $C_{i_j}$ is equal to the number of observations known to be in group $i$ and predicted to be in group $j$. The diagonal represents the number of correctly classified samples. We also show the

Table 6.2 Examples of highly discriminative domains.

| Application | Domains |
|---|---|
| Webex | ciscospark.com<br>webex.com |
| Microsoft Teams | area.microsoft.com<br>teams.microsoft.com |
| Skype | area.microsoft.com<br>skype.com |
| Jitsi | meet.jit.si |
| Zoom | zoom.us |

per-class recall and F1 score in the last two columns, as well as the precision in the bottom row. Looking at the figure, we observe that all classes exhibit an F1-score higher than 0.8 and 3 out of 5, a score higher than 0.9. Webex Teams is the only class with slightly lower precision (0.67), meaning the other applications are being confused with it, especially Jitsi. This is also why Jitsi exhibits a lower recall (0.75) than the other classes. The overall macro-averaged F1 score is 0.89. The macro average is the mean of the scores of each class. The Random Forest classifiers used to reach this score consist of as little as 100 trees and a maximum depth of 20.

Since we adopted the one-vs.-all classification schema, we can observe, for each application, which features (i.e., domains) are important for each class. Table 6.2 provides examples of the domains for which the Random Forest classifier indicates high feature importance. To ease visualization, we truncate the domains at the second level. We qualitatively note how the approach is able to identify the domains per application. For example, Webex relies on `webex.com` and `ciscospark.com`, which is the former name of the application. The domain name `area.microsoft.com` is seen very often for both Skype and Microsoft Teams, since they are both owned by Microsoft. It would not be trivial to distinguish Skype and Microsoft Teams with a heuristic approach. Another advantage of our approach is that we obtain *interpretable* results, especially if compared with other techniques based on packet-level features.

### 6.4.2 Parameter sensitivity

Here we illustrate the impact of system parameters and different algorithms on the classification performance. We first investigate the impact of $\Delta T$ – i.e., the duration of the period before the stream begins, that we use to collect the bag of domains.

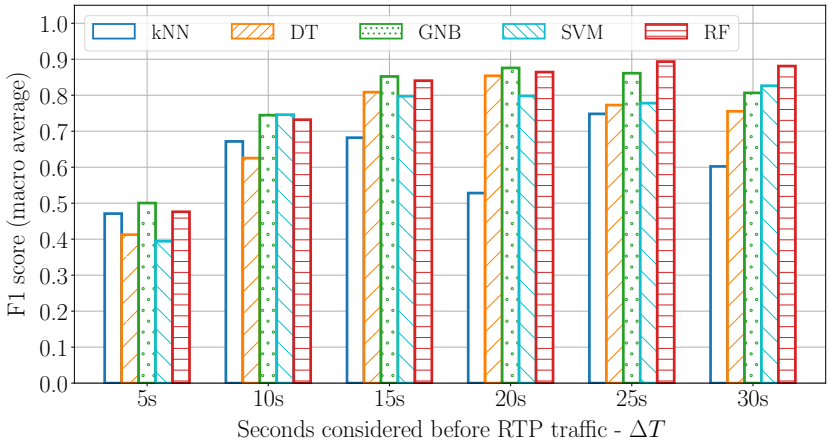



Fig. 6.3 Performance of the five algorithms for different time bins.

We vary it between 5s and 30s and show the results in Figure 6.3. We also use the figure to show the impact of different classification algorithms in terms of macro-averaged F1 score, represented on the $y$-axis. Note that all algorithms undergo hyperparameter tuning, using a 3-fold cross-validation on the training set, which leads to small improvements. In general, the larger $\Delta T$ is, the more domains we collect and the better the final performance is. Indeed, the F1-score for 15s and above is altogether higher than the one for 5s or 10s. The performance also varies with different classification algorithms. kNN shows generally worse performance for higher $\Delta T$, working better with a low number of features. DT and SVM achieve high F1 scores in general. GNB exhibits excellent results, showing the best performance of all for 15s or 20s, but the best result in absolute terms is achieved with RF, for $\Delta T = 25s$.

Next, we analyze the impact of the number of features, which are directly tuned by setting *MinFreq*. *MinFreq* is the fraction of traces in which a domain should be seen to consider it in the feature matrix. Intuitively, a large *MinFreq* allows a low number of domains to appear as columns in the feature matrix, while small values allow also infrequent domains to be considered. We vary the threshold from $MinFreq = 0.02$ (appeared in a minimum of 5 traffic traces) to 0.2 (appeared in a minimum of 48 traces). Having only domains that appeared in 48 out of 239 traces is intuitively too high. Figure 6.4 shows how the number of features (left $y$-axis) and the overall performance (right $y$-axis) vary in function of *MinFreq* ($x$-axis). Looking at the figure from right to left, we notice an increase in both number of features and F1-score as *MinFreq* decreases, since the system lets more domains in. However,

the two curves follow a different slope, with the F1 score exhibiting two main jumps. With $MinFreq$ 0.13 through 0.10, by use as little as 4 features, the F1 score jumps to 0.54. Then, with $MinFreq = 0.07$, we have 9 features and an F1 score of 0.86, with small increases when further reducing $MinFreq$. Indeed, with very low $MinFreq$, we manually observe that the domains that are included are not related to the RTC applications but refer to background jobs of the PCs or parallel activity of users (the participants of the call opening news sites, shopping sites and so on). To sum up, these results show that our approach requires a relatively low number of features to achieve good performance – in this case 9 features are enough for distinguishing up to five applications. Nonetheless, it is robust to noise and domains that may appear in the traffic by chance.

On a side note, we also vary other parameters that are not shown here, since they do not offer improvements in performance. These include: enriching the features with names of Autonomous systems contacted, using only second level domains instead of the whole domain name and considering domains that have been contacted for a $\Delta T$ also after the call starts.

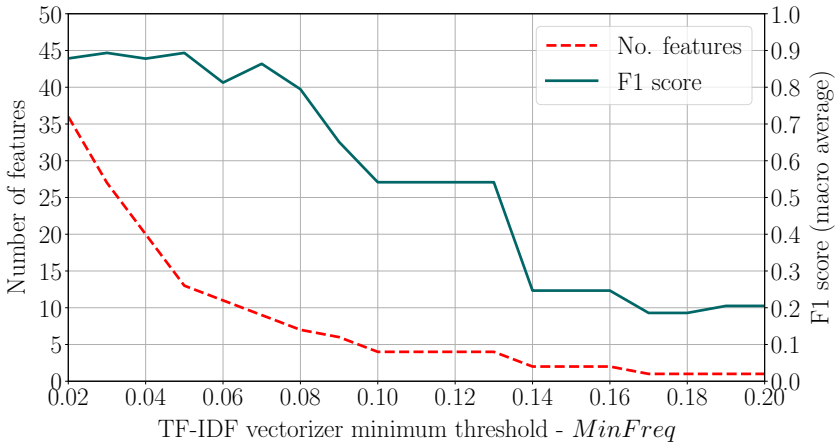



Fig. 6.4 Change of performance and number of features when varying the cut-off.

## 6.5 Takeaways

In this chapter, we presented an ML-based system for classifying RTC applications, solely based on packets seen before a call starts. Given the domain names resolved in a time frame prior to the start of a call, the system successfully distinguishes among 5

popular meeting applications with an F1 score of 0.89. It leverages NLP techniques and one-vs.-rest classifiers to build meaningful features and be robust to noisy data.

The approach can work in real-time since it just needs to read the domain names in control traffic (TLS Client Hello messages in this case) and performs immediate inference. The network can know that an RTC call is about to start and which application is going to be used. This can greatly help network management and potentially improve QoE of users of RTC applications, since different applications employ different data rates, and have different bandwidth and latency requirements. We emphasize the importance of such a system in more detail and outline possible deployment scenarios in a real network in Chapter 4.

We believe using NLP techniques in networking is very promising, since there is a lot of textual data, which is becoming more and more complex as traffic magnitude and the variety of networking protocols both grow. Here we leverage the domain names for traffic classification, but NLP can be useful for a lot of other networking problems. For example, security applications can make use of NLP for reading automated attacker scripts. It can also be extremely useful for traffic observability, as it can help in parsing application logs for effective telemetry.

# Chapter 7

# RTC Media Type Retrieval

The work we present in this chapter is mostly taken from our paper *Real-time classification of real-time communications*, published in *IEEE Transactions on Network and Service Management 2022* [116]. This chapter discusses an ML classifier that, given one second of RTC traffic, distinguishes the media type of the flow, into classes such as audio, video, screen sharing and error correction streams.

## 7.1 Introduction

In this chapter, we propose a novel ML-based application for classifying, in real-time, the RTP streams to the type of content they carry. Our approach is based on a few, but well-chosen features derived from the statistical properties of the traffic, which allow us to classify RTP streams using off-the-shelf supervised learning algorithms. Our approach identifies not only audio or video streams but also other properties of the media, such as the video quality or the use of Forward Error Correction (FEC) streams. Our solution works with minimal delay, deciding on the type of each stream within just 1 second of traffic. We design it as a software module that can be plugged into network devices (e.g., routers) or integrated into Software Defined Networks (SDN) to provide fine-grained traffic categorization and management (see Chapter 4 for more details).

Our study is based on two popular RTC applications for online multi-party meetings with audio, video, and screen sharing: Cisco Webex Teams[1] (later called Webex) as a business-oriented platform and Jitsi Meet[2] as a lightweight in-browser application. The workings of both these applications are described in Chapter 3. Using data coming from more than 62 hours of real calls, we evaluate the impact of feature selection and different classification algorithms. After careful feature selection and using a lightweight Decision tree classifier, we achieve an overall accuracy of 96% for Webex and 95% for Jitsi Meet, with no large differences across classes. Our models require little traffic to train and do not introduce systematic errors. We note that models trained for one RTC application (e.g., Webex) are hard to transfer to another application (e.g., Jitsi Meet) due to the different feature distributions. However, we show that we can partially overcome this limitation by using domain adaptation techniques.

We make our dataset, code, and trained classifiers available online.[3] We believe they can help researchers reproduce our results or apply them to different contexts.

The rest of the Chapter is organized as follows: In Section 7.2, we present and characterize our dataset, while in Section 7.3 we describe our methodology for feature engineering and classification. Section 7.4 shows our experimental results, and, finally, Section 7.5 concludes the chapter and discusses important takeaways.

## 7.2 Dataset

In this section, we describe the dataset we use throughout the paper. We first outline the RTC applications we choose for the analysis and their operation principles. Then we describe the data collection process and finally we provide a characterization study of the collected traffic.

### 7.2.1 RTC applications under study

For this chapter, we focus on two RTC applications: Cisco Webex and Jitsi Meet. As mentioned in Chapter 3, Webex is a business-oriented service that offers paid

[1] https://www.webex.com/
[2] https://meet.jit.si
[3] https://smartdata.polito.it/rtc-classification/

Table 7.1 Dataset summary.

| Class | No. of seconds | | | |
|---|---|---|---|---|
| | Webex | | Jitsi | |
| | Train | Test | Train | Test |
| Audio | 224 295 | 80 781 | 123 745 | 30 180 |
| Video LQ | 200 380 | 76 825 | 84 134 | 20 192 |
| Video MQ | 55 112 | 18 156 | 34 708 | 7 817 |
| Video HQ | 59 073 | 19 526 | 33 049 | 7 920 |
| Screen Sharing | 41 170 | 8 800 | 29 216 | 6 870 |
| FEC Audio | 146 567 | 41 247 | - | - |
| FEC Video | 45 591 | 2 164 | - | - |

plans for enterprises and institutions that require video call service. It is available as a standalone application for PC and mobile devices, but it can also be used through browsers that support the WebRTC standard. Jitsi Meet (or Jitsi for short) is a free of charge RTC application that provides a simple browser-based user interface for WebRTC-compliant browsers. It is fully open-source, and it is possible to run a private Jitsi server or use the public service available online. Both applications use RTP for streaming multimedia content along with STUN and TURN for session establishment. They support audio and video communication and allow users to share their screens with the other participants. Moreover, they adopt the Selective Forwarding Unit (SFU) approach [119], where participants send their multimedia content to a central server. The server then forwards the data, deciding which stream to send to each participant (more details in Chapter 3). Although the choice of different RTC applications (e.g., Zoom or Microsoft Teams) would be possible, we opted for Webex and Jitsi, which allow us to easily gather the classification ground truth, as we illustrate in Section 7.2.2. For other popular applications, we could not find such a convenient way to collect the needed information.

### 7.2.2 Data collection

We capture *real* calls using Webex and Jitsi, made under different network conditions, with a different number of participants (from two to ten), multimedia content (audio, video, screen sharing), and user equipment (PC, tablet, or phone). The calls run in a real environment where participants are connected via different networks from

3 countries and use different devices, from Windows PCs to iPhones and Android phones. During each call, at least one participant captures all the exchanged traffic and stores it in `pcap` format. The calls took place over a period of 6 months.

In our classification problem, we target RTP streams, which we identify with the tuple: (source IP address, source port, destination IP address, destination port and RTP SSRC). In other words, we target a single stream that carries a specific multimedia content. We divide the streams into 5-7 classes:

1. Audio
2. Low Quality (LQ) Video: 180p
3. Medium Quality (MQ) Video: 240-640p
4. High Quality (HQ) Video: 720p
5. Screen Sharing
6. FEC audio (Webex only)
7. FEC video (Webex only)

For Webex, we consider two additional classes: FEC audio and FEC video. Indeed, Webex uses FEC to mitigate packet losses, sending streams with redundant information to be used at the receiver if some packets are lost or contain errors. We observe FEC streams for audio and video, and we are interested in identifying them as separate classes. Hence, for the Jitsi classifier, we consider 5 classes and for the Webex classifier, 7.

We employ the application debugging logs to gather the ground truth, which maps each RTP stream to the content type. For Webex, logs are automatically generated during each call, while for Jitsi we use the Chrome browser WebRTC logs.[4] The logs for both applications contain per-second statistics for each stream, including the type of media (audio, video or screen sharing), the video resolution and the number of frames per second. During each call, the participant who captures the traffic also collects the logs, which we store alongside the `pcap` trace. Note that we cannot use the RTP Payload Type field for this, as it is dynamically assigned.

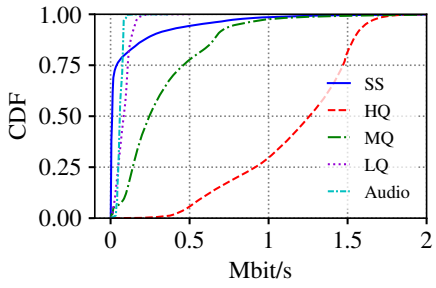
We collect traffic for approximately 62 hours of video calls, exchanged during 27 meetings with Webex and 50 meetings with Jitsi. They sum up to 90 GB of `pcap` files,

[4]This log can be obtained by creating and downloading a dump at *chrome://webrtc-internals*

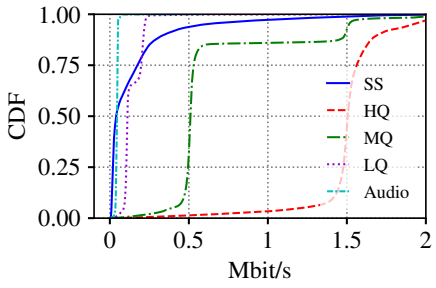
which include the call traffic as well as a small amount of background traffic that we neglect. The dataset contains 3 977 RTP streams for Webex and 521 for Jitsi. Each call contains a different mix of the above classes, and includes traffic generated by all participants as captured from the point of view of a single individual. Out of the 77 calls, 35 have only two participants, 11 have three participants and 31 include more than three. In Table 7.1 we give an overview of the dataset, separating the training and test set. In Section 7.3 we describe our training/testing methodology in detail. For each RTC application and class, we report the amount of data we collected, in seconds. The most represented classes for both applications are Audio and LQ video. While this is somewhat expected for audio, the prevalence of LQ video is due to the video thumbnails used in the applications to show inactive participants during calls with more than three participants. Note that for Webex, FEC audio is also widely represented. The least represented class is Screen Sharing, but the overall dataset imbalance is still limited, with the ratio between the support of the most and the least represented class being less than 6.

### 7.2.3 Characterization and challenges

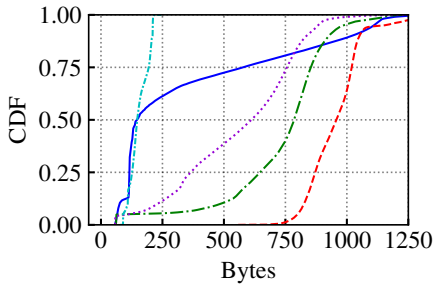
We provide a high-level overview of the dataset in Figure 7.1, where we plot the Cumulative Distribution Functions (CDFs) for different stream features, separately by application. We use different lines to contrast the four video-based classes, plus audio. The top figures show the bitrate distribution for Webex (Figure 7.1a) and Jitsi (Figure 7.1b). For each stream, we compute the average bitrate using 1-second bins. We first note that better video qualities tend to have higher bitrates (e.g., red and green lines). Audio (cyan line) has the lowest bitrate, as expected. However, the two applications present different shapes for the video curves. Webex displays smooth distributions, indicating that it adjusts the target bitrate of the video codec. In contrast, Jitsi exhibits a cascading behaviour, indicating thresholds and somewhat quantized bitrates. Note that the same video quality appears with multiple evident bitrate peaks. For example, MQ video (green dashed line) presents two peaks roughly at 0.5 and 1.5 Mbit/s, both corresponding to 640×360 video. The Screen Sharing class (solid blue line) exhibits the greatest variability. Again, this is expected, as it carries diverse contents, from slide sharing to scrolling through the screen, to effectively playing a video. This leads to a generally low bitrate with short periods of high activity. We note that setting a simple threshold on the bitrate would not yield



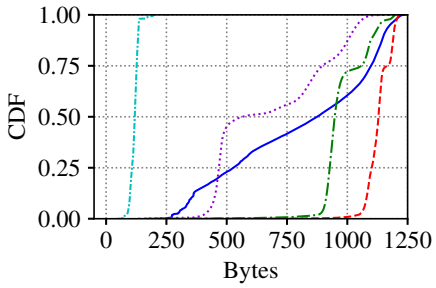

(a) Bitrate (Webex)



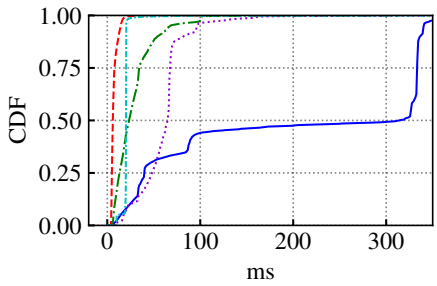

(b) Bitrate (Jitsi)



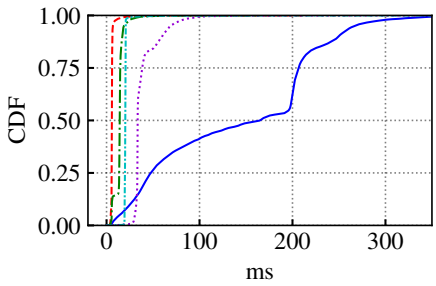

(c) Packet size (Webex)




(d) Packet size (Jitsi)




(e) Interarrival time (Webex)




(f) Interarrival time (Jitsi)

Fig. 7.1 Distribution of traffic characteristics for Webex (left) and Jitsi (right), separately for media stream type.

accurate class predictions. This is especially true for Webex, where the distributions overlap significantly. In particular, for screen sharing, the bitrate ranges from a few kbit/s to more than 1 Mbit/s. Interestingly, the Screen Sharing bitrate is often as low as an audio stream, for both applications.

Similar considerations hold for the packet size (Figures 7.1c and 7.1d - middle row). Better video qualities tend to use larger packets as they sustain a higher bitrate. Again, we observe a high overlap of Screen sharing with all other classes. For Webex (Figure 7.1c), Screen Sharing packets can be as little as those of audio streams. Conversely, for Jitsi (Figure 7.1d), only audio uses small (100-150B) packets, potentially easing its identification.

Finally, the bottom two figures show the distribution of packet inter-arrival time for Webex (Figure 7.1e) and Jitsi (Figure 7.1f). We compute the inter-arrival time as the time interval between two consecutive packets in the same RTP stream. The video distributions partially overlap, with Screen Sharing presenting inter-arrival time as large as 400 ms when nothing on the screen is changing. Figure 7.1 shows that a careful mixture of these features is required for accurate prediction. In the remainder of the paper, we show that it is possible to identify the type of media stream with high accuracy using features derived from these traffic characteristics and a machine learning classifier.

## 7.3 Methodology

In this section, we describe the proposed approach, from RTP traffic identification to feature extraction and classification. We envision an offline training of a classification model and its application to live traffic in real-time, as described in Chapter 4. We sketch a high-level overview of our approach in Figure 7.2. We also detail the methodology to build and select the features from RTP traffic. We follow the same approach for both Webex and Jitsi and create a separate classifier for each. Throughout this section, we use Webex as a running example to facilitate the understanding of the methodology.

To build the features from raw RTC traffic and align them with the ground truth from the application debug logs, we use *Retina*, our tool described in Chapter 5. In fact, *Retina* was born from the need to analyze and build features for this classifi-

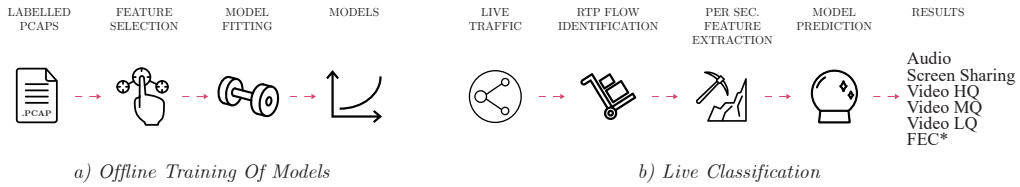



Fig. 7.2 Overview of the training and classification pipeline.

cation problem and later became a powerful tool for analyzing RTC traffic, as we added more functionalities and modules. However, for better understanding and completeness of the section, we describe the feature construction here as well.

**Problem statement.** Our goal is to classify the RTP streams that we observe on the network to one of the classes listed in Section 7.2.2 and Table 7.1. We want to solve this task in real time, i.e., make a decision based solely on the traffic observed in a short time interval, by applying a model trained on historical data. Thus, our classification target is an RTP stream as observed during a certain time bin (from 200 ms to 5 s).

**RTP stream identification.** We identify the RTP traffic with straightforward Deep packet inspection (DPI), by matching the protocol headers. Indeed, the RTP header includes fixed-sized fields that facilitate its identification, and its sequence number serves as a simple sanity check for identification, since it must increase by 1 for subsequent packets. Popular passive meters identify RTP flows using DPI – e.g., Tstat [89] or nProbe [113]. Note that we do not handle the case of RTP tunneled through an encrypted channel (e.g., over a VPN or IPSec tunnel), since we cannot distinguish the different streams. We separate multiple media streams via their SSRC. We are not interested in the control traffic for, e.g., session establishment or login, and thus neglect it. We also assume that we know the application in use (Webex or Jitsi), using our classifier from Chapter 6.

**The ML pipeline.** A single RTP stream results in many samples (one per time bin) that we shall classify. For our classification problem, we follow the classical approach of supervised learning. First, we extract meaningful features from the data, guided by domain knowledge on network traffic and the RTP protocol. Then, we perform a two-step feature selection process by first discarding highly correlated features and then performing a recursive feature elimination. Finally, we train a

machine learning classifier and evaluate its performance on an independent test set. Feature selection and algorithm training are performed offline, while the system is designed to compute features and classify new samples in real time (see Figure 7.2). The time it takes for inference is equivalent to the chosen time bin plus the feature computation and algorithm run, whose execution time is negligible. Our code is written in Python and uses the scikit-learn library [120] for machine learning. Our methodology is readily amenable to parallelization, as all processing is done on a per-flow basis – i.e., feature extraction and classification only need to obtain data from a single stream. Therefore, a multi-core parallel approach is fully feasible, and we do not expect any bottlenecks in high-speed deployments, provided packet capture is adequate. In case of deployment with off-the-shelf hardware, in addition to the deployment scenarios described in Chapter 4, high-speed packet capture libraries (e.g., DPDK[5]) together with Network Cards natively supporting load balancing (e.g., Receive-Side Scaling on Intel cards) would perfectly serve at this goal.

**Train/test methodology.** We split the video call dataset into a training and a test set, keeping separate calls in the training and test set, to prevent overfitting and obtain robust results. We perform feature selection and algorithm hyper-parameter tuning on the training set, and we evaluate classification performance on the test set (which we never use at training). Note that the streams of a single call are used either at training or testing time to keep the two sets completely independent. For Webex, out of 27 calls, we use data from 22 calls for training and data from the remaining 5 calls for testing. For Jitsi we use data from 41 for training and 9 for testing. With this split, we obtain roughly 80% of samples (1-second bins) for training and 20% for testing (see Table 7.1). We also verify that each class is well-represented in both sets. As a global performance indicator, we use the macro-average (a simple mean) of the F1-scores of each class. The F1-Score is the harmonic mean between the Precision and Recall of a class. For some analysis, we also consider accuracy as a concise index of overall performance, since classes are not strongly imbalanced. The accuracy is the share of correct predictions over the total number of predictions in the test set.

**Feature extraction.** We extract features from the packets separately by RTP stream and time bin. The features are based on the fields of the RTP protocol and take into consideration its operation. We outline the Feature extraction approach in Figure 5.2 of Chapter 5. We consider five groups of features, reported in the middle column

[5] https://www.dpdk.org/

of the figure. These include packet characteristics (size, time, volume) and the RTP timestamp field, which indicates the time at which the content was generated at the source. RTP has a few other fields that essentially indicate header extensions, which we do not include because they are very application and client-specific. Since two of the selected fields (packet time and RTP timestamp) represent time instants, we only consider their relative variation across packets (called *delta* on the Figure), since the absolute values are useless in our context. For packet size, we use both absolute and relative values. We extract these five values for all packets and compute various statistical indices to create the final features, such as range, mean, standard deviation, percentiles, third and fourth moments, etc. Since we find that the same values recur frequently in the packets, we also add features that measure the number of unique values, the percentage of occurrence of the most frequent value (mode), and the ratio between the minimum value and the range. We report the complete list of statistical indicators on the last column of Figure 5.2. Finally, we consider the traffic volume in terms of the number of packets and bitrate observed in the time bin. We also use the number of packets with the RTP marker flag set as a separate feature.

Since our goal is to design a real-time classification system, we create features that can be computed on the fly by considering only the packets observed in a time bin. Intuitively, the smaller the time bin is, the faster the stream is classified. However, features are more representative with larger time bins since they are computed over a more extensive set of packets. In Section 7.4, we explore this trade-off and evaluate how the temporal granularity affects the classification of an entire stream. Finally, note that we also avoid features that require linking multiple streams to keep our design simple and easily to parallelize.

**Feature selection.** In total, we extract 96 features derived from the four empirical distributions mentioned above, plus volume. We publish the full list of features on our research center website.[6] To remove those that are redundant and shrink the overall number of features, we perform a two-step selection process.

1. *Correlation analysis*: We perform an initial feature selection by measuring the correlation between each pair of features. We evaluate all possible pairs in a random order, and whenever we find a Pearson correlation coefficient greater than 0.9 (in absolute terms), we keep only one of the two features at random. With this step, we roughly eliminate half of the features.

[6] https://smartdata.polito.it/rtc-classification/

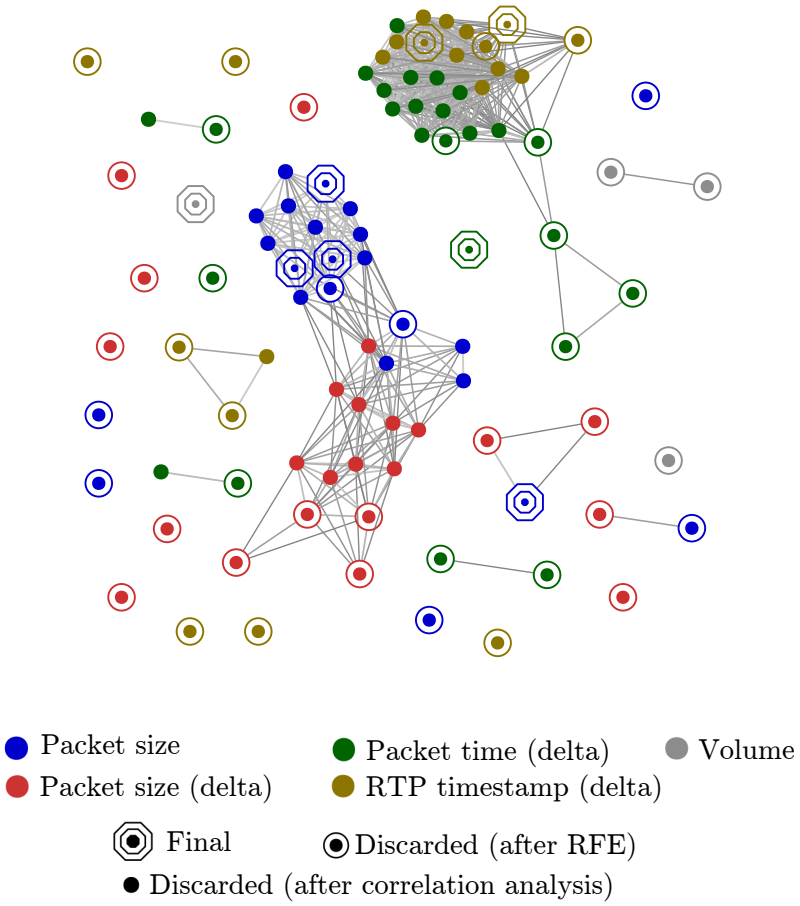



Fig. 7.3 Graph representing the correlation between features. The color indicates the feature set, the shape whether the feature is kept after feature selection and the distance represents the correlation.

2. *Recursive Feature Elimination using the ExtraTree algorithm*: We use the Recursive Feature Elimination (RFE) approach [121] to refine our list of features, maintaining only those that are most useful for our classification problem. Using RFE, we train an ExtraTree classifier on the training set and rank the features by their *feature importance* as provided by the algorithm.[7] We then eliminate the one with the least importance. We recursively repeat this procedure until we reach the minimum number of features and the best performance, which we evaluate using 5-fold cross-validation. Note that tree-based feature ranking is known to be biased in the case of groups of correlated

[7]The ExtraTree classifier natively exposes the feature importance after training.

features [122]. Thus, our first step (correlation analysis) is essential for RFE to work correctly.

We graphically illustrate the entire feature selection process for Webex with Figure 7.3, which shows the initial 96 features in the form of a graph. Each node represents a feature, and the length of edges is (roughly) inversely proportional to the correlation among pairs in absolute value – i.e., highly correlated features remain close to each other. For illustration purposes, we only show the edges where the correlation is higher than 0.5 (in absolute value). Different colours represent the feature sets, while the shape of each node indicates whether a feature is maintained or discarded at one of the selection steps: a circle means that the feature was discarded after correlation analysis, a double circle means that the feature was discarded with RFE, and an octagon means that it passed both steps and is included in the final list.

We first notice that the correlation analysis step maintains all features which are poorly correlated with other ones: all nodes without edges are either double circles or octagons. On the contrary, among groups of highly correlated features, only a few samples are retained. For example, the dense community in the top right of the figure includes the percentiles of packet time inter-arrival time and RTP timestamp, which are intuitively highly correlated. We retain only two of them.

Continuing with the running example of Webex, the first step of the feature selection shrinks our set from 96 to 47 features. We then perform RFE to obtain only those that are useful for our classification problem. We train an ExtraTree classifier on the remaining 47 features, running a 5-fold cross-validation to evaluate how accurate the obtained model is. We then eliminate the feature ranked as least important and repeat this process until we find that the classification performance starts to decrease. In Figure 7.4, we show how the average F1 score varies when removing an increasing number of features. The figure shows our results for both Webex (solid blue line) and Jitsi (red dashed line).

Considering Webex, when we use all 47 features, we get an F1-score of 0.91. The performance is almost stable (with minimal variations) until we use 8 features only – i.e., we eliminate 39. Then, the accuracy starts decreasing consistently. After analysing the curve, we decide to set the final number of features to 8. Interestingly, we notice that every feature group (except the packet size delta) appears in the set of the final features (there is an octagon of every colour except red in Figure 7.3). Among the final features, we find the packet size (mode, $25^{\text{th}}$, $70^{\text{th}}$ and $75^{\text{th}}$ per-

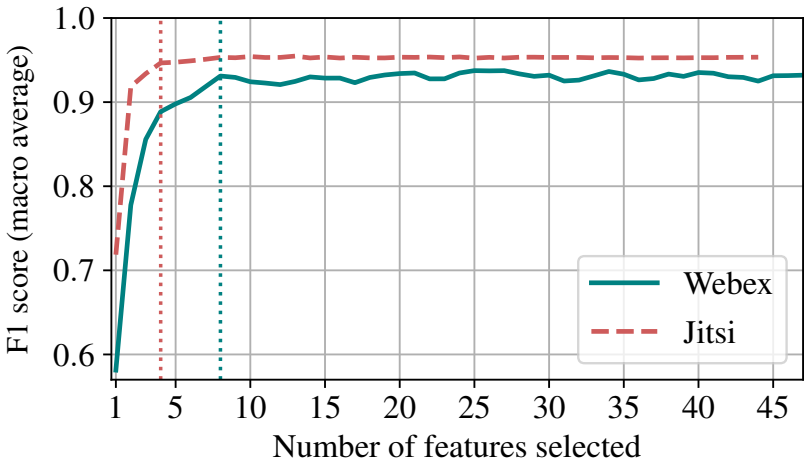



Fig. 7.4 Mean F1 score when varying the number of features. The vertical lines indicate the final number of features.

centile), the 30$^{th}$ percentile and mode of the RTP timestamp delta, the mode of the inter-arrival time and the number of packets with the RTP marker flag set. Intuitively, for each characteristic of the packets, we keep a few statistical properties of its distribution.

The process is similar for Jitsi (red dashed line in Figure 7.4). Note that the curve ends at 43 features, since for Jitsi the first step of feature selection eliminates a slightly larger number of features. The knee in the line shows that we already achieve good performance with as little as 4 features. Among them, we find three representatives of the packet size feature group and the mode of the RTP timestamp delta. This indicates that the packet length is a vital factor for this classification problem.

**Multi-class classification.** Using the features that we obtain after the feature selection, we try different classification algorithms to find the one that yields a proper trade-off between performance and simplicity. The algorithms we consider are: tree-based classifiers [Decision Tree (DT) and Random Forest (RF)], k-Nearest Neighbors (k-NN), which classifies points based on proximity to other data points, and Gaussian Naïve Bayes (GNB) as a generative probability model. We perform hyper-parameter tuning with 5-fold cross-validation for each of these models, using only the training set. We then evaluate their performance on the separate test set,

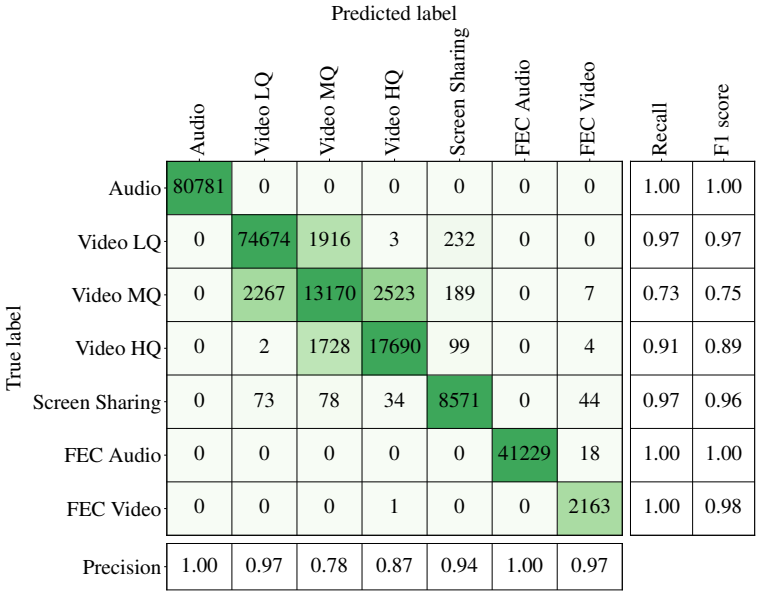
using the macro-averaged F1-score as a performance indicator. In Section 7.4, we show that the algorithm choice has a moderate impact on classification performance.

## 7.4 Experimental results

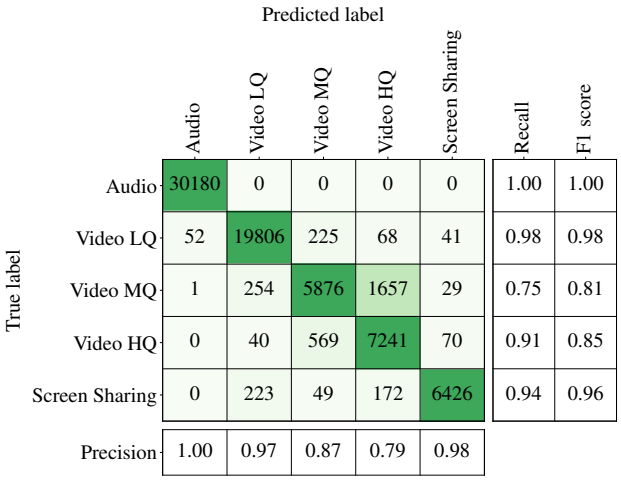
In this section, we present our experimental results for the entire classification problem. First, we discuss the overall classification performance and quantify the impact of the time bin duration, classification algorithm and training set size. Then, we discuss the importance of the features and analyze how classification errors arise. Finally, we investigate the possibility of transferring a model trained for one RTC application to another. All results are obtained by training classification models on the training set and evaluating their performance on the independent test set.

### 7.4.1 Best classification performance

We first report the performance we obtain for both RTC applications when using the best models. Indeed, we try different classification algorithms and finally opt to use a Decision Tree classifier, which provides solid performance and a simple model. Running hyper-parameter tuning, we obtain the best results when using the Gini index as a purity measure. In Figure 7.5, we show the confusion matrices for both Webex and Jitsi using a 1s time bin. By definition, a confusion matrix $C$ is such that $C_{i,j}$ is equal to the number of observations known to be in group $i$ and predicted to be in group $j$. Thus, the main diagonal represents the number of correctly classified samples. We also show the per-class recall and F1-score in the last two columns and precision in the bottom row. We note that, for both applications, all classes except Video MQ and HQ exhibit an F1-Score above 0.96, and thus high precision and recall. Audio is the best performing class for both RTC applications, together with FEC audio for Webex. Here only a handful of samples are misclassified, suggesting that audio streams are generally easy to isolate. Indeed, for Jitsi especially, audio streams tend to use smaller packets than video (see Figure 7.1), making their identification simpler. The worst performing class is video MQ, with F1 scores of 0.73 and 0.75 for Webex and Jitsi, respectively. The confusion matrices reveal that the three different video qualities are, in some cases, confused with each other. Although this is a flaw of our classification model, we tolerate this behaviour given the similar nature of




(a) Webex




(b) Jitsi

Fig. 7.5 Confusion matrices when using a Decision Tree classifier and 1s time bins.

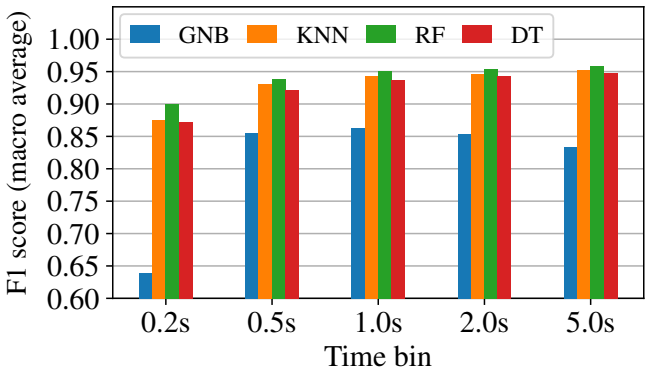
the three classes. Also, keep in mind that applications (especially Webex) use video codecs with variable bitrates that result in different network traffic (see Section 7.2). Overall, for Webex, 96.3% of the samples are classified correctly (i.e., accuracy), and the average F1-score is 0.94. For Jitsi, we obtain an accuracy of 95.3% and an average F1-score of 0.92.

Considering computational time, our system needs to perform 3 consecutive steps before providing the final classification label: (i) Wait for the time bin to gather traffic information, (ii) Calculate the features and (iii) Apply the classification model. Step (i) obviously takes most of the time. Step (ii) depends on the class, with Video HQ being the most expensive as it sends the highest number of packets, thus increasing the number of samples in the calculation. On average, this step takes a few milliseconds with our Python code on commodity servers. Finally, step (iii) is even faster, requiring the use of a light-weight decision tree model, that takes tens of microseconds. For high-speed deployments, we envision the use of a parallel multi-core architecture to scale the processing. Such an approach is completely feasible since the classification relies on features extracted on a per-UDP flow basis.

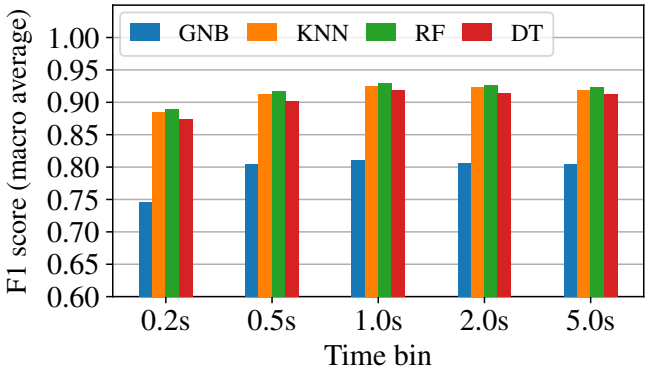
### 7.4.2 Parameter sensitivity

We now discuss the impact of the time bin duration on the classification performance. Indeed, we are interested in classifying a stream as fast as possible without sacrificing accuracy. Figure 7.6 shows how performance varies with different time bin durations, from 200ms to 5s. We provide results for 4 classification algorithms, and the $y$-axis reports the average F1-score we obtain. We find that we generally get better results with larger time bins. This is no surprise since the features are computed over more extensive sets of packets. For example, in 200ms of a typical audio stream, only 10 packets are generated. The performance flattens for values larger than 1s for both applications, implying that such a time frame is large enough to capture representative features about a stream. We believe that a delay of 1s is not critical, since RTC calls typically last minutes.

Looking at Figure 7.6, we can also compare the performance of different classification algorithms. We observe no large differences, except for Gaussian Naïve Bayes, which exhibits somewhat worse performance, probably due to the simplicity of the model. Note that the lowest F1-score is 0.62 for Webex and 0.73 for Jitsi. This




(a) Webex




(b) Jitsi

Fig. 7.6 Performance of the four algorithms for different time bins.

confirms that our careful feature engineering and selection make the results robust to the choice of algorithm. We finally opt to use a Decision Tree for its simplicity, interpretability and speed. Random Forest produces similar results, but is more computationally intensive as it uses trees in parallel, 100 in our case. k-NN also performs well, but requires the model to store the entire training set in the main memory, resulting in significant memory consumption. Using a Decision Tree instead, the model is only a few *kB* in size. Comparing the two applications confirms that they exhibit very similar performance, with Jitsi having a lower F1-score by about 0.02 in most cases.

### 7.4.3 Training set size

We now investigate how much training data is necessary to achieve good classification performance. To this end, we train many classification models, gradually increasing

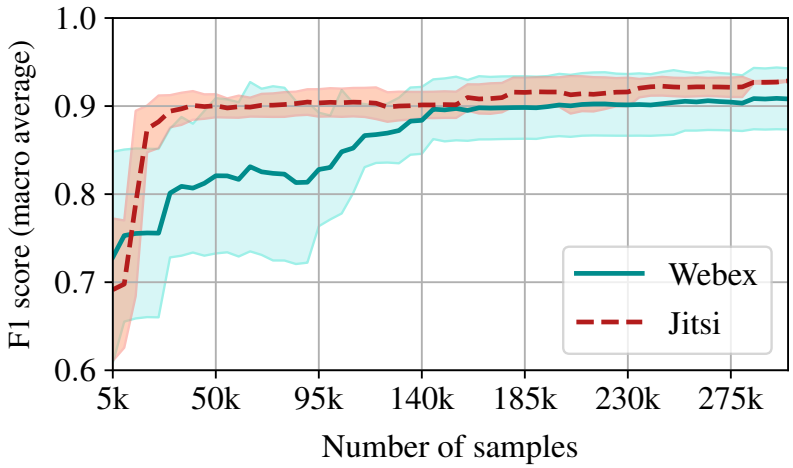



Fig. 7.7 Learning curve: Relationship between the number of training samples and the score.

the size of the training set. We vary the number of training set samples selecting them from the least possible number of calls. In other words, we entirely consume the samples from one call before drawing them from a second. In this way, we indirectly observe how many calls are required. Note that randomly selecting training data from all calls would likely sample the diversity of the entire dataset, which is unfair for our analysis. In this experiment, we use Decision Tree classifiers with 1s time bins.

Figure 7.7 shows the classification performance versus the training set size. Again, we measure the performance using macro-averaged F1- score on the test set. We repeat each experiment 5 times, shuffling the order of the calls but still drawing samples from one call altogether. The solid blue and red dashed lines indicate the mean score of the experiments for Webex and Jitsi, respectively. The areas represent the standard deviation across the runs. Starting from Jitsi, we notice that the performance improves very quickly with the training set size– with only 20k samples, the F1- score is already above 0.86. Such an amount of time corresponds to 5 hours of audio and video call. After that, it increases very gradually, reaching a local maximum of 0.92 F1 score at 200k samples (55 hours of calls). The standard deviation is generally small and stable. This result suggests that the features we extract and the nature of the problem do not require a large dataset to obtain a reliable model. Conversely, Webex requires a larger training set for accurate classification, exhibiting a slow growth and a larger standard deviation, stabilizing at 145k samples (40 hours of calls). This is likely due to the higher number of classes (with the

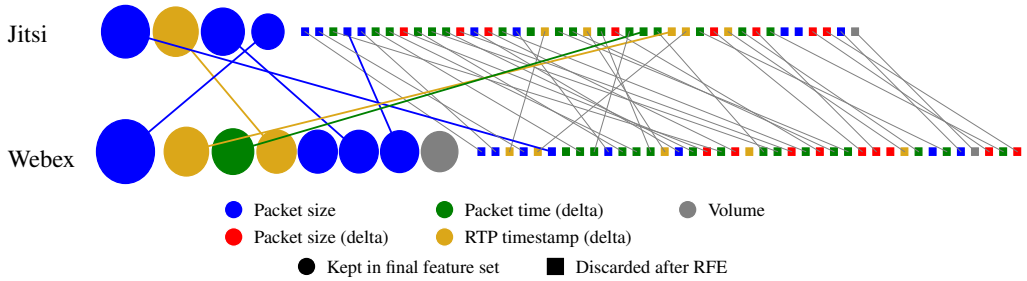



Fig. 7.8 Feature importance comparison between Webex and Jitsi.

additional audio and video FEC classes) and a variegated behaviour of the application within a call. Indeed, we observe that there is an abundance of audio and video LQ in various calls and a deficiency of the other classes. Consequently, additional calls are necessary to bridge the gap. To test this conjecture, we perform an additional experiment where we balance the number of samples per class and find that the performance converges faster.

### 7.4.4 Feature analysis

We now discuss the outcomes of the feature selection phase. Our goal is to investigate whether we can recommend a fixed set of features for any RTC application or they are specific for each one. As described in Section 7.3, we carry out a two-fold feature selection: we first remove highly correlated features, and then we perform recursive feature elimination using an ExtraTree classifier. In Figure 7.8 we compare the results of the second step for Webex and Jitsi. Each symbol represents a feature that we retain after the correlation analysis – 43 for Jitsi (upper row) and 47 for Webex (lower row). Circles represent the features that are finally selected, and their size is proportional to the relative importance given by the ExtraTree classifier. The squares represent the remaining features, that were discarded using RFE. We arrange them in the order in which they were discarded. The colours indicate the feature group and they match the colours of Figure 7.3. The edges connect the same feature on the two RTC applications so we can compare Jitsi and Webex.

As already shown in Figure 7.4, with Jitsi, 4 features are enough to achieve good performance, while Webex needs 8. Looking at Figure 7.8, we observe a large presence of features related to the packet size (blue) – 3 out of 4 for Jitsi and 4 out of 8 for Webex. This is expected, as the packet size is instrumental for

distinguishing audio and video streams (see Figure 7.1). We note that 3 of the Jitsi features also appear in Webex, albeit with different importance. Overall, the features are ranked similarly for the two applications, and the Spearman's rank correlation coefficient between the two ranks (including all features shown in Figure 7.8) is 0.70. Interestingly, two features chosen for Webex have been discarded in the first feature selection phase for Jitsi – two circles on the bottom row are not connected to any of the above shapes. A notable one is the number of packets with the RTP packets with the marker flag set (the gray circle). We note that this feature correlates strongly with frame rate in video streams, and speculate it helps identify the screen sharing class, typically with a low frame rate.

### 7.4.5 Error analysis

We now analyze misclassification cases to understand (i) how they are spread among streams and (ii) whether they can affect the prompt classification of streams.

Overall, we obtain an accuracy of 96.3% for Webex and 95.3% for Jitsi, as detailed in Section 7.4.1. Here, we want to measure whether these errors are concentrated on a few RTP streams or are scattered between all. To this end, in Figure 7.9, we plot the complementary cumulative distribution function (CCDF) of the percentage of errors per RTP stream. In other words, for each stream in the test set, we compute the percentage of misclassified samples and then show the distribution over all streams. The test set includes 508 streams for Webex and 101 for Jitsi. We observe that most of them present a rather low error rate. For Webex (solid blue curve), we notice that the probability of misclassifying more than 10% of the samples of a stream is $\approx 10\%$. Moreover, the probability of misclassifying more than 50% is less than 2%. This result suggests that, in general, mistakes span through many different streams rather than all originating from a few, and our classifier typically does not commit systematic errors. Similar considerations hold for Jitsi. There are only a handful of streams for which most samples are assigned to the wrong class – see the right-most side of the plot. These are usually short-lived streams (shorter than $10s$), except two long Webex video MQ streams where 68% of samples are misclassified and one long Jitsi video MQ stream with 73%. As reported in Section 7.4.1, video MQ is the hardest class to discern. In conclusion, these results show that the misclassification of an entire flow is very unlikely to happen.

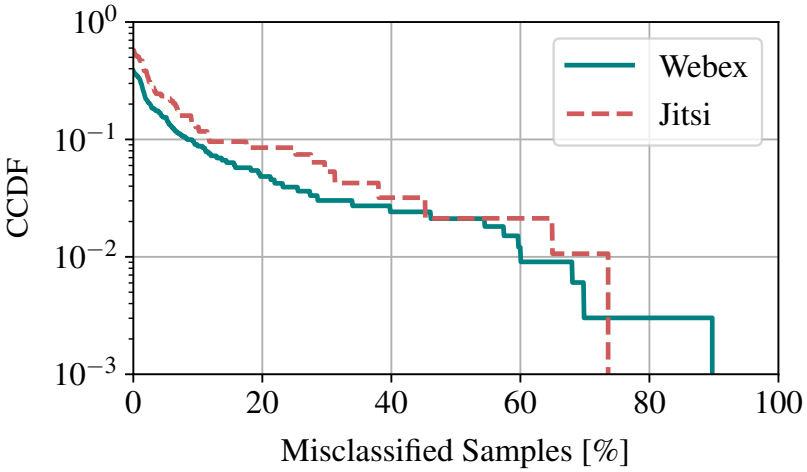



Fig. 7.9 CCDF of percentage of errors per stream.

We next investigate the possibility of classifying an entire stream just by looking at the first few samples. It might be beneficial in some real deployments when the network must react quickly to new streams to – e.g., prioritize particular traffic classes (see Chapter 4 for possible deployment scenarios). To this end, we suppose to classify a new *stream* based on the first $N$ samples, using a majority vote scheme on the labels we obtain for those samples. In other words, given the first $N$ samples of a stream, we assign it entirely to the class most samples have been assigned to. In Figure 7.10, we show the macro-averaged F1-score we obtain, varying $N$ between 1 and 30 seconds. In this case, the classification goal is a *stream* rather than a *sample*, and, as such, we compute performance metrics over the streams in the test set. When classifying the stream based solely on the first second, we obtain 0.92 macro-averaged F1-score for Webex (solid blue line) and 0.82 for Jitsi (red dashed line), as sporadic errors have the maximum impact. Increasing the number of samples $N$, we obtain better results, reaching macro-averaged F1-Score of 0.99 and 0.93 for Webex and Jitsi, respectively. Indeed, our classifier hardly perpetrates systematic errors (see the previous paragraph), making the majority voting scheme very robust to misclassification. We conclude that our approach is fully appropriate in contexts where the network is required to quickly make decisions on an entire flow, e.g., installing appropriate SDN rules on the network switches.

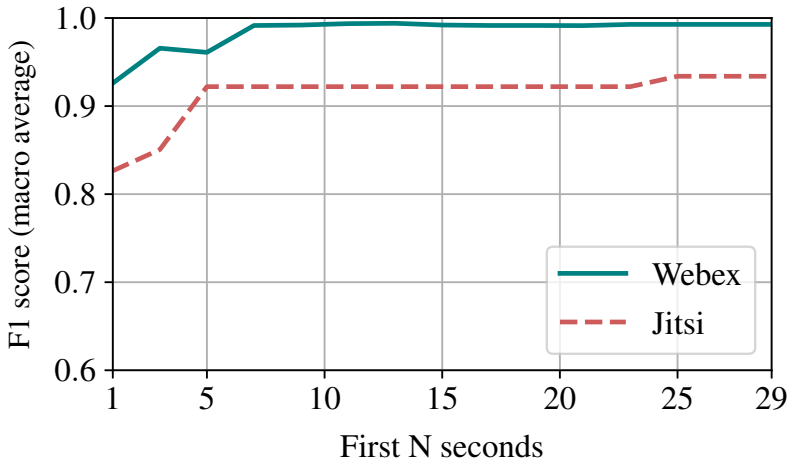



Fig. 7.10 Classification performance using first $N$ samples per stream.

### 7.4.6 Model transfer to other applications

In our previous results, we train a classifier with labelled data belonging to the same RTC application that we aim at classifying. This might not always be possible, as labelled data are hard and expensive to obtain. Moreover, new RTC applications may spread rapidly without controlled experiments being possible. In this section, we explore to what extent a classifier trained for RTC application $A$ can be used to classify streams of the application $B$.

For our goal, we investigate the use of *transfer learning* techniques [123], whose goal is to transfer knowledge from one *domain* (i.e., one RTC application) to another. These techniques are useful when we cannot collect labelled data in the second domain. In this case, we can try to use the knowledge from domain $A$ to solve the same problem in domain $B$. In general, the rationale behind transfer learning techniques is to modify and adapt an ML classifier trained in domain $A$ to classify samples in domain $B$.

Here, we employ the domain adaptation technique called CORrelation ALignement (CORAL) [55]. As the name suggests, given the feature distributions from two domains ($A$ and $B$), CORAL tries to align the covariance matrix (matrix of second-order moments) of distribution $B$ to the one of distribution $A$. Due to the nature of our problem, we hypothesize this approach suitable since we target two similar RTC applications that use the same network protocols. Necessary for our goal, CORAL is

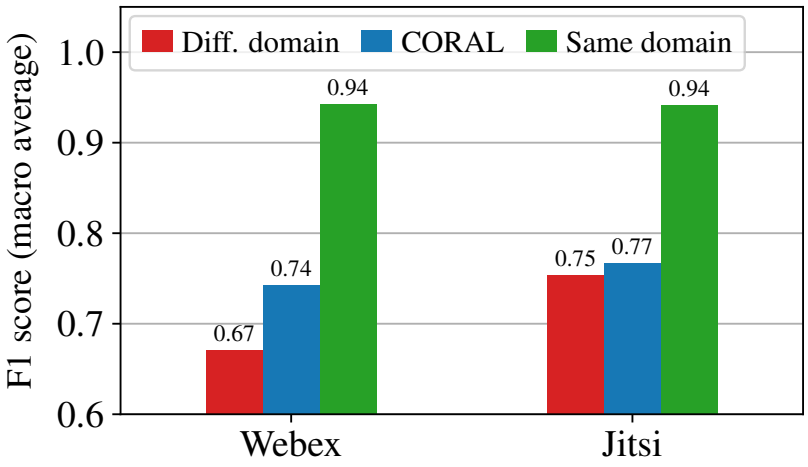



Fig. 7.11 Classification performance varying the target domain.

an unsupervised technique, as it assumes data for domain *B* are available, but without class labels.

We here investigate the performance we obtain when using a classifier trained on application *A* (e.g., Webex) for classifying data of application *B* (e.g., Jitsi). We perform experiments (i) using the classifier directly on application *B* and (ii) using CORAL to align domains *A* and *B*. Case (i) corresponds to using a classifier directly outside of the training context. In case (ii), we assume that non-labelled data for application *B* are available, allowing the use of CORAL to align the two domains. We show the results in Figure 7.11, again measuring performance in terms of macro-averaged F1-Score. The *x*-axis reports the domain on which the classifier is trained, while the colour of the bars indicates the domain on which we use it. We provide a reference using the green bars, indicating the performance we obtain when we use the classifier in its domain –i.e., the approach we used in the previous sections. For this experiment, we remove the FEC streams from the Webex traffic, since we need the same number of classes for the two applications for a fair comparison. The red bars represent case (i), while the blue bars case (ii).

We first notice how using a classifier directly on a different RTC application entails a certain performance drop (red bars). Indeed, using a classifier trained on Webex to classify Jitsi streams leads to a 0.67 macro-averaged F1 score. In the opposite direction (training on Jitsi and testing on Webex), the performance is slightly better (0.75). The use of CORAL improves the performance in both directions,

yielding similar results in both directions (blue bars). We get an F1-Score of 0.74 when training on Webex and using Jitsi and 0.77 vice-versa. Interestingly, the benefit of CORAL is higher in the former case (+0.07), while minimal in the latter (+0.02). Nevertheless, it is still far from the performance obtained by training a model on the same domain, which then soars to an F1-score of 0.94 for both applications (green bars). This might originate from the different shapes of traffic distributions between the two RTC applications, as discussed in Section 7.2. In summary, our results suggest that it is possible to use a classifier for a different application if lower performance can be tolerated. If non-labelled data for the target RTC application are available, CORAL is instrumental in increasing the performance.

## 7.5 Takeaways

In this chapter, we proposed a machine learning approach to classify the media streams generated by RTC applications in real time. Given a media stream carried within the RTP protocol, we can distinguish seven different classes, including different video qualities, screen sharing and redundant data used to mitigate losses (i.e., FEC streams). We carefully engineered features based on packet characteristics and designed the system to work with a minimal set of features using a light yet accurate tree-based model. We chose Webex Teams and Jitsi Meet as case studies and showed that we achieve high classification performance with only 1 second classification delay. Our approach is robust to the choice of classification algorithm and rarely commits systematic errors. Our experiments show that it is possible to use a model trained for one RTC application to classify streams of another, albeit with a performance penalty. If non-labelled data from the other application are available, it is possible to use transfer learning techniques to achieve better results.

Our approach is designed as a building block of a network management system that optimizes traffic engineering for RTC applications. It could help the network prioritize more important streams, for example, the screen sharing and audio of a meeting presenter, as opposed to the video of the other participants. Possible deployment scenarios of this system are outlined in Chapter 4.

It would be nice to be able to collect data from different vantage points of the network and evaluate our approach. Indeed, we base our approach on traffic from the

end-host, while the network traffic statistics would be different in different vantage points of the network [57]. However, this kind of traffic data is very hard to obtain.

What is interesting in this chapter is the difficulty in generalizing the model to multiple applications. This is expected because of the different characteristics of the traffic in both applications (Figure 7.1). However, it would be useful to have a general model that works for multiple RTC applications and to evaluate how much data and from which RTC applications would be enough to train such a model. We leave this as future work.

# Chapter 8

# ReCoCo: Reinforcement learning-based Congestion control for RTC

The contents of this chapter has been accepted for publication at the *2023 IEEE 24th International Conference on High-Performance Switching and Routing (IEEE HPSR 2023)*. This Chapter presents a Congestion Control algorithm for RTC applications based on Reinforcement Learning.

## 8.1 Introduction

One way to improve QoE of users of RTC applications is through good Congestion Control (CC). RTC applications use RTP mostly over UDP [77], so they are not subject to TCP congestion control protocols. Instead, CC is implemented via rate adaptation at the application layer, by using a feedback mechanism between the sender and receiver based on the Real-time Control Protocol (RTCP). The biggest challenge lies in the low-latency requirement of real-time applications. Thus, the goal of CC algorithms is to produce a sending rate as close as possible to the available end-to-end bandwidth, while maintaining the queue occupancy as low as possible [124]. The sending rate directly affects the packet delay, losses and throughput, which are the main drivers of network QoE [13]. The algorithms in use by RTC applications today (that are not proprietary) are heuristic schemes that make decisions on increasing or decreasing the sending rate, based on the one-way queuing delay and loss ratio [124]. The most notable open-source algorithm is Google's

GCC [14], which measures the delay variation and compares it with a dynamic threshold. However, in a complicated network scenario, such as wireless network links with very variable bandwidth, it is hard to optimize all network metrics with a heuristic scheme. *GCC* is slow to follow a changing bandwidth, since it increases the sending rate by 5% every second [125], ending up in under-utilization. It is also quite conservative in the event of bursty losses [125, 13]. To combat these limitations, we propose a novel rate adaptation scheme, based on Reinforcement learning (RL).

Using RL for congestion control in RTC has been somewhat explored in the literature [74–76]. Authors of [74] propose a hybrid solution, where they use *GCC*, but augmented by a factor given by an RL agent. [75] and [76] are full RL solutions, albeit limited in the exploration of different RL algorithms and reward designs. In [76], authors use both packet-level and frame-level statistics of traffic to make decisions, which requires substantial changes in the RTP protocol. We further elaborate on the limitations of these works and differences with respect to ours in Section 2.3.

In this chapter, we propose *ReCoCo*, a fully-RL based solution for congestion control in real-time applications. To create the system and experiments, we build upon the open-source framework OpenNetLab [126]. This framework was first built to serve the *MMSys2021 Grand challenge* [127], which called for a novel bandwidth estimation scheme for RTC. We thus use some of the performance metrics defined by this challenge to evaluate the approach. We assess 3 different RL algorithms in a number of parameter configurations, on 9 different bandwidth trace files that comprise of wired, 4G and 5G channels covering a vast array of bandwidth levels. We train both specific and general models to evaluate the difficulty of generalizing RL algorithms. We find that, when trained on each trace file separately, with specialized configuration, *ReCoCo* outperforms *GCC* for every trace, by 8.95 QoE units on average, especially for traces with high bandwidth. When training a single model for all network conditions, the best way is to use curriculum training, ordering the environments easiest to hardest based on improvement over a heuristic baseline (gap-to-baseline). In this case we observe a performance penalty of 8.76 QoE units on average over the specialized model, but still outperform *GCC* by 0.2 QoE units.

To make the research reproducible, we disclose the code and trained models[1].

[1] https://github.com/denama/ReCoCo

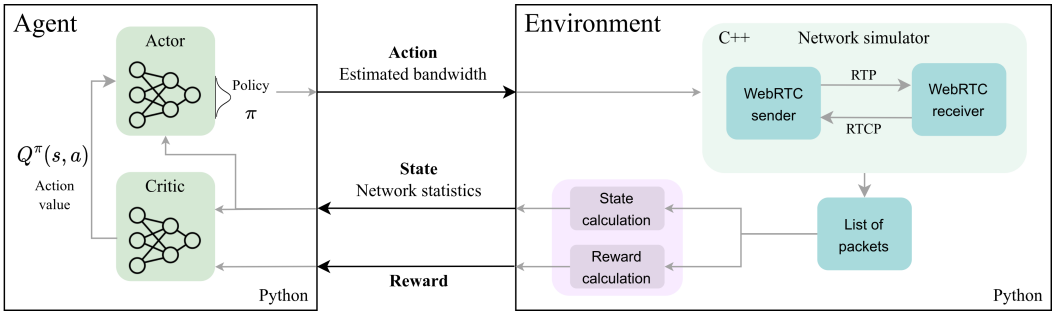



Fig. 8.1 Overview of the training mechanism.

The rest of the chapter is organized as follows: Section 8.2 introduces the main RL concepts, Section 8.3 describes our system for training *ReCoCo*, Section 8.4 outlines the experimental setup, while Section 8.5 discusses the results. Finally, Section 8.6 concludes the chapter.

## 8.2 Background: Deep RL basics

The setting of RL [128] consists of an *agent* that interacts with an *environment*, in a discrete time stochastic control process. At every time step $t$, the agent finds itself in a state $\mathbf{S_t}$ and takes an action $a_t$. This action brings the agent a reward $R_t$ and transitions it to the next state $\mathbf{S_{t+1}}$. Which action an agent takes at any given time step from any given state is defined by its policy $\pi$. The agent's goal is to learn a policy $\pi$ that maximizes the reward in the long run. In fact, it aims to maximize the *discounted return* $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$, where $\gamma$ is a discount factor decreasing the weight of past rewards. In many real-life tasks, the state space is arbitrarily large and often continuous. Here the agent learns a policy $\pi$ through a function approximator - usually a neural network (Deep RL). The algorithms we use in this chapter make use of the Actor-critic architecture [129] for Deep RL. The actor-critic framework constitutes two neural networks, an actor and a critic [130]. The actor learns the policy $\pi$ and decides the action to take at every time step. The critic evaluates how good that action was compared to the average for that state and informs the actor. The actor then changes the weights of the policy function accordingly, to adjust the probability of that action being taken. In formal terms, the critic learns an *action value function* $Q^{\pi}(s,a)$.

## 8.3 System overview

In this section, we describe the system used to train *ReCoCo*, depicted on Figure 8.1. Mapping the RL framework to the Rate adaptation problem, we get the following scenario: The agent is an RL algorithm that, at every time interval $\Delta t$, predicts the available bandwidth and sends this information to the environment as an action. The environment runs a network simulation of RTP traffic between a sender and a receiver, given a bandwidth trace file, and based on the action, adjusts the sending rate. Then the simulation runs with that sending rate for a time interval $\Delta t$ and spits out a list of packets. From these packets we calculate a set of network statistics (such as average delay, loss ratio etc.) - the state. Based on the state we also calculate an appropriate reward. For instance, if the loss ratio is high, the reward is very low. The state and reward are sent to the agent, that based on them, adjusts its policy.

For the network simulation and the agent's interaction with it, we use the OpenNetLab [126] framework, which provides a plug-and-play *gym*[2] environment for training RL algorithms to the task of congestion control in RTC, using an *ns-3* event-driven network simulator and Chrome's WebRTC. We make some changes to the environment to account for different states and rewards.

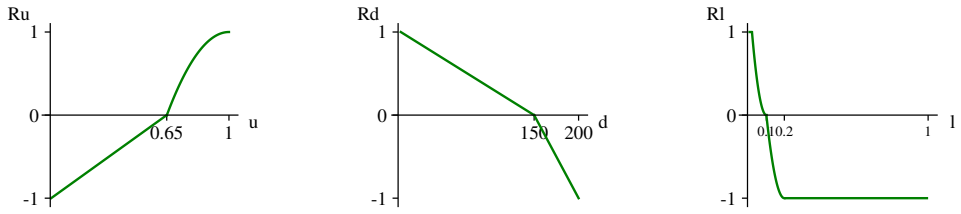
After the model is trained, *ReCoCo* is envisioned to work in real-time, on the sender side of an RTC communication. The system in that case is depicted on Figure 1.3 in Chapter 1. Namely, the receiver calculates the needed network statistics for the state and sends them to the sender via RTCP. Then, the sender runs the trained model and adjusts the sending rate accordingly.

### 8.3.1 State space

The states are histories of network statistics. In our implementation, we use a statistics vector $\overrightarrow{v_t}$ which has four components, calculated for the duration of $\Delta t$:

1. Receiving rate $r_t$
2. Average delay $d_t$
3. Loss ratio $l_t$
4. Last action taken $a_{t-1}$

[2] https://github.com/OpenNetLab/gym




(a) $R_u$ (b) $R_d$ (c) $R_l$

Fig. 8.2 Functions describing the reward.

The statistics vector can be defined as:

$$\overrightarrow{v_t} = (r_t, d_t, l_t, a_{t-1}) \tag{8.1}$$

We can then use multiple past instances of the statistics vector as the state. Thus, at time $t$, the state $\mathbf{S_t}$ is defined as:

$$\mathbf{S_t} = (\overrightarrow{v_t}, \overrightarrow{v_{t-1}}, ...\overrightarrow{v_{t-x}}) \tag{8.2}$$

where $x$ is the number of time intervals $\Delta t$ we consider in the past.

In our experiments, we fix $\Delta t$ to $200ms$ and vary the parameter $x$, by setting it to 5 (delayed states) or setting it to 0 (non-delayed states). In the latter case, $\mathbf{S_t} = \overrightarrow{v_t}$. We normalize all state values in the interval [0,1], as we find this is vital for many RL algorithms to learn.

### 8.3.2 Reward design

The reward is the most important driver of RL algorithms, so we take careful consideration in designing it. Our reward function incorporates three components: bandwidth utilization, delay and loss ratio. The reward is always normalized to the interval [-1,1]. The functions describing the different components of the reward are depicted on Figure 8.2.

**Bandwidth utilization reward component ($R_u$).** It expresses how well the algorithm is using the available bandwidth. Let $u_t$ be the bandwidth utilization:

$$u_t = r_t/B_t, \quad u_t \in [0,1] \tag{8.3}$$

where $r_t$ is the receiving rate and $B_t$ is the bandwidth in that $\Delta t$ time bin. Then $R_u$ is defined as:

$$R_u = \begin{cases} 1.538u_t - 1, & \text{if } 0 < u_t \leq u_{thres} \\ -8.2(u_t - 1)^2 + 1, & \text{if } u_{thres} < u_t \leq 1. \end{cases} \tag{8.4}$$

where $u_{thres} = 0.65$.

$R_u$ is depicted on Figure 8.2a. The closer the bandwidth utilization $u_t$ is to 1, the higher the reward. $u_{thres}$ is a threshold that distinguishes between a negative and positive reward. Note that if $u_t$ is higher than 1, it means that the receiving rate is higher than the available bandwidth (over-utilization). In that case we force the whole reward $R_t$ to -1.

**Delay reward component ($R_d$).** It expresses how acceptable the one-way delay between the sender and the receiver is. Let $d_t$ be the average one-way delay in the time bin $\Delta t$. Then $R_d$ is defined as:

$$R_d = \begin{cases} -0.00667d_t + 1, & \text{if } 0 < d_t \leq 150 \\ -0.02d_t + 3, & \text{if } 150 < d_t \leq 200. \end{cases} \tag{8.5}$$

The equation is depicted on Figure 8.2b. We design $R_d$ according to the G.114 recommendation for one-way transmission time [131], which states that if delays were kept below 150 ms, then most real-time applications would not be significantly affected. If the delay is more than 200ms, it gets a reward of -1, since we aim for a low-latency algorithm.

**Loss ratio reward component ($R_l$).** It expresses how well the algorithm is doing in terms of losses. Let $l_t$ be the loss rate in the time bin $\Delta t$. Then $R_l$ is defined as:

$$R_l = \begin{cases} 1, & \text{if } 0 \leq l_t \leq 0.02 \\ 156(l_t - 0.1)^2, & \text{if } 0.02 < l_t \leq 0.1 \\ 100(l_t - 0.2)^2 - 1, & \text{if } 0.1 < l_t \leq 0.2 \\ -1, & \text{if } 0.2 < l_t \leq 1. \end{cases} \tag{8.6}$$

The equation is depicted on Figure 8.2c. To design the first two thresholds in (8.6) we rely on the GCC thresholds for acceptable loss rate [132].

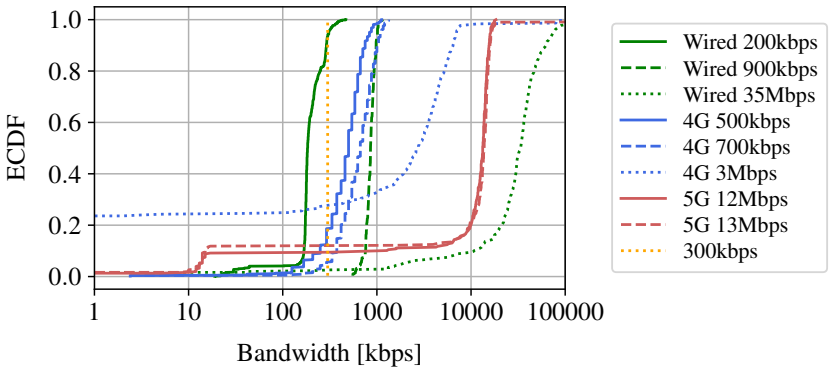



Fig. 8.3 Bandwidth distribution of different trace files.

**Final reward equation** ($R_t$). Combining all the reward components together, the final reward at a time step $\Delta t$ is:

$$R_t = \begin{cases} 0.333R_u + 0.333R_d + 0.333R_l, & \text{if } l_t > 0 \\ 0.4R_u + 0.4R_d + 0.2R_l, & \text{otherwise.} \end{cases} \tag{8.7}$$

Since losses are a rare event, we aim to mitigate the effect of a positive reward from the loss component. Thus, we decrease the weight of $R_l$ if the loss ratio is 0. In addition, we force $R_t$ to 1 if all these conditions apply: the loss ratio is below 0.02, the delay is below 30 and the bandwidth utilization is higher than 0.9.

## 8.4 Experimental setup

In this section, we outline all the experiments conducted to train *ReCoCo*. We first describe the traffic traces used for training, then the process of training and validation, the algorithms and other parameters in play and finally the performance metrics used to evaluate the employed models.

### 8.4.1 Dataset

To train the algorithm we use 9 trace files that specify the channel bandwidth in time. The trace files are open data by OpenNetLab [126]. Their distributions are depicted on Figure 8.3. Three traces represent a wired channel with different bandwidth

magnitudes (green lines), three traces represent a 4G cellular channel (blue lines), two represent a 5G cellular channel (red lines) and one is a constant trace at 300 kbps. The traces have different duration, from a minimum of 60 seconds to a maximum of 223 seconds, with a mean duration of 88.5 seconds. While training, the simulator goes through the traces many times. We believe the traces contain enough variability to represent many different network conditions.

### 8.4.2 Employed algorithms

We employ three different algorithms:

1. Soft Actor-Critic - SAC [133]
2. Twin-delayed DDPG - TD3 [134]
3. Proximal Policy Optimization - PPO [135]

They are all Deep RL algorithms, with the actor-critic architecture (see Section 8.2 and left-hand side of Figure 8.1). However, they all employ some kind of optimization to the actor-critic paradigm. SAC is an Off-Policy Maximum Entropy algorithm with a Stochastic Actor. Its key feature is that it is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. TD3 is an Off-policy algorithm that uses clipped double Q-learning (two critic networks), a delayed policy update and target policy smoothing. The main idea behind PPO is that after an update, the new policy should be not too far from the old policy. Thus it uses clipping to avoid large updates. We use the implementation of these algorithms in the Python library *Stable Baselines 3*[3].

### 8.4.3 Configuration parameters

Since Deep RL is very dependent on parameters and results can change considerably, we try a myriad of different configurations. One parameter we vary is $x$ from Equation 8.2. We either set it to 0 (non-delayed states) or to 5 (delayed states by five $\Delta t$). We try two versions of the algorithm hyperparameters - one is the default suggested by *Stable baselines 3* and another is tuned hyperparameters for an RL

[3]https://stable-baselines3.readthedocs.io/

Table 8.1 Hyperparameters of algorithm configurations.

| Parameter | Algorithm configuration | | |
|---|---|---|---|
| | TD3 not tuned | TD3 tuned | SAC tuned |
| Policy | MlpPolicy | MlpPolicy | MlpPolicy |
| Learning rate | 0.001 | 0.001 | 0.0003 |
| Gamma | 0.99 | 0.98 | 0.9999 |
| Learning starts | 100 | 10000 | 0 |
| Action noise | Normal | Ornstein-Uhlenbeck | - |
| Buffer size | 1000000 | 1000000 | 50000 |
| Batch size | 100 | 100 | 512 |
| tau | 0.005 | 0.005 | 0.01 |
| Gradient steps | -1 | -1 | 32 |
| NN architecture | [400, 300] | [400, 300] | [64,64] |

environment similar to ours (the Cartpole environment). The tuning is provided by *Stable Baselines Zoo*[4]. We call these configurations *not tuned* and *tuned*, respectively. The hyperparameters of the best configurations are shown on Table 8.1. We also conduct a few experiments with not-normalized state values and notice that this way the algorithms are completely unable to learn. Thus, we always normalize the states to the interval [0,1] and the reward to [-1,1]. The actions are not normalized (they are direct values of the predicted bandwidth in *bps*).

### 8.4.4 Training and validation strategy

For each configuration *(trace, algorithm, delayed states/not, tuned hyperparameters/not)*, we employ training of $100k$ steps. One step is equal to $\Delta t$ in simulation time. Every $10k$ steps, we save the model and perform validation on the same environment (same trace file), by observing the average reward collected on the whole trace file. This procedure helps us choose the best configuration for each trace. Namely, the configuration that performs better has a higher average reward in the final few tests. When two or more configurations perform similarly enough in terms of average reward, we evaluate them using the QoE metrics defined in Section 8.4.5.

An example of validation during training of a few configurations is shown on Figure 8.4. Here the environment is the trace *Wired 900kbps*. The lines are an average of 3 runs with different random seeds. The blue lines represent two configurations

[4]https://github.com/DLR-RM/rl-baselines3-zoo

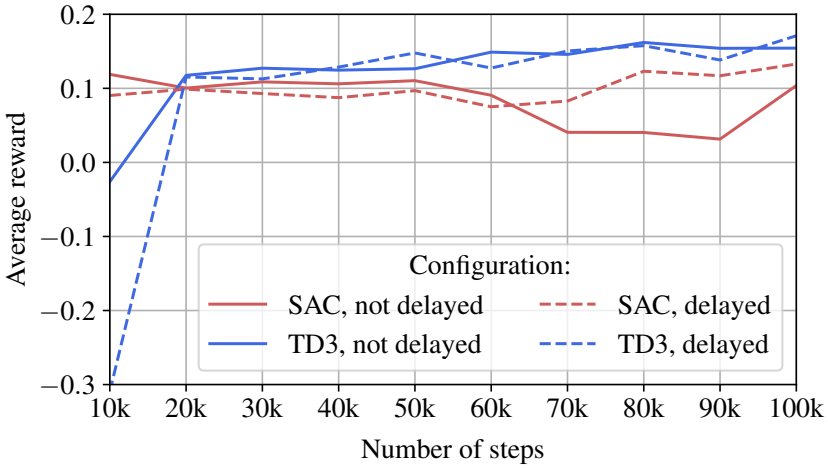



Fig. 8.4 Validation during training of some experiment configurations on *Wired 900kbps*.

with the TD3 algorithm, while the red lines with SAC. All configurations on the plot consider the algorithms with tuned parameters. We observe that the average reward grows as training progresses. Notice that the first point is already after $10k$ steps of training. Since at least three options show similar performance, we choose the best one in terms of QoE metrics, which turns out to be *(SAC, delayed)* - red dashed line.

### 8.4.5 Performance metrics

To evaluate the goodness of our models on the network traces, we employ the Quality of Experience (QoE) scores defined by OpenNetLab [126]. The QoE score is composed of three components: (i) Receiving rate QoE, (ii) Delay QoE and (iii) Loss QoE. The Receiving rate QoE is given by:

$$QoE_{rr} = 100 \times U \tag{8.8}$$

where $U$ is the median bandwidth utilization in the trace (a median of all $u_t$ from Equation 8.3). We clip all $u_t$ values that are larger than 1 to 1, since they would skew the $QoE_{rr}$ towards a good score, while the agent is sending at a higher rate than the available bandwidth, thus introducing considerable delay or losses.

The delay QoE is defined as:

$$QoE_{delay} = 100 \times \frac{d_{max} - d_{95th}}{d_{max} - d_{min}} \tag{8.9}$$

where $d_{max}$, $d_{min}$ and $d_{95th}$ are taken from the distribution of the delay ($d_t$) throughout the whole trace. Note that this score takes into account only delay variation and not absolute values. However, we mitigate high delays using the reward.

The loss QoE equation is the following:

$$QoE_{loss} = 100 \times (1 - L) \tag{8.10}$$

where $L$ is the mean of all $l_t$ in a trace. Note that even when this score is 80, which seems high, it means 20% of losses, which is a low score. The final QoE metric is a weighted average of all QoE components:

$$QoE = 0.33QoE_{rr} + 0.33QoE_{delay} + 0.33QoE_{loss} \tag{8.11}$$

All QoE components and the final score are on a scale of [0, 100].

## 8.5 Experimental Results

In this section, we present the results of training *ReCoCo*. We first discuss the QoE when training a separate model for each trace, with their best configuration. However, in a real network scenario we need more versatile models that are able to cover a variety of network conditions. Thus, we discuss the transferability of these models to other traces and the performance of a single model trained with curriculum learning.

### 8.5.1 Best configuration results

In Section 8.4.3 we outline all the different configuration combinations we try in our experiments. Here we present only the results of the best-performing configurations.

Table 8.2 shows the QoE of *ReCoCo* and *GCC*, for all traces. We find that SAC and TD3 exhibit much better performance than PPO in general, in every possible combination, thus PPO does not appear in the table. The algorithms prefer delayed

Table 8.2 QoE of best-performing configurations for each trace.

| Trace | Configuration | QoE Receiving rate | | QoE Delay | | QoE Loss | | Overall QoE | |
|---|---|---|---|---|---|---|---|---|---|
| | | ReCoCo | GCC | ReCoCo | GCC | ReCoCo | GCC | ReCoCo | GCC |
| **Wired 200kbps** | TD3, not delayed, not tuned | 85.60 | 93.78 | 98.06 | 82.47 | 100.00 | 100.00 | 94.46 | 91.99 |
| **Wired 900kbps** | SAC, delayed, tuned | 75.88 | 92.94 | 83.79 | 38.37 | 100.00 | 100.00 | 86.47 | 77.03 |
| **Wired 35Mbps** | TD3, delayed, not tuned | 20.52 | 8.66 | 66.77 | 66.51 | 99.13 | 99.47 | 62.08 | 58.16 |
| **4G 500kbps** | TD3, delayed, tuned | 69.98 | 78.81 | 91.52 | 76.25 | 99.78 | 99.83 | 87.01 | 84.88 |
| **4G 700kbps** | TD3, delayed, not tuned | 73.67 | 59.77 | 86.67 | 89.76 | 100.00 | 99.84 | 86.69 | 83.04 |
| **4G 3Mbps** | SAC, delayed, tuned | 81.91 | 12.32 | 91.07 | 96.79 | 83.51 | 87.38 | 85.41 | 65.43 |
| **5G 12Mbps** | TD3, delayed, not tuned | 55.95 | 4.19 | 72.32 | 85.38 | 94.09 | 99.82 | 74.04 | 63.07 |
| **5G 13Mbps** | TD3, not delayed, not tuned | 40.54 | 5.21 | 61.54 | 55.83 | 91.14 | 99.99 | 64.34 | 53.62 |
| **300kbps** | TD3, delayed, tuned | 81.88 | 99.21 | 79.83 | 10.49 | 100.00 | 100.00 | 87.15 | 69.83 |

states, which means that information on the network conditions in the near past proves useful. Out of 9 traces, *ReCoCo* exhibits better $QoE_{rr}$ in 5, with a significant improvement for 5G traces with variable bandwidth. *ReCoCo* has a higher $QoE_{delay}$ for 6 traces. Usually where one algorithm performs well on the receiving rate, it exhibits higher delay and vice-versa. Interestingly, *ReCoCo* shows much better results for *Wired 900kbps* and *300kbps*, which are traces with more stable bandwidth. As to $QoE_{loss}$, *GCC* exhibits slightly better results. *ReCoCo* prefers a slightly higher loss rate over a very high delay, which is not the case for *GCC*. Looking at overall QoE, where all components are given the same weight, *ReCoCo* outperforms *GCC* for all traces.

Figure 8.5 summarizes the results of Table 8.2 in a scatterplot. The x-axis is $QoE_{rr}$, the y-axis $QoE_{delay}$ and the size of the circles represents the $QoE_{loss}$. The blue circles are *ReCoCo* and the red circles *GCC*. We see that *GCC* strongly favours either delay or receiving rate (circles either on the top left corner or far right on the plot). It has decent $QoE_{loss}$ in both cases, however it rarely optimizes for both metrics. This is expected, since it employs a controller based on delay variation and loss rate. Instead, *ReCoCo* shows many circles in the top right corner of the plot, with only some lingering with a slightly worse $QoE_{rr}$ (still never lower than 60). This is where we can see the value of the reward function, which optimizes for all three of these metrics.

Figure 8.6 shows the sending rate vs. bandwidth in time, for three of the traces: *Wired 200kbps* (Figure 8.6a), *4G 700kbps* (Figure 8.6b) and *300 kbps* (Figure 8.6c). The trace bandwidth is the orange line, *GCC* is depicted with a red line and *ReCoCo* with a blue line. Another version of *ReCoCo*, discussed in Section 8.5.3, is the pink line. *Wired 200kbps* is a trace with many spikes in bandwidth. Here *ReCoCo* is more conservative, while *GCC* manages to utilize the high bandwidth during

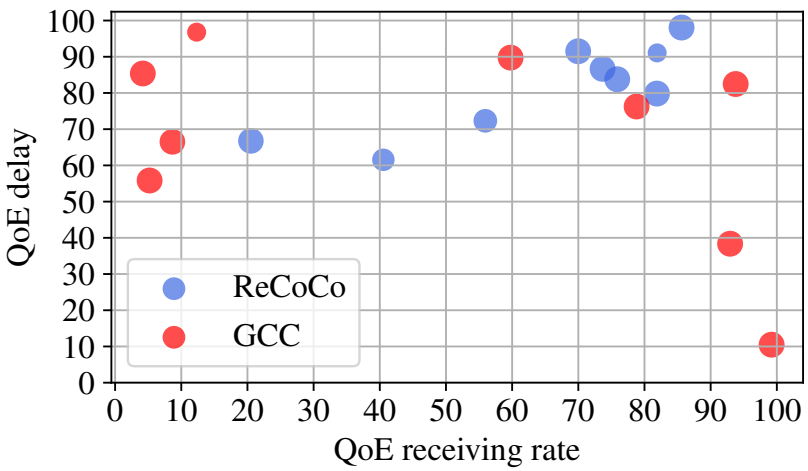


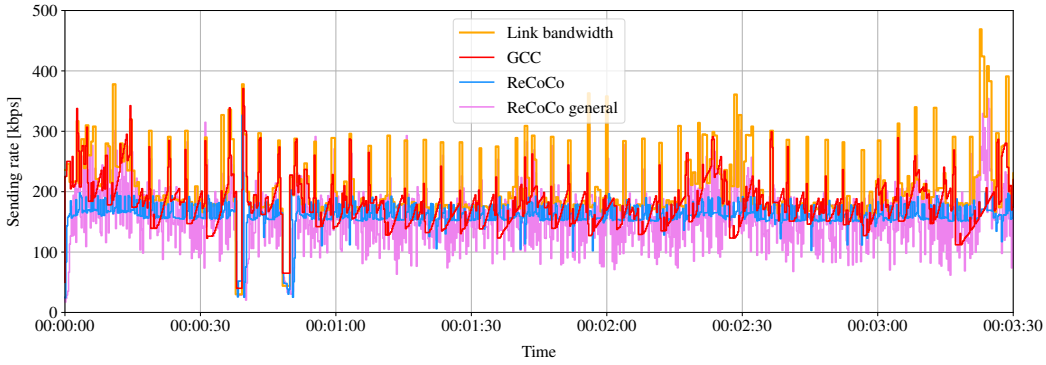

Fig. 8.5 Comparison of QoE between *ReCoCo* and *GCC* for each trace. The size of the circle indicates the loss QoE.

some of the spike periods, so it has a better $QoE_{rr}$. However, it often ends up over-utilizing it, so it has a worse $QoE_{delay}$ than *ReCoCo*. In the very variable bandwidth scenario in Trace *4G 700kbps*, we can definitely see *ReCoCo* prevail in bandwidth utilization, with very little penalty to the $QoE_{delay}$ (just 3.09 units lower than *GCC*'s) and virtually no losses (0.16% in contrast to no loss for *GCC*). Moreover, we notice that *ReCoCo* performs better on a totally stable bandwidth (*300kbps* trace). Here the *GCC* mechanism clearly struggles - we find repeated instances of slow increase of bandwidth, then over-utilization and then high drops. Thus it ends up with a very poor $QoE_{delay}$ score.

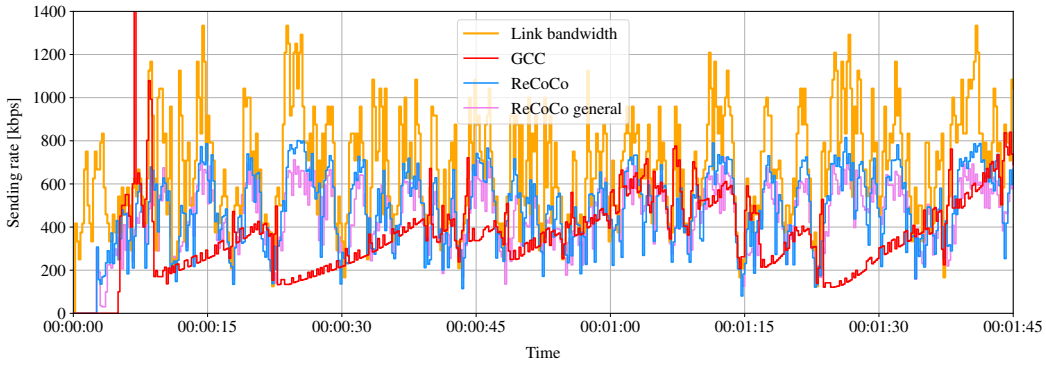
### 8.5.2 Model cross-trace performance

In this subsection we discuss the performance of the best models for each trace on the other traces. This speaks to the ability for the models trained on one trace to generalize to other traces.

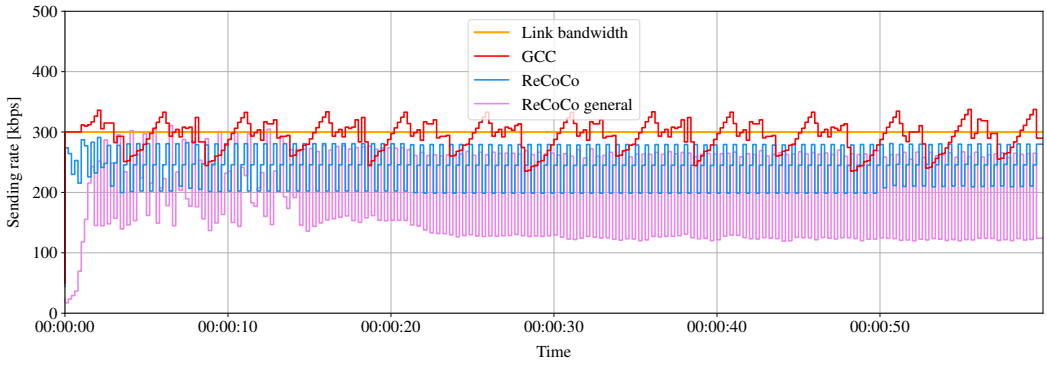
Figure 8.7 shows four heatmaps - one for each of the three QoE components and one for overall QoE, of the QoE scores when trained on one trace and tested on another. The y-scale represents the trace the agent has been trained on, while the x-sclae the trace it has been tested on. The diagonal holds the results already presented in section 8.5.1 Green indicates good QoE scores and red poor ones.




(a) Trace *Wired 200kbps*




(b) Trace *4G 700kbps*




(c) Trace *300 kbps*

Fig. 8.6 Examples on *ReCoCo* v.s *GCC* sending rate on different traces.

We can see that in general, performance is bad when testing on high bandwidth and high variability traces (*Wired 35Mbps*, *5G 12Mbps* and *5G 13Mbps*), especially for $QoE_{rr}$. This is because they operate on much higher bandwidth than the other traces and have higher bandwidth standard deviation. However, they show better performance when trained and tested among each other. Moreover, models trained


(a) Receiving rate QoE

(b) Delay QoE

(c) Loss QoE

(d) Overall QoE

Fig. 8.7 QoE components when training on one trace and testing on another.

on *Wired 900kbps*, *4G 500kbps* and *4G 700kbps* also yield good results between each other. This means that training different models based on bandwidth ranges can be a good strategy for generalization. The stable bandiwdth trace, *300kbps* proves good for model training, but hard for other models to perform well on it. Thus, another channel characteristic to take into account when generalizing could be bandwidth variability.

### 8.5.3 Curriculum learning

Despite some transfer ability, we still prefer to have a one-model-fits-all solution. In this subsection we show results when training one model on all traces. When using Deep RL with many different environments, such as traces with various bandwidth profiles, it is very hard to train one model that performs well in all of them. How we introduce the environments to the agent becomes key [136]. Curriculum learning is a

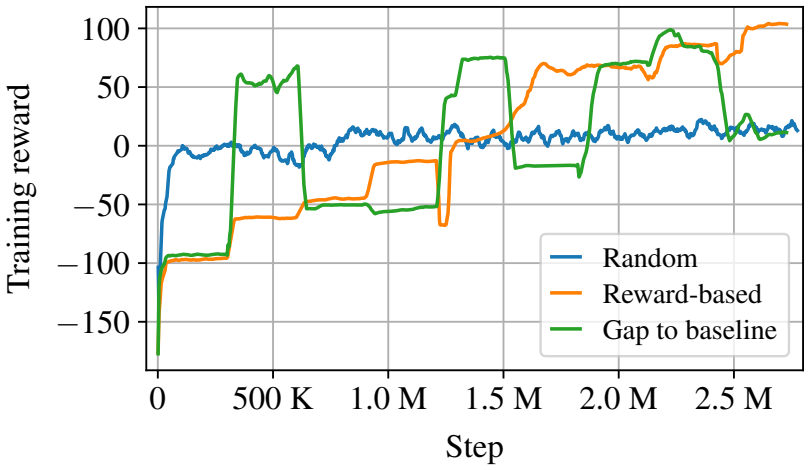



Fig. 8.8 Cumulative reward during training.

concept where, during training, one gradually increases the difficulty level of training environments, to resemble how humans learn more complex concepts [137]. It allows the RL model to make steady progress and reach good performance. Curriculum learning has been proven to improve generalization and asymptotic performance.

Inspired by this, we try three different ways of introducing the environments to train a single model:

1. **Random**: Sample training environments at random. This is the traditionally used approach.
2. **Reward-based**: Start with the trace that obtains the lowest average reward when trained on itself (*4G 3Mbps*) and order in ascending order. We hope to imitate a growing training reward.
3. **Gap-to-baseline**: Easiest to hardest environment based on how much better their QoE score is against *GCC*, when trained on themselves (the trace with highest delta goes first). A concept introduced by [136].

All the traces are trained with one configuration - using the algorithm TD3, with delayed states and not tuned (using the *Stable Baselines 3* default values).

Figure 8.8 shows the training reward, for all three training types. We run training for 2.7 Million steps, with 300$k$ steps per trace. The curves are averages of 3 runs. For both reward-based (orange) and gap-to-baseline (green) training, the plot clearly

Table 8.3 QoE of single models trained in different ways on all traces.

| Trace | QoE Receiving rate | | | QoE Delay | | | QoE Loss | | | Overall QoE | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Random | Reward-based | Gap | Random | Reward-based | Gap | Random | Reward-based | Gap | Random | Reward-based | Gap |
| **Wired 200kbps** | 73.38 | 76.75 | 79.58 | 97.31 | 97.99 | 95.55 | 100 | 100 | 100 | 90.14 | 91.49 | 91.62 |
| **Wired 900kbps** | 78.73 | 17.09 | 70.69 | 66.03 | 24.56 | 63.9 | 100 | 100 | 100 | 81.51 | 47.17 | 78.12 |
| **Wired 35mbps** | 0.55 | 0.43 | 1.93 | 68.51 | 66.67 | 72.99 | 99.52 | 99.53 | 99.57 | 56.14 | 55.49 | 58.1 |
| **4G 500kbps** | 65.56 | 31.06 | 75.01 | 90.68 | 93.83 | 89.56 | 100 | 100 | 100 | 85.33 | 74.89 | 88.1 |
| **4G 700kbps** | 64.12 | 23.06 | 69.02 | 89.6 | 93.8 | 89.4 | 100 | 100 | 100 | 84.49 | 72.22 | 86.05 |
| **4G 3mbps** | 13.69 | 5.16 | 9.75 | 85.03 | 87.75 | 95.55 | 89.99 | 86.42 | 84.74 | 62.84 | 59.72 | 63.28 |
| **5G 12mbps** | 1.66 | 1.14 | 4.73 | 80.24 | 50.06 | 70.74 | 100 | 100 | 100 | 60.57 | 50.35 | 58.43 |
| **5G 13mbps** | 1.53 | 1.07 | 4.36 | 62.06 | 47.56 | 55.92 | 100 | 100 | 99.97 | 54.48 | 49.49 | 53.36 |
| **300kbps** | 72.51 | 46.72 | 66.95 | 2.96 | 0 | 48.43 | 100 | 100 | 100 | 58.43 | 48.86 | 71.72 |

shows the change of trace, with the flat areas representing the training of each trace. In both approaches, the reward grows with time. For the random trace sampling training, the reward stabilizes very early on and only grows very slightly.

The QoE scores of the three resulting models are summarized on Table 8.3. To evaluate the QoE, we use the models trained by the mid-performer from the 3 runs, so as not to introduce a bias. Table 8.3 shows that, out of 9 traces, the model trained with the gap-to-baseline approach has superior performance for 6 traces, while the random-sampling approach for 3 traces (among which the hard case of the 5G traces). If we average out the overall QoE scores for all traces, gap-to-baseline outperforms the reward-based approach by 11 QoE units and the random approach by 1.65 QoE units.

Next, we compare the general models with the specialized models and *GCC*. Figure 8.6 shows an example of the sending rate for three traces: *Wired 200kbps*, *4G 700kbps* and *300 kbps*, where the pink line represents the general model trained with the gap-to-baseline methodology. We notice that, for this trace, the sending rate of the general model is a little more conservative than the specialized one, but still very comparable. For a full comparison of QoE scores, we look at both Table 8.3 and Table 8.2. Comparing the specialized models for each trace with the single model solutions, for overall QoE scores, we observe a performance penalty of 10.4 QoE units on average for the random-sampling single model and 8.95 QoE units for the gap-to-baseline model. This is expected, since a specialized model for each trace would always outperform a general one. Comparing with *GCC*, the gap-to-baseline model is more conservative with the sending rate, so it has worse scores for $QoE_{rr}$, but better for $QoE_{delay}$ and $QoE_{loss}$. On average, it shows an improvement in overall QoE over *GCC* of 0.2 units. The random-sampling model shows a performance drop over *GCC* of 1.46 QoE units.

The results from the curriculum training show that, with a wide variety of training data, if training is performed smartly, we could obtain a decent one-model-fits-all solution.

## 8.6 Takeaways

In this Chapter, we discussed *ReCoCo*, a Reinforcement learning-based Rate controller for real-time applications. *ReCoCo* gains information about the network conditions at the receiver-side, such as receiving rate, one-way delay and loss rate and predicts the available bandwidth in the next time bin. We showed its performance, both in a specialized and generalized version and found it outperforms current widely adopted CC heuristics, namely GCC. We find that a good trade-off between a specialized model and a general model could be to train one model per bandwidth range.

As future work, we would like to try the solutions outside a simulated environment, in a real network, covering a larger variety of bandwidths. We would also like to evaluate *ReCoCo*'s ability to continue training in the wild, when already implemented, since this is one of the main strengths of reinforcement learning.

We believe that the story of training *ReCoCo* provides valuable lessons for the Reinforcement learning for networking community. First, it shows that RL is a very promising technique for classical control tasks, such as Rate control. With a good reward design it can capture very complicated relationships between network variables, such as receiving rate, delay and packet loss, which cannot be captured by heuristic approaches. Second, it is very adaptable to dynamic networks, where conditions change in a matter of milliseconds. However, we find it is complex to train a single RL model that performs well in very different environments. Some channels may even present unsolvable environments for the RL agents. The challenge in using RL for congestion control lies in the generalization and transferability of the models to different network types.

We hereby thank the researchers from OpenNetLab for their open-source simulator that enabled a (simple) training ground for us to conduct these experiments. They are an excellent example of bridging the gap between open software and open data and networking research.

# Chapter 9

# Conclusions

The research presented in this thesis addresses the topics of monitoring and control of Real-Time Communication traffic on the Internet, with the goal of QoE improvement. Our case study is based on traffic from video-conferencing applications. We present an overview of the network protocols and mechanisms used by RTC applications today and then a myriad of ways to improve the workings of these applications, both on the network layer and on the application layer. In this conclusive chapter, we discuss the usefulness and potential of our approaches and the greatest challenges they face.

## 9.1 On the potential of ML for network monitoring and control

In this thesis, we apply classical ML classification for *observability* on the network layer. We show that ML allows us to understand a lot about the underlying workings of RTC applications, just by observing the traffic, without any contact with proprietary application servers. Even without reading packet headers, by using statistical features on the size of the packets, inter-arrival times or relationship between the packets, ML can uncover valuable information such as: "On this network there is a client talking on Webex and they're sending 720p video". We obtain high classification accuracy in both RTC application retrieval and media type classification and prove that ML holds huge potential for problems related to network monitoring.

Moreover, using statistical features of the traffic brings a few advantages. First, it characterizes up to one second of traffic with only a few numbers, so our approaches can be used at high speeds. Second, it is robust to packet encryption. Even if the packet header becomes encrypted one day, our approaches can still be applied, with minimal modifications. This way, we also preserve the privacy of the users. We use interpretable models which would allow network administrators to better understand the classification results.

In terms of network *control*, we demonstrate that using the information from the network layer ML classifiers, the network can bestow effective management policies for control of RTC traffic. Some strategies we suggest involve path selection and bandwidth allocation. We also present a Reinforcement learning-based control mechanism at the application layer and show that it can effectively do rate control for RTC applications. In addition, our proposed RL algorithm outperforms the current industry standard, which is based on heuristics, for our dataset.

## 9.2 On the adoption of ML in production networks

Regardless of its potential, the deployment of ML-based systems in real production networks is still limited. There are two main reasons for this. First, machine learning does not provide the sort of performance guarantees that are customary in computer science. Results are volatile, hard to generalize to similar problems and the models are not always interpretable. ML algorithms need to be trained on vast data in order to exhibit reliable and safe performance, such as that desirable for network operations. This is one of the limitations of our work. However, researchers are actively working to mitigate these risks. In both Chapter 7 and 8, we included experiments on the transferability of the models to new data and training general models that are appropriate for more tasks. Moreover, not all tasks in networking are critical and require such strict performance guarantees. Improving the quality of an already existing network connection is a non-critical task and with careful engineering, ML-based decision making can be made fault-tolerant, as we discuss in Chapter 4.

Another reason is that the models are computationally complex and cannot be applied to widespread network hardware, that is optimized for speed and not for computation. However, this issue can be somewhat overcome by using programmable

switches and implementing simple ML algorithms, such as Decision trees and SVMs. Indeed, efforts have emerged in the literature that apply even complex ML models like Neural networks to different programmable switches on the market (see Chapter 4). There is also a lack of expertise in ML in network administrators. Indeed, this thesis shows the amount of research necessary to understand how to effectively apply ML to networking problems related to RTC traffic. Chapter 4 outlines some efforts that tackle this problem as well.

The networking community is still sceptical of the use of ML for network *control*, for the aforementioned reasons. Although this is understandable, we believe that ML is mature enough to be very effectively used for network *monitoring*. Network practitioners could greatly benefit from insights and underlying patterns of the traffic given by ML.

To encourage the adoption of ML in real networks, the ML for networking scientific community also needs more open code and datasets, where researchers can reproduce results easily and perform benchmarking of their algorithms on the same datasets and against other algorithms. This would make the field grow even more rapidly and would foster safer adoption of ML. For this reason, we also consider the publishing of our code and datasets for each chapter a significant contribution of the thesis.

## 9.3 Ethics and vision

In this thesis, we suggest network management techniques that may prioritize RTC traffic over other traffic categories. This goes against the general concept of Net neutrality. However, our approaches are developed keeping the users' best interests in mind - the same goal as net neutrality. Moreover, they can still be useful for private networks, like campus or enterprise networks. What's more, RTC applications are not bandwidth-heavy like other common traffic, such as video streaming, so even with their prioritization, they would not take up a considerable amount of bandwidth.

We believe that RTC applications are here to stay, especially with the rise of remote work and the increasingly dynamic lifestyle of people. We expect that the number of available RTC applications on the market will rise and existing giants (e.g. Zoom) will continue to optimize their application in terms of encoding,

bandwidth, redundant streams and congestion control algorithms. On a parallel road, standardization efforts such as WebRTC will get richer with available protocols and offer various ways to build applications for the browser.

We also think that in a data-oriented society, the wider adoption of ML in networking is inevitable. Networks will become more and more softwarized, data-driven and smart. In this scenario, approaches such as those presented in this thesis will become very useful and important.

# References


[1] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia. Five years at the edge: Watching internet from the isp network. *IEEE/ACM Trans. on Networking*, 28(2):561–574, 2020.

[2] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 1–18, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Thomas Favale, Francesca Soro, Martino Trevisan, Idilio Drago, and Marco Mellia. Campus traffic and e-Learning during COVID-19 pandemic. *Computer Networks*, 176:107290, 2020.

[4] Nicholas Bloom. How working from home works out. *Stanford Institute for economic policy research*, 8, 2020.

[5] Ymer Gurra. Estimating qoe from qos in real-time traffic: A machine learning approach. Master's thesis, Politecnico di Torino, 2021.

[6] Tailai Song. Machine learning for predicting losses in the networks. Master's thesis, Politecnico di Torino, 2022.

[7] Tailai Song, Dena Markudova, Gianluca Perna, and Michela Meo. Where did my packet go? real-time prediction of losses in networks. In *Accepted. To appear in ICC 2023-IEEE International Conference on Communications*. IEEE, 2023.

[8] Amir Ligata, Erma Perenda, and Haris Gacanin. Quality of experience inference for video services in home wifi networks. *IEEE Communications Magazine*, 56(3):187–193, 2018.

[9] Nanditha Rao, A Maleki, F Chen, Wenjun Chen, C Zhang, Navneet Kaur, and Anwar Haque. Analysis of the effect of qos on video conferencing qoe. In *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pages 1267–1272. IEEE, 2019.

[10] Ashkan Nikravesh, Qi Alfred Chen, Scott Haseley, Xiao Zhu, Geoffrey Challen, and Z Morley Mao. Qoe inference and improvement without end-host control. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 43–57. IEEE, 2018.

[11] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the" s" in https. In *Proc. of the 10th ACM International on Conf. on emerging Networking Experiments and Technologies*, pages 133–140, 2014.

[12] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, 2018.

[13] Dunja Vucic and Lea Skorin-Kapov. The impact of packet loss and google congestion control on qoe for webrtc-based mobile multiparty audiovisual telemeetings. In *International Conference on Multimedia Modeling*, pages 459–470. Springer, 2019.

[14] Stefan Holmer, Henrik Lundin, Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. A Google Congestion Control Algorithm for Real-Time Communication. Internet-Draft draft-ietf-rmcat-gcc-02, Internet Engineering Task Force, July 2016. Work in Progress.

[15] Dario Bonfiglio, Marco Mellia, Michela Meo, Dario Rossi, and Paolo Tofanelli. Revealing skype traffic: when randomness plays with you. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 37–48, 2007.

[16] Dario Bonfiglio, Marco Mellia, Michela Meo, and Dario Rossi. Detailed analysis of skype traffic. *IEEE Transactions on Multimedia*, 11(1):117–127, 2008.

[17] Dario Bonfiglio, Marco Mellia, Michela Meo, Nicolo Ritacca, and Dario Rossi. Tracking down skype traffic. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 261–265. IEEE, 2008.

[18] Saikat Guha and Neil Daswani. An experimental study of the skype peer-to-peer voip system. Technical report, Cornell University, 2005.

[19] Sven Ehlert, Sandrine Petgang, Thomas Magedanz, and Dorgham Sisalem. Analysis and signature of skype voip session traffic. *4th IASTED International*, 2006.

[20] Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[21] Tobias Hoßfeld and Andreas Binzenhöfer. Analysis of skype voip traffic in umts: End-to-end qos and qoe measurements. *Computer Networks*, 52(3):650–666, 2008.

[22] J Gregorio, B Alarcos, and A Gardel. Forensic analysis of telegram messenger desktop on macos. *International Journal of Research in Engineering and Science*, 6(8):39–48, 2018.

[23] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. Forensic analysis of telegram messenger on android smartphones. *Digital Investigation*, 23:31–49, 2017.

[24] Gandeva Bayu Satrya, Philip Tobianto Daely, and Muhammad Arif Nugroho. Digital forensic analysis of telegram messenger on android devices. In *2016 International Conference on Information & Communication Technology and Systems (ICTS)*, pages 1–7. IEEE, 2016.

[25] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. Enabling passive measurement of zoom performance in production networks. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 244–260, 2022.

[26] Abdullah Azfar, Kim-Kwang Raymond Choo, and Lin Liu. Android mobile voip apps: a survey and examination of their security and privacy. *Electronic Commerce Research*, 16(1):73–111, 2016.

[27] O Karya, S Saesaria, and S Budiyanto. Rtp analysis for the video transmission process on whatsapp and skype against signal strength variations in 802.11 network environments. In *IOP Conference Series: Materials Science and Engineering*, volume 453, page 012062, 2018.

[28] Pongpisit Wuttidittachotti, Worawat Akapan, and Therdpong Daengsi. Comparison of voip-qoe from skype, line, tango and viber over 3g networks in thailand. In *2015 Seventh International Conference on Ubiquitous and Future Networks*, pages 456–461. IEEE, 2015.

[29] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. Video telephony for end-consumers: measurement study of google+, ichat, and skype. In *Proceedings of the 2012 Internet Measurement Conference*, pages 371–384, 2012.

[30] Chenguang Yu, Yang Xu, Bo Liu, and Yong Liu. "can you see me now?" a measurement study of mobile video calls. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1456–1464. IEEE, 2014.

[31] Tole Sutikno, Lina Handayani, Deris Stiawan, Munawar Agus Riyadi, and Imam Much Ibnu Subroto. Whatsapp, viber and telegram: Which is the best for instant messaging? *International Journal of Electrical & Computer Engineering (2088-8708)*, 6(3), 2016.

[32] N. Patel, S. Patel, and W. L. Tan. Performance comparison of whatsapp versus skype on smart phones. In *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–3, 2018.

[33] Hyunseok Chang, Matteo Varvello, Fang Hao, and Sarit Mukherjee. Can you see me now? a measurement study of zoom, webex, and meet. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 216–228, 2021.

[34] M. Finsterbusch, C. Richter, E. Rocha, J. Muller, and K. Hanssgen. A survey of payload-based traffic classification approaches. *IEEE Communications Surveys Tutorials*, 16(2):1135–1156, 2014.

[35] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE communications surveys & tutorials*, 10(4):56–76, 2008.

[36] Martino Trevisan, Idilio Drago, Marco Mellia, Han Hee Song, and Mario Baldi. What: A big data approach for accounting of modern web services. In *2016 IEEE Int. Conf. on Big Data (Big Data)*, pages 2740–2745. IEEE, 2016.

[37] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile encrypted traffic classification using deep learning. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–8. IEEE, 2018.

[38] Athula Balachandran, Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Srinivasan Seshan, Shobha Venkataraman, and He Yan. Modeling web quality-of-experience on cellular networks. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 213–224, 2014.

[39] Martino Trevisan, Idilio Drago, and Marco Mellia. Pain: A passive web performance indicator for isps. *Computer Networks*, 149:115–126, 2019.

[40] Irena Orsolic, Dario Pevec, Mirko Suznjevic, and Lea Skorin-Kapov. A machine learning approach to classifying youtube qoe based on encrypted network traffic. *Multimedia tools and applications*, 76(21):22267–22301, 2017.

[41] Pedro Casas, Alessandro D'Alconzo, Florian Wamser, Michael Seufert, Bruno Gardlo, Anika Schwind, Phuoc Tran-Gia, and Raimund Schatz. Predicting qoe in cellular networks using machine learning and in-smartphone measurements. In *Ninth International Conf. on Quality of Multimedia Experience*, pages 1–6. IEEE, 2017.

[42] Ignacio N Bermudez, Marco Mellia, Maurizio M Munafo, Ram Keralapura, and Antonio Nucci. Dns to the rescue: discerning content and services in a tangled web. In *Proceedings of the 2012 Internet Measurement Conference*, pages 413–426, 2012.

[43] Paweł Foremski, Christian Callegari, and Michele Pagano. Dns-class: immediate classification of ip flows using dns. *International Journal of Network Management*, 24(4):272–288, 2014.

[44] A. Finamore, M. Mellia, M. Meo, and D. Rossi. Kiss: Stochastic packet inspection classifier for udp traffic. *IEEE/ACM Transactions on Networking*, 18(5):1505–1515, 2010.

[45] A. S. Buyukkayhan, A. Kavak, and E. Yaprak. Differentiating voice and data traffic using statistical properties. In *2013 International Conference on Electronics, Computer and Computation (ICECCO)*, pages 76–79, 2013.

[46] M. Di Mauro and M. Longo. Revealing encrypted webrtc traffic via machine learning tools. In *2015 12th International Joint Conference on e-Business and Telecommunications*, volume 04, pages 259–266, 2015.

[47] T. Sinam, I. T. Singh, P. Lamabam, N. N. Devi, and S. Nandi. A technique for classification of voip flows in udp media streams using voip signalling traffic. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 354–359, 2014.

[48] Muhammad Shafiq, Xiangzhan Yu, and Asif Ali Laghari. Wechat traffic classification using machine learning algorithms and comparative analysis of datasets. *International Journal of Information and Computer Security*, 10(2-3):109–128, 2018.

[49] Petr Matousek, Ondrej Rysavy, and Martin Kmet. Fast rtp detection and codecs classification in internet traffic. *Journal of Digital Forensics, Security and Law*, 01 2014.

[50] M. C. S, S. H, and T. E. Somu. Network traffic classification by packet length signature extraction. In *2019 IEEE International WIE Conference on Electrical and Computer Engineering*, pages 1–4, 2019.

[51] E. Baştuğ, M. Bennis, and M. Debbah. A transfer learning approach for cache-enabled wireless networks. In *2015 13th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, pages 161–166, 2015.

[52] Z. Xu, D. Yang, J. Tang, Y. Tang, T. Yuan, Y. Wang, and G. Xue. An actor-critic-based transfer learning framework for experience-driven networking. *IEEE/ACM Transactions on Networking*, 29(1):360–371, 2021.

[53] Selim Ickin, Markus Fiedler, and Konstantinos Vandikas. Customized video qoe estimation with algorithm-agnostic transfer learning. *arXiv preprint arXiv:2003.08730*, 2020.

[54] Y. Hao, J. Yang, M. Chen, M. S. Hossain, and M. F. Alhamid. Emotion-aware video qoe assessment via transfer learning. *IEEE MultiMedia*, 26(1):31–40, 2019.

[55] Baochen Sun, Jiashi Feng, and Kate Saenko. Correlation alignment for unsupervised domain adaptation. In *Domain Adaptation in Computer Vision Applications*, pages 153–171. Springer, 2017.

[56] Cristina Rottondi, Riccardo di Marino, Mirko Nava, Alessandro Giusti, and Andrea Bianco. On the benefits of domain adaptation techniques for quality of transmission estimation in optical networks. *Journal of Optical Communications and Networking*, 13(1):A34–A43, 2021.

[57] Giovanna Carofiglio, Giulio Grassi, Enrico Loparco, Luca Muscariello, Michele Papalini, and Jacques 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*, pages 20–25, 2021.

[58] Boni Garcia, Micael Gallego, Francisco Gortazar, and Antonia Bertolino. Understanding and estimating quality of experience in webrtc applications. *Computing*, 101(11):1585–1607, 2019.

[59] Manuela Vaser and Sonia Forconi. Qos kpi and qoe kqi relationship for lte video streaming and volte services. In *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, pages 318–323. IEEE, 2015.

[60] P. Choudhury, K. R. Prasanna Kumar, G. Athithan, and S. Nandi. Analysis of vbr coded voip for traffic classification. In *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 90–95, 2013.

[61] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[62] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. Reinforcement learning for datacenter congestion control. *ACM SIGMETRICS Performance Evaluation Review*, 49(2):43–46, 2022.

[63] Wenting Wei, Huaxi Gu, and Baochun Li. Congestion control: A renaissance with machine learning. *IEEE Network*, 35(4):262–269, 2021.

[64] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.

[65] Salma Emara, Baochun Li, and Yanjiao Chen. Eagle: Refining congestion control by learning from the experts. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 676–685. IEEE, 2020.

[66] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[67] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2018.

[68] Kefan Xiao, Shiwen Mao, and Jitendra K Tugnait. Tcp-drinc: Smart congestion control based on deep reinforcement learning. *IEEE Access*, 7:11892–11904, 2019.

[69] Zhiyuan Xu, Jian Tang, Chengxiang Yin, Yanzhi Wang, and Guoliang Xue. Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1325–1336, 2019.

[70] Viswanath Sivakumar, Olivier Delalleau, Tim Rocktäschel, Alexander H Miller, Heinrich Küttler, Nantas Nardelli, Mike Rabbat, Joelle Pineau, and Sebastian Riedel. Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions. *arXiv preprint arXiv:1910.04054*, 2019.

[71] Xiaoqing Zhu, Rong Pan *, Michael A. Ramalho, and Sergio Mena de la Cruz. Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media. RFC 8698, February 2020.

[72] Ingemar Johansson and Zaheduzzaman Sarker. Self-Clocked Rate Adaptation for Multimedia. RFC 8298, December 2017.

[73] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. Measuring the performance and network utilization of popular video conferencing applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 229–244, 2021.

[74] Bo Wang, Yuan Zhang, Size Qian, Zipeng Pan, and Yuhong Xie. A hybrid receiver-side congestion control scheme for web real-time communication. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 332–338, 2021.

[75] Joyce Fang, Martin Ellis, Bin Li, Siyao Liu, Yasaman Hosseinkashi, Michael Revow, Albert Sadovnikov, Ziyuan Liu, Peng Cheng, Sachin Ashok, et al. Reinforcement learning for bandwidth estimation and congestion control in real-time communications. *arXiv preprint arXiv:1912.02222*, 2019.

[76] Haoyong Li, Bingcong Lu, Jun Xu, Li Song, Wenjun Zhang, Lin Li, and Yaoyao Yin. Reinforcement learning based cross-layer congestion control for real-time communication. In *2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 01–06. IEEE, 2022.

[77] Antonio Nistico, Dena Markudova, Martino Trevisan, Michela Meo, and Giovanna Carofiglio. A comparative study of rtc applications. In *2020 IEEE International Symposium on Multimedia (ISM)*, pages 1–8. IEEE, 2020.

[78] Henning Schulzrinne, Eve Schooler, Jonathan Rosenberg, and Mark J. Handley. SIP: Session Initiation Protocol. RFC 2543, March 1999.

[79] Ron Frederick, Stephen L. Casner, Van Jacobson, and Henning Schulzrinne. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, January 1996.

[80] Karl Norrman, David McGrew, Mats Naslund, Elisabetta Carrara, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711, March 2004.

[81] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Joel Weinberger. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, March 2003.

[82] Philip Matthews, Jonathan Rosenberg, and Rohan Mahy. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, April 2010.

[83] Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, April 2010.

[84] Marc Petit-Huguenin and Gonzalo Salgueiro. Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS). RFC 7983, September 2016.

[85] Mark J. Handley and Van Jacobson. SDP: Session Description Protocol. RFC 2327, April 1998.

[86] Christopher Allen and Tim Dierks. The TLS Protocol Version 1.0. RFC 2246, January 1999.

[87] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security. RFC 4347, April 2006.

[88] Christer Holmberg, Stefan Hakansson, and Goran Eriksson. Web Real-Time Communication Use Cases and Requirements. RFC 7478, March 2015.

[89] Martino Trevisan, Alessandro Finamore, Marco Mellia, Maurizio Munafo, and Dario Rossi. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *IEEE Commun. Mag.*, 55(3):163–169, 2017.

[90] Dena Markudova, Martino Trevisan, and Maurizio Munafo. RTC Pcap Cleaners. https://github.com/marty90/rtc_pcap_cleaners.

[91] IEEE Standards Association et al. Ieee standard for layer 2 transport protocol for time-sensitive applications in bridged local area networks. *IEEE Std 1722-2011*, 6:57, 2011.

[92] Jean-Marc Valin, Koen Vos, and Tim Terriberry. Definition of the Opus Audio Codec. RFC 6716, September 2012.

[93] Jean-Chrysostome Bolot, Mark J. Handley, Vicky Hardman, Isidor Kouvelas, and Colin Perkins. RTP Payload for Redundant Audio Data. RFC 2198, September 1997.

[94] Robert C Streijl, Stefan Winkler, and David S Hands. Mean opinion score (mos) revisited: methods and applications, limitations and alternatives. *Multimedia Systems*, 22(2):213–227, 2016.

[95] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *Ieee Network*, 32(2):92–99, 2017.

[96] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*, pages 25–33, 2019.

[97] Radhakrishna Kamath and Krishna M Sivalingam. Machine learning based flow classification in dcns using p4 switches. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10. IEEE, 2021.

[98] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: a data plane architecture for per-packet ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1099–1114, 2022.

[99] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-generating efficient data-plane ml pipelines for datacenter networks. *arXiv preprint arXiv:2206.05592*, 2022.

[100] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 296–309, 2020.

[101] Pedro Amaral, Joao Dinis, Paulo Pinto, Luis Bernardo, Joao Tavares, and Henrique S Mamede. Machine learning in software defined networks: Data collection and traffic classification. In *2016 IEEE 24th International conference on network protocols (ICNP)*, pages 1–5. IEEE, 2016.

[102] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, Chenmeng Wang, and Yunjie Liu. A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges. *IEEE Communications Surveys & Tutorials*, 21(1):393–430, 2018.

[103] Madeleine Glick and Houman Rastegarfar. Scheduling and control in hybrid data centers. In *2017 IEEE Photonics Society Summer Topical Meeting Series (SUM)*, pages 115–116. IEEE, 2017.

[104] International Telecommunication Union – Telecommunication Standardization Bureau. Recommendation ITU-T G.1070 – Opinion model for videotelephony applications. 2018.

[105] International Telecommunication Union – Telecommunication Standardization Bureau. Recommendation ITU-T G.107.1 – Wideband E-model. 2019.

[106] Markus Fiedler, Tobias Hossfeld, and Phuoc Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *Ieee Network*, 24(2):36–41, 2010.

[107] Tobias Hoßfeld, Poul E Heegaard, Martin Varela, Lea Skorin-Kapov, and Markus Fiedler. From qos distributions to qoe distributions: A system's perspective. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 51–56. IEEE, 2020.

[108] ZhiGuo Hu, HongRen Yan, Tao Yan, HaiJun Geng, and GuoQing Liu. Evaluating qoe in voip networks with qos mapping and machine learning algorithms. *Neurocomputing*, 386:63–83, 2020.

[109] Gianluca Perna, Dena Markudova, Martino Trevisan, Paolo Garza, Michela Meo, and Maurizio M Munafò. Retina: An open-source tool for flexible analysis of rtc traffic. *Computer Networks*, 202:108637, 2022.

[110] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014.

[111] Benoît Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, October 2004.

[112] Martino Trevisan, Alessandro Finamore, Marco Mellia, Maurizio Munafò, and Dario Rossi. Traffic analysis with off-the-shelf hardware: Challenges and lessons learned. *IEEE Communications Magazine*, 55(3):163–169, 2017.

[113] Luca Deri and NETikos SpA. nprobe: an open source netflow probe for gigabit networks. In *TERENA Networking Conference*, pages 1–4, 2003.

[114] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. A network analysis on cloud gaming: Stadia, GeForce Now and PSNow, 2021.

[115] Gianluca Perna, Dena Markudova, Martino Trevisan, Paolo Garza, Michela Meo, Maurizio M Munafò, and Giovanna Carofiglio. Online classification of rtc traffic. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2021.

[116] Gianluca Perna, Dena Markudova, Martino Trevisan, Paolo Garza, Michela Meo, Maurizio M Munafò, and Giovanna Carofiglio. Real-time classification of real-time communications. *IEEE Transactions on Network and Service Management*, 2022.

[117] Dena Markudova, Martino Trevisan, Paolo Garza, Michela Meo, Maurizio M Munafo, and Giovanna Carofiglio. What's my App?: ML-based classification of RTC applications. *ACM SIGMETRICS Performance Evaluation Review*, 48(4):41–44, 2021.

[118] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[119] Magnus Westerlund and Stephan Wenger. RTP Topologies. RFC 7667, November 2015.

[120] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[121] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[122] Laura Toloşi and Thomas Lengauer. Classification with correlated features: unreliability of feature ranking and solutions. *Bioinformatics*, 27(14):1986–1994, 2011.

[123] Lorien Y Pratt. Discriminability-based transfer between neural networks. *Advances in neural information processing systems*, pages 204–204, 1993.

[124] Luca De Cicco, Gaetano Carlucci, and Saverio Mascolo. Congestion control for webrtc: Standardization status and open issues. *IEEE Communications Standards Magazine*, 1(2):22–27, 2017.

[125] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. Performance evaluation of webrtc-based video conferencing. *ACM SIGMETRICS Performance Evaluation Review*, 45(3):56–68, 2018.

[126] Jeongyoon Eo, Zhixiong Niu, Wenxue Cheng, Francis Y Yan, Rui Gao, Jorina Kardhashi, Scott Inglis, Michael Revow, Byung-Gon Chun, Peng Cheng, et al. Opennetlab: Open platform for rl-based congestion control for real-time communications. *Proc. of APNet*, 2022.

[127] Ali C. Begen. Grand challenge on bandwidth estimation for real-time communications.

[128] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[129] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[130] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[131] ITUT ITU-T. Recommendation g. 114. *One-Way Transmission Time, Standard G*, 114:84, 2003.

[132] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[133] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[134] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[135] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[136] Zhengxu Xia, Yajie Zhou, Francis Y Yan, and Junchen Jiang. Genet: automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, 2022.

[137] Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. In *International Conference on Machine Learning*, pages 2535–2544. PMLR, 2019.