POLITECNICO DI TORINO
Repository ISTITUZIONALE

A comparative study of RTC applications

Original
A comparative study of RTC applications / Nistico, A., Markudova, D., Trevisan, M., Meo, M., Carofiglio, G.. - ELETTRONICO. - (2020), pp. 1-8. (2020 IEEE International Symposium on Multimedia (ISM) Napoli (IT) 2-4 Dicembre 2020) [10.1109/ISM.2020.00007].

Availability:
This version is available at: 11583/2867792 since: 2021-01-26T17:48:41Z

Publisher:
IEEE

Published
DOI:10.1109/ISM.2020.00007

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

18 November 2026

A comparative study of RTC applications

Antonio Nisticò†, Dena Markudova†, Martino Trevisan†, Michela Meo†, Giovanna Carofiglio ‡,
†Politecnico di Torino, ‡Cisco Systems Inc.
first.last@polito.it, gcarofig@cisco.com

***Abstract*—Real-Time Communication (RTC) applications have become ubiquitous and are nowadays fundamental for people to communicate with friends and relatives, as well as for enterprises to allow remote working and save travel costs. Countless competing platforms differ in the ease of use, features they implement, supported user equipment and targeted audience (consumer of business). However, there is no standard protocol or interoperability mechanism. This picture complicates the traffic management, making it hard to isolate RTC traffic for prioritization or obstruction. Moreover, undocumented operation could result in the traffic being blocked at firewalls or middleboxes.**

In this paper, we analyze 13 popular RTC applications, from widespread consumer apps, like Skype and Whatsapp, to business platforms dedicated to enterprises – Microsoft Teams and Webex Teams. We collect packet traces under different conditions and illustrate similarities and differences in their use of the network. We find that most applications employ the well-known RTP protocol, but we observe a few cases of different (and even undocumented) approaches. The majority of applications allow peer-to-peer communication during calls with only two participants. Six of them send redundant data for Forward Error Correction or encode the user video at different bitrates. In addition, we notice that many of them are easy to identify by looking at the destination servers or the domain names resolved via DNS. The packet traces we collected, along with the metadata we extract, are made available to the community.

***Index Terms*—Real Time Communication, Network, Protocols, RTP, VoIP**

I. Introduction

In recent years, the spread of broadband Internet access and mobile networks fostered the adoption of Real-Time Communication applications, which allow individuals and groups of people to communicate via voice and video. They are now fundamental for leisure and business, supporting people to reach friends and relatives and allowing for remote working. The importance of RTC became especially evident during the COVID-19 pandemic, when the social distancing and lockdown measures adopted to curb the outbreak forced billions of people to communicate solely by using RTC platforms. Indeed, nowadays there are countless competing applications for RTC, which differ in terms of ease of use, pricing strategy and targeted audience.

Historically, the first proposals for real-time communication over IP networks were based on the Session Initiation Protocol (SIP) [1] for session setup and the Real Time Protocol (RTP) [2] 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 [3]. 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 them is diverse in each application. Moreover, the vast majority of applications are closed-source and provide none or very little documentation. If this is somehow required to protect the intellectual property behind them, 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 paper, we study and compare different RTC applications and enlighten 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), and from 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. The notable findings we obtain are:

- Most of the applications still rely on 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.

This work has been supported by the SmartData@PoliTO center on Big Data and Data Science and Cisco Systems Inc.

- 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 paper is organized as follows. In Section II, we provide a background on the most popular protocols used in RTC applications. Section III describes the applications under test as well as the packet traces we collect. We present our findings starting with Section IV, which illustrates the network protocols we find. Section V discusses the different design choices, while Section VI provides useful guidelines for traffic identification. Section VII discusses the related work and finally Section VIII concludes the paper.

II. 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 [2]. 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, while leaving also the possibility of defining them dynamically during a session. Then, the data are 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. SRTP [4] is a variant of RTP that achieves confidentiality by encrypting the media payload while leaving all the original headers in clear.

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

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

TABLE I: 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

Session Setup. To establish a media session, it is necessary to ensure that the endpoints can communicate with each other, especially in the case of peer-to-peer communication among participants. This is complicated by the presence of NATs, firewalls and middleboxes in general. To ensure connectivity, the applications often use the STUN protocol [5] for NAT detection and TURN [6] to relay the traffic through a server that resides on the public Internet. ICE [7] combines STUN and TURN into a single technique. The RFC 7983 [8] defines a simple mechanism for multiplexing RTP, STUN and other protocols on the same UDP flow. Finally, the Session Description Protocol (SDP) [9] 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.

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 [10] and DTLS [11] protocols.

WebRTC. The above protocols need to be carefully coordinated to have a working RTC application. To ease the development, WebRTC [12] 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.

III. Data Collection

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 I. 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 [13], 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. We also manually inspect the `pcap` files (when no known protocol is found and for further analysis).

IV. 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 II, 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 II). We also find four applications using 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 publish online a simple command-line tool to strip this header so that RTP is contained directly in UDP, allowing analysis with classical packet inspectors [14].

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 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 could 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

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

[4]https://core.telegram.org/mtproto/description

TABLE II: 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

value `0x050f0100`. We cannot find any explanation of this mechanism on the online documentation of Zoom, but in the command-line tool we created [14], 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 [15]. 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.

V. Operation and design choices

We now draw our attention on 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 II, called "Operation".

A. 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. 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 II, 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.

B. 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 [16]. In other cases, the sender transmits

[5]https://blog.gotomeeting.com/gotomeeting-transports-protects-data/

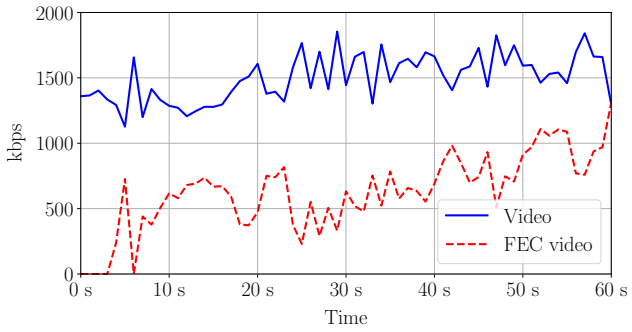


Fig. 1: Example of FEC in Webex Teams: a video stream and its corresponding FEC stream.

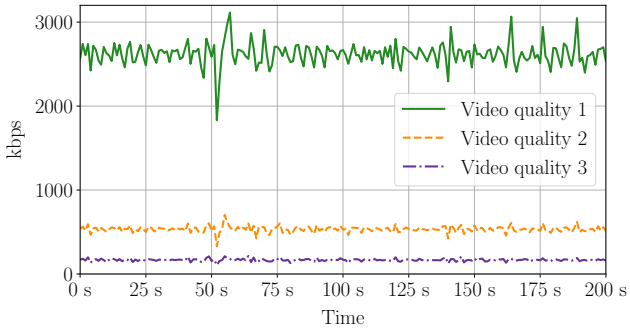


Fig. 2: Example of Simulcast in Google Meet: three video streams at different quality levels generated from one source.

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 – 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 II. 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 1. 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 [17]. 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.

A second use of redundant streams is the so called Simulcast technique that we indicate with *S* in the "Redundant data" column of Table II. 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 2, 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]

C. Particular uses of RTP

Here we report two cases of particular uses of RTP. First, we notice that FaceTime uses regular RTP traffic, but employs PT numbers which are forbidden by the protocol standard [2]. 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 source(s) 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 they are active in short

[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.

[8]See footnote 6 (Page 5).

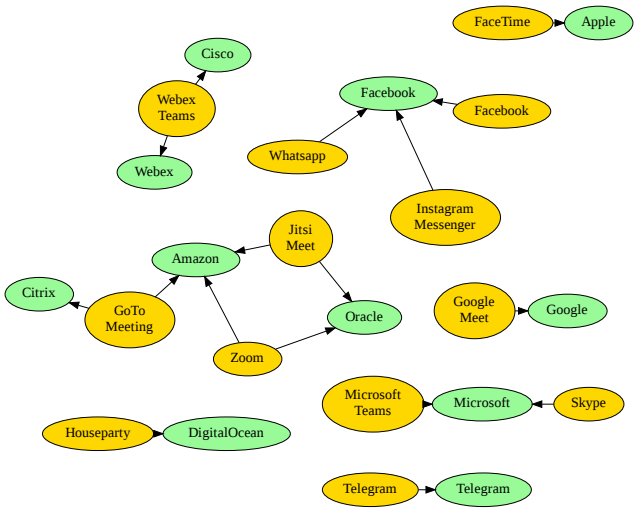


Fig. 3: Graph representation of the ASes (green) that RTC applications (yellow) use to relay RTC traffic.

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`.

VI. Identification of RTC applications

We now focus on the destination of the traffic generated by the RTC applications under test. We first investigate which ASes 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 is to provide useful guidelines for a network administrator willing to block specific RTC applications, because, e.g., not authorized within the enterprise.

A. 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) from http://www.routeviews.org/. 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 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 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 II.

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.

B. 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 II. 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.

C. 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 II. 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 [2] 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 II). In the RTC applications we study, we observe only usage of dynamic PTs, except for Skype, which sometimes uses static PTs for audio.[9] However, some applications use dynamic PTs in a static fashion and allow for easy distinction of audio, video, FEC or other types of streams, by assigning PTs to media types. Knowing the media type in real-time could pave the way for network management policies that favor Quality of Experience (QoE) of users, by letting more significant flows be prioritized. For example, if a participant is experiencing poor network conditions, this could allow for enhancing their audio over their video, as a more important stream. 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[10], 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 II.

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

VII. Related Work

The operation of RTC applications has been studied since their introduction. Several works target Skype traffic, since it was the pioneer in the world of RTC in the early 2000s. Bonfiglio *et al.* [3] 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 [18] and [19], based on traffic measurements, which leads to the same conclusions of Guha *et al.* [20]. Moreover, Baset *et al.* [21] analyze its key functions: login, NAT/Firewall traversal, and media transfer, while Hoßfeld *et al.* [22] provide an analysis of Skype VoIP traffic in mobile networks, focusing on Quality of Service (QoS) and QoE. With this work, 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 [23], Android smartphones [24] and Android devices in general [25]. 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 paper. We however testify that Telegram is peculiar among RTC applications also for the employed network protocols.

There are fewer works that compare different RTC applications. In our previous work [26], we deploy a classifier to distinguish 5 meeting applications in real-time. Azfar *et al.* [27] study ten Android VoIP applications, mainly from the security point of view. Karya *et al.* [28] compare RTP traffic in Whatsapp and Skype, under mobile networks. Wuttidittachotti *et al.* [29] provide a study on the perceived QoE of three well-known VoIP applications, using Perceptual Evaluation of Speech Quality (PESQ). Xu *et al.* [30] report a measurement study of Google+, iChat, and Skype, unveiling important information about their key design choices and performance. They extend their analysis in [31], this time

[9] FaceTime uses the reserved PT 20 as we discuss in Section IV.

[10] See footnote 6 (Page 5).

focusing on FaceTime, Google Plus Hangout, and Skype. Sutkino *et al.* [32] compare the instant messaging services of WhatsApp, Viber and Telegram in terms of security, speed and ease-of-use. Patel *et al.* [33] 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.

With respect to these works, we target a wider range of RTC applications, comparing 13 leading services in the consumer and business market segments, while past papers focus on up to four. 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 our work provides a rich but concise summary of the solutions currently adopted, unifying and updating what previous works have discovered.

VIII. Conclusion

In this paper, we presented a comparative study of 13 popular RTC applications. Our goal is to study similarities and differences in the use of the network and protocols, as well as providing useful insights for the identification of their traffic. We collected a large amount of packet traces for each application, under different conditions. We found that most of them use the RTP protocol in combination with STUN/TURN, but each has its own peculiarities. From the operation perspective, we observed that 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 or for mitigating losses (FEC). We discovered that most of them are simple to identify, by looking at the destination AS of the traffic, domains resolved via DNS, port numbers and PTs. We believe this paper is useful in providing an updated overview of the scene of RTC applications and can help network administrators in improving traffic management.

References

[1] H. Schulzrinne, E. Schooler, J. Rosenberg, and M. J. Handley, "SIP: Session Initiation Protocol." RFC 2543, Mar. 1999.

[2] R. Frederick, S. L. Casner, V. Jacobson, and H. Schulzrinne, "RTP: A Transport Protocol for Real-Time Applications." RFC 1889, Jan. 1996.

[3] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli, "Revealing skype traffic: when randomness plays with you," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 37–48, 2007.

[4] K. Norrman, D. McGrew, M. Naslund, E. Carrara, and M. Baugher, "The Secure Real-time Transport Protocol (SRTP)." RFC 3711, Mar. 2004.

[5] J. Rosenberg, C. Huitema, R. Mahy, and J. Weinberger, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)." RFC 3489, Mar. 2003.

[6] P. Matthews, J. Rosenberg, and R. Mahy, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)." RFC 5766, Apr. 2010.

[7] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols." RFC 5245, Apr. 2010.

[8] M. Petit-Huguenin and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)." RFC 7983, Sept. 2016.

[9] M. J. Handley and V. Jacobson, "SDP: Session Description Protocol." RFC 2327, Apr. 1998.

[10] C. Allen and T. Dierks, "The TLS Protocol Version 1.0." RFC 2246, Jan. 1999.

[11] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security." RFC 4347, Apr. 2006.

[12] C. Holmberg, S. Hakansson, and G. Eriksson, "Web Real-Time Communication Use Cases and Requirements." RFC 7478, Mar. 2015.

[13] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, and D. Rossi, "Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned," *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 163–169, 2017.

[14] D. Markudova, M. Trevisan, and M. Munafo, "RTC Pcap Cleaners." https://github.com/marty90/rtc_pcap_cleaners.

[15] "Ieee standard for layer 2 transport protocol for time sensitive applications in a bridged local area network," *IEEE Std 1722-2011*, pp. 1–65, 2011.

[16] J.-M. Valin, K. Vos, and T. Terriberry, "Definition of the Opus Audio Codec." RFC 6716, Sept. 2012.

[17] J.-C. Bolot, M. J. Handley, V. Hardman, I. Kouvelas, and C. Perkins, "RTP Payload for Redundant Audio Data." RFC 2198, Sept. 1997.

[18] D. Bonfiglio, M. Mellia, M. Meo, and D. Rossi, "Detailed analysis of skype traffic," *IEEE Transactions on Multimedia*, vol. 11, no. 1, pp. 117–127, 2008.

[19] D. Bonfiglio, M. Mellia, M. Meo, N. Ritacca, and D. Rossi, "Tracking down skype traffic," in *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pp. 261–265, IEEE, 2008.

[20] S. Guha and N. Daswani, "An experimental study of the skype peer-to-peer voip system," tech. rep., Cornell University, 2005.

[21] S. A. Baset and H. Schulzrinne, "An analysis of the skype peer-to-peer internet telephony protocol," *arXiv preprint cs/0412017*, 2004.

[22] T. Hoßfeld and A. Binzenhöfer, "Analysis of skype voip traffic in umts: End-to-end qos and qoe measurements," *Computer Networks*, vol. 52, no. 3, pp. 650–666, 2008.

[23] J. Gregorio, B. Alarcos, and A. Gardel, "Forensic analysis of telegram messenger desktop on macos,"

[24] C. Anglano, M. Canonico, and M. Guazzone, "Forensic analysis of telegram messenger on android smartphones," *Digital Investigation*, vol. 23, pp. 31–49, 2017.

[25] G. B. Satrya, P. T. Daely, and M. A. Nugroho, "Digital forensic analysis of telegram messenger on android devices," in *2016 International Conference on Information & Communication Technology and Systems (ICTS)*, pp. 1–7, IEEE, 2016.

[26] D. Markudova, M. Trevisan, P. Garza, M. Meo, M. Munafò, and G. Carofiglio, "What's my app? ml-based classification of rtc applications," *To appear in ACM SIGMETRICS Performance Evaluation Review*, 2020.

[27] A. Azfar, K.-K. R. Choo, and L. Liu, "Android mobile voip apps: a survey and examination of their security and privacy," *Electronic Commerce Research*, vol. 16, no. 1, pp. 73–111, 2016.

[28] 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*, vol. 453, p. 012062, 2018.

[29] P. Wuttidittachotti, W. Akapan, and T. 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*, pp. 456–461, IEEE, 2015.

[30] Y. Xu, C. Yu, J. Li, and Y. Liu, "Video telephony for end-consumers: measurement study of google+, ichat, and skype," in *Proceedings of the 2012 Internet Measurement Conference*, pp. 371–384, 2012.

[31] C. Yu, Y. Xu, B. Liu, and Y. Liu, ""can you see me now?" a measurement study of mobile video calls," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pp. 1456–1464, IEEE, 2014.

[32] T. Sutikno, L. Handayani, D. Stiawan, M. A. Riyadi, and I. M. I. Subroto, "Whatsapp, viber and telegram: Which is the best for instant messaging?," *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 6, no. 3, 2016.

[33] 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)*, pp. 1–3, 2018.