Large-Scale Measurement of Real-Time Communication on the Web

Shaohong Li

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 20.11.2017

Thesis supervisor:

Prof. Patric Östergård

Thesis advisor:

PhD. Pasi Sarolahti

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Shaohong Li

Title: Large-Scale Measurement of Real-Time Communication on the Web

Date: 20.11.2017 Language: English Number of pages: 6+51

Department of Communications and Networking

Professorship: Networking Technology

Supervisor: Prof. Patric Östergård

Advisor: PhD. Pasi Sarolahti

Web Real-Time Communication (WebRTC) is getting wide adoptions across the browsers (Chrome, Firefox, Opera, etc.) and platforms (PC, Android, iOS). It enables application developers to add real-time communications features (text chat, audio/video calls) to web applications using W3C standard JavaScript APIs, and the end users can enjoy real-time multimedia communication experience from the browser without the complication of installing special applications or browser plug-ins.

As WebRTC based applications are getting deployed on the Internet by thousands of companies across the globe, it is very important to understand the quality of the real-time communication services provided by these applications. Important performance metrics to be considered include: whether the communication session was properly setup, what are the network delays, packet loss rate, throughput, etc.

At Callstats.io, we provide a solution to address the above concerns. By integrating an JavaScript API into WebRTC applications, Callstats.io helps application providers to measure the Quality of Experience (QoE) related metrics on the end user side. This thesis illustrates how this WebRTC performance measurement system is designed and built and we show some statistics derived from the collected data to give some insight into the performance of today's WebRTC based real-time communication services. According to our measurement, real-time communication over the Internet are generally performing well in terms of latency and loss. The throughput are good for about 30% of the communication sessions.

Keywords: Measurement, Quality of Experience, Real-Time Communications, WebRTC

Preface

I want to thank my advisor Dr. Pasi Sarolahti and my supervisor Professor Patric Östergård for their guidance and valuable feedbacks on the thesis writing. Thanks to Professor Jörg Ott for creating a context within that the idea of WebRTC measurement platform is brewed. Many thanks go to my colleagues at Callstats.io: Varun Singh, Marcin Nagy, Karthik Budigere, Eljas Alakulppi, Lennart Schulte and everyone else, for the great teamwork that transformed the SaaS idea into a reliable service that helps hundreds of companies around the world to monitor and manage the quality aspects of their WebRTC services.

Espoo, 20.11.2017

Shaohong Li

Contents

Abbreviations

DTLS	Datagram Transport Layer Security
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
ISP	Internet Service Provider
PSTN	Public switched telephone network
QoS	Quality of Service
QoE	Quality of Experience
RFC	Request for Comments
RTC	Real-time Communications
RTCP	Real-Time Transport Control Protocol
RTP	Real-time Transport Protocol
SaaS	Software as a Service
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
SRTP	Secure Real-time Transport Protocol
SSRC	Synchronization Source (identifier)
STUN	Session Traversal Utilities for NAT
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VoIP	Voice over IP
W3C	World Wide Web Consortium
WebRTC	Web Real-Time Communications

1 Introduction

Real-time communication applications are booming these days. Usage of popular services such as Skype, Facebook Messenger, Google Hangout, WeChat, etc., has become an indispensable part of our everyday life. For businesses, these tools and applications help increase collaborations and productivity while saving lots of travel costs and energy consumption. On the personal front, they make it easy for people to keep the social bonds with friends and relatives long distance apart.

As more and more real-time communication applications are emerging and competing for more user engagement and market shares, it is natural for the providers of these applications to consider not only the features but also the quality of the services they are offering and seeking ways to improve the end user's experience. The main topic of this thesis is about the performance measurement system for this type of real-time communication services.

1.1 Background

WebRTC, which stands for Web Real-Time communication, is the latest addition to real-time communication developer's toolbox. This technology was initiated by Google and coordinated by working groups in both W3C and IEFT for standardization. W3C specifications define standard JavaScript APIs to let developers add real-time communication capabilities to web applications running within the browser. And the working group in IETF defines details of WebRTC at the protocol level. Following the standards, different browsers or library implementations should be able seamlessly inter-operate with each other. The main benefits for end user is to leverage the ubiquitous existence of web browsers to dynamically load and run web applications without the need to install any special software or plug-ins. As the technology is standardized and easily accessible, application developers can either write web application or standard-alone applications using WebRTC library implementations.

As more and more web based applications are adopting WebRTC technologies, there is a common need among these applications to understand the end user's quality of experience. For most web developers, firstly there is a learning curve to understand all the factors related to multimedia communication, secondly it is a nontrivial investment of time and effort to build, scale and maintain this kind of measurement and analysis system in house. Therefore, a cloud-based service which can help these application providers to perform such measurement and analysis will be very desirable.

1.2 Goals

There are two major goals we want to achieve with this thesis:

- We will motivate why a Software-as-a-Service (SaaS) approach to monitor WebRTC application performance is doable, and we show how we built such a system at Callstats.io.

- With the SaaS platform at Callstats.io, we have helped hundreds of application service providers to perform real-time communication performance measurement. We use a subset of the measurement data to draw a big picture of how real-world web based real-time communication performance looks like. Metrics we show include things such as latency, loss, throughput, NAT traversal performances, etc.

1.3 Organization

The thesis is organized as following:

- In Chapter 2, we introduce the basics of real-time communication, which include call signaling and media transport set up, etc. Then we introduce the basics of WebRTC technologies and how it can be used to implement real-time communication applications within web browsers.

- In Chapter 3, we introduce the various factors that can affect the end user's experience during a communication session and how those metrics can be measured for WebRTC calls.

- In Chapter 4, we explain the motivation for building a SaaS system that can help WebRTC application providers to monitor the quality aspects of their services. We show the architectural design of such a system at Callstats.io.

- In Chapter 5, we use a subset of the measurement data collected at Callstats.io to draw a big picture about the performance of today's web based real-time communication services in terms of various metrics.

- In the final Chapter, we summarize our findings, list our limitations and point out some further studies that shall be done to gain more insight from the measurement data.

2 Real-Time Communication and WebRTC

The Internet was created as a medium for people to exchange data. These data can be emails, files or multimedia content such as audio or video streams. Real-time communication (RTC) over the Internet requires that the audio/video contents are captured, exchanged and rendered with minimal delays so that the users can enjoy an interactive conversational experience. In this chapter we will first introduce, from networking technology perspective, the key components that enabled real-time communication applications. Then we will introduce how RTC can be implemented within the web browser (the most ubiquitously installed application).

2.1 Real-Time Communication

Back in 1973, Network Voice Protocol (NVP) [1] was implemented by researchers to carry real-time voice communication between different sites over ARPANET (Advanced Research Projects Agency Network, the precursor of Internet). The surge of using Internet to provide real-time communication services happened in the 1990s when Internet telephony services emerged and allowed people to make long distance calls over the public Internet. The audio quality of these IP (Internet Protocol) based calls are usually acceptable and the cost are typically much cheaper than the calls made over public switched telephone Network (PSTN).

Over the time, network capacity increased and technologies evolved, more and more applications were created in the consumer and the enterprises spaces enabling users to interact and collaborate in real-time via audio/video/text. Today, real-time communication applications and services such as Skype, FaceTime, Google Hangout, WeChat, etc., are serving billions of users. Video conferencing equipments from Cisco, Polycom and other vendors are also widely deployed in the meeting rooms of big and medium sized corporations, bringing the benefits of increased collaboration efficiency and reduced travel cost. Business wise, today's enterprise real-time communication market is already valued at multi billion dollars and expected to grow even bigger in the coming years[1].

The core technologies behind Voice over IP (VoIP) and video conferencing services include:

- Signaling protocols that are used to set-up and tear-down a communication session and describes its media characteristics (e.g. what kind of media will be used, what is the bandwidth requirement, etc.).
- Audio and video codecs for encoding and decoding the audio and video media content.
- Transport protocols that are used to deliver audio/video streams between the users in real-time.

[1]http://www.businesswire.com/news/home/20170921005575/en/Global-Enterprise-Video-Conferencing-Market---Growth

Works on audio and video codecs fall into the digital signal processing domain and focus on providing high compression ratio while keeping the perceived audio/video quality good. For the networking community, the standardization and application of the signaling protocols and multimedia transport protocols has been a major topic for the last two decades and we will take a look at those aspects in the following sections.

2.1.1 Signaling Protocol

In real-time multimedia communication applications, signaling messages are exchanged between participating entities to signify the start and end of a call. The messages may also contain information about the details of the session, such as what kind of media types will be used, capabilities about supported codecs, bandwidth requirements, etc., and where to send/receive the multimedia streams. Signaling messages are also used for call control purposes, such as putting a call on hold, transfer a call, etc.

Since the emergence of IP Telephony in the 1990s, organizations such as the International Telecommunication Union (ITU) and the Internet Engineering Task Force (IETF) had put on efforts to standardize the VoIP singling protocols for better interoperability between different multimedia communication service and equipment vendors. The most widely adopted signaling protocols are H.323 [2] specified by ITU and Session Initiation Protocol (SIP) [3] specified by IETF. The two organizations have different rationales behind the protocol development[2] but their capability and extensibility are all comparable. H.323 uses ASN.1 (abstract syntax notation one) protocol syntax and PER (Packed Encoding Rules) to encode messages in binary forms. The binary encoding makes it more efficient on the wire but it also makes it a bit demanding for application development and there are only a few commercial or open-source H.323 libraries available for developers to use. SIP follows a syntax similar to HTTP and use plain text for message encodings which is more human friendly to work with. There are quite a few commercial or open-source development stacks available when it comes to develop RTC applications in SIP. H.323 came out earlier than SIP. The first version of H.323 was published in 1996. The first version of SIP protocol was standardized in 1999. Although the two protocols are, mostly, functionally equivalent, SIP has become more widely used as the signaling protocol to build VoIP services. For example, the IP Multimedia Subsystem (IMS) [21] (the core multimedia service delivery network for mobile operators) specified by 3GPP (3rd Generation Partnership Project) uses SIP as the signaling protocol between its components. We will take a close look at SIP next.

The core of SIP protocol is defined in RFC 3261 [3] which specified the models and messages for call establishment and call control. RFC 3261 also specified the registration and routing mechanism to facilitate the rendezvous between entities in a communication session. SIP follows the request-response transaction model used in HTTP. Core request methods include REGISTER, INVITE, CANCEL, ACK, BYE, OPTIONS. SIP response codes also follow HTTP syntax. For example, 1xx

[2]https://www.packetizer.com/ipmc/h323_vs_sip

indicates provisional responses; 2xx for successful responses; 3xx for redirections to new contact addresses; 4xx for client error; 5xx for server error, etc. SIP is independent of transport protocols thus it can run over any transport layer such as TCP, UDP, SCTP, etc.

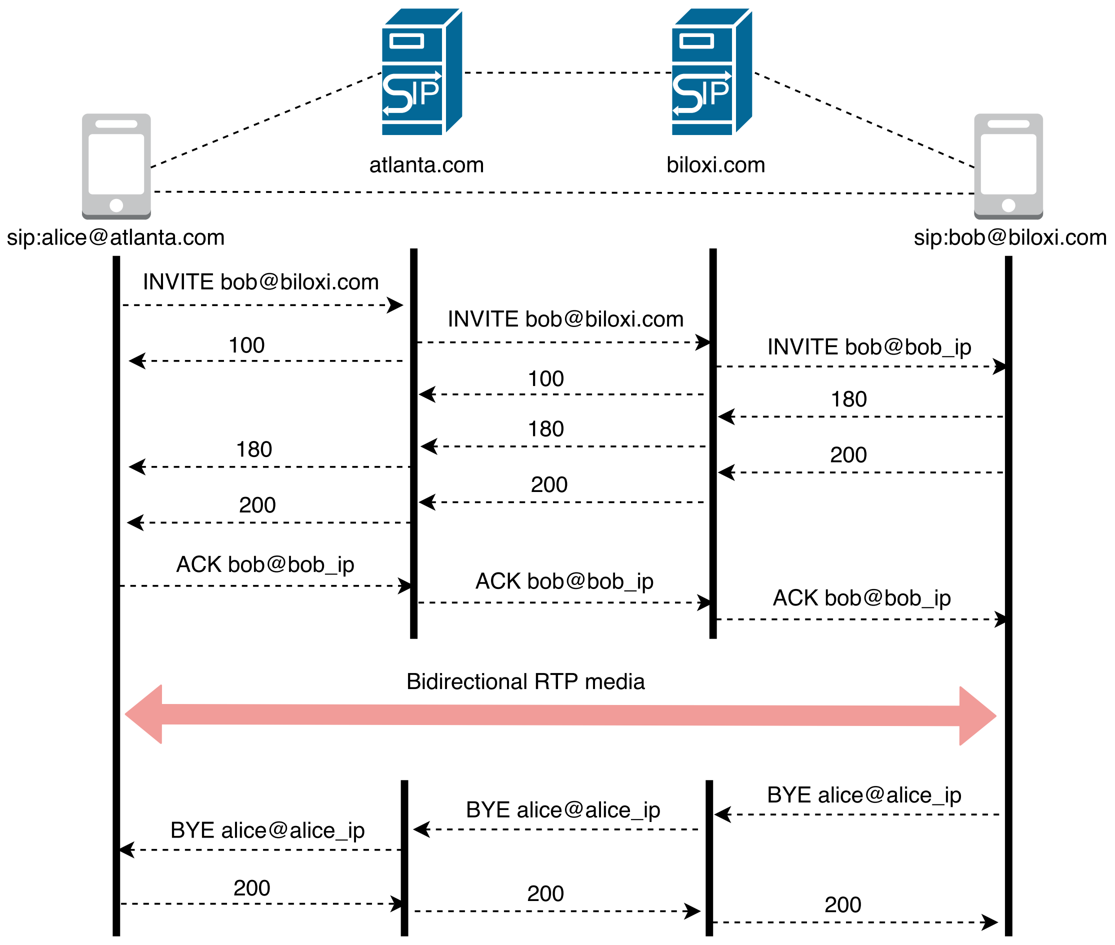


Figure 1: SIP basic call flow [3]

A basic SIP call flow is shown in Figure 1. This diagram is traditionally called "SIP trapezoid". In this scenario, user Alice is identified with URI (Uniform Resource Identifier) "sip:alice@atlanta.com" and user Bob is identified with URI "sip:bob@biloxi.com". Both users registered their contact addresses with their servers so that they can be located when being called. When Alice wants to call Bob, her user agent sends an INVITE request to her server which serves all subscribers under "atlata.com" domain. The "atlanta.com" server then proxies the request to the server which serves subscribers under "biloxi.com" domain. Bob's server receives the INVITE, finds the contact address of Bob's user agent, from its registry, and proxy the request to Bob. Response 180 signifies that Bob is being alerted by his user agent about the incoming call. Response 200 signifies Bob answered the call. ACK request assures Bob that Alice had received the 200 response. The ACK request serves the reliability mechanism for successful session establishment, that is, if ACK is not received by

Bob's user agent it will resend the 200 response.

SIP uses SDP offer/answer model [16] to exchange the multimedia capabilities of the user agent and agree upon a set of commonly supported media capabilities. For example, the INVITE from Alice can include a SDP like this:

```
v=0
o=alice 2890844526 2890844526 IN IP4 host.atlanta.example.com
s=
c=IN IP4 192.0.2.101
t=0 0
m=audio 49170 RTP/AVP 0 97
a=rtpmap:0 PCMU/8000
a=rtpmap:97 iLBC/8000
m=video 51372 RTP/AVP 31
a=rtpmap:31 H261/90000
```

where the "c=" line indicates that IP address 192.0.2.101 will be used to send/receive media, and the "m=audio" line indicates it supports audio media and use RTP [4] with payload type "0" (PCMU) and port 49170 to receive the audio stream. The "m=video" line indicates it supports video media and the payload type "31" maps to H.261 video codec. This SDP is called offer SDP as it is sent from Alice to Bob to propose the possible media types and transport methods. When Bob's user agent receives this SDP, it will check it against its own media capabilities and sends back an answer SDP which describes the capabilities that Bob supports. It may remove any codecs that it does not support and it can also reject a media by setting the port to 0. If none of the proposed media types can be supported by Bob's user agent, it can reject the INVITE by sending a 4xx response.

2.1.2 Media Transport Protocol

While signaling protocols are used to manage the state of a call and negotiate media content specifics, the actual media content are delivered through media transport protocol. For interactive real-time applications, the major challenge is the on-time delivery of media packets. Timeliness outweighs reliability in the protocol design.

Real-time Transport Protocol (RTP) [4] is the most widely adopted protocol by real-time applications to deliver real-time interactive audio/video media data. Early versions of RTP implementations were used to transport voice over the Internet's multi-cast backbone (MBONE). The first standard for RTP is RFC 1889 in 1996 and later on it was superseded by RFC 3550 in 2003. Both H.323 and SIP protocols use RTP for audio/video media content delivery.

RTP is an application layer protocol and can use either UDP or TCP as the transport. UDP is used in most cases as its fast delivery approach (send and forget) matches well with the stringent latency requirement on real-time media. TCP transport is less preferable as retransmission of real-time media packets is usually not desired, but still it can be used when UDP traffic is blocked, for example, in a strictly controlled network where UDP traffic is blocked. Some research had been done to model the feasibility and strategies to use TCP for real-time media delivery [5].

The packet structure for RTP is shown in Figure 2. Important fields include:

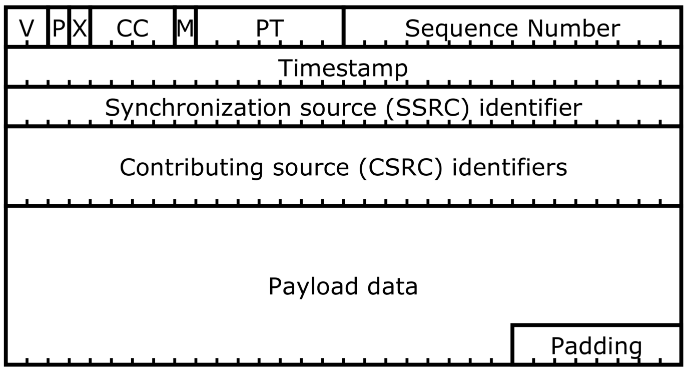

V
P
X
CC
M
PT
Sequence Number
Timestamp
Synchronization source (SSRC) identifier
Contributing source (CSRC) identifiers
Payload data
Padding

Figure 2: RTP packet structure [7]

- The payload type (**PT**) field specifies the type of the payload, thus determines how application shall interpret the received data. Some static payload types are defined in RFC 3551 [6].
- The **sequence number** field increases by one for each RTP data packet sent. Receiver can use it for packet reordering in the playback buffer as well as detecting packet loss.
- The **timestamp** field is the time instance (relative to the first sample) when the payload was sampled. If multiple RTP packets are used to carry one sample (e.g. in the case of an initial frame in a video stream), these RTP packets shall use the same time stamp. Time stamp allows receiver to playback received packets at the right time interval. This timestamp can be mapped to absolute wall-clock time with the help of RTCP Sender Report packet which is also part of RTP protocol. With the timestamp information, receiver can calculate the network delays.
- The **synchronization source (SSRC) identifier** field identifies the source of the media stream, such as a microphone or a camera. Packets with the same SSRC shares the same timing and sequence number namespace and are grouped together on the receiver side for playback.
- The **contributing source (CSRC) identifiers** are used when the sender of the RTP packets is a RTP mixer (e.g. a media server that mix several audio/video streams into one audio/video stream) or RTP translator (e.g. a trans-coder that converted the origininal media stream from one codec profile to another code profile). In these scenarios, the SSRC will be the id of the mixer/translator, but the CSRC will contain the original SSRC(s) used to generate the new media stream.

RTP is augmented by RTP Control Protocol (RTCP) which uses a port number right above the RTP port. RTCP packets are exchanged periodically to provide feedbacks about quality of service (QoS), such as latency, packet loss, etc., of the RTP streams. Details about QoS will be covered in Chapter 3. Based on the feedback, the RTP senders can make adaptations on the media encoding so that user's quality of experience can be managed properly. For example, user can send higher profile video when network condition is good and send lower profile video when network congestion is detected.

When a communication session has multiple media streams (i.e. both audio and video) and each media stream is transported over its own RTP stream, RTCP provides absolute time information so that the RTP timestamps can be mapped to absolute wall-clock time. This makes it possible to do synchronizations between multiple media streams, such as lip syncing during a video call.

The overhead incurred by RTCP packets is small, typically less than 5% of the total media session traffic [7].

2.1.3 Signaling and Media Paths in RTC

In RTC applications, the signaling messages and media streams usually follows separate paths, similar to what is shown in Figure 1.

Signaling messages between the participating entities are typically exchanged through a centralized signaling server, where things such as address translation and call control can take place. There are peer-to-peer signaling architecture and implementations such as P2P SIP [17] which allow entities to send signaling protocols directly to each other, but those are not mainstream deployment scenarios.

The media streams between participants can either go through a centralized media server or flow directly between the parties. The benefit of exchanging media streams through a media server is that features such as media mixing or selective forwarding of media streams can be done by the media server and these features are often required in a multi-party conference call. Media servers can also function as an intermediary to adapt the differences in bandwidth or media capabilities among the participants so that user agents with different or even incompatible capabilities can co-exists in the same conference. All these benefits come with the extra infrastructure cost and potentially longer delays in media exchange. For above reasons, it is desirable for a two party call to have the media streams flow directly between the peers for both efficiency and cost considerations. Unfortunately this can be difficult to achieve because of the existence of Network Address Translation (NAT) and firewalls devices in the Internet. Some special procedures are needed to find a usable media path between the media sender and receiver.

2.1.4 NAT traversal for RTC

NAT and firewall functionalities are usually built into one box which sits between the local area network (LAN) and the Internet. Firewall manages network security and blocks unauthorized traffic both to and from the private network. NAT is used for IP address conservation. Computing devices behind the NAT use private IP address spaces to communicate with each other and these devices can share a pool of public IP addresses assigned to the enterprise or ISP. Once a device in the local network needs to visit the Internet, NAT will create a binding of address and port between the private side of the network and the public side of the network and rewrites the

IP address and port when it forwards packets in both directions. A diagram of how NAT works is shown in Figure 3. This binding of private and public address and port on NAT is ephemeral, i.e. it will be removed when there is no traffic flowing on this path any more.

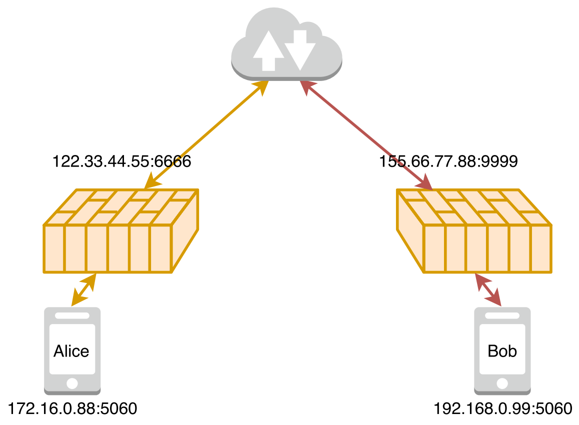


Figure 3: Network address translation example

In a multimedia communication session, the participating endpoints are typically located behind NAT while signaling servers is publicly reachable. NAT has some impact on the signaling messages, for example some message routing related headers in SIP contains IP addresses and those addresses maybe private therefore are not routable from outside of the LAN. The simplest solution for signaling messages to traverse NAT is to use TCP and always keep a persistent connection between the endpoint and the signaling server so that messages can be exchanged any time in both directions. For media streams and RTP traffic, in order for the endpoints behind the NAT to exchange media streams directly, some NAT traversal techniques are needed to help setting up the media path. The state-of-the-art solution to probe and establish connections between two endpoints is called ICE [18], which stands for "Interactive Connectivity Establishment". ICE specifies the protocol and procedures to set up a media path for UDP-based media streams to traverse NAT.

The communicating endpoints that support using ICE to establish media path through NAT are called ICE agents. ICE extends the SDP offer/answer model by including transport candidates in the exchanged session descriptions and then doing connectivity checks between the candidate pairs to find the possible media paths. The whole ICE procedure consists of several steps including: candidate address gathering, connectivity checking and concluding phase.

In the "candidates gathering" phase, based on the configurations, each ICE agent collects candidate transport addresses it could use to communicate with the peer agent. Each candidate transport address can be represented in the form of a (IP address, port, transport protocol) triplet. ICE defines four types of candidate addresses:

- The "Host" candidate is the address of network interface of the endpoint. If the host has multiple physical/logical interfaces then there could be multiple "Host" candidate addresses;

- The “Server Reflexive” candidate is the address of the endpoint as seen on the public side of the NAT. This is learned by making use of the Session Traversal Utilities for NAT (STUN) protocol [19]. Agent sends a STUN “Binding” request to its STUN server and the server copies the source transport address of the Binding request into the Binding response. When there are multiple layers of NAT between the endpoint and the Internet, the “Server Reflexive” address is the address as seen on the public side the outermost NAT;

- The “Relayed” candidate is an address allocated by the TURN server that sits in the public Internet and relays packets between the ICE agent and its peer. TURN stands for “Traversal Using Relays around NAT”. It extends the STUN protocol by adding an “Allocate” request which a TURN client can use to request the TURN server to allocate resource (in the form of a transport address) to reply packets between the client and its peer. TURN is a reliable way to by pass any type of NAT. If a TURN server is configured for an ICE agent, it will report back the “Server Reflexive” address as well as the “Relayed” address in the “Allocate” response.

The candidate gathering process depends on the end point’s configuration. For example, if TURN server is not configured for the endpoint then it will not gather any relayed candidate.

After the ICE agent collected the various candidate transport addresses, it prioritizes them numerically according to certain preference criteria and then sends the candidate list in an SDP offer through the signaling channel (e.g. in a SIP INVITE message) to its peer. When the peer receives the offer, it goes through a similar gathering process and sends back its transport candidates in a SDP answer.

When both ICE agents have the list of each other’s candidates, the “connectivity check” phase starts. In this step, each agent first forms an ordered list of candidate pairs (ordered by the combined priorities of local and peer candidate), then starts a series of checks to test which pairs work. A check is done by sending a STUN Binding request from the local candidate to remote candidate as indicated in Figure 4. A check is successful if a successful response is received and the source and destination transport addresses match with the candidate pair. Candidate pairs that passed the connectivity checks are put into the valid pairs list.

In order to reach an agreement upon which media path to use, ICE assigns one of the agent the “controlling” role and its peer the “controlled” role. The controlling agent dictates which candidate address pair will be used for media transport. By default, the ICE agent that sends the SDP offer assumes the controlling role. As the connectivity check goes on and some valid pairs are found, the controlling agent can stop the checks and then choose one from the valid list as the final transport addresses. The controlling agent “nominates” the selected candidate pair by sending a STUN Binding request with the USE-CANDIDATE attribute. Once this transaction completes, both agent will cancel any further connectivity checks. This concludes the ICE process and media stream can begin to flow over the connection between the selected transport addresses pair. With the ICE procedure, STUN and RTP are multiplex via the same network connection.

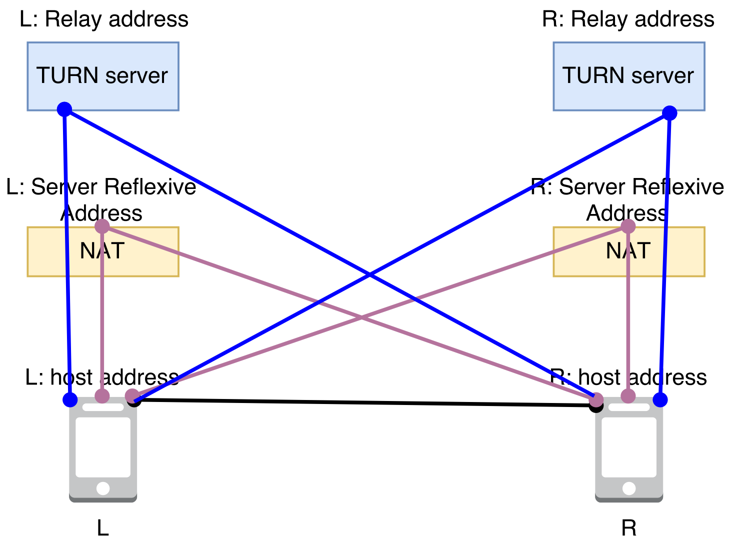


Figure 4: ICE connectivity check [18]

After ICE procedure concludes, either agent can restart it again at a later time by sending an updated SDP offer. A restart is needed for example in the case where an agent changed its IP address which invalidates the previous selected transport address pair.

To keep the NAT bindings valid during the communication session, ICE has a keep-alive mechanism. ICE agent uses STUN Binding Indication (a transaction where no response needs to be generated) to refresh its bindings on the NATs.

2.2 Web Real-Time Communication (WebRTC)

Using standard (and sometimes proprietary) signaling and media transport protocols, many VoIP software/hardware systems have been built over time. Some notable ones include Skype, NetMeeting[3], FaceTime[4], WebEx[5], etc. These applications have been successfully serving end users to engage in real-time multimedia communication either within managed enterprise networks or over the Internet. There are some annoyances with these applications, mostly on the installation and interoperability aspects. For example, customers need to install dedicated applications for each of these systems. It is also quite common that the client application for one system cannot communicate properly with client or servers applications of another system.

With the ubiquitous availability of web browsers since the 1990s, more and more applications have become web based. Web based applications offer several advantages. For the developers, there is no need to build and maintain specific application binaries for each operating systems as the applications can be dynamically downloaded from the web server and run in the browsers. The end users are also alleviated from the

[3]https://en.wikipedia.org/wiki/Microsoft_NetMeeting
[4]https://en.wikipedia.org/wiki/FaceTime
[5]https://en.wikipedia.org/wiki/WebEx

complication of installing and upgrading application binaries on their computers as all they need is a web browser. Same trend was also happening in the real-time communication field, where embedding the functionalities of such as voice and video chat inside the web browser could potentially open a new world of possibilities for web applications. But due to the complexity of technologies involved (audio, video, network) and licensing considerations, the initial offerings of RTC functionalities are done via proprietary plug-ins, such as Skype Click to Call[6], Adobe Flash Player[7], etc. All these solutions still require end user installations of browser plug-ins and there were no standard ways to use the RTC functionalities. The Web community yet need a standard approach to incorporate real-time communication functions into web applications.

2.2.1 WebRTC History

In 2010, Google acquired a video codec company, On2[8], and an audio codec library company, Global IP Solutions (GIPS)[9]. On2 was the owner of VP video codec series which provide alternatives to H.264 video codec. GIPS was a market leader for VoIP media engines. After the acquisition, Google's Chrome team integrated those technologies into their browsers and open-sourced them in Chromium in May 2011 with the aim of allowing developers to create voice and video chat applications via simple HTML and standard JavaScript APIs in a royalty free pattern [23].

The standardization works were initiated around the same time via W3C "Web Real-Time Communications" (WEBRTC) working group and IETF "Real-Time Communication in WEB-browsers" (RTCWEB) working group. Collaborators include people from Google, Mozilla, Microsoft, Ericsson, Cisco, etc. The W3C WEBRTC working group focuses on defining client-side JavaScript APIs to enable real-time communication in Web browsers. We will describe these APIs in detail in Section 2.2.7. The IETF RTCWEB working group focuses on defining the requirements and models for WebRTC functions, as well as creating or extending communication protocols to ensure interoperability between different WebRTC implementations which include browsers, gateways and application libraries, etc.

Both W3C and IETF working groups are making good progress as of today. The JavaScript APIs and communication protocol specifications are all relatively stable now. Chrome and Firefox are the two major browsers that have been actively developing WebRTC support since the beginning. Apple declared WebRTC support in Safari 11 in June, 2017. Microsoft is developing WebRTC support on the Edge browser in the form of Object RTC (ORTC) APIs which is similar to WebRTC API [22].

[6]https://www.skype.com/en/download-skype/click-to-call
[7]https://en.wikipedia.org/wiki/Adobe_Flash_Player
[8]https://en.wikipedia.org/wiki/On2_Technologies
[9]https://en.wikipedia.org/wiki/Global_IP_Solutions

2.2.2 WebRTC Stack

WebRTC implementations are under development among various browser, gateway and SDK vendors. As an example, Google published the WebRTC stack architecture in Chromium as shown in Figure 5. The notable components in Google's WebRTC stack include:

- The "Web API" (will be discussed in detail in Section 2.2.7) layer refers to the JavaScript APIs that are being standardized in W3C. Web application developers use these APIs to build features such as interactive audio/video/data communications.
- The "WebRTC Native C++ API" layer provides a library interface so that browser or native application makers can invoke these APIs to incorporate WebRTC functionalities into their applications.
- The "Voice Engine" and "Video Engine" components handle the encoding/decoding, packetization and signal processing (e.g. echo cancellation, packet loss concealment, etc.) for audio and video streams.
- The "Transport" component manages the actual peer-to-peer network communication channel setup and media content transport, using protocols such as ICE and RTP. WebRTC uses Stream Control Transmission Protocol (SCTP) [28] to transport data and secure RTP (SRTP) [27] to transport real-time media [24].

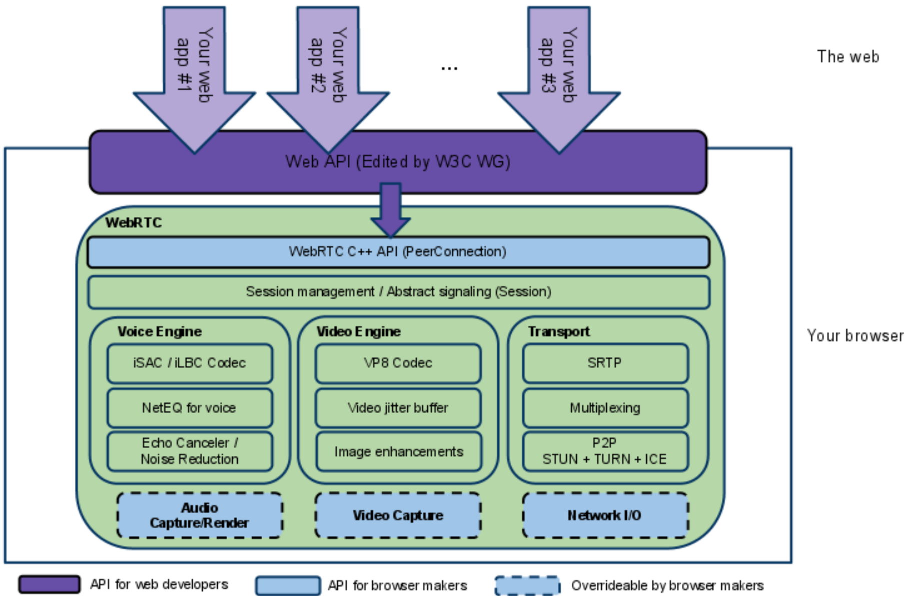


Figure 5: WebRTC stack architecture diagram
10

The IETF "WebRTC Overview" draft [9] gives a good description of the browser model as shown in Figure 6. The browser model emphasizes on the various interfaces between the web server, WebRTC application and browser's real-time communication functions. For example, it shows that HTTP or WebSocket can be used to convey signaling messages between the web application and web server, and the network communication mechanism used between peer browsers are abstracted as "on-the-wire protocols".

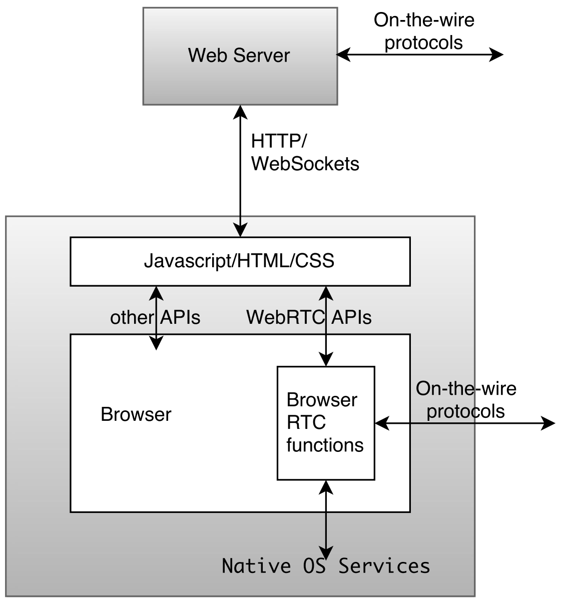


Figure 6: WebRTC browser model [9]

2.2.3 WebRTC Application

Having seen the WebRTC stack architecture and application function model in the previous section, let us now look at what kind of WebRTC applications can be built. As WebRTC provides APIs to generate multimedia content as well as setting up media and data communication channels between browsers, it can be used to create all kinds of interactive real-time communication application scenarios from within the browser, in simple and secure fashion.

The essential WebRTC applications model can be described as "WebRTC trapezoid" as shown in Figure 7. It is quite similar to the "SIP trapezoid" [3]. To set up a media path between two peer browsers, the RTC functions in the browser just need to know the local and peer's session descriptions in the form of SDPs that we have briefly introduced in Section 2.1.1. In the trapezoid model, the choice of signaling protocols between the communication endpoints (i.e. the browsers) and the communication servers (i.e. web servers) are deliberately left outside the scope of the WebRTC protocol suite. The rationale behind the decision was to give application developers the freedom to choose any signaling mechanism that matches their application needs [31] the best. To manage the call setup and tear-down, web applications can use either existing standard signaling protocols (e.g. SIP/XMPP/Jingle over WebSocket) or any proprietary customized communication protocols (e.g. JSON over WebSocket). WebSocket is a way to setup a bi-directional data communication channel between the browser and the web server so that messages exchange can be initiated by either party at anytime. It is standardized in RFC 6455 [8] and supported by all browsers.

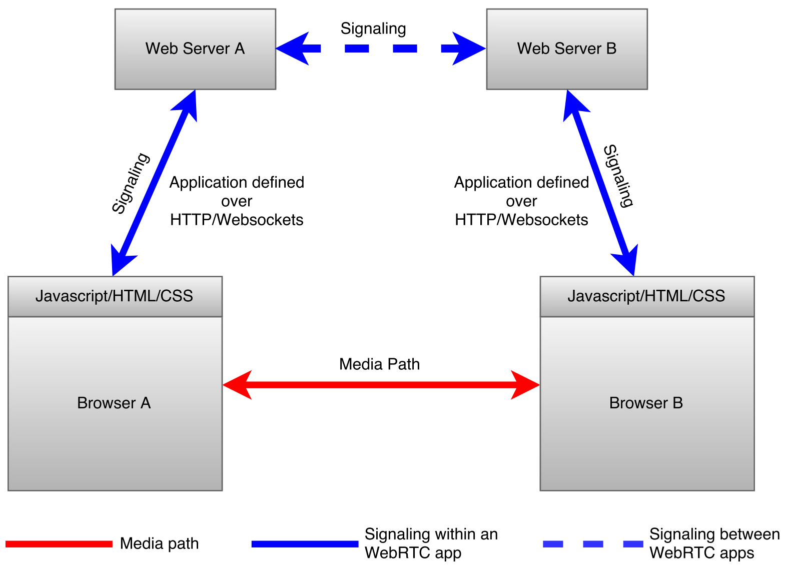
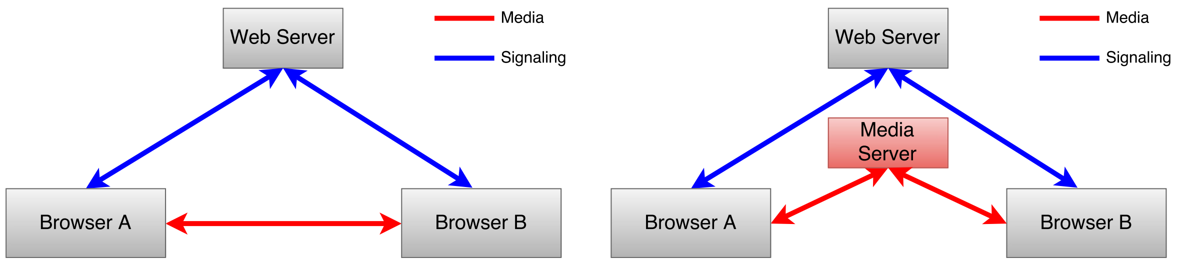

Web Server A
Signaling
Web Server B
Signaling
Application defined over HTTP/Websockets
Application defined over HTTP/Websockets
Signaling
Javascript/HTML/CSS
Browser A
Media Path
Javascript/HTML/CSS
Browser B
Media path
Signaling within an WebRTC app
Signaling between WebRTC apps

Figure 7: The WebRTC trapezoid [9]

In the basic trapezoid model, the signaling between the two servers are also not specified so the federation between the two servers can be done either using well established protocols such as SIP, or using other application protocols the two web servers agreed upon.



(a) Basic WebRTC triangle (b) With media server

Figure 8: WebRTC triangle

Variations on the basic trapezoid model will lead to different application scenarios. For example, the simplest WebRTC application model consists of two users having a communication session using the same web application. In this scenario, the two web servers in Figure 7 can be merged as one. This scenario can be called "WebRTC Triangle" as shown in Figure 8a, which is the most common WebRTC application scenario with two or more users participating in a video chat using the same URL e.g. `https://webrtc-example.com/my-chat-room`. On the media plane, the application can use either the "full mesh" model or the "centrally mixed" model which includes a media server as part of the infrastructure as shown in Figure 8b. In the “full mesh” model, each browser/endpoint participating in an N-party video conference will setup N-1 connections with the other participants. With the “centrally mixed” architecture, each conference participant will setup connection with the media server and the media server will do the mixing or selective forwarding to manage the media streams to and from each participant.

Further variations on the above application scenarios can lead to the introduction of other communication endpoints such as gateways to other communication systems (PSTN, VoIP, etc). For example, in the WebRTC triangle model, if one of the browser is replaced by a gateway to the Public Switched Telephone Network (PSTN) then the browser user can make or receive phone calls with mobile/fixed phones.

In all of the above scenarios, the browser(s) can also be replaced by custom built applications using WebRTC libraries.

2.2.4 WebRTC basic call flow

As introduced previously, WebRTC protocol suite does not specify how the call signaling is done between the browser and the web server. As long as the local and remote SDPs are provided to the browser, the RTC functions in the browser engine will use the information in SDP to setup the connections between the peers. For example, with the basic “WebRTC triangle” application model, a typical call flow can be setup as shown in Figure 9. The main steps include:

1. Browser A and B first download the application from web server.
2. Then browsers create and exchange (with the facilitation of web server) their session descriptions in the form of SDP, following the Offer/Answer model that we described in Section 2.1.1.
3. After the SDP exchange, browsers’ RTC functions will start the ICE connectivity checks and find out which transport addresses pairs can be used to transfer media/data.
4. After the connection is setup using the selected transport address pair, session keys are negotiated over DTLS (Datagram TLS) secured channel [29].
5. When key negotiation is done, the media stream will be encrypted and transmitted via SRTP [27] (Secure Real-time Transport Protocol) and data streams will transmitted over SCTP [28] (Stream Control Transmission Protocol) which can offer delivery guarantees.

2.2.5 WebRTC Protocols

The details of the networking protocols used by WebRTC are shown in Figure 10. Correlating this figure with the browser model in Figure 6, the left half of Figure 10 shows the call signaling related protocols used between browser and the web server,

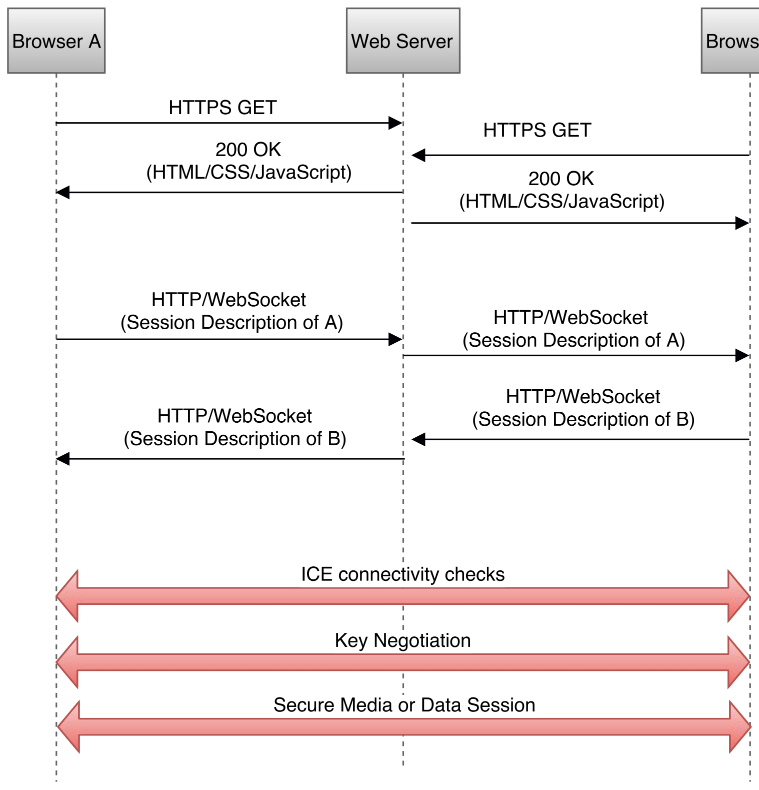


Figure 9: The WebRTC triangle call flow [37]

and the right half covers the media transport related protocols the browser RTC functions use to set up the connection between browser peers to transfer media/data streams.

From the call signaling aspects, the caller's web application needs to signal to the web server about the intent to make an outgoing call, and the server needs to inform the callee about the incoming call. SDPs of the caller and callee need to be exchanged between the peer browsers with the assistance of the web server. These application signaling messages can be wrapped in any format and can be delivered via several approaches, such as XMLHttpRequest (XHR), Server-Sent Events (SSE) or WebSocket. In order to get notification from the web server about incoming calls, XHR polling is the old/traditional way. SSE can also be used by web server to stream messages to the browser where the HTTP response content-type is set to "text/event-stream". And the most modern and flexible approach is to use WebSocket as the bi-directional communication channel between browser and web server. All three different mechanisms are encrypted over TLS as indicated on the left part of Figure 10.

The communication protocols used between browser RTC functions are the core standardization effort undertaken by the IETF RTCWEB working group. To set up connections between the browser peers, WebRTC uses ICE mechanism (introduced in Section 2.1.4) to check the connectivity of each transport address pair and establish connections on the ones that can be used to transfer media/data. WebRTC requires that all communications shall be secure and encrypted [26]. DTLS is used to

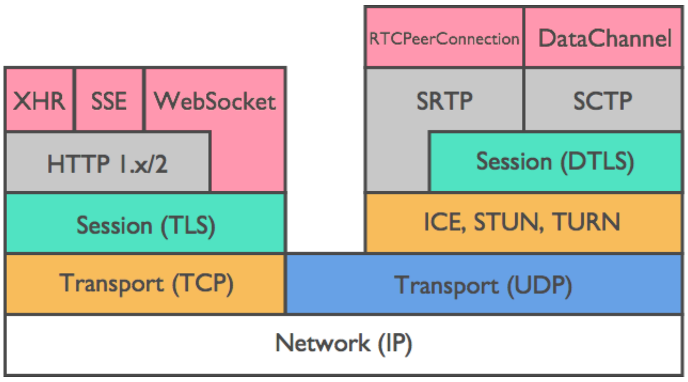


Figure 10: WebRTC protocol stack [38]

negotiate the encryption keys then media content is transported over SRTP and data is transfered via SCTP over DTLS. WebRTC uses RTP to transport media streams and explicitly specifies the mandatory RTP features that need to be supported by the implementations [25] to ensure proper interoperability. WebRTC requires endpoints must support multiplexing of DTLS and RTP over the same port pair, which ultimately was setup and kept alive via ICE [24].

2.2.6 WebRTC Use Cases

The "WebRTC Use Cases and Requirements" specification [10] mandates that WebRTC implementations must be able to support certain application use cases, including:

- Browser-to-browser video communications, possibly with multiple video streams (as device may have multiple cameras), screen sharing and file exchange.
- Multi-party on-line gaming with voice communication.
- Video conferencing system with central server.
- Browser to PSTN gateway calls, with DTMF (Dual-Tone Multi-Frequency)[11] support.

The above list looks quite exciting as it covers most (if not all) of the real-time communication features currently known in the industry for both enterprise and Internet usage context. In the next section, we will take a look at the WebRTC APIs that web application developers can use to implement the above application scenarios.

2.2.7 WebRTC APIs

WebRTC related APIs are still in the process of being standardized within the W3C community by the Web Real-Time Communications (WEBRTC) Working Group. The relevant APIs can be categorized into two groups:

[11] DTMF is a technique that use voice-frequency to send signals.

- “Media Capture and Streams API”: also known as getUserMedia API, allows applications to access media devices such as camera or microphone to capture audio/video.
- “PeerConnection API” [11] allows application to set up the connection between browser peers to deliver media streams or exchange arbitrary data.

At the time of this writing, both set of APIs are in the Candidate Recommendation (CR) [13] stage, i.e. they have been widely reviewed and met all the design requirements, no major exchanges are expected, and the working group are soliciting feedbacks about implementation experience.

2.2.7.1 Media Capture and Streams API

W3C candidate recommendation “Media Capture and Streams” specification [12] defines the set of APIs to handle generation and consumption of media streams. For example, media streams can be generated by allowing application to access local multimedia devices such as microphone and video cameras.

The API models the real-time multimedia content (audio, video) in the forms of MediaStreamTrack and MediaStream objects. A MediaStreamTrack object represents a single type of media generated from one media source. For instance, an audio MediaStreamTrack object can be generated by accessing the microphone and a video MediaStreamTrack object can be generated by accessing web cam or by capturing the screen. A MediaStream consists of a set of MedisStreamTrack objects. These audio/video tracks are grouped together with the intent to be synchronized when rendered by the browser. A MediaStream object has a single input (source) and a single output (sink) to represent the combined inputs and outputs of all the contained MediaStreamTracks. For example, the output of a MediaStream that has both audio and video tracks can be set to a HTML `<video>` element and browser will make best effort to perform proper synchronization and rendering of the audio/video tracks. The input and output of a MediaStream could also be a `RTCPeerConnection` object (which will be introduced in the next section), in this case it models the media stream being received from or sent to the peer browser.

The API to get access to local media resources is defined as:
`getUserMedia(constraints)`

Listing 1: getUserMedia example code

```
var constraints = {
  audio: true,
  video: true
};

var localVideo = document.getElementById('localVideo');

navigator.mediaDevices.getUserMedia(constraints).
    then(handleSuccess).catch(handleError);

function handleSuccess(stream) {
  // render the mediaStream in a video element
  localVideo.srcObject = stream;
  console.log("audio tracks count: " + stream.getAudioTracks().length);
  console.log("video tracks count: " + stream.getVideoTracks().length);
}

function handleError(error) {
  console.log('navigator.getUserMedia error: ', error);
}
```

Listing 1 gives an example of how to use the `getUserMedia` API to access local microphone and camera. The captured audio and video are abstracted as a `MediaStream` object and it is rendered in an HTML `<video>` element. The `constraints` parameter passed to `getUserMedia()` call specifies that the returned `MediaStream` should contain a video track and an audio track. If this call is successful (i.e. user granted the permission for the application to use microphone and camera), `handleSuccess` will be invoked with the newly created MediaStream object. If the call fails (e.g. user did not grant permission for the application to access local media source, or the computer does not have a camera), `handleError` will be invoked with an object encapsulating the details of the failure.

WebRTC API also allows application to control how the media is generated by specifying a constraint either with the `getUserMedia` call or the `applyConstraints` method on a `MediaStreamTrack` object. As an example, Listing 2 defines a constraint saying that the generated video resolution should be at least standard definition (SD) and preferably high definition (HD), and the frame rate shall be at least 20. Errors will be thrown out to the WebRTC application if the specified constraint can not be met.

Listing 2: getUserMedia constraints example

```
var videoConstraint = {
    frameRate: { min: 20 },
    width: { min: 640, ideal: 1280 },
    height: { min: 480, ideal: 720 },
}

var constraint = {
    audio: true,
    video: videoConstraint
};
```

2.2.7.2 PeerConnection API

The “WebRTC 1.0: Real-time Communication Between Browsers” specification [11] describes the APIs that can handle the network communication part of the real-time communication between browsers. The major API functionalities include:

- Connecting to browser peers using NAT-traversal technologies, i.e. ICE.

- Sending the locally-produced MediaStreamTracks to remote peers and receiving tracks from remote peers.

- Sending arbitrary data (with configurable delivery guarantees) directly to remote peers via DataChannel.

- Getting statistics about the communication. Details for this API will be covered in Chapter 3.

All the above functionalities are wrapped around the `RTCPeerConnection` API, and can be best explained by looking at some sample code. Appendix A shows an example of how the `RTCPeerConnection` API can used to setup a video chat session between two browsers and how it correlates to the various steps of basic WebRTC call flow (Figure 9).

The `RTCPeerConnection` object models a network connection between local and peer browser. The constructor of `RTCPeerConnection` takes a configuration object which specifies how the peer-to-peer connection can be setup. For example, in listing A, the configuration contains STUN and TURN server information. These configuration are needed by ICE agent in the candidate gathering phase to know what kind of transport address can be offered as candidates for receiving and sending media.

The MediaStreamTracks from local MediaStream object can be attached to the `RTCPeerConnection` by calling the `addTrack` method. When remote media streams are received on connection, application got notified via the `ontrack` event handler.

On the caller side, offer SDP can be generated by calling the `createOffer` method which contains the description of the MediaStreamTracks attached to the `RTCPeerConnection`. This SDP is passed to the PeerConnection object by calling `setLocalDescription` method. On the callee side, when it receives the SDP offer through the signaling channel, it calls the `setRemoteDescription` method to make `RTCPeerConnection` apply the supplied SDP as remote offer, then it calls `createAnswer` method to generate a SDP answer that is compatible with the caller's offer. The answer SDP is used in `setLocalDescription` to set callee's local session description and it is also sent to the caller via signaling channel. When caller receives the SDP answer, it calls PeerConnection's `setRemoteDescription` method to update local media state. Through this way, SDP offer/answer process is completed.

The ICE Agent starts gathering candidates when `setLocalDescription` or `setRemoteDescription` is called. When a new candidate is found, it is added to the `RTCPeerConnection` as well as notified to the application through the `onicecandidate` event handler. Application can either immediately send the new candidate to remote (this is called "ICE Trickling" [30] mechanism to speed up the connection establishment) or wait until ICE gathering state is "completed" and then send the SDP with the complete set of ICE candidates. When "ICE Trickling" mechanism is used, application receives the new remote candidate over signaling channel and calls the `addIceCandidate` method to updates `RTCPeerConnection` (essentially the ICE agent) about the new remote candidate. This new remote candidate will be added to ICE agent's connectivity check immediately. Media will start to flow between the two peer browsers when the ICE agent find a usable pair of transport candidates.

An `RTCPeerConnection` object has signaling state, connection state, ICE gathering state and ICE connection state. By polling these states or attaching event handlers, application can keep track of the status of the network connection and trigger corresponding application logic upon state changes. We will come back to this in later sections.

3 Quality of Real-Time Communication

In a real-time multimedia communication session, media content (audio/video) are firstly captured in their raw forms on the sender's end, then encoded and packetized before getting pushed to the network. When packets arrived at the receiver's end, they are reordered, unpacked, decoded and rendered (on speaker/monitor) to the end user. The whole pipeline from media generation till media consumption can be represented as shown in Figure 11.

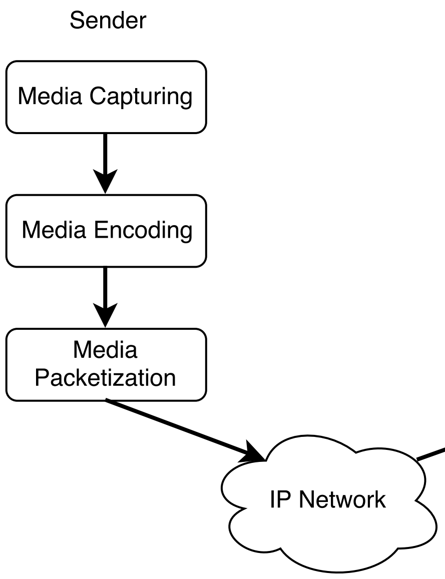


Figure 11: RTC media pipeline

For an end user, the perceived (or subjective) Quality of Experience (QoE) in a real-time interactive multimedia communication session depends on multiple factors, for instance, the codec parameters used for audio and video, the selected resolution and frame rate, the condition of the underlying computer networks, etc. In an ideal case, where wide band audio codecs and high fidelity video codecs are used to encode the audio and video streams, and the media packets are transmitted over high capacity, low latency, error free network with no congestions along the path, the user experience can be expected to be very good. In a less ideal case, network condition may not be good enough to match the bandwidth and latency requirements of the media streams and user experience would be degraded.

Quality of Service (QoS), on the other hand, refers to the objective measurements of the performance of the network based services. QoS has a strong correlation with the end user's QoE.

3.1 Quality Metrics

For real-time multimedia communication, QoE related metrics shall be measured both at the network level and the media engine level.

Measurable network QoS metrics include:

- **Latency**: Network latency (also called delay) is a measure of how long it takes a packet to travel from the sender to the receiver across the network. As IP network is based on a statistical multiplexing model, network latency is not a constant but varies over time depending on the condition of the network. Latency can be long if network is congested as the packet will be waiting in the buffer of the congested node before it is dispatched to the next hop, the packet can even get dropped which will lead to packet loss. Latency also depends on the number of hops between the two communication endpoints. If two communication endpoints are geographically far away, or they are from two Internet service providers with complicated routing and peering topologies, latency can also be high.

  For conversational audio, it is recommended by the ITU G.114 [41] that one way ("mouth-to-ear") delay shall be less than 200 ms to ensure very good real-time user experience. Delays longer than 200 ms will introduce noticeable annoyance to the some users and delays longer than 400 ms will make the conversation difficult to proceed for many users.

- **Jitter**: Jitter is a measure of the variance in latencies. As receiver receives two consecutive packets of a given media stream, the difference in the delay of these two packets can be calculated as
  $$D_{i-1,i} = (R_i - S_i) - (R_{i-1} - S_{i-1}) = (R_i - R_{i-1}) - (S_i - S_{i-1})$$
  And as packets are received continuously, the differences in the inter-arrival delays can be calculated continuously. Jitter is defined in RFC 3550 [4] as the mean deviation of D within certain time window according to the formula:
  $$J_i = J_{i-1} + (|D_{i-1,i}| - J_{i-1})/16$$
  If jitter is 0, it means packets are received in order, at fixed intervals, and there was no congestion in the network.

- **Throughput**: Throughput measures the rate (in bits/second or bps) at which streamed media are delivered to the receiver. Instantaneous throughput refers to the transfer rate within a short interval (e.g. 200ms) while average throughput refers to the transfer rate over a relatively longer period of time. Instantaneous throughput may vary over time due to either the characteristics of media codec (e.g. video I-frame, P-frame and B-frame will generate different number of packets) as well as the condition of the network. Another close related concept is **bandwidth** which refers to the average throughput that is required by a given media stream to get properly delivered. If the network capacity cannot match the media codec requirement, some dynamic adaptation at the application layer are needed so that less media data will be transfered. Such adaptations could be reducing the video resolution, or turning off the video stream, for instance.

- **Loss**: A packet sent out by the sender may not be able to reach the receiver. It can be dropped by the intermediate routers either due to error or due to network congestion. Packet loss can be detected via sequence numbering mechanism. As introduced in Section 2.1.2, each RTP media packet is associated with a sequence number. On the media receiver side, there is a playout buffer which keeps the received media packets before rendering them to the user. Within a given time interval, the media receiver sees the initial and last sequence number of the packets belonging to the playout buffer and it can check how many packets are missing from the buffer. Packet loss will lead to degradation of the rendered media. There are multiple strategies to address packet loss, such as request for retransmission, or using some loss recovery mechanism, e.g. FEC (Forward Error Correction) [42], in the media stream packetization, or just conceal it via certain algorithm during the rendering. Some level of packet loss can be tolerated but packet loss above certain threshold will surely degrade the QoE. RTP's receiver report mechanism (RR) defines two packet loss measures: the "fraction lost" metric reports the percentage of packets lost within the last reporting interval (typically a few seconds); the "cumulative number of packets lost" metric reports the total number of packets since the start of the streaming. These two metrics help keeping track of both short term and long term packet loss situation. So far there has been no standard recommendations modeling the relationship between loss and perceived video quality degradation.

All the above network QoS metrics are measured by the RTP stack for each stream (identified by SSRC) as different media streams (audio, video, data) might be treated differently while traversing the network. These metrics provide both short term and long term measurement on the condition of the media path. For example, latency and jitter can provide short-term measurement of network congestion. If network congestion is forming, latency will start to increase and also jitter will also become bigger. If network becomes quite congested, packet loss will happen and the "fraction lost" metric will give a good description of how congested the network is during the last reporting interval.

In addition to the network QoS metrics, there are other things from media characteristics and application performance perspectives that will affect end user's experience. Example media metrics include:

- **Video resolution**: i.e. the size of the video frames. Higher resolution contains more details in the image. For example, nowadays main stream TV resolution is 1080p(1920x1080) while 4K (3840x2160) TV and videos are also becoming popular.

- **Frame Rate**: This is the rate (frames per second or FPS) at which video frames are updated. Human vision will notice individual images if the frame rate is low (e.g. 10) while typical TV programs use 25 or 30 FPS[12].

When network is congested, the frame resolution or frame rate will be impacted and application needs to dynamically adapt and lower the frame rate for example. In addition, A measurement of how many video frames were sent, received, corrupted, dropped, etc., can provide important insights into the media quality of an real-time communication session.

[12]https://en.wikipedia.org/wiki/Frame_rate

From application performance perspective, metrics such as how long it takes to load the application and set up a media transport connection between the communicating parties is an important measure of the responsiveness of the application. Shorter waiting time will definitely lead to better user experiences.

All the above mentioned metrics can be measured by properly using WebRTC APIs which will be discussed next.

3.2 Quality Measurement in WebRTC

As introduced in Section 2.2, WebRTC offers APIs allowing web applications to access media devices (microphone, camera) and setup network connections with peers to deliver media streams. WebRTC specs also require implementations to measure and expose the network and media quality related metrics to the applications.

By providing the APIs to let applications become aware of the network connection state as well as the transport layer and media processing layer statistics, WebRTC makes it possible for applications to make dynamic adaptation to the underlying network condition. For example, an application can change the video resolution or it can alert its user when network issue happens.

3.2.1 Statistics API

In order to get the statistics of the underlying transport and media statistics, an application can invoke the `getStats`, on the `RTCPeerConnection` object. The API is defined as:
`Promise<RTCStatsReport> getStats(MediaStreamTrack? selector = null);`[13]
The returned report contains statistics related to the RTCPeerConnection, which includes things such as ICE candidates available for the peer-connection, certificate used to set up data and media channels, inbound-rtp (e.g. receiver) statistics, outbound-rtp (i.e. sender) statistics, codec and media stream track statistics, etc. The optional `selector` argument can be used to specify a particular `MediaStreamTrack` (introduced in Section 2.2.7.1) being sent or received over the RTCPeerConnection, and the returned report will only contain statistics about that particular stream.

The returned `RTCStatsReport` object is a collection of `RTCStats` objects and each `RTCStats` is a dictionary containing a set of values reflecting the statistics or states of certain aspects of the RTCPeerConnection at a specific time. WebRTC's Statistics API spec [14] defines all the details about what type of statistics info shall be reported and what are the mandatory fields that shall be included in each type of `RTCStats` object.

The baseline for a `RTCStats` object is:

```
dictionary RTCStats {
    DOMHighResTimeStamp timestamp;
    RTCStatsType        type;
    DOMString           id;
};
```

[13]Promise is a JavaScript mechanism to pass the result of an asynchronous function call.

The "id" is an identifier consistent across time associated with the stats object. The "type" signifies what kind of statistics the `RTCStats` object contains. For example, when the "type" is "inbound-rtp", the rest of the keys and values in the stats object are about inboud RTP stream statistics, such as "packetsReceived", "packetsLost", "jitter", etc. When the "type" is "track" the `RTCStats` object will contain media-level metrics of a `MediaStreamTrack` such as "frameWidth", "frameHeight", "framesSent", "framesPerSecond", etc., which were introduced Section 3.1.

The WebRTC statistics API specifies several different types of `RTCStats` and the mandatory metrics that shall be included in each type. WebRTC implementations such as Chrome and Firefox are required to keep track of the relevant statistics for the RTCPeerConnection object and return those statistics, in correct data format, to the application upon request.

Below is a sample `RTCStats` data reported by Firefox, containing sender's metrics for an outbound video steam track. Although it is not fully compliant with the WebRTC spec yet (as implementation is still in progress), it serves the purpose of showing what is already available for the developers to monitor the status of the WebRTC application.

```json
{
   "id": "outbound_rtp_video_1",
   "timestamp": 1492458456710.715,
   "type": "outboundrtp",
   "bitrateMean": 1248603.8604651163,
   "bitrateStdDev": 231678.58458041612,
   "framerateMean": 29.9593023255814,
   "framerateStdDev": 0.3644680195908843,
   "isRemote": false,
   "mediaType": "video",
   "remoteId": "outbound_rtcp_video_1",
   "ssrc": "3290094681",
   "bytesSent": 27148704,
   "droppedFrames": 38,
   "packetsSent": 25381
}
```

By periodically querying for statistics report, an WebRTC application can monitor the network QoS and media engine statistics throughout a communication session.

3.2.2 Network Connection State Tracking

In addition to the statistics API, The `RTCPeerConnection` object also maintains several state machines to track the status of the signaling state (i.e. the SDP exchange status), connection state (whether the two peers were connected or not), ICE gathering state, as well as the ICE connection state (whether the connectivity check succeeded or failed or still checking).

State info can be queried at any time and state transitions are exposed via event callbacks mechanism. An WebRTC application can register event listeners to the `RTCPeerConnection` object and get notified when the underlying transport layer state changes. For example, in an application, if there is a `RTCPeerConnection` object called `pc`, then the application can query the connection status of underlying transport layer via:

```
pc.iceConnectionState
```

The application can also handle connection status changes by registering an event handler via:

```
pc.oniceconnectionstatechange = function(e) {
  // logic to handle ICE connection state change
};
```

As a sample use case, an application can show proper prompt to the user when there are disruptions in the network and ICE connection state changes from “connected” to “disconnected”.

4 WebRTC Performance Measurement System

It is introduced in previous chapters that the WebRTC specification and its implementations provide a set of APIs to query media and network level statistics of the real-time communication sessions between browser peers. Using these APIs, an application can know whether the media transport connection has been setup and how the network QoS and the media characteristics look like during the communication session. All these information will help WebRTC service providers to understand the performance of their applications on the end user side. As the need for performance monitoring is common to all WebRTC applications, it is technically possible to build a cloud-based service that can perform communication quality measurement and analysis to any WebRTC applications.

4.1 Performance Monitoring as a Service

Software as a Service (SaaS) [40] model becomes popular since around 2000 with the exemplary success of companies like Concur and Salesforce[14]. SaaS is based on the multi-tenancy model where the service back-end runs in the cloud (instead of on customer's premise) and serves all subscribers/customers. On the customer side, there is usually a thin client library or API interfaces that customer can use to access the service. Data from different customers are properly managed and access controlled so that each customer can only access their own data. The biggest advantage of SaaS is on the deployment and maintenance aspects as the core services are running and maintained in the cloud, thus for the service subscribers there is no need to install or manage the service in house.

Mapping the SaaS model to the WebRTC performance monitoring requirements, we can provide a JavaScript library which acts as the thin client and performs the communication session quality measurement within the end user's browser. The library can be downloaded at the same time when an WebRTC application is loaded in the end user's browser. On the cloud side, we can run the service that collect, analyze and store the measurement data reported by the JavaScript client. WebRTC application service providers can visit the back-end service to see the measurement and analysis result. As WebRTC implementations in the browsers are still in progress, having such as measurement and monitoring system is important for the application providers to make sure that there is no malfunction or service degradation caused by some changes introduced by browser updates.

At Callstats.io[15], we are building and running such a WebRTC quality monitoring service essentially following the above model and we will show the detail of the design of the system in the following sections.

[14] https://bebusinessed.com/history/the-history-of-saas
[15] https://www.callstats.io

4.2 System Architecture

The high level solution architecture of the SaaS platform is shown in Figure 12. For a WebRTC application to monitor the performance statistics on the end users side, it needs to integrate the "callstats.js" JavaScript library. After integration, when an end user uses the web application to make audio/video calls, the relevant PeerConnection will be automatically queried for state changes and media/network statistics. The collected metrics will be sent to the collector component running in the cloud. For different users in the same call, their metrics will be correlated in the cloud by the "Conference Session Management" component so that metrics from separate users can be analyzed together to generate higher level metrics. Majority of the session or higher level metrics will be calculated as the call progresses and at the end of the call by the "Conference Quality Analyzer" component. The WebRTC application operators can visit the "Dashboard" component to check the metrics at different level of granularities. The following sections will provide more details about each of these major components. There are several other services in the system that, together with the ones shown in Figure 12, make up the complete solution. Those services will not be described here for the sake of clarity and emphasis on the big picture.

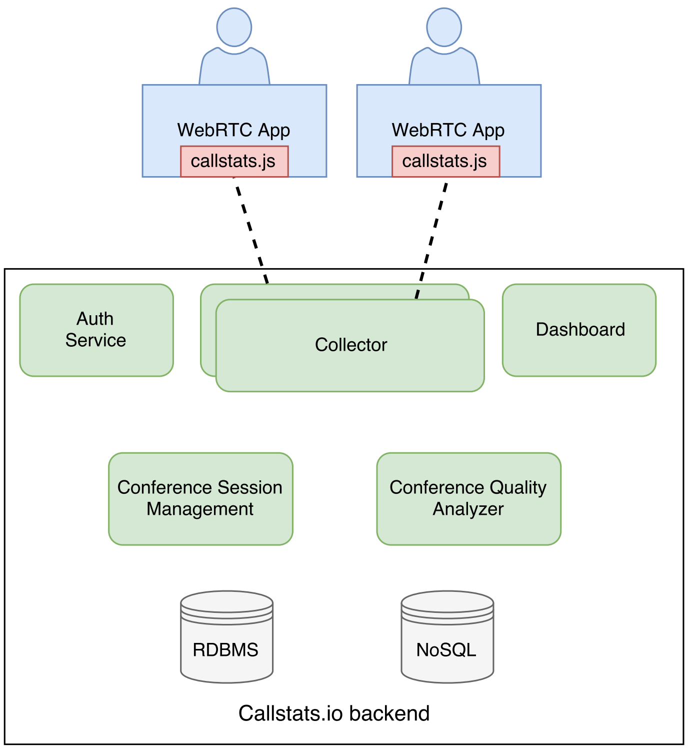


Figure 12: System architecture

4.3 Measurement Probe

The application performance measurement is done by the "callstats.js" library[16] within the end user's browser. This library monitors the peer connection status by calling the various WebRTC APIs introduced in the previous sections to fetch media and network statistics. It also registers callbacks to keep track of the signaling and network connection state changes as well as any errors reported by the PeerConnection. The collected state change events, errors, media and network statistics data, etc., are sent to Callstats.io cloud at regular or adaptive intervals.

When an WebRTC application is loaded in the end user's browser, the application also loads callstats.js library from the CDN (Content Delivery Network) via `https://api.callstats.io/static/callstats.min.js`. The callstats.js library contains the `callstats` module which exposes several APIs that the application shall integrate. Application shall call the `initialize()` methods with proper credentials to authenticate with Callstats.io's authentication service. After the authentication is done and when application is ready to start the audio/video communication with its peer application, it creates the PeerConnection object and passes it to the callstats module by calling the `addNewFabric()` method. From that point on, the callstats module will perform all the events and stats monitoring on that PeerConnection object and submit the measurement data to the Callstats.io cloud via secure WebSocket connection.

At any time during the call, the application can call `sendFabricEvent()` method to report customized events such as "audio mute", "audio resume", "dominant speaker", etc. At the end of the call, application can ask the end user to give feedback about the overall experience and then call `sendUserFeedback()` method to submit user feedback to Callstats.io backend. All the above stats, events and user's feedback will be stored in the cloud and shown to the application provider on Callstats.io dashboard.

4.4 Authentication Service

The Callstats.io backend consists of several micro services and one of them is the authentication service. The main functionality of the service is to validate that the requests are originated from valid Callstats.io customer applications and then issue data submission tokens to the clients. Data submission token will be used subsequently by the callstats module on the client side to submit data to Callstats.io's **Collector** service.

Authentication process uses the JSON Web Token (JWT) [39] mechanism and client's authenticity can be checked based on shared secrets or public/private key. The data submission token issued by the authentication service contains various configurations particular to each application based on their subscribed plan, therefore it can be utilized between "callstats.js" library and various Callstats.io micro services to specify how the collected data shall be handled for different applications.

[16] https://www.callstats.io/api/

4.5 Collector and Session Management

The collector service handles the data submitted by the "callstats.js" library from the end user's browser. Collector verifies the validity of data submission token (using a shared secret between the collector and the authentication service), and checks that the data conforms to valid data schema. After the submitted data (conference events, stats metrics, etc.) pass the validation, they are tagged with a conference session id for which the data belongs to, and then pushed to analysis and storage pipeline.

A conference session refers to the occurrence of a multi-party conference call that happened within a certain time window. All end user's data belonging to the same conference session shall be tagged with the same conference session ID for storage and analysis. The ID of a conference session consists of three components:

- an "application ID" component that identifies the WebRTC application. Customer gets it when creating an application on Callstats.io dashboard;
- a "conference ID" component supplied by the application (e.g. "weeklymeeting") which identifies a conference but might not be unique in the temporal domain;
- an additional ID issued by the "Conference Session Management" service shown in Figure 12 to add temporal uniqueness to the conference session.

The "Conference Session Management" service maintains the conference session states, for example it creates a new conference session when the first user joins a conference and it terminates a conference session after all users left the conference and no keep-alive messages have been received from the conference participants for the last 30 seconds.

4.6 Storage Component

Two different types of data storage strategy are used at callstats.io backend to store customer's configuration data and the performance monitoring data.

For customers' subscription and configuration data, the data volume is relatively small and it is very important to keep the consistencies of the data and guarantee the ACID[17] properties when customers make configuration changes. For above considerations, relational database management system PostgreSQL[18] was selected to store these application configuration data. Whenever customers login to callstats.io dashboard and change application settings, the data are persisted to PostgreSQL.

For the conferencing events and end users' performance metrics data, depending on the number of conferences occurred and the duration of the conferences, the aggregated volume can be quite big, for example, an application might submit hundreds of millions of data points per day. For this reason high performance and high scalability requirements outweighs ACID properties. Because the data format of the collected metrics might evolve over time, it is also important that the data

[17]https://en.wikipedia.org/wiki/ACID
[18]https://en.wikipedia.org/wiki/PostgreSQL

store is flexible in terms of the data schema. A couple of NoSQL[19] solutions stand out to meet the above requirements and MongoDB[20] was used at callstats.io for its high performance, high availability and ease of scaling. Each customer's metrics data are stored under a separate namespace and can be scaled and maintained in isolation of other customer's data. This way of operation fits perfectly with the SaaS model. Therefore, whenever end users are making audio/video calls, all the relevant events, stats and other conference related data submitted from the "callstats.js" library got saved to MongoDB.

4.7 Analytics Component

One major motivation for WebRTC application providers to integrate with the Callstats.io monitoring solution is to get insights into the operations of their services and find potential issues or rooms for improvement.

For each WebRTC call, analysis can be done on things such as how many participants were in the call, how many PeerConnections were used, was there any failures or retries during PeerConnection setup, the type of failures occurred, where are the end users from (Geo Locations, ISP), codecs and frame size selected, the user-agent (native application version or browser version, etc.) each participant uses, etc. Callstats.io also calculates a quality index for each media stream based on the measurement data such as delays, losses, throughput, etc., to give application provider a general idea of the performance of the communication session.

Aggregated metrics can also be calculated for the service provider. These kind of metrics include things like number of calls occurred each hour/day/month, the percentage of calls that have errors (e.g. signaling errors, network connection errors, etc.), the distribution of end user locations, distribution of the browsers, operating systems, etc.

To achieve the real-time processing capability, Apache Kafka[21] is used as the message queue to stream collected events and stats data for analysis and storage. Kafka makes it possible to persist incoming data from the collectors in a fast and scalable fashion through partitioning logic. Apache Storm[22] real-time processing system reads from Kafka, then process and analyze the data to calculated performance metrics mentioned above. This stream oriented architecture matches the requirements on reliability and scalability needed for processing large amount of incoming data, and makes it possible for the application service providers to see the status of their service metrics on Callstats.io dashboard near real time (e.g., within seconds).

4.8 Other Components

In addition to the above mentioned components, there are various other components and services providing supporting functionalities such as IP to geographic location

[19]https://en.wikipedia.org/wiki/NoSQL

[20]https://docs.mongodb.com/manual/introduction/

[21]https://kafka.apache.org/, Apache Kafka is a distributed and replicated log system.

[22]http://storm.apache.org

mapping, application log indexing, service registration and coordination, customer billing management, etc. All these services work together to provide a scalable and reliable WebRTC application performance monitoring solution.

5 Performance Measurement Observations

In Chapter 2 and 3 we introduced WebRTC APIs and the set of methods that can be used to gather network transport statistics and media characteristics of a communication session between peer browsers. In Chapter 4 we explained how a measurement platform is built at Callstats.io to provide performance monitoring service for WebRTC application service providers, so that they can focus on the customer engagement features while relying on Callstats.io service to takes care of the communication quality monitoring. In this chapter, we will show some measurement results to demo certain performance characteristics of real-world WebRTC services.

Many things can be measured with Callstats.io service. For example, the measurement probe (introduced in Section 4) gathers information such as: the user-agent used for the communication, ICE related events (connection state changes, candidates, the selected transport pair, etc), the SDPs exchanged between the communication peers, the A/V codecs and other media characteristics (frame size, frame rate), and the network metrics such as latency, throughput, packet loss, etc.

As mentioned in Section 4, Callstats.io customers (which are typically developers or service operation people) can log on to the dashboard and check the performance of their WebRTC applications for both aggregated metrics and the details of specific conferences. Figure 13 shows a screenshot of the dashboard displaying the throughput info of a particular user during a call.

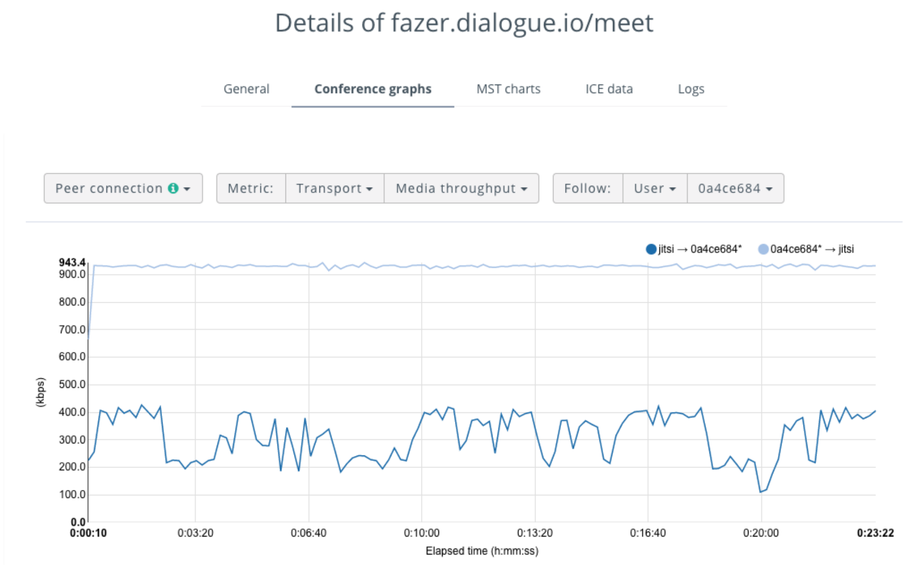


Figure 13: Sample dashboard screenshot

In the rest of this chapter, we will show some aggregated metrics calculated using data collected from selected customer applications to give a big picture of how real-world WebRTC application performance looks like. These applications include web applications as well as native desktop or mobile applications built with tools or SDKs that are WebRTC compliant. All these applications embed "callstats.js" probe and rely on Callstats.io to monitor the quality of their end users' communication sessions.

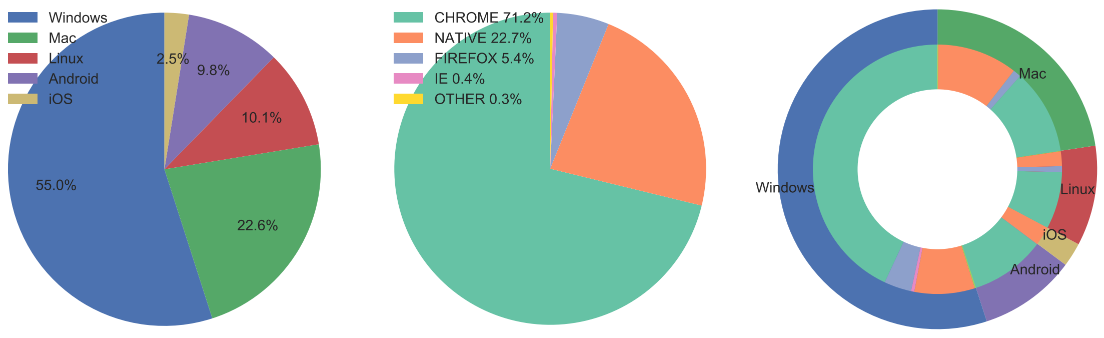
The dataset used to generate the statistics in the following sections consists of 1.5 million peer-connections measured between mid-July and mid-August in 2017.

5.1 Endpoint Statistics

Figure 14 shows the distribution of end user's operating system (OS) and browser types. It is pretty clear to see that Windows is the most popular operating system that people use to run WebRTC applications. Android and iOS together account for more than 12% of the user sessions, showing the trend of people using mobile devices to make WebRTC calls.

The percentage of Chrome usage is a bit over 70% which is on-par with the browser's market share[23]. As for other browsers, Safari announced support for WebRTC in June 2017[24] with Opus and H.264 codecs. Microsoft also announced support of WebRTC 1.0 in IE and Edge early 2017[25]. Opera has been supporting WebRTC since the beginning as it shares the core engine with Chrome. Our measurement indicates the usage of these browsers in WebRTC calls are less than 1%.

The OS and user-agent joint distribution chart shows that Android users predominately use Chrome for WebRTC calls while iOS users seem to only use native applications, which can be explained by Apple's policy of not allowing 3rd party browser engine to run on iOS[26].



(a) Operating system type (b) User-agent type (c) OS vs. user-agent

Figure 14: Endpoint distributions

End user's geographic location can be derived by examining the IP address he/she used to access the Internet. In WebRTC context, the "Server Reflexive" ICE candidate (introduced in Section 2.1.4) contains user's IP seen on the public side of the user's NAT. Table 1 lists the top 9 countries that has the highest number of end user sessions monitored by Callstats.io. This table indicates US is leading in terms

[23] 76% visits came from Chrome per https://www.w3schools.com/browsers
[24] https://webkit.org/blog/7726/announcing-webrtc-and-media-capture
[25] https://blogs.windows.com/msedgedev/tag/webrtc/
[26] https://www.howtogeek.com/184283/why-third-party-browsers-will-always-be-inferior-to-safari-on-iphone-and-ipad/

of WebRTC technology penetration. Considering the relative smaller population of European countries (DE, FR, GB), the table indicates WebRTC adoptions in Europe is about half of US.

Country Code	US	BR	FR	DE	IN	CN	GB	ZA	CA	Others
Percentage (%)	48.7	5.8	3.5	3.2	2.6	2.5	2.4	2.4	2.4	26.6

Table 1: End user distribution by country

5.2 Media Statistics

To ensure basic level of inter-operability, WebRTC requires implementations to support audio codecs such as Opus, PCMA and PCMU according to RFC 7874 [32]. WebRTC also requires implementations to support VP8 video codec and H.264 Constrained Baseline profile according to RFC 7742 [33].

Table 3 shows the video codec distribution observed from our selected calls. It is clear that VP8 is used by the majority of the calls. VP9 video codec is not mandatory in WebRTC but it is built into Google's WebRTC library and browser. We can see from Table 3 that some applications choose to use VP9 over VP8 and H.264. H.264 is the least popular video codec in WebRTC world per our observation.

Table 2 shows the audio codec usage and it is clear that Opus [34] is the audio codec for WebRTC (used by more than 99% calls).

Audio codec	Percentage (%)
Opus	99.71
PCMU	0.20
Other	0.09

Table 2: Audio codec distribution

Video codec	Percentage (%)
VP8	94.34
VP9	3.51
H.264	2.15

Table 3: Video codec distribution

5.3 Network Statistics

In this section, we will look at the network performance metrics such as latency, loss and throughput in the reported dataset. These metrics has strong correlation with end user's quality of experience.

From throughput perspective, higher throughput usually leads to better user experience. Figure 15 shows the distribution of average audio throughput. The distribution indicates that more than 70% of the audio streams use throughput lower than 40 kbps and about 20% audio streams use bit rate lower than 20kbps. This is compatible with the codec distribution shown in Table 2 as Opus is the dominant codec for WebRTC calls, and Opus works well within the 21-48 kbps range for speech audio so we can argue that majority of audio sessions have pretty good quality.

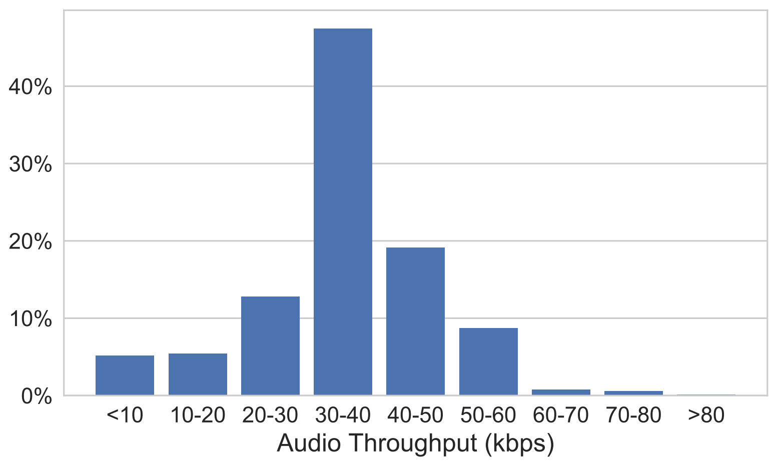


Figure 15: Audio throughput

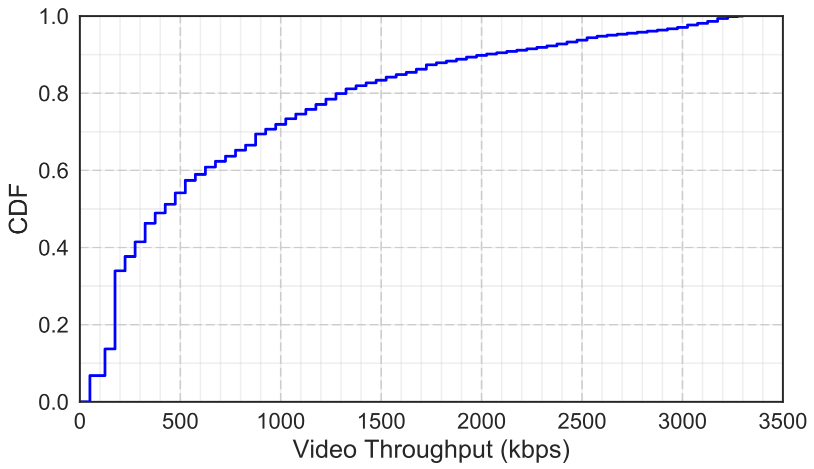


Figure 16: Video throughput

Figure 16 shows the distribution of average throughput for the video streams. It can be seen that on average, half of the video sessions use bit rate less than 400 kps, and about 30% of the video streams use bit rate higher than 1 Mbps, which can usually provide good video quality (e.g., 480P). There are about 10% video streams use throughput higher than 2 Mbps, that is typically good enough for HD quality video.

Figure 17 shows the 95th percentile Round-Trip Time (RTT) for the WebRTC calls in our reported dataset. It was introduced in Section 3.1 that one way delay less than 200 ms usually provide good real-time conversational audio experience. One way delay longer than 400 ms will lead to bad user experience for VoIP calls. Looking at the figure with the user experience model in mind, we can see that around 90% of our monitored calls have 95th percentile RTT shorter than 400 ms. So our measurement just confirmed the fact that the Internet is good enough, most of the time, to provide real-time communication services. There are about 3-4% of the calls for which the 95th percentile RTTs are longer than 800 ms. We can infer that for those calls, there must be some periods during which the network performance were degraded (e.g., the network is congested or user got switched to some low performance network) and the end users experienced some cluttering audio or choppy video.

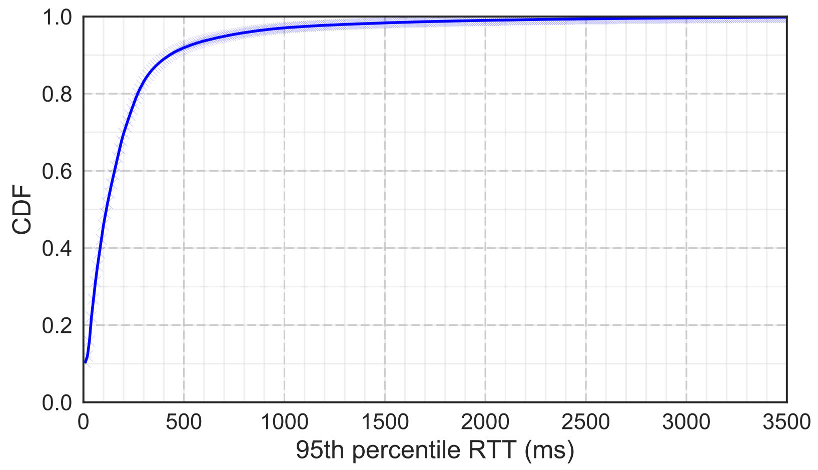


Figure 17: 95th percentile RTT

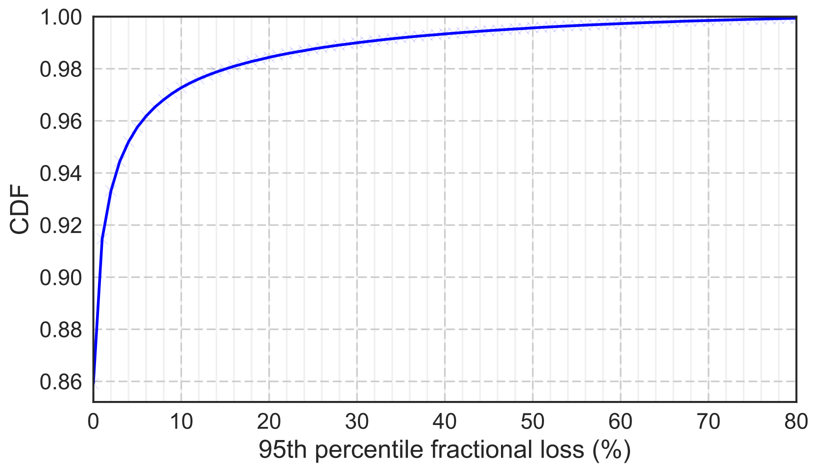


Figure 18: 95th percentile fractional loss

In Section 3.1 we mentioned that RTP protocol defined the mechanism for receivers to periodically report the fraction of packets lost during the last reporting interval. When the loss is low (e.g. less than 5%), user shall experience no major media quality degradation. If this metric is high, then the user will experience choppy or black out videos.

Figure 18 shows the distribution of the 95th percentile fractional loss observed on the PeerConnection sessions in our report dataset. It is nice to see that 85% of the sessions have 95th percentile of packet loss of 0, and about 95% of the sessions report 95th percentile loss lower than 5%. Smaller loss leads to better audio and video experiences. This observation on packet loss is really encouraging to the WebRTC service providers as we had just showed above that 90% session have a satisfactory 95th percentile RTT. Combining these two pieces of information about loss and latency, we can claim that the majority of the calls in our reported dataset are performing reasonably well from network transport perspective. Of course a satisfactory end user experience also include things such as selected video resolution, audio bandwidth, etc.

5.4 ICE Performance

ICE mechanism is used in WebRTC to set up the media path. In previous sections, we introduced ICE algorithm steps and how WebRTC API exposes the ICE state machine. Callstats.io collects and records those ICE state related information during the PeerConnection setup. In this section, we first look at the gathered ICE candidates, then we show the latency metrics which characterize how long each ICE step takes and, more importantly, how long it takes overall to setup the media transport channel between the communicating peers.

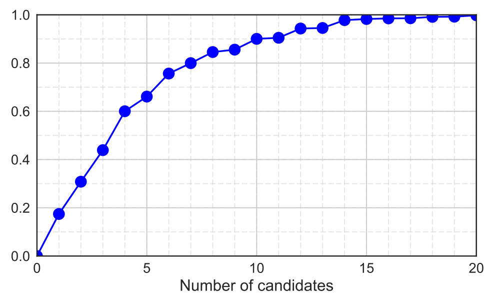
5.4.1 Gathered ICE Candidates

The number of ICE candidates gathered depends on a few factors, such as the application's configuration (e.g. whether STUN and TURN servers are configured); the number of media transport channels that application wants to use, e.g. should it use separate streams to transmit audio and video or should it bundle [35] multiple media streams over the same transport channel; whether RTP and RTCP can be multiplexed over the same transport [36]; and the number of network interfaces available and IP addresses bounded to those interfaces, etc.

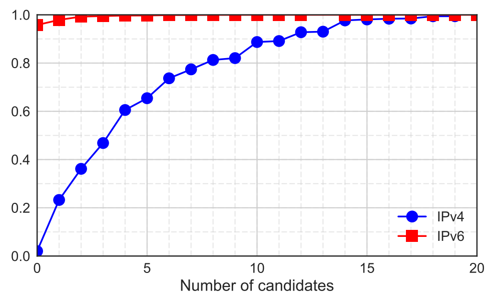
Figure 19a shows the total number of ICE candidates gathered per PeerConnection. Each candidate can be represented as a (`IP`, `port`, `transport`) triplet where the `transport` can be either UDP or TCP. From the figure, we can see that 90% of the sessions has 10 or less candidates. And 99.8% of the PeerConnections in our reported dataset has less than 20 candidates. It is interesting to note that there are 17% of PeerConnections only have one ICE candidate. The relevant application must have some very special setting to make this happen, which might deserves further analysis.

Figure 19b shows the IP address type of the ICE candidates. We can see that the curve for IPv4 address distribution is almost the same as the the distribution in Figure 19a, which signifies the dominance of IPv4 address usage. Less than 5% of the PeerConnections have IPv6 candidates.

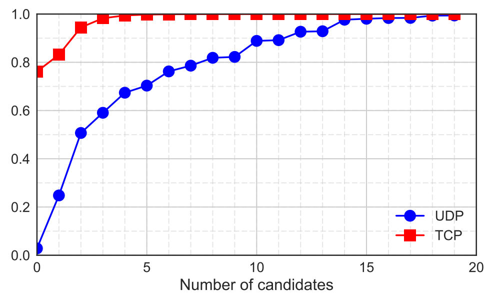
Figure 19c shows that around 80% of PeerConnections do not have a TCP transport candidate and 99% of the PeerConnections have less than 4 TCP candidates. TCP transport offers better NAT traversal capability than UDP, but because of the elastic nature of TCP, UDP shall be the preferred transport to use, unless only TCP



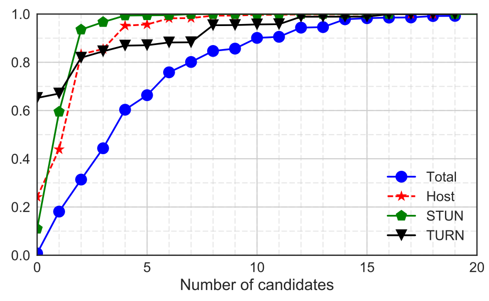
(a) Total



(b) IPv4 vs. IPv6



(c) UDP vs. TCP



(d) Candidate types

Figure 19: ICE candidates per PeerConnection

candidates can succeed the connectivity check.

Figure 19d shows the distribution of the candidate address type (introduced in Section 2.1.4). It is interesting to see that about 65% of the PeerConnections do not have TURN candidates, in these cases the successful setup of these PeerConnections will have to depend on either there are publicly reachable host addresses or the STUN address can be used to traverse the endpoint's NAT. It is also interesting to see that about 22% of the PeerConnections have 0 Host address candidate. This might be due to the application's configuration that the PeerConnection is forced to use only "relayed" candidate (e.g. by setting the `iceTransportPolicy` option in the PeerConnection configuration), so there was no need to gather "Host" candidates. This approach can the significantly optimize and accelerate the connectivity checks step.

5.4.2 Active ICE Candidates

The statistics of ICE candidates that succeeded connectivity checks are shown in Table 4, 5 and 6.

Table 4 shows that predominantly (more than 98%) the active ICE candidate pairs (i.e. the ones passed the connectivity checks) use IPv4 addresses. There are about 1% PeerConnections that have an active transport channel using IPv6 address.

Table 5 shows that active media paths are predominantly (around 97%) using only UDP as the transport. And there are close to 3% PeerConnections only have

IPv4	IPv6	IPv4 and v6
98.66%	0.96%	0.38%

Table 4: Address type distribution for active ICE candidates

UDP	TCP	UDP and TCP
96.86%	2.93%	0.21%

Table 5: Transport type distribution for active ICE candidates

TCP transport available. This might be due to the NAT configuration in enterprise environment where UDP traffics are strictly filtered and limited. A very small portion of the PeerConnections have both UDP and TCP transport available for the media path.

Host	STUN	Relay	Mixed
6.85%	81.03%	7.59%	4.52%

Table 6: Active ICE candidate types distribution

Table 6 indicates that about 6-7% PeerConnections can be successfully established just using the endpoint's host address. In these cases, the communicating peer is either located in the same network or the endpoint's host address is publicly reachable. More than 80% PeerConnections' STUN address have successfully passed connectivity checks which justified the effectiveness of NAT traversal via the server or peer reflexive address. There are about 7% PeerConnections that have to rely on relay server to set up the media path due to the most restrictive NAT policy.

5.4.3 ICE Latencies

In this section, we examine how long it takes for ICE to work through each steps to eventually setup the transport connection between peers. More specifically, we measure the following metrics:

- **Gathering Delay**: This is the interval from the PeerConnection's candidates gathering starts (e.g. state changed from "new" to "gathering") till the gathering state is changed to "complete", at which point all possible candidates have been gathered.

- **Connectivity Check Delay**: This is the the interval when ICE starts the connectivity probing till the ICE agent finishes checking all candidate pairs against one another and has found a usable connection. Candidate pairs are prioritized by the ICE agent. Higher priority pairs are checked first, and successful candidate pair can already be used to transmit media. Lower priority pairs can be kept as backup transports after they succeed the checks. Most WebRTC implementations support "ICE trickle" [30] where ICE candidates are exchanged immediately (through signaling channel) as they are gathered, thus connectivity check phase can already start before the gathering phase completes.

- **Time to First Media**: This is the time since the PeerConnection is created till the first remote audio/video frame is received and ready to be rendered in the browser. This is an important user experience indicator.

Measuring the above latencies provides a general profile about how much time it takes for the media transport to setup and how much time an end user needs to wait, after call initiation, before he/she sees the video from remote.

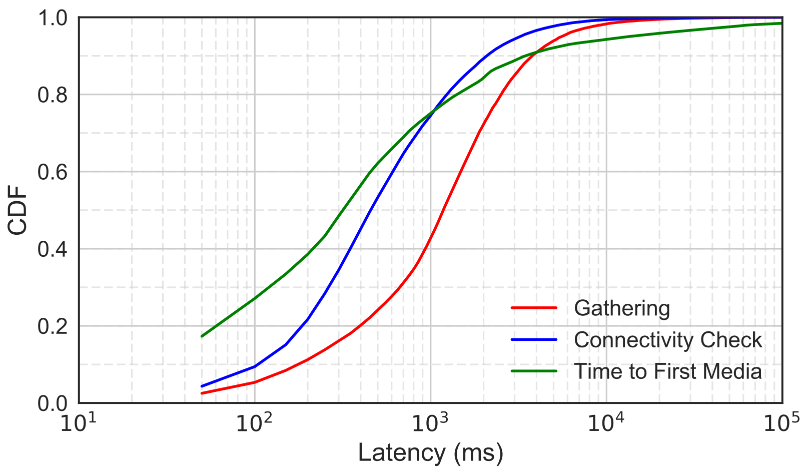


Figure 20: ICE-related call setup delay

Figure 20 shows the distribution of measurement results for the above three ICE latencies. We can see that about 50% of the PeerConnections finished gathering all ICE candidates under 1 s. And more than 90% of the PeerConnections completed gathering in less than 5 s. But for about 2% of the sessions, the ICE gathering phase took more than 10 s to conclude.

For connectivity check delay, we observed that about half of checks are finished within 500 ms, and 90% finished within 2 s. And more than 99% connectivity checks finished within 10 s. The ones that took longer time to finish are most likely due to ICE trickle where ICE gathering phase might have took long to complete hence the connectivity checks also took long to conclude.

As to the "Time to First Media" measurement, we can see that 50% of the user sessions took less than 300 ms to render the first media frame. One should note that the median for connectivity check is around 500 ms, this is exactly how ICE is designed to work, i.e. the successful transport can already be used to transmit media while further connectivity checks of lower priority candidate pairs are still in progress. About 90% of the user sessions starts to receive remote media within

10 s. Some of the longer delay can be caused by longer connectivity checks to find successful candidate pairs and some can also be caused by applications requiring the user to manually grant permissions to accept the call.

5.5 Signaling Overhead

As mentioned in Section 2.2, WebRTC does not define specific signaling protocols for call initiation and call control. ICE candidates and media codec info are exchanged between the peers via SDP and facilitated by each application's own signaling mechanism. The SDP exchange can be considered as the signaling overhead for peer connection establishment. Callstats.io collects and records SDP info for each WebRTC call.

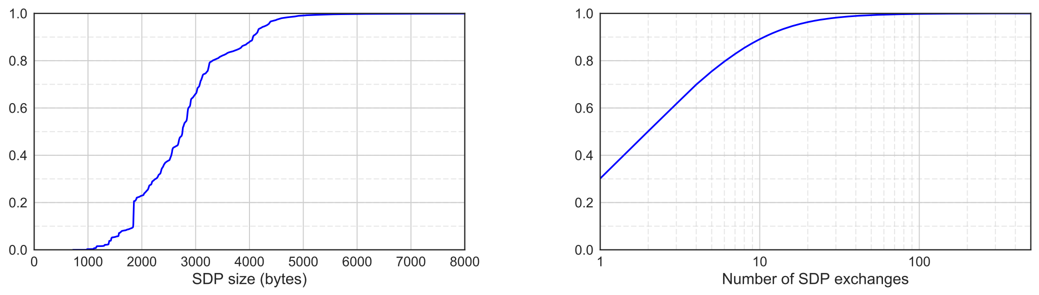


Figure 21: SDP size and exchange frequency

Figure 21 shows the size of the SDPs and how often SDP exchange happened in the reported dataset. This figure shows that 80% of the SDP descriptions are larger than 2 KB and almost all SDPs are less than 6 KB. SDP exchange can happen when audio/video streams are added or removed from the communication session. The figure also indicates that half of the calls only need 1 or 2 SDP exchanges, and 90% of the sessions involve less than 10 SDP exchanges. But there are some portions of sessions involve several hundreds of SDP exchange, this can happen when there are many participants in a SFU (Selective Forwarding Unit) bridged conference. In those cases, the media stream info for different participants are all listed in the SDP. So when a participant joins or leaves the conference, SDP updates will be sent to all other participants so that the end user's application can render video streams properly for all the participants. Same thing can happen when different participants enable or disable screen share, for example. If the conference duration is long, and there are more user join/leave activities, one can expect that there will be quite many SDP exchanges involved.

6 Summary

In this thesis, we glossed over the general real-time communication techniques for call signaling and media path setup. We introduced the basics of WebRTC technology that web application developers can use to add real-time communication functionality to web applications via W3C standard APIs. We then discussed the major factors affecting the end user's experience in a communication session and how those factors can be measured from WebRTC perspective.

As the core contribution of this thesis, we showed the architectural design of Callstats.io, a cloud-based service platform that helps WebRTC application providers to monitor the performance of their applications. Callstats.io utilizes standard APIs to keep track of signaling and network state changes, also collect network and media statistics of the end user's communication sessions. The measurement results are sent to the back-end in the cloud for further analysis and storage. This service has been running online since early 2015 and up to today hundreds of customers have signed up and integrated the "callstats.js" SDK inside their applications to monitor the quality of the audio/video calls they are serving to their end users. These integrations provide us the opportunity to collect real-time communication performance data at large scale.

We used a subset of the collected data to generate a profile showing how real-world WebRTC applications perform in terms of latency, loss, throughput, codec usage etc. Based on the statistics of our measurement we can see that the majority of these real-time communication sessions are doing reasonably well in terms of latency and packet loss. The throughput performance is good for about 30% of the sessions (e.g., above 1Mbps which is usually good enough for 480p video resolution).

For WebRTC application providers, these measurements can help them understand their application's performance either for individual calls or aggregated across different geographic regions, throughout a period of time. Based on the metrics, they can try to make adjustment to their application deployment configurations (such as setting a better constraint based on user's context; using relay servers closer to the end users; deploy more application servers to the places where majority of their users are located, etc.) so that the overall application performance can be improved over time, and the improvement should be easily reflected by the metrics collected or calculated by Callstats.io.

We are aware that the metrics reported in this thesis is limited and biased by the applications integrated with Callstats.io. But, as the dataset consists of more than 1 million measured user sessions sampled from about one hundred different applications serving end users world-wide, the statistics we generate should be a high confidence reflection of the performance of web based real-time communication at large. We are also aware that there are still some widely used real-time communication tools that use proprietary signaling and media protocols or codecs, and the media streams might be transmitted over managed or overlay networks. They might be able to outperform the Internet based WebRTC applications on some aspects, but there is no way for us to verify this.

Due to the limitation of time, the statistics studied in this thesis only scratched the surface of performance evaluation in WebRTC. Further study shall be done with finer granularity of context. For example, metrics can be studied based on geographic locations, Internet service providers, connectivity types (wired vs. wireless, public vs. enterprise networks), conferencing topologies (P2P vs. relayed), operating system types, user-agent groups, etc., so that the performance characteristics for each particular context can be profiled to give more clear picture of the status quo in WebRTC performance. Furthermore, those detailed information can be used as a feedback and source of input to various stakeholders to further improve the quality of their real-time communication services.

References

[1] D. Cohen. *Specifications for the Network Voice Protocol (NVP).* RFC 741, November 1977.

[2] *H.323 : Packet-based multimedia communications systems.* ITU-T recommendation H.323, February 1998.

[3] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, etc. *SIP: Session Initiation Protocol.* RFC 3261, June 2002.

[4] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications.* RFC 3550, July 2003.

[5] E. Brosh, S.A. Baset, D. Rubenstein, H. Schulzrinne. *The delay-friendliness of TCP.* ACM SIGMETRICS Performance Evaluation Review, 36(1), pp. 49-60, 2008.

[6] H. Schulzrinne, S. Casner. *RTP Profile for Audio and Video Conferences with Minimal Control.* RFC 3551, July 2003.

[7] C. Perkings. *RTP: Audio and Video for the Internet.* Addison Wesley, 2003.

[8] I. Fette, A. Melnikov. *The WebSocket Protocol.* RFC 6455, December 2011.

[9] H. Alvestrand. *Overview: Real Time Protocols for Browser-based Applications.* Internet Draft draft-ietf-rtcweb-overview-18, March 2017.

[10] C. Holmberg, S. Hakansson, G. Eriksson. *Web Real-Time Communication Use Cases and Requirements.* RFC 7478, March 2015.

[11] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, B. Aboba, T. Brandstetter. *WebRTC 1.0: Real-time Communication Between Browsers.* W3C Candidate Recommendation, November.

[12] D. C. Burnett, A. Bergkvist, C. Jennings, A. Narayanan. *Media Capture and Streams.* W3C Candidate Recommendation, October 2017.

[13] *World Wide Web Consortium Process Document.* World Wide Web Consortium (W3C), October 2005.

[14] H. Alvestrand, V. Singh. *Identifiers for WebRTC's Statistics API.* W3C Working Draft, November 2017.

[15] M. Handley, V. Jacobson, C. Perkins. *SDP: Session Description Protocol.* RFC 4556, July 2006.

[16] J. Rosenberg and H. Schulzrinne. *An Offer/Answer Model with the Session Description Protocol (SDP).* RFC 3264, June 2002.

[17] D. Bryan, B. Lowekamp, C. Jennings. *SOSIMPLE: A Serverless, Standards-based, P2P SIP Communication System.* First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA'05), pp. 42–49, IEEE Computer Society, 2005.

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

[19] J. Rosenberg, R. Mahy, P. Matthews, D. Wing. *Session Traversal Utilities for NAT (STUN).* RFC 5389, October 2008.

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

[21] G. Camarillo, M. Martín. *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds* (3rd Edition). Wiley, September 2008.

[22] J. Wagner. *What Developers Should Know About ORTC Versus WebRTC.* https://www.programmableweb.com/news/what-developers-should-know-about-ortc-versus-webrtc/analysis/2015/10/12.

[23] Google. *Introducing WebRTC – An Open Realtime Communications Project.* https://webrtc.org/blog/2011/05/03/introducing-webrtc-an-open-realtime-communications-project.html. May 03, 2011.

[24] H. Alvestrand. *Transports for WebRTC.* Internet Draft draft-ietf-rtcweb-transports-17, October, 2016.

[25] C. Perkins, M. Westerlund, J. Ott. *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP.* Internet Draft draft-ietf-rtcweb-rtp-usage-26, March, 2016.

[26] E. Rescorla. *WebRTC Security Architecture.* Internet Draft draft-ietf-rtcweb-security-arch-13, October, 2017

[27] M. Baugher, D. McGrew, M. Naslund, E. Carrara, K. Norrman. *The Secure Real-time Transport Protocol (SRTP).* RFC 3771, March 2004.

[28] R. Stewart (Editor). *Stream Control Transmission Protocol.* RFC 4960, September 2007.

[29] E. Rescorla, N. Modadugu. *Datagram Transport Layer Security Version 1.2.* RFC 6347, January 2012.

[30] E. Ivov, E. Rescorla, J. Uberti, P. Saint-Andre. *Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol.* Internet Draft draft-ietf-ice-trickle-14, September, 2017.

[31] J. Uberti, C. Jennings, E. Rescorla. *JavaScript Session Establishment Protocol.* Internet Draft draft-ietf-rtcweb-jsep-24, October, 2017.

[32] JM. Valin, C. Bran. *WebRTC Audio Codec and Processing Requirement.* RFC 7874, May 2016.

[33] A.B. Roach. *WebRTC Video Processing and Codec Requirements.* RFC 7742, March 2016.

[34] JM. Valin, K. Vos, T. Terriberry. *Definition of the Opus Audio Codec.* RFC 6716, September 2012.

[35] C. Holmberg, H. Alvestrand, C. Jennings. *Negotiating Media Multiplexing Using the Session Description Protocol (SDP).* Internet Draft draft-ietf-mmusic-sdp-bundle-negotiation-39, August 31, 2017.

[36] C. Perkins, M. Westerlund. *Multiplexing RTP Data and Control Packets on a Single Port.* RFC 5761, April 2010.

[37] A. Johnston, D. Burnett. *WebRTC APIs and RTCWEB Protocols of the HTML5 Real-Time Web* (3rd Edition). Digital Codex LLC, 2014.

[38] I. Grigorik. *High Performance Browser Networking* (1st Edition). O'Reilly Media, 2013

[39] M. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT).* RFC 7519, May 2015.

[40] P. Mell, T. Grance. *The NIST Definition of Cloud Computing.* NIST, Sept. 2011

[41] ITU. *One-way transmission time.* ITU G.114, May 2003.

[42] C. Perkins, O. Hodson. *Options for Repair of Streaming media.* RFC 2354, June 1998.

A PeerConnection API Usage Example

```
// setup a signaling channel with the web application server
var signalingChannel = new SignalingChannel();

var pcConfig = {
    iceServers: [{
        "url": "stun:stunserver:8888"
    }, {
        "url": "turn:user@turnserver:9999",
        "credential": "password"
    }]
};

// create peer connection
var pc = new RTCPeerConnection(pcConfig)

navigator.getUserMedia({
    "audio": true,
    "video": true
}).then(gotStream).catch(handleError);

function gotStream(stream) {
    // attach local media stream to the peerconnection
    stream.getTracks().forEach((track) => pc.addTrack(track, stream));

    if (isCaller()) {
        pc.createOffer(function(offer) {
            pc.setLocalDescription(offer);
            // send SDP offer to callee, facilitated by server
            signalingChannel.send(offer.sdp);
        });
    }
}

pc.onicecandidate = function(event) {
    if (event.candidate) {
        signalingChannel.send({
            type: 'candidate',
            label: event.candidate.sdpMLineIndex,
            id: event.candidate.sdpMid,
            candidate: event.candidate.candidate
        });
    }
}

signalingChannel.onmessage = function(message) {

    if (message.type === 'candidate') {
        // handle ICE candidate from remote
        var candidate = new RTCIceCandidate({
            sdpMLineIndex: message.label,
            candidate: message.candidate
        });
        // tell browser about new ICE candidate from remote
        pc.addIceCandidate(candidate);
    }

    if (message.type === 'answer') {
        // caller handle SDP answer from remote
        pc.setRemoteDescription(new RTCSessionDescription(message));
    }

    if (message.type === 'offer') {
        // callee handle SDP offer from remote
        pc.setRemoteDescription(new RTCSessionDescription(message));
        pc.createAnswer(function(sdpAnswer) {
            pc.setLocalDescription(sdpAnswer);
            signalingChannel.send(sdpAnswer);
        });
    }
}

// once media for a remote track arrives, show it in the remote video element
pc.ontrack = function(event) => {
  var remote_video = document.getElementById('remote_video');
  remote_video.srcObject = event.streams[0];
};
```