An Efficient Filtering Technique to Avoid the Congestion Control

R. Prathibha Spurthi, Dr. M. Nagaratna
PG Scholor,
Department of CSE, Jawaharlal Nehru Technological University College of Engineering Hyderabad
Associate Professor,
Department of CSE, Jawaharlal Nehru Technological University College of Engineering Hyderabad

Abstract : WebRTC has quickly become popular as a video conferencing platform, partly due to the fact that many browsers support it. WebRTC utilizes the Google Congestion Control (GCC) algorithm to provide congestion control for real- time communications over UDP. The performance during a WebRTC call may be influenced by several factors, including the underlying WebRTC implementation, the device and network characteristics, and the network topology. In this paper, we perform a thorough performance evaluation of WebRTC both in emulated synthetic network conditions as well as in real wired and wireless networks. Our evaluation shows that WebRTC streams have a slightly higher priority than TCP flows when competing with cross traffic. In general, while in several of the considered scenarios WebRTC Performed as expected, we observed important cases where there is room for improvement. These include the wireless domain and the newly added support for the video codec's VP9 and H.264 that does not perform as expected.

IndexTerms **– WebRTC, Congestion Control, Performance Evaluation**

I. INTRODUCTION

WebRTC provides Real-Time Communication (RTC) capabilities via browser-to-browser communication for audio (voice calling), video chat, and data (_le sharing). It allows browsers to communicate directly with each other in a peer-to-peer fashion, which differs from conventional browser to web-server communication. One of the main advantages of WebRTC is that it is integrated in most modern web browsers and runs without the need to install external plugins or applications. The World Wide Web Consortium (W3C) [4] has set up an Application Programming Interface (API), which allows developers to easily implement WebRTC using JavaScript, while the Internet Engineering Task Force (IETF) [14] defines the WebRTC protocols and underlying formats.

To realize the low latency and high throughput necessary IFIP WG 7.3 Performance 2017. Nov. 1416, 2017, New York, NY USA Copyright is held by author/owner(s). For real-time communication, WebRTC prioritizes transmit- ting data using UDP instead of TCP. WebRTC over TCP is used as a last resort, when all UDP ports are blocked, which can be the case in heavily-protected enterprise net- works. Since UDP does not support any form of congestion control, WebRTC uses a custom-designed congestion control algorithm that adapts to changing network conditions. With the high-level API, WebRTC makes it easy for application developers to develop their own video streaming applications. The disadvantage of this high-level approach is that the performance details, especially the way conges- tion is handled, are completely hidden from application developers. At the same time, recent research evaluating the performance of WebRTC has only partially addressed this gap (see Section 7 for more details). In this paper, we take a closer look at the performance of WebRTC, mainly focusing on the Google Congestion Control (GCC) algorithm, which is the most widely used congestion control algorithm for WebRTC. We evaluate its performance using the latest web browsers across a wide range of use cases. Our key contributions consist of studying the effects of different synthetic network conditions on the latest implementations of WebRTC, comparing WebRTC's performance on mobile devices, analyzing the performance of the newly added video codec's VP9 and H.264, and evaluating the impact of wired and wireless networks on WebRTC. The source code for reproducing the experimental conditions described in this paper is available at: https://github.com/Wimnet/webrtc_performance In particular, our experimental study includes the following:

Baseline Experiments: I study the effects of varying latency, packet loss, and available bandwidth by emulating different performance environments using Dummy net. We establish benchmarks for the performance of WebRTC in different scenarios.

Cross Traffic: I study the effects of TCP cross traffic and multiple WebRTC streams sharing the same bottleneck. Our evaluations indicate that with the re-cent enhancements to the congestion control mechanism, WebRTC streams receive slightly higher priority when competing with TCP flows.

Multi-Party Topology: We compare the performance of a mesh and Selective Forwarding Unit (SFU) based topologies for group video calls using WebRTC. Our evaluation highlights inherent trade-o s between performance and deploying additional infrastructure for multi-party video calls.

Video Codec's: We study the performance of three widely used video codec's, VP8, VP9, and H.264, on WebRTC. Our experiments demonstrate that the newly added H.264 and VP9 codec's do not perform as expected in the presence of congestion or packet losses.

Mobile Performance: We evaluate the performance of WebRTC on mobile devices and demonstrate the impact of limited computational capacity on call quality.

Real Wireless Networks: We experimentally evaluate video calls on WebRTC in real networks, specifically focusing on wireless networks. Our experiments show that WebRTC can suffer from poor performance over wireless due to burst losses and packet retrains-missions. We identify key areas for improvement and brie y look at cross-layer approaches for improving video quality. Here Performance evaluation and design of congestion control algorithms for live video streaming have received consider-able attention. Below, we highlight the most relevant work.

Congestion control for multimedia: TCP variants such as Tahoe and Reno [16] have shown to lead to poor performance for multimedia applications since they rely only on losses for congestion indication. The approaches to address the shortcomings of these techniques can be divided in two categories.

The first variety of congestion control algorithms uses variants of delay to infer congestion. Delay based variants of TCP such as Vegas [5], and FAST [24] rely on measuring round trip delays but they are more reactive than proactive in congestion control. LEDBAT [22] relies on measuring one way packet delays to ensure high throughput while minimizing delays. Sprout [25] utilizes stochastic forecasts of cellular network performance to achieve the same goals. The second category of congestion control relies on Active Queue Management (AQM) techniques. NADA [27] uses Explicit Congestion Notifications (ECN) and loss rate to obtain an accurate estimate of losses for congestion control.

WebRTC congestion control: SCReAM [17] is a hybrid loss and delay based congestion control algorithm for conversational video over LTE. FBRA [19] proposes a FEC-based congestion control algorithm that probes for the available bandwidth through FEC packets. In the case of losses due to congestion, the redundant packets help in recovering the lost packets.

WebRTC performance evaluation: Several papers have studied the performance of WebRTC. Most related work focuses on a single aspect of the protocol or use outdated versions of WebRTC in their performance analyses. [2] Analyzes the Janus WebRTC gateway focusing on its performance and scalability only for audio conferencing in multi-party calls. [8] Focuses on comparison of end-to-end and AQM-based congestion control algorithms. [7] Evaluates the performance of WebRTC over IEEE 802.11 and proposes techniques for grouping packets together to avoid GCC's action on bursty losses.

[10] Presents the design of the most recent version of the GCC algorithm used in the WebRTC stack. While [10] pro-vides preliminary analysis of GCC in some synthetic network conditions, it does not focus on WebRTC's performance on mobile devices or real wired and wireless networks. Its main focus is on inter-protocol fairness between different RTP streams and RTP streams competing with TCP follows.

[23] provides an emulation based performance evaluation of WebRTC. However, all aws identi ed in [23] have been subsequently addressed in WebRTC. For instance, the data rate no longer drops at high latencies (but instead responds to latency variation), the bandwidth sharing between TCP and RTP is fairer due to the newly introduced dynamic threshold, and the available bandwidth is shared more equally when competing RTP ows are added.

A more realistic performance study using real network e ects is done in [13], where the performance of WebRTC is measured with mobile users in di erent areas. Even though the WebRTC implementation used is outdated, the paper suggests that WebRTC's over-reliance on packet loss signals leads to under-utilization of the channel due to mobility.

II. SYSTEM ARCHITECTURE

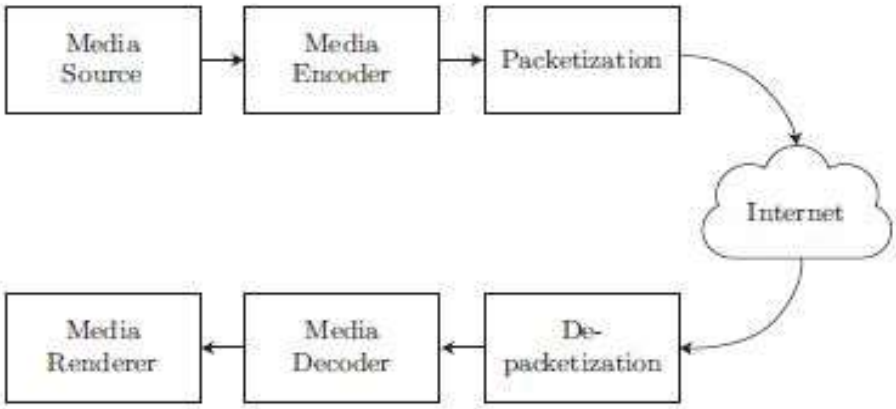

Fig: WebRTC's media processing pipeline.

III. IMPLEMENTATION

WebRTC uses the Google Congestion Control (GCC) algorithm [15], which dynamically adjusts the data rate of the video streams when congestion is detected. In this section, we provide a brief overview of GCC. More details can be found in [10]. WebRTC typically uses UDP (unless all UDP ports are blocked), over which it uses the Real-time Trans-port Protocol (RTP) to send media packets. It receives feed-back packets from the receiver in the form of RTP Control Protocol (RTCP) reports. GCC controls

congestion in two ways: delay-based control at the receiving end and loss-based control at the sender side.

Receiver-side controller

The receiver-side controller is delay-based and compares the timestamps of the received frames with the time instants of the frames' generation. The receiver-side controller consists of three different subsystems: (i) arrival time alter, (ii) over-use detector, and (iii) rate controller. These different subsystems of the receiver-side controller are shown on the right side of Figure 1. The arrival-time alter (Section estimates the changes in queuing delay to detect congestion. The over-use detector detects the congestion by comparing the estimated queuing delay changes from the arrival-time alter with an adaptive threshold. The rate controller makes the decisions to increase, decrease, or hold the estimated available rate at the receiver, A_r, based on the congestion estimated derived from the over-use detector. $A_r(i)$ for the i^{th} video frame is given as follows:

$$A_r(i) \begin{cases} \eta A_r(i-1) & \text{Increase} \\ \alpha R(i) & \text{Decrease} \\ A_r(i-1) & \text{Hold} \end{cases}$$

Where = 1:05, = 0:85, and R(i) is the measured received rate for the last 500 ms. The received rate can never exceed 1:5R(i):

$$A_r(i) = \min(A_r(i); 1{:}5R(i))$$

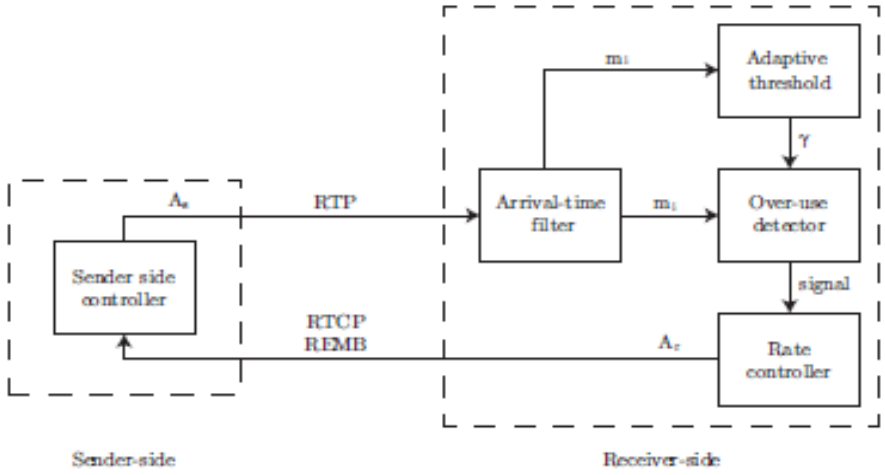

Fig: Diagram illustrating how sender and receiver determine and exchange their available rate.

Arrival-time filter

The arrival-time filter continuously measures the time instants at which packets are received. It uses the time of arrivals to calculate the inter-arrival time between two consecutive packets: $t_i\ t_{i\ 1}$, and the inter-departure time between the transmission of the same packets: $T_i\ T_{i\ 1}$. It then calculates the one-way delay variation d_i, defined as the difference between inter-arrival time and inter-departure time as follows:

$$d_i = (t_i \quad t_{i\ 1}) \ (T_i \quad T_{i\ 1})$$

This delay shows the relative increase or decrease with respect to the previous packet. The one-way delay variation is larger than 0 if the inter-arrival time is larger than the inter-departure time. The arrival-time alters estimates the one-way queuing delay variation m_i. The calculation of m_i is based on the measured d_i and previous state estimate $m_{i\ 1}$, whose weights are dynamically adjusted by a Kalman alter to reduce noise in estimation. For instance, the weight for the current measurement d_i is weighed more heavily than the previous estimate $m_{i\ 1}$ when the error variance is low. For more details, see [15].

Over-use detector

The estimated one-way queuing delay variation (m_i) is compared to a threshold. Over-use is detected, if the estimate is larger than this threshold. The over-use detector does not signal this to the rate controller, unless over-use is detected for a specified period of time. The over-use time is currently set to 100ms [10]. Under-use is detected when the estimate is smaller than the negative value of this threshold and works in a similar manner. A normal signal is triggered when m_i.

The value of the threshold has a large impact on the over-all performance of the GCC congestion algorithm. A static threshold can easily result in starvation in the presence of concurrent TCP flows, as shown in [11]. Therefore, a dynamic threshold was implemented as follows:

$$\gamma_i = \gamma_{i-1} + (t_i - t_{i-1}) * K_i * (|m_i \gamma_{i-1})$$

The value of the gain, K_i, depends on whether jm_ij is larger or smaller than $_{i\ 1}$:

$$K_i = \begin{cases} K_d & |m_i| < \gamma_{i-1} \\ K_u & \text{otherwise} \end{cases}$$

Where $K_d < K_u$. This causes the threshold to increase when the estimated m_i is not in the range of $[_{i\ 1}; _{i\ 1}]$ and decrease when it does fall in that range. This helps increasing the threshold when, e.g., a concurrent TCP follows enters the bottleneck and avoids starvation of the WebRTC streams. According to [11], this adaptive threshold results in 33% better data rates and 16% lower RTTs when there is competing traffic sharing the same bottleneck.

Rate controller

The rate controller decides whether to increase, decrease, or hold A_r at the receiver depending on the signal received from the over-use detector. Initially, the rate controller keeps increasing A_r until over-use is detected by the over-use detector. Fig further illustrates how the rate controller adjusts based on the signals received by the over-use detector.

A congestion/over-use signal always results in decreasing the rate, while under-use always results in keeping the rate unchanged. The state of the rate controller translates into available rate at the receiver, A_r, as shown in equation (1). A_r is sent back to the sender as an REMB (Receiver Estimated Maximum Bandwidth) [1] message in an RTCP report (Fig).

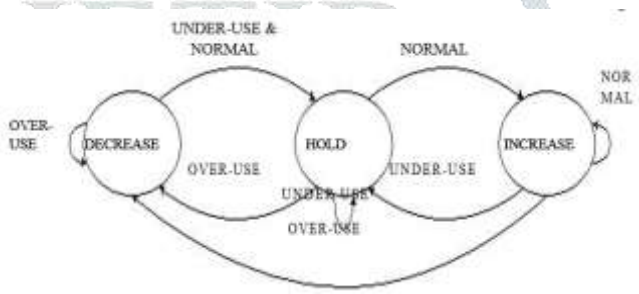

Fig: Rate controller state changes based on the signal output of the over-use detector.

Sender-side controller

The sender-side controller is loss-based and computes the sending rate at the sender, A_s in Kbps and is shown on the left side of Figure 1. A_s is computed every time (t_k) the k^{th} RTCP report or an REMB message is received from the receiver. The estimation of A_s is based on the fraction of lost packets $f_l(t_k)$ as follows:

$$A_s(t_k) = \begin{cases} A_s(t_{k-1})(1 - 0{:}5f_l(t_k)) & f_l(i) > 0{:}1 \\ 1{:}05A_s(T_{k-1}) & f_l(t_k) < 0{:}02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases}$$

If the packet loss is between 2% and 10%, the sending rate remains unchanged. If more than 10% of the packets are reported lost, the rate is multiplicatively decreased. If the packet loss is smaller than 2%, the sending rate is linearly increased. Furthermore, the sending rate can never exceed the last available rate at the receiver $A_r(t_k)$, which is obtained through REMB messages from the receiver as seen in Figure 1.

IV. EXPERIMENTAL SETUP

In this section, we describe the setup used for experimental evaluation throughout the paper. WebRTC handles all media processing as illustrated in Figure 3. Raw media from the audio and video source are first preprocessed and then encoded at a given target rate. These frames are then packetized and sent to the receiver over RTP/UDP. These frames are subsequently depacketized and decoded, which provides the raw video input that can be rendered at the receiver.

Our evaluation of WebRTC is divided into two parts. In the rest part, we emulate synthetic network conditions to study the performance of WebRTC in controlled settings. In the second part, we focus on experimental evaluation on real networks and particularly focus on wireless networks. The experimental evaluation setup for two users is shown in Figure 4.

For the rest part, we emulate different network characteristics using Dummynet [6], which allows us to add latency, packet loss, and limit the bandwidth for both uplink and downlink. To avoid additional latency and network limitations, we connect both WebRTC endpoints to the same local network via a wire.

In all of our experiments, we use devices with sufficient processing and memory capacity to ensure that the encoding and decoding of the video streams are not affected due to the devices themselves. To ensure this, we leverage WebRTC's RTC Stats Report API functionality which indicates if the video quality is limited due to memory or computation power at the devices. Unless mentioned otherwise, we use the most recent version of WebRTC (supported by Google Chrome version 52 and onwards) at all clients, with the default audio and video codec's OPUS and VP8, respectively. Instead of using a webcam feed and microphone audio signal, we exploit Google Chrome's fake-device functionality to feed the browser a looping video and audio track to obtain comparable results. For all our tests (unless mentioned otherwise), we use the following video with a resolution of 1920x1080 at 50 frames per second with constant bitrate: in to tree [2]. To obtain performance metrics, we use WebRTC's built-in RTCStatsReport [3], which contains detailed statistics about data being transferred between the peers.

V. WIRELESS PERFORMANCE

In this section, we evaluate the performance of WebRTC over real networks. I specifically focus on studying the impact of a WiFi hop on WebRTC.

Benchmarking

In Section 4, we observed that GCC is sensitive to changes in latency and packet losses. Transmitting over wireless net-works may result in bursty packet losses and dynamic latencies due to subsequent retransmissions, especially if the end-to-end Round Trip Time (RTT) of the WebRTC connection is large. In this section, we characterize the effects of wireless links on the performance of WebRTC by comparing against the performance on wired links.

I consider 3 types of WebRTC nodes: (i) a local wireless node, (ii) a local wired node, and (iii) remote wired nodes. We used a 2013 ASUS Nexus 7 tablet as a local wireless node connected to an IEEE 802.11 DD-WRT enabled Access Point (AP). The wired node is either a local machine located in our lab in New York City or a remote server running in Amazon EC2 cloud. We consider two cases for the remote server: one in the AWS Oregon availability zone and one in the AWS Sydney availability zone which provide different magnitudes of RTT. This allows us to study the impact of higher RTT as compared to the local machine.

Both the local and remote machines run Ubuntu 14.04 with Google Chrome 57.0 as the browser. We use the same injected video les for a fair comparison. Moreover, all the machines have sufficient computational power to eliminate the impact of devices on video performance. A virtual display buffer was used on the EC2 servers to run WebRTC on Chrome in headless mode. For the wireless node, we used 5GHz channels to minimize the interference from other IEEE 802.11 networks. To emulate the conditions of high loss environments, the AP transmission power was set to 1mW. We experiment with different channel conditions with the wireless node being in the same room as the AP (approximately 5 feet away), as well as outside of the room (approximately 25 feet away).

Table 7 shows average call statistics for two fully-wired calls with one wired node located in the NYC area in the lab and the other node in Oregon or Sydney. The NYC node was injecting a video encoded at 50FPS, and the remote nodes were using a video encoded at 60FPS. The average RTTs for the Oregon and Sydney calls were 77.74ms and 214.86ms, respectively. Accordingly, we term these scenarios as \medium" and \high" call latencies as compared to \short" latency scenario with both nodes in the NYC area. These results establish a baseline performance of WebRTC in realistic network conditions.

Next, we perform video calls with one wireless node and the other node either being a local wired node or one of the two remote nodes. A 720p video encoded in 50FPS was used across all 3 cases. On the wireless node, the cam-era on the Nexus tablet was used as video source, because video could not be injected into the Android distribution of Chrome without rooting the device.

VI. CONCLUSION

In this paper, we evaluated the performance of WebRTC-based video conferencing, with the main focus being on the Google Congestion Control (GCC) algorithm. Our evaluations in synthetic, yet typical, network scenarios show that WebRTC is sensitive to variations in RTT and packet losses. We also evaluated the impact of different video codec's, mo-bile devices, and topologies on WebRTC video calls. Further, our evaluations on real wired and wireless networks show that bursty packet losses and retransmissions over long RTTs can especially lead to poor video performance. The source code for setting up and evaluating the experimental environments described in this paper is available at: https://github.com/Wimnet/webrtc performance.

REFERENCES

[1] One-way transmission time. ITU-T, G.114 (May 2003).
[2] Amirante, A., Castaldi, T., Miniero, L., and Romano, S. P. Performance analysis of the janus webrtc gateway. In Proc. ACM AWeS'15 (2015).
[3] Ammar, D., De Moor, K., Xie, M., Fiedler, M., and Heegaard, P. Video QoE killer and performance statistics in WebRTC-based video communication. In Proc. IEEE ICCE'16 (2016).
[4] Bergkvist, A., Burnett, D. C., Jennings, C., Narayanan, A., and Aboba, B. Webrtc 1.0: Real-time communication between

browsers. online, 2016. http://www.w3.org/TR/webrtc/.
[5] Brakmo, L. S., and Peterson, L. L. TCP Vegas: End to end congestion avoidance on a global internet. IEEE J. Sel. Areas Commun. 13, 8 (1995), 1465{1480.
[6] Carbone, M., and Rizzo, L. Dummynet revisited. SIGCOMM Comput. Commun. Rev. 40, 2 (2010), 12{20.
[7] Carlucci, G., De Cicco, L., Holmer, S., and Mascolo, S. Making Google congestion control robust over Wi-Fi networks using packet grouping. In Proc. ACM ANRW'16 (2016).
[8] Carlucci, G., De Cicco, L., and Mascolo, S. Controlling queuing delays for real-time communication: the interplay of E2E and AQM algorithms. ACM SIGCOMM Computer Commun. Rev. 46, 3 (2016).
[9] Chen, W., Ma, L., and Shen, C.-C. Congestion-aware MAC layer adaptation to improve video telephony over Wi-Fi. ACM Trans. Multimedia Comput. Commun. Appl. 12, 5s (2016), 83:1{83:24.
[10] Cicco, L. D., Carlucci, G., Holmer, S., and Mascolo, S. Analysis and design of the google congestion control for web real-time communication (WebRTC). In Proc. ACM MMsys'16 (2016).
[11] Cicco, L. D., Carlucci, G., and Mascolo, S. Understanding the dynamic behaviour of the google congestion control for RTCWeb. In Proc. IEEE PV'13 (2013).
[12] De Cicco, L., Carlucci, G., and Mascolo, S. Experimental investigation of the google congestion control for real-time ows. In Proc. ACM SIGCOMM FhMN'13 (2013).
[13] Fund, F., Wang, C., Liu, Y., Korakis, T., Zink, M., and Panwar, S. S. Performance of DASH and WebRTC video services for mobile users. In Proc. PV'13 (2013).
[14] Hardie, T., Jennings, C., and Turner, S. Real-time communication in web-browsers. online, 2012. https://tools.ietf.org/wg/rtcweb/.
[15] Homer, S., Lundin, H., Carlucci, G., Cicco, L. D., and Mascolo, S. A Google congestion control algorithm for real-time communication. IETF draft, 2015. https: //tools.ietf.org/html/draft-ietf-rmcat-gcc-01.
[16] Jacobson, V. Congestion avoidance and control. In Proc. ACM SIGCOMM'88 (1988).
[17] Johansson, I. Self-clocked rate adaptation for conversational video in LTE. In Proc. ACM SIGCOMM CSWS'14 (2014).
[18] Mukherjee, D., Bankoski, J., Grange, A., Han, J., Koleszar, J., Wilkins, P., Xu, Y., and Bultje, R. The latest open-source video codec VP9 - an overview and preliminary results. In IEEE PCS'13 (2013).
[19] Nagy, M., Singh, V., Ott, J., and Eggert, L. Congestion control using FEC for conversational multimedia communication. In Proc. ACM MMSys'14 (2014).
[20] Nam, H., Kim, K.-H., and Schulzrinne, H. QoE matters more than QoS: Why people stop watching cat videos. In Proc. IEEE INFOCOM'16 (2016).
[21] Schulz-Zander, J., Mayer, C., Ciobotaru, B., Schmid, S., Feldmann, A., and Riggio, R. Programming the home and enterprise WiFi with OpenSDWN. In Proc. ACM SIGCOMM'15 (2015).
[22] Shalunov, S., Hazel, G., Iyengar, J., and Kuehlewind, M. Low extra delay background transport (LEDBAT). IETF RFC 6817, 2012.
[23] Singh, V., Lozano, A. A., and Ott, J. Performance analysis of receive-side real-time congestion control for WebRTC. In Proc. IEEE PV'13 (2013).
[24] Wei, D. X., Jin, C., Low, S. H., and Hegde, S. FAST TCP: motivation, architecture, algorithms, performance. IEEE/ACM Trans. Netw. 14, 6 (2006), 1246{1259.
[25] Winstein, K., Sivaraman, A., Balakrishnan, H., et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In Proc. USENIX NSDI'13 (2013).
[26] Yiakoumis, Y., Katti, S., Huang, T.-Y., McKeown, N., Yap, K.-K., and Johari, R. Putting home users in charge of their network. In Proc. ACM UbiComp'12 (2012).
[27] Zhu, X., and Pan, R. NADA: A uni_ed congestion control scheme for low-latency interactive video. In Proc. IEEE PV'13 (2013).