Delft University of Technology
Master of Science Thesis in Electrical Engineering

Rate-controlled Low Latency Service with OpenFlow

Renzo Daniel Arreaza Govea

Embedded Networked Systems

Delft University of Technology

Rate-controlled Low Latency Service with OpenFlow

Master of Science Thesis in Electrical Engineering

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Renzo Daniel Arreaza Govea
R.D.ArreazaGovea@student.tudelft.nl
renzoarreaza@gmail.com
TU Delft Student Number: 4567560

26 November 2019

Author
Renzo Daniel Arreaza Govea (R.D.ArreazaGovea@student.tudelft.nl)
(renzoarreaza@gmail.com)
(**TU Delft Student Number:** 4567560)

Title
Rate-controlled Low Latency Service with OpenFlow

MSc Presentation Date
Wednesday December 4, 2019

Abstract

The use of interactive media applications is becoming more common. With the current advancements in technologies, applications are becoming more advanced and research is being performed into adding more modalities to these applications to enrich the experience and increase their capabilities and uses.

The performance of these applications depends heavily on the network resources available to them. Through the use of congestion control algorithms, these applications are able to get a fair share of the available bandwidth. However, congestion control algorithms are unable to minimize the delay caused by competing traffic.

By combining the capabilities of OpenFlow with a priority queueing setup, we showed that we can provide low latency to a subset of flows. To control the utilization of the network resources and the effect the prioritization has on best-effort traffic, we use OpenFlow to control the bit rates of the high-priority flows. We show that by adjusting the ECN marking ratios of flows we are able to accurately control the bitrate of each of them.

Preface

This thesis marks the end of my studies at the Delft University of Technology for the Masters of Science degree in Electrical Engineering in the track of Telecommunications and Sensing Systems. This thesis was carried out at the Networks department of TNO, in The Hague. I would like to thank the department for having me and making it a great experience.

I would like to thank Niels van Adrichem, my supervisor at TNO, for all of his guidance and advice during my thesis. He provided valuable insight into various ideas I explored during the thesis. I would also like to express my gratitude to Fernando Kuipers, my thesis advisor, for bringing me in contact with Niels and for his guidance during this period. Additionally, I would like to thank various other members of the Networks department at TNO. Many of them helped me with brainstorming and gave me advise regarding the implementation. Finally, I would like to thank my family. They have supported me greatly during my studies and I couldn't have done it without them.

Renzo Daniel Arreaza Govea

The Hague, The Netherlands
26th November 2019

Contents

Chapter 1

Introduction

There have been great advancements in real-time communication over the last few decades. This went from manually switched public telephone networks to now being able to instantly establish a video call with people anywhere in the world thanks to the growth of the internet.

We have also seen an increase in the use of virtual reality (VR) and augmented reality (AR), together referred to as eXtended reality (XR), with the most common uses being playing video games and watching 360° video. XR can also be used by businesses to improve training sessions and allows for better collaboration between remote colleagues. Research is now being performed on using XR headsets to close the gap in communication quality between face-to-face and video communication [1].

The use of 360° video and spatial audio greatly increase the quality of experience compared to a simple video call. Furthermore, the addition of multiple sensory modalities such as haptics and olfaction is meant to further increase the feeling of immersiveness for the user [2][3].

1.1 Problem Description

Interactive media streams have stricter requirements compared to non-interactive traffic. The most important Quality of Service (QoS) requirements are throughput, delay, jitter and packet drop ratio [4]. The best-effort delivery with over-provisioning strategy used by internet service providers (ISPs) is typically able to provide sufficient bandwidth to media flows, and satisfies the requirements of most traffic. Furthermore, congestion control algorithms are implemented at the end points and aim to get the most resources without saturating the network. Obtaining sufficient bandwidth capacity is therefore not a problem anymore for real-time traffic. High latency can, however, still be a problem. Latency can only be minimized up to a point by congestion control algorithms. Decreasing the sending rate to minimize congestion may alleviate the problem temporarily, but will eventually result in other flows sharing the bottleneck to increase their sending rate and negate the effect of the decreased bandwidth of the first flow.

ISPs could theoretically prioritize specific flows or type of traffic to guarantee that they receive the best possible QoS. However, this could create very bad conditions for the non-prioritized flows. Using a hierarchical token bucket based approach should provide a lower and more consistent delay but doesn't minimize it since the traffic is being scheduled among the other classes.

Software Defined Networks (SDN) provide centralized and more fine-grained control over the traffic handling. Flows can be defined programmatically and the handling of each of these flows can be (re-)configured at any moment. Furthermore, statistics of each flow can also be gathered by the switch and requested by the SDN controller. In this thesis we propose a method to use SDN in combination with a class-based queueing configuration to provide minimal latency to priority flows, while maintaining control over the resources in use by each of these flows.

1.2 Research Question

The previous section discussed the need to have the network provide lower latency to specific flows. Furthermore, traditional rate limiting methods don't allow the latency to be minimized. The main objective of this thesis is to provide low latency to interactive multimedia flows while simultaneously controlling the resources used by these flows. This results in the following research question and sub-questions:

Research Question: How can the delay of specific flows be minimized while controlling bandwidth use?

- How can latency be minimized?
- How can the bandwidth used by flows be controlled?
- What is the effect of the proposed solution on real-time video traffic?
- What is the effect of the proposed solution on the best-effort traffic?

1.3 Thesis Structure

This thesis is structured as follows. Chapter 2 discusses previous work on improving media distribution, QoS for media flows and QoS with SDN. Chapter 3 starts by giving an overview of the WebRTC protocol stack and congestion control algorithms. In this chapter we also discuss the QoS requirements of the different modalities as well as the synchronization requirements. In chapter 4 we look at how QoS can be provided to traffic. We start by looking at different queueing disciplines followed by looking at SDN's QoS capabilities. We conclude this chapter by presenting our solution. In chapter 5 the measurement setup and further details about our solution are presented. We also describe how our solution will be assessed. The measurement results and analysis is given in chapter 6. The thesis is finalized in chapter 7 with the conclusions and future work.

Chapter 2

Related Work

The quality of media traffic can be improved in two ways. Firstly, this can be done by adapting the media flow based on the network fluctuations. This can for example be done by implementing congestion control algorithms at the end-hosts, and varying encoding bitrates. WebRTC is an open-source framework for real-time communication and has multiple congestion control algorithms which are still under development. An overview of these algorithms can be found in [5] and will be discussed in a later chapter. Secondly, special treatment for media flows could be offered by the network. This can be further split into two approaches. One approach is to use real-time network statistics to route the traffic through the best best-effort path. The second approach is to have the network cooperate in giving the traffic better performance [6].

The first approach, using a congestion control algorithm, is the most common since this can be implemented at the end-points, independent of network domains. Congestion control algorithms have certain limitations. Drop-based algorithms detect congestion after a queue in the path has overflown or an active queue management (AQM) mechanism detects congestion. Delay-based algorithms can suffer from inaccurate estimates due to delayed ACKs, cross traffic and queues. One way to improve this is to get explicit information from the network instead of only relying on end-point based measurements.

RFC 2481 [7] first proposed the addition of Explicit Congestion Notification (ECN) to the IP header. This allows network devices to notify the end-points of congestion before it increases further and packets have to be dropped.

Applications can also have their own in-network mechanisms for improving the detection of varying network conditions. MPEG-DASH can use its Server And Network-assisted DASH (SAND) protocol, in combination with a DASH-Aware Network Element (DANE) which provides real-time network bandwidth information to DASH clients.

There are also initiatives to improve WebRTC with the assistance of in-network elements. The TURN Revised and Modernized (TRAM) IETF working group aims to consolidate the various initiatives to update TURN and STUN. With the assistance of these servers it is possible to measure path characteristics of the available interfaces [8], perform UDP-based path MTU discovery

[9] and traceroute [10] and measure the bandwidth, latency and bufferbloat[1] on the path [12].

The other option we discussed is to have the network work to improve the performance of specific flows.
Boros et al. [13] use network orchestration in combination with SAND to control faulty clients that don't follow the instructions received via the SAND protocol. This ensures that other clients are not disturbed and simultaneously provide the badly behaving clients with a sufficiently good quality of experience (QoE).
Janczukowicz et al. [14] studies network solutions for improving WebRTC's quality based on queuing management on the uplink of wired access networks. Two approaches were compared to the default droptail[2] configuration. The first approach was bandwidth and queue length based configuration. In this approach a Hierarchical Token Bucket (HTB) was used in combination with stochastic fair queuing on the WebRTC class and Droptail or Adaptive Random Early Detection (ARED) on the best-effort class. The second approach, target-delay-based configuration, uses Fair Queuing-Controlled Delay (FQ-CoDel)[15] and Proportional Integral controller Enhanced (PIE) [16] to control the queuing delay.
In their measurement setup, using droptail caused disconnections in the WebRTC session. Using HTB resulted in a noticeable improvement of WebRTC but had a substantial effect on the best-effort queue if ARED wasn't used. FQ-Codel also had noticeable improvement, however not as high as HTB. PIE has a similar bitrate and frame rate to Droptail, but was able to provide good audio quality and didn't cause disconnections. While improvements are made in [14], the work is targeted at achieving acceptable call performance when faced with a small uplink capacity and competing TCP flows. Their results are therefore not valid for our problem.

In [17], Su et al. provide QoS guarantees by using Low Latency Queuing (LLQ), a combination of priority queuing (PQ) and Weighted Fair Queueing (WFQ), and active queue management. They compare three different queue management methods. These are Random Early Detection (RED), Weighted Random Early Detection (WRED) and using WRED in combination with Explicit Congestion Notification (ECN). They concluded that WRED with ECN performs the best in terms of delay and jitter.

Software-Defined Networking (SDN) [18] solves many of the limitations of traditional networking architectures and allows for easier provisioning and a more fine-grained control of the network. We will therefore look at research performed in this field and at how we could use SDN to achieve our goal. OpenFlow (OF) [19], the leading SDN controller-to-switch communication protocol, has limited QoS options. The main limitation when it comes to QoS is the inability to configure queues on the switches through OpenFlow. This has motivated researchers to extend the queue management and scheduling mechanisms in SDN.

To surpass the limitations of OpenFlow, Nam-Seok et al. [20] created OpenQFlow, a variant of the OpenFlow architecture. OpenQflow separates the OF flow

[1]Excessive delay caused by persistently full buffers [11].
[2]Droptail queues drop newly arriving packets when the buffer is full.

table into a flow state table, forwarding rule table and a QoS rule table. Hereby they separate flow classification from the tracking. Using this new architecture, they developed a QoS framework which provides performance guarantees and fairness. They however make no explicit mention of delay guarantees.

Ishimori et al. [21] proposed the QoSFlow module which extends the software switch specified in OF 1.0. Since many OF capable switches run on top of Linux, this module was created to enable the use of the packet schedulers available in Linux. QoSFlow adds support for HTB, Random Early Detection (RED) and Stochastic Fairness Queueing (SFQ) schedulers/AQMs. Caba & Soler [22] created an API that exposes the capabilities of OVSDB to the SDN controller and create a northbound API to allow applications to use these capabilities. Similarly, Palma et al. [23] created an OVSDB API called QueuePusher. QueuePusher was however exclusively created as an extension to the FloodLight controller.

The QoS framework developed by Kim et al. [24] is able to guarantee bandwidth and delay for a flow with a given QoS configuration. They implement a heuristic method that aims to maximize the probability of satisfying a new flow's QoS requirements while minimizing the number of rejected flows.

Wang et al. [25] provide inter-datacenter QoS guarantees on a per class basis. They classify traffic into 3 classes. In decreasing order of priority these are: interactive, elastic and background traffic. They use various guaranteed rate qdiscs in the form of modified Weighted Fair Queueing (WFQ) service disciplines. WFQ is able to provide each flow with a minimum guaranteed bandwidth independent of other flows sharing the path. In combination with traffic policing and admission control, a bounded delay can be provided.

In [26], Lin et al. use SDN to provide end-to-end QoS to multimedia services. With the proposed framework they are able to provide bandwidth guarantees to multicast flows. Due to the limited number of queues supported by most commercial SDN switches and the inability to configure them with the OpenFlow protocol they use meters to limit and reserve bandwidth for each flow. To minimize wasted bandwidth by reserving too much, they leverage the additive-increase/multiplicative-decrease (AIMD) behaviour of TCP's congestion control algorithms to predict the rate of the flow. They use additive-decrease/multiplicative-increase (ADMI) to adapt the bandwidth reservation. Furthermore, if congestion is detected on a link, low priority flows are rerouted and distributed over other available links/paths.

Chapter 3

Interactive Multimodal Application

We will start this chapter by giving an overview of WebRTC, a framework to deploy interactive media applications with. Thereafter an overview of the QoS requirements of interactive multimodal applications will be given. This will be done on a per modality basis and conclude with an inter-modality overview.

3.1 WebRTC

WebRTC is a free open-source project that provides real-time communication capabilities to web browsers and mobile applications via an API. This simplifies the creation of interactive audiovisual applications and eliminates the need for browser plugins such as Adobe Flash. WebRTC allows the establishment of a peer-to-peer media pipeline. Using such a connection for interactive multimodal applications allows the traffic to take a more direct route and eliminates latency introduced by an intermediary server. A signaling server is however required to establish the connection. We will be using WebRTC for our assessment since it's already being used by researchers for interactive multimodal applications [1, 27–31] and is used by Google Hangouts and Facebook messenger.

3.1.1 Protocol Stack

There are multiple factors that determine the suitability of a protocol for transporting interactive multimodal traffic. Some of these are: multiplexing capabilities, syncronization mechanisms, reliability, overhead, interoperability, adaptability and scalability [4]. Figure 3.1 shows the protocol stack used by WebRTC.

WebRTC supports the use of TCP or UDP as the transport protocol. TCP is a reliable, connection-oriented protocol which keeps track of packets using a sequence number and has controls for lost, erroneous and duplicate packets. These mechanisms can adversely impact the QoE of multi-media applications.

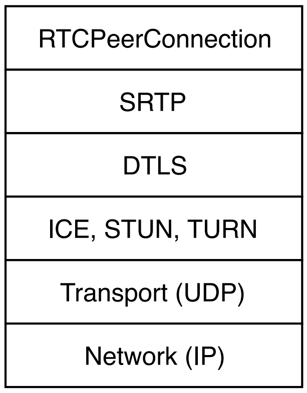


Figure 3.1: **WebRTC real-time media protocol stack**

Furthermore, TCP has a larger overhead, which is especially a problem when transporting data from a modality with a high update rate requirement such as haptics [4]. UDP is therefore the preferred choice for real-time data since it has a lower overhead and no retransmission mechanism which delays the delivery of the data to the application.

WebRTC uses an Interactive Connectivity Establishment (ICE) agent to establish the connections between peers. The ICE agent tries to establish a direct connection using the addresses of the hosts. This will fail if at least one of the hosts is behind a NAT. In that case, a Session Traversal Utilities for NAT (STUN) server is used to get the public addresses used by the hosts and establish the connection using these. If this fails, the last option is to use a Traversal Using Relay NAT (TURN) server. In this case the communication is relayed through the TURN server.

The WebRTC specification requires that the data sent is encrypted. Assuming UDP is being used as the transport protocol, the Datagram Transport Layer Security (DTLS) protocol is used to secure it. DTLS is derived from the Transport Layer Security (TLS) protocol and prevents eavesdropping, tampering an message forgery [32]. The keys establisched by DTLS are used by SRTP and SRTCP to encrypt the data [33].

Secure Real-time Transport Protocol (SRTP) [34] is used to transfer the media streams. The SRTP header has a series of fields that aid the receiver in processing the stream. The sequence number allows the receiver to detect out-of-order and dropped packets. The timestamp field represents the sampling instant of the first byte in the payload and is used to synchronize different media streams e.g., video and audio tracks. The synchronization source (SSRC) identifier is used to identify each individual stream. Up to 15 contributing source (CSRC) identifiers can be added to the header which contain the contributing sources for the payload.

SRTP doesn't use all of the header information directly. Instead, the Secure Real-time Transport Control Protocol (SRTCP) tracks transmitted and lost packets, jitter and last received sequence number. SRTCP sends this information back to the source via a feedback channel. The browser uses this information to adjust the sending rate as necessary via a congestion control algorithm.

3.1.2 Codecs

Video and audio codecs are used to encode the respective media streams before transmission to reduce the bandwidth requirements. The WebRTC standard requires WebRTC endpoints to implement the VP8 video codec and H.264 Constrained Baseline video codec [35]. The required audio codecs are (1) Opus; (2) G.711 (A- and μ-law); (3) comfort noise (CN), synthetic background noise to fill in silence in the transmission and (4) telephone-event, the transmission of in-band signaling in the form of dual-tone multi-frequency (DTMF) tones [36]. Other supported codecs include VP9 for video and ISAC, iLBC and G.722 for audio.

A list of supported codecs of a browser can be obtained by executing the command below in the browser console. *RTCRtpSender* can be changed to *RTCRtpReceiver* and *video* can be changed to *audio* to get the corresponding codecs lists. The lists are returned in decreasing order of codec preference.

```
> RTCRtpSender.getCapabilities("video").codecs
```

The exact working of the different codecs is beyond the scope of this thesis. There are however two characteristics of the codecs that are of interest, (1) frame types and (2) packet/frame loss resilience.

Current video codecs encode video into different types of frames to achieve a higher compression rate. The two main frame types are intraframes (I-frames in MPEG terminology) and interframes (P-frames in MPEG terminology). Intraframes are coded completely independent of other frames. Interframes are encoded based on previous intra- or interframes. This allows only the difference between the reference and current frame to be encoded, thereby reducing the amount of information.

VP8 has strictly speaking only the two previously mentioned frame types. These frametypes are however enhanced with the use of alternate prediction frames (golden frames) and alternative reference (atlref) frames. Besides only using the immediately previous frame, interframes may additionally use the most recent golden or altref frame as reference. Every intraframe is automatically a golden and altref frame and any interframe may optionally replace the most recent golden or altref frame.

VP9 additionally has hidden frames. These frames are decoded as normal, but are not shown. Instead they are saved to be used as a reference frame for inter prediction of future frames. It is possible to later send a short frame to tell the decoder to show the saved hidden frame.

In H.264, P-frames can be coded based on multiple previous frames (I- and P-frames). H.264 has one more frame type, the B-frame. This is similar to the P-frame, except it can use successive frames as reference and other frames cannot use B-frames as reference.

The frame drop tolerance of the different codecs varies. It's clear that the effect of a frame drop depends on what type of frame is dropped and how many other frames used it as reference. Furthermore, each codec can be configured to encode the data in different ways, which changes the frame drop tolerance. VP8 could for example be configured to code golden frames only with reference to the prior golden and intraframes. This subset can then still be decoded, regardless of the loss of interframes.

The G.711 ITU standard, Pulse code modulation of voice frequencies, was first released in 1988 with it primarily being used in telephony. It is a narrowband encoder focused on the sub 4 kHz frequencies, where most of the human voice harmonics are present. It therefore uses a sampling frequency of 8 kHz and uses 8 bits per sample. This results in a bitrate of 64 kbit/s. The standard defines two algorithms, namely the A-law and the μ-law algorithms. The different algorithms convert the samples into 8 bit codewords in slightly different ways. μ-law has a higher resolution at the higher range while A-law provides more quantization levels at the lower signal levels.

Opus is a speech and audio codec designed for interactive audio applications. It is able to use Linear Prediction (LP), the Modified Discrete Cosine Transform (MDCT) or both simultaneously to achieve good compression of both speech and music. It scales from narrowband mono speech at 6 kbit/s up to high quality stereo music at 510 kbit/s.

Opus has many control parameters that can be changed dynamically, some of which influence the frame and packet loss resiliency. Opus has a set of possible frame durations and can combine multiple frames into a packet of up to 120 ms. This lowers the bitrate but increases latency and increases the impact of a packet loss. Similar to other audio codecs, Opus exploits interframe correlations to reduce the bitrate. This comes at the cost of greater error propagation, where several packets have to be received before the decoder is able to properly reconstruct the speech signal. Opus employs forward error correction for packets that contain important speech information. This information is re-encoded at a lower bitrate and sent in the succeeding packet.

3.1.3 Application layer Congestion Control

As explained in section 3.1.1, UDP is the preferred transport layer protocol for WebRTC, despite the lack of a built-in congestion control algorithm. This allows the application to implement its own congestion control algorithm in the application layer. Requirements for congestion control algorithms for interactive real-time media are defined in [37]. The goal of the congestion control algorithm is to produce a sending rate as close as possible to the available bandwith over the entire path while maintaining the queue occupancy as low as possible [5]. Furthermore, the media flows should fairly share the network bandwidth with other concurrent flows.

One of the components of a congestion control algorithm is the congestion detection. This can be implicit, in the form of measurements performed by the end nodes, or explicit, by having network elements signal the end nodes when congestion is detected.

Implicit congestion detection can be divided into three categories, namely loss-based, delay-based and hybrid algorithms. Delay-based algorithms are generally preferred due to their ability to detect congestion before packets are dropped due to buffer overflows or active queue management schemes. Furthermore, drop-based algorithms suffer from significant delay variations, since they probe for available network by increasing their sending rate until packets start getting dropped. This means that the buffers of network elements are being filled up in the process.

The IETF RTP Media Congestion Avoidance Techniques (RMCAT) working group has proposed three congestion control algorithms for real-time communication at the application level: (1) Google Congestion control (GCC) [38], (2) Network Assisted Dynamic Adaptation (NADA) [39] and (3) Self-Clocked Rate Adaptation for Multimedia (SCReAM) [40]. An overview is presented at the end of this section in table 3.1

GCC [38] consists of a delay-based controller and a loss-based controller and can be implemented either as a hybrid or a sender side algorithm. In the hybrid version, the delay-based controller will be at the receiver side. In the sender side version the receiver simply records the arrival time and the sequence number of each packet and sends this information back to the sender to be processed. In the hybrid version, the delay controller part will calculate a maximum sending rate value and sent this to the sender via a Receiver Estimated Maximum Bitrate (REMB) message in a (S)RTCP packet.

For the delay-based controller, GCC uses the packet inter-departure and inter-arrival times to calculate the one-way delay variation and thus determine if the delay has increased or decreased. After passing this value through a Kalman filter, it's compared to a threshold to determine if the bottleneck is being over-used, under-used or normal. The value of the threshold is dynamic due to the considerable impact it has on the dynamics and performance of the algorithm [41]. Whether it was normal, under- or over-used determines in combination with the current state if the rate should be increased, decreased or held the same. This returns a new bandwidth estimate.

The loss-based controller determines if the rate should be increased, decreased, or held based on packet drop percentage thresholds. This rate is compared to the rate that was determined by the delay-based controller, and the minimum of the two is used as the actual transmission rate.

NADA [42] also uses delay and loss and additionally supports ECN. The congestion level is determined entirely at the receiver, and only a single value congestion indicator is sent to the sender. Using a single congestion indication value keeps the sender operations independent of the congestion indications (delay, loss or marking) being used by the receiver. When a packet is received, the one-way delay (d_{fwd}), packet loss ratio (p_{loss}) and packet marking ratio (p_{mark}) are estimated. The queueing delay is calculated, $d_{queue} = d_{fwd} - d_{base}$, where d_{base} is the minimum measured d_{fwd}. To avoid starvation due to increased delays when competing with loss-based algorithms, the calculated queueing delay is scaled with a non-linear function. This results in $\widetilde{d}$. $\widetilde{d}$ decreases as the d_{queue} increases past a threshold. This gives a higher weight to the loss and mark metrics when calculating the aggregate congestion indicator x_{curr}. The aggregate congestion indicator is calculated as follows,

$$x_{curr} = \widetilde{d} + D_{mark}(\frac{p_{mark}}{pmr_{ref}})^2 + D_{loss}(\frac{p_{loss}}{plr_{ref}})^2, \tag{3.1}$$

where D_{mark} and D_{loss} are the delay penalties for packet marking ($2ms$) and packet loss ($10ms$) respectively. pmr_{ref} and plr_{ref} are the reference packet marking and loss ratios respectively, both set at 0.01. The given values are the current recommendations, but may require adaptation for specific scenarios. Every $100ms$ a new report is generated and sent to the sender. The report contains (1) the rate adaptation mode; accelerated ramp-up or gradual rate update, (2) the aggregate congestion indicator and (3) the received rate.

From the received feedback report r_{ref} is calculated, which is the reference rate based on network congestion. The way in which r_{ref} is calculated depends on the rate adaptation mode provided in the feedback report. In the accelerated ramp-up mode, r_{ref} is calculated as follows,

$$r_{ref} = \max(r_{ref}, (1+\gamma)r_{recv}), \tag{3.2}$$

where r_{recv} is the receiving rate passed in the report. γ, the rate increase multiplier is calculated based on the RTT, self-inflicted queueing delay, filtering delay and target feedback interval; and has an upperbound of 0.5. The function is inversly proportional to RTT with the rationale that the longer it takes for the sender to observe self-inflicted queueing delay build-up, the more conservative it should increase its rate, hence the smaller the rate increased multiplier should be.

In the gradual update mode, the rate is changed only in proportion to itself and is affected by two terms. (1) The offset of the aggregate congestion indicator from its value at equilibrium and (2) the difference between the current and past value of the aggregate congestion indicator. r_{ref} is used to calculate the target video encoding rate and the sending rate of the output rate shaping buffer.

SCReAM [40] takes into account the distinct characteristics of a wireless (LTE) channel and the additional challenges that it brings when designing a congestion control algorithm for interactive traffic. The available bandwidth over a wireless cellular channel can vary considerably in a short time frame due to propagation effects such as shadowing and multipath fading. Additionally, base station handovers also cause the throughput to drop for short time intervals. These and other factors such as cell occupancy result in LTE connections having a wide range of available throughput. The rate adaptation solution for such an environment should therefore be quick and able to operate over a large range of channel capacity.

SCReAM is a sender side algorithm and consists of three parts: network congestion control, sender transmission control and media rate control. Information from the receiver is needed to determine the transmission rate at the sender. The receiver side therefore sends feedback reports containing a list of sequence numbers of received RTP packets, the timestamp corresponding to the received packet with the highest sequence number and the total number of CE marked packets.

The *network congestion control* determines the congestion window, which sets an upper bound on the number of in-flight packets. The congestion window is calculated based on the feedback from the receiver. The congestion window may be increased if the queueing delay is below a predefined threshold. If the delay surpasses the threshold, the window is decreased. The queueing delay is calculated by substracting the minimum measured delay from the current delay measurement. The threshold is typically set between 50 and 100ms.
Packet loss and ECN markings also lead to a reduction of the congestion window. For each of these events it holds that after the event is detected, further occurrences of the same event are ignored for a full (smoothed) round trip time. The intention is to limit the congestion windows decrease to at most one per round trip time. The congestion window reduction may be smaller when triggered by an ECN event compared to a packet loss event.

The *sender transmission control* controls the data transmission rate on a more granular level. The difference between the congestion window and the number of in-flight bytes determines how much data can be sent. The sender transmission control limits the transmission rate based on the estimated link throughput. This eliminates problems with ACK compression which may cause increased jitter and packet loss.

The *media rate control* adjusts the target bitrate of the encoder. It ramps up fast enough to get a fair share of resources when the available throughput increases and also has reduces the bitrate quick enough to avoid getting too much data queued. Reduction in target bitrate is triggered if the RTP sender queue size surpasses a threshold.

Table 3.1: **Overview of WebRTC congestion control algorithms**

	GCC	NADA	SCReAM
Metrics	one-way delay variation, loss ratio	one-way delay, loss ratio	one-way delay, loss ratio
Architecture	sender-side or hybrid	sender-side	sender-side
Actuation mechanism	rate-based	rate-based	window-based
Network support	none	ECN, PCN	ECN
Implementation status	Google Chrome	NS-2 and NS-3 simulators	OpenWebRTC and simulator

3.2 Requirements

3.2.1 Video

The bandwidth requirements depend on the video (resolution and framerate) and on the coding method being used. The bandwidth requirements will typically vary from 2.5 to 5 Mbit/s [4]. VP8 is limited to 2.5 Mbit/s. Packet loss tolerance is also dependent on the coding method. As was discussed in section 3.1.2, the packet drop resilience depends on the codec, the frame type contained in the lost packet and the number of packets dropped. According to [4], video packet drops should be constrained to approximately 1%. However, the google congestion control considers 2%-10% packet loss as acceptable. This discrepancy is most likely to avoid being starved of resources by flows using loss-based congestion control algorithms. Maximum delay should be kept between 100 ms and 400 ms, depending on the level of interaction. Jitter should be below 30 ms [4][43].

3.2.2 Audio

Similar to other modalities, the bandwidth depends on the coding method being used. G.711 produces an audio stream of 64 kbit/s. Opus can range between 6 kbit/s and 510kbit/s. Opus is able to encode full band speech at 28-40 kbit/s, and full band stereo music at 64-128 kbit/s when using a 20 ms frame size [44]. The one-way delay should be kept below 150 ms [45] and jitter below 30 ms. In G.711, each frame is coded independently. This means there is no build-in redundancy for FEC, but also no error propagation due to frames being encoded based on other frames. Opus on the other hand, does use the information in other frames to achieve a lower bitrate at equivalent quality. It also employs FEC for important frames. As was discussed in section 3.1.2, the tolerance varies based on multiple internal settings and on which packets are dropped.

3.2.3 Haptic

Haptic systems allow users to experience and exert touch, force and motion. While haptics isn't officially supported by WebRTC, there is research integrating it into WebRTC-based real-time applications. Furthermore, haptics has shown to greatly improve immersiveness in applications and is likely to be included in more applications as the technologies advance. The most common use cases are: (1) human operator controlling a remote actuator, referred to as Master-Slave Teleoperation (MST) and (2) Collaborative Virtual Environments (CVEs) in which multiple users can interact and collaborate with each other in a virtual environment. Excessive delay, jitter and packet drops can lead to instability in the control system.

The requirements are influenced by various factors. The degrees of freedom of the teleoperation system dictates how much data is generated. The specific use case or task being performed influences how noticeable network effects are to the user. Finally, there has been a lot of research into methods to increase the robustness against network effects. These methods include motion and force prediction to compensate for network delays and haptic data compression to reduce the required bandwidth.

The delay requirements range between 1 and 50 ms, with a maximum jitter of 2 ms. The bandwidth requirement will be around 512 kbit/s and packet loss tolerance is between 0.01% and 10% [4].

3.2.4 3D Graphics/ 3D Video

3D video streaming is similar to normal video streaming in that users gradually download the content and are able to immediately render it. There are however some key differences. (1) The type of data being sent. It consists of 3D mesh models, texture and animation. (2) In 2D video streaming, the video encoding is predictable and the access pattern is linear. In 3D streaming, the data to be encoded and transmitted depends on the user's actions. This results in unique transmission sequences and are therefore hard to anticipate.

The exact requirements will depend on the use case. Use cases with more inter-user interaction will have higher requirements compared to CVEs where users are mostly interacting with the environment [46]. The bandwidth requirement of a 3D video stream will typically vary between 2 Mbit/s and 5 Mbit/s [47]. Typical recommended values for latency and jitter are 100 ms and 50 ms respectively [48]. However, Park et al. [46] found no significant difference between 200 ms latency with no jitter and 10 ms delay with jitter. Up to 10% packet drop is acceptable [4].

3.2.5 Inter-modality synchronization

The synchronization of the streams also affects the experience of the user. ITU recommendation BT.1359 gives relative timing recommendations for audiovisual media. The detectability thresholds were found to be +45 ms and -125 ms, where the positive value indicates that the audio precedes the visual medium. The inter-modality synchronization requirements vary mostly depending on the specific modalities and how these are perceived by humans. Furthermore, specific use cases can tighten the requirements. Some tasks in CVEs may for example require more precise hand-eye coordination than others. Synchronization techniques such as adaptive synchronization [49] do exist to aid in this.

Ultimately, the requirements for each modality are highly dependent on the coding scheme in use, the exact use-case and even the users.

Chapter 4

Providing Quality of Service

In this chapter, we will look at different aspects of providing Quality of Service. First, we will look at different queuing disciplines (qdiscs) followed by looking at what the QoS capabilities are of OpenFlow-based software-defined networks and various software switches. Lastly, we will propose our solution to provide low latency to high priority traffic.

4.1 Overview of queuing disciplines

Queuing disciplines (qdiscs) control how incoming packets are arranged for output. Qdiscs can be divided into two categories, namely classful and classless. Classful qdiscs can contain classes containing other qdiscs and provide a handle to attach filters for classification into these child classes. Classless qdiscs cannot contain classes nor is it possible to use filters. The qdisc may still consist of multiple queues but traffic can't be sent to a specific queue via a filter or other external mechanisms [50].

4.1.1 Classfull queuing disciplines

Hierarchical Token Bucket
Hierarchical token bucket (HTB) [51] allows for bandwidth reservation on a per class basis. The bandwidth is guaranteed by using a token bucket filter (TBF) on each class. Tokens are generated at the desired rate and are accumulated in a bucket. A packet can only be dequeued when there is a token available. The rate at which the tokens are generated sets the maximum sustainable rate at which packets can be dequeued. The bucket allows tokens to accumulate up to a maximum number, which allow bursts of traffic to be dequeued at a higher rate than the token generation rate.

Besides the TBF functionality, HTB allows the creation of a hierarchy of classes. Each class can be configured as described above. Additionally, child classes can borrow tokens from their parent class if required and available. The absolute maximum rate per class including borrowing can be configured (ceiling rate, ceil) as well as a priority value which is used when selecting from which class to dequeue.

Priority queue
The priority queueing discipline consists of three first in, first out (FIFO) queues, each of which with a priority assigned to it. When the server is ready to send a packet, it will check the queue with the highest priority first. If there is a packet to be sent it will do so, if not it will check the next highest priority queue.

Hierarchical Fair Service Curve
The basis of the Hierarchical Fair Service Curve (H-FSC) [52] queuing discipline is the concept of a service curve. The service curve defines a QoS model taking into account both the bandwidth and latency requirements. With a linear service curve these two requirements are coupled. Unlike other queuing disciplines, H-FSC is able to guarantee non-linear service curves. With non-linear service curves, both delay and bandwidth allocation can be defined simultaneously and independently. While in theory any non-decreasing service curve can be used, in practice only piece-wise linear curves are used. Furthermore, the available implementation only supports a two-piece linear curve, which is enough to specify the bandwidth and latency requirements.

A two-piece linear curve can be defined using three parameters. These are the slopes of the two linear subsections, named *m1* and *m2* respectively, and the x-axis value of the intersection of the two lines, named *d*.

Three different service curves can be defined per class. These are the Real Time (rt), Link Share (ls) and Upper Limit (ul) curves. The real time curve gives its class hard guarantees on the maximum delay until a packet is sent. The link share service aims to satisfy the service curves of interior classes and fairly distribute excess bandwidth. This is also what is used by default to schedule the packets. The real-time service is only used when there is a potential danger that the service guarantees for leaf classes are violated.

4.1.2 Classless queuing disciplines

Flow Queue Controlled Delay
Flow Queue Controlled Delay (FQ-CoDel) [15] is a combination of per flow queuing, the Controlled Delay (CoDel) [53] active queue management (AQM) scheme and a modified version of Deficit Round Robin ++ (DRR++) [54]. The per flow queuing and modified DRR++ ensure fairness between the flows. CoDel is used to control bufferbloat-generated excess delay. CoDel uses sojourn time (time spend by a packet in the queue + transmission time) as the indicator for congestion. Sojourn directly measures the delay experienced by a packet independent from the link rate and scheduling of other queues on the same interface. CoDel tracks the minimum sojourn time over a time interval to determine if there is a standing queue. When bufferbloat is detected, packets are dropped from the head of the queue. This results in the packet drops being detected sooner by the endpoint and allows for a faster reaction.

4.2 Quality of Service in Software-Defined Networks

Network elements consist of two planes; the control plane, which contains the configuration and determines the forwarding rules, and the data plane, which forwards the packets. In traditional networks, each network element contains both planes, and independently determines how and where to forward traffic to. This has some advantages and disadvantages. With respect to QoS, the distributed architecture is a disadvantage as it complicates the design and implementation of these systems.

Software-defined networks separate the control and data planes. The control plane is moved to a centralized system, referred to as the SDN controller. The programmability of the controller allows for easy implementation of custom flow control configurations. The fact that it is centralized facilitates gathering statistics from the entire network. The global view this provides facilitates the implementation of potentially better routing algorithms as well as QoS architectures.

OpenFlow is the most widely used protocol for communication between the control and data planes. In the next section we will look at its QoS capabilities.

4.2.1 OpenFlow

Over the different OpenFlow versions, a few features have been added that can be used to implement QoS frameworks.
Since OpenFlow 1.0 there is an optional action called *enqueue*, which allows flow entries to forward packets through a specific queue of a port. OpenFlow can query for information about the queues but is unable to configure these. OpenFlow 1.2 added support for querying all queues in a switch simultaneously and defined additional queue properties. OpenFlow 1.3 added support for rate limiting and rate-based packet remarking on a per-flow basis. This is done with meter tables consisting of meter entries. Each meter entry can define multiple meter bands. Each meter band consists of a "Band Type", either *drop* or *dscp remark*, "rate" (rate and burst), "counters" and optionally, "type specific arguments". OpenFlow 1.5 replaced the meter instruction with a meter action. This allows for multiple meters to be attached to a flow entry and for meters to be used in group buckets. Egress tables were also added in this version and can be quite useful for QoS.

The *DSCP remark* band was made to be used for a Differentiated services (DiffServ) policer that increases the drop precedence of the DSCP field of packets that exceed the band rate value. The field formerly known as the Type of Service (ToS) field and repurposed by RFC2474 [55] can be seen in table 4.1. The Assured Forwarding DiffServ specification [56] specifies the following definitions for the DSCP field. Bits DS5, DS4 and DS3 define the class, bits DS2 and DS1 specify the drop probability and bit DS0 is always zero. Furthermore, [56] defines low, medium and high drop precedence which can be seen in table 4.2.
This means that the *dscp remark* meter can only rewrite an incoming packet with low drop precedence to medium or high and an incoming packet with me-

dium drop precedence to high drop precedence.

Table 4.1: **Type of Service Field**

Type of Service							
DSCP						ECN	
DS5	DS4	DS3	DS2	DS1	DS0	ECN	ECN

Table 4.2: **Drop Precedence**

Drop Precedence	**DS2-DS0**
Low	010
Medium	100
High	110

4.2.2 Software Switches

Open vSwitch
Open vSwitch (OvS) is a multilayer virtual switch supported by various hypervisors and cloud computing platforms. It has full support for OpenFlow 1.1 and 1.2. Later versions are also supported but are still missing features to be compliant with the specifications. Meters were implemented in the userspace datapath in OvS 2.7 and in the kernel datapath in version 2.10. However, only the meter band of type *drop* is supported [57]. The OvS configuration is stored in the Open vSwitch Database (OVSDB) which is configurable via remote procedure calls (RPCs) [58]. This facilitates the configuration of ports and queues from a remote device.

BOFUSS
BOFUSS [59], also known as the CPqD switch or ofsoftswitch13, is a userspace software switch implementation compatible with OpenFlow 1.3. It was build upon the Stanford OpenFlow 1.0 reference switch and Ericsson's Traffic Lab OpenFlow 1.1 switch. Unlike OvS, this switch is purely meant for experimental purposes. It supports meters, and both band types defined in the OpenFlow 1.3 specification.

Lagopus
Lagopus [60] is another OpenFlow 1.3 software switch. It can be run in raw socket mode or DPDK mode. The later allows the switch to run on multiple cores and use other techniques to provide high performance packet processing. Queues and meters can be configured through the Lagosh shell.

4.2.3 SDN Controllers

Most SDN controllers support all the (QoS) features specified in the OpenFlow 1.3 switch specification. This is because unlike with the switches, implementing these features in a controller entails simply implementing the corresponding

OpenFlow messages. A few controllers implement additional functionality to provide more advanced QoS features. In this section we will look at these extra QoS features.

The OpenDayLight (ODL) controller has an OVSDB southbound plugin, which allows queues to be configured on OVSDB capable switches from the controller application. It also supports OF-CONFIG, an accompanying standard to the OpenFlow switch standard that enables configuration of OpenFlow (logical) switches from the controller.

The Open Network Operating System (ONOS) controller has no official support for OVSDB, OF-CONFIG or other datapath configuration protocol. There is a third-party plugin which implements a minimal OVSDB interface[1]. In [61] they created support for queue configuration via an SDN application which used ovs-vsctl, the OvS virtual switch control utility, to configure the switches.

The Floodlight controller doesn't support OVSDB or OF-CONFIG by default. Wallner et al. [62] implemented a QoS module[2] that sets up matching, classification and policy handling for QoS. Palma et al. [23] created the queuepusher extension which utilizes OVSDB in combination with the Floodlight northbound API to generate OVSDB messages that can be triggered via the northbound API.

The Ryu controller has limited NETCONF and OF-CONFIG support. Furthermore, it has a OVSDB Manager library and a OVSDB library. The OVSDB Manager library spawns a server which the OVSDB devices can connect to. The server can then configure the OVSDB devices. The IP address and port number of the server have to be configured on the devices. The OVSDB library differs from the Manager library in that it is able to initiate connections from the controller side to the OVSDB devices.

4.3 Proposed Solution

There are a few things we want to achieve with our solution. (1) Low latency for specific flows. (2) Minimize packet drop. (3) Minimize packet reordering. 4) Having the ability to control the bandwidth usage of the high priority flows.

As was discussed in section 1.1, only self-inflicted latency can be minimized by the congestion control algorithms in use by the flow. Other flows can still cause congestion and increase the latency of the delay-sensitive flows. We will therefore use queueing disciplines to properly minimize the delay of these flows. The simplest way is to use the priority queue (see section 4.1.1). Using a priority queue and sending the WebRTC traffic through the highest priority class will guarantee it has the smallest possible delay and jitter. This however also makes the full interface bandwidth available to the flows in the highest priority queue. The problem with this is that sending too much traffic to this queue will

[1]`https://github.com/netgroup-polito/onos-applications/tree/master/ovsdb-rest`
[2]`https://github.com/wallnerryan/floodlight-qos-beta`

degrade its performance and can starve the lower priority queues of resources.

Using a queue-based setup is the best way to properly limit the bandwidth used by flows. This is however not straightforward in our case due to the low latency requirement. Hierarchical token bucket based setups would not be suitable since queues can build up. This will also act as a lower rate interface, therefore even if the congestion control algorithm manages to minimize queue buildup, the delay will be larger compared to having the full transmission rate available. HTB classes have a *prio* parameter which dictates which classes are offered the excess bandwidth first. This also dictates which class has a lower delay [51].

The H-FSC queueing discipline is another way to limit the bandwidth and provide the desired maximum latency. To receive the configured latency, the flow's arrival rate should be below the configured rate. Small bursts above the configured rate will cause queueing and increased delay. Furthermore, even if we assume a media stream has an arrival rate below the configured rate, a group of media flows will not necessarily hold this property. Configuring a H-FSC class per media flow is not scalable since switches usually have a small maximum number of queues that can be configured per interface. Additionally, dynamically configuring a switch dataplane is not very feasible. A few protocols exist (OF-Config, OVSDB) but these have limited options when it comes to configuring H-FSC. H-FSC is also a very complex queueing discipline, which many switches don't support.

As we can see, limiting the bandwidth of flows while simultaneously providing low latency is not trivial from a network point of view. What ultimately decides the rate of a flow is the sending host. The rate is usually determined by a congestion control algorithm. To control the bandwidth usage of the delay sensitive flows, we will use the network to manipulate the congestion control algorithm at the endpoints.

As was discussed in section 3.1.3, congestion control algorithms use a subset of packet drops, delay (variation) and ECN marks as congestion indicators. Furthermore, the default congestion control algorithm used by WebRTC, Google congestion control, uses REMB messages to indicate a maximum transmission rate when used in hybrid mode.

The ideal solution would be to create these REMB messages from the network. This would allow us to precisely specify the maximum sending rate, and it would reach the sending host faster than the REMB messages sent by the receiver. This is unfortunately not possible, since these messages are encrypted by the endpoints. Allowing these messages to be sent by any host would also expose the endpoints to DoS attacks, since a few messages per second could set the maximum bandwidth to an unusable value.

We will therefore manipulate the standard congestion indication metrics. The preferred metric to use is ECN, since manipulating this will not impair any in-flight packets. We will implement and test four different ECN marking strategies, as will be explained in section 5.1.5. Delaying (a subset of the) packets is another way to signal to the congestion control algorithm to lower its sending rate. Delaying a non-continuous set of packets could cause packet reordering, and if the delay is larger than the current receiving jitter buffer size, the packet will be classified as lost and have the associated effect. We therefore plan to delay all packets of the flow for a short duration of time. As a result, packet reordering will be limited to the transition moment, jitter will be smaller compared to intermittent delaying and the jitter buffer will adjust accordingly resulting in less packets being classified as lost. The last resort is to drop packets for flows using a purely loss-based algorithm. UDP-based media flows will typically use a delay-based or hybrid algorithm, so this shouldn't be necessary. As opposed to the delay manipulation, the preferred method is to drop a non-continuous set of packets, since this increases the chances of the decoder still being able to successfully decode the media flow.

We need a system that can gather traffic statistics on a per-flow basis and is able to manipulate the flows. The most flexible method to manipulate flows is OpenFlow. The built-in OpenFlow statistics provide sufficient information for our needs and thus we decided to use this instead of dedicated monitoring protocols such as sFlow. Besides the controller we will also implement an application that requests statistics and will determine if any actions need to be taken for each priority flow.

The base setup will be as follows. We will have statically configured queueing setup with the basis being a priority queue. The WebRTC flows will be sent through the highest priority (lowest ID) class. We will configure a standard FIFO queue in this class. The best-effort traffic will be sent out through the second highest priority class. We will configure FQ-CoDel on this queue to prevent the latency from increasing too much. FQ-CoDel should work well for this since its queue measurement method, sojourn time, is independent of the available bandwidth. The lowest priority class will remain unused. Figure 4.1 shows a simple diagram of this solution.

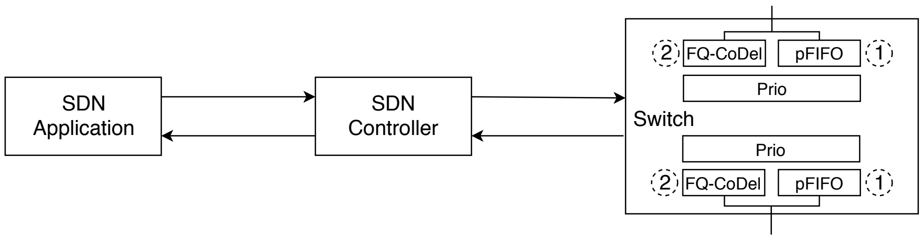


Figure 4.1: **Proposed solution showing the three main components and switch queues with the priorities in the dashed circles**

Chapter 5

Measurement Setup and Implementation

This chapter starts by describing the measurement setup, the SDN controller design and the SDN application. This is followed by a performance evaluation of the software switch. Lastly, we describe the measurement we will perform to assess our solution.

5.1 Experimental Setup

5.1.1 Network setup

The metrics we are interested in measuring are throughput, latency, jitter, packet drop and packet reordering. The most challenging of which is latency. To be able to accurately measure this, we implemented our test network as seen in figure 5.1. We run both clients of the WebRTC application on the same node (Node-1) and separated them using network namespaces. Having both clients on the same node allows us to directly compare the timestamps of the captured packets going through the two interfaces using the exact same clock. Using network namespaces allows us to force the traffic to flow through the other nodes.

5.1.2 SDN Controller Platform and Software Switch

For the SDN part of the setup we had a few requirements. We needed support for meter band of type *DSCP remark* and the *set-queue* action, which are both optional in the OpenFlow specification. From the switch we also needed support for hierarchical queue configuration.

For the SDN controller we chose to use Ryu. It supports all the needed features, is more suitable to customize and it's the one we are most familiar with.

We looked at three software switches. These were Open vSwitch, BOFUSS and Lagopus. Open vSwitch is the most well known and perhaps the best

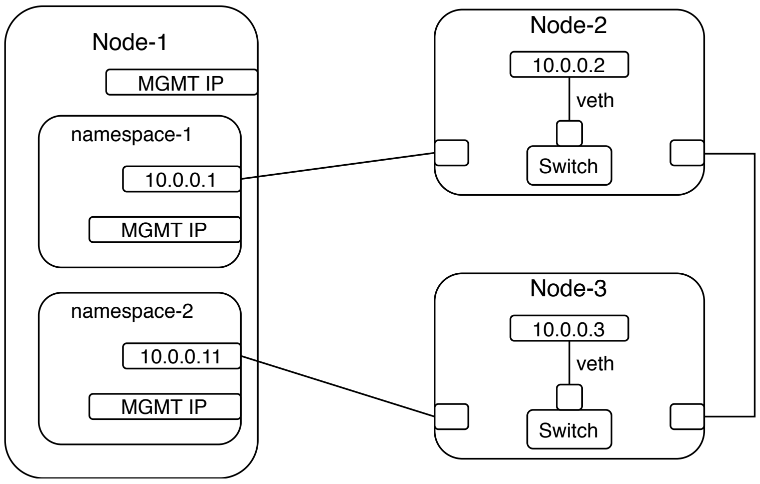


Figure 5.1: **Measurement topology in OpenStack cloud**

performing one. Meters are supported according to the documentation. What they fail to mention is that it only supports the drop band. Sending the add-meter command with a DSCP remark band returns a "ofpmmfc_bad_band" error message which indicates that the band is not supported. In the meter datapath source code we also found a comment[1] stating that currently only the drop band type is supported.

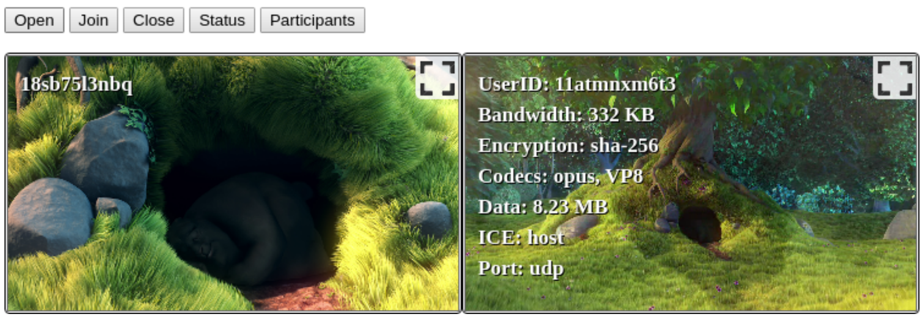
We had a few problems with the Lagopus switch. Firstly, we were unable to configure multiple queues per port. Secondly, after installing meters and sending flows through them, the flows weren't being metered. We also found that flows were being removed after the *idle_time* passed, even if packets were matching the flow entry and we could see the packet count statistic increase.

This left us with the BOFUSS switch. It supports meters with DSCP remark bands as well as sending traffic to specific queues. Queues are configured via their datapath control utility. It supports multiple queues per port, but the only configuration parameter is bandwidth. Looking at the queue configuration with tc[2], this is mapped to a HTB qdisc containing a default class and an additional class for each configured queue. The tc class-id of these queues is the same as what was passed in the configuration command. We found that we could replace this queue configuration with our own using tc, and the switch would still successfully sent packets to the classes with the class-ids that were used when configuring it via the datapath control utility. This switch thus has all the needed features, and is therefore the one we used for our measurements. We did find a small bug when using the set-queue action. Besides sending the packet out of the configured port and queue, it would also receive a duplicate of this packet through that same port. We circumvented this issue by installing extra table entries to drop these packets.

[1]`https://github.com/openvswitch/ovs/blob/v2.11.0/datapath/meter.c#L176`
[2]Linux's traffic control utility

5.1.3 WebRTC setup

For the signaling server required by WebRTC to establish the connection between clients, we used the RTCMultiConnection[3] socket.io server. Furthermore, we created a simple website that allows clients to create/join a video-call room (figure 5.2). This webpage also sets a few parameters for the WebRTC session. We use VP8 for the video encoding and opus for audio encoding and disable the use of STUN and TURN servers.



18sb75l3nbq

UserID: 11atmnxm6t3
Bandwidth: 332 KB
Encryption: sha-256
Codecs: opus, VP8
Data: 8.23 MB
ICE: host
Port: udp

Figure 5.2: **Screen capture of an in-progress video stream.**

We use the getStats[4] library which uses the w3 standardized getStats() function to get the WebRTC call statistics. We query the statistics every second during the entire duration of the call. This returns JSON formatted data and gives us insight into how our solution affects the media stream.

To automate the call establishment through the webpage we used Selenium. Since the web clients are run on a server, we ran chrome in headless mode and passed a video file to Chrome to use as a fake video capture. We used an uncompressed 720p version of the Big Buck Bunny video which was long enough to last the duration of each measurement. To facilitate the identification of the WebRTC traffic in the network, we configured an iptables mangle rule to set the IP DSCP bits to 0x02 (listing 5.1).

Listing 5.1: **Rewriting DSCP bits of WebRTC packets**

```
$ sudo iptables -t mangle -A OUTPUT -s $LOCAL_IP -d $REMOTE_IP \
> -p udp -m multiport --dports 32768:60999 -j DSCP --set-dscp 2
```

5.1.4 Queueing setup

As we will explain in section 5.2, we have to slow down the transmission rate of various interfaces sending traffic towards the switch. We also want to create a bottleneck at a specific switch. This will be achieved by configuring the switch interfaces to be slightly slower than the rest of the network. We achieve this by configuring a HTB on top of the queueing setup we will be using for the experiment. Queue configuration commands can be seen in listing 5.2.

[3]https://github.com/muaz-khan/RTCMultiConnection-Server
[4]https://github.com/muaz-khan/getStats

Listing 5.2: **Switch queueing configuration**

```
$ sudo tc qdisc replace dev $DEV root handle 100: htb default 1
$ sudo tc class add dev $DEV parent 100: classid 100:1 htb \
> rate 18mbit

$ sudo tc qdisc add dev $DEV parent 100:1 handle 1: prio
$ sudo tc qdisc add dev $DEV parent 1:1 pfifo limit 1000
$ sudo tc qdisc add dev $DEV parent 1:2 fq_codel quantum 300 \
> limit 800 target 2ms interval 50ms noecn
```

The *handle* and *classid* numbers are qdisc and class identifiers of the form *major:minor*. All classes sharing a parent must share the *major* number, and have a unique *minor* number. The *minor* number of a qdisc must always be zero and can therefore be omitted during declaration.

The first two lines install a HTB qdisc with a single class which is rate limited to 18 Mbit/s. Next we configure a priority qdisc in this (HTB) class, which automatically creates three child classes of its own with the same *major* handle number and *minor* numbers 1 to 3. On the highest priority class we add a pfifo qdisc. On the next priority class we configured FQ-CoDel. While FQ-CoDel was designed to be a "no knobs" qdisc, running it at such low speeds requires the preset parameters to be adjusted to get it working as designed [63].

Figure 5.3 combines the previous sections into one diagram, showing the measurement setup in more detail.

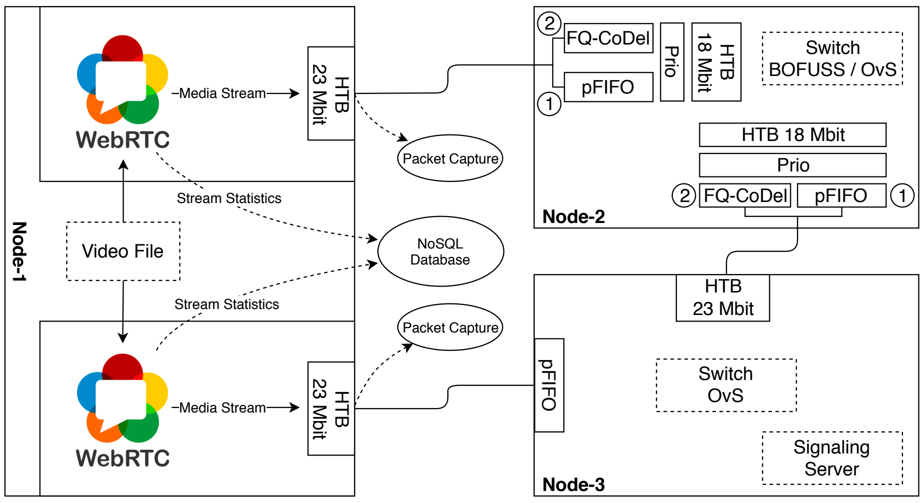


Figure 5.3: **Detailed measurement setup**

5.1.5 SDN Controller

Figure 5.4 gives an overview of the base OpenFlow table entries. Best-effort flows only use table 0 and match with preconfigured table entries with priority 2. The packets are matched based on the destination IP address and are sent out through queue 2. The highest priority entries (priority=4) are used to match individual high-priority flows. We set the output port at this stage which allows us to consolidate entries in table 1. Each of high-prio table 0 entries is configured with its own meter entry and is sent to table 1 for further processing. All prio-flows receiving the standard low latency treatment can be handled by a single table entry. We simply match all IPv4 packets and set the queue to 1. Prio-flows that need special treatment can be handled by configuring entries with a higher priority than the default entry.

As explained in section 4.3, we will signal (fictitious) congestion to the endpoints using ECN, delay and packet drops.

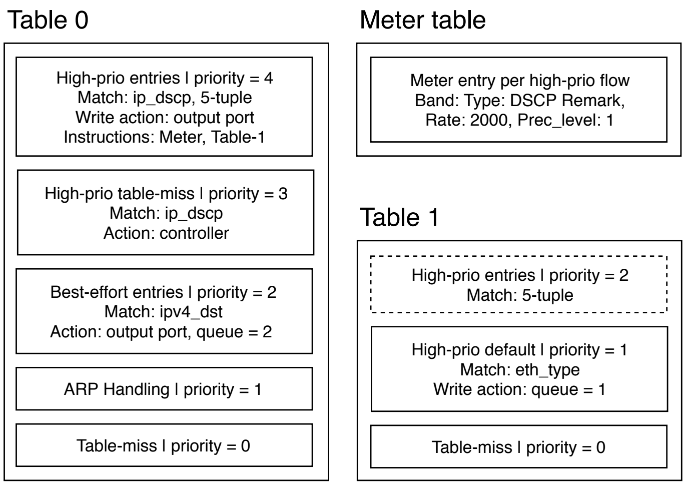


Table 0
High-prio entries | priority = 4
Match: ip_dscp, 5-tuple
Write action: output port
Instructions: Meter, Table-1
High-prio table-miss | priority = 3
Match: ip_dscp
Action: controller
Best-effort entries | priority = 2
Match: ipv4_dst
Action: output port, queue = 2
ARP Handling | priority = 1
Table-miss | priority = 0
Meter table
Meter entry per high-prio flow
Band: Type: DSCP Remark,
Rate: 2000, Prec_level: 1
Table 1
High-prio entries | priority = 2
Match: 5-tuple
High-prio default | priority = 1
Match: eth_type
Write action: queue = 1
Table-miss | priority = 0

Figure 5.4: **OF switch entries structure**

Increased latency is interpreted as buffers/queues filling up. Congestion control algorithms will not react on a single packet having a larger delay, since this could simply have been caused by small bursts of competing traffic and doesn't signify true congesiton. We therefore have to delay multiple packets.

Delaying a non-continuous set of packets can lead to a few undesired effects. Firstly, it can lead to out-of-order delivery when the subsequent packet isn't delayed. The second effect is due to the jitter buffers implemented at the endpoints. Having a low latency flow will lead the jitter buffer size to be small. This is likely to cause intermittently delayed packets to be categorized as lost and thereby having a larger impact on the media stream than intended. We will therefore delay a subsequent set of packets. This will restrict the potential for packet reordering to the transitioning moment from delaying packets back to low latency treatment. This will also give the jitter buffer the opportunity to adapt and prevent packets from being catergorized as lost.

We will delay packets by sending them through the best-effort queue, class 2 of the priority qdisc, instead of the highest priority class. Class 2 will always have a higher latency than class 1 since the queue of class 1 has to be empty before class 2 will be served. The amount of extra delay will depend on the utilization of both queues, queue size and AQMs configured. If the link is being underutilized and there is no congestion, the difference will be negligible. This is however not a problem. If there is unused capacity available, there is no need for the priority-flow to lower its transmission rate.

ECN is another congestion indicator that some congestion control algorithms use. As discussed in section 3.1.3, NADA effectively converts the packet mark ratio to an equivalent delay value. In other words, for a value x representing the packet mark ratio, there exists a value $\widetilde{x}$ extra delay that would be interpreted the same by the congestion control algorithm[5]. SCReAM is less precise with the interpretation of the ECN congestion encountered (CE) marks, as it only acts on up to one CE mark per RTT.

We had some restrictions when delaying packets due to packet reordering problems and delayed packets being categorized as lost. We have no such restrictions with ECN markings. Therefore, besides marking a subsequent set of packets for a short duration, we can also mark a subset hereof using OpenFlow groups. More specifically, we create a group of type *select* for each flow that needs it. We create two action buckets in this group, one will rewrite the ECN bits to CE and the other leaves the packets unchanged. We also specify weights for each of the action buckets. The *select* group will process each packet using one of the buckets which is selected based on the weighted round-robin algorithm.

Besides solely using the meter statistics to determine and configure extra table entries, we can also make use of the remarking being done by the meter. We configure the meter bands with *prec_level* = 1, which means that the DSCP field of packets surpassing the configured band rate will be rewritten. Another marking strategy is to CE mark all packets that have been remarked by the meter. This can be done by matching on the DSCP field on the marking table entry. Like above, we can also rewrite a subset of these packets using a group. The effectiveness of setting CE on the meter-marked packets depends on the congestion control algorithm being used by the media flow. The effect can be made smaller using the aforementioned groups and adapting the bucket weights.

Finally, we can also drop packets to signal congestion. This is the fundamental congestion indicator used since the first TCP congestion control algorithm. OpenFlow has a *drop* meter band which will limit flows to the configured band rate (and burst). This is a way to forcibly limit the bandwidth usage of flows without queueing at the cost of packet drops. Instead of using the drop meter we left the default meter configuration and dropped the packets based on the DSCP bits in table 1. This allows us to keep the standard meter configuration we are using. Again, this also allows us to drop only a subset of these marked packets by using a group.

[5]This mapping varies since the delay undergoes a non-linear transformation.

As explained in section 4.3, this controller will be used in combination with an application (see section 5.1.6). The controller and application communicate via an API we implemented. The API allows the application to request a list of DPIDs (datapath IDs) and the meter statistics per DPID. Finally, the controller also accepts requests to apply one of the above described actions to a priority-flow.

5.1.6 SDN application

The goal of this application is to determine which flows are surpassing their allocated bandwidth quota and take appropriate action to reduce the bandwidth in use by these flows. We monitor the utilization by periodically requesting meter statistics from the controller using the previously described API. The relevant statistics received are total byte- and packet- count, band byte and packet counts and meter duration in milliseconds. Using the total byte count we can determine the average rate of the flow over its lifespan or during a shorter interval using the data from two reports. Furthermore, we use the band statistics to detect peaks in bandwidth usage that might not be visible with the calculated average, depending on the statistics request interval and burst duration. We want to have the ability to detect these peaks since these can cause queueing delays (for other flows). We can look at the ratio over the entire duration of the flow by using the values in the last report, or look at this ratio over the last n seconds of the flow by using the statistics from two reports. In our implementation have a sample interval of 1 second and we store the newest 10 samples. This means we can calculate the above described metrics over any sub-interval of the last 10 seconds.

Our implementation mainly looks at the band byte ratio and not at the average bandwidth.

We take the ratio of band bytes and total bytes over the last 2 seconds and compare it with a threshold. How we use the statistics varies slightly per method and will be explained in chapter 6. The basis of it is that we look at the ratio of band bytes and total bytes and compare it to a threshold. If it's above this threshold we sent a request to the controller to configure entries to signal congestion. For this proof of concept, we tested the different approaches manually. Furthermore, we configured many variables statically which could be changed on a per-flow basis using the flow statistics. However, it's trivial to let the controller/application determine if ECN is supported. Furthermore, after applying extra rules to manipulate the congestion indicators, the application could look at how the flow reacts and determine if it was successful or if another approach should be taken, thereby having an application that can adapt to the congestion control algorithm in use by the flow.

Figure 5.5 shows a diagram of our solution. Below we list the steps that are completed during each iteration.

1. Send GET request for statistics of all meters (per switch).

2. Send OpenFlow OFPST_METER statistics request message.

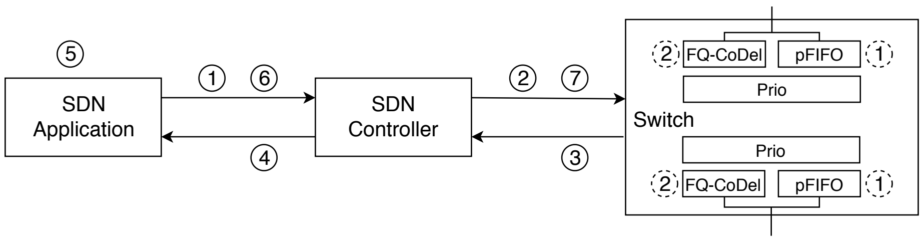


Figure 5.5: **Proposed solution with numbered steps**

3. Reply to OFPST_METER. An array of ofp_meter_stats structures is returned, one for each meter.
4. Reply to GET request (step 1) with meter stats in JSON format.
5. Parse the received statistics and determine if actions are required.

For each flow requiring a change in action we have the following steps.

6. Send POST request containing the command and required key-value pairs in the request body.
7. Send required OFPT_FLOW_MOD and/or OFPT_GROUP_MOD messages to the switch.

Table 5.1 show the implemented API requests. The *flowentry* request is used to manipulate the media flows. The {cmd} variable can be *add*, *modify*, or *delete*. This request is sent with a JSON-formatted body containing the required information. The three required fields are *flag*, *dpid* and *meter-id*. The flag indicates to the controller which method it should use to signal congestion. Table 5.2 show the available flags and additional required fields. The *dpid* and *meter-id* fields are used to uniquely identify each flow.

Table 5.1: **API requests**

Request	HTTP Method	Description
/switches	GET	Request list of datapath ids
/meter/{dpid}	GET	Request meter statistics of all meters
/meter/{dpid}/{meter_id}	GET	Request meter statistic of specific meter
/flowentry/{cmd}	POST	Send flow manipulation change request

5.2 Benchmarking BOFUSS

5.2.1 Issue with the switch

In the early phases of testing, we noticed consistent packet drops from the switch when sending a video stream with ffmpeg, even though the average transmission rate was multiple times smaller than the maximum bandwidth measured by iperf.
We created a measurement setup with two nodes. The first node generates

Table 5.2: **Flags and required fields**

Flags	Fields
FLAG_ECN_METER_DIR	N.A
FLAG_ECN_METER_GROUP	group_weight1, group_weight2
FLAG_ECN_BURST	hard_timeout
FLAG_ECN_GROUP	group_weight1, group_weight2, hard_timeout (optional)
FLAG_DELAY_BURST	hard_timeout
FLAG_DELAY_PERM]	queue (optional)
FLAG_DROP_METER_DIR	N.A
FLAG_DROP_METER_GROUP	group_weight1, group_weight2

the traffic and sends it to the second node. Node-2 forwards it back to node-1 through a different pair of interfaces. Node-2, the device under test, had BOFUSS installed.

The first test was to look at the instantaneous bandwidth of the video traffic and of iperf. To have the same conditions as we would have during later measurements, we made use of the same controller which preinstalls flows, marked the traffic as high priority and used the meter instruction during these measurements as well. To strictly measure the datapath capabilities, we also preinstalled the entries needed by the test traffic.
The purpose of this first test was simply to show the issue. We first send a video stream using ffmpeg over RTP/UDP followed by an iperf data stream over UDP. The flow statistics can be seen in table 5.3. Here we can see that the standard deviation of the video stream is over an order of magnitude larger than the average bandwidth. This large variation in bandwidth results in the video stream having a greater packet loss even though the average bandwidth is lower. Given these results, we decided to perform a benchmark of BOFUSS to assess what its limits are.

	Average bandwidth	Standard deviation	Packet loss
Video	$12.17X10^6$ bit/s	$4,43X10^8$ bits/s	4.115%
Iperf	$41.14X10^6$ bits/s	$7.63X10^6$ bits/s	0%

Table 5.3: **Preliminary measurement results**

5.2.2 Benchmarking

RFC 2544 [64] provides benchmarking methodology for network interconnect devices. The purpose of this section is not to present a full benchmark of the switch but to determine the limitations that are relevant to us. We therefore perform only a subset of the recommended tests.

As per the guidelines in [64] we use the measurement setup with one sending and receiving node and the device under test, use the same controller as what will be used during our experiment, pre-populate the flow tables and pre-populate the arp table. We will perform a bidirectional test since we will also have bidirectional traffic during our experiments.

To approximate the traffic we will encounter during our experiment we had a high-priority udp flow with DSCP set to 2 and best effort traffic that was changed for each run. Both of these flows were generated with iperf, running in bi-directional mode.

First we ran the meausurement with the best effort flow ranging from 30 to 140 Mbps with 10 Mbps increments (fig 5.6). With these results we narrowed down our range and ran the measurement with 2 Mbps increments (fig 5.7). From these results we decided to run the links towards the switch at 23 Mbit/s. While the measurements shown here show some packet drops, we observed stable behaviour across measurements when using this rate. We accomplished this by

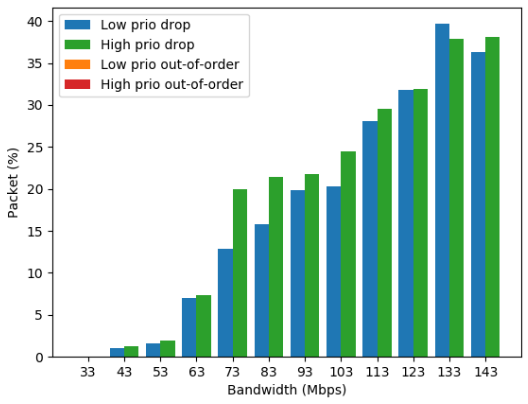
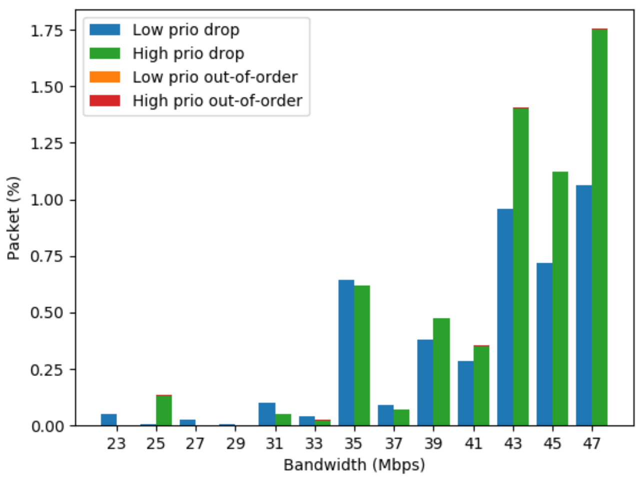
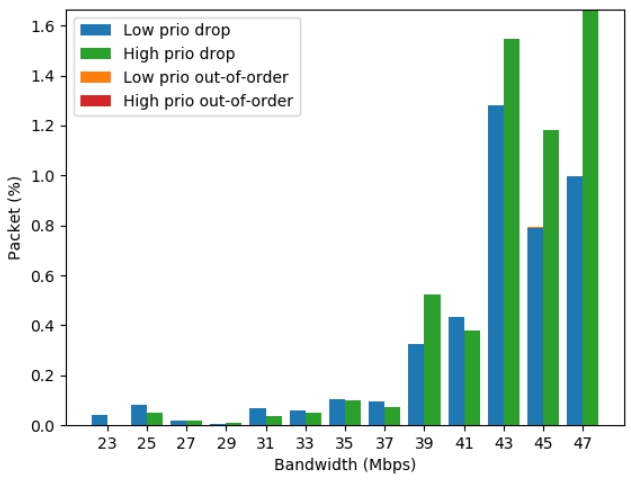


Figure 5.6: **Packet drop and out-of-order of iperf UDP stream; out-of-order bars are stack on top of the flow's corresponding drop bar**



(a) **Direction I**



(b) **Direction II**

Figure 5.7: **Packet drop and out-of-order of UDP flows; out-of-order bars are stack on top of the flow's corresponding drop bar**

5.3 Measurements

We will focus our measurements on the delay- and ECN-based manipulations since the loss-based approach is more trivial and is an action that we would like to avoid as much as possible in a real-time media stream. Furthermore, all real-time congestion control algorithms are designed to predominantly use delay as congestion indication.

We tested the delay-based actions with GCC using the setup described in section 5.1.3 via the Google Chrome browser[6]. Testing the ECN-based actions with WebRTC is a bit more complex. We can't use the same WebRTC setup as before since GCC doesn't support ECN. There is no ready-made software for the other two congestion control algorithms. NADA only has implementations for use in NS2 and NS3 (Network Simulator v2 and v3). SCReAM has a plugin for GStreamer[7] which provides classes with the main algorithm components. The use hereof requires more development than time permitted. However, we expect it to behave similar to TCP's congestion algorithms with respect to it's interpretation of ECN.

We decided to test the ECN actions using a TCP flow with the default congestion control algorithm in Ubuntu 18, namely Cubic. We won't be able to see changes in frame rate and resolution as with the WebRTC testing, but we will still be able to analyze the performance since all the packet capture measurements can still be executed.

[6]Google Chrome version 76.0.3809.87

[7]An opensource multimedia framework.

It's important to point out that there are some key differences between the interpretation of CE marked packets in TCP congestion control algorithms and the algorithms designed for real-time traffic.

Use of ECN in TCP
When a packet with the CE mark is received at the destination, the sender has to be informed about about the detected congestion. This is done by setting the ECN-Echo (ECE) flag of the subsequent acknowledgement (ACK) packet. For robustness, the ECE flag will be set on all ACK packets sent subsequently. The sender will interpret the received ECE marked acknowledgement as an indication to congestion in the same way as a lost packet. The sender will lower its congestion window and slow start threshold and will signal this to the receiver by setting the congestion window reduced (CWR) flag on the next data packet. The receiver will stop setting the ECE flag on the ACK packets after it receives the packet with the CWR flag. The way the ECN marks are handled means that multiple CE marks within one RTT of the first mark will not have an effect on the transmission rate.

Use of ECN in real-time congestion control algorithms
While this rough congestion indication is good enough for TCP, which can easily retransmit lost packets, real-time flow are much more sensitive to delay and packet loss. It is therefore beneficial to have a more precise indication of the congestion level and being able to adjust the sending rate in a more granular way [65]. As explained in section 3.1.3, NADA uses the ratio of CE marked packets. SCReAM does however deviate from the above logic and depends on delay for more fine tuned adjustment of its sending rate. SCReAM can optionally be implemented in such a way that allows for a smaller reduction in sending rate due to CE marked packets compared to packet loss. SCReAM does however ignore multiple CE marked packets within one RTT, the same as the TCP's congestion control algorithms. Our tests with TCP will therefore be representative of how SCReAM would behave.

Finally, we will compare our solutions to a HTB setup in which we configure the same bandwidth as in the OpenFlow meters. This will allow us to compare our solution not only to a baseline but also to a less complex solution with reserved resources. The configuration of this setup is shown in listing 5.3. It maintains the same structure as that of our solution but replaces the *prio* qdisc with a HTB qdisc. We allow both classes to use the full link capacity but guarantee 2Mbit/s to the prio-flow class. In the HTB class configuration we also experimented with priority configuration. We performed tests with and without the priority parameter configured.

Listing 5.3: **HTB queueing configuration**

```
$ sudo tc qdisc replace dev $DEV root handle 100: htb default 1
$ sudo tc class add dev $DEV parent 100: classid 100:1 htb rate 18mbit

$ sudo tc qdisc add dev $DEV parent 100:1 handle 1: htb default 2
$ sudo tc class add dev $DEV parent 1: classid 1:10 htb rate 18mbit
$ sudo tc class add dev $DEV parent 1:10 classid 1:1 htb prio 1 \
> rate 2mbit ceil 18mbit
$ sudo tc class add dev $DEV parent 1:10 classid 1:2 htb prio 2 \
> rate 16mbit ceil 18mbit
```

ECN testing with iperf and TCP

Self-clocked congestion control algorithms control the transmission rate via the congestion window (CWND) size, which controls the number of in-flight bytes. The rate can be calculated with the CWND size and the round-trip time (RTT):

$$Rate = \frac{CWND}{RTT}. \tag{5.1}$$

The iperf flow will be configured to have a similar rate to the WebRTC video stream we are testing with (2.5 Mbit/s). This test flow will have the highest priority in the network, and will therefore have a minimum RTT of approximately 1 ms. The actual delay will often be higher due to self-inflicted congestion. Still, due to how small the RTT is, even if the CWND is equal to one packet, the resulting rate will be way above our target. Under the assumption that the self-inflicted congestion will be negligible and taking the smallest possible CWND, the rate results in

$$Rate = \frac{CWND}{RTT} = \frac{MSS}{RTT} = \frac{MTU - 40bytes}{RTT} = \frac{1460bytes}{1ms} = 11.68Mbit/s. \tag{5.2}$$

This doesn't mean that iperf is unable to send at 2.5 Mbit/s as configured, but it means that indicating congestion and the resulting reduction in the CWND size won't actually reduce the bandwidth in use by the flow since the rate set by the CWND will be greater than the application level configured rate. To allow the rate to be controlled by the CWND, we will configure a smaller Maximum Segment Size (MSS). We set the MSS to the minimum value that can be configured with iperf, 88 bytes. This allows us to achieve a much lower rate.

The above problem doesn't exist in WebRTC. The congestion control algorithms developed for WebRTC also modify the target encoding rate based on congestion. The transmission rate is therefore controlled at the information source in the application layer and not purely with the congestion window like TCP's congestion control algorithms.

Chapter 6

Performance Evaluation and Analysis

In this chapter we will perform various measurements with the proposed solution and analyze the performance hereof. First, we will look at baseline measurements and validation thereof. This will be followed by analysis of the delay- and ECN-based actions. Afterwards we will measure the performance with a HTB-based setup and conclude by comparing the different solutions.

For experiments using a WebRTC flow we will have plots showing the video bitrate, frame rate and resolution. This data was gathered using WebRTC's built-in *getStats()* function. The latency scatter plots are generated from the packet captures, where each dot represents a packet. Using information from *getStats()*, we also verified that the bandwidth was not being limited by the CPU. Throughout this chapter we will use q-1 and q-2 to refer to classes 1:1 and 1:2 respectively of the configured priority qdisc. Q-1 has the highest priority, followed by q-2. Class 1:3 remains unused. Unless stated otherwise, all measurements were performed using 5 TCP flows as competing best-effort traffic using the Cubic congestion control algorithm and sent through q-2.

6.1 Baseline

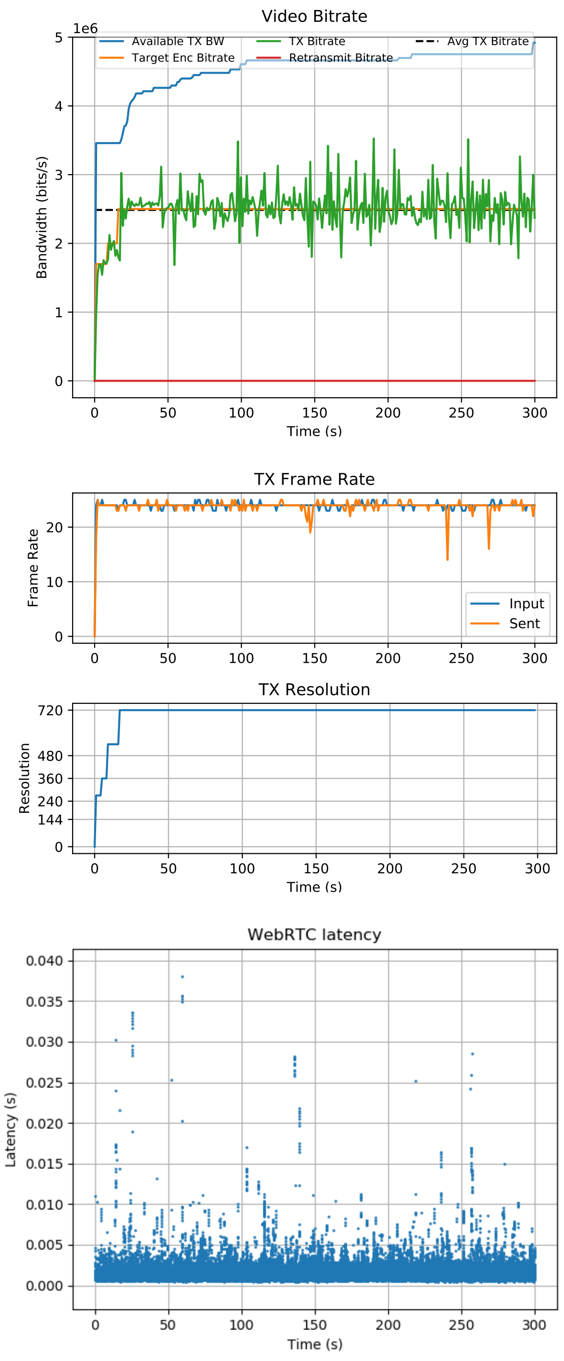
Table 6.1 shows the baseline statistics gathered by packet captures. Each baseline measurement sends the WebRTC flow solely through the specified queue. Sending the WebRTC flow through q-1 (BOFUSS q-1 measurement) behaves as expected and allows the stream to reach its maximum encoding rate, have a smaller latency compared to the BE flows and no packet loss. As expected, the BOFUSS q-2 measurement shows a much lower rate, higher delay and packet loss.

We validated these results by performing the same measurements using OvS. The OvS q-1 statistics are very similar to those measured with BOFUSS, with OvS having a slightly smaller delay. This could be due to OvS running in kernel space unlike BOFUSS which runs in user-space. We see a larger variation between the two switches when sending the WebRTC flow through q-2 (OvS q-2). The most prominent difference is the mean bitrate achieved by the WebRTC flow. When using OvS, the WebRTC flow gets a larger share of the available bandwidth, but has a larger delay and higher packet loss. This increase in delay and packet loss is due to higher rate WebRTC is transmitting at and the higher total bandwidth utilization. We believe the cause for the lower rate of the WebRTC flow when using BOFUSS is the resulting delay pattern. As can be seen in figures 6.1b and 6.2, using the different switches lead to very different delay (and bitrate) plots. This delay behaviour is likely the cause of the lower rate GCC is transmitting at. BOFUSS is therefore essentially influencing the way the available resources are used by GCC, and ultimately, influencing the division of resources between loss- and delay-based congestion control algorithms. The division of resources between the loss based (BE) flows is unaffected by the switch.

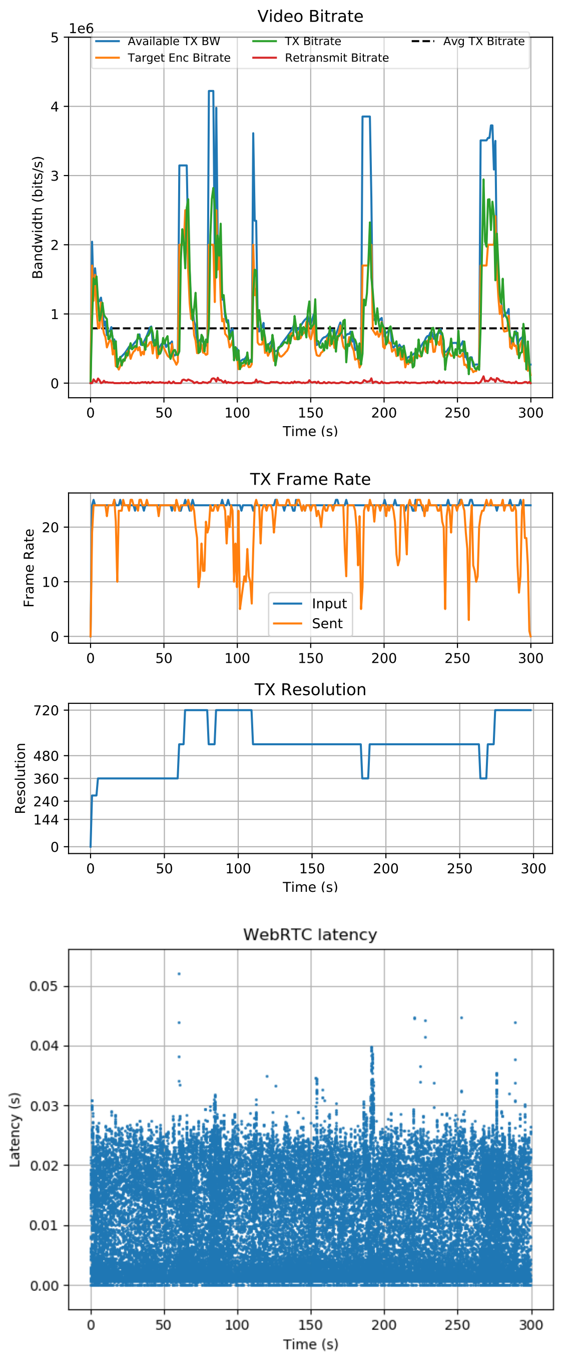
While undesirable, this behaviour doesn't pose a problem for our measurements. The q-2 measurements are ultimately mostly influenced by the type and number of competing (BE) flows, a number which we chose arbitrarily. Using 10 competing TCP flows and OvS (OvS q-2 10p measurement) has similar statistics to the BOFUSS q-2 measurement using 5 TCP flows. The WebRTC/priority flow we will use to measure will mostly be sent through q-1, which is well behaved. We can therefore move past this discrepancy and continue with our measurements as planned.

Table 6.1: **Baseline measurements with BOFUSS, OvS and both queues. Data extracted from packet capture. WebRTC rate calculated using 0.15s averages.**

Packet capture data	BOFUSS q-1	BOFUSS q-2	OvS q-1	OvS q-2	OvS q-2 10p
WebRTC rate - mean (Mbit/s)	2.66	0.89	2.66	1.81	0.900
WebRTC rate - std (Mbit/s)	0.97	0.66	0.96	1.01	0.419
WebRTC delay - min (ms)	0.41	0.45	0.45	0.54	0.52
WebRTC delay - max (ms)	38.09	52.10	40.32	315	157.06
WebRTC delay - mean (ms)	1.57	10.23	1.40	16.99	8.54
WebRTC delay - 90th perc. (ms)	2.23	21.52	1.86	42.24	17.59
WebRTC delay - 95th perc. (ms)	2.65	23.52	2.06	75.14	31.03
WebRTC delay - 99th perc. (ms)	4.78	27.36	2.57	167.36	85.18
WebRTC packet loss (%)	0.000	1.789	0.000	5.517	2.670
WebRTC Jitter - max (ms)	36.88	35.82	38.96	44.82	47.18
WebRTC Jitter - mean (ms)	-9.83e-05	1.38e-04	2.03e-06	4.59e-04	2.87e-06
WebRTC packet out of order - tot	0	0	0	0	0
BE flows rate - mean (Mbit/s)	13.38	14.81	14.27	15.05	15.70
BE RTT - mean (ms)	25.56	23.89	7.77	4.485	6.60

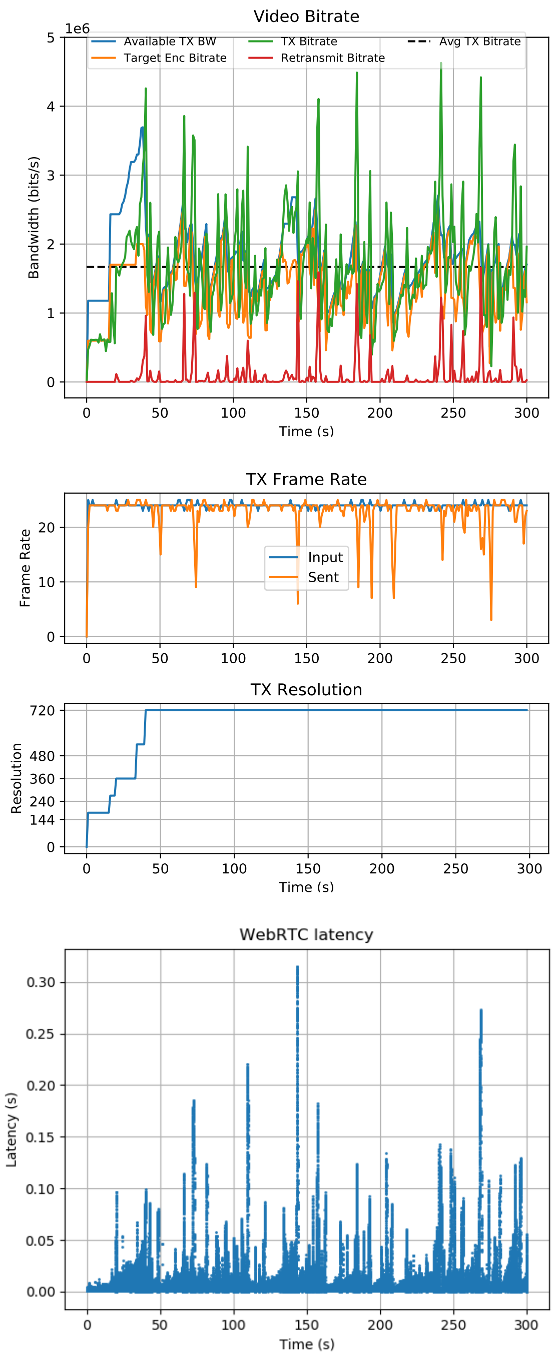


(a) **With network prioritization (BOFUSS q-1)**

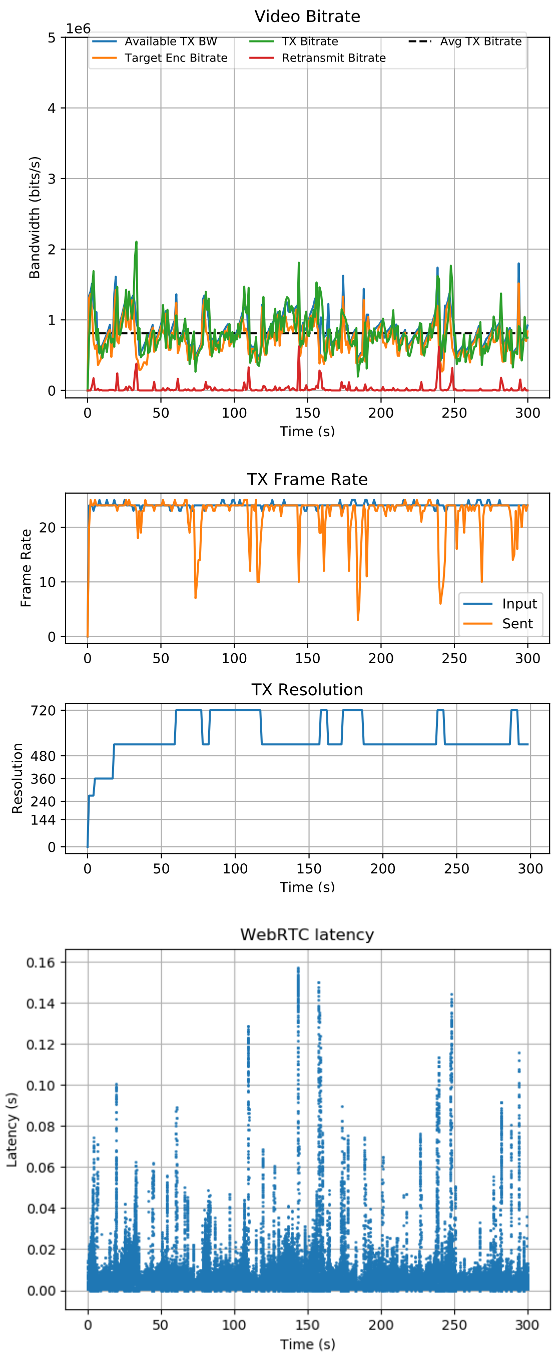


(b) **Without network prioritization (BOFUSS q-2)**

Figure 6.1: **Baseline measurement of WebRTC flow using BOFUSS and 5 competing TCP flows**



(a) **5 parallel TCP flows (OvS q-2)**



(b) **10 parallel TCP flows (OvS q-2 10p)**

Figure 6.2: **Validation with Open vSwitch, without network prioritization**

6.2 Delay-based actions

As was explained in section 5.1.5, we will be moving the priority flows between the two queues based on the switch-measured flow statistics. We will start by looking at what effect the queue switch has on the WebRTC flow. We temporarily moved the WebRTC flow to q-2 for 3 seconds at the 120s and 240s marks. The duration was enforced by configuring the *hard_timeout* parameter of the OF table entry. The results of this test can be seen in figure 6.3. Here we can see the increase in latency, the resulting drop in available bandwidth detected by GCC and the reduction in transmission rate.

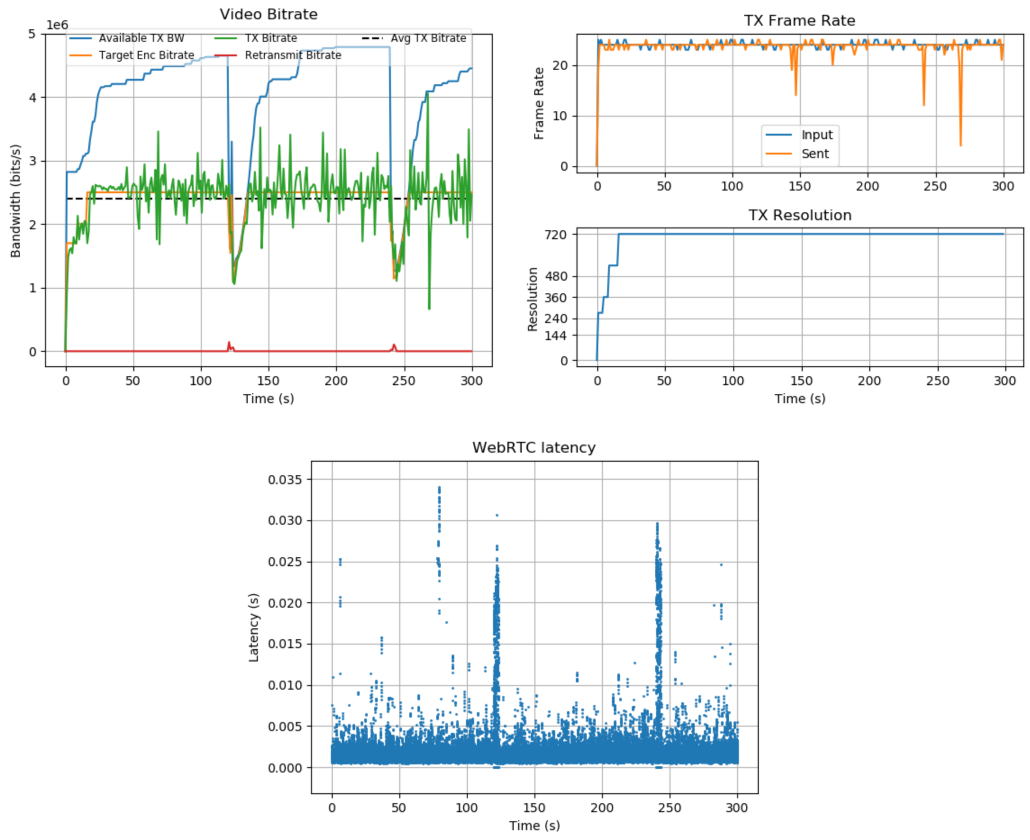


Figure 6.3: **Effect of temporarily moving the WebRTC flow from the high priority queue to the BE queue**

For the actual measurements we ran two tests using different metrics for each to decide if a queue change is necessary. Besides the original idea which was to use the meter to limit the flow to a maximum rate equal to the configured meter band, we also performed a measurement using the mean bitrate of the flow. The statistics of both measurements are presented in table 6.2, and the relevant plots in figure 6.4. For the mean rate based test we calculated the mean bitrate of the flow based of the byte count of the meter and the meter duration.

For the meter band ratio test we looked at the ratio between the number of packets in the configured band and the total number of packets since the last query.

As expected, the more we limited the bitrate, the higher the delay is. Figure 6.4 shows that this method results in repeated and drastic changes in bitrate. To try to alleviate this problem, we also performed measurements using a smaller *hard_timeout* value. This resulted in less drastic drops in bitrate but also more inconsistent results. We found that the flow entry stays present for up to a second more than the configured *hard_timeout*[1]. This has a noticeable effect when using smaller *hard_timeout* values. Another factor that will influence the behaviour of this method is the utilization of q-2, which we have no control over.

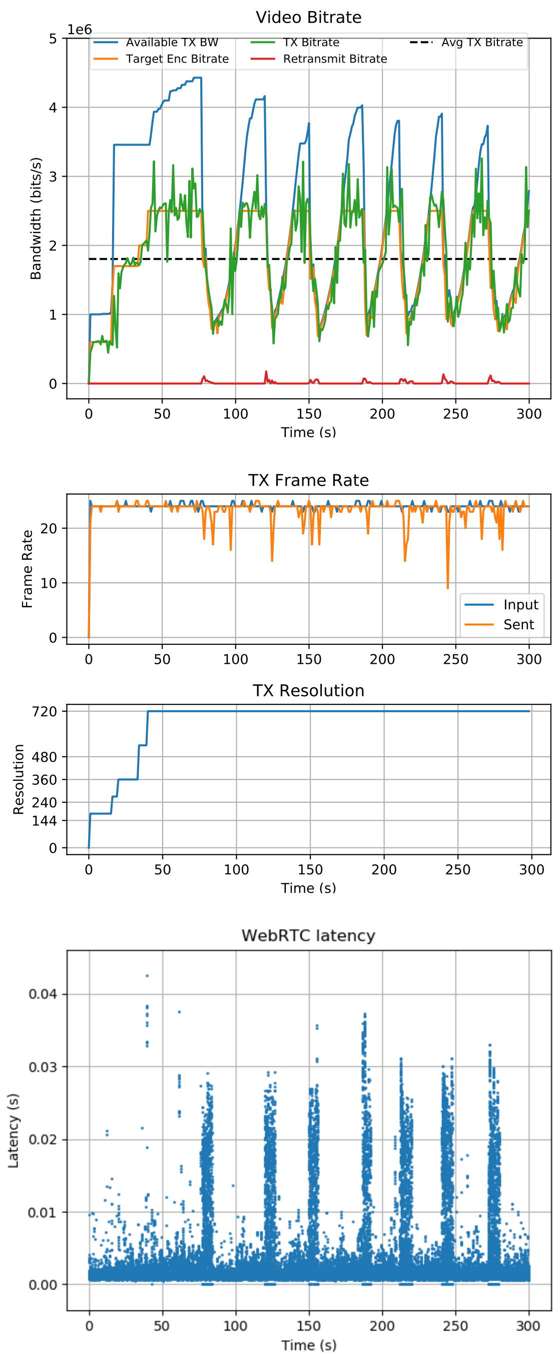
We can see clear dips in frame rate when the packets are delayed, while the resolution is more stable. The way in which the frame rate and resolution are adapted due to variation in available capacity can be changed using the *degradationPreference* setting. It can be configured to maintain frame rate or resolution and degrade the other if necessary. We set this to *balanced*, the default setting.

Depending on the frequency of the delay action and the implemented jitter buffer, the application end-to-end delay might still be high. The more the target bandwidth differs from the nominal flow bitrate, the more often the flow will have to be delayed and the more likely this is to be a problem.

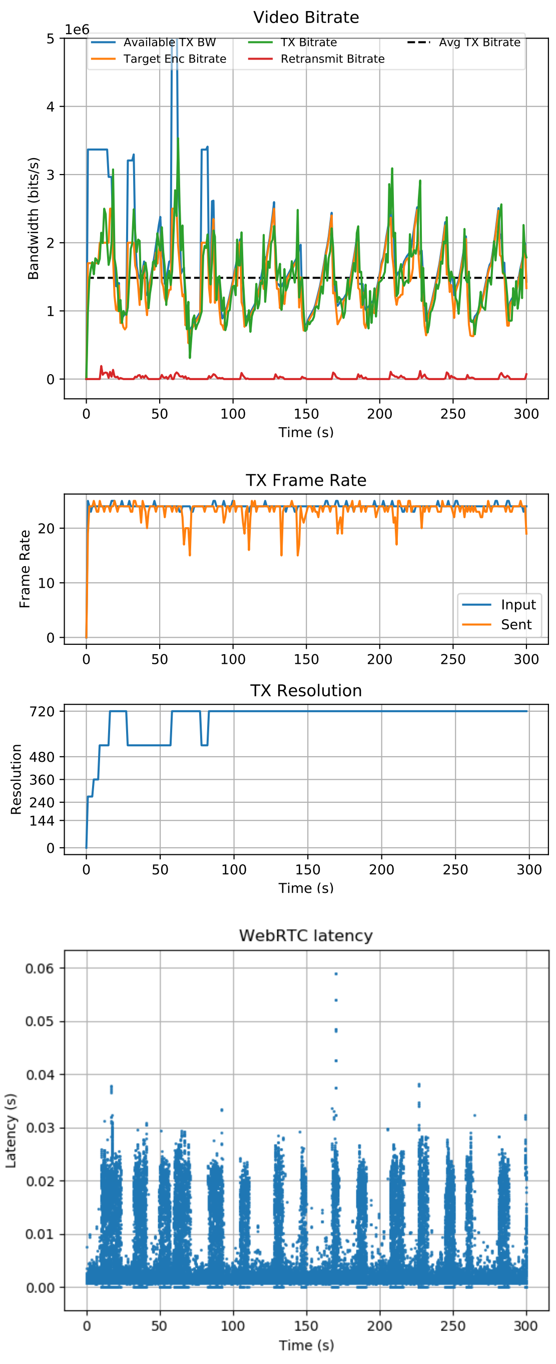
Table 6.2: **Delay based measurements. Mean bitrate and meter band ratio based flow monitoring. WebRTC rate calculated using 0.15s averages.**

Packet capture data	Mean BW	Meter band ratio
WebRTC rate - mean (Mbit/s)	1.95	1.49
WebRTC rate - std (Mbit/s)	0.99	0.67
WebRTC delay - min (ms)	0.48	0.53
WebRTC delay - max (ms)	42.49	59
WebRTC delay - mean (ms)	2.81	5.1
WebRTC delay - 90th perc. (ms)	4.52	16.91
WebRTC delay - 95th perc. (ms)	14.43	20.14
WebRTC delay - 99th perc. (ms)	23.87	24.24
WebRTC packet loss (%)	0.309	0.908
WebRTC Jitter - max (ms)	41.21	39.88
WebRTC Jitter - mean (ms)	-1.10e-04	-3.47e-06
WebRTC packet out of order	5/79938	36/69308
BE flows rate - mean (Mbit/s)	13.97	14.41
BE RTT - mean (ms)	24.60	22.76

[1]Measured for 1, 2 and 3 seconds configuration via packet captures; 50 iterations for each value.



(a) **Based on mean bitrate**



(b) **Based on meter band ratio**

Figure 6.4: **Delay based methods**

6.3 ECN-based actions

In this section we evaluate the ECN-based methods. Since we will be using TCP as the priority flow instead of WebRTC, we performed a new baseline q-1 measurement for reference (figure 6.5). The statistics hereof and of the other measurements performed in this section can be found in table 6.3.

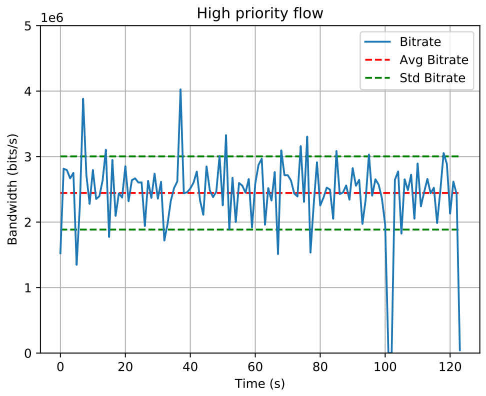


Figure 6.5: **Iperf baseline q-1**

The bitrate plot of the ECN burst marking action (figure 6.6a) shows a widely fluctuating bitrate, similar to what we saw with the delay burst method. Cubic recovers much faster after the congestion indication stops. This means that the marking burst has to be repeated much more frequently and is the reason we see many more variations compared to the delay based method. This effect is made worse by the way iperf behaves. As opposed to WebRTC, iperf aims to have the mean bitrate over the entire life of the flow be equal to the configured rate. This means that after a congestion period ends, it starts transmitting above the configured bitrate until the mean is equal to the configured rate. This will cause the ECN burst marking action to be triggered even sooner.

The group-based marking method provides a much more granular way to control the bitrate since we are able to adjust the action buckets' weights as opposed to only being able to enable and disable the marking like above. We simplified this adjustment by setting the weight of bucket-1 to be 1 and only adjusting the weight of bucket-2. Bucket-1 contains the ECN remark action and bucket-2 only forwards the packet. The starting weight for bucket-2 is 80. If the meter ratio over the last second is larger than the threshold (0.05), the weight is decreased with 10 (marked more often). If the meter ratio over the last 3 seconds is smaller than the threshold and the mean bitrate over the last 3 seconds is smaller than the bitrate-threshold (meter band value, 2 Mbit/s) the weight is increased with 5. The weight of bucket-2 over the duration of the measurement is shown in figure 6.6b. This implementation results in increased marking at the first sign of overuse and a slower and delayed decrease in packet marking ratio.
Still, the achieved mean bitrate is very close to the configured 2 Mbit/s in the network and has a small bitrate standard deviation.

Table 6.3: **Measurement of new q-1 baseline and ECN-based methods. WebRTC rate calculated using 1s averages.**

Packet capture data	Baseline q-1	Burst	Group	Meter	Meter-Group
Prio flow rate - mean (Mbit/s)	2.44	1.35	1.89	2.25	2.13
Prio flow rate - std (Mbit/s)	0.56	1.27	0.23	0.65	0.25
Prio flow delay - min (ms)	0.47	0.48	0.47	0.47	0.58
Prio flow delay - max (ms)	96.41	59.46	77.32	65.55	82.94
Prio flow delay - mean (ms)	8.03	4.15	2.46	5.20	5.61
Prio flow delay - 90th perc. (ms)	13.07	8.23	3.84	9.62	10.48
Prio flow delay - 95th perc. (ms)	14.77	10.06	4.79	11.37	12.16
Prio flow delay - 99th perc. (ms)	19.21	13.75	7.97	15.32	15.89
Prio flow - packet loss (%)	0.861	0.005	0.004	0.023	0.021
Prio flow Jitter - max (ms)	81.49	46.22	58.91	46.95	62.34
Prio flow Jitter - mean (ms)	-5.19e-05	-1.47e-04	-4.89e-05	-5.64e-05	-3.13e-05
BE flows rate - mean (Mbit/s)	12.90	14.13	13.59	13.36	13.36
BE RTT - mean (ms)	32.79	26.33	25.67	28.55	29.16

Next we tried the pure meter-based remarking, in which we set the CE bits on the packets marked by the the meter to have surpassed the 2 Mbit/s band. This, however, proved to not be sufficient since the resulting mean bitrate is higher than the band value. The meter-group marking combination, in which a subset of the meter marked packets are remarked with CE, gives a similar result. This was expected after looking at the meter-only measurements. This method uses the same action-bucket weight adaption method as the group only method. We see that the weight goes straight to the the minimum value and mostly remains there. At this point, half of the meter marked packets are being remarked with CE. Since TCP's congestion control algorithms ignore multiple markings within one RTT, this method has practically the same result as the meter-only method.

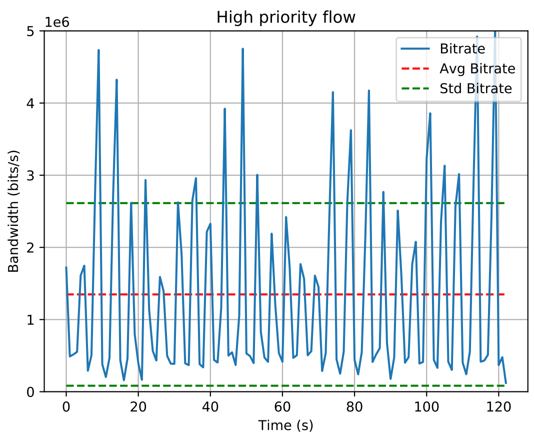
Even with the non-precise use of ECN markings employed by TCP, which only acts on one CE marked packet per RTT, we were able to control the bitrate quite well. This is of course aided by the low RTT of the setup and using q-1. Still, we expect even better results with NADA since it looks at the marking ratio, which we can modify.

Comparing the delay statistics we can observe that it varies between the different marking methods. This wasn't the expected result since they are all flowing purely through q-1. We can see that the higher the mean and standard deviation of the rate is, the higher the delay is as well. The increased delay is self-inflicted and is especially noticeable due to our relatively slow measurement network setup. WebRTC produces a smoother flow (see table 6.4) and is also able to detect and react on this increase in delay. We can see this in the baseline measurement (BOFUSS q-1) shown in table 6.1, which gets lower delays.

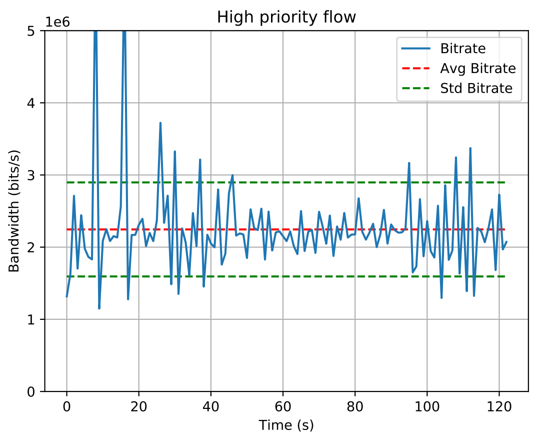
Furthermore, the higher the delay is in q-1, the higher it is in q-2 as well (BE RTT in table 6.3). This is expected since all packets in q-2 are effectively at the tail of q-1.

Table 6.4: **Bitrate statistics of WebRTC and Iperf calculated with a 0.15s averaging window.**

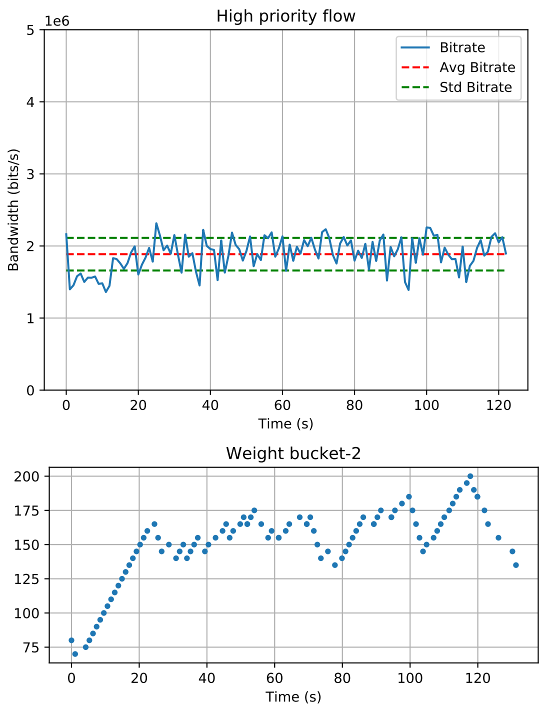
Packet capture data	BOFUSS q-1 WebRTC flow	Iperf q-1 Iperf flow
Prio Flow rate - mean (Mbit/s)	2.66	2.46
Prio Flow rate - std (Mbit/s)	0.97	1.37



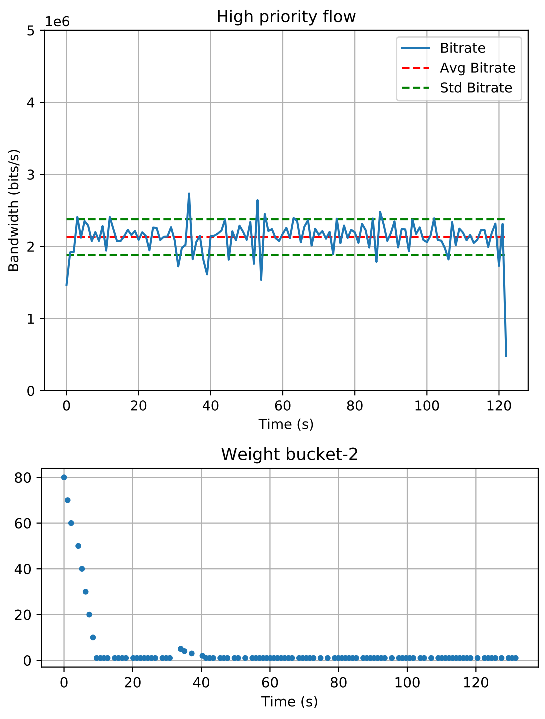
(a) **ECN burst**



(c) **ECN meter**



(b) **ECN group**



(d) **ECN meter-group**

Figure 6.6: **ECN based methods**

6.4 HTB-based Setup

The purpose of this section is to gather performance data of a simpler setup that is able to guarantee resources. We configured a HTB-based setup as shown in listing 5.3. We did two tests, with and without priority configured on the WebRTC HTB class.

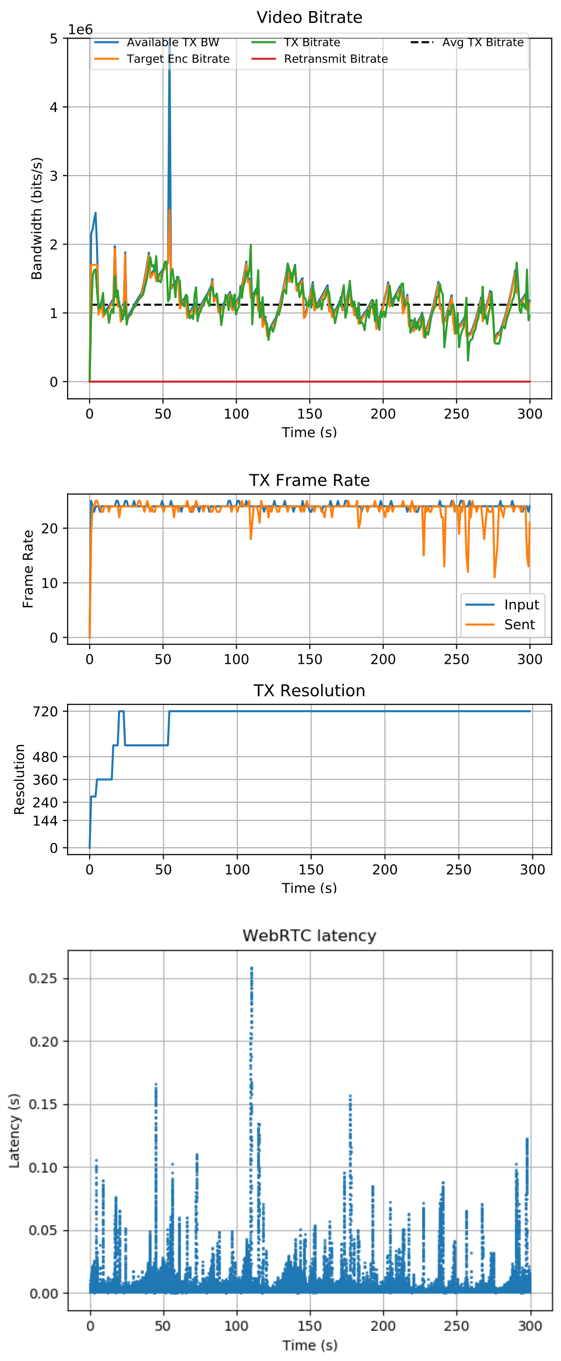
Table 6.5 show the BE flows only using 15.5 Mbit/s on both measurements instead of the expected 16 Mbit/s. These rates were measured by iperf, which measures the bitrate at the application level. Adding the data from the headers, this results in a bitrate of 16.2 Mbit/s at the data link layer.

Despite having 2 Mbit/s guaranteed bandwidth, the WebRTC flow doesn't make full use of it. While HTB allows borrowing, and we configured it as such, the BE flows are making full use of their share. This means that the configured 2 Mbit/s rate is not only the guaranteed rate but effectively also the maximum rate. This 2 Mbit/s transmission rate leads to queueing, self-inflicted delay and lower transmission rate.

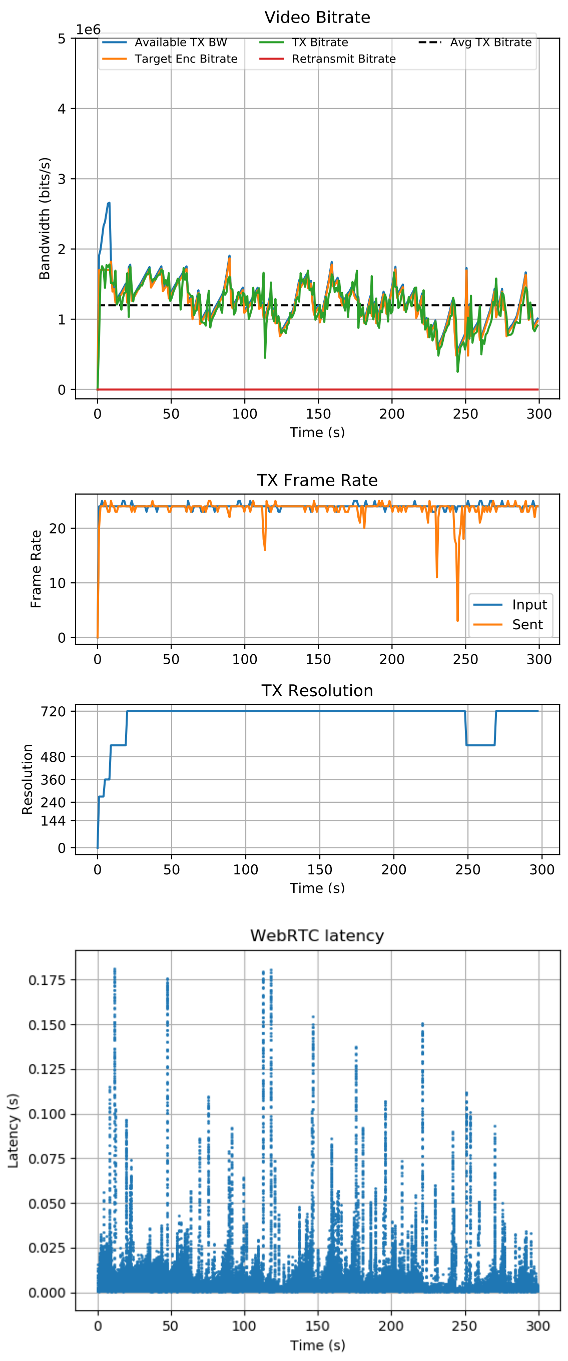
The priority setting works as follows. HTB offers excess bandwidth to high priority classes first, while still following the rate and borrowing rules. This holds as long as the class under consideration is not using more than the configured rate. The effect of this setting is very small and not visible in our measurements. In [66], they had to slow the link down to 100 kbytes/s to be able to show its effect.

Table 6.5: **Results using a HTB based setup. With and without class priority configured**

Packet capture data	HTB	HTB Prio
WebRTC rate - mean (Mbit/s)	1.12	1.31
WebRTC rate - std (Mbit/s)	0.42	0.44
WebRTC delay - min (ms)	0.44	0.44
WebRTC delay - max (ms)	258	181
WebRTC delay - mean (ms)	6.61	7.52
WebRTC delay - 90th perc. (ms)	13.20	15.30
WebRTC delay - 95th perc. (ms)	28.11	27.96
WebRTC delay - 99th perc. (ms)	80.12	90.88
WebRTC packet loss (%)	0.000	0.000
WebRTC Jitter - max (ms)	38.83	36.20
WebRTC Jitter - mean (ms)	-1.53e-04	-7.61e-05
BE flows rate - mean (Mbit/s)	15.51	15.49
BE RTT - mean (ms)	64.09	64.19



(a) **HTB setup without prioritization setting**



(b) **HTB setup with prioritization setting**

Figure 6.7: **Comparison of HTB-based setups**

6.5 Comparison of methods

Table 6.6 shows the best performing method of each set of measurements. Comparing the meter-based delay burst and HTB results we see that the former seems to performs better if we only look at the achieved mean bitrate and delay. However, due to the large variations and dips in the bitrate when using the delay-based approach, its frame rate is worse. The delay is also only marginally better, and not consistently better (90th percentile is worse), thus there is no real gain here either. HTB provides a more consistent bitrate without large dips in available bitrate. We see that the ECN-based group marking method is able to get the closest to the target rate compared to all other methods, and is independent of BE traffic. How close it can get depends on the burstiness of the flow and how much over-use we are willing to tolerate. As discussed in section 6.3, the delay of the measurements performed with TCP have a higher delay than those performed with WebRTC traffic. Using the ECN-based methods with WebRTC should result in similar delays to those measured in the WebRTC baselines. All of this makes the ECN group-based marking method the best performing one.

Table 6.6: **Comparison of methods**

Packet capture data	BOFUSS q-1	Meter-based delay	HTB Prio	iperf q-1	ECN Group
Prio flow rate - mean (Mbit/s)	2.66	1.49	1.31	2.44	1.89
Prio flow rate - std (Mbit/s)	0.97	0.67	0.44	0.56	0.23
Prio flow delay - min (ms)	0.41	0.53	0.44	0.47	0.47
Prio flow delay - max (ms)	38.09	59	181	96.41	77.32
Prio flow delay - mean (ms)	1.57	5.1	7.52	8.03	2.46
Prio flow delay - 90th perc. (ms)	2.23	16.91	15.30	13.07	3.84
Prio flow delay - 95th perc. (ms)	2.65	20.14	27.96	14.77	4.79
Prio flow delay - 99th perc. (ms)	4.78	24.24	90.88	19.21	7.97
Prio flow packet loss (%)	0.000	0.908	0.000	0.861	0.004
Prio flow Jitter - max (ms)	36.88	39.88	36.20	81.49	58.91
Prio flow Jitter - mean (ms)	-9.83e-05	-3.47e-06	-7.61e-05	-5.19e-05	-4.89e-05
BE flows rate - mean (Mbit/s)	13.38	14.41	15.49	12.90	13.59
BE RTT - mean (ms)	25.56	22.76	64.19	32.79	25.67

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Today's interactive media applications have strict network requirements to be able to provide a good quality service. Congestion control algorithms implemented by real-time applications such as WebRTC are able to get sufficient bandwidth [67]. However, getting a low delay is more difficult. Congestion control algorithms are able to minimize self-inflicted delay but are unable to cope with unnecessary delay caused by other flows.

Our contribution in this thesis is a new concept to provide low latency. We provide minimal latency by using the priority queueing discipline and sending real-time flows through class 1, the highest priority class. The performance will depend, among other things, on the utilization of this class. An easy way to control this is by limiting the number of flows allowed to use this class. This is a relatively straightforward problem to solve. We focused on controlling the utilization of each flow using class 1. By controlling the utilization of each flow we could also ensure fairness between all delay sensitive flows. Based on performance measurements, we show that by employing granular, ratio-based ECN-CE marking strategy we have a good control over the bitrate of the flow, without any drawbacks to these flows. With the support of various measurements, we show that the benefits of absolute priority can be provided with regards to delay while having control over the flows from the network.

7.2 Future Work

The implemented system shows a working proof-of-concept, but could still be improved. The application, which performs the logical work could be optimized and improved. The two main things are, (1) enabling the application to determine the support of congestion indicators (ECN, delay) on a per-flow basis. With this information, the action that will be used for each flow can be chosen individually. (2) the application could benefit from smarter/adaptive request of statistics instead of requesting all statistics at every iteration.

Our implementation applies the actions on the same switch as the meters. This was sufficient for our proof-of-concept using a single SDN switch. However, if this were to be deployed on a larger network, we could place the meters only on the ingress nodes of the network instead of placing a meter for every flow on every switch in its path. We could then choose to only forward the packets through class 1 on the bottleneck switch, were most of the queueing will take place, or on all nodes it flows through to minimize the delay over the entire network.

On the other hand, if such a system is to be implemented in a setting where the bottleneck is known in advance, such as an ADSL or DOCSIS uplink, it could then theoretically be consolidated into a self-contained scheduler.

Bibliography

[1] M. J. Prins, S. N. Gunkel, H. M. Stokking, and O. A. Niamut, “Togethervr: A framework for photorealistic shared media experiences in 360-degree vr,” *SMPTE Motion Imaging Journal*, vol. 127, no. 7, pp. 39–44, 2018.

[2] Z. Yuan, G. Ghinea, and G.-M. Muntean, “Quality of experience study for multiple sensorial media delivery,” in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 1142–1146, IEEE, 2014.

[3] Z. Yuan, S. Chen, G. Ghinea, and G.-M. Muntean, “User quality of experience of mulsemedia applications,” *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 11, no. 1s, p. 15, 2014.

[4] M. Al Jaafreh, M. Alowaidi, H. Al Osman, and A. El Saddik, “Multimodal systems, experiences, and communications: A review toward the tactile internet vision,” in *Recent Trends in Computer Applications* (J. M. Alja’am, A. El Saddik, and A. H. Sadka, eds.), (Cham), pp. 191–220, Springer International Publishing, 2018.

[5] L. De Cicco, G. Carlucci, and S. Mascolo, “Congestion control for webrtc: Standardization status and open issues,” *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 22–27, 2017.

[6] E. Janczukowicz, A. Braud, S. Tuffin, G. Fromentoux, A. Bouabdallah, and J.-M. Bonnin, “Specialized network services for webrtc: Turn-based architecture proposal,” in *Proceedings of the 1st Workshop on All-Web Real-Time Systems*, p. 3, ACM, 2015.

[7] I. N. W. Group *et al.*, “Rfc 2481 a proposal to add explicit congestion notification (ecn) to ip,” *IETF*, 1999.

[8] P. Martinsen, T. Reddy, D. Wing, and V. Singh, “Measurement of round-trip time and fractional loss using session traversal utilities for nat (stun),” RFC 7982, RFC Editor, September 2016.

[9] F. G. M. Petit-Huguenin, G. Salgueiro, “Path mtu discovery using session traversal utilities for nat (stun) draft-ietf-tram-stun-pmtud-11,” *Network Working Group, IETF*, 2019.

[10] D. W. P. Martinsen, “Stun traceroute draft-martinsen-tram-stuntrace-01,” *Network Working Group, IETF*, 2015.

[11] J. Gettys and K. Nichols, "Bufferbloat: dark buffers in the internet," *Communications of the ACM*, vol. 55, no. 1, pp. 57–65, 2012.

[12] G. S. M. P.-H. P. Martinsen, T. Andersen, "Traversal using relays around nat (turn) bandwidth probe draft-martinsen-tram-turnbandwidthprobe-00," *Network Working Group, IETF*, 2015.

[13] T. Boros, P. Zuraniewski, R. Hindriks, N. van Adrichem, E. Thomas, and L. D'Acunto, "Enabling superior and controllable video streaming qoe with 5g network orchestration," in *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 124–129, IEEE, 2019.

[14] E. Janczukowicz, A. Braud, S. Tuffin, A. Bouabdallah, and J.-M. Bonnin, "Evaluation of network solutions for improving webrtc quality," in *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–5, IEEE, 2016.

[15] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "Rfc8290: The flow queue codel packet scheduler andactive queue management algorithm, january 2018," *IETF*, 2018.

[16] R. Pan, P. Natarajan, F. Baker, and G. White, "Proportional integral controller enhanced (pie): A lightweight control scheme to address the bufferbloat problem," RFC 8033, RFC Editor, February 2017.

[17] G. Su, T. O. Pham, and S. He, "Qos guarantee for iptv using low latency queuing with various dropping schemes," in *2012 International Conference on Systems and Informatics (ICSAI2012)*, pp. 1551–1555, IEEE, 2012.

[18] R. Masoudi and A. Ghaffari, "Software defined networks: A survey," *Journal of Network and computer Applications*, vol. 67, pp. 1–25, 2016.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[20] N.-S. Ko, H. Heo, J.-D. Park, and H.-S. Park, "Openqflow: Scalable openflow with flow-based qos," *IEICE transactions on communications*, vol. 96, no. 2, pp. 479–488, 2013.

[21] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém, "Control of multiple packet schedulers for improving qos on openflow/sdn networking," in *2013 Second European Workshop on Software Defined Networks*, pp. 81–86, IEEE, 2013.

[22] C. Caba and J. Soler, "APIs for QoS Configuration in Software Defined Networks," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pp. 1–5, IEEE, 2015.

[23] D. Palma, J. Goncalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, and D. Staessens, "The queuepusher: Enabling queue management in openflow," in *The European Workshop on Software Defined Networking (EWSDN 2014)*, pp. 125–126, IEEE, 2014.

[24] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula, "Automated and scalable qos control for network convergence.," *INM/WREN*, vol. 10, no. 1, pp. 1–6, 2010.

[25] J. M. Wang, Y. Wang, X. Dai, and B. Bensaou, "Sdn-based multi-class qos guarantee in inter-data center communications," *IEEE Transactions on Cloud Computing*, vol. 7, no. 1, pp. 116–128, 2015.

[26] T.-N. Lin, Y.-M. Hsu, S.-Y. Kao, and P.-W. Chi, "Opene2eqos: Meter-based method for end-to-end qos of multimedia services over sdn," in *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 1–6, IEEE, 2016.

[27] S. N. Gunkel, M. Prins, H. Stokking, and O. Niamut, "Social vr platform: Building 360-degree shared vr spaces," in *Adjunct Publication of the 2017 ACM International Conference on Interactive Experiences for TV and Online Video*, pp. 83–84, ACM, 2017.

[28] S. Gunkel, M. Prins, H. Stokking, and O. Niamut, *WebVR meets WebRTC: Towards 360-degree social VR experiences.* IEEE, 2017.

[29] S. N. Gunkel, H. M. Stokking, M. J. Prins, N. van der Stap, F. B. t. Haar, and O. A. Niamut, "Virtual reality conferencing: multi-user immersive vr experiences on the web," in *Proceedings of the 9th ACM Multimedia Systems Conference*, pp. 498–501, ACM, 2018.

[30] S. Gunkel, H. Stokking, M. Prins, O. Niamut, E. Siahaan, and P. Cesar, "Experiencing virtual reality together: Social vr use case study," in *Proceedings of the 2018 ACM International Conference on Interactive Experiences for TV and Online Video*, pp. 233–238, ACM, 2018.

[31] S. N. Gunkel, H. Stokking, T. De Koninck, and O. Niamut, "Everyday photo-realistic social vr: Communicate and collaborate with an enhanced co-presence and immersion," *International Broadcasting Convention*, 2019.

[32] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2," RFC 6347, RFC Editor, January 2012. `http://www.rfc-editor.org/rfc/rfc6347.txt`.

[33] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and web performance.* " O'Reilly Media, Inc.", 2013.

[34] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The secure real-time transport protocol (srtp)," RFC 3711, RFC Editor, March 2004. `http://www.rfc-editor.org/rfc/rfc3711.txt`.

[35] A. Roach, "Webrtc video processing and codec requirements," RFC 7742, RFC Editor, March 2016.

[36] J. Valin and C. Bran, "Webrtc audio codec and processing requirements," RFC 7874, RFC Editor, May 2016.

[37] Z. S. R. Jesup, "Congestion control requirements for interactive real-time media; draft-ietf-rmcat-cc-requirements-09," *Network Working Group, IETF*, 2014.

[38] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and design of the google congestion control for web real-time communication (webrtc)," in *Proceedings of the 7th International Conference on Multimedia Systems*, p. 13, ACM, 2016.

[39] X. Zhu, R. Pan, M. Ramalho, S. Mena, P. Jones, J. Fu, and S. D'Aronco, "Nada: A unified congestion control scheme for real-time media; draft-ietf-rmcat-nada-10," *Network Working Group, IETF*, 2019.

[40] I. Johansson and Z. Sarker, "Self-clocked rate adaptation for multimedia," RFC 8298, RFC Editor, December 2017. `http://www.rfc-editor.org/rfc/rfc8298.txt`.

[41] S. Holmer, H. Lundin, G. Carlucci, L. D. Cicco, and S. Mascolo, "A google congestion control algorithm for real-time communication," Internet-Draft draft-ietf-rmcat-gcc-02, IETF Secretariat, July 2016. `http://www.ietf.org/internet-drafts/draft-ietf-rmcat-gcc-02.txt`.

[42] X. Zhu, R. *, M. Ramalho, and S. de la Cruz, "Nada: A unified congestion control scheme for real-time media," Internet-Draft draft-ietf-rmcat-nada-13, IETF Secretariat, September 2019. `http://www.ietf.org/internet-drafts/draft-ietf-rmcat-nada-13.txt`.

[43] I. Rec, "Y. 1541: Network performance objectives for ip-based services," *International Telecommunication Union, ITU-T*, 2003.

[44] J. Valin, K. Vos, and T. Terriberry, "Definition of the opus audio codec," RFC 6716, RFC Editor, September 2012.

[45] T. ITU, "Recommendation g. 114, one-way transmission time," *Series G: Transmission Systems and Media, Digital Systems and Networks, Telecommunication Standardization Sector of ITU*, 2000.

[46] K. S. Park and R. V. Kenyon, "Effects of network characteristics on human performance in a collaborative virtual environment," in *Proceedings IEEE Virtual Reality (Cat. No. 99CB36316)*, pp. 104–111, IEEE, 1999.

[47] M. Hoecker and M. Kunze, "An on-demand scaling stereoscopic 3d video streaming service in the cloud," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, p. 14, Sep 2013.

[48] L. Skorin-Kapov, D. Mikić, and D. Huljenić, "End-to-end qos for virtual reality services in umts," in *Proceedings of the 7th International Conference on Telecommunications ConTEL*, pp. 337–344, 2003.

[49] C. Liu, Y. Xie, M. J. Lee, and T. N. Saadawi, "Multipoint multimedia teleconference system with adaptive synchronization," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1422–1435, 1996.

[50] M. A. Brown, "Traffic control howto version 1.0.2." `http://tldp.org/HOWTO/Traffic-Control-HOWTO/`, 2006. [Online; accessed 29-May-2019].

[51] Devera, Martin, "Hierarchical token bucket theory." http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm, 2002. [Online; accessed 3-June-2019].

[52] I. Stoica, H. Zhang, and T. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," *IEEE/ACM transactions on Networking*, vol. 8, no. 2, pp. 185–199, 2000.

[53] K. Nichols, V. Jacobson, and E. McGregor, "Rfc8289: Controlled delay active queue management," *IETF*, 2018.

[54] M. H. MacGregor and W. Shi, "Deficits for bursty latency-critical flows: Drr++," in *Proceedings IEEE International Conference on Networks 2000 (ICON 2000). Networking Trends and Challenges in the New Millennium*, pp. 287–293, IEEE, 2000.

[55] K. Nichols, S. Blake, F. Baker, and D. Black, "Rfc2474: Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers," *Network Working Group, IETF*, 1998.

[56] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, "Rfc 2597: Assured forwarding phb group," *Network Working Group, IETF*, 1999.

[57] Nicira Inc., "Ovs 2.11 meter datapath." https://github.com/openvswitch/ovs/blob/v2.11.0/datapath/meter.c#L176, 2018. [Online; accessed 6-June-2019].

[58] B. Pfaff and B. Davie, "The open vswitch database management protocol," RFC 7047, RFC Editor, December 2013. http://www.rfc-editor.org/rfc/rfc7047.txt.

[59] Eder Leão Fernandes, "Openflow 1.3 software switch." http://cpqd.github.io/ofsoftswitch13/, 2018. [Online; accessed 30-May-2019].

[60] , "Lagopus software switch." https://github.com/lagopus/lagopus, 2018. [Online; accessed 30-May-2019].

[61] P. Jha, "End-to-end quality-of-service in software defined networking," Master's thesis, University of Dublin, Trinitiy College, 2017.

[62] R. Wallner and R. Cannistra, "An sdn approach: Quality of service using big switch's floodlight open-source controller," *Proceedings of the Asia-Pacific Advanced Network*, vol. 35, p. 14, 06 2013.

[63] "Best practices for benchmarking codel and fq codel." https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel/. Accessed: 2019-10-14, Version: 1.5.

[64] S. Bradner and J. McQuaid, "Rfc 2544 benchmarking methodology for network interconnect devices," *IETF*, 1999.

[65] M. Westerlund, I. Johansson, C. Perkins, P. O'Hanlon, and K. Carlberg, "Explicit congestion notification (ecn) for rtp over udp," RFC 6679, RFC Editor, August 2012. http://www.rfc-editor.org/rfc/rfc6679.txt.

[66] Devera, Martin, "Htb manual - user guide." `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`, 2002. [Online; accessed 3-June-2019].

[67] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers, and G. Zussman, "Performance evaluation of webrtc-based video conferencing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 45, no. 3, pp. 56–68, 2018.

Glossary

AQM Active queue management. 5, 7, 20, 32

CoDel Controlled Delay; an active queue management scheme . 20

CVE Collaborative Virtual Environment. 16, 17

DSCP Differentiated Services Code Point; a 6-bit field in the IP header used for packet classification. 28, 29, 32, 36

DTLS Datagram Transport Later Security. 10

ECN Explicit Congestion Notification. iii, 5, 6, 13, 15, 24, 31–33, 37, 38, 41, 48, 49, 53, 55

FQ-CoDel Fair Queueing Controlled Delay. 6, 20, 25, 30

GCC Google Congestion Control; A congestion control algorithm created by Google. 13, 15, 37, 45

H-FSC Hierarchical Fair Service Curve; a scheduler designed to provide a guaranteed rate and delay. 20, 24

HTB Hierarchical Token Bucket. 6, 7, 19, 24, 28–30, 38, 41, 51, 53

ICE Interactive Connectivity Establishment. 10

NADA Network Assisted Dynamic Adaptation; A congestion control algorithm created by Cisco. 13, 15, 37, 38, 49

OF OpenFlow. 6

qdisc queueing discipline; defined in Linux as a scheduler that orders packets to be sent according to given rules.. 19, 28, 30, 38, 41

QoE Quality of Experience; A measure of the user-perceived quality. 6, 9

QoS Quality of Service. 2

REMB Receiver Estimated Maximum Bitrate. 13, 24

RTCP Real-time Transport Control Protocol. 13

RTP Real-time Transport Protocol. 13–15, 35

SCReAM Self-Clocked Rate Adaptation for Multimedia; A congestion control algorithm created by Ericsson. 13–15, 32, 37, 38

SFQ Stochastic Fair Queueing. 7

SRTCP Secure Real-time Transport Control Protocol. 10, 11

SRTP Secure Real-time Transport Protocol. 10

STUN Session Traversal Utilities for NAT; It is used to get the public IP of a client behind a NAT. 10, 29

tc traffic control; a Linux utility to configure traffic control in kernel space. 28

TURN Traversal Using Relay NAT; A relay server used by WebRTC when a direct connection is not possible. 10, 29

WFQ Weighted Fair Queueing; . 6, 7