iRED: Improving the DASH QoS by dropping packets in programmable data planes

Leandro C. de Almeida[1, 2], Guilherme Matos[1], Rafael Pasquini[3], Chrysa Papagianni[4], and Fábio L. Verdi[1]

[1]Computing Department, Federal University of São Carlos, Sorocaba, SP Brazil
[2]Academic Unit of Informatics, Federal Institute of Paraíba, João Pessoa, PB, Brazil
[3]Faculty of Computing, Federal University of Uberlândia, Uberlândia, MG, Brazil
[4]Informatics Institute, University Of Amsterdam, Amsterdam, Netherlands
leandro.almeida@ifpb.edu.br, guilherme.matos@estudante.ufscar.br, rafael.pasquini@ufu.br, c.papagianni@uva.nl, verdi@ufscar.br

***Abstract*—Video services account for the largest share of all Internet traffic, demanding a network capable of supporting the requirements of delay-sensitive traffic. Fluctuations in network load can cause high delays in the queues of network routers, which tend to degrade the Quality of Service (QoS) for adaptive video streaming, such as Dynamic Adaptive Streaming over HTTP (DASH). This work is positioned in the scope of active management queues (AQM) to improve the QoS of a DASH service by means of dropping packets. One traditional AQM that adopts a packet drop policy is Random Early Detection (RED), developed to drain the flow in times of congestion and thus reduce queueing delay. We revisited and implemented a P4-based implementation of RED, named iRED (ingress RED), an algorithm capable of dropping packets at the ingress pipeline, an innovation compared to other AQM strategies based on dropping at the egress. iRED was evaluated in two scenarios. First, we compare iRED against state-of-art AQM algorithms employing egress packet dropping in terms of Round-Trip Time (RTT), throughput and their impact on resources usage. Our findings indicate that iRED outperforms existing P4-based approaches by approximately up to 2.5x in RTT and 0.75x in throughput for the given buffer sizes. Next, we compare iRED versus Tail Drop (TD) approach in an emulated programmable Content Delivery Network (CDN) employing DASH. Experiments indicate that the iRED improve the QoS by approximately 0.85x in terms of cached video available in the client's buffer and 0.9x in Frames Per Second (FPS) played.**

***Index Terms*—Quality of Service, Dynamic Adaptive Streaming over HTTP, Active Queue Management, Random Early Detection, Data plane programmability, P4.**

I. Introduction

Currently, the video service is dominating Internet traffic, reaching more than 80% in 2021 [1]. Within merely three months during the COVID-19 pandemic, video conferencing services increased by about 30x [2]. To support the required QoS, cloud and network service providers have had to scale their infrastructure to accommodate this massive demand.

Conceptually, the Internet was designed as a end-to-end network offering best-effort packet switching services, a non ideal scenario for streaming services which present stringent requirements in terms of delay and bandwidth assurance [3], usually defined as QoS and, more recently, also linked with the concept of Quality of Experience (QoE). The Internet evolved in several directions over time, to cope with ever increasing demands and new challenging applications, and it was not different for video streaming services. The killing technology nowadays offers adaptive bitrate schemes over HTTP and places video content closer to customers, seeking to mitigate the intrinsic properties of best-effort packet switching environments.

In this scenario, MPEG-DASH or simply DASH [4], occupies a prominent position, being the adopted solution by important players on this market, including Netflix® and Google® [5]. In essence, DASH supports video encoding at various mixtures of resolution, bitrate, frames per second, and other parameters, offering an adaptive self-service menu to customers, which are now able to pick the best combination according to their conditions [6]. Such combination is defined based on the resources available on the devices consuming such content and also based on the infrastructure status. A video is divided into chunks of the same time duration, allowing the video player to switch between the multiple video quality levels [7]. Although DASH is considered a "smart" mechanism due to its capability to adapt to fluctuations caused by network load, unfortunately such solution by itself is not enough to keep the pace required by certain Service Level Agreements (SLA) and other auxiliary mechanisms should be designed to jointly work with DASH.

Given this context, we revisited the RED algorithm [8], a well-known AQM mechanism used for randomly dropping packets, as a candidate solution to improve the DASH QoS. Thanks to recent advances in programmable hardware, we implement a RED-like algorithm using the P4 language [9], named iRED (ingress RED). iRED going beyond other AQM solutions in the data plane, such as CoDEL [10] and PI2 [11], that are based on dropping at the egress pipeline, where queuing information is accessible as part of the standard packet metadata. Our iRED design overcomes this limitation using advanced resources of mirroring and recirculation (with minimal overhead), sharing the queue congestion status per port from egress to ingress pipeline.

In our evaluations, we ran two types of experiments using the BMv2 software switch. First, we assess the impact of dropping packets at the ingress versus egress pipelines. We evaluate whether dropping packets at the ingress with iRED would have

978-3-903176-51-5 © 2022 IFIP

some advantage over other well-known AQM egress dropping algorithms, like CoDEL [10] and PI2 [11]. We analyze these algorithms in terms of RTT, throughput and utilization of resources.

In the second experiment, we evaluate the hypothesis that dropping packets is not necessarily bad for a video service [12]. Over the years, there was a common sense that networks should avoid dropping packets, since for each dropped TCP segment, a retransmission would be necessary. However, some studies [12] observed that for the specific case of video transmissions, dropping packets reduces end-to-end latency. We evaluate if iRED could improve the DASH QoS in an emulated programmable CDN, delivering content to video clients. For this case, we evaluated iRED and Tail Drop (TD) approaches.

In summary, the main contributions of this work are:

1) Implement an AQM algorithm, named iRED (ingress RED), for programmable data planes;
2) Design the AQM employing successfully an ingress packet drop policy with a minimal overhead and compare against existing P4-based implementations;
3) Deploy and evaluate the impact of using iRED algorithm in a DASH service.

This work is organized as follows; the fundamental concepts regarding implementing AQMs using programmable data planes and related works are briefly described in Section II. The proposal is described in Section III. In Section IV, the experiments and evaluation are detailed, including a brief view about DASH video service and the workloads used. Finally, the conclusions are depicted in Section V.

II. Related Work

Queuing delay occurs when a packet waits in the queue before being transmitted through an interface on a router. This waiting time can vary according to traffic load in the network that is crucial for sensitive-delay applications such as DASH. If the packet spends more time in the queue than a threshold, the DASH QoS tends to be degraded. This happens because packets are held in router queues, and hence the video player will not have frames to play, demanding video rebuffering [13]. We believe in the hypothesis that subtle packet drops can be used to control the queue occupancy, thus latency [10], which can be used for enhancing the quality of the video service in terms of delay [12].

We have observed that decreasing queuing delay is a topic that has been discussed with AQM proposals for the last three decades, such as RED [8], BLUE [14], CoDEL [15], CAKE [16] and PI2 [17]. Recently, CoDEL and PI2 were rewritten to be supported in state-of-the-art network equipment and available as open-source in P4 language with the aliases P4-CoDEL [10] and PI2 for P4 [11].

The queue occupancy is key for AQM algorithms since such metric is an input to calculate the probability of dropping a packet. Some AQM algorithms such as CoDEL [10] and Pi2 [11] utilize the queue delay per packet to decide if the packet should be dropped or not. However, in some programmable devices[1] such metric is available only at the egress pipeline which suggests that the AQM algorithm needs to be run at the egress, where the metric can be obtained.

A. The P4 programming language

P4 language [9] emerged in 2014 as an alternative and natural evolution of the network programmability paradigm, bringing new horizons to program the data plane of a network element. P4 is a declarative programming language for expressing the behavior of packet processors. It is a domain specific language with constructs (e.g., headers, parsers, actions, tables, control flows etc.) optimized for writing packet forwarding functions. Using P4, developers can program data plane packet pipelines based on a match/action architecture (see Fig. 1); they can create custom parsers for new protocol headers, define custom flow tables, the control flow between the tables, and custom actions. P4 programs allow developers to uniformly specify packet processing behavior for a variety of targets (ASICs, CPUs, NPUs etc.).

The execution of a P4 program follows a simple abstract forwarding model with distinct phases; when the packet arrives at the switch, it will cross the programmable parser and then the ingress block. In the programmable parser, the headers are exposed to be analyzed by the following blocks. The match-action logic determines the ingress and egress blocks. The non-programmable traffic manager encompasses functions such as packet scheduling, shaping, AQM etc., depending on the target's predefined and usually limited functionality. The packet crosses the queue, match-action pipeline in the egress block, and sent to the deparser, responsible for serialising the packet.

The current specification of the P4 language (P4_16), introduced the concept of the P4 architecture that defines the programmable blocks of a target and their interfaces. Along with the corresponding P4 compiler, it enables programming the P4 target. At the moment, the v1model architecture, depicted in Fig. 1, is used by the reference BMv2 software switch.

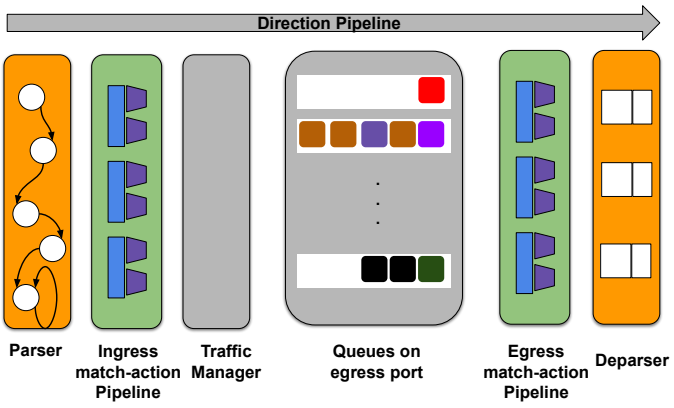


Fig. 1. Queues in V1model architecture.

[1]This is the case for BMv2 and tofino1. Tofino2 and Tofino3 allow to obtain queue occupancy from the ingress pipeline.

B. CoDEL

CoDEL is an AQM algorithm specified by the IETF in RFC 8289, focused on addressing the Bufferbloat problem [18]. The logic of the packet drop policy is applied in the egress pipeline, as described in Fig. 2. Codel tries to keep the queueing delay below a specified threshold (TARGET parameter) in a one-time interval (INTERVAL parameter). Codel ensures that the queueing delay will be periodically lower than the threshold, following these steps for each packet: i) if the queueing delay is below the threshold, a packet is never dropped; ii) if the threshold is reached by more than interval time units, the first packet will be dropped; iii) from then on, the interval between dropping packets is getting smaller until the threshold delay is reached.

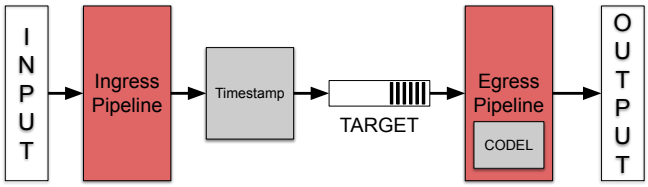


Fig. 2. P4-CoDEL integration in P4 reference pipeline. Adapted from [10].

C. PI2

PI2 is a linearized AQM for both classic and scalable TCP, based on Proportional Integral algorithm [17]. The PI2 uses queueing information (delay) per packet in conjunction with PI gain factors (α and β) to trigger the packet drop policy. The output probability of the basic PI controller is squared when dropping classic TCP packets or doubled when marking scalable TCP traffic. As with P4-CoDEL, PI2 logic is applied in the egress pipeline, as can be show in the Fig. 3.

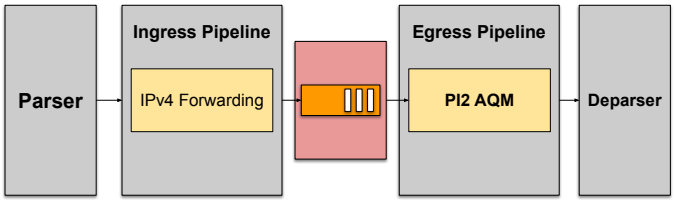


Fig. 3. PI2 for P4. Adapted from [11].

III. Design of iRED

Both studies presented above apply the packet dropping policy in the egress pipeline due to the constraints imposed by some programmable devices (e.g., tofino1). However, there is a price to pay for this choice. In both cases, the packets to be dropped have to go through the entire programmable hardware pipeline, wasting switch resources. Our solution goes beyond, using a mechanism based on cloning, recirculation, and dropping to overcome this barrier. We split our algorithm into two pieces, positioned at the ingress (action to drop) and egress (decision to drop) programmable blocks, as is shown in Fig. 4.

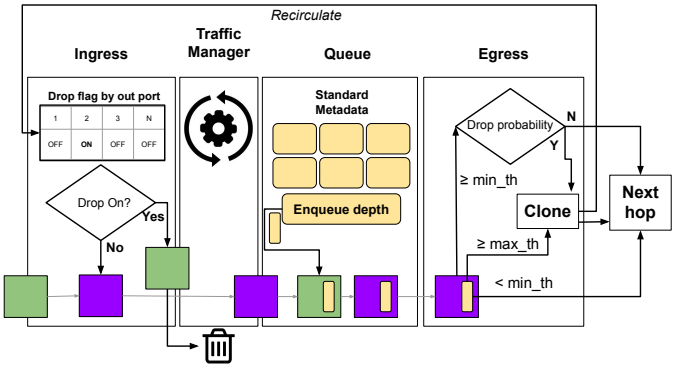


Fig. 4. iRED solution in V1model architecture.

For a better understanding, we start with the egress pipeline. As can be seen in Fig. 4, the decision on dropping the packet or not, employing when needed the dropping probability is done in the egress pipeline. If iRED verifies that the average queue size is between min-max thresholds, the current packet can be cloned or not, and the original packet is sent out to the next hop. The cloned packet is recirculated to the ingress pipeline, indicating the queue congestion status. In the ingress pipeline, the iRED algorithm (action to drop) turns on the drop flag by output port. The next packet for the particular output port with drop flag ON will be dropped, and the drop flag turns off. In this case, we use the concept of dropping future packets, already discussed in another work [19]. The future packets are the next packets that arrive in the switch after the drop flag be turned on. With iRED, the ingress pipeline may then prevent this packet from exacerbating the imminent queue buildup.

Algorithm 1 shows the iRED part running at the egress, responsible for calculating the probability of dropping (similar to the original RED algorithm), cloning and recirculating packets when the threshold is reached.

In our design, the cloned packets ($pktCloned$) are used to inform that future packets should be dropped in the ingress pipeline. However, there is no way to clone packets from egress directly to ingress in actual data plane architectures, including PNA [20] and PSA [21]. To overcome this barrier, we use one type of primitive clone (E2E, Egress-to-Egress) to make an identical copy of the packet. In this way, the original packet is sent to its destination without increasing delay, and the cloned packet goes back to the egress pipeline. When the cloned packets enter the egress pipeline, they are fast-forwarded to the ingress pipeline thanks to the primitive recirculation (RECIRC). The cloned packet carries the port number (additional 9 bits) which means congestion is happening in that port. In the ingress pipeline, we use a structure to map the congested output ports that should indicate to drop future packets going to them, using

Algorithm 1 DECISION TO DROP - EGRESS

Input: *pkt, maxLenQueue, N, enq_qdepth, registerProb, n*

1: $minThsld = \frac{maxLenQueue}{2} - N$
2: $maxThsld = minThsld \times 2$
3: **for** $each\ pkt$ **do**
4: **if** $pkt == pktCloned$ **then**
5: $recirculate(pkt)$
6: **end if**
7: $qAvg = oldQdepth * (1 - 2^{-n}) + enq\_qdepth * (2^{-n})$
8: **if** $(qAvg \geq minThsld)$ **and** $(qAvg \leq maxThsld)$ **then**
9: $dropProb = registerProb^2$
10: $registerProb = dropProb$
11: $randomVal = random(0, N)$
12: **if** $dropProb \geq randomVal$ **then** ▷ Flipping a coin
13: $clone(pkt)$
14: **end if**
15: **end if**
16: **if** $qAvg > maxThsld$ **then**
17: $clone(pkt)$
18: **end if**
19: **end for**

only 1 bit per port. In this sense, Algorithm 1 should receive the parameters as input:

- pkt: the current packet;
- $maxLenQueue$: max queue size;
- N: it is used to calculate the minimum threshold and random values for the dropping logic;
- enq_qdepth: the queue depth when the packet was queued[2];
- $registerProb$: register that keeps the probability of dropping;
- n: number used to calculate the average queue.

The $minThsld$ will always be half the maximum queue size minus N, and the $maxThsld$ is twice the $minThsld$ (lines 1-2), following the rule of thumb, setting $maxThsld$ to at least twice $minThsld$ [8]. Algorithm 1 works as follows, if the packet is a clone, it should be recirculated to the ingress pipeline (lines 4-6), that is, it indicates that the future packets should be dropped in the ingress for the specific output port, using only 9 additional bits per packet. For regular packets, the current queue depth is used to compute the queue average ($qAvg$), according to the Equation 1 (line 7) that is based on the Cisco RED equation [23]. In this part, to overcome the math constraints regarding the programmable hardware, we use n equals one[3] and multiplying all elements by ten. This way, we do not use float point operations to compute the average queue size. Furthermore, we need to multiply by ten our min-max thresholds to compare correctly.

$$\begin{aligned} qAvg &= oldQdepth * (1 - 2^{-n}) + enq\_qdepth * (2^{-n}) \\ &= oldQdepth * (0.5) + enq\_qdepth * (0.5) \\ &= oldQdepth * (5) + enq\_qdepth * (5) \end{aligned} \quad (1)$$

[2]This metadata was chosen because it has more influence on the QoS of the DASH service, according to [22].

[3]Once, $2^{-1} = 0.5$, gives the same weight to the old average and the current queue occupancy [23].

If the average queue size is below the $minThsld$, the packet is forwarded directly to the output port. However, if the value is between the min-max thresholds (line 8), the packet may be cloned; in other words, it's like flipping a coin. In this case, the clone process will happen if the value saved in the register ($registerProb$) is greater than or equal to the randomly generated number (lines 9-13). The $randomVal$ is a number between 0 and N, that is, if N is too large, the probability of dropping a packet will be lower. On the other hand, if N is small, the probability of dropping packets will be higher. Finally, if the queue size exceeds $maxThsld$, all packets will be cloned (lines 16-18).

Algorithm 2 ACTION TO DROP - INGRESS

Input: *pkt, pktRecirc*

1: **for** $each\ pkt$ **do**
2: **if** $pkt == pktRecirc$ **then**
3: $dropFlag[output\_port] = 1$ ▷ Flag to drop ON
4: $dropPktRecirc$
5: **end if**
6: $ip\_forward$
7: $dropPort = dropFlag[output\_port]$
8: **if** $dropPort == 1$ **then**
9: $dropPkt$ ▷ Packet dropped
10: $dropFlag[output\_port] = 0$ ▷ Flag to drop OFF
11: **end if**
12: **end for**

The action to drop a packet is performed in the ingress pipeline, as described in Algorithm 2. Initially, the algorithm checks if the incoming packet was recirculated from egress (line 2). We use a register with a length equal to the number of ports so that each port is mapped by an index, where each index is linked to each output port. If the packet was recirculated, we turn on the flag to drop, that is, set the value to one in the index register (line 3). After that, the recirculated packet role is finished, being discarded (line 4). The remaining part of Algorithm 2 is essentially to forward packets as usual (line 6), by defining the output port. However, at this moment, the algorithm checks if the output port has the flag to drop ON or OFF. If the flag is ON (value 1), the packet should be dropped (lines 8-9) and the register is reset. Note that only one packet is dropped at a time and future packets going to the same output port will be dropped only if a recirculated packet is received in the ingress pipeline, as an indicator of congestion.

IV. EVALUATION

To validate our design and evaluate our hypothesis, we design two types of experiments. The first one was conceived to verify the advantages and impact in terms of recirculation for dropping packets at the ingress using iRED versus dropping at the egress. The second evaluation analyzes the usage of iRED versus TD for a DASH CDN, in terms of cached video available in client's buffer and the number of frames per second played.

Both experiments were done in a virtual environment built on a physical server model Dell EMC PowerEdge R720 with 2 Intel Xeon processors® E5-2630 v2 2.60GHz, 6 cores per socket (24 vCPUs), 48GB RAM, 2TB HDD and Ubuntu 18.04.5

LTS. Virtualbox (6.1.28) was used as the hypervisor together with Vagrant (2.2.19) and Ansible (2.10.8) for infrastructure provisioning. All the artefacts are available for replication purposes in a repository[4].

TABLE I
VALUES OF N USED IN THE EXPERIMENTS.

Buffer size ref. value	N value	minThsld	maxThsld
64	4	28	56
128	8	56	112
256	16	112	224
512	32	224	448
1024	64	448	832

For each experiment, we evaluate if the buffer size in the routers has some influence on the performance results. In this context, we variate the reference values regarding the buffer size in terms of the number of packets, from 64 to 1024 packets. Because of this variation, we had to adjust the value of N[5] (Algorithm 1) for each buffer size according to Table I.

A. Experiment 1

In this first case, we want to verify if dropping packets in the ingress could give some advantage over dropping packets compared to the egress pipeline in terms of latency, throughput and their impact on resources usage. We designed the topology described in Fig. 5, in which host 1 (h1) sends synthetic traffic (packets have 1514 bytes of size) to host 2 (h2) through the router. We use iperf3 (TCP connection) and ping (ICMP packets) tools to collect performance statistics. Furthermore, telemetry instructions save the number of dropped packets across all algorithms, allowing us to calculate bandwidth usage in AQM egress solutions and recirculation cost in iRED.

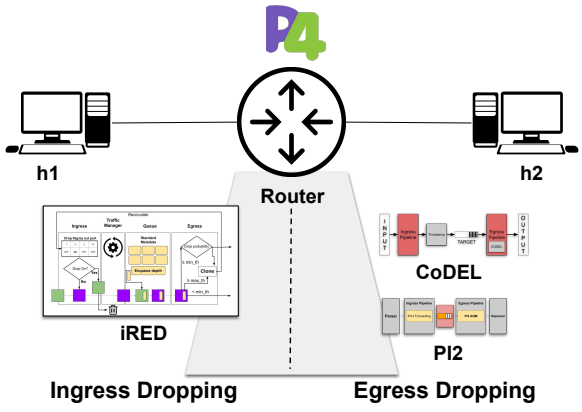


Fig. 5. Ingress vs egress dropping.

Each independent execution of this experiment lasted 60 seconds, with 5 repetitions for each algorithm. This experiment lasted approximately 35 minutes, divided into 3 parts, that is, 25 minutes for iRED (5 minutes for each buffer size reference value), 5 minutes for CoDEL/PI2 for P4. We set the buffer size high enough (10.000 packets) for all approaches to make sure that the only congestion control is performed by the AQM[6]. The results shown represent the mean of observations. The parameters used in CoDEL and PI2 are described in Table II.

TABLE II
AQM PARAMETERS.

Algorithm	Target delay	Control int.	PI int.	α	β
Codel	5 msec	100 msec	-	-	-
PI2	20 msec	-	33 msec	$0.3125Hz$	$3.125Hz$

Fig. 6 shows the average RTT for each AQM algorithm. iRED performed better than CoDEL and PI2 for smaller buffers reference values (64 and 128 packets) and statistically the same for larger buffers reference values. For a buffer size reference value of 64 packets (best case), iRED outperforms PI2 (worst case since its target delay is set to 20msec) by up to 2.48x.

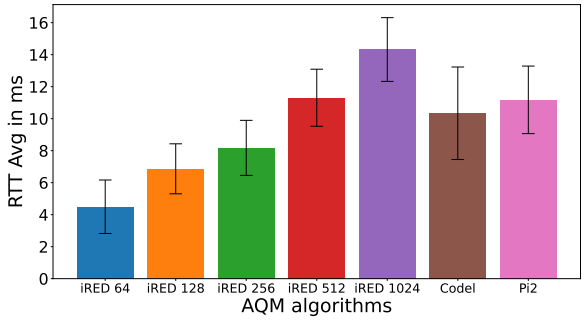


Fig. 6. Average RTT.

In Fig. 7, we can see the RTT rolling average with a window that includes the last 10 observations. This result indicates that smaller buffers reference values tend to have a lower average RTT. This information can help hardware manufacturers set the size of buffers routers for delay-sensitive applications.

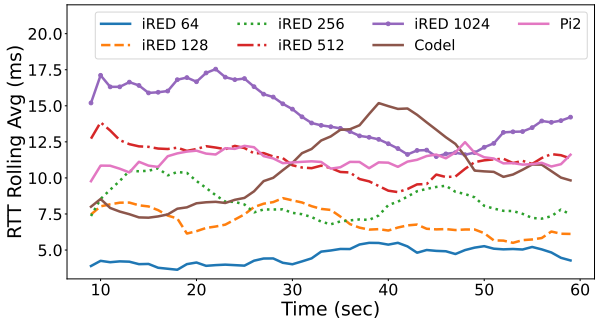


Fig. 7. Rolling average RTT.

Fig. 8 shows the CDF (Cumulative Distribution Function) for each algorithm in terms of throughput. In this case, the

[4]https://github.com/leandrocalmeida/.

[5]The N value is adjusted to bring the $minThsld$ to less than half the buffer size reference value, satisfying the rule of thumb [8].

[6]In other words, we have no tail drop discards.

algorithms that converge first (PI2 and CoDEL) presented a lower throughput. On the other hand, iRED converged later, indicating greater throughput. For a buffer size reference value of 64 packets (best case), iRED outperforms PI2 (worst case) by up to 0.74x.

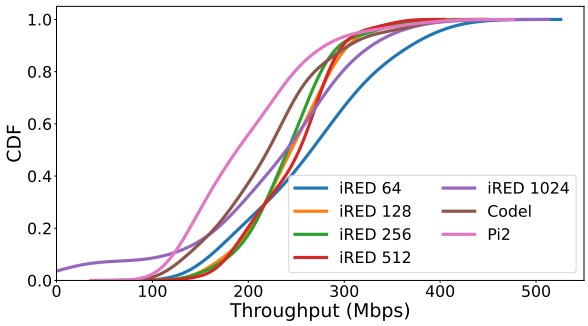


Fig. 8. CDF Throughput.

We evaluate the impact of dropping packets at the egress in terms of bandwidth usage. To do this, we use counters to save the number of dropped packets and calculate the drop rate, which gives us an indication of wasted resources.

TABLE III
EGRESS DROPPING RATE.

Egress algorithms	Dropping rate (%)
CoDEL-P4	0.005
PI2-P4	0.005

Table III shows that both, Codel and PI2, have the same dropping rate, 0.005%. This value, which may seem small, must be observed from the impact on the bandwidth wasted on state-of-art network equipment. Wasting happens because dropped packets go through all the switch pipelines, consuming unnecessary bandwidth. Considering that state-of-the-art network equipment currently can reaches rates of 25.6Tbps (64 x 400Gbps) [24], dropping on egress can consumes 1.31Gbps of total internal switch bandwidth.

We also evaluate the impact of iRED in terms of resources usage for recirculation. On programmable switches, a specific internal port is designated for the recirculation function [25]. Table IV shows how much bandwidth of this recirculation port, considering 400Gbps, is used based on the percentage of the recirculated packets collected in our experiment.

TABLE IV
PERCENTAGE OF RECIRCULATED PACKETS AND BANDWIDTH CONSUMPTION.

iRED evaluated	Recirculated packets (%)	Bandwidth consumed
iRED 64	0.038	152Mbps
iRED 128	0.022	88Mbps
iRED 256	0.016	64Mbps
iRED 512	0.005	20Mbps
iRED 1024	0.005	20Mbps

B. Experiment 2

This experiment is based on the fact that there is a correlation between the queue depth and the DASH QoS as shown in [22]. That is, when the queue occupancy increases in the buffers, the number of FPS decreases in the video player, causing QoS degradation. Moreover, we assume that when the queue is full, there is one (maybe several) culprit flows.

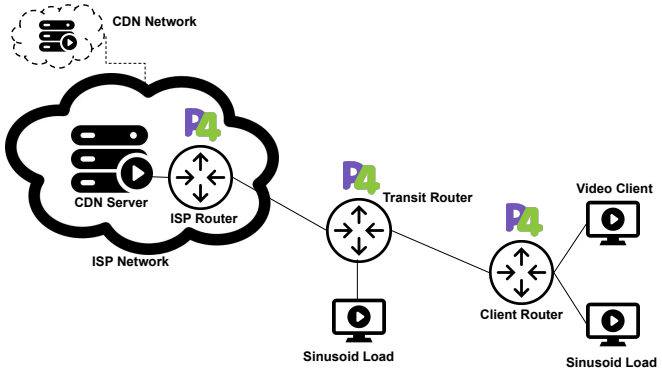


Fig. 9. Emulated programmable CDN.

When randomly dropping packets from the queue, there is a higher probability of dropping packets from the guilty flow. With these discards, the TCP congestion control algorithm must act, relieving the pressure on the buffers and consequently reducing the queue delay. This reduction should reflect in fewer rebuffering interrupts, thus improving the DASH QoS. Although not new, this is the hypothesis that we want to verify.

We created a realistic emulated programmable CDN for our assessments, as shown in Fig. 9. The CDN server is positioned inside of ISP (Internet Service Provider) network, offering the videos catalog from a DASH service. The ISP router connects the CDN server outside of ISP network, through the Transit Router. Attached to the Transit Router and to the Client Router there is a sinusoid load generator.

The video client and the load generators consume the video catalog available from the CDN server. Furthermore, to observe QoS in the video client, the video service metrics were collected at each millisecond by the video player[7]. Fluctuations in the network traffic, generated by the sinusoid load, cause bottlenecks in the routers' output interfaces.

Each independent execution of this experiment lasted 600 seconds, with 5 repetitions for each approach. This experiment lasted approximately 300 minutes, divided into 2 parts, that is, 250 minutes for iRED (50 minutes for each buffer size reference value) and 50 minutes for Tail Drop. The results represent the mean of observations. In Table V, the components are described. All their connections are provided by BMv2[8] switches, which are P4-capable virtual equipments.

The CDN server was responsible for providing a video catalog in the DASH standard for the client and the load

[7]https://github.com/Dash-Industry-Forum/dash.js.
[8]https://github.com/p4lang/behavioral-model.

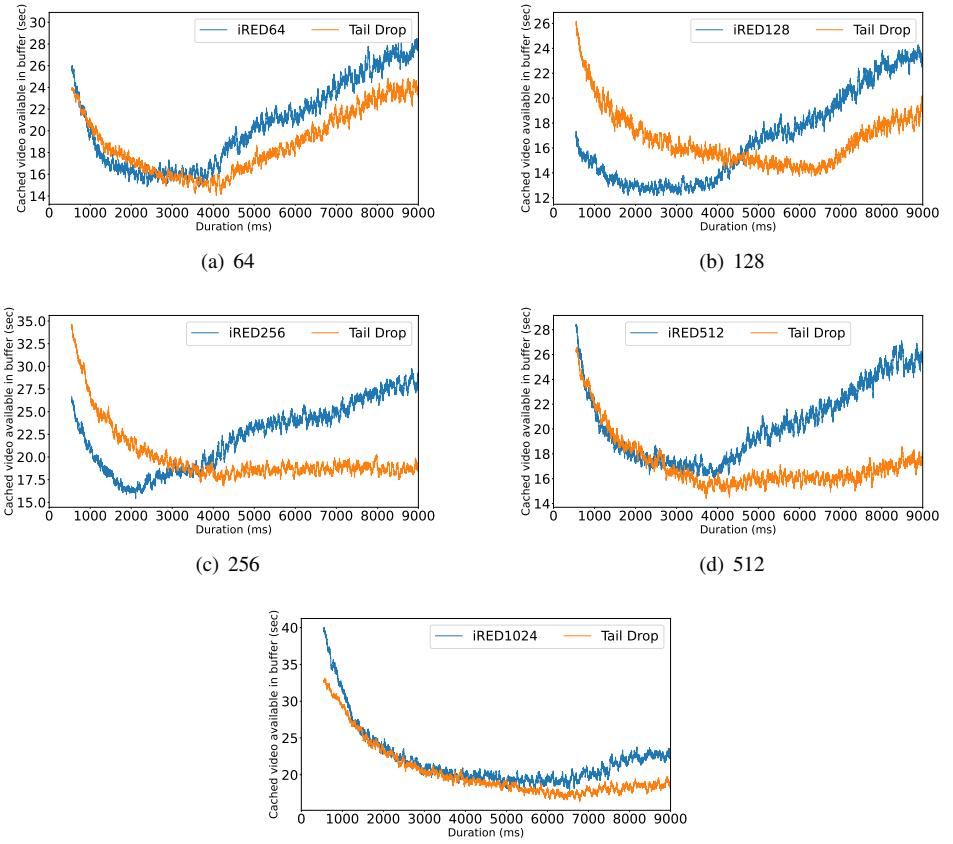


Fig. 10. Occupancy of the video player buffer in seconds.

TABLE V
VMs DETAILS.

Name	OS	vCPUs	Memory
CDN Server	Ubuntu 20.04.1 LTS	1	512MB
Video Client	Ubuntu 20.04.1 LTS	4	4GB
Sinusoid Load 1	Ubuntu 20.04.1 LTS	12	16GB
Sinusoid Load 2	Ubuntu 20.04.1 LTS	12	16GB
ISP Router	Ubuntu 20.04.1 LTS	4	4GB
Transit Router	Ubuntu 20.04.1 LTS	4	4GB
Client Router	Ubuntu 20.04.1 LTS	4	4GB

generators. Two video streams were made available, one being a transmission of a soccer game for the video client access; and a playlist containing the ten most accessed videos on Youtube® for the load generators. The Webserver Apache version 2 was installed for hosting videos; FFmpeg (2.8.17) was used for encoding the videos; and MP4box (0.5.2) for creating the MPEG-DASH manifest files.

The video client, by means of DASH.js video player, consumes the video streaming of the soccer game, using the TCP New Reno as the congestion control algorithm.

The load generators [26] make service requests to the CDN server by dynamically adjusting the number of active sessions, spawning, and terminating video clients. The load generator produces requests following a Poisson process whose arrival rate is modulated by a sinusoidal function, described in Equation 2, where: A represents an amplitude; F the frequency; and λ is a phase in radians.

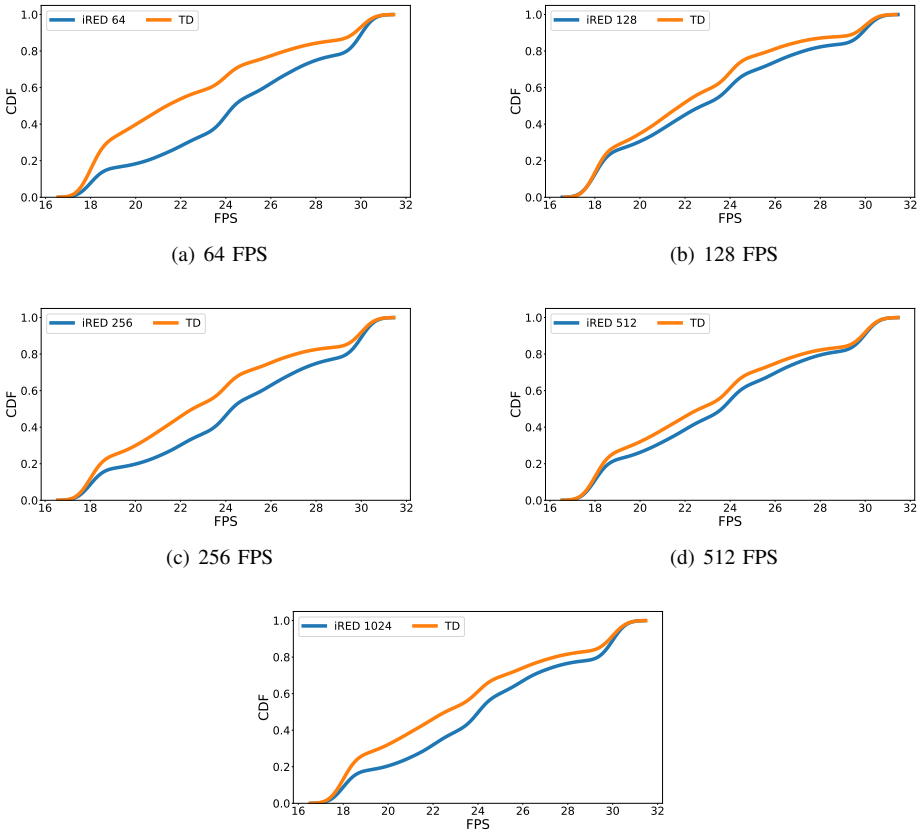
$$f(y) = A\sin(F + \lambda) \tag{2}$$

The CDN server hosts the video catalog with different quality levels, as shown in Table VI. The video client can choose from each of the different quality levels according to the traffic load on the network. That is, when the load on the network is high, the client plays the video in low resolution. However, when the load on the network decreased, the client choses the video at its highest resolution. This fluctuating behavior in traffic obeyed the sinusoid load generators.

TABLE VI
VIDEO PARAMETERS USED IN A CDN SERVER.

Type	Resolution	FPS	GOP[9]	Kbps	Buffer	Codec
vídeo	426x240	18	72	280	140	h264
vídeo	854x480	24	96	980	490	h264
vídeo	1280x720	30	120	2080	1040	h264
áudio	-	-	-	128	-	AAC
áudio	-	-	-	64	-	AAC

[9]Group of Pictures.



(a) 64 FPS

(b) 128 FPS

(c) 256 FPS

(d) 512 FPS

(e) 1024 FPS

Fig. 11. Frames per second (FPS) CDF.

The parameters used by the load generators were: $A = 10$; $F = 2$; $\lambda = 25$. With these parameters, the load fluctuation with the number of video clients over time generated by the sinusoidal load varied between 15 (minimum) and 35 (maximum) over time.

Since in experiment 1 we evaluated the AQM strategies between them (iRED, Codel, and PI2), in this second experiment we decided to evaluate iRED as an AQM strategy versus the Tail Drop (TD) approach. Here we want to assess whether using an AQM strategy based on packet drop policy improves the QoS of a DASH video service. TD is the default mode of the routers, without any AQM policy applied to congestion control. Based on this context, we evaluate the cached video available in client's buffer (in seconds), for each buffer size reference value on the routers. Video clients use this cache to not interrupt the video playback in bottleneck conditions.

In Fig. 10, the abscissa axis represents the cached video in the client's buffer in seconds, and the ordinate axis represents the duration of experiment in microseconds. In this case, a bigger cache (frames to be played) is enhancing the QoS [7]. In all subfigures of Fig. 10 we can to observe that iRED algorithm (blue lines) maximizes the level of occupancy of cache buffer as video is played. This cache is an important component for streaming video in congested conditions [7], as the video player uses locally stored frames while congestion control is helped by the probabilistic packet drop of iRED. For a buffer size of 256 packets (best case), iRED outperforms TD (worst case) by up to 0.85x. However, as in TD (orange lines) the discard is always by the tail, TCP global synchronization happens, equitably punishing all flows that cross the network. In these situations, the video player will have little time (video frames) for transmission in the cache buffer under bottleneck conditions, decreasing the QoS.

Another way to evaluate the video quality is to look at the number of FPS displayed on the video client screen [7]. In our configuration, the DASH server offered three levels of quality for the same video stream so that the client could choose according to the load on the network. In this case, the video client could play the video in 18, 24, or 30 FPS, self-adjusting according to the feedback provided by the infrastructure status.

Fig. 11 shows the CDF for iRED (blue lines) and TD (orange lines) in terms of frames per second played on the video clients. In this case, the approach that converges first played the video at a lowest resolution (low fps). In a broader perspective, for all analyzed buffer sizes, iRED showed signs of maximizing the FPS presented in the video client, that is, good evidence that packet dropping for a video service improve the DASH QoS. On the other hand, in the TD approach, the video client played

the video in a minor resolution most of the time of transmission. For a buffer size of 256 packets (best case), iRED outperforms TD (worst case) by up to 0.91x.

Overall, the experiments showed evidence that DASH on a TD approach network can lower QoS for a video clients. Although DASH is considered smart and adjusts resolutions on demand, the video player is unable to maximize the local buffer occupancy. However, iRED proved to be useful in bottleneck conditions, aiding TCP's congestion control mechanism through the probabilistic drops.

V. Conclusions

Assessing the QoS of video streaming is a way for the provider to know if the service is being delivered at a level of quality expected by consumers or not. In this context, we designed and implemented iRED, an AQM inspired by the well-know RED, to run in programmable data planes. The evaluation of iRED shows that dropping packets at the ingress minimizes the latency and resources consumed and maximizes the throughput when compared with state-of-the-art works.

We also created a realistic emulated programmable CDN, in which a DASH service was evaluated on a network with programmable nodes in the data plane. The results indicate that iRED improves the local buffer level of the video player and the number of frames per second played. These findings show that, under certain circumstances, QoS is positively impacted by packet dropping.

Next steps include the evaluation of iRED having a mix of different application flows in the network so that we can observe the impact of dropping packets in different types of traffic. Furthermore, adjust iRED to identify the culprit flow to dropping packets only of this flow.

References

[1] Cisco, "Global forecast highlights," Cisco Systems, Tech. Rep., 2021.

[2] S. Schaevitz, "Three months, 30x demand: How we scaled google meet during covid-19," https://cloud.google.com/blog/products/g-suite/keeping-google-meet-ahead-of-usage-demand-during-covid-19, 2020, accessed: 2021-02-01.

[3] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, and R. Zimmermann, "A survey on bitrate adaptation schemes for streaming media over http," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 562–585, 2019.

[4] ISO, "Dynamic adaptive streaming over http (dash)-part 1: Media presentation description and segment formats," *ISO/IEC*, pp. 23 009–1, 2014.

[5] S. Lederer, "Why youtube & netflix use mpeg-dash in html5," https://bitmovin.com/status-mpeg-dash-today-youtube-netflix-use-html5-beyond/, 2015, accessed: 2020-03-24.

[6] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, and R. Zimmermann, "A survey on bitrate adaptation schemes for streaming media over http," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 562–585, 2019.

[7] Y. Chen, K. Wu, and Q. Zhang, "From qos to qoe: A tutorial on video quality assessment," *IEEE Communications Surveys Tutorials*, vol. 17, no. 2, pp. 1126–1165, 2015.

[8] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[10] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, and R. Steinmetz, "P4-codel: Active queue management in programmable data planes," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2018, pp. 1–4.

[11] C. Papagianni and K. De Schepper, "Pi2 for p4: An active queue management scheme for programmable data planes," in *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '19 Companion. New York, NY, USA: Association for Computing Machinery, 2019, p. 84–86. [Online]. Available: https://doi.org/10.1145/3360468.3368189

[12] B. Spang, B. Walsh, T.-Y. Huang, T. Rusnock, J. Lawrence, and N. McKeown, "Buffer sizing and video qoe measurements at netflix," in *Proceedings of the 2019 Workshop on Buffer Sizing*, ser. BS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3375235.3375241

[13] T. De Pessemier, K. De Moor, W. Joseph, L. De Marez, and L. Martens, "Quantifying the influence of rebuffering interruptions on the user's quality of experience during mobile video watching," *IEEE Transactions on Broadcasting*, vol. 59, no. 1, pp. 47–61, 2013.

[14] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The blue active queue management algorithms," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, p. 513–528, aug 2002. [Online]. Available: https://doi.org/10.1109/TNET.2002.801399

[15] K. Nichols and V. Jacobson, "Controlling queue delay: A modern aqm is just one piece of the solution to bufferbloat." *Queue*, vol. 10, no. 5, p. 20–34, may 2012. [Online]. Available: https://doi.org/10.1145/2208917.2209336

[16] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of cake: A comprehensive queue management solution for home gateways," in *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2018, pp. 37–42.

[17] K. De Schepper, O. Bondarenko, I.-J. Tsang, and B. Briscoe, "Pi¡sup¿2¡/sup¿: A linearized aqm for both classic and scalable tcp," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 105–119. [Online]. Available: https://doi.org/10.1145/2999572.2999578

[18] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Commun. ACM*, vol. 55, no. 1, p. 57–65, jan 2012. [Online]. Available: https://doi.org/10.1145/2063176.2063196

[19] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-grained queue measurement in the data plane," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–29. [Online]. Available: https://doi.org/10.1145/3359989.3365408

[20] T. P. L. Consortium, "P4 portable nic architecture (pna)," P4 Consortium, Tech. Rep., 2018.

[21] T. P. A. W. Group, "P416 portable switch architecture (psa)," P4 Consortium, Tech. Rep., 2018.

[22] L. C. de Almeida, R. Pasquini, and F. L. Verdi, "Using machine learning and in-band network telemetry for service metrics estimation," in *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, 2021, pp. 33–39.

[23] C. Systems, *QoS: Congestion Avoidance Configuration Guide, Cisco IOS Release 15MT*. Americas Headquarters, 2013.

[24] B. Wheeler, "Tomahawk 4 switch first to 25.6tbps," Broadcom, Tech. Rep., 2019.

[25] Z. Xi, Y. Zhou, D. Zhang, J. Wang, S. Chen, Y. Wang, X. Li, H. Wang, and J. Wu, "Hypergen: High-performance flexible packet generator using programmable switching asic," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 42–44. [Online]. Available: https://doi.org/10.1145/3342280.3342301

[26] R. Stadler, R. Pasquini, and V. Fodor, "Learning from network device statistics," *J. Netw. Syst. Manag.*, vol. 25, no. 4, pp. 672–698, 2017. [Online]. Available: https://doi.org/10.1007/s10922-017-9426-z