Adaptive Chunklets and AQM for Higher Performance Content Streaming

JONATHAN KUA, Swinburne University of Technology, Australia
GRENVILLE ARMITAGE, Netflix Inc, USA and Swinburne University of Technology, Australia
PHILIP BRANCH and JASON BUT, Swinburne University of Technology, Australia

Commercial streaming services such as Netflix and YouTube use proprietary HTTP-based adaptive streaming (HAS) techniques to deliver content to consumers worldwide. MPEG recently developed Dynamic Adaptive Streaming over HTTP (DASH) as a unifying standard for HAS-based streaming. In DASH systems, streaming clients employ adaptive bitrate (ABR) algorithms to maximise user Quality of Experience (QoE) under variable network conditions. In a typical Internet-enabled home, video streams have to compete with diverse application flows for the last-mile Internet Service Provider (ISP) bottleneck capacity. Under such circumstances, ABR algorithms will only act upon the fraction of the network capacity that is available, leading to possible QoE degradation. We have previously explored *chunklets* as an approach orthogonal to ABR algorithms which uses parallel connections for intra-video chunk retrieval [23]. Chunklets effectively make more bandwidth available for ABR algorithms in the presence of cross-traffic, especially in environments where Active Queue Management (AQM) schemes such as Proportional Integral controller Enhanced (PIE) and FlowQueue-Controlled Delay (FQ-CoDel) are deployed. However, chunklets consume valuable server/middlebox resources which typically handle hundreds of thousands requests/connections per-second. In this paper, we propose *'adaptive chunklets'* – a novel chunklet enhancement that dynamically tunes the number of concurrent connections. We demonstrate that the combination of adaptive chunklets and FQ-CoDel is the most effective strategy. Our experiments show that adaptive chunklets can reduce the number of connections by almost 30% and consume almost 8% less bandwidth than fixed chunklets while providing the same QoE.

CCS Concepts: • **Information systems** → **Multimedia streaming**; • **Networks** → **Transport protocols**; **Application layer protocols**; **Network performance analysis**; **Network measurement**; *Home networks*;

Additional Key Words and Phrases: DASH, HTTP, TCP, FIFO, AQM, PIE, CoDel, FQ-CoDel, QoE, content streaming, chunklets, adaptive bitrate algorithm, queue management, latency, packet loss

ACM Reference Format:
Jonathan Kua, Grenville Armitage, Philip Branch, and Jason But. 2019. Adaptive Chunklets and AQM for Higher Performance Content Streaming. *ACM Trans. Multimedia Comput. Commun. Appl.* 1, 1, Article 1 (July 2019), 23 pages. https://doi.org/10.1145/3344381

1 INTRODUCTION

In recent years, we have seen a significant shift in consumers' choice of entertainment, from watching broadcast television to streaming multimedia content online. Video traffic currently

This work was enabled in part by PhD stipend support from Netflix, Inc and an Australian Government Research Training Program Scholarship scheme.
Authors' addresses: Jonathan Kua, jtkua@swin.edu.au, Swinburne University of Technology, John Street, Hawthorn, VIC, 3122, Australia; Grenville Armitage, Netflix Inc, 100 Winchester Cir, Los Gatos, CA, 95032, USA, Swinburne University of Technology, John Street, Hawthorn, VIC, 3122, Australia, garmitage@swin.edu.au; Philip Branch, pbranch@swin.edu.au; Jason But, jbut@swin.edu.au, Swinburne University of Technology, John Street, Hawthorn, VIC, 3122, Australia.

1551-6857/2019/7-ART1
https://doi.org/10.1145/3344381

accounts for almost 58% of worldwide downstream traffic[1], representing a significant portion of inbound Internet traffic to the home environment. This figure is expected to reach 82% by 2021[2]. The demand for a better Quality of Experience (QoE) has also concurrently increased, with many studies showing that users will quickly abandon video sessions if the startup delay is high, quality is not sufficient or if the video rebuffers/stalls frequently throughout the session, leading to revenue losses for content providers [22].

Modern streaming companies such as Netflix and YouTube use proprietary HTTP-based adaptive streaming (HAS) techniques for delivering content to subscribers and users worldwide. The Moving Pictures Experts Group (MPEG) recently developed Dynamic Adaptive Streaming over HTTP (DASH) as a unifying standard for live and on-demand HAS-based video streaming services [19]. DASH systems aim to deliver an uninterrupted user experience by employing adaptive bitrate (ABR) algorithms in streaming clients. ABR algorithms dynamically adapt the requested video quality on-the-fly to match the network capacity based on feedback signals such as estimated throughput and/or playout buffer occupancy. Much research efforts have been invested into developing ABR algorithms, ranging from simple throughput and buffer-based techniques to recent complex control-theory, machine learning and neural network approaches. All of these approaches aim to deliver the best QoE given a certain amount of network capacity.

In a typical Internet-enabled home, video streams have to compete with diverse application flows for the last-mile Internet Service Provider (ISP) bottleneck link or home gateway that connects home users to the Internet. Many of these applications rely on the Transmission Control Protocol (TCP) to ensure reliable transport of data and effective use of available bandwidth. However, TCP is known to fill bottleneck buffers until packet losses occur, causing cyclical queue filling and draining and inflation of end-to-end Round Trip Time (RTT) delays. The use of oversized and unmanaged bottleneck buffers such as conventional first-in-first-out (FIFO) buffer management leads to the well-known *bufferbloat* phenomenon [13]. When competing with long-lived TCP flows (such as file transfers), video streams suffer from bufferbloat effects due to their bursty nature and limited share of bandwidth [16]. Under such circumstances, the best of ABR algorithms will only have a fraction of the bottleneck capacity to act on.

Recently, the Internet Engineering Task Force (IETF) has developed new Active Queue Management (AQM) schemes to mitigate the effects of bufferbloat by allowing short-term/transient bursts in traffic while maintaining low long-term queuing delays. Three prominent examples are Proportional Integral controller Enhanced (PIE) [34], Controlled Delay (CoDel) [33] and FlowQueue-CoDel (FQ-CoDel) [14]. A variant of PIE has been integrated into DOCSIS 3.1 [40] and FQ-CoDel has been highly recommended for embedded Linux gateways. FQ-CoDel was made widely available as part of OpenWRT's Smart Queue Management (SQM) scheme in 2014. It is now the default queue discipline on most Linux distributions and is readily available in commercial products such as pfsense[3] and OPNsense[4]. The low latencies presented by AQMs benefit latency-sensitive flows such as Voice over IP (VoIP) calls, interactive online games and live video streaming. With live streaming, content is made available at the server during viewing and the system should achieve a low glass-to-glass latency (delay between camera capture and content display) for a better QoE. AQMs can lower glass-to-glass latency by directly controlling queuing delays, which is a latency component that can otherwise be significant with FIFO.

[1] https://www.sandvine.com/press-releases/sandvine-releases-2018-global-internet-phenomena-report
[2] https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf
[3] https://www.pfsense.org/
[4] https://opnsense.org/

FQ-CoDel offers an attractive solution with its ability to isolate flows and share capacity evenly. It isolates individual traffic flows into sub-queues then serves each sub-queue with a modified Deficit Round Robin (DRR) scheduler. The result is a relatively even capacity sharing, which is desirable in most cases, but can actually be detrimental to a DASH video flow (which often uses a single and persistent TCP connection) that is competing with multiple other concurrent traffic flows. The DASH flow is limited to a certain fraction of the available bandwidth as a consequence of FlowQueue scheduling, and if the 'allocated' capacity is lower than the available video bitrates, the DASH client's ABR algorithm will select sub-optimal video bitrates.

In [23] we proposed and implemented *chunklets* – an approach orthogonal to ABR algorithms which uses multiple concurrent TCP connections to retrieve different parts of a video chunk (intra-chunk) in order to gain a larger share of the bandwidth in the presence of other cross-traffic competing in the same direction (from the Internet into the home). We demonstrated that chunklets are advantageous when competing with multiple long-lived flows, thus improving user experience. Intuitively, we need to increase the number of chunklets as the number of competing flows increases. Although a large number of chunklets will always guarantee the best possible video bitrates, they present other challenges. Firstly, chunklets (with their open TCP connections) consume valuable server/middlebox resources which typically handle hundreds of thousands of requests/connections per-second. The resources needed will increase dramatically as the number of chunklet-enabled clients increases. Secondly, a large number of chunklets will unnecessarily starve other traffic flows that concurrently share the same bottleneck. In this work, we address these challenges by proposing '*adaptive chunklets*' – a novel algorithm that dynamically tunes the number of concurrent TCP connections to achieve the best possible user experience while ensuring clients do not initiate more concurrent connections than necessary. The primary focus of our work is *on-demand* video streaming, where videos are pre-encoded and stored at the servers. Further work is required to best identify how our approach might benefit live streaming applications.

In this paper, we substantially extend [23] with the following key contributions:

- Experimentally evaluate and characterise chunklets under a broad set of network settings and their impact on competing flows (Section 4).
- Propose and design a novel '*adaptive chunklets*' algorithm that dynamically tunes the number of concurrent TCP connections (Section 5).
- Discuss deployment considerations and identify potential future work (Section 6).

We then offer concluding remarks in Section 7. In the next section, we start by providing background information and related work (Section 2), followed by a description of our use case model and evaluation methodology (Section 3).

2 BACKGROUND

In this section, we provide an overview of DASH systems and QoE, summarise modern AQM schemes, describe how chunklets can improve streaming experience and discuss related work.

2.1 Dynamic Adaptive Streaming over HTTP (DASH)

Modern Internet video streaming platforms employ adaptive streaming techniques based on the MPEG-DASH specification [19]. In DASH systems [39], video content is encoded into multiple versions at different discrete bitrates/qualities, or *Representation Rates (RR)*. Each encoded video is then segmented into small video segments or chunks, each containing a few seconds of video. Chunks from one bitrate are aligned in the video timeline to chunks from other bitrates so that, if necessary, the client can smoothly switch bitrates at the chunk boundary. The server provides a corresponding Media Presentation Description (MPD) file which describes the information of the

available content and the associated encoding bitrates or RR. Video content and MPDs are served by standard HTTP servers.

DASH is layered on top of HTTP/TCP, hence it does not control the content transmission rate directly. Instead it relies on the underlying TCP algorithm to regulate the content transmission rate, which is determined by congestion feedback from the client-server network path. To begin a streaming session, the client requests an MPD file from the content server and then starts requesting video chunks as quickly as possible to fill the playout buffer (typically back-to-back requests in sequence using one persistent TCP connection). Once the playout buffer is full, the player enters an ON-OFF steady-state phase of periodically downloading new chunks as previous chunks are consumed and rendered as audio/video content.

In steady-state, the ON period represents the DASH client downloading a chunk, and the OFF period represents otherwise. The time between the start of two consecutive ON periods is typically the chunk size – amount of video content within each chunk – in seconds (*aka* cycle time). The client typically keeps tens to hundreds of seconds worth of video in the playout buffer (depending on its memory capacity) to maintain adequate playback.

TCP uses a congestion window (*cwnd*) to estimate the number of bytes that can be 'in flight' and unacknowledged at any given time. *cwnd* starts low (2 packets by default, with some recent OSes starting with a *cwnd* value of 10 packets), grows as packets are received and acknowledged by the client, and shrinks when packets are lost or if the connection has gone idle for too long [10]. A DASH client's chunk retrieval process means TCP sends repeated bursts of packets followed by some periods of inactivity. If *cwnd* is too low, or fails to grow quickly enough, the DASH client experiences low per-chunk *Achieved Rate (AR)*. AR is an estimate of per-chunk TCP throughput (video chunk size divided by the time taken to receive it). If *cwnd* grows beyond the path's bandwidth-delay product (BDP), it starts filling bottleneck queues and inflicting additional queuing delays on all traffic flows sharing the bottleneck.

DASH clients use an ABR algorithm to adapt to fluctuating network conditions. ABR algorithms use various feedback signals observed for each chunk (*e.g.* recent AR estimates and/or playout buffer occupancy) to select a suitable RR for the next chunk to be downloaded. Consider a simple illustration when a DASH client uses AR as feedback signal. If AR is high, ABR should select a higher RR. On the other hand, if AR decreases, ABR should dynamically switch to a lower RR level to avoid playout buffer under-run. A good ABR algorithm will strike a delicate balance between reacting to network conditions and adapting video RRs smoothly. In practice, there are numerous implementations of ABR algorithms that aim to optimise QoE with advanced techniques. ABR algorithms have undergone extensive research [25] and we briefly describe them in Section 2.5.1.

2.2 Quality of Experience (QoE)

In Internet data communications, network parameters such as packet losses, latency and jitter are commonly used to measure application performance, or the Quality of Service (QoS). However, such objective QoS metrics do not directly translate to user experience, prompting the introduction of Quality of Experience (QoE) as a more subjective metric for measuring user perception [11].

In HAS systems, factors that influence QoE can be classified into technical and perceptual factors [35]. Technical factors involve the underlying technology that drive user perception, such as video encoding schemes, chunk sizes, ABR logic, the interaction between multiple streaming clients or with other applications. On the other hand, perceptual factors are directly perceived by the end-user. They can be broadly categorised into waiting times (*e.g.* video startup delay, rebuffer frequency, rebuffering duration), video adaptation (*e.g.* RR switching frequency and amplitude) and video quality. Some studies correlate these metrics with user Mean Opinion Score (MOS) and proposed various QoE models [30, 32]. An excellent survey on QoE metrics can be found in [35].

Understanding and predicting user QoE for adaptive streaming is currently an active research area. Recently, a *state-of-the-art* video quality assessment metric, Video Multimethod Assessment Fusion (VMAF) was introduced [7]. VMAF predicts subjective quality by combining multiple elementary quality metrics (each with its own strengths and weaknesses). A machine learning model trained with extensive real-world experiments is then used to produce a final score that preserves the strengths of individual metrics. Research showed that VMAF is a more effective metric than traditional Peak Signal to Noise Ratio (PSNR) or Structural Similarity (SSIM) measurements.

2.3 Modern Active Queue Management (AQM)

Unlike earlier AQM schemes, PIE, CoDel and FQ-CoDel are modern approaches that aim to keep long-term queuing delays low instead of merely controlling the queue occupancy. PIE and CoDel operate on single queues and keep queuing delays low by dropping packets when queuing delays persistently exceed a target delay, T_{target} .

PIE [34] introduces a burst tolerance parameter which allows packets arriving within the first 150ms of an empty queue to pass successfully. After this, when a packet arrives, it is randomly dropped with a certain probability. This probability is periodically updated, based on how much the current queuing delay (estimated from the queue length and the dequeue rate) differs from $T_{target} = 15ms$ and whether the queuing delay is currently increasing or decreasing. Packets of Explicit Congestion Notification (ECN)-enabled flows will be marked instead of being dropped when the dropping probability is <10%. RFC 8033[34] also stipulated a probability de-randomisation and auto-tuning mechanisms to avoid consecutive packet drops.

CoDel [33] tracks the (local) minimum queuing delay experienced by packets in a certain interval (initially 100ms). When the minimum queuing delay is less than $T_{target} = 5ms$ or the buffer size is less than one full-size packet, packets are neither dropped nor ECN marked. When the minimum queuing delay exceeds T_{target}, CoDel enters the drop state where a packet is dropped and the next drop time is set. The next drop time decreases in inverse proportion to the square root of the number of drops since the dropping state was entered. When the minimum queuing delay is below T_{target} again, CoDel exits the drop state.

FQ-CoDel [14] classifies flows into one of 1024 (by default) different queues by hashing the 5-tuple of IP protocol number, source and destination IP and port numbers with the Jenkins hash function. Each queue is separately managed by the CoDel algorithm. A modified Deficit Round Robin (DRR) scheduler services these queues, in which each queue can dequeue up to a quantum of bytes (one MTU by default) per iteration. In FQ-CoDel, this scheme gives priority to queues with packets from new flows or from 'sparse' flows with packet arrival rate small enough so that a new queue is assigned to them upon packet arrival (*e.g.* DNS, VoIP, online games). A bottleneck managed by FQ-CoDel can achieve low latency (due to per-queue CoDel), relatively even capacity sharing (due to the fixed hashing function) and priority for low-rate or transactional traffic.

2.4 Chunklets

We first proposed using chunklets in AQM environments in [23]. "Chunklets" is entirely a client-side technique we use to sub-divide the HTTP request-range for DASH video chunks to smaller sub-ranges (representing sub-chunks) and concurrently retrieve them with separate persistent TCP connections. There are no modifications associated with the server-side or video content preparation process. We named each sub-chunk a "*chunklet*" and all chunklets making up a chunk "*a cluster of chunklets*". We coin the artificial verb 'chunkleting' to define the process of sub-dividing a video chunk to multiple chunklets. Our motivation stems from combining FQ-CoDel's flow isolation properties at the dominant bottleneck and the goals of DASH ABR algorithms. Since the FlowQueue scheduler shares bandwidth evenly across all flows, having multiple flows belonging to

a single video chunk will effectively provide a "larger perceived bandwidth pipe", resulting in *higher per-chunk throughput* (AR), which then influences the client's ABR algorithm to select higher RRs.

At the start of an on-demand video streaming session, a chunklet-enabled DASH client will obtain all the chunk size information from the MPD[5] (typically in terms of byte-ranges for each chunk). Using the byte-range information, the client will then construct N consecutive sub-ranges of bytes to represent N chunklets, and send N concurrent HTTP GET requests (one for each chunklet) over N parallel persistent TCP connections to the content server. Chunklets are transparent to the server, it will serve each chunklet request like any regular HTTP GET request. As HTTP responses arrive, the client will reassemble chunklets into a full chunk as originally described in the MPD, regardless of their arrival order. All chunklets need be completely retrieved before reassembling them into a fully decodable chunk. The entire chunklet retrieval and reassembly process is hidden from the ABR layer. There are no ABR changes required as the ABR algorithm will still perceive the retrieval of "chunks" instead of "chunklets" and subsequently make decisions based on the feedback signals (*e.g.* throughput) experienced by "chunks".

2.5 Related work

2.5.1 Adaptive bitrate algorithms. Adaptive bitrate (ABR) algorithms have been studied extensively in the literature. They can be broadly categorised into three classes: throughput/rate-based, buffer-based and hybrid algorithms. Throughput-based algorithms estimate the available network bandwidth using past chunk downloads and select the highest bitrate sustainable by the network. Buffer-based approaches such as [18] and [37] use the client's playout buffer occupancy as a feedback signal for selecting video bitrates for future chunks. They aim to keep the playout buffer occupancy at a certain level to ensure a minimum number of rebuffering events. Buffer-based approaches are more conservative than throughput-based approaches. Throughput-based approaches perform best when network bandwidths are stable, while buffer-based approaches are more robust in time-varying networks [36]. Recent algorithms are 'hybrid', they use sophisticated mathematical approaches (*e.g.* neural networks, control theory, game theory) to optimise QoE metrics based on both throughput and buffer signals [8, 17, 31, 41]. These approaches yield better results compared to pure throughput/buffer-based approaches, but they are computationally more expensive. [25] and [9] provide good surveys and taxonomies of *state-of-the-art* ABR algorithms in the field.

Our chunklet approach underlies the ABR adaptation layer with an orthogonal goal – making more bandwidth available for ABR algorithms to act upon. Since all ABR algorithms use throughput signals directly or indirectly, chunklets further enhance the user experience by enabling the DASH client to obtain a larger share of the bottleneck bandwidth when competing with cross-traffic.

2.5.2 Parallel TCP connections. The idea of using parallel connections to accelerate file downloads is not new. MulTCP [12] mimics the behaviour an aggregate of N standard TCP connections. During steady-state, its *cwnd* growth factor is N times higher than a standard TCP ($\frac{N}{cwnd}$) and it reduces *cwnd* by ($\frac{N-0.5}{N}$) when congestion is detected. Other examples include Probe-Aided MulTCP, Stochastic TCP, MPAT, MulTFRC and Ensemble-TCP. Applications such as GridFTP and modern Web browsers use multiple parallel connections to accelerate file transfers. In video streaming, multiple connections are generally used to accelerate the download of different video chunks (inter-chunk/full chunk concurrency) [27, 29]. dash.js prior to version 1.4 included a prototype for downloading different video chunks in parallel[6].

[5]There are several ways to provide a DASH client with chunk size information. For example, the MPD may point the DASH client to the initialisation segment for each RR, which then contains all the byte-range information for the whole video.
[6]https://github.com/Dash-Industry-Forum/dash.js/issues/1029

We use parallel connections differently in that we divide individual range-based chunk requests into a set of intra-chunk range requests and send them in parallel over long-lived connections to the content server. The underlying idea has previously appeared in [4] (applied in wireless environments, without giving it a particular name), and has been discussed among dash.js developers and several companies but not yet implemented.

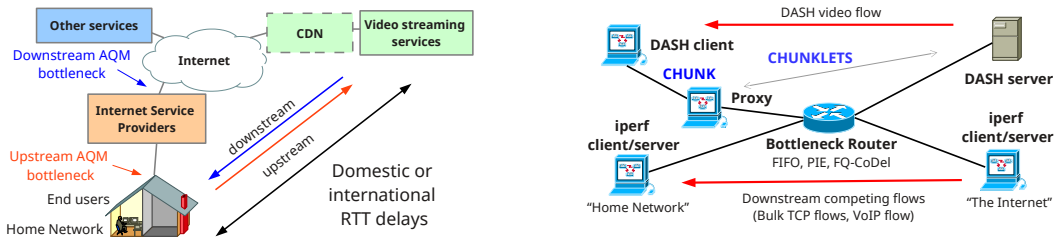
2.5.3 Active Queue Management for home networks. Prior studies highlight the importance of having good queue management schemes in consumer home environments [38]. Evaluations of modern AQMs are beginning to emerge as PIE, CoDel and FQ-CoDel have recently being standardised by IETF. [5] provides experimental evidence that latency-sensitive traffic benefits from FlowQueue-based AQMs in terms of latency and packet losses. [6] provides in-depth social and technical discussions on how the deployment of AQMs can lower household bandwidth requirements. However, evaluations of DASH-based streaming over modern AQM schemes are only beginning to emerge. A recent experimental study showed that (standard, single-connection) DASH video streams benefit from PIE's higher burst tolerance and queuing delay targets when there is no cross-traffic. In the presence of bidirectional cross-traffic, FlowQueue-based AQMs provide flow isolation in both directions, protecting the client's downstream video packets and upstream TCP acknowledgement (ACK) packets from the detrimental effects of cross-traffic competition [24].

The current paper substantially extends our chunklet work in [23] with a novel adaptive chunklets technique and a significantly improved experimental evaluation and characterisation of chunklets.

3 EVALUATION METHODOLOGY

In this section, we describe the technical choices made to experimentally explore chunklets. We start by introducing our experiment testbed setup which closely models a typical consumer home network. We then explain the design and implementation details of chunklets. Finally, we describe the QoE metrics used as performance indicators in our experiments.

3.1 Consumer home network environment



(a) Use case model

(b) Experimental testbed setup

Fig. 1. Our use case model and experimental testbed setup emulating a home network connected to the Internet with AQM-enabled at the ISP-end or/and home gateway.

Figure 1a illustrates our use case model – a home network that is connected to local or international video streaming and other Internet-based services over a last-mile broadband link. This link typically offers asymmetric bandwidth. The *downstream* bottleneck (for traffic flowing into the home) is at the ISP-end while the *upstream* bottleneck (traffic flowing away from home) is at the home gateway. AQMs are likely to replace FIFO in modern and affordable home gateways, with ISPs progressively upgrading their equipment with AQMs. We consider use cases where DASH, latency-tolerant (bulk TCP) and latency-sensitive (VoIP) flows are competing over a shared bottleneck link managed by FIFO and {PIE, FQ-CoDel} AQMs.

3.1.1 Sharing the last-mile bottleneck. Due to its chunked video retrieval, DASH generates a periodic ON-OFF traffic pattern in the steady-state, effectively creating repeated bursts of elastic traffic. Interesting challenges arise when DASH's OFF period overlaps with other downstream traffic. Competition with other downstream bulk (elastic) data transfers can lead the DASH client into a 'downward spiral' of progressively lower rate estimations [16]. Upstream bulk transfers compete with a DASH connection's TCP ACK packets. When the upstream link is congested, the DASH connection's ACKs suffer additional queuing delays (and potentially loss), indirectly limiting the DASH client's estimate of available network capacity.

Latency-tolerant applications include long-lived bulk transfers whereas *latency-sensitive* applications are interactive real-time applications that require low RTTs for optimal performance and end-user experience. Examples of competing traffic in the home include file downloads/uploads, VoIP calls, video conference calls, online gaming and so on. All these activities will result in link congestion in the direction of data flow, leading to RTT inflation for all traffic sharing the bottleneck at the time, which can be detrimental to DASH video streams and latency-sensitive applications.

3.1.2 Representative network conditions. Video streaming providers typically try to install content servers or caches close to their customers, to minimise delays caused by a path's base RTT (minimum distance in time between two endpoints), and reduce the cost of transiting intermediate network providers. Hence, we choose to emulate relatively low RTTs (20ms) in our experiments.

Home broadband services vary around the world. We choose to emulate a last-mile offering 12Mbps downstream / 1Mbps upstream (12/1 Mbps), such as would be achieved by a medium-performance ADSL2+ connection [6]. The 12Mbps downstream significantly exceeds the highest RR of our video dataset (Table 1), ensuring the highest RR is attainable for our DASH client.

The base RTT for realistic bulk downloads varies according to the services accessed. Many studies have shown that bulk flows utilise bandwidth more effectively when the base RTT is low (share of bandwidth biased towards the path with lower RTT). So we set the base RTT of the bulk flows to be the same as the DASH flows to allow for a more aggressive inter-flow competition.

3.2 Experimental setup

Figure 1b shows our TEACUP-based [43] testbed. The router runs 64-bit FreeBSD 10.1-RELEASE to provide a configurable bottleneck (bandwidth, delays, queuing disciplines, buffer sizes) between client(s) and server(s) on either side of the router. The DASH client, proxy host and nginx[7] Web server run 64-bit FreeBSD 10.1-RELEASE whereas the iperf[8] client and server run on 64-bit Linux openSUSE 12.3 (kernel 3.17.4). Each end-host is physically an Acer Veriton X6630G (8GB RAM, Intel Core™ i5, Intel 82574L Gigabit NIC) machine and the router is a Supermicro X9SRi-F (16GB RAM, 3.70 GHz Intel Xeon® E5-1620v2, Intel I350-T2 dual port Gigabit NIC) machine.

3.2.1 Emulating network conditions. We use FreeBSD's ipfw/dummynet [2] to provide FIFO, PIE and FQ-CoDel queue management schemes and emulate specific last-mile rate limits and base RTTs. Packets sit in a configurable bottleneck queue while being rate-shaped to the bottleneck bandwidth, and then sit in a 1000-packet buffer while being delayed. The bottleneck queue is 340 packets for FIFO experiments (to exceed each path's unloaded BDP, ensuring TCP does not under-utilise the path even after *cwnd* reduces upon congestion) and 1000 packets for PIE and FQ-CoDel experiments (recommended by IETF). We configure upstream and downstream queues separately, allowing us to apply different queuing disciplines and buffer sizes in each direction if necessary.

[7] https://www.nginx.com/
[8] https://iperf.fr/

3.2.2 Creating DASH flows. We use dash.js version 2.9.0[9] in Mozilla Firefox browser as our DASH client, and nginx version 1.12.1 (with persistent HTTP connections and range-requests support enabled) as our DASH server. Our dash.js client uses the default 'Dynamic ABR' strategy, which combines both throughput and buffer-based algorithms [36]. We use FreeBSD and TCP NewReno on our DASH server and client. The combination of FreeBSD NewReno and nginx Web server in our test environment is inspired by Netflix's Open Connect platform[10], which is responsible for serving all of Netflix's video traffic on the Internet. Since Netflix uses a customised version of NewReno and our work focus on the client-end, we use the publicly released version of FreeBSD NewReno in our emulated network.

Table 1. Representation Rates (RR) available in the '*TheSwissAccount*' video dataset with 2-second chunks

Resolution	RR	Resolution	RR
320x240	91, 131 kbps	1280x720	1.3, 1.7, 2.1 Mbps
480x360	174, 216, 257, 337 kbps	1920x1080	2.7, 3.5, 4.0, 4.3 Mbps
854x480	431, 602, 764, 967 kbps		

We use '*TheSwissAccount*' dataset [28] and select content encoded in 2-second chunks of video. Chunks are available at 17 different encoding RRs ranging from approximately 91kbps to 4.3Mbps as shown in Table 1. We ran experiments for 450 seconds to ensure sufficient 2-second samples were collected. The DASH video stream is started after all competing flows have started and stabilised in order to construct "worst-case" scenarios to stress-test chunklet performance.

3.2.3 Creating competing flows. We use Linux and TCP CUBIC to generate bulk TCP flows with iperf, as many online services run on Linux and CUBIC is known for its effective bandwidth utilisation (providing a good stress test for DASH video streams). All hosts disable ECN and enable both TCP window scaling and receive buffer auto-tuning. We emulate bidirectional VoIP calls with iperf, generating 280kbps upstream and downstream User Datagram Protocol (UDP) flows.

3.2.4 'Chunkleting' proxy (engine). 'Chunklets' is a conceptual idea that should ideally be implemented inside a DASH client. However, in our experimental setup (Figure 1b), we use a Python-based HTTP proxy to instantiate chunkleting heuristics (Section 2.4) over our emulated network without needing to modify dash.js. This is a *proof-of-concept*, to motivate native integration of chunklets into future DASH clients.

Our approach enables us to run experiments directly using the latest version of the dash.js client with its latest enhancements to ABR algorithms (upgraded independent of chunkleting heuristics) and provide better insights into *chunk-chunklet* interactions. Since both hosts are connected via a 1Gbps local link, we have confirmed that the time delays between the client and proxy are negligible (~1ms). The ARs calculated at the client are almost identical to the ARs measured at the proxy.

In our implementation, the proxy transparently turns chunk requests into chunklet requests, then stitches the replies back together into chunks. The proxy intercepts each HTTP GET request from the client and parses any range-requests to identify the chunk being requested. The proxy calculates N consecutive sub-ranges of bytes to represent the N chunklets, and sends N simultaneous HTTP GET requests (one for each chunklet's sub-range) over N parallel persistent connections to the DASH server. As responses arrive from the DASH server, the proxy reassembles and concatenates N chunklet responses, recreating the full chunk requested by the client. The proxy's actions are thus transparent to both the client and the server. For implementation simplicity, a chunk of Y bytes

[9] https://github.com/Dash-Industry-Forum/dash.js/
[10] https://openconnect.netflix.com/software

results in $(N-1)$ chunklets of $int(Y/N)$ bytes and a final chunklet carrying between $[int(Y/N)]$ and $[int(Y/N) + (N-1)]$ bytes. Our proxy also has a configurable 'minimum chunklet size' threshold that prevents the chunkleting of small video chunks due to considerations related to network efficiency and TCP dynamics when dealing with a small number of packets.

3.3 Performance indicators

3.3.1 QoE Metrics. We do not endorse any particular QoE models in this work. Our goal is to demonstrate the impact of chunklets and AQM on directly measurable QoE metrics. These metrics can then be applied to the QoE model of choice. We assessed chunklets' performance by means of *AR*, *RR*, *instability index*, *video startup delays* and *rebuffers* (as defined in the literature [35]). In order to ensure their relevance to long-term viewing, all performance indicators (except video startup delays) are calculated after the playout buffer is pre-filled (*i.e.* during the ON-OFF steady-state).

We analysed DASH AR at the network level so as to better understand the interactions between DASH, TCP and AQM. We calculated per-chunk ARs by extracting payload lengths from HTTP response headers and the time taken to transfer packets making up each chunk. We then take the AR average from the past four chunks to mimic the signal used by dash.js' ABR algorithm for selecting the next chunk's RR. We verified that AR values are close-to-identical with the throughput values calculated by the dash.js client. RRs are extracted by parsing the client's HTTP GET requests.

Users are sensitive to frequent and significant RR switches [32] so we use the extracted RRs to derive the number of RR transitions and calculate the following *instability index* as defined in [20]:

$$\frac{\sum_{d=0}^{k-1} |b_{x,t-d} - b_{x,t-d-1}| . w(d)}{\sum_{d=1}^{k} b_{x,t-d} . w(d)}$$

Our instability index is obtained by calculating the weighted sum of the number of RR level (Table 1) transitions within the last $k = 10$ video chunks (where $b_{x,t}$ is the RR level retrieved at time t), which is then normalised to a value between 0 and 1 by dividing it by the weighted sum of all RR levels observed in the last 10 chunks (corresponding to 20 seconds of video). The weight function $[w(d) = k - d]$ is applied to add a linear penalty to the more recent RR switches. The window is then slided in steps of one, and the instability index is re-calculated for each window. Although an instability index close to zero indicates a more stable system, the instability index cannot be used alone to determine a DASH stream's likely QoE. It must be considered in the context of corresponding RR distributions. For instance, a DASH stream experiencing high instability with higher median RR might provide better experience than a DASH stream with low instability at lower median RR. A DASH stream that self-limits itself to a lower range of RRs might be perceived as having a low instability index but it is in fact a "consistently bad" user experience.

Our 450-second experiments with 2-second video chunks ensure we have approximately 225 AR and RR samples per-experiment. Since our instability index is derived from the number and magnitude of RR transitions, it indirectly captures the statistical spread of AR and RR values.

3.3.2 TCP and AQM statistics. Traffic was captured using tcpdump on all host and router interfaces. This data was then processed by Synthetic Packet Pairs (SPP) [42] to construct per-packet network-layer RTT measurements. One Way Delay (OWD) values are calculated by matching source and destination packets. TCP connection statistics were logged using packet-driven TCP stack statistics gathering tools, SIFTR under FreeBSD and ttprobe [3] under Linux.

We modified FreeBSD's AQM to log internal queue states of FreeBSD AQMs and detect hash collisions in FQ-CoDel experiments (discussed in more detail in Section 6.3). All experiment results presented in Section 4 are free from hash collisions.

4 PERFORMANCE ANALYSIS

In this section, we first illustrate the '*AR multiplication effect*' brought by chunklets, then evaluate chunklets' performance across a broad range of settings. We construct experiment trials to model home broadband environments where an end-user is streaming video content while:

- Other users/applications are concurrently downloading multiple files (greedy but latency-tolerant). We show how chunklets can improve streaming experience in such scenarios.
- There is a VoIP-like (latency and packet loss-sensitive) call during the video streaming session amidst other competing file downloads. We demonstrate the impact of chunklets on latency and packet losses, and show the need for *'adaptive chunklets'*.

4.1 Chunklets: The Achieved Rate multiplication effect

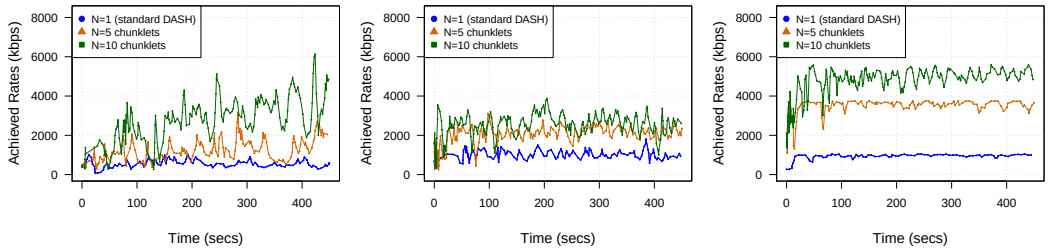
4.1.1 Definition and illustrations. With standard single-connection DASH, a chunk is retrieved and transferred every cycle time sequentially. Simplistically, when competing with M flows across an FQ-CoDel bottleneck, its effective per-chunk achieved rate (AR_1) is $\frac{1}{M+1}$-*th* of the bottleneck bandwidth (C), hence the DASH client can only retrieve RR that is sustainable by AR_1. However, when using N concurrent connections (chunklets), each chunklet flow is perceived as a unique flow to FQ-CoDel, hence DASH's AR sees a multiplication effect with an approximate increased rate of $AR_N = (\frac{N}{M+N}) * C$, allowing the client to retrieve RR sustainable up to AR_N, effectively "stealing" capacity from cross-traffic to provide a higher aggregated throughput. In the absence of competing flows M, a single-connection DASH client will achieve very similar AR to a chunklet-enabled client.

Figures 2 and 3 show our experimental results gathered using the methodology in Section 3.2 where a DASH client uses N={1, 5, 10} concurrent connections (chunklets) to retrieve content while facing competition from M=10 bulk TCP flows. The figures clearly illustrate the impact of increasing N on AR and RR when streaming over different bottleneck queue types. A standard DASH flow (N=1) can only achieve a limited fraction of the bottleneck bandwidth, leading to the retrieval of low RRs. FQ-CoDel remains the best choice for standard DASH as it stabilises the RR fluctuations by enforcing even capacity sharing.

When chunklets are used (using N=5 and 10), the overall AR increases, allowing the DASH client to select higher RRs and provide better QoE. Figure 3 shows the direct QoE benefits of chunklets, with FQ-CoDel being the most significant. The DASH client sees stable (no RR fluctuations in steady-state) and higher RRs when compared with PIE and FIFO. Single queue AQM (PIE) helps the DASH client to maintain a reasonable share of the bandwidth. Although chunklets allow the client to achieve a higher AR and RR with FIFO, the variations in RRs retrieved are large (unstable), hence one might argue not to use chunklets when the bottleneck uses FIFO, preferring low video quality to constantly-changing qualities.

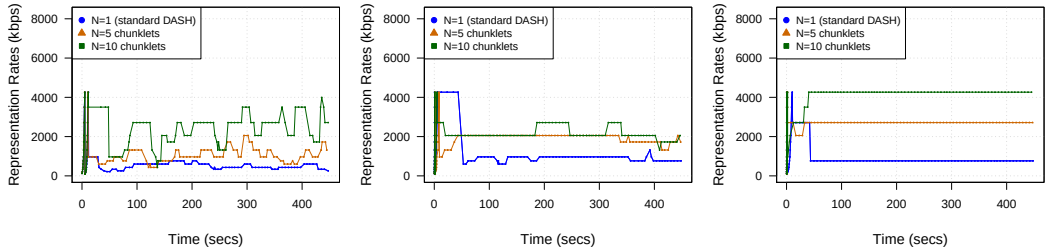
4.1.2 QoE Evaluations. Figure 4 compares the median AR, RR and instability index values when N={1, ...,10} chunklets compete with M={2, 4, 6, 8, 10} bulk flows over bottlenecks managed by {FIFO, PIE, FQ-CoDel}. The bottleneck capacity is set to 12/1Mbps (ADSL2+ service) and the base RTT is 20ms for all flows (streaming and downloading from local caches/servers).

In all {FIFO, PIE, FQ-CoDel} scenarios, overall AR increases with N with lower AR each time M increases. Figure 4b shows the number of chunklets (N) needed in order to achieve the *best RR* (encoded at 4.3Mbps) as the number of competing flows (M) varies. In FIFO and PIE experiments, the DASH flow could not achieve the best RR when M > 6 even with N=10 chunklets. However, PIE performs much better as higher RRs are being retrieved more consistently (lower instability index). PIE also enables the DASH stream to achieve the best RR with fewer chunklets when M=2 and M=4. For example, when M= 4, PIE enables the DASH client to achieve the best RR with N=3 whereas it needs N=7 when competing across a FIFO bottleneck.



(a) FIFO (b) PIE (c) FQ-CoDel

Fig. 2. DASH ARs when competing with 10 bulk flows (M=10): With (N=5, 10) and without (N=1) chunkleting over {FIFO, PIE, FQ-CoDel} bottlenecks. Chunklets enabled the DASH stream to achieve higher ARs, with FQ-CoDel seeing the most notable AR multiplication effects with its ability to share capacity evenly.



(a) FIFO (b) PIE (c) FQ-CoDel

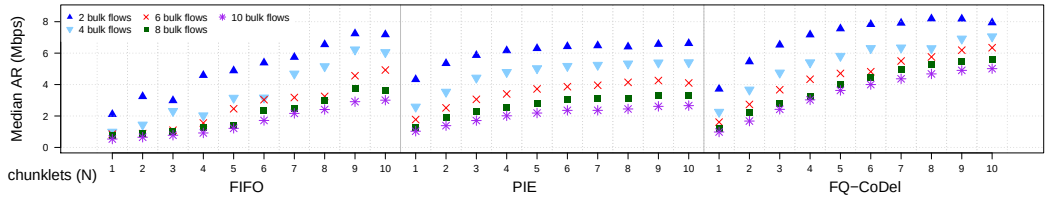
Fig. 3. DASH RRs when competing with 10 bulk flows (M=10): With (N=5, 10) and without (N=1) chunkleting over {FIFO, PIE, FQ-CoDel} bottlenecks. The higher ARs enabled by chunklets led to the retrieval of higher RRs, with FQ-CoDel resulting in the best user experience – higher and more stable RRs in steady-state.

Chunklets across a FQ-CoDel bottleneck performs the best. FQ-CoDel allows the DASH client to achieve the best RR with N=8, even when there are 10 competing bulk flows (not achievable with FIFO or PIE). Instability index values are zeros, indicating an excellent viewing experience without any RR transitions after the initial playout buffer pre-filling phase.

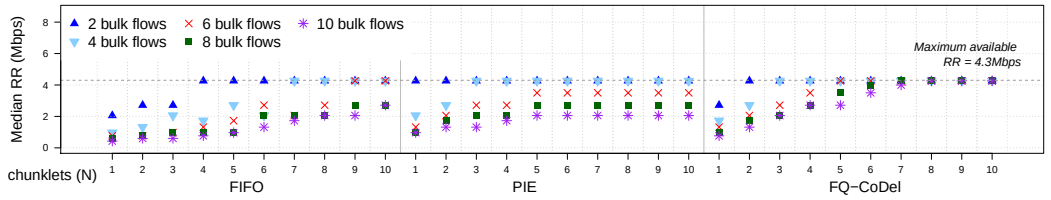
Interestingly, when M=2, a standard DASH client (N=1) could not select the best RR with FQ-CoDel as opposed to PIE. This is due to the fact that FQ-CoDel shares capacity strictly among all flows. In this scenario, FQ-CoDel ensures the DASH client only gets one third of the 12Mbps capacity (~4Mbps) which could not sustain the highest RR (4.3Mbps), causing the client to select the next best RR. But once N is increased to 2, the client comfortably selects the best RR. Since PIE achieves fair sharing by dropping packets from all flows probabilistically within a single queue, the DASH flow might well receive ARs just over 4.3Mbps to retrieve the best RR.

In addition to RR and instability index, we also captured video startup delays, number of rebuffers and rebuffering duration. We did not observe any rebuffers in our AQM experiments (PIE and FQ-CoDel). However, there were some occasional rebuffers in FIFO experiments (2-3 rebuffers in a 450-second experiment in extreme cases such as M=10, analysis omitted due to space constraints). Our results indicate that the client's ABR algorithm is selecting the appropriate RRs sustainable by the network. Since the lowest RR in our experiments is 91kbps, rebuffers are unlikely to happen as long as the capacity is greater than the lowest RR.

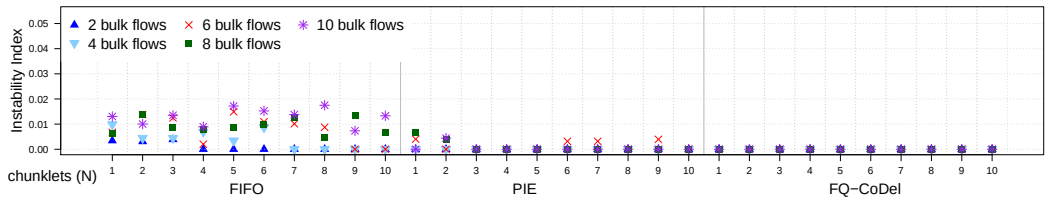
Users highly prioritise low video startup delays. A study of a large streaming network in 2012 observed that viewers start abandoning a video if it takes more than two seconds to load, with each



(a) Median Achieved Rates



(b) Median Representation Rates



(c) Median instability index

Fig. 4. Median AR, RR, instability index for N={1, ...,10} chunklets and M={2, 4, 6, 8, 10} competing bulk TCP flows over {FIFO, PIE and FQ-CoDel} bottlenecks. All three queue types see benefits of chunklets, but FQ-CoDel provides the best QoE with higher median RRs without any RR oscillations (zero instability index).

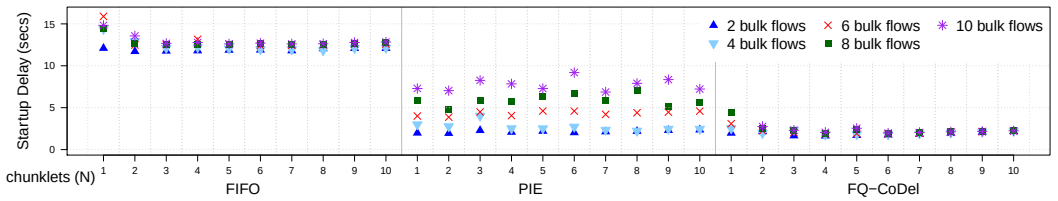


Fig. 5. Median video startup delays: AQMs lower startup delays with FQ-CoDel providing the lowest and most consistent startup delays.

incremental delay of one second resulting in a 5.8% increase in abandonment rate [22]. Figure 5 shows the effects of queue types and chunklets on video startup delays. We repeated each scenario 10 times and show the median startup delay values for each *AQM-N-M* combination. Multiple chunks ('startup chunks') need to arrive at the streaming client before it can start playing the video. Since the competing bulk flows started well before the video stream, video packets arrive at an already-congested queue. Video startup delays are directly proportional to both FIFO buffer size

and the number of competing flows (conclusion derived from a separate set of experiments, results not shown in Figure 5 due to space constraints). As non-video packets start filling the FIFO queue, the 'gap time' between video packets increases. The DASH flow perceives an inflated RTT path, causing the client's ACKs to be sent at a lower rate, which then rate-limits the server's sending rate (TCP *cwnd* growth), hence increasing the time needed to deliver the startup chunks – an evident side-effect of bufferbloat. Chunklets effectively allow more video packets to be transmitted per unit time, hence lowering the time needed to retrieve all startup chunks.

AQMs mitigate bufferbloat by controlling the queuing delay directly, allowing all flows sharing the same bottleneck to achieve relatively even throughput and low/stable latency. These properties indirectly allow the server to send video chunks at a higher rate, reducing the time needed for multiple chunks to arrive at the client, thus lowering the startup delay.

PIE provides improvements but with FQ-CoDel, the DASH client is able to consistently achieve low (lowest among all queue types) and similar (1-2 seconds) video startup delays regardless of M (except when M=10). In our FQ-CoDel experiments, all flows experienced very similar (low) RTT when DASH flows compete with bulk TCP flows due to FQ-CoDel's flow isolation properties. Sparse, low-rate flows through an FQ-CoDel bottleneck will not experience RTTs similar to bulk flows competing through the same bottleneck. The FQ-CoDel DRR scheduler prioritises these low-rate sparse flows. Section 4.2.2 illustrates this effect. The low RTTs are a result of CoDel-managed sub-queues. However, when N=1, the startup delay increases significantly as M increases. The extra delays are caused by the FlowQueue scheduler. In scenarios where no sparse flows are present, the scheduler serves each sub-queue in a round robin fashion. The time taken to serve the same sub-queue again is *{M × serialisation delay of each sub-queue}*, where serialisation delay is the time needed to transmit a data packet. Hence, as M increases, the time taken to serve all the packets making up the startup chunks increases. We will discuss this artefact further in Section 4.2.2.

4.2 Collateral impact on competing flows

The question of inter-flow fairness and friendliness is always raised in contexts where a single application initiates multiple connections. Such applications are often seen as 'greedy and selfish'. Using chunklets to increase per-application throughput can be perceived as an unfair technique that could possibly impact other competing flows negatively. Here we investigate the collateral impact of chunklets on both *latency-tolerant* (bulk TCP transfers) and *latency-sensitive* (VoIP traffic) flows and show that chunklets do not adversely harm cross-traffic when used with AQMs.

4.2.1 Impact on latency-tolerant bulk TCP flows. Figure 6 shows the aggregate throughput of all competing TCP flows. We calculated per-flow throughput by averaging all the bytes transferred across time. The aggregate throughput is then calculated by adding all (M) per-flow throughputs, illustrating the amount of bandwidth consumed by competing flows during DASH steady-state.

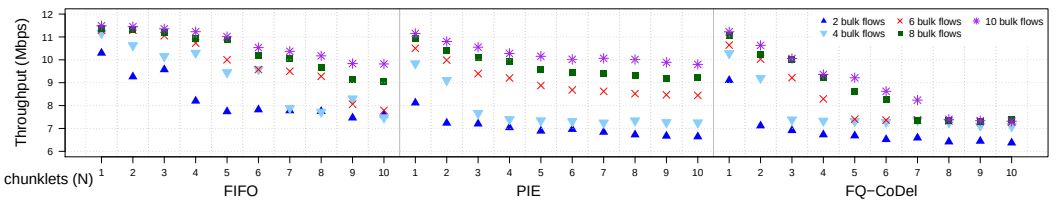


Fig. 6. Aggregate bulk transfer throughput, M={2, 4, 6, 8, 10} flows: An increase in chunklets consumes more bandwidth, leaving less bandwidth for competing flows.

AQM sacrifices throughput for lower latency. Bulk flows are able to achieve the most out of a FIFO queue as they constantly fill its buffer (its size typically approximates or exceeds BDP) to the full, whereas PIE and CoDel emulate queues that are smaller than BDP (similar observations in prior work [26]). As N increases, the aggregated throughput of bulk flows trends downwards. The decrease is more significant with FQ-CoDel as it enforces strict bandwidth sharing. The combined throughput can be simplistically described as $Throughput_{total} = (\frac{M}{M+N}) * C$ and the throughput of one bulk flow is $(\frac{1}{1+N}) * C$.

End-users will have to choose between having a better streaming experience or 'quicker' bulk transfers. End-users usually prioritise streaming experience over 'quick file downloads', and they are more tolerant to higher file download times than low-quality videos. In Section 5 we will describe how a friendly chunkleting client can be designed to not unnecessarily starve other flows.

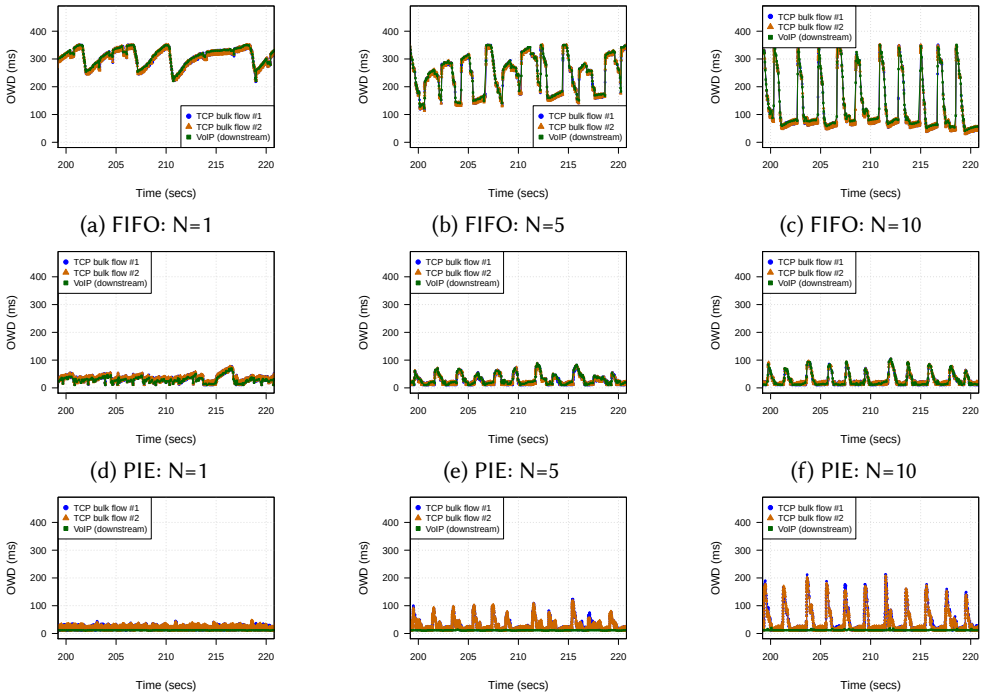
4.2.2 Impact on latency-sensitive VoIP flows. Low latency, jitter and packet loss rates are stringent requirements for real-time, latency-sensitive applications such as VoIP calls and interactive online games. These applications typically generate UDP flows that do not react well to network congestion. In another set of experiments, we added a 280kbps bidirectional VoIP traffic flow into the mix of chunklets and bulk flows, and evaluate the impact of chunklets on its QoS in terms of latency and number of packet losses. We choose to present results from N={1, 5, 10} to represent {no chunkleting, moderate chunkleting, aggressive chunkleting} and M=2 for presentation clarity. The same conclusions apply to other experimental scenarios.

Figure 7 illustrates the interactions between number of chunklets, queue types and one-way-delay (OWD) experienced by two bulk TCP flows and a downstream VoIP flow. The actual results depend on a variety of factors, including the specific TCP implementations driving each flow. Figure 7 illustrates a particularly aggressive scenario where the chunklet flows' FreeBSD source does not reduce *cwnd* during short inter-chunk idle (OFF) periods. The consequences are profound in the FIFO case (Figures 7a, 7b and 7c). The *cwnd* of each bulk flow going into an OFF period typically reflects its individual share of available bandwidth by the end of a preceding ON period. This share drops as N increases during each ON period, leaving the bulk flows increasingly unable to fill the bottleneck queue during the short OFF periods (Figures 7b and 7c).

In both single queue settings (FIFO and PIE), the VoIP flow yields an OWD curve similar to other competing flows. When N=10, PIE managed to keep OWD under 100ms whereas the bufferbloated FIFO caused the VoIP flow to experience > 300ms delay and large variations (jitter) when N=10. With FQ-CoDel, the VoIP traffic experiences zero packet loss (Figure 8c) and OWD close to the path's intrinsic 10ms OWD (Figures 7g, 7h and 7i) for two reasons. Firstly, FQ-CoDel hashes the flow into its own queue, isolating the flow from competing traffic. Secondly, FQ-CoDel's modified DRR scheduler prioritises the VoIP packets because they arrive infrequently relative to the competing traffic. Prior work also illustrates the benefits of FQ-CoDel on low-rate/interactive traffic [5].

Bulk flows see higher spikes in OWD with FQ-CoDel when N increases. Due to strict DRR scheduling, peak delays in FQ-CoDel are influenced by the number of active queues at any point in time and their respective serialisation delays [15]. With every increase in the number of flows, OWD increases by a unit of serialisation delay for a *quantum of bytes* (a quantum is the amount of bytes that is served by the FlowQueue scheduler for a sub-queue per-iteration, it is set to 1500 bytes by default), contributing to the overall delay spikes observed in FQ-CoDel in addition to queuing delays within a CoDel-managed sub-queue.

Bulk transfers are more tolerant of packet losses as they rely on TCP to ensure ordered and reliable packet delivery. On the other hand, latency-sensitive flows are less tolerant of packet losses as they often use UDP for quick and real-time delivery. Figure 8 shows the cumulative packet losses of bulk transfer flows and the downstream VoIP flow for {N=10, M=2} trials (similar conclusions



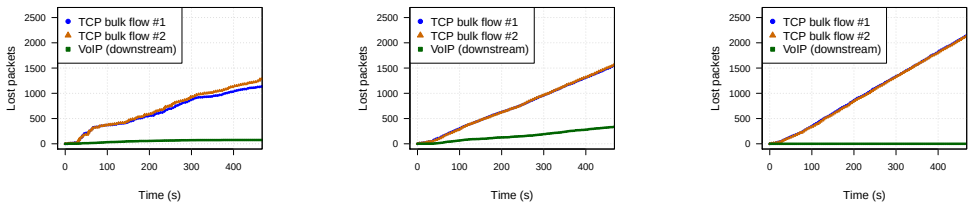
(a) FIFO: N=1 (b) FIFO: N=5 (c) FIFO: N=10

(d) PIE: N=1 (e) PIE: N=5 (f) PIE: N=10

(g) FQ-CoDel: N=1 (h) FQ-CoDel: N=5 (i) FQ-CoDel: N=10

Fig. 7. OWD experienced by TCP bulk flows and downstream VoIP for N={1, 5, 10} and M=2, in a 20-sec window. The VoIP flow is protected and prioritised by FQ-CoDel, hence it experienced delays that are close to the path's intrinsic OWD (10ms).

apply to other scenarios). FQ-CoDel reliably ensures *zero packet loss* for the VoIP flow. VoIP in both FIFO and PIE loses packets, more with PIE as it indiscriminately drops packets based on the calculated probability. Although VoIP loses fewer packets with FIFO, its latency is a lot higher than PIE. Both PIE and FQ-CoDel AQMs induce similar numbers of packet losses on bulk flows, with CoDel (within FQ-CoDel) being more aggressive in dropping packets to control queuing delays.



(a) FIFO: N=10 (b) PIE: N=10 (c) FQ-CoDel: N=10

Fig. 8. Cumulative packet losses of TCP bulk flows and downstream VoIP flow when N=10 and M=2. FQ-CoDel protects and priorities the low-rate VoIP flow, hence no VoIP packet losses.

In conclusion, the combination of chunklets and FQ-CoDel is the most effective strategy. Video streams achieve better user experience (Section 4.1.2), latency-sensitive applications are unharmed (Section 4.2.2) with tolerable throughput and latency sacrifices from non-real-time TCP bulk

transfers (Section 4.2.1). In the following section, we will discuss how to minimise such collateral impact on bulk transfer flows with '*adaptive chunklets*'.

5 ADAPTIVE CHUNKLETS

Section 4.2's results show that an unbounded increase of chunklets will have undesirable effects on competing flows (lower throughput, intense latency spikes and packet losses). Chunklets need to behave like a good net-citizen – tuned to *achieve maximum QoE with the minimum number of chunklets*, so as not to starve other flows unnecessarily. Limiting the number chunklets (and hence open TCP connections) will also reduce the load of servers/middleboxes, which often devote valuable resources maintaining per-flow connection state and information logging.

Hence, we propose *'adaptive chunklets'* – a system that dynamically determines the optimal number of chunklets at any given time. In this section, we describe the design and implementation of the adaptive chunkleting engine. We start by explaining the architecture and the functionality of each component underlying the engine. We then experimentally demonstrate its operation.

5.1 Design

As observed experimentally in Section 4.1.2, the relationship between number of chunklets (N), competing flows (M), the resulting Achieved Rates (AR) and Representation Rates (RR) can be illustrated with Figure 9 (assuming the bottleneck capacity is greater than maximum RR).

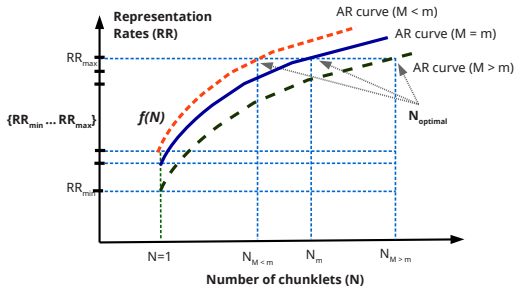


Fig. 9. Chunklets rate map

Figure 9 illustrates three scenarios – when $M = m$, $M < m$ and $M > m$, where m is an arbitrary number of competing bulk flows. A smaller m will result in an AR curve starting off (when N=1) at a higher point and achieves AR values > RR_{max} "earlier" (with a smaller N) than scenarios with higher m. We define the minimum N needed to achieve RR_{max} as the optimal operating point, $N_{optimal}$.

The adaptive chunkleting engine uses a minimally intrusive probing mechanism to determine $N_{optimal}$. We define a three-chunk probe period in which the three chunks are retrieved using {N-1, N, N+1} chunklets respectively, where N is the current number of chunklets used. We then measure their ARs respectively and decide on $N_{optimal}$. We now describe our design principles below.

5.1.1 Where: Identifying eligible probe-triples. Probing should be done with eligible 'probe-triples' – when three consecutive chunks are the most similar in size within an RR level, with minimal influences of video chunk sizes on measured ARs during the probe period. Due to VBR encoding, chunk sizes can vary significantly within the same RR level, as illustrated in Figure 10, motivating research and development of chunk/segment-size aware ABR algorithms [21].

We define an *eligible probe-triple* as three consecutive chunks with similar chunk sizes. In order to identify these eligible probe-triples, our chunkleting engine identifies all the available RRs and the chunk sizes (using byte-range information) within them. We slide a three-chunk window in

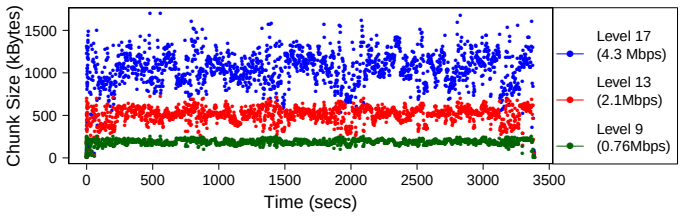


Fig. 10. 'TheSwissAccount' chunk size variations for RR Level {17, 13, 9} encoded at {4.3, 2.1, 0.76} Mbps

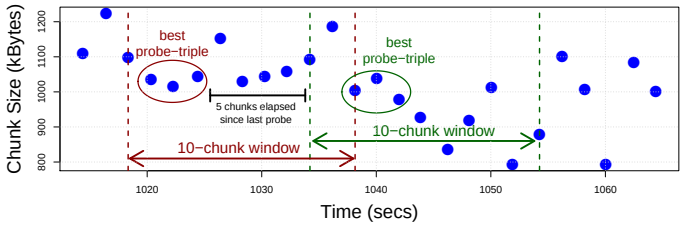


Fig. 11. Illustrating how best probe-triples are selected (chunk sizes from 4.3Mbps RR, t=1010-1070s)

steps of one and calculate the 'worst-case' chunk-size-difference in percentage by dividing the largest size difference with the smallest chunk size. For every probe-triple, we also identify the "chunk size order" (which of the three chunks is the largest/smallest/in-between in size).

We then determine eligible probe-triples by selecting the *best probe-triple* within a certain time window. For example, we declare a probe-triple as eligible if its percentage difference in size is the *smallest* amongst all the triples within a period of 10 chunks. This approach ensures that we always have an opportunity to probe (using the best available probes) within a 10-chunk period.

5.1.2 When: Identifying suitable probe times. A suitable probing interval should be carefully chosen so chunklets can react to network changes relatively quickly without probing too frequently. Using the heuristics described in Section 5.1.1, the chunkleting engine knows all the possible 'probe locations' at the start of a streaming session, but in order to execute the probe, a certain amount of time must have elapsed since the last probe to avoid frequent probing. A suitable interval will allow the chunkleting engine to react to network changes relatively quickly so as to maintain or improve user experience when more flows start competing during a streaming session. Figure 11 illustrates how the best probe-triples are selected using a 10-chunk window and a 5-chunk interval. Five chunks will have to elapse since the last probing phase before starting the next window. In the first window, Chunk #{2, 3, 4} are selected as the 'best probe-triple' because they are the most similar in size, whereas Chunk #{3, 4, 5} are selected as the 'best probe-triple' in the second window.

5.1.3 How: Probing heuristics. A different N should only be used for one chunk during the probing phase with an increment/decrement of one. Probes should happen swiftly without any effects noticeable to the user. In a setup where the DASH client and chunkleting engine are decoupled, an aggressive probe – increasing/decreasing N by a large step size and keeping it for a long period of time (*e.g.* more than one chunk) – can cause large fluctuations in AR values, thus causing unwanted RR oscillations. Since DASH clients typically use throughput (AR) signals calculated based on the average of throughput estimates measured across multiple chunks, increasing/decreasing N one chunk at a time will less likely cause fluctuations in RRs. Intuitively, if chunkleting is implemented natively in the DASH client, its ABR algorithm can avoid this problem by keeping RR constant during the probing phase. In both cases, varying N one chunk at a time with a single increment/decrement step is both *time efficient* and *conservative*. One chunk is also sufficient to test the network path as

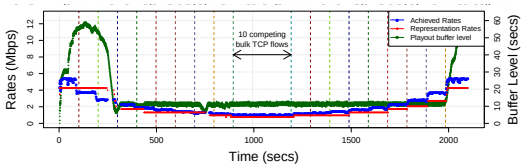
the TCP *cwnd* should experience multiple congestion epochs when retrieving a multi-second video chunk. If the probe is deemed executable, we enter the probing phase with the following steps:

(1) Determine which chunk to use {N-1, N, N+1}. Since we want AR values to reflect the utmost impact of varying N (reducing the impact of chunk sizes further), we will use N+1 for the largest chunk, N-1 for the middle-size chunk and keeping N for the smallest chunk.
(2) Calculate the AR gain percentage of N+1 and N-1 relative to keeping N constant.
(3) We increase N if using {N+1} provides significant improvements, decrease N if using {N-1} does not detrimentally impact AR. Otherwise, we keep N constant. In this work, we consider a fixed value of ±8% as a threshold. The value is derived from a separate analysis (not presented due to space constraints) according to the network settings presented in this paper. If we have achieved RR_{max}, we will try to decrease N if the resultant AR still sustains RR_{max}.

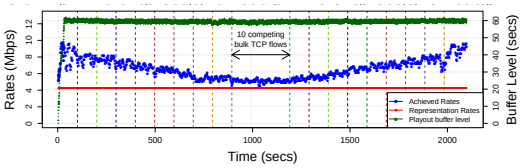
The net effect will be an increase in N if needed but a downward drift in N if a higher N is no longer necessary (*e.g.* when competing flows stop and cede capacity). In the event where the client decides to switch RR, we exit the probing phase and wait for the next eligible probe.

5.2 Evaluation Results

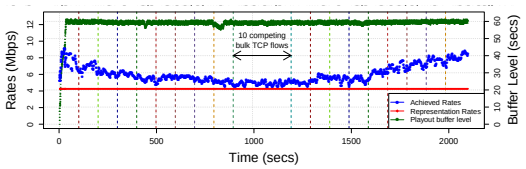
We now demonstrate adaptive chunklets across a shared {12/1 Mbps, 20ms base RTT, FQ-CoDel} bottleneck with three distinct scenarios – *Without chunklets* (standard DASH, N=1), *with fixed chunklets* (N=10) and *with adaptive chunklets* (with an upper bound of 10). These trials essentially combine scenarios presented in Section 4 with M increasing from 1 to 10 within a single experimental trial. In these experiments, one bulk TCP flow starts well before the DASH video flow; every 100 seconds, a new bulk TCP flow arrives until all 10 flows are active for 300 seconds, and then one flow terminates every 100 seconds.



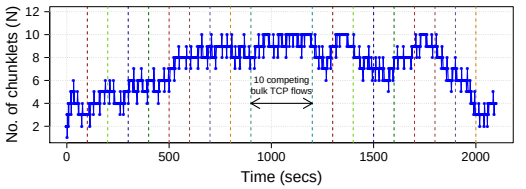
(a) AR, RR, playout buffer vs time: N=1, no chunkleting



(b) AR, RR, playout buffer vs time: N=10, fixed chunklets



(c) AR, RR, playout buffer vs time: Adaptive chunklets



(d) N vs time: Adaptive chunklets

Fig. 12. Comparing N={1, 10, adaptive} chunklets: DASH client competes with 10 staggered start/stop bulk flows. Each pair of coloured dotted lines represents the start and end time of a bulk TCP flow.

As shown in Figure 12a, a standard DASH client selects a lower RR as new competing bulk flows start due to a reduced share of bottleneck capacity. It then recovers and selects higher RRs when the competing flows gradually cede capacity. Figure 12b shows that with fixed chunklets (N=10), the DASH client consistently retrieves the highest RR because 10 concurrent TCP connections provide sufficient AR even when 10 bulk flows are present. However, the ARs are higher than necessary when fewer flows are competing, causing unnecessary damages to other flows.

Adaptive chunklets enable the DASH client to retrieve the highest RR consistently without using 10 active TCP connections throughout the session. Figure 12c shows how adaptive chunklets adjust N so that ARs are 'just enough' to retrieve the highest RR (bringing the AR values closer to the RR_{max}). Figure 12d shows an upward trend in N as new bulk flows start to compete and stabilises between 8-9 connections when 10 bulk flows are competing ($N_{optimal}$ when M=10, *cf.* Figure 4b), then trends downwards as competing flows terminate. At t=1200-1300s, we observed that N decreased for a while before returning to a more stable N. A temporary decrease in N will not result in an immediate decrease in the AR signal and RR retrieval since the AR input into ABR algorithms is typically calculated with an average across multiple video chunk retrievals. Here the engine tries to decrease N if the current RRs are already the best possible RR. When it discerned that the ARs provided by N=6 could not sustain the best RR anymore, it probes to increase N.

Overall, our experimental results show that adaptive chunklets can reduce the number of TCP connections by almost 30% while maintaining the best RR. Adaptive chunklets also utilised almost 8% less bandwidth than fixed chunklets, making more bandwidth available for other flows. During the period when they are competing with 10 bulk flows (t=1000-1300s), adaptive chunklets consumed almost 11% less bandwidth, affirming the benefits of adaptive chunklets.

6 DISCUSSION AND FUTURE WORK

6.1 Deployment considerations

In our current implementation, chunklets are decoupled from the client and managed at the proxy-level. Chunklets should ideally be integrated natively into the DASH client, creating a tighter coupling between chunkleting engine and the ABR algorithm. Commercial servers typically serve hundreds of thousands of requests per-second. With each connection state being maintained by the server (and middleboxes) and the underlying transport layer, using chunklets will increase server load and network overhead. The resources needed will increase dramatically as the number of chunklet-enabled clients increases. Hence, a properly-tuned adaptive chunkleting scheme should ensure streaming clients do not initiate more concurrent connections than necessary. One limitation of our current adaptive chunklets system is that we still keep a number of unused TCP connections open in case they are needed again. In practice, a client should aim to close unused connections after an appropriately-determined timeout to reduce server load. On the same token, constantly opening and closing connections should be also avoided.

6.2 Decorrelating chunklet requests

We assume the packets making up clusters of chunklets are interleaved on the return path out of the HTTP server. However, it is also true that the initial packets of the first chunklet will arrive at the bottleneck earlier than those of the second chunklet, and so forth. We observed that the second to N^{th} chunklets (and their associated TCP connections) experience slightly higher loss probabilities, particularly when passing through a FIFO bottleneck (*cf.* Figure 8). We decorrelate this source of self-interference by rotating chunklet requests across the underlying TCP connections. The negative effects of correlated requests are not obvious in our implementation as a chunk always needs to wait for the whole chunklet cluster to arrive before reassembly.

6.3 Hash collisions in FlowQueue-based AQM

In FlowQueue-based AQMs, hash collisions occur when the hash function generates the same hash for more than one flow, resulting in multiple flows being placed into the same sub-queue. If X flows collide in a single sub-queue, all X flows will share $\frac{1}{Y+1}$-th of the bandwidth (where Y represents other concurrent flows in separate sub-queues). Since FQ-CoDel implements 1024 hash buckets

by default, hash collisions will definitely occur when there are more than 1024 concurrent flows. RFC 8290 discusses the probability of hash collisions when the number of flows is less than 1024 using analytical equations and concludes that the probabilities are minuscule [14].

However, we did observe the rare hash collision problem in our experiments. Figure 13 shows a trial where a hash collision has occurred. Chunklet #5 is hashed into the same sub-queue as Bulk Flow #5, resulting in reduced throughput for both flows. Since a chunk can only be re-assembled after all chunklets have arrived, a delay in the arrival of Chunklet #5 has caused AR to drop, which then led to the DASH client's ABR algorithm selecting a lower RR.

(a) Impact on the DASH video stream

(b) Impact on bulk TCP flows

Fig. 13. FQ-CoDel, N=6, M=8: Chunklet #5 collides with Bulk Flow #5 resulting in reduced throughput.

6.4 Future work

A number of interesting research questions and future work items have arisen from this work. Our evaluation showed that both chunklets and adaptive chunklets work best with FQ-CoDel. Further experimentation is required to design a feedback algorithm so that the client can dynamically detect the bottleneck queue type and adjust its chunkleting heuristics accordingly. Statistically analysing per-chunk time-to-first-byte values and per-chunklet ARs will shed some light as their distributions will be similar over a multi-queue/FlowQueue-based AQM bottleneck but can vary substantially across a single-queue bottleneck. Future work will also include the evaluation of chunklets for low-latency and live streaming.

The benefits of chunklets depend on maximum overlap of chunklet responses on the return path from the server to the client. Hence, future work will include experiments with different server types running on different OSes with different TCP congestion control algorithms, and an analysis of how various server loads affect chunklet performance. Studies have shown that problems arise when multiple DASH streams compete, especially when their ON-OFF periods are synchronised [1]. The interactions between (adaptive/fixed) chunklet-enabled/single-connection DASH clients will be of interest. Future work will also include studying the impact of different ABR classes using representative algorithms such as FESTIVE [20], BOLA [37], MPC [41] and Pensieve [31].

Our current work considered baseline scenarios with a fixed rate-limited bottleneck. Future evaluations will consider publicly available bandwidth traces in testbed experiments and performance analyses of chunklets *in-the-wild*, *i.e.* using a chunklet-enabled client to retrieve content over the public Internet, and possibly in environments with "moving bottlenecks". Further optimisations need to be done for adaptive chunklets so as to make better decisions under various network conditions. A predictive numerical model of chunklet performance as a function of N, M, bottleneck bandwidth, network path's RTT and AQM type will also be pertinent.

7 CONCLUSIONS

In this paper, we presented *'adaptive chunklets'* – a novel technique orthogonal to ABR algorithms for improving user experience with a dynamic number of parallel connections. Chunklets effectively

provide 'extra' bandwidth capacity for ABR algorithms to act upon when faced with cross-traffic competition. We experimentally explored and characterised the impact of chunklets on DASH video flows and cross-traffic when they compete across a shared bottleneck managed by FIFO, single-queue AQM (PIE) and multi-queue AQM (FQ-CoDel). We showed that chunklets provide significant AR and RR improvements. Both PIE and FQ-CoDel AQMs reduce video startup delays but FQ-CoDel provides the best QoE by indirectly stabilising RRs with its flow-isolation ability.

However, at the start of every chunk retrieval, chunklets can cause intense queue buildup, delay spikes and packet losses to other competing flows, which can be particularly damaging to latency-sensitive flows. We showed how FlowQueue-based AQM can mitigate such effects with flow-isolation and traffic prioritisation for low bitrate flows such as VoIP, but most importantly, we demonstrated the effectiveness of adaptive chunklets in maximising QoE while keeping the number of concurrent connections low. Our experimental results show that adaptive chunklets can reduce the number of TCP connections by almost 30% and consume almost 8% less bandwidth than fixed chunklets while maintaining high RRs. Finally, we discussed deployment considerations of chunklets and identified potential future work that might be of interest to the streaming community.

REFERENCES

[1] S. Akhshabi, L. Anantakrishnan, A. C. Begen, and C. Dovrolis. 2012. What Happens when HTTP Adaptive Streaming Players Compete for Bandwidth?. In *Proceedings of the 22nd International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '12)*. ACM, 9–14.

[2] R. Al-Saadi and G. Armitage. 2016. *Dummynet AQM v0.2 – CoDel, FQ-CoDel, PIE and FQ-PIE for FreeBSD's ipfw/dummynet framework.* Technical Report 160418A. CAIA, Swinburne University of Technology, Melbourne, Australia.

[3] R. Al-Saadi, G. Armitage, and J. But. 2015. *ttprobe v0.1: Packet-Driven TCP Stack Statistics Gathering for TEACUP.* Technical Report 150911A. CAIA, Swinburne University of Technology, Melbourne, Australia.

[4] M. Ansari and M. Ghaderi. 2016. Parallel HTTP for Video Streaming in Wireless Networks. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 337–342.

[5] G. Armitage and R. Collom. 2017. Benefits of FlowQueue-Based Active Queue Management for Interactive Online Games. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 1–9.

[6] G. Armitage, J. Kennedy, S. Nguyen, J. Thomas, and S. Ewing. 2017. *Household Internet and the 'need for speed': Evaluating the impact of increasingly online lifestyles and the Internet of Things.* Technical Report 170113A. CAIA, Swinburne University of Technology, Melbourne, Australia.

[7] C. Bampis, Z. Li, I. Katsavounidis, T. Huang, C. Ekanadham, and A. Bovik. 2018. Towards Perceptually Optimized End-to-end Adaptive Video Streaming. *arXiv e-prints*, Article arXiv:1808.03898 (Aug 2018).

[8] A. Bentaleb, A. C. Begen, S. Harous, and R. Zimmermann. 2018. Want to Play DASH?: A Game Theoretic Approach for Adaptive Streaming over HTTP. In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys '18)*. ACM, 13–26.

[9] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, and R. Zimmermann. 2019. A Survey on Bitrate Adaptation Schemes for Streaming Media Over HTTP. *IEEE Communications Surveys & Tutorials* 21, 1 (Firstquarter 2019), 562–585.

[10] E. Blanton, V. Paxson, and M. Allman. 2009. TCP Congestion Control. RFC 5681. (Sept. 2009).

[11] K. Brunnström and S. A. Beker *et al.* 2013. Qualinet White Paper on Definitions of Quality of Experience. (2013). Fifth Qualinet meeting, Novi Sad, March 12, 2013.

[12] Jon Crowcroft and Philippe Oechslin. 1998. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Comput. Commun. Rev.* 28, 3 (July 1998), 53–69.

[13] J. Gettys and K. Nichols. 2011. Bufferbloat: Dark Buffers in the Internet. *Queue* 9, 11 (Nov. 2011), 40:40–40:54.

[14] T. Høeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. 2018. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290. (Jan. 2018).

[15] T. Høiland-Jørgensen. 2018. Analyzing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm. *IEEE Communications Letters* 22, 11 (Nov 2018), 2266–2269.

[16] T. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari. 2012. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proceedings of the 2012 ACM Internet Measurement Conference (IMC '12)*. ACM, 225–238.

[17] T. Y. Huang, C. Ekanadham, A. Berglund, and Z. Li. 2019. Hindsight: Evaluate Video Bitrate Adaptation at Scale. In *Proceedings of the 10th ACM Multimedia Systems Conference (MMSys '19)*. ACM, 86–97.

[18] T. Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. 2014. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM*

'14). ACM, 187–198.

[19] ISO/IEC. 2012. ISO/IEC 2309-1:2012 Information Technology: Dynamic Adaptive Streaming over HTTP (DASH) Part 1: Media presentation description and segment formats. (2012).

[20] J. Jiang, V. Sekar, and H. Zhang. 2012. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. ACM, 97–108.

[21] P. Juluri, V. Tamarapalli, and D. Medhi. 2015. SARA: Segment aware rate adaptation algorithm for dynamic adaptive streaming over HTTP. In *2015 IEEE International Conference on Communication Workshop (ICCW)*. 1765–1770.

[22] S. Shunmuga Krishnan and Ramesh K. Sitaraman. 2012. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proceedings of the 2012 ACM Internet Measurement Conference (IMC '12)*. ACM, 211–224.

[23] J. Kua and G. Armitage. 2017. Optimising DASH over AQM-Enabled Gateways Using Intra-Chunk Parallel Retrieval (Chunklets). In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 1–9.

[24] J. Kua, G. Armitage, and P. Branch. 2016. The Impact of Active Queue Management on DASH-Based Content Delivery. In *2016 IEEE 41st Conference on Local Computer Networks (LCN)*. 121–128.

[25] J. Kua, G. Armitage, and P. Branch. 2017. A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming Over HTTP. *IEEE Communications Surveys & Tutorials* 19, 3 (thirdquarter 2017), 1842–1866.

[26] J. Kua, S. H. Nguyen, G. Armitage, and P. Branch. 2017. Using Active Queue Management to Assist IoT Application Flows in Home Broadband Networks. *IEEE Internet of Things Journal* 4, 5 (Oct 2017), 1399–1407.

[27] R. Kuschnig, I. Kofler, and H. Hellwagner. 2010. Improving Internet Video Streaming Performance by Parallel TCP-Based Request-Response Streams. In *2010 7th IEEE Consumer Communications and Networking Conference*. 1–5.

[28] Stefan Lederer, Christopher Müller, and Christian Timmerer. 2012. Dynamic Adaptive Streaming over HTTP Dataset. In *Proceedings of the 3rd ACM Multimedia Systems Conference (MMSys '12)*. ACM, 89–94.

[29] C. Liu, I. Bouazizi, and M. Gabbouj. 2011. Parallel Adaptive HTTP Media Streaming. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*. 1–6.

[30] T. Maki, M. Varela, and D. Ammar. 2015. A Layered Model for Quality Estimation of HTTP Video from QoS Measurements. In *Proceedings of the 2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS '15)*. IEEE Computer Society, Washington, DC, USA, 591–598.

[31] H. Mao, R. Netravali, and M. Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 197–210.

[32] R. Mok, E. Chan, X. Luo, and R. Chang. 2011. Inferring the QoE of HTTP Video Streaming from User-viewing Activities. In *Proceedings of the First ACM SIGCOMM Workshop on Measurements Up the Stack (W-MUST '11)*. ACM, 31–36.

[33] Kathleen M. Nichols, Van Jacobson, Andrew McGregor, and Jana Iyengar. 2018. Controlled Delay Active Queue Management. RFC 8289. (Jan. 2018).

[34] R. Pan, P. Natarajan, F. Baker, and G. White. 2017. Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem. RFC 8033. (Feb. 2017).

[35] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia. 2015. A Survey on Quality of Experience of HTTP Adaptive Streaming. *IEEE Communications Surveys & Tutorials* 17, 1 (Firstquarter 2015), 469–492.

[36] K. Spiteri, R. Sitaraman, and D. Sparacio. 2018. From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player. In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys '18)*. ACM, 123–137.

[37] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman. 2016. BOLA: Near-optimal bitrate adaptation for online videos. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9.

[38] L Stewart, G Armitage, and A Huebner. 2009. Collateral Damage: The Impact of Optimised TCP Variants on Real-Time Traffic Latency in Consumer Broadband Environments. In *NETWORKING 2009*. Springer Berlin Heidelberg, 392–403.

[39] T. Stockhammer. 2011. Dynamic Adaptive Streaming over HTTP: Standards and Design Principles. In *Proceedings of the 2nd Annual ACM Conference on Multimedia Systems (MMSys '11)*. ACM, 133–144.

[40] G. White and R. Pan. 2017. Active Queue Management (AQM) Based on Proportional Integral Controller Enhanced (PIE) for Data-Over-Cable Service Interface Specifications (DOCSIS) Cable Modems. RFC 8034. (Feb. 2017).

[41] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. 2015. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, 325–338.

[42] S. Zander and G. Armitage. 2013. Minimally-Intrusive Frequent Round Trip Time Measurements Using Synthetic Packet Pairs. In *The 38th IEEE Conference on Local Computer Networks (LCN 2013)*.

[43] S. Zander and G. Armitage. 2015. *TEACUP v1.0 - A System for Automated TCP Testbed Experiments*. Technical Report 150529A. CAIA, Swinburne University of Technology, Melbourne, Australia.

Received November 2018; revised June 2019; accepted July 2019