Octopus: In-Network Content Adaptation to Control Congestion on 5G Links

Yongzhou Chen
University of Illinois
Urbana-Champaign, IL, USA
yc28@illinois.edu

Ammar Tahir
University of Illinois
Urbana-Champaign, IL, USA
ammart2@illinois.edu

Francis Y. Yan
Microsoft Research
WA, USA
francisy@microsoft.com

Radhika Mittal
University of Illinois
Urbana-Champaign, IL, USA
radhikam@illinois.edu

ABSTRACT

It is challenging to meet the bandwidth and latency requirements of interactive real-time applications (e.g., virtual reality, cloud gaming, etc.) on time-varying 5G cellular links. Today's feedback-based congestion controllers try to match the sending rate at the endhost with the estimated network capacity. However, such controllers cannot precisely estimate the cellular link capacity that changes at timescales smaller than the feedback delay. We instead propose a different approach for controlling congestion on 5G links. We send real-time data streams using an imprecise controller (that errs on the side of overestimating network capacity) to ensure high throughput, and then adapt the transmitted content by dropping appropriate packets in the cellular base stations to match the actual capacity and minimize delay. We build a system called Octopus to realize this approach. Octopus provides parameterized primitives that applications at the endhost can configure differently to express different content adaptation policies. Octopus transport encodes the corresponding app-specified parameters in packet header fields, which the base-station logic can parse to execute the desired dropping behavior. Our evaluation shows how real-time applications involving standard and volumetric videos can be designed to exploit Octopus, and achieve 1.5–18× better performance than state-of-the-art schemes.

KEYWORDS

Video Conferencing, 5G Networks, Edge Computing, Congestion Control, In-Network Computation

ACM Reference Format:
Yongzhou Chen, Ammar Tahir, Francis Y. Yan, and Radhika Mittal. 2023. Octopus: In-Network Content Adaptation to Control Congestion on 5G Links. In *The Eighth ACM/IEEE Symposium on Edge Computing (SEC '23), December 6–9, 2023, Wilmington, DE, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3583740.3628438

1 INTRODUCTION

A combination of two factors makes data transfer over 5G cellular networks extremely challenging: *(i) Stringent performance requirements of interactive 5G applications:* Interactive real-time applications, such as video conferencing, augmented and virtual reality (AR/VR), tele-robotics, cloud gaming, etc., are increasingly relying on cellular connectivity [9]. These applications typically transmit standard or volumetric real-time videos, which require both high throughput (e.g., for high video resolution and quality) and low latency (e.g., to ensure high responsiveness for cloud gaming and low motion-to-photon latency for AR/VR). *(ii) Dynamic link conditions:* The above requirements must be met on wireless links with time-varying capacities [23, 35, 54, 55, 60, 74]. These capacity variations are caused by varying signal strength between the base station and user devices (e.g., due to dynamic obstacles or changing distance and directionality). The higher directionality required by mmWave [40] makes 5G even more susceptible to such effects [54, 76].

Real-time data transfer over 5G continues to rely on *feedback-based* congestion controllers, where the endhost selects a sending rate to match the network capacity estimated using network feedback. The feedback could be implicit (e.g., packet loss [37, 41] or delay [19, 21, 74, 78]) or may involve more active engagement from the routers (e.g., early congestion notifications [31, 35], early drops [32, 57], or explicit rate signalling [48, 70]). It is important for a feedback-based controller to precisely match the sending rate with network capacity in order to meet the performance requirements for 5G applications—sending too little leads to low throughput (and thus low content quality), whereas sending too much leads to high queuing delays and packet drops (resulting in lags in video frame delivery).[1] However, as we show in §2, feedback-based controllers are fundamentally limited in their applicability to 5G networks—it is not possible for them to precisely estimate the link capacity when it changes at timescales that are smaller than the time taken to receive feedback from the network (as is the case in dynamic cellular links).

In this paper, we explore an alternative approach for transmitting real-time data streams over cellular links that circumvents the need for a precise feedback-based controller: We allow the sender to transmit excessive data to guarantee high throughput, and subsequently adapt the transmitted content in the 5G base station by

[1] Inter-flow scheduling schemes (e.g., fair queuing or flow prioritization [12, 25, 58, 62, 65, 66]) can provide isolation across flows, but minimizing the self-inflicted queuing delay for a given flow still requires precise congestion control.

strategically dropping packets (as specified by the application) to match the available capacity and minimize delay.

Our approach is motivated by two observations: *(i)* 5G applications, transmitting real-time (standard or volumetric) video streams, can typically adapt their content (e.g., frame rate or quality) based on available network capacity to achieve low message latency. The state-of-the-art real-time video transmission schemes (e.g., [22, 33, 43, 79, 80]) require the sender to first use a feedback-based controller to estimate network bandwidth, with the sender-side application then adapting its content based on this estimate. Our approach, in contrast, enables such content adaptation to take place directly inside the network. *(ii)* The ongoing evolution of the 5G infrastructure [8, 59] provides an opportune time to study how smarter mechanisms implemented at the network edge (base stations) can help tackle the extreme challenges of high-performance data transfer over time-varying 5G links.

So how do we go about enabling in-network content adaptation for 5G applications? First of all, it requires the application (app) to encode its real-time data stream, comprising a series of multi-packet *messages* (e.g., video frames), in a way that supports content adaptation via packet drops. As detailed in our case studies (§6), existing stream encoding techniques (e.g., scalable video codec [5, 64, 72] and point clouds for volumetric videos [39, 51]) already provide such adaptability. The in-network packet dropping logic would then depend on the app's requirements and how it encodes the stream. For instance, some apps may support reducing the spatial resolution (e.g., by dropping packets corresponding to higher quality layers in a layered video stream [5, 64], or to higher density levels in a point cloud [39]), whereas others may solely allow reducing temporal resolution (e.g., by dropping certain frames).

This leads us to the following question: how do we support different packet dropping policies that may vary across applications? Since it might not be practical and scalable for 5G base stations to implement customized dropping logic specific to each app, we look for generalized dropping *primitives* that can be configured differently by different apps to express their requirements. The mode of configuration we adopt involves parameters that can be specified in packet header fields.

We first considered using well-known queue management techniques that seemingly provide such configurability. One option is to use priority dropping [4, 18], where the app marks different packets with different priorities, and the router drops lower priority packets when the buffer is full. Another option is to tag different packets with deadlines, and the router drops the packets that exceed their deadlines [73]. However, it is not immediately obvious how an app could use these schemes to express its content adaptation policies which often involve complex dependencies across frames. For instance, if the app requires minimizing the latency of the latest message, how do we tune the buffer threshold for priority dropping or the packet deadlines, the optimal value of which would vary with message sizes and network bandwidth.

Instead, we design parameterized dropping primitives to more directly capture the requirements of real-time apps. Our primitives drop packets at the granularity of multi-packet messages, where the app specifies the message boundaries and expresses its dropping policy as per message parameters. Our primitives trigger message drops based on two natural conditions: *(i)* the arrival of a new message, where the app can specify which messages trigger a drop in which subset of older queued-up messages using priority levels, and *(ii)* when the link capacity falls below specified bitrate thresholds, where the app can configure different thresholds for different messages based on the (known) bitrate of the generated content. Apps can configure these primitives by setting message parameters to express different content adaptation policies, in a manner that is agnostic of underlying network characteristics.

We build a system, Octopus (§4), that is centered around the above primitives and comprises: *(i)* an interface for applications to specify their message boundaries and per-message parameters to configure the dropping primitives; *(ii)* a transport protocol that encodes the app-specified message parameters into packet headers, performs (imperfect) congestion control that errs on the side of over-estimating the network capacity, and implements the parameterized dropping primitives for content adaptation in the transport buffer; *(iii)* an in-network buffer management scheme that implements the dropping primitives and parses the parameters in packet header fields to enforce app-specified content adaptation policies.

We prototype our system (§5) using UDT [36] (a user-space transport framework) for the endhost logic, and srsRAN [34] (an open-source cellular platform) for the in-network logic. We evaluate our prototype using three different case studies involving real-time standard and volumetric videos (§6), and across a variety of scenarios (§7), to show that our approach results in 1.5–18× better performance than state-of-the-art schemes (including WebRTC [7], AWStream [79], Salsify [33], and ViVo [39]).

We acknowledge that our system cannot be *immediately* deployed as it requires changes at both the endpoints and the cellular base stations. However, we believe that our approach presents an interesting design point that is worth exploring, given the increasing volume and significance of real-time streaming [9, 10, 38] over 5G that is well-suited to in-network content adaptation, the increasing flexibility of modern cellular infrastructure [77], and the scope for significant improvement in performance (as promised by our evaluation).

2 MOTIVATION

The cellular link from the base station to the user device is often the bottleneck for data transmission [14, 74, 82]. These links are prone to bandwidth (capacity) variation [23, 35, 74, 78], triggered by factors such as dynamic obstacles, and changing directionality or distance as a device moves. More specifically, such factors lead to variations in wireless signal quality between the base-station and user device, which then translate to variations in available network capacity. The higher directionality required by higher frequency bands (e.g. mmWave) makes 5G even more susceptible to such effects [54, 76].

We conduct a series of experiments to evaluate how the fluctuations in cellular link capacity impact the performance of feedback-based congestion controllers. We used Mahimahi to emulate a cellular link with time-varying bandwidth drawn from a Verizon trace [56]. We fixed the round-trip time (RTT) to 60ms, and the buffer size to 375 KB. A sender attached to the emulated link sends backlogged data to a receiver, using different congestion controllers.

Prior works fail to achieve both high link utilization and low queuing delay. Figure 1(a) shows that TCP BBR [21] (designed

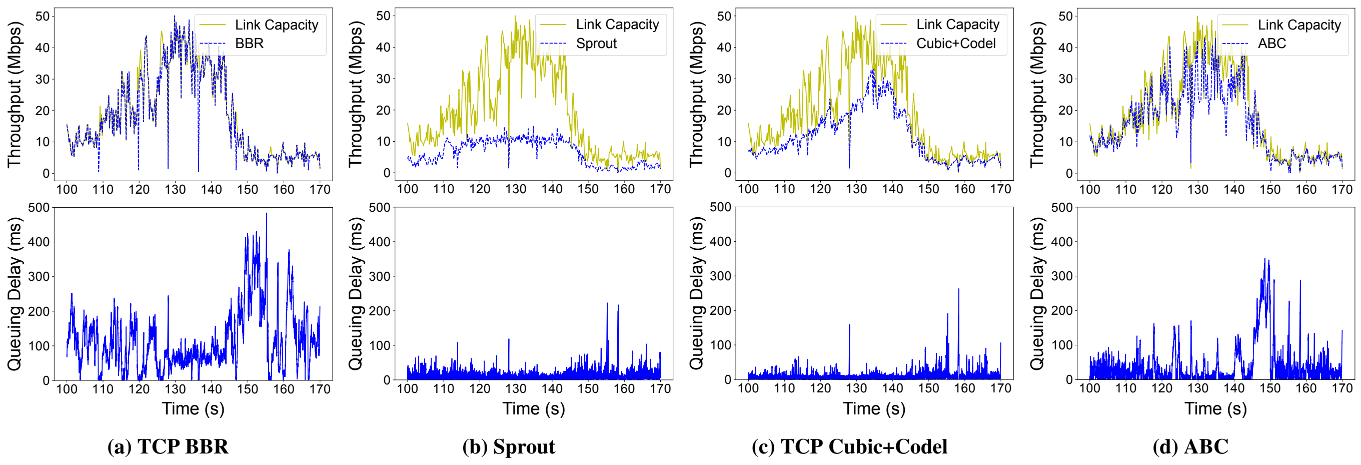


Figure 1: Throughput (top) and queuing delay (bottom) for different protocols on a link emulating the Verizon download trace with an RTT of 60 ms.

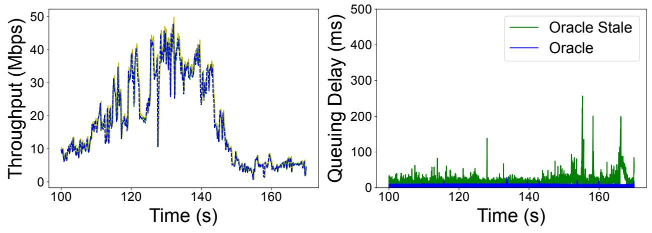


Figure 2: Throughput and queuing delay for Oracle on a link emulating the Verizon download trace with an RTT of 60 ms. Queuing delay spikes when the sending rate is based on Oracle knowledge of link capacity that is stale by 5ms.

for WAN) achieves high link utilization, but also results in queuing delays as high as 500 ms. We observe similar results with TCP Cubic (not shown for brevity). This led to the development of congestion controllers for cellular networks that react faster to changing network bandwidth [35, 74, 78]. Our experiment with Sprout [74] revealed that it can be too cautious, which results in under-utilization of the link. Even so, it cannot avoid spikes in delay when the bandwidth suddenly plunges (Figure 1(b)). Leveraging active queue management (AQM) for congestion control, such as Cubic+Codel and ABC [35], exhibit similar issues of link under-utilization and delay spikes (Figure 1(c,d)).

We cannot practically design a perfect congestion controller. The above are only a few samples from the vast repertoire of congestion control algorithms designed for wide-area and cellular networks [21, 27, 28, 35, 37, 52, 74, 75, 78]. This raises the natural question of whether a different controller would have perfectly achieved high link utilization and consistently low queuing delay. Rather than taking up the insurmountable task of experimenting with all proposed congestion controllers, we instead exploited our emulation environment for a simple exercise. We implemented an "Oracle" that used the knowledge of the emulated bandwidth trace to compute the number of packets that the link can serve over each period of 5 ms, and sent precisely that many packets during the corresponding time period. We observed that this indeed resulted in perfect behavior – maximum link utilization with consistently low queuing delay below 30 ms (as shown by the blue line in Figure 2).

We then added a small offset, making the sending rate stale by 5 ms relative to the link capacity. This staleness models real-world scenarios where an endhost can only receive network feedback after certain inevitable delay.

The green line in Figure 2 shows the resulting performance with such a "stale Oracle". We find that even with an offset as small as 5 ms, our Oracle was unable to avoid spikes in queuing. The queue keeps accumulating packets that arrive 5 ms late during a bandwidth reduction (thus missing their chance to be transmitted), until the next increase in bandwidth allows transmitting them.

These results show that spikes in queuing delay cannot be avoided without knowledge of *(i)* precisely how long a packet would take to reach the base station, and *(ii)* the available bandwidth at that (future) time, which may change due to external factors such as sudden obstacles and mobility. Since it is not possible to obtain such information in practice, we do not expect any feedback-based congestion controller to behave perfectly.

Our approach. Given the pessimistic results above, how can we meet the stringent throughput and latency requirements of networked real-time applications? Our system, Octopus, side-steps the problem of designing a perfect feedback-based congestion controller by using a reactive approach that exploits the content adaptability of real-time streams. Octopus sends data using the BBR congestion control to achieve high link utilization, and then drops appropriate packets in the cellular network buffers to match the actual available link capacity and minimize queuing delay. We design parameterized primitives for implementing the dropping logic, that the applications at the endhost can configure differently to express different content adaptation policies. Octopus transport encodes the application-specified parameters in packet headers, which the routers can parse to execute the desired dropping behavior. We provide a detailed description of Octopus' design in §4.

3 RELATED WORK

Video adaptation at endhosts. State-of-the-art schemes for real-time video streaming adapt quality or frame rate at the endhosts, relying on a feedback-based controller to estimate network capacity [22, 33, 43, 45, 50, 79]. For example, WebRTC [7] configures the video bitrate based on the network capacity estimated by GCC (Google Congestion Control) [22]. Salsify [33] (that outperforms WebRTC, Skype, Hangouts, and FaceTime) relies on bandwidth estimated by Sprout [74] (a feedback-based controller for cellular networks) to adjust the size of the transmitted frame. AWStream [79] adapts streamed content (e.g., video frame rate and/or resolution) based on the bandwidth estimated by the underlying TCP controller (Cubic or BBR). Some of the recent proposals for real-time video transmission (OnRL [81], Loki [80], and Concerto [83]) use data-driven techniques to configure the real-time video bitrate at the endhost based on network feedback. There are also schemes that use SVC [63, 72] and drop higher quality layers at the endhost based on the bandwidth estimated by a feedback-based controller. As we show in §2, inaccuracies in bandwidth estimation at the endhost often result in sub-optimal throughput or higher latencies. Octopus adopts an alternative approach of adapting the transmitted real-time content in cellular router buffers, rather than relying on precise bandwidth estimation at the endhosts.

Packet scheduling. Scheduling packets *across* different flows (e.g., to achieve fairness or small flow completion time) [12, 25, 62, 65, 66, 69] is orthogonal and complementary to our goals of ensuring optimal throughput and latency for a given real-time flow. In the context of intra-flow scheduling, recent work [16, 47] analyzes the benefits of using LCFS (last-come-first-serve) for maximizing freshness at per-packet granularity. Since a real-time message is typically consumed by the receiver as soon as it is received, there is little value of delivering an older message out-of-order, after a later message in the stream has been delivered. Therefore, rather than determining the packet scheduling order, the key question we consider is whether a message in a given real-time stream should be transmitted by the router or dropped altogether. We enforce our (more general) dropping primitives at the granularity of multi-packet messages.

In-network dropping policies. CoDel [57] and RED [32] proactively drop packets on the onset of congestion to send early signals to endhost congestion controllers (§2 shows how this can lead to link under-utilization). Octopus, instead, drops packets to directly adapt the transmitted content based on the network conditions and app-specified requirements.

Bhattacharjee et al. proposed intelligent packet discard for MPEG transfers [17, 18] as a use case of active networking, where lower priority video frames are discarded when the buffer overflows. Octopus, in contrast, minimizes latency by proactively discarding messages (before the buffer fills up) using more direct triggers in the form of new message arrivals and instantaneous link capacity. We evaluate the significance of doing so in §7. Bhattacharjee et al.'s proposal further required the routers to maintain app-specific logic. In contrast, Octopus routers are app-agnostic, and implement parameterized primitives to enforce app-specified policies.

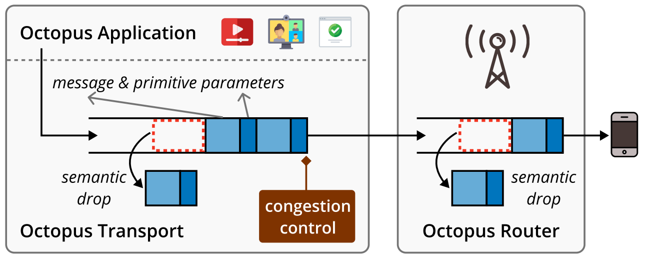


Figure 3: Overview of Octopus

4 OCTOPUS DESIGN

As depicted in Figure 3, Octopus is a cross-layer system consisting of the following elements.

Octopus-aware Application. An app specifies its desired dropping policy (as per its requirements and stream encoding format) to the underlying Octopus transport in the form of parameters for Octopus' *dropping primitives* (detailed in §4.1). The app specifies these parameters for individual *messages* (where a message may contain one or more packets, and is the unit of packet drops in Octopus).

In order to fully exploit Octopus, a real-time stream must be able to tolerate in-network packet drops. For instance, in a single-layered video codec that uses every single frame as a reference to encode the next frame (as in H.265 [67] and VP9 [2]), a drop in one frame will disable decoding all subsequent frames. We show how existing multi-layer extensions of such codecs (e.g., SVC [3, 64]) can be effectively used with Octopus in §6.

Octopus Transport. The underlying transport (detailed in §4.2) encodes the dropping policies (that the app specifies in the form of per-message parameters) in individual packet headers. It paces out the data packets sent into the network using BBR's congestion control logic. This ensures that the data transfer rate is limited to the peak capacity of the bottleneck cellular link, and that the Octopus traffic competes fairly with the rest of the wide-area traffic. If the pacing rate is slower than the rate at which an app generates data, messages get queued up in the transport buffer. Octopus implements its message-dropping primitives at the transport buffer (as per the app-specified parameters). This direct and timely content adaptation in the transport buffer mitigates the need for additional app-layer content adaptation strategies that are based on feedback from the transport.[2] It acts as a first-level of content adaption at the endpoint itself, and is sufficient if the network bandwidth is stable and matches the pacing rate. It also extends the utility of Octopus to scenarios where the bottleneck is the uplink from the device to the base station. [3]

Octopus logic at the base station. Given the inherent inaccuracy in estimating volatile link bandwidth and the relatively aggressive congestion controller (BBR) used by Octopus, the pacing rate at an

[2]We provide a direct comparison of Octopus' endpoint content adaptation (without the in-network logic) with other endpoint-based solutions in §7.1.

[3]To handle uplink bottlenecks with Octopus, the endhost can use a back-pressure based mechanism (similar to TCP small queues [30]) to restrict the transport from sending more data when the (small) NIC buffer is full. The Octopus logic implemented at the transport buffer will then appropriately drop messages. We leave a detailed implementation of this to future work, and limit the scope of our implementation and evaluation to scenarios where the bottleneck is at the downlink from the base station to the users.

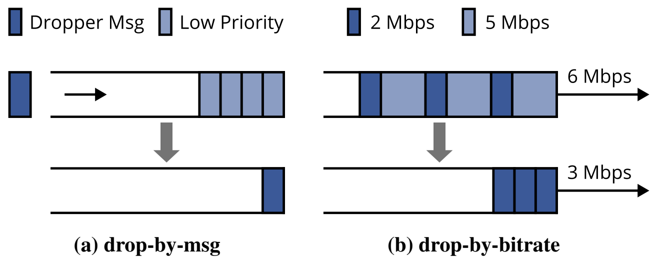


Figure 4: Two dropping conditions in Octopus

endpoint might be higher than what the cellular link can support. Octopus's base-station logic (§4.3) kicks in to adapt stream content and minimize in-network delays in such cases. It implements the Octopus dropping primitives, parses the packet headers to read app-specified parameters, and enforces the desired dropping behavior.

4.1 Dropping Primitives

A real-time data stream consists of a series of temporal data units, each corresponding to a point in time. Generalizing the terminology used for video streams, we refer to each such temporal unit as a frame. Each frame has a base layer at a specific quality level, and may optionally have additional quality-enhancing layers. If the link bandwidth is larger than the incoming rate of the stream, the link can sustain the stream and no frames get queued up. If not, multiple frames get queued up at the link buffer, and we need to adapt the buffered content by dropping packets in order to sustain the stream. There are two ways in which the content of real-time streams can be adapted: *(i)* reducing the temporal resolution (or the frame rate) by dropping entire frames, and, *(ii)* reducing the spatial resolution (or data quality) by dropping one or more quality enhancing layers within a frame. Such content adaptation must further take the stream encoding (or the frame dependency structure) into account—if a "reference" frame/layer is dropped, subsequent frames/layers that semantically depend on it cannot get decoded.

We define a *message* as the atomic granularity at which Octopus drops data packets in a stream. A message would correspond to an entire frame if the content is adapted solely via reducing the temporal resolution. In streams that also support adapting spatial resolution, a message would correspond to a quality layer in each frame. An app identifies its message boundaries, and specifies its dropping policies (as per its requirements and stream encoding format) in terms of per-message parameters supported by Octopus' dropping primitives. We now detail these primitives.

Dropping Condition 1: Newer Message Arrival. In our first primitive, the arrival of a newer (more important) message triggers dropping (a subset of) staler queued-up messages. Octopus tags each message with app-specified *(i)* priority value (*msg_priority*), where a higher value implies lower priority, and *(ii)* a *drop_flag* with an associated *priority_threshold* value. When a new message is enqueued at the buffer, if its drop_flag is set, it triggers dropping all previously queued up messages in the same stream which have msg_priority $\geq$ priority_threshold. We refer to the messages with the drop_flag set as *dropper messages*, and refer to this primitive as *drop-by-msg* primitive.

An app can configure the message boundaries and the primitive's parameters to specify which messages' arrival will result in the drop of which subset of queued messages. For streams with independent frames (e.g., a stream of images), simply setting the drop_flag in each message (frame) will maximize freshness. If a stream requires fixed temporal resolution but allows tuning spatial resolution, the app can configure the message boundaries and parameters to only drop queued-up quality-enhancement layers when a new frame arrives. Alternatively, depending on the stream encoding, an app can configure the parameters to avoid dropping a reference frame upon the arrival of a new frame that depends on it.

In general, a queue build-up of two or more frames is a good indicator of the inability of the link to sustain the incoming stream. The drop-by-msg primitive is designed to immediately react to such queues in order to minimize queuing delay. However, small transient queues of multiple frames may build up when there is a mismatch between the average and the instantaneous rate of the incoming stream due to differences in frame sizes. Our second primitive is designed to provide increased tolerance towards such transient queues.

Dropping Condition 2: Bandwidth Lower than Data Bitrate. In certain stream encodings, the frame sizes may differ significantly. For example, in a layered video codec, the size of the initial reference frame within each "group of pictures" (on which subsequent frames in the group depend) can be 4× larger than the non-reference frames. Even if the link bandwidth is sufficient to sustain the average bitrate across several frames, a transient queue may accumulate while serving the larger (reference) frames. If the dropper messages in drop-by-msg primitive are spaced very closely, the quality-enhancement layers in the reference frame might be dropped in the transient queue, leading to subsequent dependent frames being decoded at the lowest quality level (even though the link bandwidth is sufficient to sustain the average bitrate of high-quality layers). On the other hand, if the dropper messages are spaced too far apart (or configured to avoid dropping reference layers), and the link bandwidth is indeed lower than the average bitrate of higher-quality layers, there is a chance that, by the time a suitable dropper message arrives, the higher-quality layers (that should have been dropped) have already exited the queue after having contributed to a significant queuing delay for the remaining frames.

To handle such scenarios, we additionally need a dropping condition that is directly based on the available link bandwidth, and does not rely on subsequent dropper messages. Our second primitive provides this.

An app can tag each message with a *bitrate_threshold* (using its knowledge of the stream encoding and the data rate that the app explicitly configures) Octopus drops a message if the stream is currently being served at a bandwidth *BW* that is less than its bitrate_threshold. We refer to this primitive as *drop-by-bitrate* primitive. It allows an app to specify different bitrate_thresholds for different spatial layers in the stream, thus enabling the Octopus buffer to directly determine whether a spatial layer should be dropped or transmitted based on the available link capacity.

Summary. Octopus supports two conditions to drop messages. The combination of the two primitives provides an expressive mechanism for real-time apps to specify different content adaptation policies. We exemplify their usage in §6.

4.2 Transport Design

API. An app conveys its dropping policies via Octopus' transport interface. In particular, the app conveys its atomic message boundaries to the transport, and for each message, specifies its stream_id (which explicitly identifies the stream that the message belongs to), msg_priority, and the dropping parameters (that include the drop_flag, the associated priority_threshold, and the bitrate_threshold).

By default, Octopus transport sets all dropping parameters in a message to zero, if not explicitly configured by the app, that disables any content adaptation.

Encoding dropping policies. Octopus transport tracks per-stream message sequence space. Upon receiving a new message from the application, it increments the corresponding msg_id counter, packetizes the data, and encodes the msg_id, priority, and parameters in designated packet header fields. The header fields additionally carry information about whether the packet is the first, last, or only packet of the message.

Transport buffer management. Similar to TCP, Octopus transport enqueues the packets in a send buffer, until they can be sent out to the lower layers. As mentioned earlier, it enforces the same message-dropping policies in this transport buffer as the one enforced by the base station (described in more detail in §4.3). It uses the bandwidth estimated by the congestion control algorithm as the currently available bandwidth for the drop-by-bitrate primitive.

Loss handling and message delivery. Octopus transport is unreliable, in line with the requirements of real-time apps (future extensions can support partial reliability). The receiver acknowledges received data (required for congestion control), although no packets are retransmitted. As soon as a message is completely received, the Octopus receive-side transport delivers it to the application, irrespective of whether prior messages have been delivered.

Congestion control. Octopus requires congestion control to ensure that the data transfer rate is limited to the peak capacity of the bottleneck cellular link, so as to not overwhelm the rest of the network. Moreover, there might be scenarios where the bottleneck lies elsewhere in the network (e.g., at a legacy switch that does not support Octopus), requiring Octopus to compete fairly with the cross-traffic. We incorporate BBR's congestion control logic in Octopus transport. BBR, by design, tracks the maximum packet bandwidth and the minimum RTT over a configurable period of time, and uses this to compute the delivery rate and the congestion window. This enables BBR to continue sending at the peak cellular capacity, without getting perturbed by transient dips in available bandwidth. As shown in §2, this does come at the cost of high queuing delays which we handle through our in-network dropping policies.

4.3 Base-station Logic

Octopus logic at the cellular base station enforces the dropping primitives, as per app-specified policies encoded in packet headers. Cellular base stations already maintain separate queues for individual users [20, 34, 74]. We further assume that a user's real-time traffic is isolated from their non-real-time traffic (standard mechanisms for doing so already exist in cellular networks [11, 20]).

Algorithm 1 Octopus' packet dropping logic

```
variable dropper_msgs_, msg_in_drop_
procedure ENQUEUE(packet)
    sid ← packet.streamID()
    if packet.hasDropFlag() and packet.isTail() then
        threshold ← packet.priorityThreshold()
        dropper_msgs_[sid][threshold] ← packet.msgID()
    end if
    buffer_.push(packet)
end procedure
procedure DEQUEUE(void)
    packet ← buffer_.pop()
    msgid ← packet.msgID()
    sid ← packet.streamID()
    if packet.isHead() then
        prio ← packet.priority()
        latest_dropper ← max(dropper_msgs_[sid][0],
            ..., dropper_msgs_[sid][prio])
        isdrop ← msgid < latest_dropper
            or packet.bitrateThreshold() > BW[sid]
        if isdrop then
            msg_in_drop_[sid] ← msgid
        end if
    end if
    if msg_in_drop_[sid] = msgid then
        return Drop(packet)
    end if
    return packet
end procedure
```

Octopus requires the base station to track the available bandwidth BW for each (per-user) queue. This is often directly available in cellular links [1, 35]. It can also be measured at the router by tracking the rate at which packets are dequeued and transmitted, as we do in our experiments (§5). If a given user downloads multiple real-time streams, the router computes the max-min fair rate BW_i for each stream i from the observed per-stream arrival rate and the overall rate BW at which the user's queue is served.

Algorithm 1 shows the pseudocode for the packet-dropping logic. For each stream, the Octopus base station logic maintains a table indexed by the priority_threshold that records the msg_id of the latest dropper message corresponding to that threshold. It updates this table when enqueuing the tail packet of a dropper message. When the head packet of a message from stream i is dequeued, the Octopus logic marks this message to be dropped if: *(i)* its bitrate_threshold is higher than BW_i, or *(ii)* its msg_priority is greater than or equal to the priority_threshold of all dropper messages in the queue that belong to the same stream. If a message is marked for drop, all the following packets belonging to it are dropped during dequeuing. To avoid starvation, Octopus does not drop a message that is in transmission (i.e., one or more of whose packets have been transmitted).

Notice that Octopus requires maintaining a very modest amount of per-stream state at the base station, which remains constant with the number of packets or messages. It only grows with the number of priority threshold levels in a stream; we expect this number to

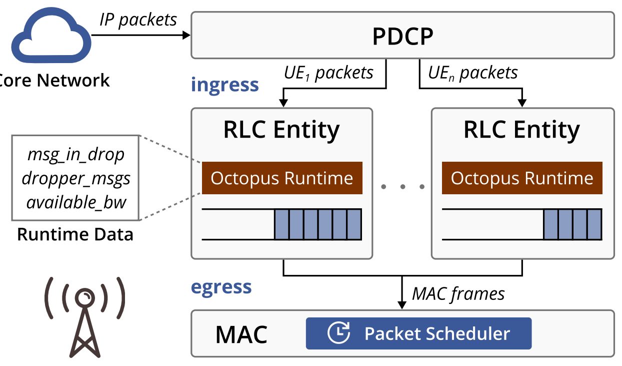


Figure 5: Base-station (eNB/gNB) protocol stack with Octopus

be small for most use cases (§6), and it is restricted to 8 in our prototype.

5 PROTOTYPE IMPLEMENTATION

Octopus Transport. We implement Octopus transport in 3000 LoC by extending the UDT framework, a user-space transport protocol over UDP [36]. This includes implementing BBR's congestion control logic. Our prototype uses UDT's app-layer header to encode message properties and dropping policies, which take 12 bytes. An actual deployment can instead make use of IPv4 options or IPv6 extension fields to encode Octopus parameters.

Octopus Logic at Base-station. We implement Octopus' in-network logic in srsRAN [34, 68], an open-source software cellular platform for 5G and LTE radio access networks (RAN). Our changes to srsRAN fit within 420 LoC. We deploy the srsRAN platform on a server with eight Intel Xeon E5 cores.

The network protocol stack of the 5G/LTE base station in srsRAN is shown in Figure 5. Specifically, the RLC (radio link control) layer instantiates an entity to manage an isolated logical queue for each connected user. We implement Octopus' dropping primitives and maintain runtime data in the RLC entity. In the ingress stage, the RLC entity parses the app-layer headers of incoming packets, and accordingly updates the table of the latest dropper messages. In the egress stage, Octopus determines whether the current message is to be dropped based on the dropping conditions, and drops packets belonging to it in that case. For the drop-by-bitrate primitive, our implementation tracks the bandwidth availability by computing the dequeuing rate over a sliding time window of 50 ms (discounting the idle periods when the queue is empty).

Octopus' in-network logic is light-weight and introduces negligible latency overhead. To quantify the processing delay in srsRAN, we sent video packets (with headers carrying Octopus' parameters) over an emulated network link at a small rate of one packet per second, and measured the round-trip time (after discounting the propagation delays) with and without Octopus' in-network logic. Without Octopus' in-network logic, the average processing delay was 3.53ms. Enabling Octopus' in-network logic increased the average processing delay to 3.59ms, thereby resulting in an overhead of 1.7%.

6 CASE STUDIES

We evaluate Octopus using three case studies involving real-time video streaming with frame rate adaptation (§6.1) and quality adaptation (§6.2), and live volumetric video streaming (§6.3). We focus on real-time 2D/3D video streaming in our case studies due to their relative popularity across apps (in conferencing, gaming, VR, surveillance, robotics, etc.) and due to the existence of comparative baselines. One can design similar policies to exploit Octopus for other forms of real-time streams. We start with presenting our basic comparative results in this section, followed by a more in-depth evaluation in §7.

Experiment Testbed. Our testbed involves a sender and a receiver node (both running Octopus transport on UDT) communicating via the srsRAN platform. We emulate cellular link bandwidth and delay on the downlink from the base station to the receiver.[4] We set the RTT to 60ms (we also present results with 120ms RTT in §7). For the first two case studies, we experiment with bandwidth traces from three different LTE cellular providers (Verizon, T-Mobile, and AT&T) [74]. The video sources are five different videos taken from MOT16 and MOT20 datasets [26, 53].

For the third case study involving volumetric videos, we experiment with two different 5G cellular download traces [60]. To support higher 5G data rates in this case study, we replace our srsRAN platform with Mahimahi network emulator [56], and implement Octopus' router logic in Mahimahi. [5] The volumetric video sources are three videos in point cloud format taken from CMU panoptic dataset [46].

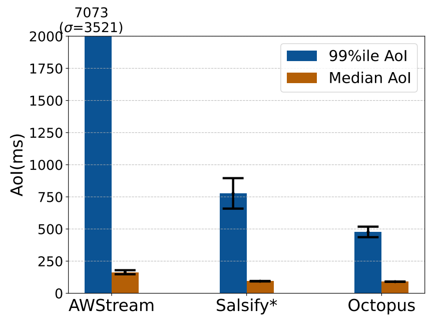
6.1 Real-time Video with Frame Rate Adaptation

Our first case study considers the requirement where the freshest video frame must be delivered at fixed quality (e.g., a gaming app that requires high responsiveness without compromising on quality, or apps that process the received video using an ML algorithm).

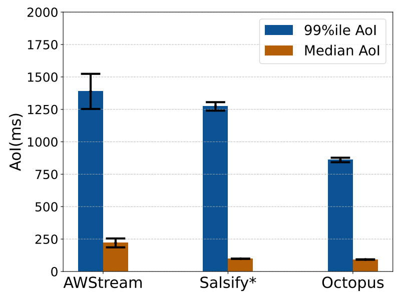
Video Encoding. Commonly used video codecs (e.g., VP8 [15] and VP9 [2]) encode video frames in chunks called "group of pictures" (GoP). As shown in Figure 7 (top), the first frame in each GoP (the I-frame) is intra-coded and can be decoded independently. Each subsequent P-frame only encodes the delta from the previous frame. The successful decoding of a P-frame at the receiver therefore requires the successful delivery of all previous frames in the GoP. This limits the tolerance to in-network packet drops.

To better exploit Octopus, we make use of multiple temporal layers supported by scalable video codec (SVC) [3, 5, 64]. We apply VP8-SVC with three temporal layers. Figure 7 (bottom) shows the dependency structure between frames. Frames marked with priority value i serve as reference only for frames with priority value $j > i$, and can be dropped upon the arrival of a new frame with priority_threshold $k \leq i$, without affecting any subsequent frames. The usage of VP8-SVC introduces slight overhead—the average frame size is 15%–18% larger than that encoded in default VP8. Our results show how Octopus improves overall performance in spite of these overheads, in comparison with baselines that use the default codec.

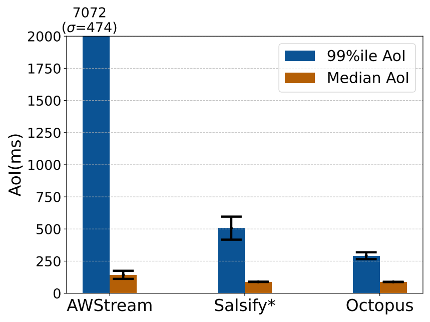
[4]For this, we use the cellular downlink traces [74] as input to set the maximum MAC frame size in srsRAN every TTI.
[5]srsRAN platform can support a data rate of up to 75 Mbps.



(a) AT&T



(b) T-Mobile



(c) Verizon

Figure 6: Median and 99%ile AoI in different LTE download network traces with an RTT of 60 ms (averaged over 5 videos, with error bars showing the standard deviation).

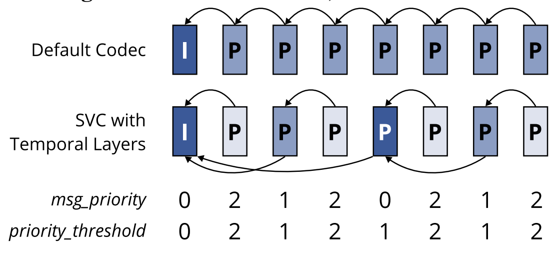

Default Codec
SVC with Temporal Layers
msg_priority 0 2 1 2 0 2 1 2
priority_threshold 0 2 1 2 1 2 1 2

Figure 7: Single-layered video codec (top) and SVC with three temporal layers (bottom).

Dropping Policy. Using the video codec described above, we treat each frame as a single message and only make use of the drop-by-msg primitive. We set the drop_flag in every message. Figure 7 shows the msg_priority, and priority_threshold that we set for each message. The bitrate_threshold is set to the default value of zero to disable the drop-by-bitrate primitive.

Baselines. We compare Octopus against two state-of-the-art baselines for real-time video streaming: *(i)* AWStream [79], configured to adapt frame rate based on the bandwidth estimated by the underlying transport (TCP BBR). We use the default VP8 encoding for the video. *(ii)* Salsify [33], modified to solely tune the frame rate keeping the frame quality fixed (we refer to this as Salsify*). As in the original Salsify design, it uses a functional codec based on VP8 encoding. We encode the videos for each baseline and Octopus with a fixed quantization level of 17.

Metric. We use "age of information" (AoI) [49] to capture freshness as the time elapsed since the latest frame delivered at the receiver application was sent out by the sender application (lower AoI implies higher freshness). To capture the worst-case freshness, we measure the AoI just before a frame is received, and compute the 99%ile and median values across all frames in a video. AoI helps to capture both frame rate and latency—the AoI value can be high either due to high queuing delays or low frame rate (or both). We also report the median and tail per-frame latency in the Appendix.

Results. As shown in Figure 6, Octopus outperforms AWStream and Salsify*, in spite of using a less efficient encoding strategy. The tail AoI with Octopus is 1.6–18× lower than that of AWStream. We found that AWStream (designed for more stable WAN bandwidth) is overly conservative in increasing its sending rate (upgrading to a higher throughput configuration only after its application buffer has been empty for 2 seconds). The tail AoI with Octopus is 1.5–2× lower than Salsify*: Salsify* suffers from higher tail latencies and lower link utilization due to a slower reaction to changing network capacity as compared to Octopus.

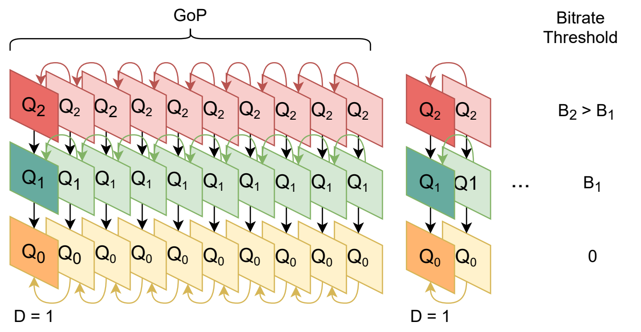


Figure 8: SVC encoding with three quality layers, and the dropping primitive parameters.

6.2 Real-time Video with Quality Adaptation

Many real-time video apps (e.g., video conferencing and live streaming) allow tuning the quality of the video content to sustain a stable frame rate with low latency. Our second case study considers this requirement.

Video Encoding. We exploit support for multiple spatial (quality) layers in SVC for quality adaptation with Octopus.[6] Figure 8 shows how frames are encoded in SVC with 3 spatial layers (depicted as Q_0, Q_1, and Q_2 from lowest to highest). The decoding of a higher-layer frame depends on the successful decoding of all lower layers and the corresponding layer of the previous frame.

In our SVC-based Octopus application, we fix the number of quality levels to 3, and use the bandwidth estimation (B) provided by the underlying Octopus transport to dynamically adjust the encoding quality levels for each GoP. Specifically, quality Q_2 is configured such that its cumulative target bitrate is $B_2 \approx B$, Q_1 is configured to a cumulative bitrate of $B_1 \approx 0.5B$, and Q_0 is configured to a bitrate $B_0 \approx 0.2B$. We reduce the GoP size to 10 frames, which has a slightly

[6]Future work can explore using other layered codecs, e.g., neural video codec [24].

higher bandwidth overhead but allows faster switching across quality levels.

We use a publicly available single-threaded software encoder (libvpx [3]), which has a high latency overhead. We therefore pre-encode each video for our experiments (as discussed in §8, we expect real-time encoding overhead to be lower in practice, with more sophisticated multi-threaded or hardware-based codecs). We use the three-layered SVC to pre-encode the videos with 9 different base quality levels. At runtime, we dynamically switch between these levels, such that the highest layer of the chosen level best matches the bandwidth estimated by the underlying Octopus transport.

Dropping Policy. We use the drop-by-bitrate primitive, treating each layer of each frame as an individual message. We mark the messages corresponding to quality Q_0, Q_1, and Q_2 with bitrate_thresholds of zero, B_1 and, B_2 respectively.

Overall, this policy achieves the desired trade-off in quality, throughput, and message latency. However, it has a few caveats: *(i)* It may result in high queuing when the bandwidth is lower than B_0 (the required bitrate for the lowest-quality frames). We handle this by using our drop-by-msg primitive and setting the drop_flag in the first Q_0 message in each GoP. With a priority_threshold of zero, this triggers a drop in all queued-up frames of the previous GoP. *(ii)* There might be scenarios where the bandwidth increases after some of the Q_2 or Q_1 messages towards the beginning of a GoP have been dropped. Transmission of Q_2 messages in the GoP that are then enqueued is wasteful as they cannot be decoded at the receiver. Nonetheless, we observe that if the bandwidth is high enough to sustain their bitrate, transmission of these messages does not generally block transmission of newer frames in the stream. Moreover, by limiting the GoP to 10 frames, we limit the scope of such wasted frame transmissions.

Baselines. We compare Octopus for this use case with: *(i)* WebRTC [7], which uses GCC [22] as the underlying congestion controller, *(ii)* Unmodified Salsify.

Metric. We measure the video quality using SQI-SSIM [29, 61]—it computes the quality of each decoded frame using structural similarity (SSIM) [42, 84], and the quality of each undecoded (or dropped) frame as the product of the SSIM of the last available frame and an exponential decay function. The resulting video QoE score is the average quality across each (decoded and undecoded) frame that was sent by the sender app. We also record the latency of each delivered frame. It is desirable to achieve high SQI-SSIM and low latency.

Results. As shown in Figure 9, Octopus achieves higher average SQI-SSIM with lower tail latency than the baselines, while the median latency is similar across different schemes. Octopus achieves 41–57% higher SQI-SSIM than WebRTC, with 2-10× lower tail latency. The difference in SQI-SSIM stems from WebRTC's conservative behavior when upgrading to a higher throughput configuration. WebRTC's tail latency is relatively high due to poor reaction when link capacity suddenly drops to very low values. Octopus achieves 16% higher SQI-SSIM than Salsify, with 1.6–16× lower tail latency. The difference in SQI-SSIM largely stems from the inefficiency of the functional VP8 codec used in Salsify. The difference in tail latency stems from Octopus' faster in-network adaptation to sudden bandwidth drops.

6.3 Real-time Volumetric Video Streaming

Real-time volumetric streaming enables viewers to watch videos in six degrees of freedom (6DoF), and is becoming popular in education, entertainment, and healthcare. We look at how Octopus can adapt the quality of bandwidth-intensive volumetric streams to minimize delay while trying to sustain a stable frame rate.

Video Encoding. Uncompressed volumetric video frames are represented as point clouds that record attributes such as coordinates and colors of every point. A frame is spatially segmented into multiple cells, and each cell can be independently encoded and decoded [39, 51]. This segmentation enables the server to stream a subset of cells or adapt the point density level (PDL) of cells based on the viewer's current viewport to significantly reduce network bandwidth requirements.

Figure 10 shows how we encode a volumetric frame. In our experiments, every video frame is segmented into four cells. Instead of encoding every cell with a specific PDL, we further divide the origin cell into five layers, each comprising 20% points, and encode every layer independently so that Octopus can safely drop specific layers in a cell to adapt its PDL. This results in 10%–12% overhead in bandwidth usage which, as our results show, is more than compensated by Octopus' fast reaction to bandwidth changes compared to the baseline.

Dropping Policy. Based on the Occlusion Visibility (OV) and Distance Visibility (DV) adaptive streaming approaches from ViVo [39], we first drop layers of occluded cells, and then drop higher density layers of non-occluded cells, as queues start building up. For simplicity, we assume the viewport is fixed, where cell 0 and cell 1 serve as front cells, while cell 2 and cell 3 are occluded cells (we can dynamically update the cell's priority for the changing viewport using the viewport movement and prediction model from ViVo).

We apply drop-by-msg primitive, and treat each layer inside a cell as an individual message. The message priority for each such layer is indicated in Figure 10. We set the drop_flag in the first layer of the non-occluded cell 0 in every frame. The drop_thresholds in these dropper messages are set to a repeating pattern of 3, 2, 1, and 0, which enables the smooth adaptation of cell PDLs in the buffer.

Baselines. We compare Octopus with ViVo in this use case. Since the source code of ViVo is unavailable, we implemented a version to our best knowledge. Here, the ViVo client uses a throughput-based rate adaptation algorithm [44] to estimate the link capacity and determine the PDL of occluded cells and front cells to fetch. We use an open-sourced software encoder (Google Draco[13]) to pre-encode the volumetric video frames both for ViVo (at different PDL values) and for Octopus (with the encoding strategy modified to incorporate PDL layers, as described above).

Metric. Similar to §6.2, we use SQI-SSIM to compute the quality of received volumetric video frames, and 99% frame delivery tail latency to capture the freshness of frames. To calculate the SSIM, we use the mapping developed by ViVo that maps cell PDLs to SSIM for a given distance between the viewer and the display (set to 2.5m in our experiments).

Results. From Figure 11, Octopus achieves 9% percent higher average SQI-SSIM and 83% lower tail latency than ViVo. ViVo has lower frame quality due to its conservative rate adaptation algorithm,

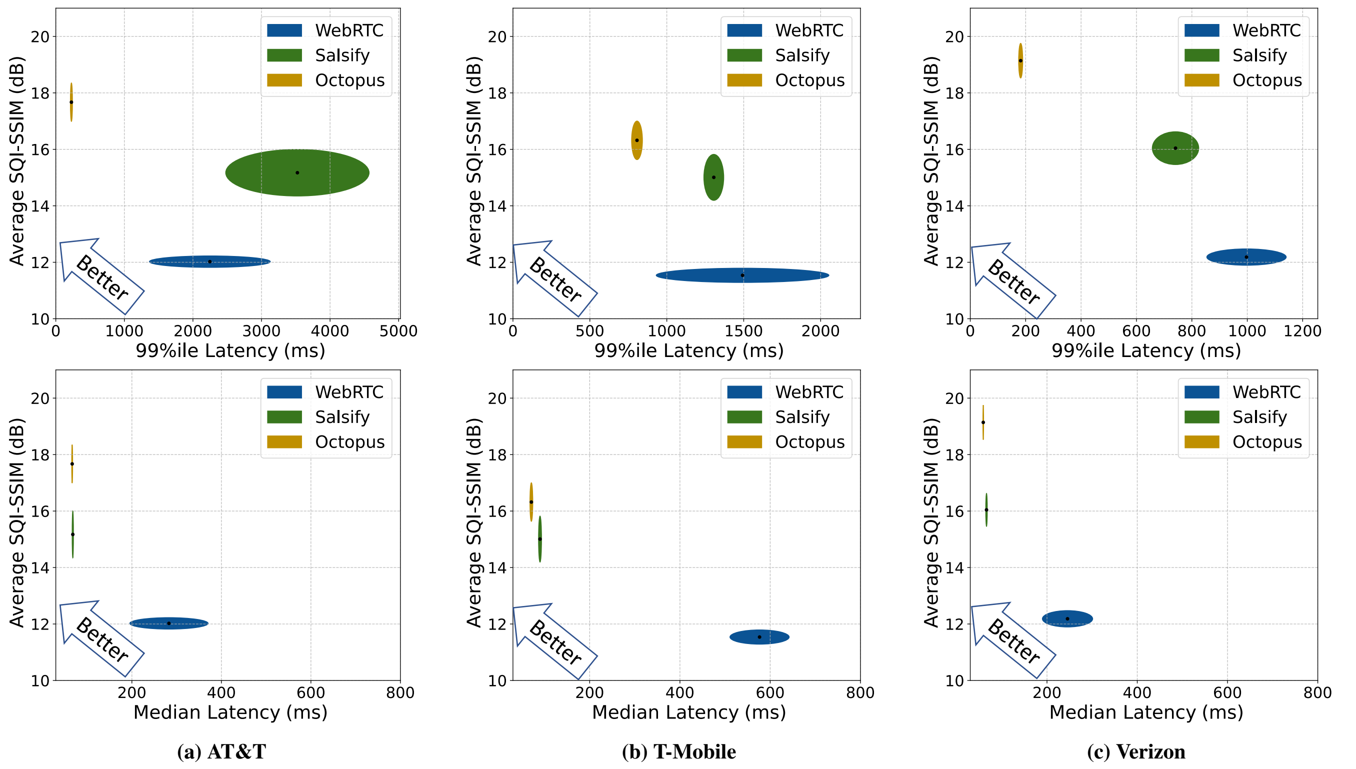


Figure 9: The 99%ile tail and median latency vs. video quality of Octopus and other baselines in different LTE download traces with an RTT of 60 ms. The axes of the ellipse reflect the standard deviation in SQI-SSIM and tail latency.

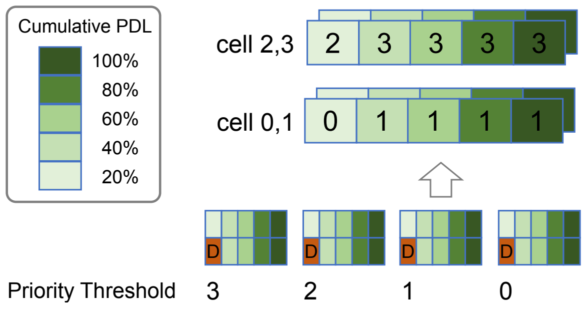


Figure 10: The upper graph shows a volumetric video frame with four cells, where each cell has five layers. The lower graph shows dropper messages and their priority thresholds.

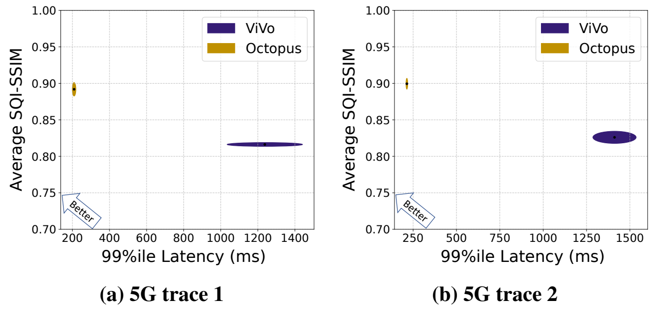


Figure 11: The 99%ile tail latency vs. volumetric video quality of Octopus and ViVo over two 5G traces with a 60 ms RTT.

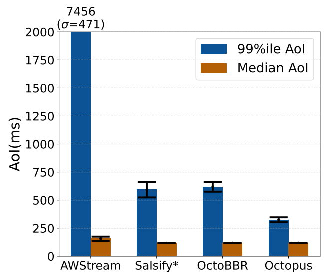
which uses the harmonic mean of the download rate of the previous 20 video frames as the capacity estimate for the next frame. ViVo's tail latency is relatively high because TCP reliably delivers all packets even when the real network bandwidth suddenly drops below the estimate.

7 DETAILED EVALUATION

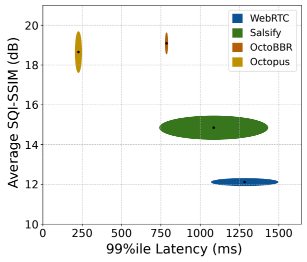
We use our case study in §6.1 and §6.2 for a more in-depth evaluation of Octopus. For brevity, we only present results on the Verizon download trace at 120 ms RTT (we see similar trends with other traces). We use our srsRAN testbed for the results in §7.1, and Mahimahi emulator (that allows for greater configurability in experimental scenarios and baselines) for the remaining experiments.

7.1 Decoupling Impact of Octopus Endpoint Logic

We evaluate the performance impact of the Octopus endpoint's logic in isolation. For this, we use the Octopus transport protocol as it is, but disable the Octopus base-station logic. This effectively models BBR using Octopus logic for transport buffer management and app content adaptation. We refer to this scheme as OctoBBR. Figure 12 compares OctoBBR with Octopus, along with other baselines from §6. There are two key takeaways from these results:

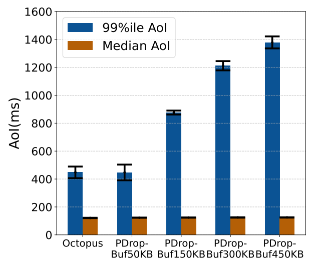


(a) Video w/ frame rate adaptation

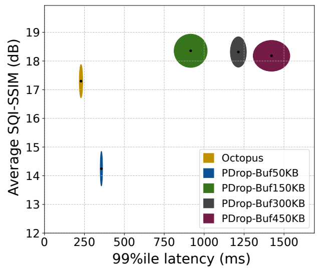


(b) Video w/ quality adaptation

Figure 12: Comparing OctoBBR (Octopus without in-network support) with Octopus and other baselines.



(a) Video w/ frame rate adaptation



(b) Video w/ quality adaptation

Figure 13: Comparing Octopus and priority-based dropping approaches with different buffer sizes.

(i) Octopus performs better than OctoBBR. For our first case study, Octopus has 49% lower tail AoI than OctoBBR (Figure 12(a)). Similarly, for our second case study, Octopus achieves a more desirable trade-off than OctoBBR for the specified policy with 61% lower tail latency and only 4% lower SQI-SSIM (Figure 12(b)). This shows that in-network content adaptation (based on timely and accurate knowledge of link capacity and queue build-up) is useful.

(ii) In general, OctoBBR performs no worse (and often better) than the state-of-the-art endpoint-based solutions (Salsify, WebRTC, and AWStream). This highlights the benefits of direct content adaptation at the endpoint's transport buffer by dropping messages using Octopus' primitives, which quickly reacts to a mismatch between the application sending rate and the pacing rate of the transport protocol.

7.2 Comparison with Priority Dropping

We next compare Octopus' in-network message-dropping logic with a simpler priority-based message-dropping mechanism (inspired by [17, 18]). With this mechanism, when the router buffer is full, and a new message with priority p arrives, instead of dropping the incoming message, the router drops all queued-up messages with lower priority (i.e., having priority value $> p$). We refer to this scheme as PDrop.

Figure 13 compares Octopus with PDrop. We fix the buffer size to 375 KB for Octopus, and use varying buffer sizes for PDrop. For video with frame rate adaptation using temporal layers (Figure 13(a)), we set the message priority for the PDrop to be the same as that for Octopus (described in §6.1). For video with quality adaptation using spatial layers (Figure 13(b)), we set the highest priority 0 for the base (Q_0) layer, followed by priority 1 for Q_1, and then priority 2 for Q_2. We find that PDrop is sensitive to buffer sizes. Larger buffer size results in fewer message drops, and larger latency. A small buffer size, on the other hand, results in significantly lower video quality. Packet delay thresholds or deadlines would be similarly difficult to tune. Octopus primitives are able to react faster and more appropriately by triggering drops directly based on message arrival and link capacity, instead of relying on a buffer threshold. This allows Octopus to achieve a more desirable trade-off.

7.3 Bottleneck at a Legacy Switch

Our experiments so far explored scenarios with a single bottleneck, always at an Octopus-enabled base station. We now evaluate the policy from §6.2 in scenarios with non-Octopus bottlenecks.

Competing backlogged flow at a legacy (non-Octopus) switch. Our first scenario emulates a legacy (non-Octopus) switch with a static link bandwidth of 12 Mbps. In addition to the real-time stream, we generate a competing backlogged TCP (BBR) flow that shares the switch buffer. Figure 14a presents the results of Octopus and comparative baselines in this scenario. With the Octopus base-station logic disabled for the legacy switch, only the Octopus endpoint logic kicks in (which, in line with §7.1 is represented as OctoBBR). Overall, we find that OctoBBR achieves the most desirable trade-off between SQI-SSIM and latency. We find that Salsify (using Sprout as the underlying transport) competes poorly with the backlogged TCP flow, utilizing less than its fair share of the link capacity, which in turn reduces the video quality. OctoBBR competes fairly with the backlogged flow, allowing it to achieve 40% higher SQI-SSIM than Salsify. OctoBBR also achieves higher SQI-SSIM than WebRTC with lower tail latency.

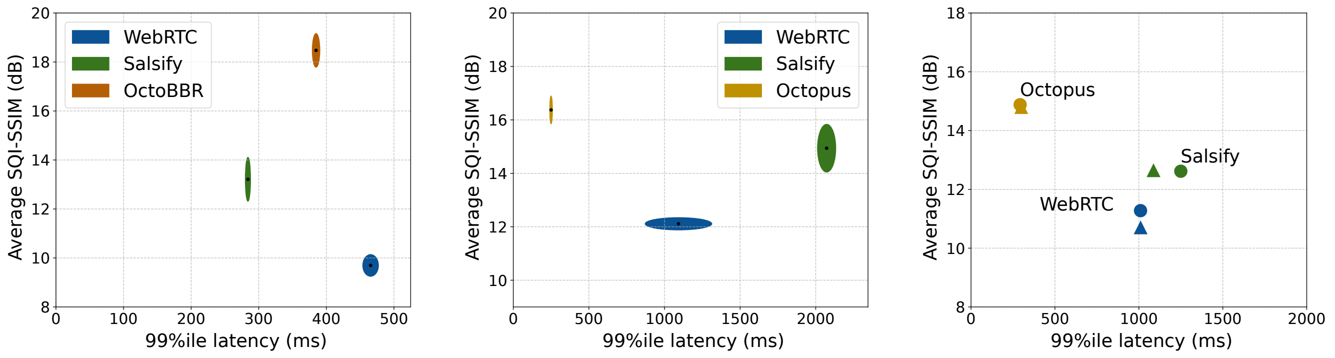
Multiple Bottlenecks. Figure 14b evaluates a scenario with multiple bottlenecks – the first at a legacy (non-Octopus) switch with a 12 Mbps bandwidth, and the second at a cellular (Octopus) link with bandwidth drawn from the Verizon downlink trace. We see similar performance trends as in §6.2.

7.4 Competing Real-time Streams

We next evaluate a scenario with two real-time video streams sharing the same router queue. For the drop-by-bitrate condition, the router computes the max-min fair rate BW_i for each stream i from the observed per-stream enqueuing rate and the overall rate at which the queue is served. We use the policy from §6.2 for both streams. Figure 14c compares Octopus with Salsify and WebRTC (the result for each video stream is denoted using two different markers). We find that Octopus achieves higher SQI-SSIM and lower tail latency for both video streams, as compared to Salsify and WebRTC. This shows that Octopus can appropriately handle multiple real-time streams when the dropping parameters in each stream are configured to minimize the self-inflicted delay.

8 SCOPE AND LIMITATIONS

The goal of our work was to present a new design point for controlling congestion that is based on in-network content adaptation, and show the promise of this approach. We acknowledge that our approach requires changes at both the endpoints and the cellular



(a) When competing with a backlogged flow on a legacy switch.

(b) When there are multiple bottleneck network links.

(c) Two competing video streams (indicated by different markers).

Figure 14: The 99%ile tail latency vs. video quality of Octopus and other baselines for real-time video under different scenarios on the Verizon trace with a 120 ms RTT.

routers, making it difficult to deploy immediately. We list a few other limitations of our work below:

Applicability. Our approach does not entirely replace the need for feedback-based controllers; it is well-suited only for those apps that can support in-network content adaptation. We believe that interactive 5G apps, that most need the fast reaction enabled by Octopus, can indeed be designed to exploit Octopus.

Application development effort. Enabling new apps to exploit Octopus can be non-trivial – specifying dropping parameters requires understanding how the data stream is encoded and the dependency structure between frames, along with semantic insights on the relative importance of different frames. Nonetheless, app-layer frameworks that support mechanisms to encode real-time streams and adapt their content based on estimated bandwidth already exist (e.g., WebRTC for real-time videos [7]). We envision that rather than changing individual apps, one would extend such frameworks to support Octopus, which can enable apps using such frameworks to exploit Octopus without additional efforts.

Encoding overhead. We used an open-sourced single threaded software codec (libvpx[3]) for our experiments, and found the per-frame decoding latency to be small (4-5ms), but the encoding latency to be prohibitively high (90ms for SVC with three quality layers). While we pre-encoded the videos for our experiments, we expect the encoding latency to much lower in practice with more sophisticated multi-threaded or hardware-based encoders. In particular, the broader interest in using SVC for multi-party conferencing (e.g. [6, 71]) has led to the design of new hardware accelerators to support it [6]. We expect such efforts will lower the barrier of using scalable codecs for real-time video streaming. Developing efficient real-time codecs for volumetric videos also remains an open challenge that goes beyond the scope of this work.

On the generality of our primitives. While our primitives can seemingly capture a range of requirements for real-time apps, we are yet to formally analyze their expressiveness.

Coordination via packet headers. Octopus needs 12 bytes of header space to convey the per-message dropping parameters. While our prototype uses application-layer headers, an actual deployment may need to use other alternatives (e.g., IP options/extensions). Diving into the feasibility of these alternatives is beyond the scope of our work.

Competition with buffer-filling cross-traffic. We assume that a user's real-time traffic is isolated from their non-real-time traffic using standard mechanisms [11, 20]. We find that in the absence of such isolation, Octopus' dropping logic competes poorly with flows that aggressively fill up the network buffer. Such a fate is fundamental to any scheme that is designed to react fast to bandwidth variations, and isolation from buffer-filling cross-traffic is therefore a common assumption that is also made by prior works [33, 35, 74].

Hand-overs. Octopus optimizes data transfers while the user device (UE) is connected to a specific base station. Hand-over to another base station would incur high latency due to user state migration. Minimizing the overhead due to that is an orthogonal problem beyond the scope of this work.

9 CONCLUSION

This paper presented Octopus, a system designed to achieve high throughput and low latency for real-time transmissions over cellular networks. Typical real-time apps adapt their content (data quality and frame rate) based on the network bandwidth estimated by the endpoint transport. Octopus allows these apps to send data aggressively and instead specify how content can be adapted by the transport and in-network base stations using per-message parameters. This allows Octopus to react in a timely manner and perform significantly better than the state-of-the-art. Our work leaves several interesting directions open for future research. For example, can we design new techniques for encoding sensor streams to better exploit in-network content adaptation enabled by Octopus? Can Octopus' approach be applied to other contexts beyond cellular networks?

10 ACKNOWLEDGEMENT

We would like to thank our anonymous reviewers and shepherd for their helpful comments. We would also like to thank Brighten Godfrey, Haitham Al-Hassanieh, and Aurojit Panda for their feedback on earlier versions of the paper. This work was supported in parts by Intel, Facebook, AG NIFA grant 2021-67021-34418, NSF grant 2217144, and UIUC's Smart Transport Infrastructure Initiative.

REFERENCES

[1] 2010. 3GPP technical specification for lte. https://www.etsi.org/deliver/etsi_ts/132400_132499/132450/09.01.00_60/ts_132450v090100p.pdf.

[2] 2017. VP9 Video Codec. https://www.webmproject.org/vp9/.

[3] 2020. vp9_spatial_svc_encoder. https://chromium.googlesource.com/webm/libvpx/+/master/examples/vp9_spatial_svc_encoder.c.

[4] 2021. Understanding Packet Loss Priorities. https://www.juniper.net/documentation/en_US/junos/topics/concept/cos-packet-loss-priority-understanding-security.html.

[5] 2022. Scalable Video Coding (SVC) Extension for WebRTC https://www.w3.org/TR/webrtc-svc/. https://www.w3.org/TR/webrtc-svc/

[6] 2022. Support VA-API VP9 K-SVC Encoding on ChromeOS for Intel® Architecture. https://www.intel.com/content/www/us/en/developer/articles/technical/support-va-api-vp9-k-svc-encoding-on-chromeos.html.

[7] 2022. WebRTC: Real-time communication for the web https://webrtc.org/. https://webrtc.org/

[8] 2023. Open RAN explained: Openness, innovation and flexibility. https://www.ericsson.com/en/openness-innovation/open-ran-explained.

[9] 2023. Top Use Cases for 5G Technology. https://www.intel.com/content/www/us/en/wireless-network/5g-use-cases-applications.html.

[10] Admin. 2021. 20 Astonishing Video Conferencing Statistics for 2021. https://digitalintheround.com/video-conferencing-statistics/.

[11] Mehdi Alasti, Behnam Neekzad, Jie Hui, and Rath Vannithamby. 2010. Quality of service in WiMAX and LTE networks [Topics in Wireless Communications]. *IEEE Communications Magazine* 48, 5 (2010), 104–111. https://doi.org/10.1109/MCOM.2010.5458370

[12] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. ACM SIGCOMM*.

[13] The Draco authors. 2023. Draco: 3D Data Compression. https://google.github.io/draco/.

[14] Arjun Balasingam, Manu Bansal, Rakesh Misra, Kanthi Nagaraj, Rahul Tandra, Sachin Katti, and Aaron Schulman. 2019. Detecting If LTE is the Bottleneck with BurstTracker. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. New York, NY, USA. https://doi.org/10.1145/3300061.3300140

[15] James Bankoski, John Koleszar, Lou Quillio, Janne Salonen, Paul Wilkins, and Yaowu Xu. 2011. VP8 Data Format and Decoding Guide. https://www.rfc-editor.org/rfc/rfc6386.

[16] A. M. Bedewy, Y. Sun, and N. B. Shroff. 2019. Minimizing the Age of Information Through Queues. *IEEE Transactions on Information Theory* 65, 8 (2019), 5215–5232. https://doi.org/10.1109/TIT.2019.2912159

[17] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. 1997. An Architecture for Active Networking. In *Proceedings of the IFIP TC6 Seventh International Conference on High Performance Netwoking VII* (White Plains, New York, USA) *(HPN '97)*. Chapman & Hall, Ltd., GBR, 265–279.

[18] Samrat Bhattacharjee, Kenneth L Calvert, and Ellen W Zegura. 1998. *Network support for multicast video distribution*. Technical Report. Georgia Institute of Technology.

[19] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications* (London, United Kingdom). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/190314.190317

[20] F. Capozzi, G. Piro, L.A. Grieco, G. Boggia, and P. Camarda. 2013. Downlink Packet Scheduling in LTE Cellular Networks: Key Design Issues and a Survey. *IEEE Communications Surveys Tutorials* (2013). https://doi.org/10.1109/SURV.2012.060912.00100

[21] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* (2016).

[22] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. 2016. Analysis and Design of the Google Congestion Control for Web Real-Time Communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems* (Klagenfurt, Austria) *(MMSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2910017.2910605

[23] Yongzhou Chen, Ruihao Yao, Haitham Hassanieh, and Radhika Mittal. 2023. Channel-Aware 5G RAN Slicing with Customizable Schedulers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1767–1782.

[24] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. 2022. Swift: Adaptive Video Streaming with Layered Neural Codecs. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 103–118. https://www.usenix.org/conference/nsdi22/presentation/dasari

[25] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review* (1989).

[26] P. Dendorfer, H. Rezatofighi, A. Milan, J. Shi, D. Cremers, I. Reid, S. Roth, K. Schindler, and L. Leal-Taixé. 2020. MOT20: A benchmark for multi object tracking in crowded scenes. *arXiv:2003.09003[cs]* (March 2020). http://arxiv.org/abs/1906.04567 arXiv: 2003.09003.

[27] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 395–408. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/dong

[28] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[29] Zhengfang Duanmu, Zeng Kai, Kede Ma, Abdul Rehman, and Zhou Wang. 2016. A Quality-of-Experience Index for Streaming Video. *IEEE Journal of Selected Topics in Signal Processing* (Feb. 2016).

[30] Eric Dumazet. 2012. TCP small queues. https://lwn.net/Articles/506237/.

[31] Sally Floyd. 1994. TCP and Explicit Congestion Notification. *SIGCOMM Comput. Commun. Rev.* (1994). https://doi.org/10.1145/205511.205512

[32] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.* (1993).

[33] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. 2018. Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 267–282. https://www.usenix.org/conference/nsdi18/presentation/fouladi

[34] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. 2016. SrsLTE: An Open-Source Platform for LTE Evolution and Experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization* (New York City, New York) *(WiNTECH '16)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/2980159.2980163

[35] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. 2020. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 353–372. https://www.usenix.org/conference/nsdi20/presentation/goyal

[36] Yunhong Gu and Robert L. Grossman. 2007. UDT: UDP-Based Data Transfer for High-Speed Wide Area Networks. *Comput. Netw.* (2007).

[37] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review* (2008).

[38] Cathy Hackl. 2021. The Future Of Live Events Begins In The Metaverse. https://www.forbes.com/sites/cathyhackl/2021/07/20/the-future-of-live-events-begins-in-the-metaverse/?sh=57044a2d7ac9.

[39] Bo Han, Yu Liu, and Feng Qian. 2020. ViVo: Visibility-Aware Mobile Volumetric Video Streaming. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) *(MobiCom '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 13 pages. https://doi.org/10.1145/3372224.3380888

[40] Haitham Hassanieh, Omid Abari, Michael Rodriguez, Mohammed Abdelghany, Dina Katabi, and Piotr Indyk. 2018. Fast Millimeter Wave Beam Alignment. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

[41] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. 2012. The NewReno Modification to TCP's Fast Recovery Algorithm. https://datatracker.ietf.org/doc/html/rfc6582.

[42] A. Horé and D. Ziou. 2010. Image Quality Metrics: PSNR vs. SSIM. In *2010 20th International Conference on Pattern Recognition*. 2366–2369. https://doi.org/10.1109/ICPR.2010.579

[43] Tianchi Huang, Rui-Xiao Zhang, Chao Zhou, and Lifeng Sun. 2018. QARC: Video Quality Aware Rate Control for Real-Time Video Streaming Based on Deep Reinforcement Learning. In *Proceedings of the 26th ACM International Conference on Multimedia* (Seoul, Republic of Korea) *(MM '18)*. Association for Computing Machinery, New York, NY, USA, 1208–1216. https://doi.org/10.1145/3240508.3240545

[44] Junchen Jiang, Vyas Sekar, and Hui 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* (Nice, France). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2413176.2413189

[45] Jing Zhu, R. Vannithamby, C. Rödbro, Mingyu Chen, and S. Vang Andersen. 2012. Improving QoE for Skype video call in Mobile Broadband Network. In *2012 IEEE Global Communications Conference (GLOBECOM)*. 1938–1943.

https://doi.org/10.1109/GLOCOM.2012.6503399

[46] Hanbyul Joo, Hao Liu, Lei Tan, Lin Gui, Bart Nabbe, Iain Matthews, Takeo Kanade, Shohei Nobuhara, and Yaser Sheikh. 2015. Panoptic Studio: A Massively Multiview System for Social Motion Capture. In *2015 IEEE International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/ICCV.2015.381

[47] I. Kadota and E. Modiano. 2019. Minimizing the Age of Information in Wireless Networks with Stochastic Arrivals. *IEEE Transactions on Mobile Computing* (2019), 1–1. https://doi.org/10.1109/TMC.2019.2959774

[48] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pittsburgh, Pennsylvania, USA) *(SIGCOMM '02)*. Association for Computing Machinery, New York, NY, USA, 89–102. https://doi.org/10.1145/633025.633035

[49] Sanjit Kaul, Marco Gruteser, Vinuth Rai, and John Kenney. 2011. Minimizing age of information in vehicular networks. In *Proc. IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*.

[50] Eymen Kurdoglu, Yong Liu, Yao Wang, Yongfang Shi, ChenChen Gu, and Jing Lyu. 2016. Real-time bandwidth prediction and rate adaptation for video calls over cellular networks. In *Proceedings of the 7th International Conference on Multimedia Systems*. 1–11.

[51] Kyungjin Lee, Juheon Yi, Youngki Lee, Sunghyun Choi, and Young Min Kim. 2020. *GROOT: A Real-Time Streaming System of High-Fidelity Volumetric Videos*. Association for Computing Machinery, New York, NY, USA.

[52] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. 2022. Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544225

[53] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. 2016. MOT16: A Benchmark for Multi-Object Tracking. *arXiv:1603.00831 [cs]* (March 2016). http://arxiv.org/abs/1603.00831 arXiv: 1603.00831.

[54] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. 2020. A First Look at Commercial 5G Performance on Smartphones. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 894–905. https://doi.org/10.1145/3366423.3380169

[55] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, Feng Qian, and Zhi-Li Zhang. 2021. A Variegated Look at 5G in the Wild: Performance, Power, and QoE Implications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. Association for Computing Machinery, New York, NY, USA.

[56] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proc. USENIX ATC*.

[57] Kathleen Nichols, Van Jacobson, Andrew McGregor, and Janardhan Iyengar. 2011. Controlled Delay Active Queue Management. https://www.rfc-editor.org/rfc/rfc8289.html.

[58] Abhay K. Parekh and Robert G. Gallager. 1993. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-node Case. *IEEE/ACM Trans. Netw.* (1993).

[59] Eric Parsons and Gabriel Foglander. 2023. The four key components of Cloud RAN. https://www.ericsson.com/en/blog/2020/8/the-four-components-of-cloud-ran.

[60] Darijo Raca, Dylan Leahy, Cormac J. Sreenan, and Jason J. Quinlan. 2020. Beyond Throughput, the next Generation: A 5G Dataset with Channel and Context Metrics. In *Proceedings of the 11th ACM Multimedia Systems Conference* (Istanbul, Turkey). Association for Computing Machinery, New York, NY, USA.

[61] Devdeep Ray, Jack Kosaian, K. V. Rashmi, and Srinivasan Seshan. 2019. Vantage: Optimizing Video Upload for Time-Shifted Viewing of Social Live Streams. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. New York, NY, USA.

[62] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. 1998. An Architecture for Differentiated Services. https://tools.ietf.org/html/rfc2475.

[63] T. Schierl, C. Hellge, S. Mirta, K. Gruneberg, and T. Wiegand. 2007. Using H.264/AVC-based Scalable Video Coding (SVC) for Real Time Streaming in Wireless IP Networks. In *2007 IEEE International Symposium on Circuits and Systems*. 3455–3458. https://doi.org/10.1109/ISCAS.2007.378370

[64] H. Schwarz, D. Marpe, and T. Wiegand. 2007. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology* 17, 9 (2007). https://doi.org/10.1109/TCSVT.2007.905532

[65] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. *ACM SIGCOMM Computer Communication Review* (1995).

[66] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*.

[67] G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand. 2012. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology* (2012).

[68] Software Radio Systems(SRS). 2021. srsRAN Your own mobile network. https://www.srslte.com/.

[69] Ammar Tahir and Radhika Mittal. 2023. Enabling Users to Control their Internet. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 555–573. https://www.usenix.org/conference/nsdi23/presentation/tahir

[70] Chia-Hui Tai, Jiang Zhu, and Nandita Dukkipati. 2008. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM 2008. 27th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 13-18 April 2008, Phoenix, AZ, USA*.

[71] Vikram Sachdeva. 2020. Zoom - Video conf app at scale. https://medium.com/@vsachdeva/zoom-video-conf-tool-at-scale-e86289c290b8.

[72] M. Wien, R. Cazoulat, A. Graffunder, A. Hutter, and P. Amon. 2007. Real-Time System for Adaptive Video Streaming Based on SVC. *IEEE Transactions on Circuits and Systems for Video Technology* 17, 9 (2007), 1227–1237. https://doi.org/10.1109/TCSVT.2007.905519

[73] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2018436.2018443

[74] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proc. USENIX NSDI*.

[75] Yaxiong Xie, Fan Yi, and Kyle Jamieson. 2020. PBE-CC: Congestion Control via Endpoint-Centric, Physical-Layer Bandwidth Measurements. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. https://doi.org/10.1145/3387514.3405880

[76] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. 2020. Understanding Operational 5G: A First Measurement Study on Its Coverage, Performance and Energy Consumption. 479–494. https://doi.org/10.1145/3387514.3405882

[77] Mao Yang, Yong Li, Depeng Jin, Li Su, Shaowu Ma, and Lieguang Zeng. 2013. OpenRAN: A Software-Defined Ran Architecture via Virtualization. In *Proceedings of the ACM SIGCOMM 2013 Computer Communication Review* (Hong Kong, China). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2486001.2491732

[78] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. 2015. Adaptive Congestion Control for Unpredictable Cellular Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. https://doi.org/10.1145/2785956.2787498

[79] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive Wide-area Streaming Analytics. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

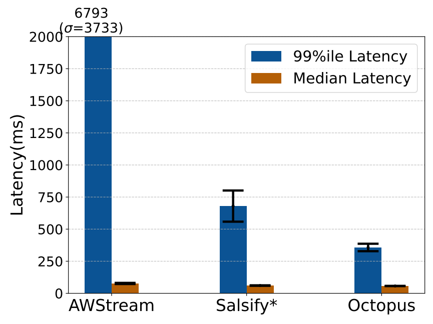
[80] Huanhuan Zhang, Anfu Zhou, Yuhan Hu, Chaoyue Li, Guangping Wang, Xinyu Zhang, Huadong Ma, Leilei Wu, Aiyun Chen, and Changhui Wu. 2021. Loki: Improving Long Tail Performance of Learning-Based Real-Time Video Adaptation by Fusing Rule-Based Models. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (New Orleans, Louisiana) *(MobiCom '21)*. New York, NY, USA, 775–788. https://doi.org/10.1145/3447993.3483259

[81] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. 2020. OnRL: Improving Mobile Video Telephony via Online Reinforcement Learning. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) *(MobiCom '20)*. New York, NY, USA, 14 pages. https://doi.org/10.1145/3372224.3419186

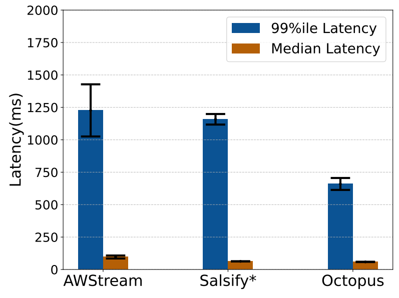
[82] Yunfei Zhang, Gang Li, Chunshan Xiong, Yixue Lei, Wei Huang, Yunbo Han, Anwar Walid, Y. Richard Yang, and Zhi-Li Zhang. 2020. MoWIE: Toward Systematic, Adaptive Network Information Exposure as an Enabling Technique for Cloud-Based Applications over 5G and Beyond (Invited Paper) *(NAI '20)*. New York, NY, USA. https://doi.org/10.1145/3405672.3409489

[83] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. 2019. Learning to Coordinate Video Codec with Transport Protocol for Mobile Video Telephony. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3345430

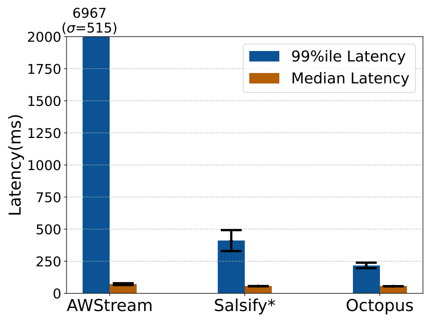
[84] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. https://doi.org/10.1109/TIP.2003.819861



(a) AT&T



(b) T-Mobile



(c) Verizon

Figure 15: Median and 99%ile latency in different LTE download network traces with an RTT of 60 ms (averaged over 5 videos, with error bars showing the standard deviation).

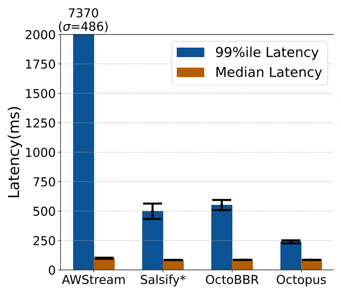
A SUPPLEMENTARY CASE STUDY RESULTS

We provide several additional case study results here.

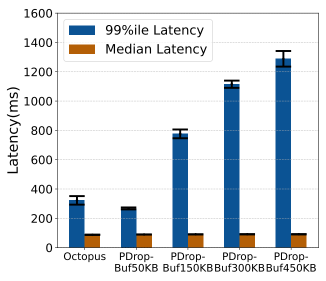
Figure15 shows the median and tail latency of frames transmission in the first case study (6.1). We see similar trends with tail latency as with the AoI metric – Octopus has 1.8–28× lower tail latency than the baselines.

Figure 16(a) presents the frame transmission latency of OctoBBR, when compared with Octopus and the other baselines for the first case-study. Consistent with the AoI trends reported in 7.1, OctoBBR exhibits a slightly higher 99%ile latency than Salsify, while Octopus achieves at least 2× lower tail latency than all other schemes. The performance improvement of Octopus compared with OctoBBR highlights the benefit of in-network adaptation policies.

Figure 16(b) compares the frame latency of Octopus with PDrop (priority-based dropping) using different buffer sizes. Again, consistent with the AoI trends in 7.2, we find that PDrop only outperforms Octopus when the buffer is small(50KB), and suffers from long tail latency when the buffer size goes large. As we showed in 7.2, PDrop with a small buffer size of 50KB results in degraded video quality for our second case-study, when compared to Octopus. This shows that the performance of PDrop highly depend on the buffer threshold, which is non-trivial for the network operator to tune.



(a) OctoBBR(Octopus without in-network support)



(b) PDrop with different buffer sizes

Figure 16: Median and 99%ile latency of OctoBBR and the priority-based dropping approach.