Computer Networks 134 (2018) 152–166

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Responsive high throughput congestion control for interactive applications over SDN-enabled networks

Aous Thabit Naman*, Yu Wang, Hassan Habibi Gharakheili, Vijay Sivaraman, David Taubman

School of Electrical Engineering and Telecommunications, UNSW, Sydney, Australia

A R T I C L E   I N F O

Article history:
Received 23 August 2017
Revised 17 January 2018
Accepted 22 January 2018

Keywords:
Software defined network
OpenFlow
Interactive video
JPEG2000
JPIP
Congestion control

A B S T R A C T

New interactive video applications are increasingly emerging over the Internet; these interactive applications are characterized by high bandwidth requirements that fluctuates depending on end-user actions (e.g. less bandwidth is usually needed for stationary scenes). More importantly, this interactive class of services also involves a requirement for high responsiveness (i.e. low latency) from the network, in order to respond in real-time to end-user actions. One emerging service of this nature is 360° video streaming; another example is cloud-based gaming services. In this paper, we focus specifically on JPIP (JPEG 2000 Interactive Protocol) applications that support remote interactive video browsing with dynamic pan and zoom capabilities, as a highly representative example of the interactive service class. Existing network communication services are mostly agnostic to latency implications, and hence are not well adapted to such interactive applications. Meanwhile, explicit resource reservation protocols have not been widely deployed, and do not consider the time-varying dependencies that naturally arise in interactive applications. In this work, we leverage software defined networking (SDN) principles to support a proposed “interactive service” class. The main contributions of this work are a network-exposed application programming interface (API) that provides visibility into the state of the network, an SDN-assisted congestion control algorithm that utilizes network state information to achieve the desired low latency and high bandwidth utilization requirements, and a fair resource assignment algorithm that shares available bandwidth among interactive and non-interactive traffic dynamically – all without a reservation protocol.

1. Introduction

We are witnessing an emergence of on-line interactive video applications. The requirements of interactive video differ from conventional video streaming; while both are amenable to optimization of the received data, they have different requirements on pre-fetching, round trip time, and retransmissions. For cloud-based gaming services, such as PlayStation Now, round trip time should be kept to a minimum, such that the player feels that the system is responding to his/her actions, and therefore pre-fetching is not desirable, since it increases the round trip time. Similar considerations apply to interactive video streaming based on the JPEG 2000 Interactive Protocol (JPIP) [1], where a client interactively browses and retrieves remotely stored video, based on one or more interactively controlled windows of interest (WOI), governing both scale/zoom and spatial support. For viewport adaptive streaming of 360° immersive video for head-mounted displays, keeping round trip time to a minimum is of great importance, because this keeps the end-user feeling immersed in the video when he/she moves his/her head. To address this problem in part, Gudumasu et al. [2] explore a variety of tiling strategies to hide network latency, at the expense of wasting bandwidth in sending content that the end user may not view. Augmented reality applications also share many of the same properties.

All of these applications belong to an interactive class of services that is addressed by the methods proposed in this paper. The bandwidth required for an interactive stream is time varying, and depends on end-user actions as well as the contents being delivered. Less bandwidth is usually needed in stationary portions of a scene, but also when the end-user pauses, reduces the playback rate or navigates over previously visited content that may be partially held in a local cache. Additionally, retransmission of lost packets is only worthwhile if these packets are delivered in time for them to be used in the reconstruction of yet to be dis-

* Corresponding author.
E-mail addresses: aous@unsw.edu.au (A.T. Naman), yu.wang1@unsw.edu.au (Y. Wang), h.habibi@unsw.edu.au (H.H. Gharakheili), vijay@unsw.edu.au (V. Sivaraman), d.taubman@unsw.edu.au (D. Taubman).

https://doi.org/10.1016/j.comnet.2018.01.043
1389-1286/© 2018 Elsevier B.V. All rights reserved.

played frames, and then only if the packets are relevant to a possibly modified window of interest or viewpoint.

In summary, these interactive video applications demand a considerable amount of network bandwidth, just like other video streaming applications, but more importantly they also need low latency to maintain real-time responsiveness. The low latency requirement of interactive video flows cannot be guaranteed when these flows are intermixed with other flows, since other flows (such as TCP) can queue up data over the bottleneck link, causing high latency. Therefore, in this work, we partition traffic originating from interactive and non-interactive services into two different queues at each switch. Such an approach has been successfully employed by Podlesny and Williamson [3], where they utilize one queue for TCP NewReno (loss-based latency-agnostic) flows and another queue for TCP Vegas (delay-based latency-sensitive) flows to prevent TCP NewReno from dominating the available bandwidth and causing high latency. It is also the approach adopted by the newly proposed L4S [4] protocol for low latency traffic.

Our goal is to leverage SDN to provide a highly responsive and bandwidth-efficient mechanism for serving interactive traffic, while fairly allocating bandwidth for other traffic, without the need for reservation or static allocation of resources. The contributions of this work are as follows: We develop an SDN-based system architecture with a RESTful API that exposes network state information to interactive applications on short time scales (tens to hundreds of milliseconds). We also develop a low-delay high-throughput congestion control algorithm that utilizes information provided by the SDN controller for responsive and bandwidth-efficient delivery of interactive traffic. Additionally, we develop a dynamic service policy scheme for interactive applications that adjusts the minimum bandwidth available to interactive flows based on network conditions and the state of non-interactive flows. Lastly, we evaluate the benefits of our proposed scheme via experiments in the *mininet* environment with real JPIP endpoints; JPIP is chosen because it exemplifies interactive video applications and is already well-defined and in use. We argue that the proposed scheme offers a viable solution for a richer interactive video experience.

A good approach for congestion control is to employ a congestion window mechanism. Most if not all of TCP variants employ such a window, relying on end-to-end probing to adjust the size of this window. A commonly accepted size for this window, which meets the low latency requirement, is the product of available bandwidth and the minimal round trip delay; a congestion window larger than the bandwidth-delay product buffers data in the network (imposing delay), while a smaller window does not utilize enough of the available bandwidth. For this approach to work efficiently, reasonable estimates of bandwidth and round trip time are important. While it is possible to estimate available bandwidth using packet pair probing [5–7], it is not clear, when video content is being served, how much of an observed round trip delay results from queuing delay; in order to get an accurate measure of the delay, conventional techniques need to let the link between the interactive endpoints go idle at times. This reduces overall utilization of the available capacity. In this work, we employ software-defined networking (SDN) techniques to expose network state information via a RESTful API; interactive applications can utilize this state information to obtain estimates of available bandwidth and latency that are current and more accurate than what can be obtained by probing, since they are measured in real-time by the SDN controller.

The proposed RESTful API uses SDN only as a passive monitoring tool. In this work, we also employ SDN to dynamically adapt the service policy for the two service classes under consideration: interactive and non-interactive. In this case, SDN is used to actively adjust queue service rates at the individual switches based on the number of flows and bandwidth utilization of each queue. Both the passive and active uses of SDN are important to the delivery of a fair, efficient and appropriate communication infrastructure for interactive and non-interactive flows, as envisaged here.

Other researchers find it useful to expose a network API to video applications [8–10], but their main focus is on request/control of bandwidth reservation per video stream over a short period of time. In our proposed approach, neither clients nor servers make any explicit request for resources. They are assigned to the interactive or non-interactive class by the SDN controller. While a differential charging model might be imposed by network providers, there is no fundamental reason why this should be required, since the benefits of being assigned to the interactive service class apply only to actual interactive streaming applications.

The rest of the work is organized as follows: Section 2 summarizes relevant prior work. The proposed system architecture and API are explained in Section 3. We present the SDN-assisted congestion control mechanism and related experimental results in Section 4 for a single provider network. Section 5 explains our dynamic service policy algorithm with experimental results. Section 6 extends the proposed approach to federated networks, and Section 7 concludes the paper and gives directions for future work.

2. Related work

In this section, we cover three areas of related research; namely, quality of service, quality control, and congestion control.

2.1. Quality of service

Several techniques have been proposed to maintain quality of service for on-line video delivery. Video content providers employ techniques such as TCP instrumentation [11] and video rate adaptation (using real-time probing of available bandwidth). These methods are not so helpful for interactive videos, especially when the network is congested, since they primarily aim at bandwidth provisioning with latency being out of their concern. We note that the networking research community has also developed a variety of service quality control solutions ranging from ATM switched virtual circuits (SVC) to RSVP and IntServ/DiffServ with limited success. These techniques mainly require fairly static configurations, offer “soft” assurance of quality, and do not expose control to applications in the process of fair network resource provisioning.

2.2. Quality control

There is a large body of research that advocates an application-aware networking paradigm for improving application performance and network utilization. The work in [12] explores utilizing an integrated network control and configuration for big data applications, performing bulk transfers. Our focus in this paper is on delay-sensitive interactive video that demands low RTT and high throughput. Many SDN-based frameworks have been proposed to explicitly control network services. PANE [8] proposes a set of network programmable interfaces that enable applications to query the network state and request a specialized treatment (e.g., bandwidth reservation). Similarly, the system in [10] allows users and content providers to request provisioning special lanes for certain flows (e.g., application-specific or device-specific) over the bottleneck access link from the broadband network operator. By contrast, in the approach proposed in this paper, the video application does not specify or exercise network control; instead, an SDN controller exposes network state information to application endpoints, and dynamically adapts provisioning in the network.

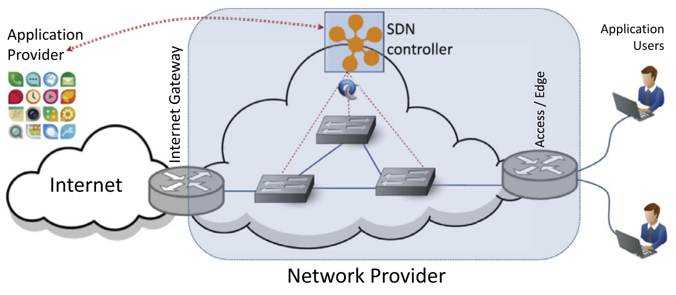


Fig. 1. System architecture.

2.3. Congestion control

Numerous congestion control mechanisms have been developed in the past; examples include [13–15]. OpenTCP [13] is the first attempt to employ SDN to dynamically adapt TCP, improving its performance. TIMELY [14] and DX [15] use delay measurements to detect congestion in datacenter networks. In L4S [4] and DCTCP [16], network devices use packet marking as a soft indicator for congestion; end-points notice these markings and reduce their transmission rate, easing congestion, while maintaining low delay high throughput transmission. In this work, we leverage the visibility provided by an SDN controller, which sits at a central vantage point, to obtain quicker and more accurate estimate of the network state. Section 3.2.2 gives a detailed description of the exposed network state.

3. System architecture design

In this section, we describe our solution for enhancing the delivery of interactive flows (e.g. JPIP video) in an SDN-enabled network. We begin by outlining our major architectural decisions in Section 3.1. This is followed by the proposed network API in Section 3.2.1 and the proposed network quality control in Section 3.2.3.

3.1. Architectural decisions

Fig. 1 shows the proposed system architecture; the aim is to enhance the performance of interactive video applications and to efficiently utilize network resources. We rely on the SDN to provide network visibility to interactive video applications and to dynamically control bandwidth provisioning for them. Since delay-sensitive applications tend to compete poorly with aggressive TCP traffic [14], we propose to isolate interactive flows from the rest of the traffic by using an application-specific queue, which we identify as the interactive queue, at each hop, where a guaranteed fraction (slice) of a hop's bandwidth is provisioned for that queue. This slice of bandwidth is adjusted dynamically, as explained in Section 5. The SDN controller detects interactive flows and assigns these flows to the interactive queues, without the need for any reservation of resources. A registration API call is employed to assist in the identification of interactive flows, as explained in Section 3.2.1. In Sections 3–5 of this work, we focus on a single domain network, extending the proposed approach to federated networks in Section 6.

The SDN controller in this work has good knowledge of the network topology, including link capacities and physical delays. It also knows the network path between any two endpoints (video server and client in our case), since it manages data flows in the network. Our controller runs an application that is capable of probing the state of the data-plane elements (network switches and routers) several times a second (e.g. every 50 ms) to collect network information such as queue lengths and cumulative bytes transferred. This information is stored by the SDN application as entries in a data table, which we identify as the network state table (NST); each reading in the NST is associated with an index. It is sufficient for this table to store a few seconds' worth of data (e.g. in a circular buffer), since older data become of little use for the estimation of the current network state needed by the interactive video endpoints, which are the JPIP servers and clients in the context of this work.

The data thus collected by the SDN application from network elements is exposed via a RESTful (REpresentational State Transfer) API; this way interactive endpoints can poll the SDN application for network state using standard HTTP commands over TCP. The use of a binary format for state information, which is more efficient but less flexible, might be explored in a future work.

3.2. Network exposure and control

3.2.1. Network visibility

The SDN controller in the proposed architecture provides a RESTful API that accepts two types of calls:

- **Registration**: This call aims at providing the SDN controller with attributes of the interactive video application. In registration, an application specifies (a) the identity of the caller application requesting the special treatment; and (b) the 5-tuple flow information; a flow's 5-tuple information is the source IP address and port number; the destination IP address and port number; and the protocol in use. This API is called by the application at the start of the video session and prior to querying the network.
- **Network Query**: This call provides the caller application with network state information. The reply includes (a) the number of hops (and their physical delay) between application endpoints; (b) the number of waiting (or queued) bytes in the interactive queue at each hop; (c) the minimum bandwidth of the interactive queue at each hop; and (d) the throughput of the interactive queue at each hop. In practice all these parameters are estimates, since exact information may not be discoverable and cannot generally be current.
  This query API is employed by interactive video server(s) and clients. Network state information is then utilized by the congestion control algorithm of the interactive application; the objective of the proposed congestion control algorithm is for each flow to approximately maintain a target number of bytes on the queue of its most congested hop (i.e. bottleneck link), minimizing the overall RTT. A detailed description of this call is given in Section 3.2.2.

Note that the above API calls need to authenticate the caller application, but in this work we mainly focus on our proposed congestion control algorithm and dynamic bandwidth provisioning.

3.2.2. Detailed description of the network query call

To request data from the NST, the interactive client or server sends a GET request of the form:

`GET /stats/<MyIP>/<PeerIP>/<LastIdx>/<MaxEntr>/` specifying its IP address (`MyIP`), its peer IP address (`PeerIP`), the last NST entry index it has received (`LastIdx`), and the maximum number of entries that it is willing to receive (`MaxEntr`). The first two values, myIP and PeerIP, help the SDN controller identify the two network endpoints and a direction along the path between these two endpoints. The value of LastIdx prevents the SDN controller from sending entries that endpoints already know about. The MaxEntr value serves to limit the amount of response data in cases when there is no previous request (in this case LastIdx is 0) or there is no request for a long time.

The SDN controller replies with a concatenation of network state entries of the form:

$[ns\_entry,\ ns\_entry, \ldots]$

where each entry is of the form:

$[i,\ L,\ link\_entry^i_1,\ link\_entry^i_2,\ \ldots,\ link\_entry^i_L]$

Here, i is the entry index, and L is the number of links on the path between a client and its server; these links can be within one or multiple service providers. A link entry for a link l, where $1 \le l \le L$, has the form:

$[\Delta^i_l,\ b^i_l,\ q^i_l,\ R^i_l,\ d^i_l]$

where Δ^i_l is the acquisition interval (seconds) between entry i and entry $i-1$ for link l., b^i_l is the number of bytes transmitted through link l in that interval, q^i_l is the number of bytes that are queued at the buffer before link l when this entry is created, R^i_l is the minimum data rate allocated to the interactive queue at link l, and d^i_l is the physical delay on link l. We choose to communicate the link rate and physical delay with each entry, since it is possible that the SDN controller changes the path[1] between the endpoints. This is also useful when the SDN controller changes bandwidth provisioning for interactive streams, as explained in Section 5. Communicating a time duration Δ^i_l for each link l, instead of one time duration for each index i, is useful when multiple SDN controllers are employed along a path, within one network provider or multiple providers; in this case, each controller can have its own set of state acquisition intervals. We explore this scenario further in Section 6.

The API above is designed to provide minimal yet sufficient information (bytes transmitted and queued on each link) to interactive endpoints, without revealing per-flow information. The latter can impose significant burden on the network and the controller in terms of data collection and communication. Our implementation (in Sections 4 and 6) demonstrates feasibility, though we defer larger-scale deployment to future work. In order to use the API effectively, interactive endpoints need to supplement the aggregate information that they receive with properties of their own flow, estimated separately, as described below.

In the current implementation, each request receives a reply that describes the path from MyIP to PeerIP. If the information for the reverse path is needed, a separate request must be issued. In this work, we always pipeline the request for MyIP to PeerIP with the request for PeerIP to myIP, and we refer to them as a forward-backward request.

Interactive endpoints post requests to the SDN controller to obtain the network state that is of importance to them. The arrangement used for experimental results is for the client to send a forward-backward request, and wait for the reply, before sending a new forward-backward request. The server posts forward-backward requests for all the clients of interest, and waits for all the replies, before issuing new requests.

3.2.3. Network quality control

The objective is to maintain the performance of interactive videos, while being fair to other flows in the network. The SDN controller achieves this “fairness” by periodically running a dynamic service policy that sets the minimum rate for the interactive queue to a fraction of the link capacity that is equal to the ratio of the number of interactive video flows to the total number of flows. Section 5 details this service policy.

[1] The proposed approach requires some path stability; e.g., it is not clear if it can work with link load balancing.

4. SDN-assisted congestion control

Both interactive endpoints need to estimate network bandwidth and delay attributes while serving or receiving video content. An interactive client uses delay information to estimate when to request subsequent frames' data, such that this data can be delivered in time for rendering at the client. The client also needs bandwidth information to choose an ideal number of bytes to request for each frame; requesting too little data does not utilize enough of the available bandwidth, while requesting too much data increases delay, since some of this data needs to be buffered in the network. An interactive server uses available bandwidth to decide how many bytes to send in given period of time (for throttling). Further exploration of the operation of JPIP is beyond the scope of this work. In this work, the server and client use SDN-supplied information to obtain accurate and timely estimates of available bandwidth and delay.

4.1. Using network state at the interactive server

Using the state information received from the SDN controller, the interactive server estimates the average stream bandwidth going through each link along the communication path, using an exponential smoothing strategy, which is also known as Exponential Weighted Moving Average (EWMA). For the purpose of this section, we consider that the bandwidth allocated to the servicing of the interactive queues is fixed, as this allows for better investigation of the proposed approach; a more realistic policy is explored in Section 5, where the bandwidth allocated to the interactive queues is dynamically adjusted. We write $\Lambda_{S\to C}$ for the set of all links in the path from the server to the client; we similarly write $\Lambda_{C\to S}$ for the set of links in the path from the client to the server. This way, the average bandwidth $\bar{\lambda}^i_l$ going through link l at time index i is given by:

$$\bar{\lambda}^i_l = \bar{\lambda}^{i-1}_l + \alpha \cdot \left(\frac{b^i_l}{\Delta^i_l} - \bar{\lambda}^{i-1}_l \right), \quad l \in \Lambda_{C\to S}, \Lambda_{S\to C} \tag{1}$$

where α controls how fast bandwidth estimates respond to changes in the observed bandwidth b^i_l/Δ^i_l. We set the smoothing time constant Δ^i_l/α to 1s in this work. However, for exponential smoothing, $0<\alpha<1$; therefore, we use $\alpha = \min(1, \Delta^i_l)$. The average number of queued bytes, $\bar{q}^i_l$, at the buffer of link l is similarly estimated using :

$$\bar{q}^i_l = \bar{q}^{i-1}_l + \alpha \cdot (q^i_l - \bar{q}^{i-1}_l), \quad l \in \Lambda_{C\to S}, \Lambda_{S\to C} \tag{2}$$

This average is for queued bytes at link l from all flows; next, we explore how an interactive endpoint can estimate the average number of queued bytes associated with its own flow.

In the following discussion, we assume that the path from the interactive server to the client carries more data than the client to server path, which is the case in all video streaming applications. Both endpoints estimate this server-to-client bandwidth $\hat{\lambda}^i_f$; the server monitors acknowledgments from the client and counts how many bytes are associated with these acknowledgments, while the interactive client keeps track of the number of bytes received from the server in a given period of time. Both use a moving average approach with a window size of 0.5 seconds.

Naturally, the bandwidth estimate of interactive flow $\hat{\lambda}^i_f$ should be no larger than the bandwidth $\bar{\lambda}^i_l$ experienced at each link $l \in \Lambda_{S\to C}$. It is also fair to assume that the number of bytes $\hat{q}^i_{f,l}$ queued at any link $l \in \Lambda_{S\to C}$ and associated with server-to-client flow f is no larger than the estimated total number of bytes $\bar{q}^i_l$ queued at the same link. With that in mind, we estimate the number of queued bytes associated with flow f at the buffer of link l

using:

$$\hat{q}^i_{f,l} = \begin{cases} \frac{\hat{\lambda}^i_f}{\bar{\lambda}^i_l} \cdot \bar{q}^i_l, & \bar{\lambda}^i_l \geq \hat{\lambda}^i_f \text{ and } \bar{\lambda}^i_l > 0 \\ \bar{q}^i_l, & \text{otherwise} \end{cases}, \quad l \in \Lambda_{S\to C} \tag{3}$$

The congestion control policy in this work is based on the interactive server attempting to maintain a certain number of bytes S from its flow f at the most congested (bottleneck) link for that flow, so long as it has something to send; if it has nothing to send to a client, then it can let the link go idle for that flow f. This is similar to delay-based TCP Vegas [17] and FAST TCP [18] congestion control algorithms in that the transmitter attempts to maintain a small number of packets queued in the network by allowing a correspondingly small increase in observed delay over the idle case (or BaseRTT); in these algorithms, the exact number of queued packets cannot be controlled as accurately as we can in this work. We assume that all *interactive* flows are characterized by the same value for S. The server estimates the rate $\lambda^i_{f,l}$ that achieves S queued bytes at the buffer of link l using:

$$\lambda^i_{f,l} = \frac{S}{S + (\bar{q}^i_l - \hat{q}^i_{f,l})} \cdot R^i_l, \quad l \in \Lambda_{S\to C} \tag{4}$$

That is, at the steady state, we should have S bytes from flow f and $\bar{q}^i_{f,l} - \hat{q}^i_l$ from other flows. The corresponding data rate is decided by the most congested link l, as:

$$\lambda^i_f = \min_l \left\{\lambda^i_{f,l}\right\}, \quad l \in \Lambda_{S\to C} \tag{5}$$

To find the congestion window, we need to estimate the round trip time, which is the sum of the network delay δ^i_f experienced by server-to-client flow f and the network delay $\delta^i_{\bar{f}}$ experienced by its associated client-to-server flow $\bar{f}$. We estimate the server-to-client delay δ^i_f using:

$$\delta^i_f = \sum_l \left(\frac{S + \bar{q}^i_l - \hat{q}^i_{f,l}}{R^i_l} + d^i_l \right), \quad l \in \Lambda_{S\to C} \tag{6}$$

That is, a packet from flow f experiences an expected delay due to buffering of $\bar{q}^i_{f,l} - \hat{q}^i_l$ bytes from other flows and to storing and forwarding its S bytes.[2] We find the client-to-server delay $\delta^i_{\bar{f}}$ using:

$$\delta^i_{\bar{f}} = \sum_l \left(\frac{\bar{q}^i_l}{R^i_l} + d^i_l \right), \quad l \in \Lambda_{C\to S} \tag{7}$$

The congestion window is then set, following Little's Law [19], to

$$W^i_f = \lambda^i_f \cdot (\delta^i_f + \delta^i_{\bar{f}}) \tag{8}$$

The main regulator of traffic flow in our proposed model is the server's congestion window W^i_f, as calculated above, representing the total number of unacknowledged bytes that the server is prepared to have outstanding within the network. Additionally, however, the server shapes its outgoing traffic to a maximum data rate of $1.25 \cdot \lambda^i_f$, so as to minimize the impact of sudden changes in W^i_f on the intermediate link buffers.

4.2. Using network state at the interactive client

The interactive client maintains a request window, constraining the number of bytes that might still be received in response to outstanding requests, based on these requests' byte limits.[3] To estimate this request window, the client could use its estimate of the server-to-client bandwidth $\hat{\lambda}^i_f$, together with an estimate of the minimum round trip delay $\delta_{\min}$ derived directly from the proposed SDN API, setting the request window to $1.5 \cdot \hat{\lambda}^i_f \cdot \delta_{\min}$. However, for more accurate estimates, the client employed in this work uses the exact same procedure as that used by the server; in particular, the client's request window is set to $1.5 \cdot W^i_f$, obtained via (8).

4.3. Performance evaluation of SDN-assisted congestion control

We now evaluate the performance of the interactive application in an SDN-enabled network using our JPIP video streams and endpoints.

4.3.1. Test setup

The various results of this section are produced using the test setup shown in Fig. 2. The network is emulated inside the Mininet tool[4] [20], which is run on an Ubuntu Linux platform[5] Inside mininet, we have two switches,[6] one server with an IP address of 10.0.0.2, and one SDN Controller with an IP address of 10.0.0.254, which works as a DHCP server as well. A separate machine runs the JPIP server. To obtain accurate results, all servers inside Mininet are configured to disable using "jumbo" frames. The Linux machine is physically connected to an OSX machine[7] with a DHCP-assigned IP address; the OSX machine runs JPIP clients. There are also two hosts with IP addresses 10.0.0.3 and 10.0.0.4 that can be used to simulate non-interactive traffic; these host are useful for the experiments in Section 5.

The traffic control command, `tc`, is run every 50 ms on the Linux machine to collect the state of the different switches and network interfaces inside Mininet. This data is stored in the NST table, and made available for serving by an HTTP server. We use a Python-based HTTP server running on 10.0.0.254.

We set the number of queued bytes S in (4) to 1500. The experiment is not an exact emulation of reality; for example, the 4 octet FCS field of each Ethernet frame is not included in the reported rates. Also, the experiment ignores the 7 octet preamble, the 1 octet at the start of each delimiter, and the 12 octet gap that exists between each pair of Ethernet frames. For convenience, we also ignore these fields in calculating byte counts and estimated rates.

For testing, we employ a very high quality video with a highly scalable representation. In particular, we use the "Sintel" sequence [21] with a resolution of 4096×1744 and a frame rate of 24 frames per second. The first 3000 frames of the "Sintel" sequence are compressed using JPEG2000 Part 2 with 20 quality layers, 7 levels of DWT decomposition, 32×32 coefficient codeblocks, 4×4 codeblock precincts, and 8 bits per color component. The 9/7 CDF wavelet is used in the irreversible path of JPEG2000.

This highly scalable representation enables the JPIP server to select and serve the best available quality for the client's view-

[2] In experimental results, we ignore this S bytes store and forward delay, using $\delta^i_f = \sum_l ((\bar{q}^i_l - \hat{q}^i_{f,l})/R^i_l + d^i_l)$ instead of (6), as the Mininet tool appears to ignore the store-forward delay for packets. This might be legitimate for switches that do not require a packet to be fully buffered before transmission can commence.

[3] The client may receive fewer bytes in response to any request than its request limit would suggest, since the server models the client's cache and only sends information that the client does not already have.

[4] Mininet version 2.1, http://mininet.org.

[5] An Intel Xeon W3520, 4 cores/8 threads, with 12GB RAM.

[6] Two additional switches are inserted on the path between the two switches of Fig. 2; these inner switches are used to emulate the 50ms propagation delay, while the outer switches are used to emulate the change in data rate between 5 Mbps and 10 Mbps. We do so to overcome Mininet's behavior of using egress buffers of switches to emulate delay, which interferes with the number of queued bytes used in this work; we are interested here in data bytes that are buffered in egress buffers due to change in data rate between ingress and egress links, and not in those that are used to emulate delay.

[7] An Intel i7-3667U machine, 2 cores/4 threads, with 8GB RAM.

port, given network conditions and the client's cache state, without playback stalls, and without re-encoding the already compressed codestream [22]. The playback system used for experimental results adopts a viewport size of 1024×436, which is the extent of the window of interest requests forwarded to the JPIP client; this viewport is a portal into the full content that is being zoomed and panned.

4.3.2. *Performance improvement on SDN*

This section explores the performance improvement of our proposed SDN-assisted approach compared to JPIP over traditional networks. The JPIP server[8] implementation used in this work employs the packet pair approach to estimate the available bandwidth over conventional networks; this implementation also lets its communication link to the client go idle at times in order to correctly estimate network delay, as discussed earlier.

Fig. 3 shows that over SDN a JPIP server can use more of the available capacity, up to the full 5 Mbps, which is available on the link between the JPIP endpoints. Fig. 4 shows that using SDN also lowers the number of bytes queued inside the network; this helps improve responsiveness, because data buffered in the network needs to be delivered before any new data, which corresponds to the client's new WOI, can arrive, as explained before. Fig. 5 shows video quality improvement that is obtained from the proposed approach.

4.3.3. *Fair sharing among interactive flows*

To facilitate testing the proposed congestion control approach for a large number of clients, we developed synthetic JPIP clients and servers, which follow the proposed approach, but have a low computational cost; to emulate traffic, a synthetic server sends zero-filled packets of desired size to synthetic clients. It is important to stress that these synthetic clients and servers are used as additional clients and servers beside one real JPIP server and one or more real JPIP clients.

Fig. 6 shows the behavior of the proposed approach as multiple clients share a limited-capacity link. The figure shows that the number of queued bytes is proportional to the number of clients sharing the link; each client has approximately the same number of queued bytes, and therefore each client gets a fair share of bandwidth. Additionally, the link is fully utilized most of the time.

An important feature of JPIP is that the server needs only to send data that the client does not already have within its cache. Since interactive navigation may involve forward or backward traversal of the video, with changing window of interest, this introduces substantial user-dependent dynamics into the traffic. To simulate this behavior within our emulated JPIP servers also, we use a two-state Markov model for each client, where the ON state refers to the server having something to send and the OFF state refers to having no data to send. The model is characterized by the transition probability, which we set to 0.1; that is, $P(\mathrm{ON}|\mathrm{OFF}) = P(\mathrm{OFF}|\mathrm{ON}) = 0.1$, and $P(\mathrm{OFF}|\mathrm{OFF}) = P(\mathrm{ON}|\mathrm{ON}) = 0.9$. This model is evaluated for a transition once every 250 ms.

Fig. 7 shows that the number of queued bytes increase linearly with the number of clients receiving data; this is because the server attempts to queue up S bytes (1500 bytes in this work) for each client flow. This way each client flow gets a fair share of bandwidth because the server queues up the same amount for that flow. Fig. 7 also shows little disruption to the server's throughput when the server resumes or stops sending to clients; in fact, full utilization of the link's capacity is achieved most of the time.

[8] Kakadu Software v7.4, http://www.kakadusoftware.com/.

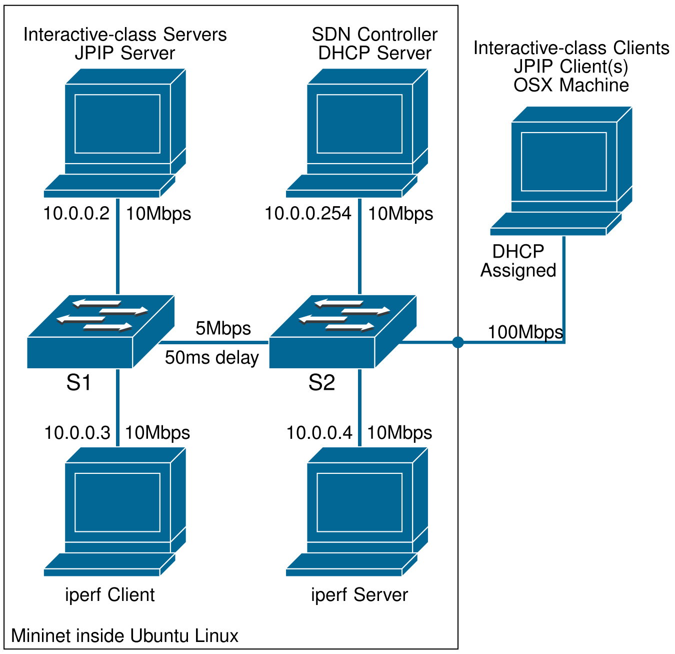


Fig. 2. The test setup used to obtain the experimental results. A Ubuntu Linux machine is used for running Mininet; inside Mininet, a network of two switches is emulated. This Ubuntu machine is physically connected to an OSX machine that runs an interactive client and any interactive-class clients. Non-interactive client (Iperf) and server are shown in the lower half of the figure.

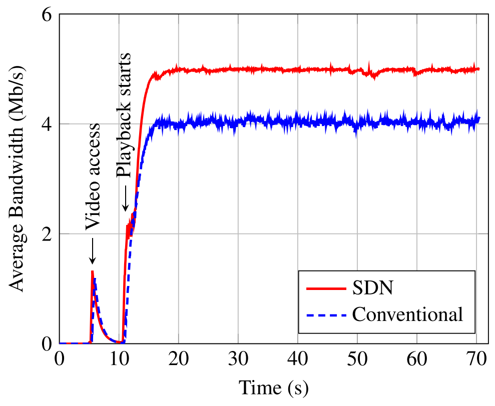


Fig. 3. JPIP-based video bandwidth over traditional network and over SDN, where the network state supplied by the SDN controller is used for congestion control. Video bandwidth over SDN uses all of the bottleneck link's 5 Mbps bandwidth. The first peak, which happens at the 5th second corresponds to first access to video, while the actual playback started around the 10th second.

5. Dynamic service policy using SDN

In the previous section, we proposed an SDN-assisted congestion control algorithm that can achieve high throughput with low latency. In that section, however, we took the available bandwidth for interactive traffic at each link to be fixed. This is reasonable only if all traffic belongs to the interactive class. In this section we investigate how interactive traffic can be intermixed with non-interactive traffic. We show in Section 5.1 that the widespread use of loss-based TCP traffic can dominate delay-sensitive interactive traffic, degrading the latter's throughput and latency. Then, in Section 5.2, we present an SDN-based approach for fairly dividing bandwidth between interactive and non-interactive flows. In Section 5.3, we examine the robustness of this bandwidth partitioning approach. Experimental results are presented in Section 5.4.

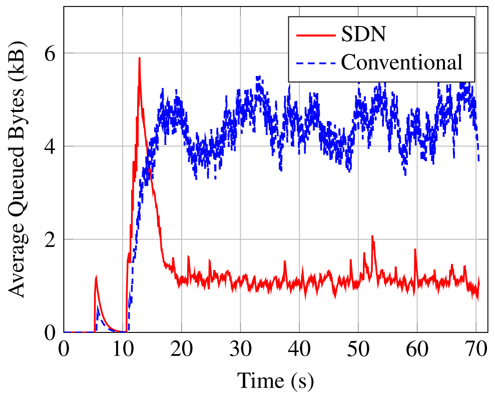


Fig. 4. In JPIP-based video browsing over traditional network, more data is buffered in the network compared to the proposed approach which uses information from the SDN controller for congestion control. The time axis here corresponds to that in Fig. 3.

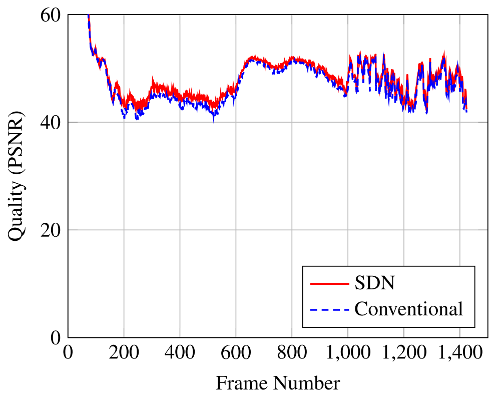


Fig. 5. A JPIP server using SDN, as proposed in this work, can deliver higher quality video by around 1.1 dB compared to a JPIP server over a traditional network.

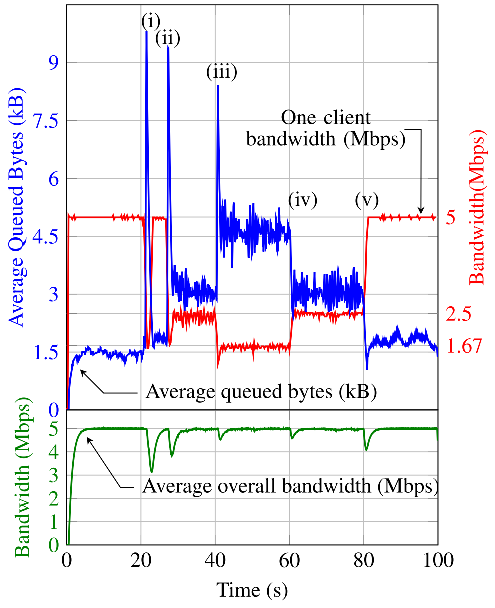


Fig. 6. Fair sharing among multiple clients sharing the 5 Mbps link of Fig. 2. (a) The figure starts with one synthetic client. At point (i) a JPIP client is connected; this point is the first video access. At (ii) video playback starts. At (iii) another synthetic client starts using the link. At (iv) the JPIP client leaves. At (v) one synthetic client leaves. The upper part of the figure shows fair bandwidth sharing among clients, and that the number of queue bytes is proportional to the number of client. The lower part of the figure shows that almost all the link's bandwidth is utilized.

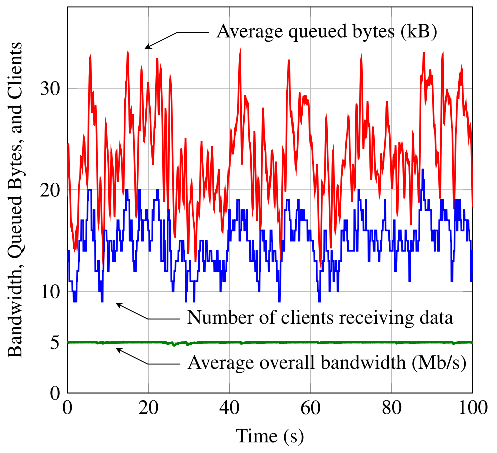


Fig. 7. In an interactive session, a JPIP server can have little data, potentially zero, to send to a client for certain frames and a lot for others. This figure shows the near full utilization of bandwidth as the number of clients for which the server sends data changes over time.

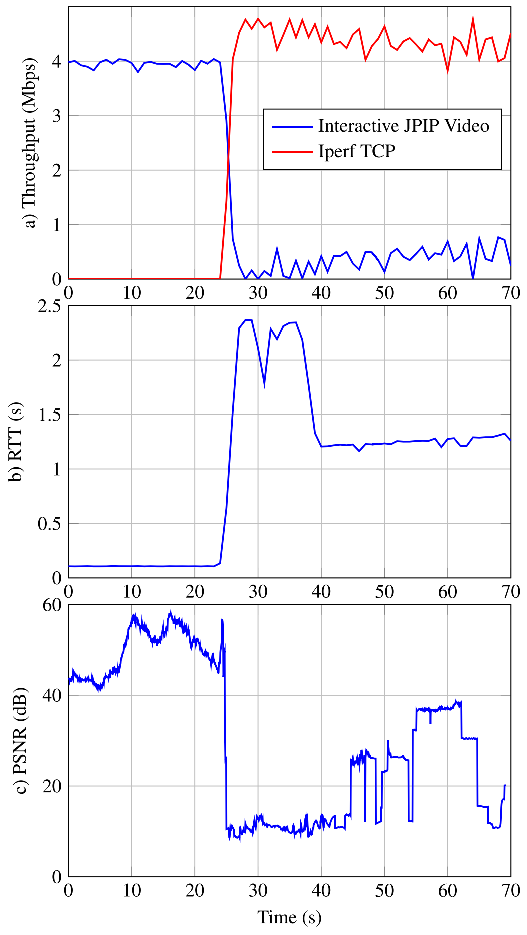


Fig. 8. Performance of an interactive JPIP video flow over a best-effort network with congestion.

5.1. Interactive flows in a traditional network

To examine the performance of interactive traffic in the presence of non-interactive traffic, we employ one best-effort queue for all traffic, interactive and non-interactive. Fig. 8(a) shows the throughput of a JPIP video flow[9] (solid blue line) and an Iperf TCP flow (dotted red line). At time $t = 0$ s, we have only one JPIP video flow; then at time $t = 24$ s, an Iperf TCP flow is introduced. In absence of other traffic (i.e. when $t \in [0, 24]$), JPIP video throughput is close to the full link's capacity. During this time, we also notice that the round-trip time experienced by the JPIP video flow, shown in Fig. 8(b), is around 106 ms, only 6ms larger than the 100 ms round-trip propagation delay. Unsurprisingly, a high quality video (PSNR of above 40 dB) is delivered to the client during this time, as shown in Fig. 8(c).

On the other hand, once an aggressive TCP flow is introduced at time $t = 24$, the bandwidth consumed by the interactive video flow drops significantly; JPIP video throughput falls below 1 Mbps and the aggressive TCP flow dominates the link's capacity, receiving more than 4 Mbps of throughput, as shown in Fig. 8(a) when $t \in [24, 70]$ s. Concurrently, the JPIP flow witnesses an increase in RTT to above 1000 ms, reaching as high as 2400 ms, as shown in Fig. 8(b). Naturally, the decrease in JPIP throughput is accompanied by a decrease in the quality of delivered video down to a PSNR of around 10dB, as depicted in Fig. 8c; this extremely low PSNR indicates that none of the data sent to the client arrives in time for it to be used in rendering.

This behavior can be understood by remembering that loss-based TCP flows aggressively fill up queuing buffers, while delay-sensitive interactive traffic attempts to minimize delay by buffering a small number of packets. Thus loss-based TCP flows cause high latency and take a large proportion of available bandwidth because they dominate queuing buffers.

5.2. Interactive flows over a network with SDN-controlled dynamic service policy

To maintain the performance of interactive video flows, and at the same time, provide other traffic with a reasonable portion of a link's capacity, we propose using two queues for each link, an interactive queue for interactive flows and a non-interactive queue for other traffic, together with a dynamic service policy that aims at fairly adjusting the minimum available rate to each of the queues. We note that this service policy is utilized by the network provider for quality control and will be independent of the congestion control mechanism that is run by the application provider.

We write $\mathcal{Q}_\mathrm{I}$ and $\mathcal{Q}_\mathrm{NI}$ for interactive and non-interactive queues, respectively. The SDN controller is able to track active flows in each queue, where we write f^I and f^NI for the number of flows in $\mathcal{Q}_\mathrm{I}$ and $\mathcal{Q}_\mathrm{NI}$.

In this work, we consider a fair distribution of a link's capacity C between these two queues to occur when each queue gets a share of the capacity that corresponds to the number of flows utilizing it. In practice, we use the ratio $(f^\mathrm{I} + 1) : (f^\mathrm{NI} + 1)$ for this purpose; this ratio is the MAP (maximum a posteriori probability) estimate of the ratio of the flows, given unscaled counts f^I and f^NI, and no prior information is available concerning the actual distribution of flows. Other definitions of fairness are conceivable, and we may explore some of them in future work.

The SDN controller uses this definition of fairness to provision a *long-term* available rate $\hat{R}_\mathrm{L}$ for the interactive queue $\mathcal{Q}_\mathrm{I}$ at all switches along the interactive flow path. It also provisions the non-interactive queue[10] $\mathcal{Q}_\mathrm{NI}$ with the rest of the available capacity, given by $C - \hat{R}_L$. The SDN controller updates these provisioned rates sufficiently frequently to adapt to the changing state of the network. The value of $\hat{R}_L$ is computed using an exponential average with a small parameter α_L, and is given by:

$$\hat{R}_\mathrm{L}^i = (1 - \alpha_\mathrm{L}) \cdot \hat{R}_\mathrm{L}^{i-1} + \alpha_\mathrm{L} \cdot C \cdot \frac{f^\mathrm{I} + 1}{f^\mathrm{I} + f^\mathrm{NI} + 2} \tag{9}$$

The use of exponential averaging helps smooth sudden transitions in the provisioned rate. The network provider may choose to use other criteria for the long-term rate, such as putting a lower limit on the rate allocated to interactive flows.

Here, we are also interested in enabling interactive flows to utilize any spare bandwidth that may occur in the non-interactive queue, but we do not want interactive applications to resort back to probing, since, as we have seen, probing is inefficient. For this purpose, the SDN controller also estimates a short-term rate $\hat{R}_\mathrm{S}$; this is the rate communicated to interactive application endpoints (e.g. a JPIP server) as is described in Section 3.2.2. The short-term rate represents a short-time guarantee of what rate interactive flows can potentially achieve.

The short-term rate $\hat{R}_\mathrm{S}$ gradually decreases to the long-term rate $\hat{R}_\mathrm{L}$ when there are queued packets q^NI in the non-interactive queue $\mathcal{Q}_\mathrm{NI}$, and gradually increases to $C - M$ in their absence, where C is the link's capacity and M is a small bandwidth margin chosen by the network operator. The bandwidth margin M enables the SDN controller to check if the non-interactive queue is being utilized; i.e., there is non-interactive traffic on the network. The short-term rate $\hat{R}_\mathrm{S}$ is given by

$$\begin{aligned}\hat{R}_\mathrm{S}^i &= (1 - \alpha_\mathrm{S}) \cdot \hat{R}_\mathrm{S}^{i-1} \\ &\quad + \alpha_\mathrm{S} \cdot \left(\left[1 - \mathcal{I}(q^\mathrm{NI})\right] \cdot (C - M) + \mathcal{I}(q^\mathrm{NI}) \cdot \hat{R}_\mathrm{L}^{i-1}\right)\end{aligned} \tag{10}$$

where $\mathcal{I}$ is a binary indicator function; that is, it is equal to 1 when x is nonzero and 0 otherwise. The parameter α_S controls how fast the short term rate $\hat{R}_\mathrm{S}$ responds to packet queuing in the non-interactive queue $\mathcal{Q}_\mathrm{NI}$; to make this short-term rate $\hat{R}_\mathrm{S}$ responds faster to changes than the long-term rate $\hat{R}_\mathrm{L}$, the short-term rate parameter α_S should be larger than the corresponding long-term parameter α_L.

5.3. Robustness to overstating the short term rate $\hat{R}_S$

For interactive endpoint applications, the congestion window W_f^i of (8) depends on the short term rate $\hat{R}_\mathrm{S}$ through (4), (5), (6), and (7). In this section, we show that an interactive queue is not congested when the short-term rate $\hat{R}_\mathrm{S}$, communicated by the SDN controller, temporarily exceeds the actual rate available to that interactive queue; this can occur when the utilization of the corresponding non-interactive queue increases suddenly, keeping in mind that an interactive queue is provisioned to guarantee a rate of at least $\hat{R}_\mathrm{L}$, not $\hat{R}_\mathrm{S}$.

Consider the case of N interactive flows passing through one such interactive queue. The server or servers of these flows attempt to queue an average of S bytes of data per flow in this queue; thus, this interactive queue should have $N \cdot S$ bytes queued, as we have seen before. If the actual available bandwidth for this queue is temporarily lower than the value reported by the SDN-exposed API (i.e. $\hat{R}_\mathrm{S}$), packets will accumulate within the queue, say $N \cdot \alpha \cdot S$ bytes are queued, where $\alpha > 1$. Using the SDN-exposed API, end-point servers notice this increase in data packets; each server correctly identifies, using (3), that a fraction $1/N$ of the accumulated bytes belong to it, i.e. $\alpha \cdot S$ bytes. The subsequent computation of available rate, via (4), is based on the assumption that each other server has queued only S bytes, yielding

$$\lambda_{f,l}^i = \frac{S}{S + (N - 1) \cdot \alpha \cdot S} \cdot \hat{R}_\mathrm{S}$$

[9] A JPIP server over a traditional network uses the packet pair approach to estimate the available bandwidth, letting its communication link to the client go idle at times to correctly estimate network delay. This is the same JPIP server used over conventional networks in Section 4.3.2.

[10] It is conceivable that the SDN controller provisions other queues, but all these other queues belong to the non-interactive class, and therefore the SDN is free to choose how to partition the available bandwidth $C - \hat{R}_L$ among them.

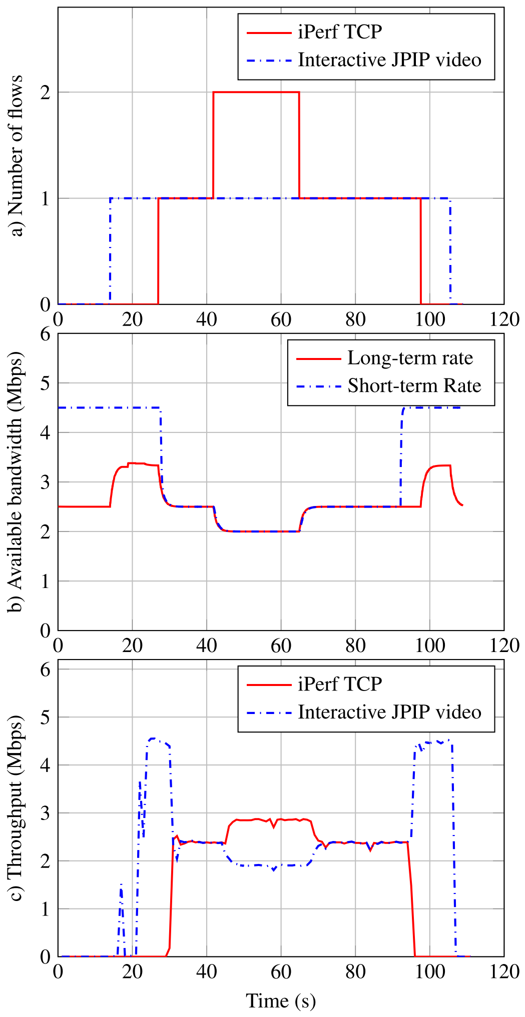


Fig. 9. Network performance of an interactive JPIP video flow over an SDN-enabled network with congestion.

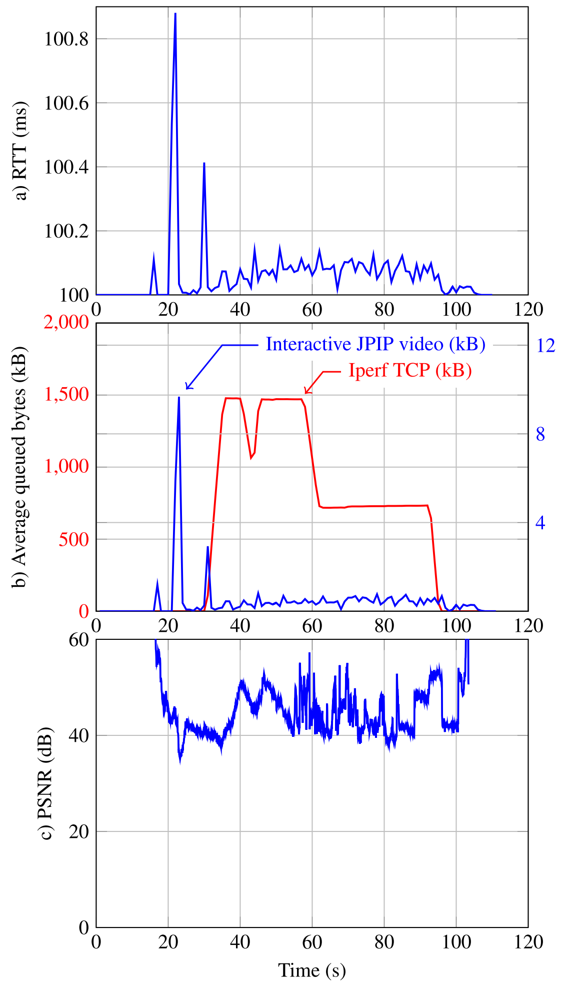


Fig. 10. Application performance of an interactive JPIP video flow over an SDN-enabled network with congestion. Note that for (b) different y-axis scales are used for different colors.

which is always lower than $\hat{R}_S/N$, approaching $\hat{R}_S/(N \cdot \alpha)$ as N becomes large. This effectively means that the congestion control algorithm substantially compensates for inaccuracies in the rate $\hat{R}_S$, avoiding large increases in latency, especially in the critical case where many servers are competing for the available bandwidth.

We remind the reader that $\hat{R}_S$ will be larger than $\hat{R}_L$ only when the SDN-controller considers that the non-interactive traffic is temporarily under-utilizing its fair share of available bandwidth. Once the SDN-controller responds to the sudden increase in utilization by non-interactive traffic, it should report a lower short term rate $\hat{R}_S$, approaching the long term rate $\hat{R}_L$ if the increased utilization persists.

5.4. Performance evaluation of the dynamic service policy

We evaluate the proposed dynamic service policy on the same test setup shown in Fig. 2. To test the performance of the proposed scheme, we start a JPIP flow at $t = 14$ s and stop it at $t = 104$ s. During that time, we also employ Iperf flows following the pattern 0, 1, 2, 1, 0; Fig. 9(a) shows the number of flows on the link between S1 and S2 of Fig. 2. We expect the long-term bandwidth, which is the minimum bandwidth available to interactive flows, to converge to 2.5 Mbps when there are no flows, 3.3 Mbps when JPIP flow is the only flow, 2.5 Mbps when we have one JPIP flow and one Iperf flow, and so on, as given by (9). The solid line in Fig. 9(b) shows that the experimental results corroborate this expectation. The dotted line in Fig. 9(b) shows the short-term bandwidth; this is the rate exposed to interactive applications by the SDN controller via the query API call.

Fig. 9(c) shows that JPIP flow throughput (dotted line) quickly tracks the reported short-term bandwidth, and the rate of the interactive queue (and JPIP flow throughput accordingly) is adapted as the number of TCP flows changes. For example, at about $t = 30$s, JPIP throughput is approximately halved after introducing an Iperf TCP flow and after ten seconds it again slightly drops when the second Iperf TCP arrives. This behavior is a result of the SDN-controller reducing the minimum available bandwidth allocated to interactive queue as more flows enter the default queue. The network controller may choose a policy whereby the bandwidth allocated to interactive flows cannot drop below a certain fraction of the total link capacity, but this is not the case here.

Fig. 10 shows the performance of the interactive video application under the proposed approach by a number of metrics. The RTT, shown in Fig. 10(a), is minimal at 101 ms or less, compared to a round trip propagation delay of 100 ms; a minimal RTT enables interactive end-points to quickly respond to client changes, and as such improves responsiveness. In Fig. 10(b), we see that the interactive queue holds around a kilobyte of JPIP data during video playback, because the SDN-assisted congestion control, presented in Section 4, attempts to have 1500 bytes per flow of interactive data (e.g. JPIP data), even when the interactive queue buffer is configured to hold a thousand packets. For the default non-interactive queue, even a single Iperf TCP flow can quickly fill up this queue's buffer, as shown by the dotted red line in Fig. 10(b) between $t = 35$ s and $T = 40$ s, where around 1.5 MB are queued. Finally, Fig. 10(c) shows the high-quality of the delivered JPIP video, averaging around 40 dB and never falling below 35 dB; most importantly, it is not significantly harmed by other traffic as is the case in Section 5.1, Fig. 8(c). These results are considerably better for interactive applications, compared to the results of Section 5.1, and in particular Fig. 8(b).

Summary: With the help of an SDN-controlled dynamic service policy, interactive JPIP flows can achieve the desired low-latency and high-throughput, while other traffic still experiences a decent share of a link's capacity.

6. Extending SDN-assisted congestion control to federated networks

In Section 4, we presented an SDN-assisted congestion control algorithm; all discussions and results in that section are limited to one domain, suitable for one network provider. In this section, we explore how the approach can be extended to multiple service providers.

6.1. The proposed approach

To employ the proposed SDN-assisted congestion control algorithm through multiple service providers, these providers need to exchange their network state information that is necessary for the operation of this algorithm. The pricing mechanisms that might be used in such a federated service are beyond the scope of this work.

In this work, a provider's SDN controller[11] collects network state information from other providers for all interactive flows originating from or terminating into its own network, consolidates this information with its own network state information, and makes this information available to its own interactive end-points and other providers' SDN controllers; thus, a provider's SDN controller also acts as a proxy for other providers' state information, but only for interactive flows that originate from or terminate into its network. This way, it is sufficient for an SDN controller to only query the SDN controller of the immediate network provider on the path of an interactive flow.

To see how this works, consider the interactive end-point shown in Fig. 11, represented by the JPIP server in ISP1. This end-point needs to only query its own SDN controller about the network state information for its interactive flow to ISP4; the SDN controller of ISP1 should be able to supply this end-point with the state information corresponding to its flow, from data it already has. The SDN controller of ISP1 obtains relevant state information for other providers' networks by querying its adjacent network (i.e., the SDN controller in ISP2) about the state of relevant interactive flows between ISP2 and ISP4 and from ISP2 to itself; the SDN controller of ISP1 consolidates this state information with that of its own network, and makes it available to its clients and SDN controllers of other network providers.

The advantage of this approach is that communication of network state information is done only at the local level; switches and end-points communicate only with their local controller, and controllers communicate only with the nearest adjacent controller. Interrogating all controllers along the path of a flow would incur significantly more bandwidth, even though it might allow more immediate (lower latency) responses.

[11] Conceptually, every provider has one SDN controller; if more than one controller exists within a provider, then these controllers can exchange network state information using the proposed approach.

An SDN controller does not issue network state information requests until an interactive flow is established; it then continues to issue state requests until all such flows are closed. In this work, a polling mechanism is employed, using the RESTful API of Section 3.2.2, to obtain state information, but it is also conceivable that other more efficient mechanisms might be used, such as streaming telemetry, whereby network state information is streamed from an SDN controller to other SDN controllers that have enrolled (or subscribed) to receive this information.

To allow interactive flows to mix with other flows, each network provider needs to implement the dynamic service policy of Section 5; this policy does not need any network state information from other providers, so each network implements the policy independently, reporting the short-term rate for its switches as part of its network state information.

An SDN controller does not have to reveal network state information for all of its internal switches; instead, it is sufficient to provide visibility into switches that can delay traffic or limit the capacity available to interactive flows.

6.2. Consolidation of network state information

When an SDN controller generates a new network state entry, the entry has a new index, and a concatenation of link entries from its own network and from other networks; the format of a network state entry is described in Section 3.2.2.

It is possible that SDN controllers collect, and possibly consolidate, network state information at different frequencies. Therefore, when a network state entry is created in one controller, it is possible that no new information is available for a link l that lies within a different network provider; it is also possible that multiple new link entries are available from the other provider's controller. If there are multiple entries, indexed by $j \in J$, then the combined entry indexed by i has $\Delta_l^i = \sum_{j \in J} \Delta_l^j$, $b_l^i = \sum_{j \in J} b_l^j$, $q_l^i = q_l^k$, $R_l^i = R_l^k$, $d_l^i = d_l^k$, where q_l^k, R_l^k, and d_l^k are the most recently known number of queued bytes, rate, and propagation delay for link l. In the absence of any new entry for link l, we generate an entry indexed by i that has $\Delta_l^i = 0$, $b_l^i = 0$, $q_l^i = 0$, $R_l^i = R_l^k$, $d_l^i = d_l^k$.

The interactive end-points ignore entries that have Δ_l^i equal to zero, because such entries carry no recent information; they continue using their latest estimates of the average bandwidth $\bar{\lambda}_l^i$ and number of queued bytes $\bar{q}_l^i$. For the other entries, (1) and (2) are used as in Section 4.

6.3. Experimental results

In this section, we show that the proposed SDN-assisted congestion control algorithm can successfully provide low latency and high throughput across multiple network operators. We emulate each ISP in the federated network shown in Fig. 11 using an Ubuntu virtual machine; these machines are run inside an instance of VMWare Workstation Pro. Each ISP's network is emulated using Mininet, with three routes, a few switches, and multiple hosts. We configure each peering link to have a 50 ms delay, and a 5 Mbps capacity, giving a minimum RTT delay of 300 ms; links within a network provider are assumed to have sufficient capacity, not to interfere with results. Network state information is collected and exchanged every 100 ms.

6.3.1. Experiment 1

At time $t = 0$ s, one interactive client at ISP4 starts communicating with an interactive server at ISP1. At $t \approx 18$ s, another interactive client at ISP3 starts communicating with the interac-

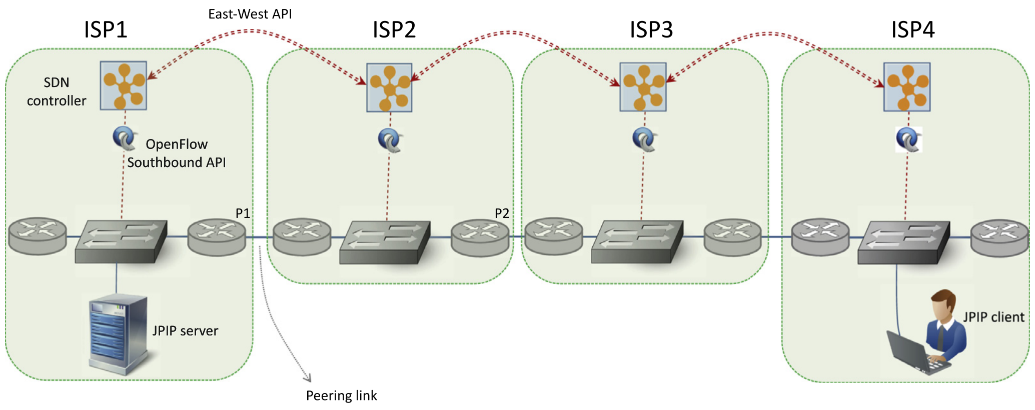


Fig. 11. Interactive flows over a federated network.

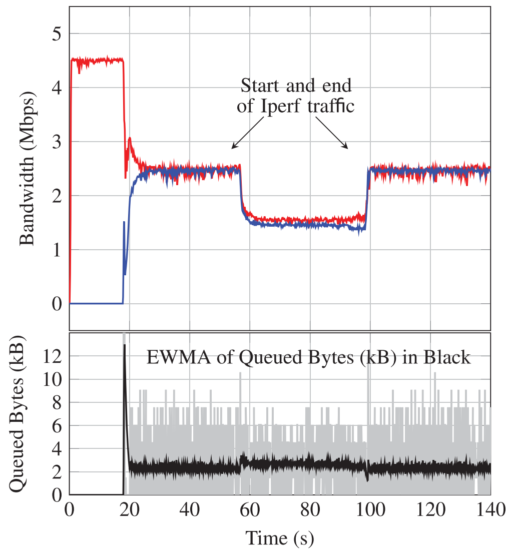


Fig. 12. Bandwidth for two interactive clients and the number of queued bytes at P1 of Fig. 11 when east-west link propagation delays match that of the peering link delays. Flows from these two interactive clients compete with an Iperf flow at P1, starting at $t = 25$s.

tive server at ISP1. At $t = 58$ s, an Iperf client[12] connected to ISP1 starts communicating with an Iperf server connected to ISP3. Fig. 12 shows results at port P1, which is the interface between ISP1 and its peering link to ISP2, when east-west link delays match the peering link propagation delay of 50 ms. In this scenario, P1 is the buffer at which flows compete for bandwidth; network state information from P1 can be up to 100 ms old or slightly more.

6.3.2. Experiment 2

At time $t = 0$ s, one interactive client at ISP4 starts communicating with an interactive server at ISP1. At $t \approx 18$ s, another interactive client at ISP3 starts communicating with the interactive server at ISP1. At $t = 58$ s, an Iperf client connected to ISP2 starts communicating with an Iperf server connected to ISP3. Fig. 13 shows the bandwidth observed by the two interactive clients. It also shows the number of queued bytes at port P1 and P2, which is the interface between ISP2 and its peering link to ISP3.

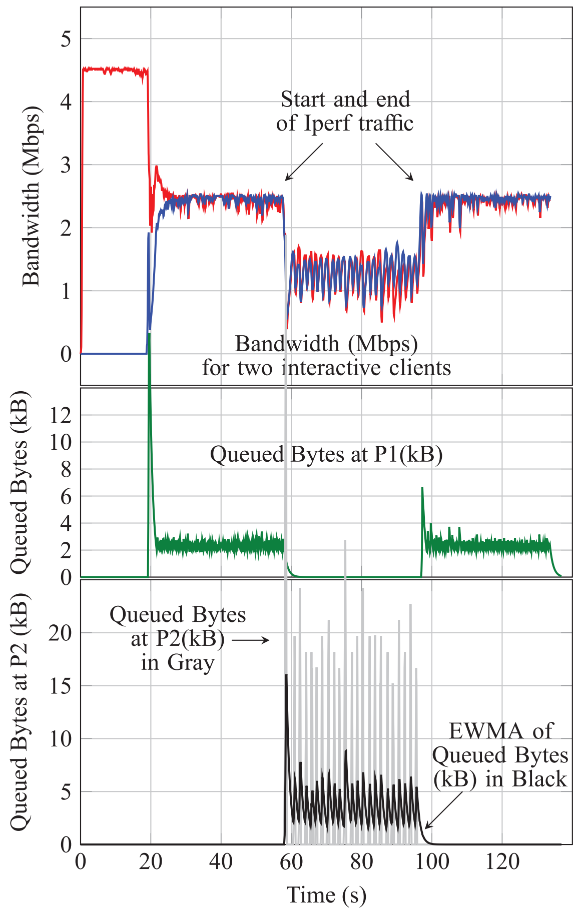


Fig. 13. Bandwidth for two interactive clients and the number of queued bytes at P1, and P2 of Fig. 11 when east-west link propagation delays match that of the peering link delays. Flows from these two interactive clients compete with an Iperf flow at P2, starting at $t = 58$ s.

It can be seen that, during the period $t \in [58, 98]$, the bandwidth utilized by interactive clients fluctuates, with an average per client bandwidth of 1.25 Mbps, which is lower than the ideal case of 1.5 Mbps. Moreover, the number of queued bytes is peaky with an average of around 2kB per client instead of ideally being 1.5kB. This occurs when the network state information for P2 is of the order of 200 ms old. The reason behind this fluctuation is explored in Section 6.4; additionally, that section discusses the implications of larger network delays and proposes approaches to reduce band-

[12] For interactive flows, the server generates data streams, while, for Iperf, it is the client.

width fluctuations. In any case, the presence of these bandwidth fluctuations in high delay configurations such as this one, does not undermine the efficacy of the proposed methods, but it does suggest an opportunity for future research and mathematical modeling for federated systems.

6.4. Network delay and interactive applications

For the interactive applications considered in this work, the latency between a user's action and the response to this action should be small. Pantel and Wolf [23] experimental results for interactive real-time racing show that it is best to keep latency at no more than 150 ms; at 200 ms, the racing experience becomes unrealistic. Similar results are obtained by Waltemate et al. [24]; they found that "simultaneity," which is the feeling that the action and its response are occurring at the same time, starts declining between 125 ms and 210 ms. This latency includes network delay and any data processing performed at the server and client; therefore, for truly interactive applications, such as virtual reality and 360° video streaming, network delay should be of the order of 150 ms or less.

The bandwidth fluctuations in Section 6.3.2 occur because of a rather aggressive policy for adjusting the congestion window (4) and (8). For modest network delays, as considered in Section 6.3.1, the server can observe the response to its adjustment of the congestion window in a short period of time, and therefore can adapt these adjustments to the new network state. For the rather long network delay considered in Section 6.3.2, the server can only observe the network state after a longer period of time, and therefore it can only provide a rather late response. To reduce fluctuations, the server should make smaller adjustment to the congestion window; i.e., it should be less aggressive. We next present a couple of modifications that reflect this strategy.

6.4.1. Limiting the smallest number of queued bytes to S

A simple strategy to limit the maximum size of the congestion window is to make the number of queued bytes $\bar{q}_l^i$ no smaller than S, which is the smallest meaningful value; that is, we change (3) to

$$\hat{q}_{f,l}^i = \begin{cases} \frac{\hat{\lambda}_f^i}{\bar{\lambda}_l^i} \cdot \max\{\bar{q}_l^i, S\}, & \bar{\lambda}_l^i \geq \hat{\lambda}_f^i \ \&\ \bar{\lambda}_l^i > 0 \\ \max\{\bar{q}_l^i, S\}, & \text{otherwise} \end{cases}, \quad l \in \Lambda_{S\to C} \tag{11}$$

and (4) to

$$\lambda_{f,l}^i = \frac{S}{S + \left(\max\{\bar{q}_l^i, S\} - \hat{q}_{f,l}^i\right)} \cdot R_l^i, \quad l \in \Lambda_{S\to C} \tag{12}$$

In the case where there are two sources sharing the bottleneck link, with $\hat{\lambda}_f^i/\bar{\lambda}_l^i \approx 0.5$, this modification has the effect of limiting $\lambda_{f,l}^i$ to at most $R_l^i/1.5$, which significantly reduces excursions in the congestion window, considering that the ideal value for $\lambda_{f,l}^i$ would be $R_l^i/2$ in this case. Fig. 14 shows the experimental results when $\bar{q}_l^i$ is limited to S for the same setup used in Section 6.3.2. Evidently, both the bandwidth and number of queued bytes have become substantially closer to their ideal values. We also observe that bandwidth fluctuations reduce over time. We stress the fact that these experimental results involve two interactive sources that share a common path. In applications where many sources interact incoherently, we expect the number of queued bytes to be both larger and more stable. The modification represented by (11) and (12) would have no impact in such cases. Fig. 7 shows the case when flows from many clients interact.

6.4.2. Limiting the changes to the congestion window

The modification above imposed an entirely justifiable lower bound on the value of $\bar{q}_l^i$ used in the derivation of each flow's rate

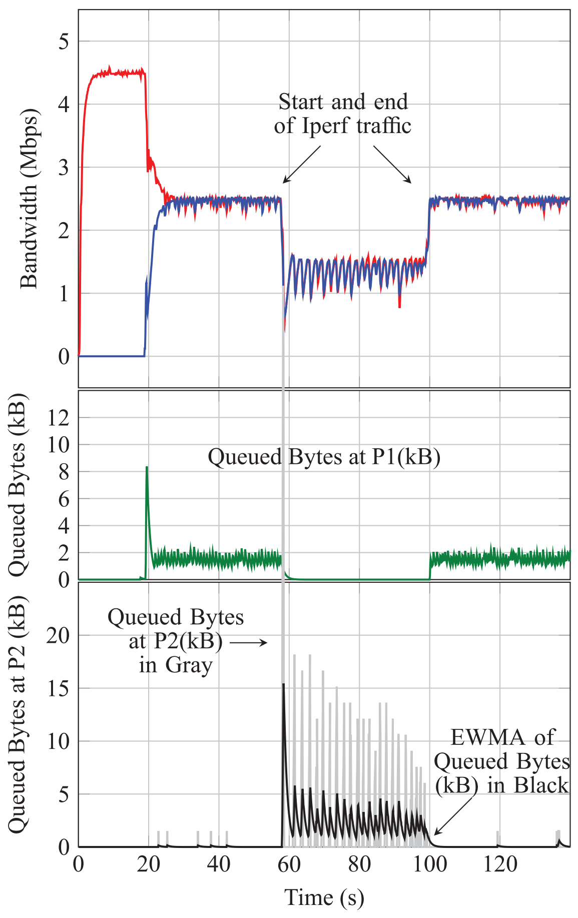


Fig. 14. Experimental results when the minimum number of queued bytes are limited to S. The same setup of Section 6.3.2 is used.

and hence window size, with the effect that the window size for any given flow is bounded above in a manner that depends on the observed flow ratio $\hat{\lambda}_f^i/\bar{\lambda}_l^i$. We can take this idea further by imposing both lower and upper bounds on the window size, dynamically adjusted based on the observed flow ratio, as well as the network delay. We write τ_l for the time it takes for the interactive server to observe the influence of the changes it made to the congestion window on the number of queued bytes at link l; this includes propagation delays and any processing delays for the network state information. Then, the upper bound on the congestion window is:

$$U_{f,l}^i = \frac{\hat{\lambda}_f^i}{\bar{\lambda}_l^i} \cdot R_l^i + \max\left\{0, (R_l^i - \bar{\lambda}_l^i) \cdot \frac{\hat{\lambda}_f^i}{\bar{\lambda}_l^i}\right\} + \frac{n \cdot S}{\tau^i} \tag{13}$$

The first term represents a fraction of the link's capacity R_l^i that is suggested by the observed flow ratio, noting that this flow ratio may have been based on a different value for the capacity R_l^i, in periods when the SDN controller is changing the distribution of bandwidth between interactive and non-interactive flows, as explained in Section 5. The second term enables the server to utilize any available spare capacity on the link in the same proportion; the maximum operator prevents this term from becoming negative. The last term limits how many additional bytes the server can queue during τ_l, the time needed by the server to observe the newly queued bytes at link l. Choosing a high value for n enables faster response to the increase in available capacity but the number of queued bytes becomes more "peaky"; thus, n is a tunable parameter that might be determined by the network operator.

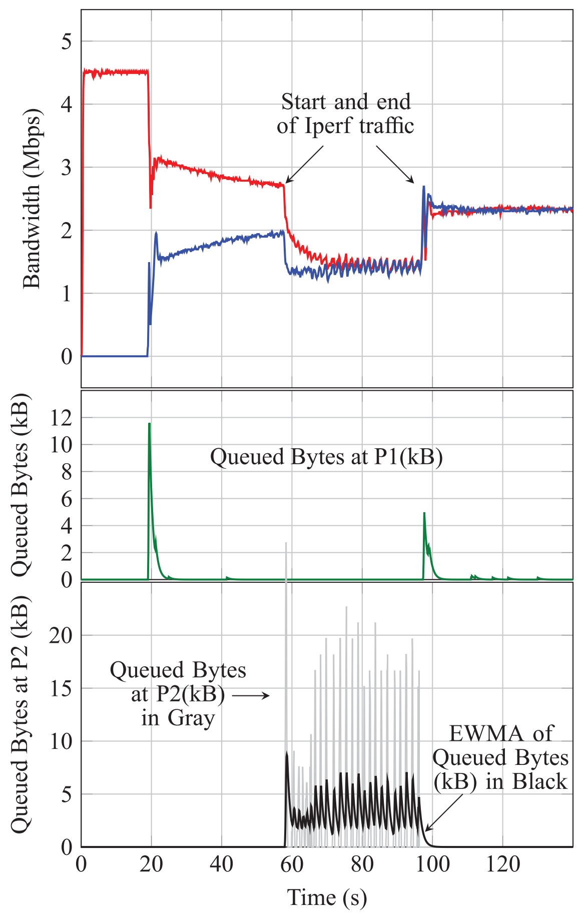


Fig. 15. Experimental results when the change in the congestion window is limited. The same setup of Section 6.3.2 is used.

A lower bound can also be deduced using similar reasoning. Here, we use

$$V^{i}_{f,l} = \frac{\hat{\lambda}^{i}_{f}}{\bar{\lambda}^{i}_{l}} \cdot R^{i}_{l} + \frac{S - \bar{q}^{i}_{f} - m \cdot S}{\tau_l} \tag{14}$$

The first term is similar to that above while the second term limits the drop in the number of queued bytes during τ_l; the drop is limited to $S - \bar{q}^{i}_{f} - m \cdot S$. When m is zero the number of queued bytes can drop to S bytes. When m is greater than zero, the drop can be larger. Larger m enables faster drop of the rate, such as when a new client joins the queue. Similar to n, the choice of m can be left to the network operator.

For both the upper and lower limits, we can see that larger changes to the congestion window are possible when τ is small.

Fig. 15 shows the experimental results for the approach of this section for the same setup used in Section 6.3.2 with the conservative choice of $n = 2$ and $m = 2$. The figure shows that when the second client joins the interactive server at $t = 18$ s, we have a slower change towards equal bandwidth sharing between the two clients. The figure also shows that when the iPerf flow starts at $t = 58$ s, there is almost no fluctuation in client bandwidths.

It is useful to keep in mind that these results represent worst case scenarios, because, for realistic deployment of the proposed approach, we expect to have many clients. In the presence of many clients, congestion window changes that are made to accommodate a single client should have a relatively smaller effect. We also expect that a link's bandwidth is shared among many clients from many servers, and therefore, they pose requests that are temporally incoherent; i.e., it is very unlikely that a large portion of these requests occur or disappear at the same time.

7. Conclusions

There is an emerging need for the management of high throughput real-time traffic, with JPIP as the prima facie example in this paper.

In this work, we have proposed an SDN-based architecture that offers real-time network services to interactive applications, such as remote media browsing. Interactive applications need high bandwidth with low latency; however, their required bandwidth fluctuates, depending on the end-user's action and the amount of data that needs to be delivered to satisfy the response to this action.

We have also presented a RESTful API that exposes the network state to interactive applications. These applications can employ this API in a congestion control algorithm that achieves full or near full utilization of bandwidth and fair sharing among interactive client, all while having a minimal number of queued data packets in the network (i.e. minimal latency).

The work also presents an SDN-controlled dynamic service policy that aims at dynamically and fairly sharing network resources between interactive flows and other traffic, without a reservation protocol.

We have evaluated the efficacy of our architecture and algorithms using real JPIP endpoints in a Mininet environment. Our results show that low-latency high-throughput interactive applications are possible on an SDN-enabled network while other traffic still experience an appropriate share of throughput.

We have also shown how the approach can be extended to federated networks, where we have presented experimental results to show the viability of the approach.

One area of future work is to replace the proposed RESTful API with a more efficient protocol, such as streaming telemetry information. Another area of future work lies in the development of even more effective strategies to control flow rates and queue sizes in high latency federated networks.

References

[1] ISO/IEC 15444-9, Information technology – JPEG 2000 image coding system – Part 9: interactivity tools, APIs and protocols, 2004.
[2] S. Gudumasu, E. Asbun, Y. He, Y. Ye, Segment scheduling method for reducing 360° video streaming latency, Applications of Digital Image Processing XL, 2017.
[3] M. Podlesny, C. Williamson, Providing fairness between tcp newreno and tcp vegas with rd network services, in: 2010 IEEE 18th International Workshop on Quality of Service (IWQoS), 2010, pp. 1–9, doi:10.1109/IWQoS.2010.5542752.
[4] B. Briscoe, K. De Schepper, M. Bagnulo Braun, Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture, Technical Report, IETF, 2017.
[5] R.L. Carter, M.E. Crovella, Measuring bottleneck link speed in packet-switched networks, Perform. Eval. 27–28 (0) (1996) 297–318.
[6] J. Navratil, R.L. Cottrell, Abwe: a practical approach to available bandwidth estimation, in: Proc. of Passive and Active Measurement (PAM), 2003.
[7] N. Hu, P. Steenkiste, Evaluation and characterization of available bandwidth probing techniques, IEEE J. Sel. Areas Commun. 21 (6) (2003) 879–894, doi:10.1109/JSAC.2003.814505.
[8] A. Ferguson, A. Guha, C. Liang, R. Fonseca, S. Krishnamurthi, Participatory Networking: an API for Application Control of SDNs, in: Proc. ACM SIGCOMM, 2013. Hong Kong, China
[9] V. Sivaraman, T. Moors, H. Habibi Gharakheili, D. Ong, J. Matthews, C. Russell, Virtualizing the access network via open apis, in: Proc. of ACM CoNEXT, 2013. Santa Barbara, California, USA
[10] H. Habibi Gharakheili, V. Sivaraman, A. Vishwanath, L. Exton, J. Matthews, C. Russell, SDN APIs And models for two-Sided resource management in broadband access networks, IEEE Trans. Netw. Serv. Manag. 13 (4) (2016) 823–834, doi:10.1109/TNSM.2016.2615067.
[11] M. Ghobadi, Y. Cheng, A. Jain, M. Mathis, Trickle: rate limiting youtube video streaming., in: Proc. of USENIX Annual Technical Conference, 2012. Boston, MA, USA.
[12] G. Wang, T.E. Ng, A. Shaikh, Programming your network at run-time for big data applications, in: Proc. of the First Workshop on Hot Topics in Software Defined Networks, 2012. Helsinki, Finland.

[13] M. Ghobadi, S.H. Yeganeh, Y. Ganjali, Rethinking end-to-end congestion control in software-defined networks, in: Proc. of ACM Workshop on Hot Topics in networks, 2012. Redmond, Washington, USA.

[14] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al., TIMELY: RTT-based congestion control for the datacenter, in: Proc. of ACM SIGCOMM, 2015. London, United Kingdom.

[15] C. Lee, C. Park, K. Jang, S. Moon, D. Han, Accurate latency-based congestion feedback for datacenters, in: Proc. of USENIX ATC, 2015. Santa Clara, CA, USA.

[16] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data center tcp (dctcp), SIGCOMM Comput. Commun. Rev. 41 (4) (2010).

[17] L.S. Brakmo, S.W. O'Malley, L.L. Peterson, TCP Vegas: new techniques for congestion detection and avoidance, SIGCOMM Comput. Commun. Rev. 24 (4) (1994) 24–35, doi:10.1145/190809.190317.

[18] D. Wei, C. Jin, S. Low, S. Hegde, FAST TCP: Motivation, architecture, algorithms, performance, Netw. IEEE/ACM Trans. 14 (6) (2006) 1246–1259, doi:10.1109/TNET.2006.886335.

[19] J.D.C. Little, S.C. Graves, Little's Law, Springer US, Boston, MA, pp. 81–100. 10.1007/978-0-387-73699-0_5

[20] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proc. of ACM SIGCOMM Workshop on Hot Topics in Networks, ACM, New York, NY, USA, 2010.

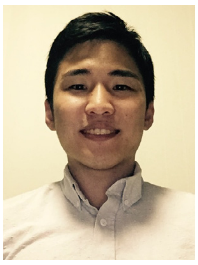
[21] Roosendaal, T. (Producer), Sintel. Blender Foundation, Durian Open Movie Project (2010), 2010.

[22] D. Taubman, R. Prandolini, Architecture, philosophy and performance of JPIP: internet protocol standard for JPEG 2000, Int. Symp. Visual Comm. and Image Proc. 5150 (2003) 649–663.

[23] L. Pantel, L.C. Wolf, On the impact of delay on real-time multiplayer games, in: Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV '02, ACM, New York, NY, USA, 2002, pp. 23–29, doi:10.1145/507670.507674.

[24] T. Waltemate, I. Senna, F. Hülsmann, M. Rohde, S. Kopp, M. Ernst, M. Botsch, The impact of latency on perceptual judgments and motor performance in closed-loop interaction in virtual reality, in: Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology, VRST '16, ACM, New York, NY, USA, 2016, pp. 27–35, doi:10.1145/2993369.2993381.

Aous Thabit Naman (SM'16) received the B.Sc. degree in Electronics and Telecommunication Engineering from Al-Nahrain University, Baghdad, Iraq, in 1994, the M.Eng.Sc. degree in Engineering from University of Malaya, Kuala Lumpur, Malaysia, in 2000, and the Ph.D. degree in Electrical Engineering from the University of New South Wales (UNSW), Sydney, Australia, in 2011. He is currently pursuing postdoctoral research with the School of Electrical Engineering and Telecommunications, UNSW. His research interests are in image/video compression and delivery. He has served as a reviewer for many journals and conferences, such as the IEEE Transactions on Image Processing, IEEE Signal Processing Letters, and the International Conference on Image Processing.



Yu Wang received his Bachelor's and Master's degrees in Electrical Engineering and Telecommunications from the University of New South Wales in Sydney, Australia in 2013 and 2017 respectively. His research interests include automated network technologies and software defined networking.

Hassan Habibi Gharakheili received his B.Sc. and M.Sc. degrees in Electrical Engineering from the Sharif University of Technology in Tehran, Iran in 2001 and 2004 respectively, and his Ph.D. of Electrical Engineering and Telecommunications from the University of New South Wales in Sydney, Australia in 2015. He is currently a Postdoc Research Fellow in the School of Electrical Engineering and Telecommunications at the University of New South Wales. His research interests include network architectures, software-defined networking and broadband networks.

Vijay Sivaraman (M'94) received his B. Tech. degree from IIT in Delhi, India, in 1994, his M.S. from North Carolina State University in 1996, and his Ph.D. from the University of California at Los Angeles in 2000, all in Computer Science. He has worked at Bell-Labs and a Silicon Valley startup. He is now a Professor at the University of New South Wales in Sydney, Australia. His research interests include software-defined networking, and sensor networks for environmental and health applications.

David Taubman (M'92) received the B.S. and B.Eng. degrees from the University of Sydney, Sydney, Australia, in 1986 and 1988, respectively, and the M.S. and Ph.D. degrees from the University of California, Berkeley, in 1992 and 1994, respectively. From 1994 to 1998, he was with Hewlett–Packard's Research Laboratories, Palo Alto, CA, joining the University of New South Wales, Sydney, in 1998, where he is a Professor with the School of Electrical Engineering and Telecommunications. He is the coauthor, with M. Marcellin, of the book *JPEG2000: Image Compression Fundamentals, Standards and Practice* (Boston, MA: Kluwer, 2001). His research interests include highly scalable image and video compression, inverse problems in imaging, perceptual modeling, joint source/channel coding, and multimedia distribution systems. Dr. Taubman was awarded the University Medal from the University of Sydney; the Institute of Engineers, Australia, Prize; and the Texas Instruments Prize for Digital Signal Processing, all in 1998. He has received two Best Paper awards, one from the IEEE Circuits and Systems Society for the 1996 paper, "A Common Framework for Rate and Distortion Based Scaling of Highly Scalable Compressed Video," and from the IEEE Signal Processing Society for the 2000 paper, "High Performance Scalable Image Compression with EBCOT."