InSite: QoE-Aware Video Delivery from Cloud Data Centers

Vijay Gabale[1], Partha Dutta[2], Ravi Kokku[2], Shivkumar Kalyanaraman[2]
[1]IIT Bombay, Mumbai, India
[2]IBM Research, Bangalore, India
vijay.gabale@gmail.com, {parthdut, ravkokku, shivkumar-k}@in.ibm.com

***Abstract*—The Internet is witnessing a rapid increase in video traffic. Due to the scalability and the cost-savings offered by cloud-computing, Internet video service providers are increasingly delivering their content from multi-tenant cloud data centers. One of the major challenges faced by such a video service provider is the management of the Quality-of-Experience (QoE) of the end-users in the presence of Variable Bit Rate (VBR) video flows, time varying network conditions in the Internet, and the bounded egress bandwidth provided by the data center. To this end, we present InSite, a light-weight and easy-to-deploy solution for managing the QoE of a set of video flows of a service provider, which are served from a datacenter.**

InSite is deployed at the egress of a data center, between the video servers and the clients, and manages the video flows that are transmitted over TCP. The solution uses a novel generalized binary search technique to concurrently search for the appropriate flow rates for a set of flows, with the goal of maximizing the QoE-fairness across the flows, as opposed to TCP-fairness. The search takes into account the total egress bandwidth allocated for the set of video flows at the datacenter, the unknown and possibly time-varying capacities of any remote bottleneck links, and the playout buffer sizes of the video flows. The solution is also designed to operate with minimal modifications to the video servers and the clients. In our evaluations using extensive ns-3 simulations and a testbed implementation for serving videos over TCP, we observe that deploying InSite achieves several folds reduction in playout stalls over a system without InSite.

I. Introduction

Motivation. A majority of traffic in Internet today is video and this share is increasing at a rapid pace [1]. Due to the associated steep rise in the bandwidth, computing and storage requirements, the video service providers are increasingly moving to cloud computing platforms for content delivery [2], [3]. For instance, a popular Video-on-Demand (VoD) service provider in North America has moved its streaming servers to a cloud computing platform [4]. Such a transition helps the service providers to avoid the cost of owning and managing their private video server farms, and dynamically change the amount of rented resources [5], [6]. Also, the cloud platforms can serve the videos out of multiple geographically distributed data centers, and hence, reduce latency to the end-users.

Despite the benefits of cloud-based delivery, the videos delivered from a cloud datacenter are limited by the egress bandwidth at the datacenter [6], [7], [8], and the time-varying network conditions over the Internet. Cloud data centers are frequently shared by multiple service providers (multi-tenant), and the data center operator may bound the peak available bandwidth (see Figure 1) for each service provider to provide performance management and isolation [5], [9]. In addition, video flows are typically Variable Bit Rate (VBR) and delay-sensitive. Not surprisingly, it is challenging to manage the Quality-of-Experience (QoE) of the end-users in the presence of such dynamic video and network conditions. Since the revenue of the video services, whether subscription-based or advertisement-based, depends crucially on the end-users' experience, managing QoE is essential for a widespread deployment of cloud-based video delivery.

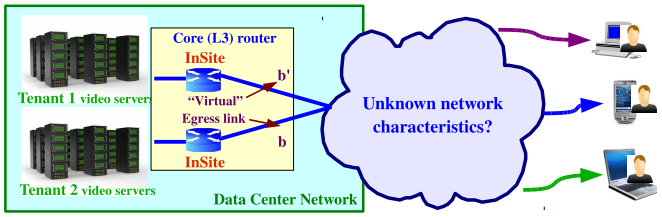


Fig. 1. A Data Center showing two tenants; each tenant reserves a certain amount of egress bandwidth (b or b') as a part of SLAs, and the data center uses this reservation to ensure bandwidth isolation across tenants. One instance of InSite is placed for each tenant to manage the QoE, given the "virtual" egress link and the unknown in-network bottlenecks.

Our Solution. In this paper we present a solution, InSite, for managing QoE of video delivery from cloud data centers. InSite is positioned at the egress of a data center hosting the video service, e.g., at a core router of the data center (see Figure 1). It manages the set of video flows between the video servers and the clients of a given video service provider. InSite focuses on video flows over TCP, as a significant fraction of Internet video is delivered over HTTP/TCP. Also, in terms of deployment, a transport layer solution can work across different video delivery applications.

InSite considers playout stalls as the primary QoE metric for a video playout. A playout stall, also called a buffering event, occurs when the playout buffer at the video client becomes empty. A recent large-scale study [10] has observed that the QoE of a video service is judged (by the end-user) primarily based on the number of playout stalls, along with other metrics such as ratio of buffering time to the total session time, and the video resolution.

To manage the playout stalls, InSite controls the rate of each flow based on the following three simple but crucial observations: (1) at any point in a video playout, the key

factor that determines its vulnerability to stalling is the current playout buffer size at the client, (2) a video flow typically requires variable bit rate and it may encounter unknown and time-varying bottlenecks over the Internet, and hence, the rate allocation must be dynamically adapted over time, and (3) as TCP is oblivious to the video application characteristics (e.g., its variable bit rate requirement), TCP's sharing of the egress bandwidth according to the network characteristics (e.g., Round-Trip-Time (RTT)) might be far from the one that ensures high QoE as well as a fair distribution of QoE.

InSite's rate control is implemented using the well-known technique of advertised receiver window based TCP rate control [11], [12]. At the heart of InSite is a fast algorithm that concurrently searches for the right TCP receiver window size for each flow, over a network where bottleneck links for the flows may have unknown capacities. InSite runs this algorithm iteratively in an *epoch-by-epoch* manner so as to reduce the number of playout stalls, and to fairly distribute the stalls across clients. In particular, in each iteration of InSite, the window sizes are set such that a given function of the playout buffer sizes of all the clients is maximized. InSite also adapts to dynamic bandwidth SLAs between the data center and its tenants (i.e., dynamic changes in the egress bandwidth b), at a fine granularity of epoch-boundaries.

Contributions. This paper makes two contributions. **(1)** InSite provides a fast and simple concurrent binary search mechanism to share bandwidth across video flows of a given service provider at a datacenter, with the objective of *maximizing QoE fairness, as opposed to TCP fairness*. To the best of our knowledge, InSite is the first solution to focus on QoE fairness (measured in terms of playout stalls) of a set of video flows over TCP. **(2)** InSite is transparent to video servers, clients, and applications. In particular, it does not require any modification to the TCP/IP stack or the video application at the servers and the clients, and it does not need any feedback from the remote clients. Consequently, InSite requires minimal disruption or modifications in a cloud data center serving videos, and facilitates "offloading" QoE optimization as a revenue-generating service for data center owners, thereby significantly increasing the chance of its adoption.

Roadmap. The rest of the paper is organized as follows. In Sec. II, we formulate the underlying network optimization problem. Subsequently, Sec. III describes the core algorithm of our solution, InSite. We evaluate the performance of InSite subject to different network and video characteristics using ns-3 simulations in Sec. IV. Sec. V, describes a prototype implementation of InSite in our test-bed. In Sec. VI, we compare InSite with related work on video delivery. We conclude the paper with discussion on future work in Sec. VII.

II. Preliminaries And Problem Formulation

In this section we formulate the main network optimization problem that is handled by InSite. For ease of presentation, in the rest of the paper we consider one given tenant (service provider) in a multi-tenant datacenter (see Fig. 3). We assume that the datacenter enforces network performance isolation among different tenants, and provides each tenant a known (virtual) egress bandwidth, and hence, a separate instance of InSite can be deployed for each tenant. Also, we assume that, apart from the egress link, the rest of the data center network has enough capacity to support all the video flows.

A. Preliminaries

Video flows. Typically compressed videos play at constant number of frames per second, but the frames may vary in size. The *playback curve* of the video specifies the amount of video data that the client needs by time t (from the start of the playout) to have an uninterrupted playout. After requesting the video, the client starts playing out the video after an initial buffering delay. If at any point in the playout, the next video frame to be played out has not yet been received by the client, the playout stalls. Such a stall adversely impacts the QoE of the video playout.

TCP rate control. We assume that the rates of the TCP flows are changed using advertised receiver window modification in the TCP header, which is a well-known TCP rate control mechanism [11], [12]. The rate of a TCP flow is controlled both by the congestion control and the flow control algorithms, which in turn specify the congestion window size and the receiver advertised window size, respectively. The effective size of the TCP window is set to minimum of the congestion and receiver windows, and therefore, instantaneous rate of a TCP flow cannot exceed the product of the receiver window size and the TCP segment size, divided by the Round Trip Time (RTT) of the flow.

Network topology. We aim to manage video flows passing through an egress link (also called a shared link) of a datacenter with a known capacity b allocated for the video flows (see Fig. 3). To account for the dynamic nature of the Internet, each video flow i is assumed to have a remote bottleneck link (that is distinct from the shared link), whose spare capacity b_i is unknown and time-varying.[1]

B. Problem Formulation

Capacity constraints. Suppose there are n video flows $\{1, 2, ..., n\}$ in progress, and the (receiver advertised) window size of the i^{th} flow is x_i. We know that the instantaneous rate of a TCP flow can be expressed as $\frac{\text{window in bytes}}{\text{RTT in seconds}}$. Now, the rate constraint due to egress link can be expressed as $\sum_{i=1}^{n} a_i x_i \leq b$, where a_i is the inverse of round trip time (RTT). This constraint says that the sum of the flow rates over the shared egress link cannot exceed the link's capacity. Similarly, for the remote bottleneck of each flow i, we can express a constraint of the form $a_i x_i \leq b_i$ $(i = \{1, ..., n\})$, which states that the rate of a flow cannot exceed its remote bottleneck spare capacity.

QoE-aware objective. We use the number and duration of playout stalls as the primary metrics for measuring the QoE of a video playout. Following [13], [14] we observe that the playout buffer size of a flow strongly affects the chance of a playout stall, and hence, we base our objective on the buffer sizes. The buffer size of a client is defined as the amount

[1]Our algorithm can be extended for the general case where there are subsets of flows that share a common bottleneck, apart from individual remote bottlenecks, if any.

of data that is received by the client but not yet played out. In particular, the objective for the QoE optimization that we consider in this paper is maximizing the weighted proportional fairness [15] given by $\mathbf{f}(\mathbf{x}) = \sum_{i=1}^{n} \phi_i log(x_i)$, where x_i is TCP (receiver advertised) window size for flow i. The weight ϕ_i is computed as follows. At any time t, let D_i be the difference between the amount of bytes streamed and the amount of bytes that needs to be sent by time t for an uninterrupted playout of a video flow i, as given by the playback curve.[2] If there are n flows in progress, the inverse of normalized buffer size for flow i is given by $\phi_i = \frac{\sum_{i=1}^{n} D_i}{D_i}$. Note that, this choice of weights gives higher weights to the flows that have a smaller buffer size.

Our goal is to find an optimal window vector $\bar{\mathbf{x}}*$ which maximizes the QoE-aware objective function with respect to capacity constraints given in Fig. 2.

Window Size Variables: $x_i, \forall i$.
Capacity Constants: b, $\phi_i, a_i, \forall i$ (**known**). $b_i, \forall i$ (**unknown**).
QoE-aware Objective: max $\mathbf{f}(\mathbf{x}), \mathbf{f}(\mathbf{x}) = \sum_{i=1}^{n} \phi_i log(x_i)$.
Egress (Shared) Link Constraint: $\sum_{i=1}^{n} a_i x_i \leq b$.
Remote Bottleneck Constraints: $a_i x_i \leq b_i$, $\forall i$.

Fig. 2. Network Optimization Problem

InSite solution setting. The optimization problem presented in Fig. 2 cannot be solved directly as a mathematical program because the values of the b_is are unknown. Thus our overall approach to solve this problem is to start with a window vector $\bar{\mathbf{x}}$, and through a series of window vector modifications arrive at an optimal window vector $\bar{\mathbf{x}}*$ where each element of x_i corresponds to the desired TCP receiver window size for a flow. As described in Sec. II-A, InSite-router sits at the egress link of a datacenter (between video servers and video clients), and modifies the receiver advertised field in the TCP acknowledgment headers to set to x_i. This achieves the desired rate for a flow, and maintains the required buffer size at each video client.

We now briefly discuss following three questions that formulate our solution setting, (1) how should we estimate the buffer sizes at the video clients? (2) how frequently should we estimate the buffer sizes? (3) what should be the granularity of modifying the window vector?

In InSite, we divide the video session time in terms of 'epoch(s)' and an epoch is further divided into 'sub-epoch(s)'. To estimate the buffer sizes of video clients at InSite-router, we measure the amount of data streamed to the user in an epoch, and then, compare it with the playback curve of the corresponding video. The difference between the two quantities gives us an estimate of the buffer size at the video client, and thus the weight ϕ_i for flow i in subsequent epoch. The playback curves (of only few KB) of all flows can be quickly downloaded at InSite-router from some server in the datacenter incurring negligible communication and storage overhead.

[2]If $D_i < 1$, we set D_i to a minimum value of 1.

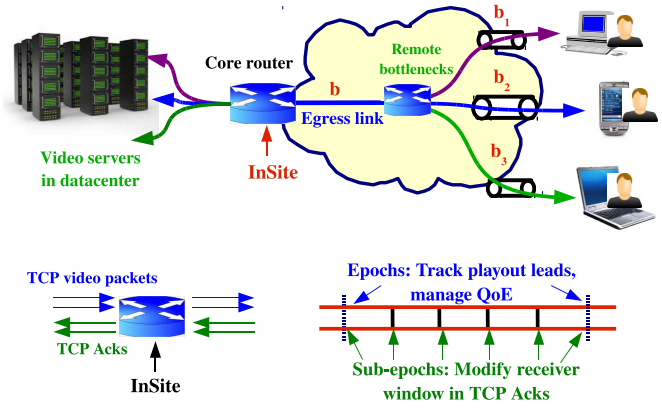


Fig. 3. InSite system architecture for a given tenant

To answer the second question, we note that some past studies (e.g. [10]) have indicated that 10 to 20 seconds is a reasonable amount of time for which a users may wait until the playout starts. We adopt this time granularity for the epoch duration. For the third question, we observe that, it typically takes a few (2 to 4) RTTs for a TCP flow to show the effect of a window modification, and consequently, to converge to a rate (assuming stable network conditions). Therefore, we modify the window vector at the boundaries of sub-epochs (of order of 1 to 2 seconds), whose durations are a small multiple of average RTT of the flows.

Thus, within an epoch, we assume that network conditions remain static (i.e., the values of RTT $(1/a_i)$ and remote (b_i) bottleneck bandwidths remain constant), and then over sub-epochs, we modify window vector $\bar{\mathbf{x}}$ to solve optimization problem in Fig. 2 . This is a reasonable assumption with respect to work in [16] which observes that, for a period of 20 seconds within a connection, RTT remains fairly stable on the Internet. Moreover, in our evaluation, we observe that InSite can handle infrequent link capacity variations even within an epoch of 10 seconds. We give further details in Sec. IV.

In the next section, we present our algorithm which takes $O(n + (2 \times log_2 m))$ (m is maximum window size assumed to be sufficiently large in practice) iterations to discover unknown capacities b_is and find the optimal point $\bar{\mathbf{x}}*$.

III. InSite: Managing QoE of Video Flows

In our algorithm for solving the optimization problem in Fig. 2, we start with an optimal solution for a modified problem which excludes the n unknown constraints (i.e., the remote bottleneck constraints), and then use a variant of multi-dimensional binary search to arrive at an optimal solution to the original problem which considers the n unknown constraints. The search step to discover the unknown values of b_is in our algorithm is based on *membership queries* which checks if a given set of window values results in a flow rate allocation that satisfies *all*, known and unknown bottleneck constraints.

Before explaining our algorithm, we first describe window vector based membership queries and how the system responds to such a query, which forms the basic framework of our solution. This framework to find unknown capacity is partially

inspired by [17], [12]. Then we present few lemmas on the properties of optimal solutions, which we use in the description and analysis of our algorithm. Subsequently, we present our algorithm in detail.

A. System Model

Membership queries. At an abstract level, the shared and remote bottleneck link constraints in the problem formulation are half-spaces in the n dimensional space, $\mathbf{R}^n$, of window vector values. Our goal is to find an optimal point, given by a window vector $\bar{\mathbf{x}}* = \{x_1^*, x_2^*, ..., x_n^*\}$ which (1) lies in the polytope defined by these half-spaces and (2) maximizes the objective. We know one of the half-spaces of the convex polytope (since we know b). However, the remaining half-spaces are unknown (since we do not know b_is). To discover the optimal point $\bar{\mathbf{x}}*$, our algorithm runs in iterations, and in every iteration, we propose a window vector $\bar{\mathbf{x}}$. In response, we receive a feedback $\bar{\mathbf{x}}^{\mathbf{sys}}$ from the system as to whether the point is inside or outside the polytope.

Queries for single flow vs. multiple flows. If we consider a single flow in progress, say flow 1, the problem is to find the value of unknown remote bottleneck, b_1, by setting the value of the window size x_1. The window size x_1 in turn sets an upper limit of a_1x_1 on the allowed TCP rate of flow 1. Each time we set x_1 and observe the data transmitted, it is equivalent to learning whether $a_1x_1 > b_1$. Once we set window size to x_1, we observe the amount of data that is transmitted in the sub-epoch. The window size that corresponds to the actual amount of data transmitted is selected as the system feedback x_1^{sys}. If $a_1x_1 > b_1$, i.e., if we allow higher flow rate than the capacity of its remote bottleneck link, we get a system feedback x_1^{sys} to be less than x_1. Otherwise, we obtain $x_1^{sys} = x_1$. What does this mean in terms of the network flow? When we send more data than what the bottleneck link can handle, the data either experiences packet loss or substantial delay, because of which the number of acknowledgments is lower than expected. Greater the value of a_1x_1 compared to b_1, the lower is the value of x_1^{sys} returned by the system, because of more packet losses or larger delays. On the other hand, if we send less or equal amount of data to what the bottleneck link can handle, we will receive acknowledgments for (almost) all the sent packets. This signifies that x_i can be increased further.

Next, suppose there are two flows 1 and 2 in progress. We can *simultaneously* set the window size of flow 1 to x_1 and flow 2 to x_2 to test whether $a_1x_1 > b_1$ and whether $a_2x_2 > b_2$, only if the corresponding allowed flow rates do not exceed the capacity of the shared link b (i.e., only if $a_1x_1 + a_2x_2 \leq b$). Otherwise, if $a_1x_1 + a_2x_2 > b$ then system feedback might be dominated by the shared link constraint.

Therefore, in our algorithm we never violate the shared bottleneck constraint when we set a window vector to query the window values for all flows. However, a proposed window vector $\bar{x} = \{x_1, x_2, ..., x_n\}$ may still violate the constraints imposed by the remote bottlenecks, in which case, we use binary search algorithm to converge to an optimal window vector.

B. Properties of an Optimal Solution

We now present some properties of an optimal solution that are needed to describe our algorithm. Let N denote the set $\{1, \ldots, n\}$ of all flows. We omit the proof of the first two lemmas due to lack of space.

Lemma 1 below states that if the sum of remote bottleneck spare capacities is at most the shared link capacity, one can set the window vector to transmit data at the rate which consumes spare capacities of all the remote bottlenecks to achieve an optimal solution.

Lemma 1: If $\sum b_i \leq b$, the point $\left(x_i = \frac{b_i}{a_i}\right)$, $\forall i$, is an optimal solution.

The next lemma states that if the sum of the spare capacities of remote bottlenecks is greater than the capacity of the shared link, then the objective can be maximized by a solution such that the solution transmits the data at the rate which makes the shared link full.

Lemma 2: If $\sum b_i \geq b$, then there is an optimal solution that satisfies $\sum_{i=1}^{n} a_i x_i = b$.

The next two lemmas presents some more properties of the optimal solution.

Lemma 3: An optimal solution to the problem with objective $\sum_{i=1}^{n} \phi_i log(x_i)$ and a single constraint $\sum_{\mathrm{i=1}}^{n} a_i x_i \leq b$, is given by $(\frac{\phi_i/a_i}{\sum_i \phi_i}) \times b$, $\forall i$. (Also, note that for the optimal point, $\sum_{\mathrm{i=1}}^{n} a_i x_i = b$ holds.)

Proof: Maximizing $\sum_{\mathrm{i\,=\,1}}^{n} \phi_i log(x_i)$ is equivalent to minimizing $\sum_{\mathrm{i\,=\,1}}^{n} -\phi_i log(x_i)$. This optimization problem is now convex since the constraint is a linear inequality and the objective function is a sum of convex functions, and hence convex. If $f(x_1, x_2, ..., x_n) = \sum_{\mathrm{i\,=\,1}}^{n} -\phi_i log(x_i) + \lambda(\sum_{\mathrm{i=1}}^{n} a_i x_i - b)$, then by KKT conditions [18] for convex optimization problems, we have the following equations: $\frac{\partial f}{\partial x_i} = -\frac{\phi_i}{x_i} + \lambda a_i = 0$, $\forall i$, and $\sum_{\mathrm{i=1}}^{n} a_i x_i = b$. Solving these equations, we have a point $(\frac{(\phi_i/a_i)}{\sum_i \phi_i}) \times b$, $\forall i$, with $\lambda = \frac{\sum_i \phi_i}{b}$. And by KKT conditions, this point is the (global) optimal solution. ■

Lemma 4: An optimal solution for the problem in Figure 2 is given by $\bar{x}$ such that there is a set $S \subseteq N$ (S can possibly be N or $\emptyset$), and the i^{th} element of $\bar{x}$ (the window for flow i) is $\frac{b_i}{a_i}$ if $i \in S$, or is x_i^{share}, otherwise, where x_i^{share} values are given by an optimal solution to the following problem: maximize $\sum_{i \notin S} \phi_i log(x_i)$ with respect to the constraint $\sum_{i \notin S} a_i x_i \leq (b - \sum_{i \in S} b_i)$.

Proof: We will explain the proof in two parts.

Case 1: $\sum_{i=1}^{n} b_i \leq b$**.** In this case by Lemma 1, all x_i variables converge to $\frac{b_i}{a_i}$, and hence $S = N$ gives an optimal solution.

Case 2: $\sum_{i=1}^{n} b_i > b$**.**: The optimal point in this case is either (*sub-case 2.1*) an optimal solution to the problem maximize $\sum_{i \in N} \phi_i \ log(x_i)$ subject to $\sum_{i \in N} a_i x_i \leq b$, [3] or (*sub-case 2.2*) an optimal solution $\bar{x}$ of the problem in Fig. 2 such that there is a set $T^1 = \{i | x_i = \frac{b_i}{a_i}\}$ ($T^1 \subseteq N$), and $\forall i \notin T^1$, $x_i < \frac{b_i}{a_i}$.

If sub-case 2.1 is true then the lemma is true with $S = \emptyset$. Now, suppose that sub-case 2.2 is true. Given $\bar{x}$ and T^1,

[3] That is, a problem similar to our original problem but without the $a_i x_i < b_i$ remote bottleneck constraints.

we can transform our problem to the following problem: maximize $\sum_{i\in T^1}(\phi_i log(b_i/a_i) + \sum_{i\notin T^1}\phi_i log(x_i))$ subject to $(\sum_{i\in T^1} b_i + \sum_{i\notin T^1} a_i x_i \leq b)$, and $a_i x_i \leq b_i, \forall i \notin T^1$. This problem in turn can be transformed to the following optimization problem: maximize $\sum_{i\notin T^1}\phi_i log(x_i)$ subject to $\sum_{i\notin T^1} a_i x_i \leq (b - \sum_{i\notin T^1} b_i)$, and $a_i x_i \leq b_i, \forall i \notin T^1$. Let $b^1 = (b - \sum_{i\notin T^1} b_i)$.

Since, $\sum_{i=1}^{n} b_i > b$, we have $\sum_{i\notin T^1} b_i > b^1$, and our new transformed problem formulation is same as Case 2 above with b replaced by b^1, and N (set of all flows) replaced by $N \setminus T^1$. Now we iteratively continue applying the argument in Case 2, and in iteration j define set T^j (similar to the definition of set T^1 in the first iteration), until in some iteration p, either sub-case 2.1 becomes true or sub-case 2.2 is true such that $\cup_{j=1}^{j=p} T^j = N$. If sub-case 2.1 is true in iteration p then the lemma is true lemma with $S = \cup_{j=1}^{j=p-1} T^j$. Otherwise, if sub-case 2.2 is true and $\cup_{j=1}^{j=p} T^j = N$, then the optimal solution is $\{b_1/a_1, \ldots, b_n/a_n\}$, and the lemma is true with $S = N$. ■

C. InSite Algorithm

We now give a description of the algorithm, which is (re)executed in every epoch. In an epoch, InSite algorithm runs in iterations, and it has two main tasks which constitute one iteration (one iteration corresponds to one sub-epoch) of the algorithm: (1) calculating window vector, (2) applying window vector and tracking system feedback. This forms a membership query, as described in Sec. III-A.

Step 1: First iteration (sub-epoch) of the epoch.

Apply window vector: Initially, all the remote bottleneck capacities are unknown. Therefore, in the first iteration, we solve the problem in Fig. 2 subject to only the shared link constraint to calculate the window vector. For this initial step, the algorithm uses Lemma 3 to find the optimal window vector, $\bar{x}^{LOS}$ which maximizes the objective. Subsequently, as mentioned in III-A, we apply the window vector and track the system feedback on the boundaries of sub-epochs in each iteration.

Tracking system feedback: To compare window vector given by $\bar{x}_i^{LOS}$ with system feedback vector $\bar{x}^{sys}$, we record the number of bytes sent, and calculate the achieved rate AR_i at the end of a sub-epoch as the total number of bytes sent in the sub-epoch divided by the sub-epoch duration. The window size x_i^{sys} given by the system feedback is defined as the window size that if applied to the flow would have ideally achieved the rate AR_i without any packet loss, i.e., $x_i^{sys} = \frac{AR_i}{a_i}$. (Recall that $a_i = 1/RTT_i$.)

Binary search for window vector: If there exists a flow i for which $x_i^{sys} \leq x_i^{LOS}$, we put the flow i in set F. The significance of set F is as follows. Based on the discussion in Sec. III-A, if $x_i^{sys} \leq x_i^{LOS}$ for any flow i, it implies that setting x_i to x_i^{LOS} results in reduced throughput. This reduction in throughput will ideally occur when if $a_i x_i^{LOS} > b_i$, i.e., we have set the window size to a value that cannot be supported by the unknown remote bottleneck link capacity b_i for flow i. Such a flow i is put in set F, and in subsequent iterations, the algorithm searches for b_i using a binary search routine over the window size x_i. It then takes at most $log_2 m$ iterations of main loop for all flows in F to converge to their respective b_i values The remaining flow are put a set H.

On the other hand, for all flows for which $x_i^{sys} = x_i^{LOS}$, the spare capacity b_i of remote bottleneck link is at least $a_i x_i^{LOS}$. (Note that x_i^{sys} cannot exceed x_i^{LOS}, since the receiver window imposes an upper bound on the transmission rate.) If for all flows, $x_i^{sys} = x_i^{LOS}$, then it follows that $a_i x_i^{LOS} \leq b_i$ for all i, and $\sum_{i=1}^{n} b_i \geq \sum_{i=1}^{n} a_i x_i^{LOS} = b$. (The last equality follows from Lemma 3.) Thus, from Lemma 2, $\bar{x}^{LOS}$ gives us an optimal solution.

Step 2: Main loop for subsequent iterations (sub-epochs).

Apply window vector: The window vector calculated for the next iteration of the algorithm is denoted by x^{curr}. For flows in F, x^{curr} is given by the binary search. Thus, we have $b' = b - \sum_{i\in f} a_i x_i$ remaining capacity for the shared link for the flows not in F. Here, we use Lemma 2 and Lemma 3 to compute x^{curr} to distribute the remaining capacity among the flows that are not in F. The flows that are not in F are divided into two sets, G and H, which we describe below.

Tracking system feedback: Once window vector x^{curr} is applied to the system, we get a system feedback $\bar{x}^{sys}$. We retrieve following information from the feedback. (1) We decide the search-window of binary search for next x_i^{curr} depending on the feedback, and (2) if for some flow $j \notin F$, $x_j^{curr} > x_j^{sys}$, we move flow j to set G. The significance of set G is as follows. Before entering set G, for flow j, x_j^{curr} was set to the value given by Lemma 3. However, from the system feedback $x_j^{curr} > x_j^{sys}$ for flow j, we deduce that this window allocation exceeds the unknown value $\frac{b_j}{a_j}$.

Binary search for window vector: Therefore, similar to flows in set F, we also apply binary search for a flow in G in our algorithm, so that the flow can converge to the window size $\frac{b_j}{a_j}$ corresponding to its remote bottleneck capacity. We denote this binary search vector of flows in set G by $\bar{x}^{temp}$. However, we note that if the binary search value for a flow in G exceeds its share x_j^{share} obtained from Lemma 3 using the remaining capacity b' in the shared link, the rates corresponding to the window size allocation may violate the shared link constraint. Hence for flows in set G, we set $x_j^{curr} = min(x_j^{temp}, x_j^{share})$ for the next iteration.

The flows that are not in sets F and G at the end of the iteration are included in set H. Now, because we allocate $min(x_j^{temp}, x_j^{share})$ as the window size for flows in G, and if $x_j^{temp} < x_j^{share}$, there is further remaining capacity in the shared link, which is given by $\sum_{j\in g}(x_j^{share} - x_j^{temp})$. We distribute this spare capacity in flows in H using Lemma 3.

We continue the step 3 of the algorithm until we reach an allocation that is of the form given by Lemma 4, i.e., window size of some flows j have converged to their $\frac{b_j}{a_j}$, and for the other flows, the window sizes are given by an optimal solution of a problem with only the shared link constraints (Lemma 3).

D. Algorithm Correctness and Efficiency

Theorem 1: The above algorithm terminates at an optimal solution for the problem in Figure 2, and takes at most $(n + (2 \times log_2 m))$ iterations to converge.

Proof: If solution in the *step 1*, $\bar{x}^{LOS}$, is accepted by the system, then $x^{sys} = \bar{x}^{LOS}$, $a_i x_i^{LOS} \leq b_i$ for all i, and therefore, $\sum_{i=1}^{n} b_i \geq \sum_{i=1}^{n} a_i x_i^{LOS} = b$. Then by Lemma 2, $\bar{x}^{LOS}$ gives us an optimal solution.

In other case, suppose that the solution vector is $\bar{x}^{curr} = \{x_1, x_2, x_k, x_{k+1}, ..., x_n\}$. Consider the flows in set F. The algorithm applies binary search on flows in set F, and each flow i in F eventually converges to $\frac{b_i}{a_i}$ in at most $log_2 m$ iterations. Let $b' = b - \sum_{i \in f} b_i$. From the remaining flows, a flow either moves to G or remains in H.

If $b' > \sum_{\text{j in \{G, H\}}} b_j$, all flows will eventually move to G since in each iteration, at least one flow will have $x_j^{share} > \frac{b_j}{a_j}$. If there are n flows, this will take n iterations in worst case. As the flows move to G, the algorithm applies binary search on flows in set G, and each flow j in F eventually converges to b_j in at most $log_2 m$ iterations.

If $b' = \sum_{\text{j in \{G, H\}}} b_j$, a flow j either moves to set G or remains in set H. In set G, the flow either converges to b_j or is set to x_j^{share} in $log_2 m$ iterations, in the worst case. However, by lemma 1, $x_j^{share} = \frac{b_j}{a_j}$. Thus we get $\{\frac{b_1}{a_1}, \frac{b_2}{a_2}, ..., \frac{b_n}{a_n}\}$ which maximizes the objective.

If $b' < \sum_{\text{j in \{G, H\}}} b_j$, then the flows in H will have $x_j = x_j^{share} < \frac{b_j}{a_j}$. Now, it takes at most n iterations for a flow to move from set H to set G, and once a flow comes to G, it takes $log_2 m$ iterations, in worst case, to converge converge to $\frac{b_j}{a_j}$ or x_j^{share} (where x_j^{share} values are given by an optimal solution of a problem with only the shared link constraints (Lemma 3)). This results in a window vector of the form required by Lemma 4 and hence, is an optimal solution. ■

Prior Work on Learning Unknown Polytope. In a related work, [19] shows that an unknown upper convex k-gon can be learnt exactly by asking at most $O(k(nlog_2 n + log_2 m))$ membership queries. For our problem, directly applying this algorithm will take $O(n(log_2 n + log_2 m))$. In contrast, by exploiting the simpler structure of the unknown polytope in our problem, and by directly trying to converge on the optimal point (instead of trying to discover the whole polytope), our algorithm terminates in $(n + 2log_2 m)$ iterations. In [20], [21], the authors discuss the expected cost of greedy active learning algorithms, however they only provide upper bounds on the learning cost.

IV. Performance Evaluation

In this section, we evaluate performance of InSite using ns-3 simulator.

Simulation parameters: In our evaluation, we use the network topology shown in Fig. 4 with a set of video servers along with a capacity on an egress link allocated to a tenant in a datacenter. To simulate a number of video flows, we use video traces of 14 unique video streams, from the ASU video repository [22]. The average bit-rates of these videos are shown in Tab. I.

We simulate 1000 flows using each of these traces multiple times. We emulate the video servers and video clients using TCP socket programs. The video server reads the trace file, and sends the video fragment as specified in the trace. A video client timestamps each fragment when it is received, and re-builds the trace at its end. In our simulation set-up, the capacity of the links is set to 30% more of the aggregate expected bandwidth. The 30% setting is based on the observation that a TCP connection achieves about 75% of the available capacity. This results in 620Mbps bandwidth allocation to a tenant. This setup is similar to a datacenter configuration in [6] in terms of number of flows, and egress bandwidth.

Video No.	1	2	3	4	5	6	7
Avg. rate (kbps)	111	136	167	189	404	411	416
Video No.	8	9	10	11	12	13	14
Avg. rate (kbps)	558	570	638	715	769	840	1165

TABLE I
AVERAGE VIDEO RATES OF THE TRACES USED IN THE EVALUATION

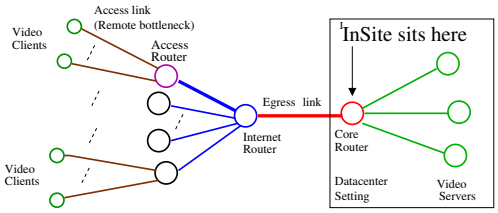


Fig. 4. Simulation network topology

Comparison metrics: By post-processing a downloaded video at the client, we calculate following metrics: (1) number of playout stalls experienced by a video, (2) average buffer size maintained by a video, and (3) the ratio of time for which video is played to the time required to download the video (referred henceforth as the p2d ratio).

Comparing InSite, TCP New Reno, E-WFQ: We compare InSite with TCP New Reno and a variant of Weighted Fair Queuing (WFQ) which we call epoch-by-epoch WFQ (E-WFQ). *Note that, InSite is built on the same TCP New reno.* WFQ is an effective and well-known queuing technique used to impose desired rates on the network flows accordingly to the weights of the flows. In E-WFQ, instead of static weights as in WFQ, we dynamically calculate the weights at the boundary of the epochs. The weight is same as the inverse of normalized buffer size ϕ_i used in InSite (see Sec. II-A). Thus, in E-WFQ, we have a queue for each video flow, and we calculate the flow weight at the boundary of each epoch. However, note that, in InSite, we modify the receiver windows at the sub-epoch boundaries, and thus, InSite can adapt to varying network characteristics within an epoch. Our objective to compare InSite with E-WFQ, is to investigate this adaptiveness of InSite and resulting efficiency.

For our evaluations, we use 1 second as the sub-epoch duration and 10 seconds as the epoch duration. To have fair comparison with E-WFQ, we varied epoch duration for E-WFQ from 1 second to 20 seconds, and chose 5 seconds as the epoch duration for E-WFQ for which it resulted in the best QoE. For video processing, we assume an initial playout buffer of 15 seconds. Although re-buffering strategy is player dependent, we assume that, in case of a playout stall, the client waits until buffer is full to play 5 seconds of video. We also study the effect of client buffer size later in the evaluations.

We run the simulations for 30 minutes. If a video completes streaming within 30 minutes, we restart the video playback, without disrupting the flow. We start flows uniformly randomly within first 60 seconds of simulation.

A. Robustness of InSite to video and network characteristics

Experiment 1: We first examine the performance of InSite when each flow experiences a different RTT. We assign propagation delays to access links from the following set: {10 ms, 20 ms, 40 ms, 80 ms}. This introduces variations in the network characteristics for different video flows. The unfairness of TCP to connections with a large end-to-end delay is well-known, and our objective, in setting different propagation delays, is to investigate the fairness, and the effectiveness of rate control by InSite.

Fig. 5 and 6 compare the playout stalls experienced by each flow for TCP New Reno, E-WFQ, and InSite. To show the results, we group the 14 video flows (in the order shown in Tab. I) in 4 bins, where each bin corresponds to a given propagation delay. Thus 14 video flows in Fig. 5 have propagation delay of 10ms, and 14 video flows shown in Fig. 5 have propagation delay of 80ms. From Fig. 6, we observe that both TCP New Reno and E-WFQ perform poorly (i.e., have a large number of stalls) when network characteristics are unfavourable, i.e., when RTT of the flows are large. However, we see that aggregate as well as per flow playout stalls are drastically low in case of InSite, irrespective of RTTs of the flows. In particular, flows with higher bit-rate requirements suffer more in case of TCP New Reno and E-WFQ, whereas InSite, due to its fair distribution of QoE, results in better performance irrespective of the average video rates. Overall, this result shows that InSite controls the rates of the flows as per their playout requirements, while compensating the effects of uneven RTTs and average bit-rate requirements.

Another key reason for better QoE of InSite is its adaptiveness to busty nature of the videos. In Fig. 8, we plot the instantaneous rate required by video flow number 7, and the instantaneous rates [4] achieved by InSite, TCP New Reno and E-WFQ for the same flow. From the instantaneous rates in Fig 8, we see the bursty nature of the video, and it also shows that InSite reacts fast to the ups and downs in the instantaneous rate of the video. This is because the increase or decrease in the instantaneous rate reflects into corresponding increase or decrease in the buffer size requirement, and using the granularity of adaptive window search, InSite can quickly adapt to the video playback rate. We observe that, E-WFQ reacts slowly to cope with the busty nature of the video.

Algorithm convergence: From the start of the flow, it takes InSite on an average 3 epochs to stabilize on a window value. However, once receiver window becomes dominant, we observe that 1 epoch (which has 10 search steps) is sufficient to converge on the optimal window value.

B. Adaptiveness of InSite

We observed in experiment 1 that InSite adapts fast to impose instantaneous video rates. In this experiment, to evaluate our choice of receiver-window based rate control, we take a closer look at the speed with which InSite converges to required additional bandwidth.

Experiment 2: To test the effectiveness of InSite in this regard, we cap the bandwidth available on access links as shown in Fig. 9 and plot the instantaneous rate achieved by E-WFQ and InSite for two sample flows. As shown in Fig. 9, InSite reacts fast to the change in bandwidth within a few sub-epochs and reduces or increases the rate as per the available network bandwidth. This is because, once the receiver advertised window dominates the congestion window, due to the self-clocking mechanism of TCP, the congestion window is put in linear increase phase. Thus, while the actual window size is dictated by the receiver window, the congestion window rises to a high value. Now, even in case of a packet loss, the value to which the congestion window drops is almost always greater than the receiver window which InSite intends to set. As a result, InSite is able to impose the desired instantaneous rate on the video flows.

In comparison, E-WFQ reacts slowly to the available bandwidth, and this is largely due to the sawtooth behavior, and additive increase phase of the congestion control algorithm.

C. Buffer size requirement

Experiment 3: Until this point, we assumed playout buffer size of 5 seconds. We now vary this period from 1 seconds to 10 seconds, and plot the number of stalls for flow number 7 (one of the most bursty flows) in Fig. 7. In comparison to TCP New Reno, the number of stalls are much lower and fall rapidly in case of InSite, as we increase playout buffer value.

D. Effect of client buffering strategies

Experiment 4: We compare the playout stalls encountered by InSite and E-WFQ for different video flows under different buffering schemes. We use setting mentioned in experiment 1. For this experiment, we consider three buffering strategies: (1) waiting for a fixed amount of time (FT), (2) waiting for a fixed amount of future playout bytes (FB), and (3) waiting for a fixed number of future playout time (PT). Although, we experiment with different values, for results in Fig. 10, we use 5 seconds for FT, 175 Kbytes for FB (considering mean frame size of 1024 bytes, 33 frames per second, and frames worth of 5 seconds), and 5 seconds for PT. As shown in figure, InSite encounters significantly lower playout stalls irrespective of the client buffering strategy. This shows that InSite can work with different video clients in current video eco-system.

V. Prototype

We implement a prototype of InSite as a user level module running on a Linux router. We address several additional issues that arise in a real setting, which we describe below.

Tackling congestion window mismatch. For the initial period of a flow's lifetime when the exponential increase phase of the congestion window is active, the rate achieved by the TCP flow may not be dominated by the receiver window. To avoid such cases, and to increase the receiver window according to the increase in congestion window, we add an *exploration phase* in InSite, where InSite doubles the receiver

[4]The instantaneous rate is calculating by dividing video into 2 second clips, and computing the average rate required to stream a 2 second clip.

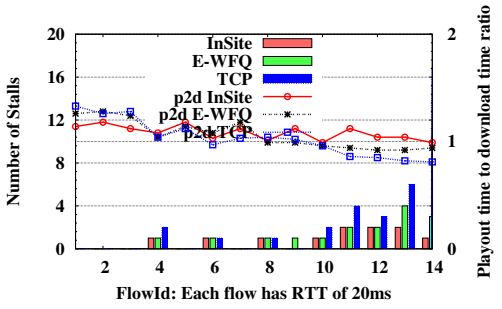


Fig. 5. Experiment 1: Comparison of playout stalls for RTT of 20ms

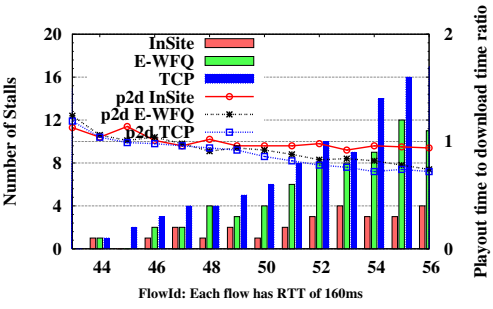


Fig. 6. Experiment 1: Comparison of playout stalls for RTT of 160ms

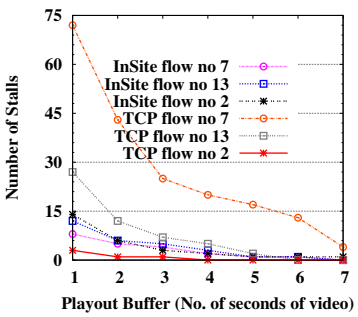


Fig. 7. Experiment 3: Playout buffer requirement

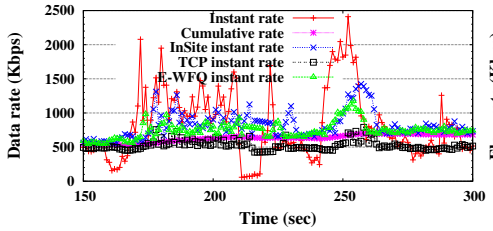


Fig. 8. Experiment 1: InSite reaction to instantaneous video rate and robustness of video.

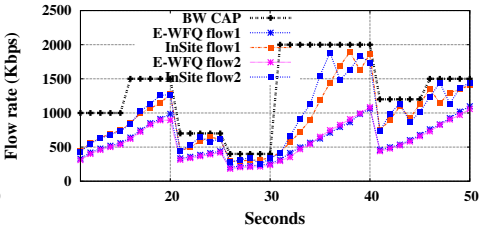


Fig. 9. Experiment 2: InSite reacts fast to the increase or decrease in the available bandwidth.

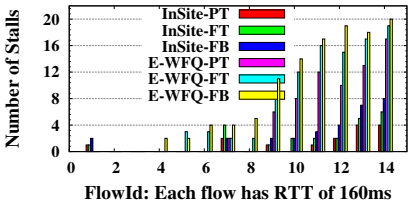


Fig. 10. Experiment 4: InSite has lower stalls irrespective of buffering strategy.

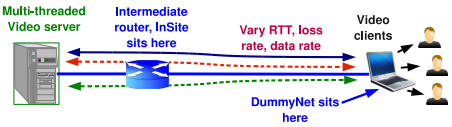
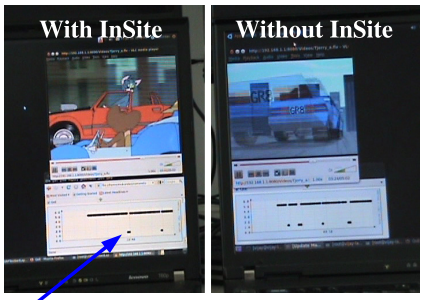


Fig. 11. Evaluation Test-bed



Graphs showing stalls. Low point indicates a stall. Each dot indicates a 1 second stall. Client observes 3 stalls with InSite, and 6 stalls without InSite.

Fig. 12. Screenshot of a demonstration.

window when it observes a steady increase in number of bytes streamed for one or more sub-epochs.

Adaptive search. To make InSite agile when available bandwidth is dynamically changing, we add an adaptive search phase in InSite. When InSite observes a steady flow of bytes for one or more sub-epochs, it increases the receiver window by *a multiplicative factor*. In our experiments, we consider a multiplicative factor of 2. This acts as bandwidth probing when the available bandwidth increases over time. When available bandwidth decreases, the binary search mechanism ensures that InSite does not send more than the remote bottleneck.

Rewriting receiver advertised window. While rewriting the receiver window, we check whether the window-scaling option is enabled by tracking the SYN packet of each TCP connection. If it is enabled, the window is appropriately adjusted. Further, after rewriting the receiver advertised window, we adjust the TCP checksum as mentioned in [11].

Testbed and Analysis. We instantiate a testbed with three Linux machines, each emulating the server, the client and the router functionalities. The three machines have dual-core 2GHz processors running Linux 2.6.26 kernel, connected in a LAN (see Fig. 11), with WAN conditions emulated using DummyNet [23]. We impose static routes on client and server machines so that the packets flow through the router machine in both directions.

Figure 12 shows a screen shot of a demo video displaying the effect of InSite compared to a system without InSite. The graphs in the figure show the stalls perceived in real video streaming. Without InSite, the number of stalls is higher.

Overhead-wise, InSite is light-weight in that it just requires a reveiver window rewrite, and a checksum computation for each packet. We believe that InSite is easy to instantiate on multi-threaded wirespeed processors such as IBM's PowerEN [24], and hence will scale to multi-Gbps speeds easily.

VI. Related work

We classify related work into the following categories.

Video streaming over TCP: Dong et. al. [25] propose to control the streaming rate close to the video playback rate, and provides differential rates for different qualities of videos. However, they do not consider the problem of unknown remote bottleneck bandwidths and varying network conditions. In comparison, InSite sends the video at the rate of the bottleneck link, while taking into account the playback rate of the video. The work in [26] concludes that TCP provides good streaming performance when the achievable TCP throughput is roughly twice the media bit-rate. With InSite, we show that it is

possible to operate TCP at a rate marginally higher than the media bit-rate, and still achieve good streaming performance. The study in [27] analytically determines the proper receiver buffer size to ensure a desired video quality for TCP streaming. In comparison, our objective is to control TCP rate and maintain sufficient receiver buffer so as to make TCP streaming tolerant to time-varying network and video characteristics. Also, unlike our work, [27] does not capture the interactions between different video flows. In another related work [28], the authors develop a periodic broadcast protocol, that can be optimized for a given population of clients with heterogeneous reception bandwidths and quality-of-service requirements. Our work differs by considering unicast connections, which is mostly the case with current VoD systems.

Managing QoE: [14] proposes an on-line algorithm which computes the percentage share of the bottleneck wireless bandwidth based on the knowledge of the current buffered data and the estimation of the future network conditions. However, the algorithm is applicable to only a single flow and the corresponding bottleneck link. [13] presents an epoch-by-epoch framework to allocate wireless transmission slots to streaming videos in a fair manner. Although [13] investigates the fair allocation of bandwidth, it is not applicable to wide-area TCP flows and unknown remote bottlenecks.

VII. Discussion & Conclusion

In this work, we investigate the problem of managing the QoE of Video-on-Demand (VoD) flows over TCP. We believe that several interesting extensions are possible as future work. For instance, primarily to maximize QoE, InSite currently leads to short-term unfairness when some users access higher bandwidth videos relative to others; InSite needs extensions to incorporate a notion of long-term fairness. Since InSite's decisions are based on playout curves, InSite can get out of sync with actual client-side playout when stalls happen, or when a user performs rewind or fast-forward operation. For stalls and rewind operations, InSite's choices are conservative, and hence do not hurt QoE. For fast-forward operations that generate new content requests (e.g. HTTP Range requests) from clients, InSite can use DPI and correct the mismatch.

Going forward, we plan to investigate InSite's applicability to adaptive streaming [29]; the primary idea of managing QoE across TCP flows, by estimating the user's buffer size, can be easily extended to adaptive streaming. Secondly, a recent study [30] shows that 40% of data transferred by Youtube is unnecessary, since many users terminate videos early. This is critical in cloud data centers, especially with pay-per-use policies. Thus, it is important to study how to strike a balance between QoE maintenance and bandwidth wastage. We believe that InSite can be extended to achieve an appropriate balance between these two goals.

References

[1] "Visual networking index: Global mobile data traffic forecast update, 2009-2014," http://www.cisco.com/en/US/solutions/collateral/ns341/ns537/ns705/ns827/white-paper-c11-520862.html.

[2] "Amazon CloudFront," http://aws.amazon.com/cloudfront/.

[3] "Akamai HD Network," http://www.akamai.com/html/solutions/hdnetwork.html.

[4] "Four Reasons We Choose Amazons Cloud as Our Computing Platform," The Netflix Tech Blog, December 14, 2010.

[5] D. Niu, H. Xu, B. Li, and S. Zhao, "Quality-assured cloud bandwidth auto-scaling for video-on-demand applications," in *IEEE Infocom*, 2012.

[6] D. Niu, C. Feng, and B. Li, "A theory of cloud bandwidth pricing for video-on-demand providers," in *IEEE Infocom*, 2012.

[7] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *ACM CoNEXT*, 2010.

[8] H. Ballani, P. Costa, T. Karagiannis, and A. I. T. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM*, 2011.

[9] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: supporting bandwidth guarantees for multi-tenant datacenter networks," in *Proceedings of the 3rd conference on I/O virtualization*, ser. WIOV'11. USENIX, 2011.

[10] F. Dobrian, A. Awan, I. Stoica, V. Sekar, A. Ganjam, D. Joseph, J. Zhan, and H. Zhang, "Understanding the impact of video quality on user engagement," in *Sigcomm*, 2011.

[11] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer, "Tcp rate control," *In SIGCOMM Comput. Commun. Rev.*, vol. 30, January 2000.

[12] P. Key, L. Massoulié, and B. Wang, "Emulating low-priority transport at the application layer: a background transfer service," *In SIGMETRICS Perform. Eval. Rev.*, vol. 32, June 2004.

[13] P. Dutta, A. Seetharam, V. Arya, M. Chetlur, S. Kalyanaraman, and J. Kurose, "On managing quality of experience of multiple video streams in wireless networks," in *IEEE Infocom*, 2012.

[14] G. Ji and B. Liang, "Stochastic rate control for scalable vbr video streaming over wireless networks," in *GLOBECOM*, 2009.

[15] L. Massoulié and J. Roberts, "Bandwidth sharing: Objectives and algorithms," *Trasactions On Networking*, vol. 10, June 1998.

[16] H. Jiang and C. Dovrolis, "Passive Estimation of TCP Round-Trip Times," *SIGCOMM Comput. Commun. Rev.*, vol. 32, 2002.

[17] R. Karp, E. Koutsoupias, C. Papadimitriou, and S. Shenker, "Optimization problems in congestion control," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000.

[18] Stephen P. Boyd and Lieven Vandenberghe, *Convex Optimization*. Cambridge University Press 2004, ISBN/ASIN: 0521833787.

[19] S. Kwek, "An efficient algorithm for learning upper convex polyhedra using membership queries," in *Artificial Intelligence and Mathematics*, 2000.

[20] P. M. Vaidya, "A new algorithm for minimizing convex functions over convex sets," *Math. Program.*, vol. 73, June 1996.

[21] A. Guillory and J. Bilmes, "Average-case active learning with costs," in *Proceedings of the 20th international conference on Algorithmic learning theory*, ser. ALT, 2009.

[22] "Asu video traces," http://trace.eas.asu.edu/mpeg4/index.html.

[23] "Dummynet bandwidth manager," http://info.iet.unipi.it/ luigi/dummynet/.

[24] C. L. Johnson, D. H. Allen, J. D. Brown, S. Vanderwiel, R. Hoover, H. D. Achilles, C.-Y. Cher, G. A. May, H. Franke, J. Xenedis, and C. Basso, "A wire-speed powerTM processor: 2.3GHz 45nm SOI with 16 cores and 64 threads," in *ISSCC*, 2010.

[25] Y. Dong, R. Rakshe, and Z. li Zhang, "A practical technique to support controlled quality assurance in video streaming across the internet," in *Proc. Packet Video Workshop*, 2002.

[26] Bing Wang and Jim Kurose and Prashant Shenoy and Don Towsley, "Multimedia Streaming via TCP: An Analytic Performance Study," in *ACM MULTIMEDIA*, 2004.

[27] T. Kim and M. Ammar, "Receiver Buffer Requirements For Video Streaming Over TCP," in *SPIE 6077, 607718*, 2006.

[28] Phillipa Gill and Liqi Shi and Anirban Mahanti and Zongpeng Li and Derek L. Eager, "Scalable On-demand Media Streaming for Heterogeneous Clients," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 5, October 2008.

[29] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *Proceedings of the second annual ACM conference on Multimedia systems*, ser. MMSys '11, 2011.

[30] A. Finamore, M. Mellia, M. Munafo, R. Torres, and S. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," in *IMC*, 2011.