

# Trickle: Rate Limiting YouTube Video Streaming

Monia Ghobadi\*  
University of Toronto  
monia@cs.toronto.edu

Yuchung Cheng     Ankur Jain     Matt Mathis  
Google  
{ycheng, jankur, mattmathis}@google.com

## Abstract

YouTube traffic is bursty. These bursts trigger packet losses and stress router queues, causing TCP’s congestion-control algorithm to kick in. In this paper, we introduce Trickle, a server-side mechanism that uses TCP to *rate limit* YouTube video streaming. Trickle paces the video stream by placing an upper bound on TCP’s congestion window as a function of the streaming rate and the round-trip time. We evaluated Trickle on YouTube production data centers in Europe and India and analyzed its impact on losses, bandwidth, RTT, and video buffer under-run events. The results show that Trickle reduces the average TCP loss rate by up to 43% and the average RTT by up to 28% while maintaining the streaming rate requested by the application. Further, our results show that Trickle has little impact on video buffer under-run events experienced by the users. We investigate the effectiveness of Trickle based on user bandwidth and demonstrate that Trickle has more benefits for high bandwidth users than low bandwidth users.

## 1 Introduction

YouTube is one of the most popular online video services. In fall 2011, YouTube was reported to account for 10% of Internet traffic in North America [1]. This vast traffic is delivered over TCP using HTTP progressive download. The video is delivered just-in-time to the video player, so when the user cancels a video, only a limited quantity of data is discarded, conserving network and server resources. Since TCP is designed to deliver data as quickly as possible, the YouTube server, *ustreamer*, limits the data rate by pacing the data into the connection. It does so by writing 64kB data blocks into the TCP socket at fixed intervals. Unfortunately, this

technique, termed **application pacing**, causes bursts of back-to-back data packets in the network that have several undesirable side effects. These bursts are responsible for over 40% of the observed packet losses in YouTube videos on at least one residential DSL provider [2].

This problem is not specific to YouTube videos. Similar rate limiting techniques are implemented in other popular video websites [6], and all are expected to experience similar side effects. For example, Netflix sends bursts as large as 1 to 2MB.

As an alternative to application pacing, we present Trickle to rate limit TCP on the server side. The key idea in Trickle is to place a dynamic upper bound on the congestion window (*cwnd*) such that TCP itself limits both the overall data rate and maximum packet burst size using ACK clocking. The server application periodically computes the *cwnd* bound from the network Round-Trip Time (RTT) and the target streaming rate, and uses a socket option to apply it to the TCP socket. Once it is set, the server application can write into the socket without a pacing timer and TCP will take care of the rest. Trickle requires minimal changes to both server applications and the TCP stack. In fact, Linux already supports setting the maximum congestion window in TCP.

The main contribution of this paper is a simple and generic technique to reduce queueing and packet loss by smoothly rate-limiting TCP transfers. It requires only a server-side change for easy deployment. It is not a special mechanism tailored only for YouTube. As TCP has emerged to be the default vehicle for most Internet applications, many of them require certain kinds of throttling. The common practice, application pacing, may cause burst losses and queue spikes. Through weeks-long experiments on production YouTube data centers, we found that Trickle reduces the packet losses by up to 43% and RTTs by up to 28% compared to the application pacing. The rest of the paper covers the design, our experiments, and discussions of other techniques and protocols.

---

\*Ghobadi performed this work on an internship at Google mentored by Cheng.

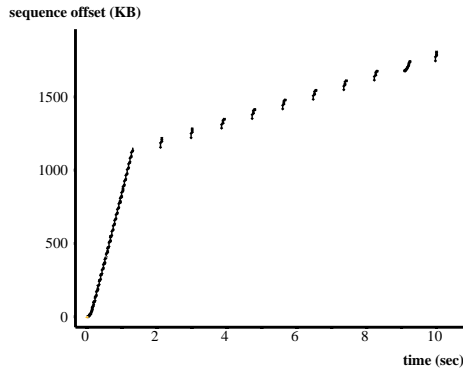


Figure 1: Time vs. sequence of bytes graph for a sample YouTube video with RTT 20ms.

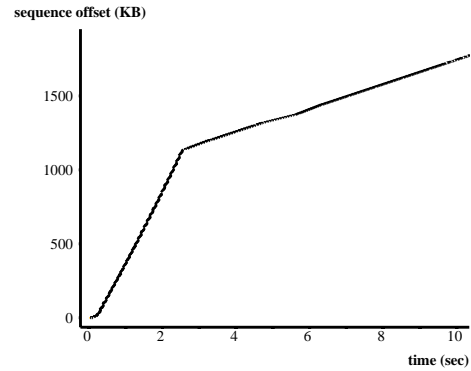


Figure 2: Time vs. sequence of bytes graph for a YouTube video with RTT 30ms using Trickle.

## 2 YouTube Video Streaming

The YouTube serving infrastructure is complicated, with many interacting components, including load balancing, hierarchical storage, multiple client types and many format conversions. Most of these details are not important to the experiment at hand, but some need to be described in more detail.

All YouTube content delivery uses the same server application, called **ustreamer**, independent of client type, video format or geographic location. Ustreamer supports progressive HTTP streaming and range requests. Most of the time, a video is delivered over a single TCP connection. However, certain events, such as skipping forward or resizing the screen can cause the client to close one connection and open a new one.

The just-in-time video delivery algorithm in YouTube uses two phases: a *startup* phase and a *throttling* phase. The startup phase builds up the playback buffer in the client, to minimize the likelihood of player pauses due to the rebuffering (buffer under-run) events. Ustreamer sends the first 30 to 40 seconds of video (codec time, not network time) as fast as possible into the TCP socket, like a typical bulk TCP transfer.

In the throttling phase, ustreamer uses a token bucket algorithm to compute a schedule for delivering the video. Tokens are added to the bucket at 125% of the video encoding rate. Tokens are removed as the video is delivered. The delay timer for each data block (nominally 64kB) is computed to expire as soon as the bucket has sufficient tokens. If the video delivery is running behind for some reason, the calculated delay will be zero and the data will be written to the socket as fast as TCP can deliver it. The extra 25% added to the data rate reduces the number of rebuffering events when there are unexpected fluctuations in network capacity, without incurring too much additional discarded video.

Figure 1 illustrates the time-sequence graph of a

packet trace from a sample YouTube video stream. The x-axis is time and the y-axis is the bytes of video. Vertical arrows represent transmitted data segments that carry a range of bytes at a particular time. After 1.4 seconds of the flow being served in startup phase (which in this case corresponds to the first 30 seconds of the video playback), the YouTube server starts to throttle the sending of bytes to the network. During the throttling phase, every network write is at most one block size plus headers.

In some environments the data rate is limited by something other than the ustreamer-paced writes. For example, some video players implement their own throttling algorithms [6], especially on memory and network-constrained mobile devices. These devices generally stop reading from the TCP socket when the playback buffer is full. This is signalled back to the sender through TCP flow control using the TCP receiver window field. As a consequence, ustreamer is prevented from writing more data into the socket until the video player reads more data from the socket. In this mode, the sender behavior is largely driven by the socket read pattern of the video player: sending bursts is determined by the player read size.

For short videos (less than 40 seconds) and videos traversing slow or congested links, ustreamer may never pause between socket writes, and TCP remains in bulk transmit mode for the entire duration of the video.

## 3 Trickle

### 3.1 The Problem: Bursty Losses

The just-in-time delivery described above smoothes the data across the duration of each video, but it has an unfortunate interaction with TCP that causes it to send each 64kB socket write as 45 back-to-back packets.

The problem is that bursts of data separated by idle periods disrupt TCP's self clocking. For most applications

TCP data transmissions are triggered by the ACKs returning from the receiver, which provide the timing for the entire system. With YouTube, TCP typically has no data to send when the ACKs arrive, and then when ustreamer writes the data to the socket it is sent immediately, because TCP has unused *cwnd*.<sup>1</sup>

These bursts can cause significant losses, e.g., 40% of the measured YouTube losses in a residential ISP [2]. Similar issues have also been reported by YouTube network operations and other third parties. Worse yet, these bursts also disrupt latency-sensitive applications by incurring periodic queue spikes [11, 18]. The queueing time of a 64kB burst over an 1Mbps link is 512ms.

Our goal is to implement just-in-time video delivery using a mechanism that does not introduce large bursts and preserves TCP’s self clocking.

### 3.2 Basic Ideas

A quick solution to the burst problem is to use smaller blocks, e.g., 16kB instead of 64kB. However, this would quadruple the overhead associated with write system calls and timers on the IO-intensive YouTube servers. A better solution is to implement a rate limit in TCP itself. One approach could leverage TCP flow control by fixing the receiver’s window (*rwin*) equal to the target streaming rate multiplied by RTT. Once the receiver fixes *rwin*, the ustreamer can write the video data into the socket as fast as possible. The TCP throughput will be limited by the receive window to achieve the target streaming rate.

However, this receiver-based approach is not practical because YouTube does control user browsers. Our solution, in contrast, sets an upper-bound on *cwnd* of  $target\_rate \times RTT$ , where the *target\_rate* is the target streaming rate of a video in the throttling phase. Fortunately, Linux already provides this feature as a per-route option called *cwnd\_clamp*. We wrote a small kernel patch to make it available as a per-socket option.

To illustrate the smoothing effect of Trickle we show the time-sequence plot of a real YouTube connection in Figure 2. The initial full rate startup phase lasts for 2.5 seconds. In the throttled phase the transmission is rate-limited to 600kbps, by sending *cwnd\_clamp* amount of data per RTT. The result is in a smoother line compared to the staggered steps (bursts) seen in Figure 1 with application pacing.<sup>2</sup>

<sup>1</sup>In some cases using congestion window validation [13] would force TCP to do new slow starts after idling over several retransmission timeouts (RTO). This would not always be useful in YouTube as the application writes are more frequent.

<sup>2</sup>An animated demo of Trickle is available at <http://www.cs.toronto.edu/~monia/tcptrickle.html>

### 3.3 Challenges

The above idea encounters two practical challenges:

(1) **Network congestion causing rebuffering.** Following a congestion episode, ustreamer should deliver data faster than the target rate to restore the playback buffer. Otherwise, the accumulated effects of multiple congestion episodes will eventually cause rebuffering events where the codec runs out of data. The current application pacing avoids rebuffering after congestion events implicitly: when TCP slows down enough to stall writes to the TCP socket, ustreamer continues to accumulate tokens. Once the network recovers, ustreamer writes data continuously until the tokens are drained, at which point the average rate for the entire throttled phase matches the target streaming rate. On the other hand, clamping the *cwnd* will not allow such catch-up behavior.

(2) **Small *cwnd* causing inefficient transfers.** For instance, sending at 500kbps on a 20ms RTT connection requires an average window size of 1250 bytes, which is smaller than the typical segment size. With such a tiny window all losses must be recovered by timeouts, since TCP fast recovery requires a window of at least four packets [5]. Furthermore, using a tiny window increases the ustreamer overhead because it defeats TCP segmentation offload (TSO) and raises the interrupt processing load on the servers.

---

#### Algorithm 1: Trickle algorithm in throttling phase

---

```

R = target_rate(video_id)
while (new data available from the cache)
    rtt = getsockopt(TCP_INFO)
    clamp = rtt * R / MSS
    clamp = 1.2 * clamp
    goodput = delivered / elapsed
    if goodput < R:
        clamp = inf
    if clamp < 10:
        clamp = 10
        write_throttle = true
    setsockopt(MAX_CWND, clamp)
    if write_throttle:
        throttles writes at rate R
    else:
        write all available data

```

---

### 3.4 Design

Trickle starts from the basic design of *cwnd* limitation and addresses both challenges. Algorithm 1 presents it in

exp.	DC1					DC2				
	#flows (M)	size (MB)	avg comp. time (sec)	avg $RTT_{startup}$ (ms)	med. BW (Mbps)	#flows (M)	size (MB)	avg comp. time (sec)	avg $RTT_{startup}$ (ms)	med. BW (Mbps)
baseline1	5.5	9.2	83	99	6	0.3	11.0	234	556	0.7
baseline2	5.4	9.2	84	98	6	0.3	11.0	234	554	0.7
Trickle	5.4	9.2	83	98	6	0.3	11.0	234	557	0.7
shrunk-block	5.5	9.1	82	93	6	0.3	11.0	233	554	0.7

Table 1: Network statistics in DC1 and DC2 binned by each experiment group.  $RTT_{startup}$  is the smoothed RTT (srtt) at the end of the startup phase.

pseudocode.<sup>3</sup> After the startup phase, the ustreamer determines the streaming rate,  $R$ , based on the video encoding rate. When the data from the cache system arrives, the ustreamer gets the RTT and the Maximum Segment Size (MSS) of the connection (using a socket option) to compute the upper bound of the `clamp`. But before applying the `clamp` on the connection, ustreamer takes two precautions to address the challenges described previously.

First, to deal with transient network congestion, ustreamer adds some headroom to the `clamp`. In the experiments we used 20% headroom but we also get similar results with 5%. If the link is experiencing persistent congestion and/or does not have enough available bandwidth, the ustreamer removes the `clamp` by setting it to infinity and let TCP stream as fast as possible. When the goodput has reached  $R$ , the ustreamer will start clamping again.

Second, the ustreamer never reduces `clamp` below 10 MSS to address the second constraint. Studies have shown that Internet paths can tolerate burst of this size [7, 10]. However, doing this also increases the streaming rate beyond  $R$  to  $10 \times \text{MSS}/\text{RTT}$ . Ustreamer thus throttles writes to rate  $R$  using application pacing. Unlike the original ustreamer, however, our modified ustreamer never causes bursts of more than 10 packets.

Finally, the ustreamer clamps the `cwnd` via a socket option. If the write throttling is enabled, it throttles the write at rate  $R$ . Otherwise it writes all data into the socket.

## 4 Experiments

We performed live experiments to evaluate Trickle on production YouTube data centers. We begin this section with the methodology to compare Trickle and existing application pacing, followed by details of the data centers. Then, we present the measurement results that validate the A/B test setup and Trickle implementation.

<sup>3</sup>An animated demo of Trickle is available at <http://www.cs.toronto.edu/~monia/tcptrickle.html>

### 4.1 Methodology

We setup A/B test experiments on selected servers in production YouTube data centers. The first goal is to evaluate if Trickle reduces burst drops and queueing delays. The second goal is to ensure the streaming quality is as good or better than current systems. This is done by measuring the average streaming rate and the rebuffering events. In addition to comparing with current systems, we also compare with the simplest solution, namely reducing the block size from 64kB to 16kB. We ran 4-way experiments by splitting the servers into four groups:

1. Baseline1: application pacing with 64kB blocks,
2. Baseline2: application pacing with 64kB blocks,
3. Trickle,
4. shrunk-block: application pacing with 16kB blocks.

In order to make an apples-to-apples comparison, new TCP connections (video requests) are randomly assigned to the servers in different experiment groups. Thus each experiment group received similar distributions of users and video requests. We use two baseline groups to estimate the confidence level of the particular metric evaluated in our analyses.

All servers use the standard Linux 2.6 kernel with CUBIC [12] congestion control. TCP configuration details can be found in Dukkupati *et al.* [9]. For every connection, we recorded statistics including video ID, IP and ports, bytes sent and retransmitted, RTTs, and rebuffering events in both phases. We further filtered the connections that never enter throttling phase (short video playbacks less than 30 seconds).

### 4.2 Validation

We ran experiments for 15 days during the fall of 2011 in two data centers representing relative extremes of the user network conditions: DC1 in Western Europe and

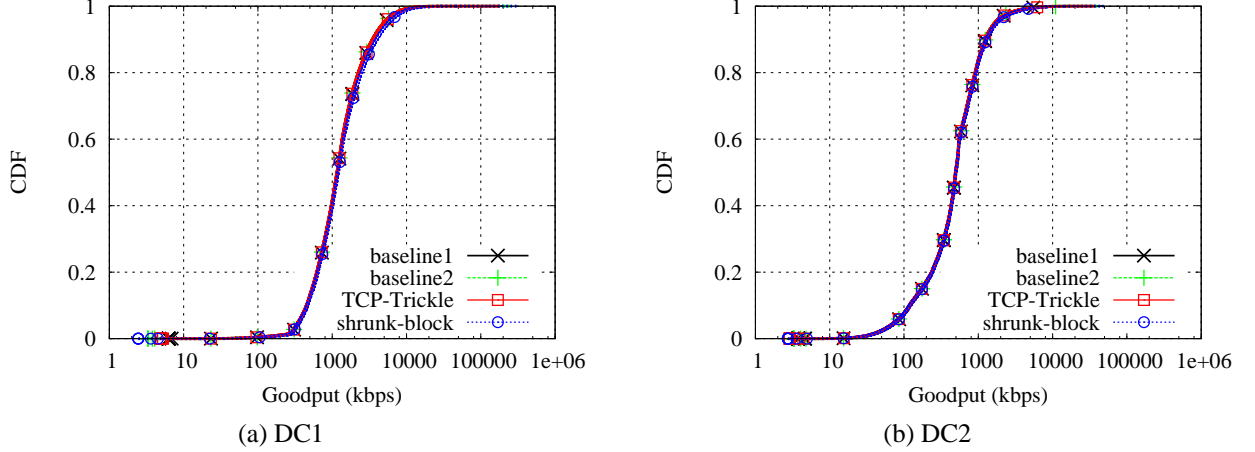


Figure 3: CDF of goodput in throttling phase in DC1 and DC2. The x-axis is in log scale.

DC2 in India. We first compare the statistics in the control variables in the experiments to validate the A/B test setup. Table 1 summarizes these statistics in both data centers. Each experiment group has roughly the same number of flows within each data center. DC1 has 16 times more flows than DC2 because DC1 has more servers participating in the experiment. The average flow length and flow completion in each group are also similar across different groups in the same data center.

We also measure the RTT and goodput, denoted as  $RTT_{startup}$  and BW respectively, at the end of the startup phase of each connection. BW is the average goodput computed across the startup phase (typically for more than 1mB of video data). In the rest of the paper, this easily computed metric is used to gauge the network capacity relative to the target data rate. Since every experiment group uses the original mechanism in the startup phase,  $RTT_{startup}$  and BW should be similar across different group in the same data center; these results indicate that each experiment group receives similar network and application load, as well as the vast difference of user network speed in the two regions.

One of the design goals of Trickle is to maintain the same video streaming rate as the existing ustreamer application. To verify this, for each video in our experiments, we calculate the goodput during the throttling phase and ensure the distributions match between all experiment groups. Figure 3 demonstrates that the CDF of goodput in all experiments groups match; i.e. the video streaming rate is consistent whether or not Trickle is enabled.

## 5 Analysis

In the following sections, we present our experiments' results. The results are statistics collected during the throttling period only. As explained in the Section 4.2, all the four experiment groups use the same mechanism in the startup phase and their results are identical.

### 5.1 Packet Losses

The most important metric is packet loss because Trickle is designed to reduce burst drops. Since losses can not be accurately estimated in live server experiments [4], we use retransmissions to approximate losses. Figure 4 plots the CDF of flow retransmission rate in the throttling phase for DC1. As shown, the Trickle curve is consistently above all three lines, indicating that it successfully lowers the retransmission rate consistently compared to the other three groups. In Trickle, 90% of connections experience retransmission rate lower than 0.5%, while 85% have this behavior using shrunk-block and 80% in baselines. On average, Trickle reduces the average retransmission rate by 43% and 28% compared to the baselines and shrunk-block experiments groups, respectively. Overall, Trickle effectively reduces the drop rate compared to application pacing using 64kB or 16kB block sizes.

Unlike the results in DC1, however, we measured that all four groups in DC2 have similar retransmission rate distributions. This is because most DC2 users have insufficient bandwidth to stream at the target rate. As described in Section 3.4, Trickle will detect the delivery is falling behind the target rate and stop clamping the  $cwnd$ . Therefore connections are not rate-limited by the

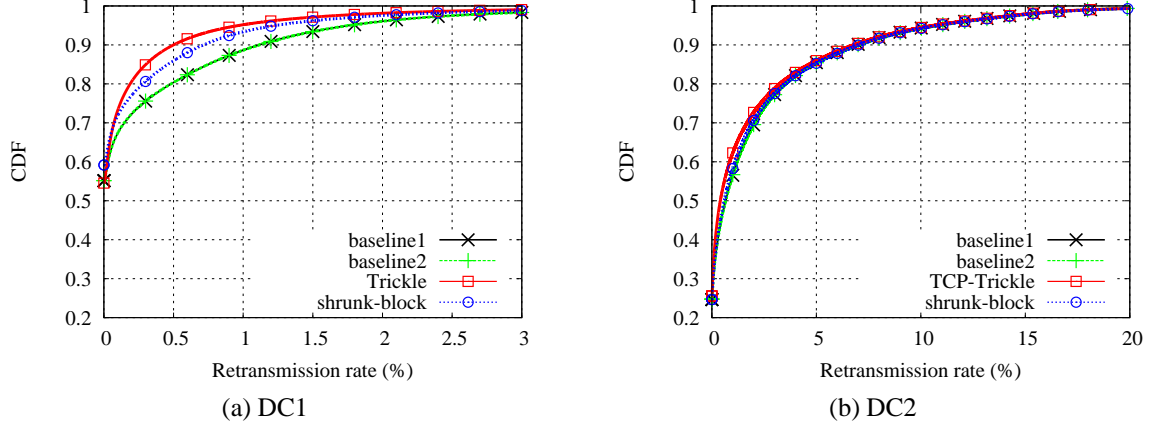


Figure 4: CDF of retransmission rate in the throttling phase.

BW (Mbps)	DC1		DC2	
	% flows	avg. retrans. imprv.	% flows	avg. retrans. imprv.
< 0.5	1%	5%	39%	3%
0.5 – 1	3%	23%	26%	8%
1 – 2	10%	40%	17%	32%
2 – 4	28%	53%	9%	47%
4 – 8	35%	56%	6%	47%
≥ 8	23%	53%	3%	44%

Table 2: The retransmission rate improvement bucketed by user bandwidth.

ustreamer throttling but by the network bandwidth and behave like bulk download in all experiment groups.

To demonstrate the effect of bandwidth, we show the average reduction of the retransmission rate between Trickle and baseline1 bucketed by flows’s BW in Table 2. Given that the average target rates are 677kbps and 604kbps in DC1 and DC2 respectively, the table shows that users with low bandwidth do not benefit from Trickle. On the other hand, about half of packet losses can be avoided in for high-bandwidth users in YouTube using Trickle.

## 5.2 Burst Size

The previous results show that Trickle effectively reduces the loss rate. In this section, we demonstrate that the reduction is achieved by Trickle sending much smaller bursts. We randomly sampled 1% of flows and collected tcpdump packet traces at the server to investigate the burstiness behavior. Following the convention of prior studies [7, 14], we use packets instead of bytes

to measure the burst size. We use the same definition of micro-burst as Blanton *et al.* [7]; a burst is a sequence of four or more data packets with inter-arrival time less or equal to 1 millisecond. We use four or more packets because TCP congestion control, e.g., slow start, naturally sends bursts up to three packets.

Figure 5 plots the burst sizes in DC1. Intuitively, most bursts in the two baseline groups should be about 43 packets (64kB) but in reality only 20% are. This mismatch is due to packet losses and TCP congestion control. After a packet is lost, TCP congestion control reduces *cwnd* and gradually increases it while streaming new data. These *cwnd* changes fragment the intermittent 64kB application writes. The TCP time-sequence graph of a sample flow in the baseline1 group in

Figure 6 illustrates this point. The black arrows are data bursts, the red dots are retransmits, and the purple dots are ACKs with SACK blocks. Around time 77 seconds, a large tail drop occurs causing TCP to reduce *cwnd* and enter slow start. At the 80 seconds mark, TCP *cwnd* has increased to about 32kB. When ustreamer writes 64KB into the socket, TCP sends a 32kB burst followed by a series of small bursts triggered by returning ACKs. The queue overflows once the *cwnd* reaches 64kB at time 85 second. In this example, one loss event has caused later 15 application writes to be fragmented into many smaller bursts.

Back to Figure 5, The shrunk-block curve exhibits interesting steps at 12, 23, and 34 packets corresponding to 16, 32, and 48 kB block sizes, respectively. These steps suggest that either the application and/or the kernel (TCP) is bunching up the writes. We then discovered that the ustreamer token bucket implementation does not pause the write for intervals less than 100ms to save timers. For a large portion of the flows, ustreamer continues to write 16kB blocks due to this special handling.

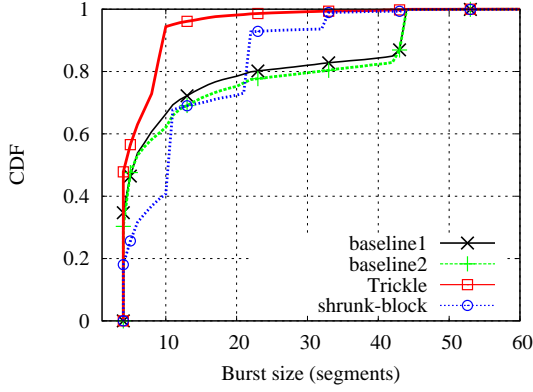


Figure 5: CDF of burst size in DC1.

Lastly, in Trickle 94% of bursts are within 10 packets, because DC1 users have short RTT such that most videos require less than a 10 packet window to serve. As described in Section 3.4, Trickle lower-bounds the clamp to 10 packets to avoid slow loss recovery. The remaining 6% of bursts over 10 packets are contributed by either high RTT, or high resolution videos, or other factors that cause Trickle to not clamp the *cwnd*. In summary, over 80% of bursts in Trickle are smaller than the other mechanisms.

Next we investigate the correlation of loss and burst size. Figure 7 shows, for each burst size, the fraction of bursts of that size that experience at least one retransmission. Trickle not only reduces the bursts size but the chances of losses with any given burst size. We believe one factor is because the transmission is better ack-clocked and has a smaller queue occupancy, as we will show in Section 5.3. Interestingly the two baseline groups suggest high drop rate for bursts of size 11, 17, or 35. The shrunk-block group shows half of the bursts of size 23 and 35 will experience losses. We are not able to provide a satisfactory explanation due to the lack of information at the bottlenecks, which are likely to reside at the last mile. We hypothesize the phenomenon is produced by some common buffer configurations interacting with YouTube application pacing.

### 5.3 Queueing Delay

Sending smaller bursts not only improves loss rate, it may also help reduce the maximum queue occupancy on bottleneck links. It is certainly not uncommon for users to watch online videos while surfing the Web at the same time. Since networks today are commonly over-buffered [23], shorter queue length improves the latency of interactive applications sharing the link.

We evaluate the impact of queue length by studying the RTT measurements in TCP, due to the lack of di-

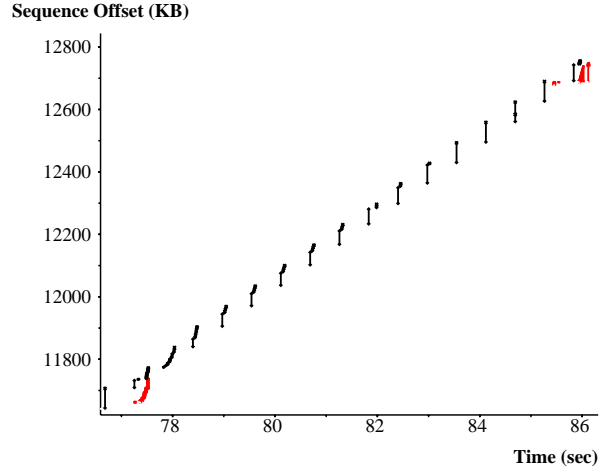


Figure 6: Time vs. sequence of bytes graph of a flow in baseline1 group. The black arrows are data bursts and the red dots are retransmits. After it experiences losses, the 64kB application writes are fragmented into smaller bursts.

rect information of the bottleneck queues. Recall that a RTT sample in TCP includes both the propagation delay and queueing delay. Given that the four experiment groups receive similar load, the propagation delay distribution in each group should be close. Each video stream often has hundreds to thousands of RTT samples partly because Linux samples RTT per ACK packet. In order to reduce the sample size, we instead use the smoothed RTT (*srtt*) variable at the end of the connection. Since *srtt* is a weighted moving average of all the RTT samples, it should reflect the overall queueing delay during the connection.

Figure 8 plots the CDF of the *srtt* samples for DC1. On average, the *srtt* of connections in the Trickle group is 28% and 10% smaller than the connections in the baseline and shrunk-block groups, respectively. In DC2, the improvement over baseline is only 7% and 1%. The reason is similar to the analysis in Section 5.1: throttling is seldom activated on the slow links in DC2. We measured that the links in India are alarmingly over-buffered: 20% of the *srtt* samples were over 1 second while 2% were over 4 seconds. While these videos are likely being streamed in the background, the interactive applications sharing the same bottleneck queue certainly will suffer extremely high latency. In summary, for fast networks, Trickle connections experience much lower queueing delays, which should improve interactive application latencies. For slow users, the solution is to use Trickle but serve at a lower rate (lower resolution video).



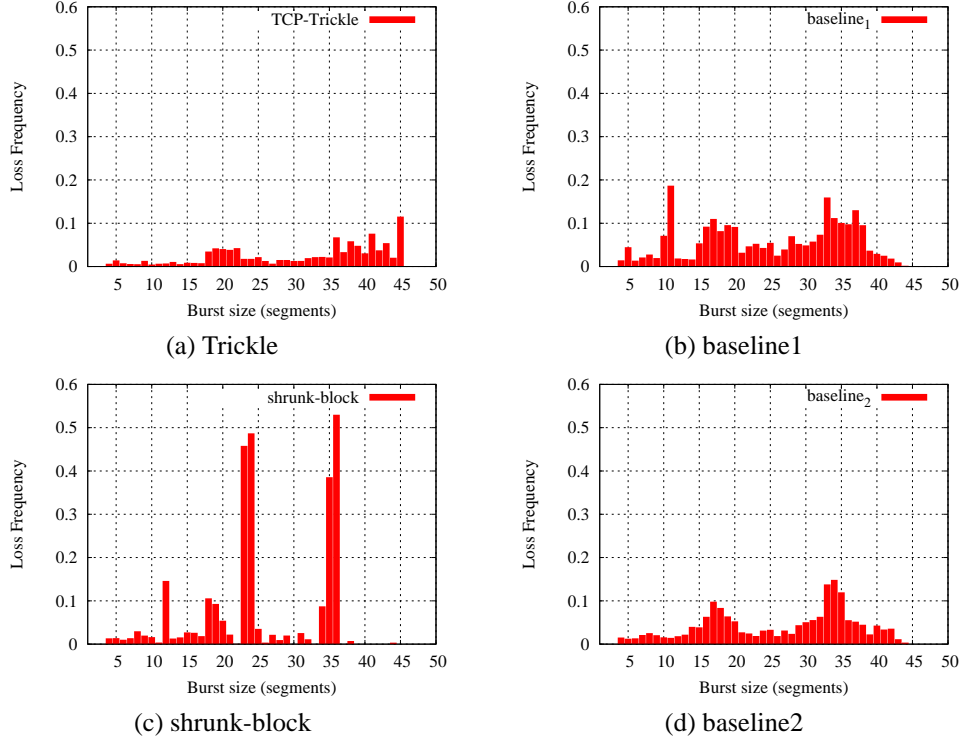


Figure 7: Frequency of losing at least one segment as a function of burst size in DC1.

## 5.4 Rebuffering

Rebuffering happens when a reduction in throughput within a TCP streaming session causes receiver buffer starvation. When this happens, the video player stops playing video until it receives enough packets. Rebuffering rate is an important metric in video streaming as it reflects user experience watching videos.

YouTube has a built-in mechanism to provide real-time monitoring of video playbacks. During a playback, the video player sends detailed information about user interactions to the server. The information includes the timestamps of all rebuffering events in each TCP connection.

To quantify the user perceived performance of video streaming, we use rebuffering chance and rebuffering frequency suggested by previous works [19]. The rebuffering chance measures the probability of experiencing rebuffering events and is defined by percentage of flows that experience at least one rebuffering event. Rebuffering frequency measures how frequent the rebuffering events occur and is defined by  $r/T$ , where  $r$  is the number of rebuffering events and  $T$  is the duration of a flow.

Table 3 shows the average of rebuffering metrics in DC1 and DC2. DC2 users clearly have much worse expe-

	DC1		DC2	
	rebuff. freq. (1/s)	rebuff. chance (%)	rebuff. freq. (1/s)	rebuff. chance (%)
baseline1	0.0005	2.5%	0.005	26%
baseline2	0.0005	2.5%	0.005	26%
Trickle	0.0005	2.5%	0.005	26%
shrunk-block	0.0005	2.5%	0.005	27%

Table 3: A comparison of rebuffering frequency and rebuffering chance.

rience than DC1 users. However, in both data centers the rebuffering frequency and rebuffering chance are similar between all four groups, suggesting Trickle has negligible impact on the streaming quality.

Initially the results puzzled us as we expected Trickle to improve rebuffering by reducing burst drops. To explain the results, we studied the correlation of rebuffering and various network metrics. We found that the bandwidth deficit, the difference between the target streaming rate and the bandwidth, is the key factor for rebuffering. In both DC1 and DC2, among the flows that do not have sufficient bandwidth (positive deficit), 55% to 60% of them have experienced at least one rebuffering event.



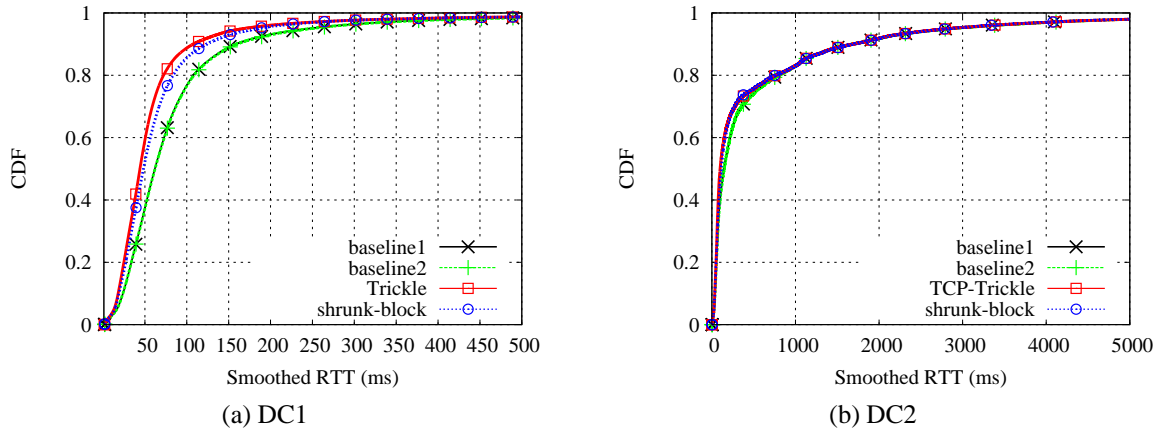


Figure 8: CDF of the smoothed RTT ( $srtt$ ) samples in DC1 and DC2. In DC1, the average  $srtt$  of connections in Trickle is 28% and 10% lower than the connections in the baselines and shrunk-block, respectively. In DC2 the numbers are 7% and 1%.

Another major factor is when a user requests a different resolution by starting a new connection.

However, the correlation between rebuffering and retransmission rate is less than 5%. This is because the browser typically buffers enough video that losses rarely cause buffer under-run. The exception is when the available bandwidth is or becomes too low, in which Trickle can not help that since the congestion is not caused by burst drops.

## 6 Discussions and Related Work

Trickle is motivated by Alcock *et al.*'s work [2], which identified a YouTube burst drops problem in residential broadband and university networks. Further, Ashwin *et al.* showed that popular browsers also throttle the video streaming in addition to server side throttling in YouTube and Netflix [6]. The bilateral throttling mechanisms sometimes result in packet bursts up to several MBs. Blanton *et al.* studied the correlation between burst size and losses in TCP [7]. They discovered that bursts less than 15 packets rarely experience loss but large (over 100) bursts nearly always do. Allman *et al.* evaluated several changes to mitigate bursts created by TCP [3]. We did not try these solutions because the bursts in YouTube are created by the application, not TCP congestion control.

We have also considered other solutions to rate limit video streaming. A similar idea that requires no kernel TCP change is to set the TCP send socket buffer size [20]. In the case of YouTube, the ustreamer TCP send buffer remains auto-tuned [21] during the startup phase in order to send data as fast as possible. Upon entering the throttling phase, the buffer usually is already larger than the intended clamp value. Setting a new

send buffer size is not effective until the buffered amount drops below the new size, making it difficult to implement the throttling. Some previous work control the rate by dynamically adjusting the TCP receive window at the receiver or the gateway [15, 17, 22]. Instead, Trickle is server-based making it easier to deploy in a CDN.

Another approach is TCP pacing [24], i.e., pacing  $cwnd$  amount of data over the RTT. While this may be the best TCP solution to suppress bursts, it is also more complex to implement. Moreover, studies have shown that Internet paths can absorb small amount of packet bursts [7, 10]. Our goal is to reduce large burst drops caused by disruptions to the TCP self clocking. It is not to eliminate any possible burst completely.

Delayed-based TCP congestion controls, e.g., Vegas [8], may be another interesting solution to avoid burst drops by detecting queues early. However it may not directly address the problem because these algorithms still require proper ACK-clocking. But we plan to experiment Trickle with delay-based congestion controls in the future.

A non-TCP approach to consider is traffic shaping underneath TCP. To our knowledge most software implementations, e.g., the Linux tc, can not scale to support the number of connections of YouTube servers. Doing this may require dedicated hardware while Trickle is a relatively simple yet inexpensive solution. DCCP [16] supports unreliable delivery with congestion control based on UDP. While DCCP is suitable for video streaming, our work focuses on improving the current HTTP / TCP model.

## 7 Conclusions

The current throttling mechanism in YouTube sends bursts that cause losses and large queues. We presented Trickle, which removes these large bursts by doing rate-limiting in TCP. Trickle dynamically sets a maximum *cwnd* to control the streaming rate and strictly limit the maximum size of bursts. Through large-scale real user experiments, Trickle has effectively reduced the retransmissions by up to 50% in high bandwidth networks. It also reduces the average RTT by up to 28%.

Trickle requires minimal sender-side changes, which allows fast deployment at the content providers. It can be generically applied to rate-limit other kinds of video streaming or similar transfers. For example, variable bit rate video streaming can use Trickle to serve individual encoded blocks at different rates. We are actively deploying Trickle on YouTube servers and monitoring its performance impact on various networks and devices. We are also looking into new systems to improve the bandwidth efficiency and rebuffering events to improve user experience. We are hopeful that both Trickle in YouTube and this paper can help reduce the burst stress in the Internet.

## 8 Acknowledgments

We would like to thank Tayeb Karim, Yu-Ting Tseng, Bernhard Beck, Brian Rogan, Nandita Dukkhipati, Dean Sturtevant, Alex Nelson, Andy Berkheimer, Alan Su, Neal Cardwell, and Mukarram Bin Tariq, John Wilkes, Phillipa Gill and Farhang Farid.

## References

- [1] Sandvine global Internet report, Oct. 2011.
- [2] ALCOCK, S., AND NELSON, R. Application flow control in YouTube video streams. *CCR 41* (April 2011), 24–30.
- [3] ALLMAN, M., AND BLANTON, E. Notes on burst mitigation for transport protocols. *CCR* (Apr. 2005), 53–60.
- [4] ALLMAN, M., EDDY, W. M., AND OSTERMANN, S. Estimating loss rates with TCP. *SIGMETRICS 31* (December 2003), 12–24.
- [5] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP congestion control, September 2009. RFC 5681.
- [6] ASHWIN, R., ARNAUD, L., YEON, L., DON, T., CHADI, B., AND WALID, D. Network characteristics of video streaming traffic. *CoNEXT* (Dec. 2011), 1–12.
- [7] BLANTON, E., AND ALLMAN, M. On the impact of bursting on TCP performance. *PAM* (March 2005), 1–12.
- [8] BRAKMO, L., AND PETERSON, L. TCP Vegas: end to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications 13*, 8 (1995), 1465–1480.
- [9] DUKKIPATI, N., MATHIS, M., CHENG, Y., AND GHOBADI, M. Proportional rate reduction for TCP. *IMC* (November 2011), 155–170.
- [10] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An argument for increasing TCP’s initial congestion window. *CCR 40* (2010), 26–33.
- [11] GRINNEMO, K., AND BRUNSTROM, A. Impact of traffic load on SCTP failovers in SIGTRAN. *ICN* (2005), 774–783.
- [12] HA, S., RHEE, I., AND XU, L. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS’08 42*, 64–74.
- [13] HANDLEY, M., PADHYE, J., AND FLOYD, S. TCP congestion window validation, June 2000. RFC 6298.
- [14] JIANG, H., AND DOVROLIS, C. Source-level IP packet bursts: causes and effects. *IMC* (October 2003), 301–306.
- [15] KARANDIKAR, S., KALYANARAMAN, S., BAGAL, P., AND PACKER, B. TCP rate control. *CCR 30* (January 2000), 45–58.
- [16] KOHLER, E., HANDLEY, M., AND FLOYD, S. Designing DCCP: congestion control without reliability. *SIGCOMM* (September 2006), 27–38.
- [17] LOMBARDO, A., PANARELLO, C., AND SCHEMBRA, G. Applying active window management for jitter control and loss avoidance in video streaming over TCP connections. *IEEE Globecom* (December 2010), 1–6.
- [18] MOHAMMAD, R., AND ANNA, B. On the Effectiveness of PR-SCTP in Networks with Competing Traffic. *ISCC* (2011), 898–905.
- [19] MOK, R., CHAN, E., AND CHANG, R. Measuring the quality of experience of HTTP video streaming. *IFIP* (May 2011), 485–492.
- [20] PRASAD, R., JAIN, M., AND DOVROLIS, C. Socket buffer auto-sizing for high-performance data transfers. *Journal of GRID computing 1*, 4 (2004), 361–376.
- [21] SEMKE, J., MAHDAVI, J., AND MATHIS, M. Automatic TCP buffer tuning. *CCR 28* (October 1998), 315–323.
- [22] SPRING, N., CHESIRE, M., BERRYMAN, M., SAHASRANAMAN, V., ANDERSON, T., AND BERSHAD, B. Receiver based management of low bandwidth access links. *INFOCOM* (March 2000), 245–254.
- [23] TAHT, D., GETTYS, J., AND TURNER, S. The bufferbloat problem, 2011. <http://www.bufferbloat.net/>.
- [24] ZHANG, L., SHENKER, S., AND CLARK, D. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *CCR 21* (August 1991), 133–147.