Anatomy of a Personalized Livestreaming System

Bolun Wang[†], Xinyi Zhang[†], Gang Wang[†‡], Haitao Zheng[†] and Ben Y. Zhao[†]
[†]Department of Computer Science, UC Santa Barbara
[‡]Department of Computer Science, Virginia Tech
{bolunwang, xyzhang, gangw, htzheng, ravenben}@cs.ucsb.edu

ABSTRACT

With smartphones making video recording easier than ever, new apps like Periscope and Meerkat brought personalized interactive video streaming to millions. With a touch, viewers can switch between first person perspectives across the globe, and interact in real-time with broadcasters. Unlike traditional video streaming, these services require low-latency video delivery to support high interactivity between broadcasters and audiences.

We perform a detailed analysis into the design and performance of Periscope, the most popular personal livestreaming service with 20 million users. Using detailed measurements of Periscope (3 months, 19M streams, 705M views) and Meerkat (1 month, 164K streams, 3.8M views), we ask the critical question: "Can personalized livestreams continue to scale, while allowing their audiences to experience desired levels of interactivity?" We analyze the network path of each stream and break down components of its end-to-end delay. We find that much of each stream's delay is the direct result of decisions to improve scalability, from chunking video sequences to selective polling for reduced server load. Our results show a strong link between volume of broadcasts and stream delivery latency. Finally, we discovered a critical security flaw during our study, and shared it along with a scalable solution with Periscope and Meerkat management.

1. INTRODUCTION

The integration of high quality video cameras in commodity smartphones has made video recording far more convenient and accessible than ever before. In this context, new mobile apps such as Periscope and Meerkat now offer users the ability to broadcast themselves and their surroundings using real-time interactive live streams. With the flick of a finger, a user can switch from a first person view of Carnival in Rio, a guided stroll outside the Burj Khalifa in Dubai, or a live view of swans in Lake Como, Italy. What makes these apps compelling is that viewers can interact in real time with broadcasters, making requests, asking questions and giving direct feedback via "likes." Today, popular livestreams capture thousands of users, and can cover everything from press conferences, political protests, personal interviews, to backstage visits to concerts, TV shows, and sporting events.

Unlike existing video-on-demand streaming services or live video broadcasts, real time interactivity is critical to the livestreaming experience for both streamers and their audience. First, applications like Periscope allow audience to generate "hearts" or "likes," which is directly interpreted by the broadcaster as positive feedback on the content, *e.g.* "applause." Second, many streams involve broadcasters soliciting input on content or otherwise "poll" the audience. Explicit support for polling is an feature on Twitter that has yet to be integrated into Periscope. In both cases, the immediacy of the feedback is critical, and delayed feedback can produce negative consequences. For example, a "lagging" audience seeing a delayed version of the stream will produce delayed "hearts," which will be misinterpreted by the broadcaster as positive feedback for a later event in the stream. Similarly, a delayed user will likely enter her vote after the real-time vote has concluded, thus discounting her input. Participants in a recent user study answered that as broadcasters, the immediate interaction with audience members was an authentic, higher level of engagement, and it was valuable input for their work and personal branding [45].

Minimizing streaming delay is already a significant challenge for livestreaming services today. To minimize delay for those commenting on broadcasts, Periscope only allows 100 viewers to comment on a broadcast (usually the first 100 to join the stream) [30]. In practice, the first 100-200 Periscope users to join a stream are connected to a more direct distribution network with much lower delay (running Real Time Messaging Protocol (RTMP) [2]), and later arrivals to a high delay CDN for better scalability (running HTTP Live Streaming (HLS) [3]). While Periscope claims this is to avoid overwhelming broadcasters, users have already found this "feature" highly limiting, and have proposed multiple hacks to circumvent the system [22, 19, 20]. Not only does this approach artificially limit user interactions, but it also prevents more advanced modes of group in-

IMC 2016, November 14-16, 2016, Santa Monica, CA, USA
 ISBN 978-1-4503-4526-2/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2987443.2987453

teraction, *e.g.* instantaneous polls for the audience (already a feature inside Twitter).

In this paper, we study the critical issue, "can personalized livestreaming services like Periscope continue to scale up in viewers and streams, while allowing large audiences to experience the interactivity so core to their appeal?" This leads us to perform an experimental study of Periscope and its main rival, Meerkat[1] [2]. We perform detailed measurements of Periscope over 3 months in 2015, capturing activity from 19 million streams generated by 1.85 million broadcasters, with a total of 705 million views (482M via mobile and 223M via web). We also gathered measurement data of Meerkat over 1 month, including 164K streams generated by 57K broadcasters, with a total number of 3.8 million views.

Our work is focused around three specific questions. First, we need to understand how popular these systems are, how they're used by broadcasters and viewers. We want to understand current traffic loads, as well as longitudinal growth trends in both users and streams. Second, we need to understand the structure of the livestream delivery infrastructure, and its contributions of end-to-end delivery delay. More importantly, we are interested in understanding what the tradeoffs are between delay and scalability, and how expected growth in users is to impact streaming delay. Finally, can the current system be optimized for improved performance, and how is continued growth likely to affect future performance?

Our study produces several key findings:

- We find that livestream services are growing rapidly, with Periscope more than tripling number of daily streams in 3 months. In contrast, its competitor Meerkat is rapidly losing popularity, *e.g.* losing half its daily streams in a single month. Similar trends are observable in daily active users. In network structure, Periscope more resembles the structure of Twitter (likely due to the role of asymmetric links in both networks), and less that of Facebook (bidirectional links).
- In Periscope, a combination of RTMP and HLS protocols are used to host streams. For small streams, RTMP using server-side push minimizes end-to-end delay. For the most popular streams, chunking is used with HLS to reduce server-side overhead, which introduces significant delays from both chunking and client-side polling. In both cases, end-to-end delay is significantly exacerbated by aggressive client-side buffering. Barring a change in architecture, more streams will require servers to increase chunk sizes, improving scalability at the cost of higher delays.
- We use stream traces to drive detailed simulations of client-side playback, and find that current client-side buffering strategies are too aggressive. We believe client-side buffers (and associated latency) can be reduced by half while still maintaining current playback quality.
- Finally, we find a significant vulnerability to stream hijacking in the current architectures for both Periscope and Meerkat, possibly driven by a focus to scale up number of streams. We propose a lightweight solution and informed management teams at both companies[3].

The results of our work serve to highlight the fundamental tension between scalability and delay in personalized livestream services. As Periscope and similar services continue to grow in popularity (Periscope is now being integrated into hardware devices such as GoPro cameras [7]), it remains to be seen whether server infrastructure can scale up with demand, or if they will be forced to increase delivery latency and reduce broadcaster and viewer interactivity as a result.

2. BACKGROUND AND RELATED WORK

2.1 The Rise of Periscope and Meerkat

Meerkat was the first personalized livestreaming service to go live, on February 27, 2015 [38]. It was a smartphone app integrated with Twitter, using Twitter's social graph to suggest followers and using Tweets for live comments to broadcasts. Two weeks later, Twitter announced its acquisition of Periscope [13]. Within a week, Twitter closed its social graph API to Meerkat, citing its internal policy on competing apps [32]. Our measurement study began in May 2015, roughly 2 months after, and captured both the rise of Periscope and the fall of Meerkat. By December 2015, it is estimated that Periscope has over 20 million users [37].

Periscope. The initial Periscope app only supported iOS when it launched in March 2015. The Android version was released on May 26, 2015. The app gained 10 million users within four months after launch [15]. On Periscope, any user can start a live video broadcast as a *broadcaster*, and other users can join as *viewers*. While watching a broadcast, viewers can send text-based comments or "hearts" by tapping the screen. For each broadcast, only the first 100 viewers can post comments, but all viewers can send hearts. Hearts and comments are visible to the broadcaster and all viewers.

All active broadcasts are visible on a global public list. In addition, Periscope users can follow other users to form directional social links. When a user starts a broadcast, all her followers will receive notifications. By default, all broadcasts are public for any users to join. Users do have the option to start a *private* broadcast for a specific set of users.

Facebook Live. Facebook Live was initially launched in August 2015 (called "Mentions") as a feature for celebrities only. In December 2015, Facebook announced that the app would open to all users. Facebook Live went live on January 28, 2016. While we have experimented with the new functionality during its beta period, it is unclear how far the rollout has reached. Our paper focuses on Periscope because of its scale and popularity, but we are considering ways to collect and add Facebook Live data to augment our measurement study.

[1]Facebook only recently began a similar service called Facebook Live, and its user count is far smaller than its two earlier rivals.

[2]Our study has obtained approval from our local IRB.

[3]We informed CEOs of both Periscope and Meerkat in August, 2015 to ensure they had plenty of time to implement and deploy fixes before this paper submission.

2.2 Related Work

Live Streaming Applications. Researchers have studied live streaming applications such as CNLive [28], and Akamai live streaming [43] with focus on user activities and network traffic. Unlike Periscope and Meerkat, these applications do not support real-time interactivity between users. Siekkinen *et al.* studied Periscope with a focus on user experience and energy consumption, using experiments in a controlled lab setting [39]. Twitch.tv is a live streaming service exclusively for gaming broadcast [21, 16, 48]. Zhang *et al.* use controlled experiments to study Twitch.tv's network infrastructure and performance [48]. Tang *et al.* analyzed content, setting, and other characteristics of a small set of Meerkat and Periscope broadcasts, and studied broadcasters' motivation and experience through interviews [45]. Compared to prior work, our work is the first large-scale study on personalized live streaming services (*i.e.,* Periscope and Meerkat) that support real-time interactivity among users.

A related line of work looks into the peer-to-peer (P2P) based live streaming applications [42]. In these systems, users form an overlay structure to distribute video content. The system is used to stream live video but does not support user interactivity. Researchers have measured the traffic patterns in popular P2P live streaming systems [17, 40] and proposed mechanisms to improve scalability [49, 31, 41].

Streaming Protocol. Existing works have studied streaming protocols under the context of Video-on-Demand systems. Most studies have focused on HTTP-based protocols such as DASH, HDS, and HLS including performance analysis [27, 33, 25] and improvement of designs [47, 18]. Others have examined the performance of non-HTTP streaming protocols such as RTMP in Video-on-demand systems [26]. Fewer studies have examined streaming protocols in the context of live streaming. Related works mostly focus on HTTP based protocols [29, 11].

Content Distribution Network (CDN). Most existing literatures focus on general-purpose CDNs that distribute web content or Video-on-Demand. Su *et al.* and Huang *et al.* measure the CDN performance (latency, availability) for popular CDNs such as Akamai and LimeLight [44, 6]. Adhikari *et al.* measure the CDN bandwidth for Video-on-demand applications such as Netflix [10] and Hulu [9]. Krishnan *et al.* analyze CDN internal delay to diagnose networking issues such as router misconfigurations [24]. Kontothanassis *et al.* explore Akamai's CDN design for regular media streaming [23]. Few studies have looked into CDNs to deliver "real-time" content. In our study, we focus on live streaming services that also demand a high level of real-time user interactivity. We analyze streaming protocols and CDN infrastructures in Periscope to understand the trade-offs between latency and scalability.

3. BROADCAST MEASUREMENTS

We perform detailed measurements on Periscope and Meerkat to understand their scale, growth trends and user activities. Our goal is to set the context for our later analysis on stream delivery infrastructures. In the following, we first describe the methodology of our data collection and the resulting datasets, and then present the key observations. Our primary measurements focus on Periscope. For comparison, we perform similar measurements and analysis on Meerkat, Periscope's key competitor.

App	Months in 2015	Broadcasts	Broadcasters	Total Views	Unique Viewers
Periscope	3	19.6M	1.85M	705M	7.65M
Meerkat	1	164K	57K	3.8M	183K

Table 1: Basic statistics of our broadcast datasets.

Network	Nodes	Edges	Avg. Degree	Cluster. Coef.	Avg. Path	Assort.
Periscope	12M	231M	38.6	0.130	3.74	-0.057
Facebook [46]	1.22M	121M	199.6	0.175	5.13	0.17
Twitter [36]	1.62M	11.3M	13.99	0.065	6.49	-0.19

Table 2: Basic statistics of the social graphs.

3.1 Data Collection

Our goal is to collect a complete set of broadcasts on the Periscope and Meerkat networks. For each network, we analyze the network traffic between the app and the service, and identify a set of APIs that allows us to crawl their global broadcast list. Our study was reviewed and approved by our local IRB. Both Periscope and Meerkat were aware of our research, and we shared key results with them to help secure their systems against potential attack (§7).

Data Collection on Periscope. To collect a complete set of broadcasts/streams on Periscope, we build a crawler to monitor its global broadcast list. The global list shows 50 random selected broadcasts from all active broadcasts. To obtain the complete list, we use multiple Periscope accounts to repeatedly query the global list. Each account refreshes the list every 5 seconds (the same frequency as the Periscope app) and together we obtain a refreshed list every 0.25 seconds. While our experiments show that a lower refresh rate (once per 0.5 seconds) can already exhaustively capture all broadcasts, *i.e.,* capture the same number of broadcasts as once per 0.25 seconds, we use the higher refresh rate to accommodate potential burst in the creation of broadcasts. Whenever a new broadcast appears, our crawler starts a new thread to join the broadcast and records data until the broadcast terminates. For each broadcast, we collect the broadcastID, starting and ending time of the broadcast, broadcaster userID, the userID and join time of all the viewers, and a sequence of timestamped comments and hearts. Only metadata (no video or message content) is stored, and all identifiers are securely anonymized before analysis.

We ran our crawler for 3+ months and captured all Periscope broadcasts between May 15, 2015 and August 20, 2015[4]. As listed in Table 1, our dataset includes 19,596,779 broad-

[4]We received a whitelisted IP range from Periscope for active measurements, but our new rate limits were unable to keep up with the growing volume of broadcasts on Periscope.

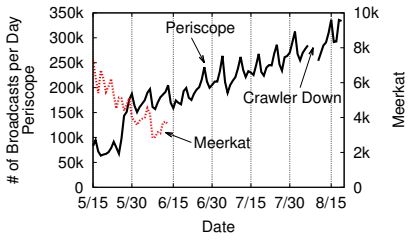


Figure 1: # of daily broadcasts.

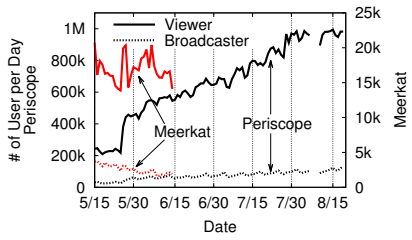


Figure 2: # of daily active users.

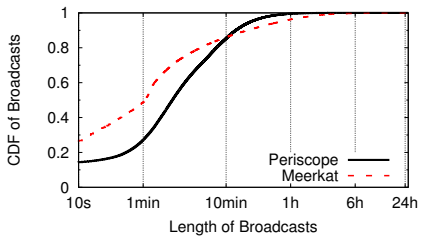


Figure 3: CDF of broadcast length.

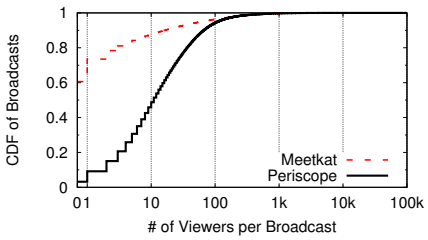


Figure 4: Total # of viewers per broadcast.

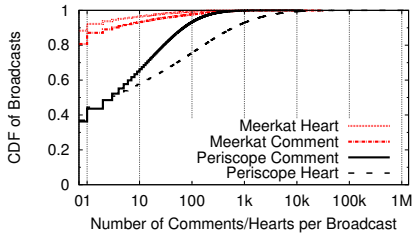


Figure 5: Total # of comments (hearts) per broadcast.

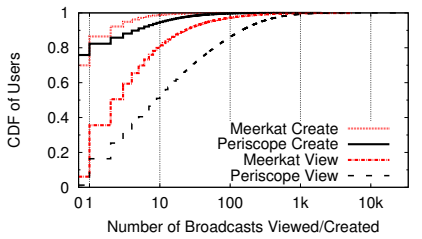


Figure 6: Distribution of broadcasts views and creation over users.

casts, generated by 1,847,009 unique broadcasters. Altogether, these broadcasts had a total number of 705,751,096 viewers, including both registered users on mobile apps and anonymous viewers on web browsers. Of these, more than 482 million views were generated by 7,649,303 registered mobile users, and the rest were from anonymous views on the Periscope webpage.

Our crawler was interrupted briefly between August 7–9, 2015 due to a communication bug with the Periscope server. As a result, our dataset is missing roughly 4.5% of the broadcasts during this period. We believe this missing data is small enough not to affect our data analysis or conclusions. Finally, since Periscope assigns userID sequentially[5], we can identify the total number of registered users. As of August 20, 2015 (the last day of our measurement), Periscope had 12 million registered users. For each user, we crawled her follower and followee lists to analyze the structure of the social graph.

Data Collection on Meerkat. We use the same methodology to collect broadcast data from Meerkat between May 12 and June 15, 2015. After one month of data collection, Meerkat management informed us our measurements were introducing noticeable load on their servers, and we terminated our measurements. As shown in Table 1, the Meerkat dataset includes a complete list of 164,335 broadcasts, involving 189,075 unique users (57K broadcasters, 183K viewers). During our measurement period, these broadcasts collected a total of more than 3.8 million views, of which 3.1 million views were generated by 183K viewers. Since Meerkat does not assign userID sequentially, and the follower/followee graph was not fully connected at the time of our measurements, we were unable to reliably estimate the total number of registered users.

[5]At launch, Periscope assigned userIDs as sequential numbers. In September 2015, they switched to a 13-character hash string.

3.2 Analysis: Broadcasts & Users

Using the collected datasets, we now explore application-level characteristics of Periscope and Meerkat. We focus on three aspects: 1) scale and growth trends in terms of number of users and broadcasts; 2) duration and intensity of user interactions during broadcasts; and 3) Periscope's social network and its impact on broadcast popularity.

Scale & Growth. Figure 1 plots the number of daily broadcasts over time on both networks. For Periscope, the daily broadcast volume grew rapidly over time (more than 300% over three months). This rapid growth is consistent with prior observations of initial network growth [50]. The biggest leap occurred shortly after May 26, when Periscope launched their Android app. The broadcasts show a clear weekly pattern, where peaks in daily broadcasts usually coincide with the weekend. Broadcasts drop to their lowest weekly level on Mondays, then slowly grow during the week. In contrast, the volume of Meerkat broadcasts is orders of magnitude lower, weekly patterns are harder to identify, and daily broadcast volume nearly dropped by half over a single month to below 4000. It is clear that closing off the Twitter social graph had the intended effect on Meerkat.

Similarly, Periscope's daily number of active users (both viewers and broadcasters) also grows rapidly (Figure 2). Viewers jumped from 200K in May to more than a million by August. The ratio between the active viewers and broadcasters is roughly 10:1. In comparison, Meerkat daily viewers fluctuates, but averages around 20K. Meerkat broadcasters show a clear descending trend over our measurement period, eventually dropping to less than 3K (roughly 40 times smaller than Periscope).

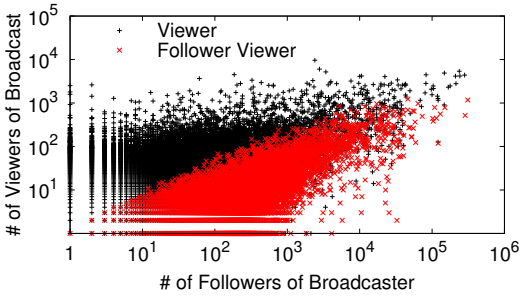


Figure 7: Broadcaster's followers vs. # of viewers (Periscope).

Broadcast Duration & User Interaction. For both networks, the majority of broadcasts are short: 85% of broadcasts last <10 minutes (Figure 3). Periscope broadcasts are more even in length, whereas Meerkat streams are more skewed by a smaller number of longer broadcasts. In terms of viewers, the two systems are quite different. For Meerkat, most (60%) broadcasts have no viewers at all. In contrast, nearly all Periscope broadcasts received at least one viewer, with the most popular broadcasts attracting as many as 100K viewers. Anecdotal evidence shows that recent Periscope sessions have grown much bigger: a single Periscope of a large rain puddle collected hundreds of thousands of viewers, and had more than 20,000 simultaneous viewers at its peak [14].

Within both Periscope and Meerkat broadcasts, viewers interact with the broadcast by posting real-time comments and hearts. As shown in Figure 5, while not all broadcasts received responses, a small portion of popular broadcasts are highly interactive. As expected, Periscope broadcasts generate much more user interactions. About 10% Periscope broadcasts received more than 100 real-time comments and more than 1000 hearts. The most popular broadcast attracted 1.35 million hearts. Recall that the number of total viewers per broadcast who can comment is limited by Periscope to roughly 100 (presumably for scalability reasons). This puts a strong cap on the total number of comments.

We wanted to take a closer look at the distribution of activities over each network's user population. In Figure 6, we plot the CDF distribution of broadcast views and broadcast creation activities over users. Here the trends are roughly consistent across Periscope and Meerkat: user activity is highly skewed, and small groups of users are much more active than average in terms of both broadcast creation and viewership. For Periscope, viewers are especially skewed, with the most active 15% of users watching 10x more broadcasts than the median user.

Social Network. We found that the social network in Periscope has a clear impact on the broadcast viewership. Figure 7 shows the correlation between a broadcaster's number of followers and number of viewer per broadcast. It's clear that users with more followers are more likely to generate highly popular broadcasts. This is because when a user starts a new broadcast, all her followers will receive notifications on their mobile app. On Periscope, celebrities like Ellen DeGeneres already have over one million followers, thus creating built-in audiences for their broadcasts. As we observed in Table 2, Periscope's follow graph is similar in graph structure to social graphs in Twitter and Facebook. Its follow graph exhibits negative assortativity similar to the Twitter graph, which likely comes from the prevalence of asymmetric, one-to-many follow relationships.

Summary. Our measurements show that Periscope is experiencing astounding growth in users and broadcast streams. Growth in broadcasts is particularly noteworthy, as it is the main contributor to issues of scalability in video delivery. On March 28, 2016, Periscope hit a milestone of 200 million total broadcasts [35]. And a quick check at the time of this submission (May 8, 2016) shows an average of roughly 472K daily broadcasts, and the numbers are still growing steadily.

At this rate, Periscope will face even bigger scalability challenges in the near future. We also observe that broadcasts can attract a large number of viewers (up to 100K). These broadcasts are also highly interactive, with frequent, real-time hearts from viewers to broadcasters. Periscope comments, in contrast, are severely constrained by their limit of 100 active users. Given the audience size of some broadcasts, it is clear that more inclusive support for group interaction is necessary in these broadcasts.

4. UNDERSTANDING END-TO-END DELAY

Our initial measurements have demonstrated the scale and real-time features of personalized livestream services. As they continue to grow, a natural question arises:

"*Can personalized livestreams scale to millions of streams, and what, if any, technical challenges limit their scalability*?"

To answer this question, we must first study how Periscope delivers its streams over its content distribution network (CDN), which is responsible for delivering real-time video streams to viewers and scaling up to a massive number of simultaneous live streams. In particular, we seek to understand the key design choices that Periscope has made to support both scalability and real-timeness, and the resulting system performance.

Our analysis of Periscope broadcast delivery includes four elements.

- We explore Periscope's CDN infrastructure to support message delivery and live video streaming services. We reverse-engineer their streaming protocols and CDN locations, and compare with Meerkat and Facebook Live (§4).
- We perform detailed measurements on Periscope CDN by breaking-down the video transmission delay to each step of the transmission process (§4.2).
- Based on collected data, we perform detailed analysis on streaming delay and explore key design trade-offs between delay and scalability (§5).
- We analyze the tradeoff between scalability and latency in the context of the three largest contributors to delay, including chunking and polling (§5.2), geolocation factors in CDN servers (§5.3), and client-side buffering strategies (§6).

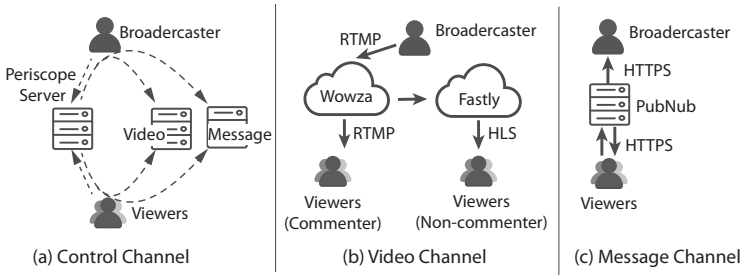


Figure 8: Periscope CDN infrastructure.

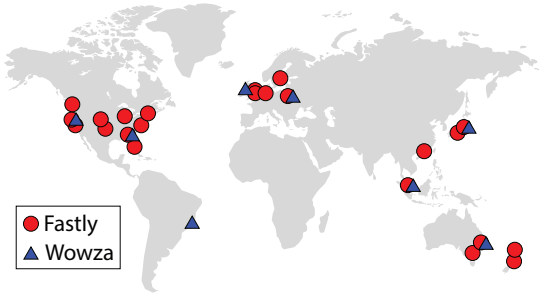


Figure 9: Wowza and Fastly server locations.

4.1 Video Streaming CDN and Protocols

We start by describing Periscope's CDN infrastructure. By analyzing the network traffic between Periscope app and server, we found that Periscopes uses two independent channels to deliver live video content and messages (comments/hearts) and the Periscope server only acts as a control panel (see Figure 8). Specifically, when a user starts a broadcast, Periscope redirects the user to a video server to upload live video, and a message server to receive real-time comments and hearts. Viewers receive video frames and messages and combine them on the client side based on timestamps. For messaging, Periscope uses a third-party service called PubNub and users connect to the PubNub via HTTPS. For video, Periscope leverages collaboration among multiple CDNs.

In the following, we focus on Periscope video CDN and describe our reverse-engineering efforts to understand its streaming protocols and server deployment.

Streaming Protocols. As shown in Figure 8(b), Periscope uses two CDNs, Wowza [8] and Fastly [4], to handle video uploading (from broadcasters) and downloading (to viewers). Wowza uses the RTMP protocol, and Fastly uses the HLS protocol, which are two fundamentally different video streaming protocols. In RTMP, the client maintains a persistent TCP connection with the server. Whenever a video frame ($\approx$ 40ms in length) is available, the server "pushes" it to the client. In HLS, Wowza servers assemble *multiple* video frames into a chunk ($\approx$ 3s in length), and create a chunk list. Over time, Fastly servers poll Wowza servers to obtain the chunk list. Each HLS viewer periodically "polls" the chunk list from the (Fastly) server and downloads new chucks.

When creating a new Periscope broadcast, the broadcaster connects to a Wowza server to upload live video. Wowza maintains a persistent RTMP connection with the broadcaster's device during the entire broadcast. Wowza also transmits the live video content to Fastly, and thus both Wowza and Fastly distribute the video content to end-viewers. The first batch of viewers to join a broadcast directly connect to Wowza to receive video using RTMP. Our tests estimate the number to be around 100 viewers per broadcast. Once the threshold is reached, additional viewers automatically connect to Fastly to download video using HLS. Recall that Periscope's default policy allows only the first 100 viewers to post comments in each broadcast [30]. Those *commenters* are effectively the first batch of viewers who connect directly to Wowza, and receive their streams via low delay from RTMP.

We believe that Periscope adopts this "hybrid" approach to improve scalability. Due to its use of persistent TCP connections, RTMP offers low-latency streaming when the number of viewers is small (<100), while HLS provides scalable streaming (with higher delay) to a large number of users. Among the complete set of periscope broadcasts (19.6M) over the three months, 1.13M broadcasts (5.77%) had at least one 1 HLS viewer, and 435K had at least 100 HLS viewers.

Server Distribution. Both Wowza and Fastly have distributed data centers around the globe. Fastly provides the IP range and location of each data center on its website [5, 1]. At the time of our measurements, Periscope uses all 23 data centers of Fastly covering North America, Europe, Asia, and Oceania [6], as shown in Figure 9. Wowza does not offer any public information on its data centers thus we obtained both IPs and locations using a simple experiment. We used 273 PlanetLab nodes (from 6 continents) to join all the broadcasts as viewers on June 11, 2015. By analyzing the resolved IPs from different PlanetLab nodes, we found that Wowza runs on 8 Amazon EC2 datacenters, as shown in Figure 9. Interestingly, for 6 out of 8 Wowza datacenters, there is a Fastly datacenter co-located in the same city, and 7 out of 8 are co-located in the same continent. The only exception is South America where Fastly has no site. Later in §5.3 we will present our measurement study that explores the impact of data center location on Periscope CDN performance.

Meerkat & Facebook Live. We also study the streaming protocols in Meerkat and Facebook Live by analyzing their app-to-server traffic. In Meerkat, each broadcaster uses a single HTTP POST connection to continuously upload live video to Meerkat server (hosted by Amazon EC2), while viewers download video chucks from the server using HLS. For Facebook live, the protocol setting is similar to that of Periscope. Each broadcaster uses RTMPS (RTMP over TLS/SSL) to upload live video via an encrypted connection, and viewers download video from Facebook's own CDNs using RTMPS or HLS. Later in §7, we will discuss the implications of using RTMPS to system security and performance.

[6]Fastly deployed 3 new data centers (Perth, Wellington and Sao Paolo) in December 2015, which are not covered by our measurement.

4.2 Breakdown of Broadcast Delays

Figure 10 illustrates the breakdown of Periscope's end-to-end delay, which is different for viewers who download video via Wowza (RTMP) and those who download from Fastly (HLS).

Wowza (RTMP). In this case the broadcast delay (per video frame) includes 3 parts:

- **Upload delay** is for transferring the video frame from the broadcaster to the Wowza server (②-①).
- **Last-mile delay** is for downloading the video frame from Wowza to the viewer's device (③-②).
- **Client-buffering delay** is the gap between the time that the video frame arrived at the viewer device to the time it got played (④-③).

Fastly (HLS). Here the process is more complicated. Wowza first assembles each video frame into chunks. When a new chunk is available (⑦), Wowza updates the chunklist and notifies Fastly to expire its old chunklist (⑧). As viewers poll the Fastly chunklist periodically, the first viewer polling (⑨) after the chunklist expiration triggers Fastly to poll Wowza (⑩) and copy the fresh chunk from Wowza (⑪) that serves future polling requests. With this process in mind, we divide each HLS viewer's broadcast delay (per video chuck) into 6 parts (Figure 10(b)):

- **Upload delay** is for the broadcaster uploading a video frame to Wowza (⑥ - ⑤).
- **Chunking delay** is the latency involved to form a chunk. It is equal to the chunk duration (⑦ - ⑥).
- **Wowza2Fastly delay** is the time gap between a fresh chunk is ready till the chunk arrives at the Fastly server (⑪ - ⑦). Since transferring a chunk from Wowza to Fastly is triggered by the first HLS viewer polling (⑨), this delay value also includes the contribution of the first viewer's polling delay.
- **Viewer polling delay**: Delay between Fastly getting a local cache and viewers polling cached chunklist from Fastly (⑭ - ⑪).
- **Last-mile delay** is between Fastly and the viewer's phone (⑫ - ⑪, ⑮ - ⑭).
- **Client-buffering delay** is the gap between the time of receiving the chunk and the time it got played (⑯ - ⑫ and ⑰ - ⑮).

4.3 Experimental Methodology

To measure each delay component, we conduct both fine-grained passive measurements by crawling a large number of real-world Periscope broadcasts, and controlled experiments by monitoring Periscope broadcasts initiated by ourselves. *First*, by recording detailed, per-frame (or chunk) events, the passive measurements allow us to accurately characterize the delay between broadcasters and Periscope CDN servers, covering the key processes that Periscope can control via its CDN design. *Second*, we use the controlled experiments to estimate the delay associated with the processes between viewers and Periscope CDN servers and those on viewer devices. These include the "last-mile" delay which depends heavily on the viewer's local network condition, and the latency of "client-buffering" and "polling" which depend heavily on the client app configuration.

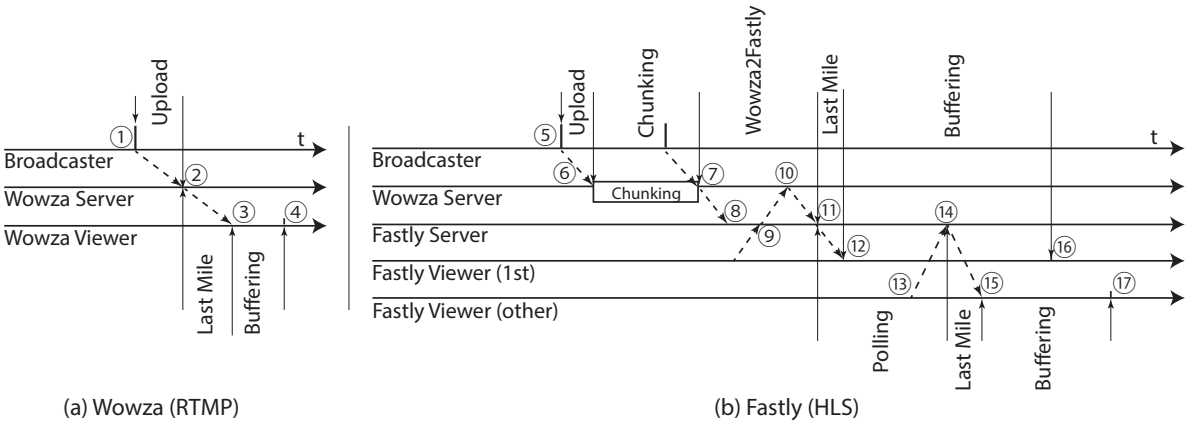
Passive Broadcast Crawling. Using our whitelisted servers, we randomly selected 16,013 broadcasts from all broadcasts between November 29, 2015 and February 5th, 2016. We carefully engineered our crawlers to capture fine-grained events within each broadcast. *First*, since Periscope provides both RTMP and HLS URLs to RTMP viewers, we designed our RTMP crawler to join immediately when a broadcast starts and obtained both URLs. *Second*, for each broadcast, our HLS crawler acted as an HLS viewer but polled Fastly at a very high frequency, immediately triggering chunk transfers between Wowza and Fastly. This way, our crawlers captured both RTMP and HLS delay performance for each broadcast regardless of its actual viewer count. *Finally*, since Wowza and Fastly have multiple datacenters, we deployed our crawlers on 8 nearby or co-located Amazon EC2 sites to collect data from all these datacenters.

More specifically, for each RTMP channel, our crawler established an RTMP connection with Wowza to download video streams. To accurately estimate ②, we set the stream buffer to 0 so that every video frame is pushed to our crawler as soon as it becomes available on Wowza. For each Wowza datacenter, we setup a dedicated crawler co-located at the same EC-2 site to minimize the delay. In addition, we obtained ① from the meta data of each video keyframe. This timestamp is recorded by the broadcaster's device and embedded into the keyframe. It may not always be a universal timestamp. Note that this error does not affect the delay measurement of Periscope's CDN, since all timestamps at server side are correctly synced.

For each HLS channel, our crawler constantly pulled the chunklist from Fastly. To accurately estimate Wowza2Fastly delay (⑪ - ⑦), the crawler used a very high polling frequency (once every 0.1 second) to minimize ⑨ - ⑦, and recorded the time when a chunk first becomes available at Fastly (⑪). Also, for each Fastly datacenter, we had a dedicated crawler at the nearest EC2 site. Similar to RTMP, we obtained ⑤ from the meta data in the video chuck. Note that the timestamp ⑥ was already obtained by the RTMP crawler (equivalent to ②).

Controlled Broadcast Experiments. We performed controlled experiments to estimate the rest of the delay components. We configured one smartphone as a broadcaster and the other two as a RTMP and a HLS viewer[7], respectively. All three phones were connected to the Internet via stable WiFi connections. During each broadcast, we collected the network traffic on both viewers' devices to extract two key timestamps ③, ⑬, and ⑮. At the same time, we

[7]We forced one smartphone to become a HLS viewer by intercepting and modifying its message exchange with Periscope, which contains both RTMP and HLS urls. We deleted the RTMP url manually, forcing the smartphone to connect to the HLS server.



(a) Wowza (RTMP) (b) Fastly (HLS)

Figure 10: RTMP/HLS end-to-end delay breakdown diagram.

used our crawlers to record the set of timestamps from the CDN (①, ②, and ⑤, ⑥, ⑦, ⑪). These measurements allowed us to estimate the last-mile delay for RTMP and HLS, and the polling delay for HLS.

Estimating the "buffering" delay requires knowledge of ④ and ⑰, the timestamp of video playing. This cannot be extracted from network traffic. Instead, we estimated this value by subtracting other delay components from the end-to-end delay. To record the end-to-end delay, we used the broadcaster's phone camera to videotape a running clock, and manually recorded the end-to-end delay based on the difference of displayed time on broadcaster's phone screen and the viewers'. The error is less than 1s. We also repeated each experiment 10 times and take the average delay to further reduce the measurement error.

5. DELAY ANALYSIS

In this section, we analyze the delay measurements of Periscope broadcasts to identify key contributors to the end-to-end video streaming delay. By comparing the performance of RTMP (which targets low latency delivery) and HLS (which offers higher scalability), we seek to understand the tradeoffs between latency and scalability.

5.1 Key Delay Contributors

Figure 11 provides a breakdown of the end-to-end broadcast delay captured by our controlled experiments. The RTMP delay is for each video frame, and the HLS delay represents the chunk-level delay (*i.e.* the frame-level delay of the first frame in each chunk). The delay value of each component is averaged across 10 repetitions. As expected, RTMP sessions experienced significantly less latency (1.4s) compared to HLS sessions (11.7s).

The key contributors to HLS' higher latency are client-buffering (6.9s), chunking (3s), polling (1.2s) and Wowza2Fastly (0.3s). Chunking and polling delays are direct results of the protocol difference between RTMP and HLS; Wowza2Fastly is caused by the handshaking and data transfer between Wowza and Fastly servers, a specific CDN design choice by Periscope. Finally, HLS' large buffering delay comes from the specific buffer configuration in the HLS viewer client, which is much more "aggressive" than that of RTMP[8] and will be used to mitigate heavy delay jitters.

Based on these observations, next we will study the chunking and polling delay in more detail, and explore how CDN geolocation affects Wowza2Fastly (and other delay components). Later in §6 we will explore approaches to reduce the buffering delay.

5.2 Comparing RTMP and HLS Latency

Targeting two different goals (supporting scalability vs. minimizing latency), RTMP and HLS protocols differ significantly in their design. The key design differences are 1) the operating data unit (frame vs. chunk) and 2) the communication model (push vs. poll). Next, we investigate how these choices affect latency and scalability.

Frame vs. Chunk. RTMP operates on individual video frames while HLS operates on chunks (a group of video frames). The chunk size (in terms of the total video duration) varies across broadcast sessions, but translates directly into the chunking delay. We extracted the chunk size distribution from our passive crawling of 16,013 broadcasts, and found that mass majority (>85.9%) of HLS broadcasts used 3s chunks (or 75 video frames of 40ms in length).

We believe that Periscope's choice of chunk size already reflects the tradeoff between scalability and latency. Using smaller chunks obviously reduces the chunking delay but also increases the number of chunks. This translates into higher server overhead for managing data and handling client polling (*i.e.* viewers will send more requests to servers). Thus to support a large number of users, HLS must configure its chunk size with care. As a reference, today's livestreaming services all use ≈3s chunks (3s for Periscope and Facebook live, 3.6s for Meerkat), while Apple's video-on-demand (VoD) HLS operates on 10s chunks.

Push vs. Poll. Another key difference between RTMP and HLS is that when stream videos to viewers, HLS uses poll-based operations while RTMP is push-based. That is, HLS viewers must poll Fastly servers periodically to dis-

[8]Our control experiments show that the HLS client uses a roughly 9s pre-fetching buffer size while RTMP uses 1s.

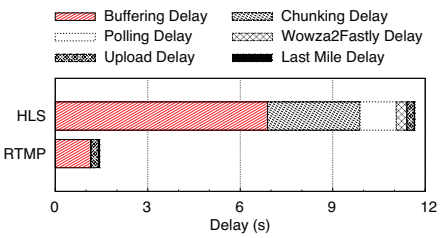


Figure 11: HLS/RTMP end-to-end delay breakdown.

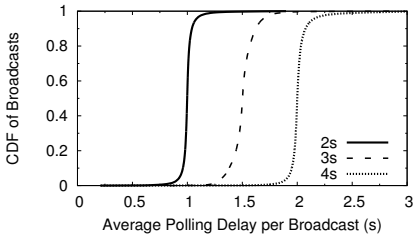


Figure 12: CDF of average polling delay with different polling intervals.

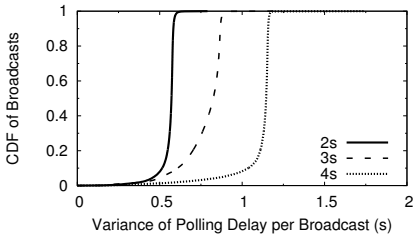


Figure 13: CDF of polling delay variance with different polling intervals.

cover new chunks. Furthermore, after a fresh chunk is available on Wowza, it is only transferred to Fastly when an HLS viewer polls Fastly. These extra overheads lead to a considerable amount of polling delay for HLS viewers.

To study the polling delay in detail, we use our passive measurements to perform trace-driven simulations[9]. Since our crawler polls the Fastly server at a very high frequency of once every 0.1s, it can immediately identify the arrival of new chunks on Fastly servers. Using the collected chunk arrival timestamps, we simulate periodic polling for each single HLS viewer of the broadcast[10], and study both the average and standard deviation of the polling delay within each broadcast. The results are shown in Figure 12 and 13.

We make two key observations. *First*, the average polling delay is large (1-2s) and increases with the polling interval. Using 2s and 4s polling intervals, the average delay is half of the polling interval. Yet using 3s interval, the average delay varies largely between 1s and 2s. This is because the chunk inter-arrival time varies slightly around 3s, thus polling at the same frequency creates large delay variations. *Second*, the STD results in Figure 13 show that the polling delay varies largely within each broadcast as viewers are unable to predict the chunk arrival time. Such high variance translates into delay jitters at the viewer device which are then handled by the client-buffering.

The choice of polling interval also reflects the tradeoff between scalability and latency. Smaller polling interval reduces average polling delay and variance, but comes at the cost of generating more polling requests to the CDN servers. Our controlled experiments revealed that in Periscope, the polling interval varies between 2s and 2.8s.

Scalability. We also compare the cost of supporting scalability when running RTMP and HLS. We set up a Wowza Stream Engine[11] on a laptop, serving as a CDN node. The laptop has 8GB memory, 2.4GHz Intel Core i7 CPU, and 1Gbps bandwidth. We start RTMP and HLS viewers (separately) on other machines connecting to the laptop via Ethernet, and measure the laptop's resource usage including CPU and memory as we vary the number of viewers. Our results show that RTMP and HLS result in similar and stable memory consumption but very different CPU usage with respect to different numbers of viewers. Specifically, Figure 14 shows that supporting RTMP users requires much higher CPU power than that for HLS. The gap between the two elevates with the number of viewers, demonstrating the higher cost of supporting RTMP scalability. The key reason for RTMP's higher CPU usage is that it operates on small frames instead of large chunks, leading to significantly higher processing overhead.

5.3 Impact of CDN Geolocation

We now examine the impact of CDN geolocation on Periscope performance. By studying our detailed broadcast measurements, we seek to understand how Wowza and Fastly connect Periscope users to their data centers and how such assignment affects end-to-end delay. Furthermore, using our knowledge on the Wowza and Fastly data center locations, we examine the chunk transfer delay between their data centers (*i.e.* the Wowza2Fastly delay).

Periscope's Geolocation Optimization. We discovered three types of geolocation optimization that Periscope uses to improve scalability and latency. *First*, Periscope connects each broadcaster to the *nearest* Wowza data center[12], which will effectively reduce the upload delay. *Second*, Fastly fetches video chunks from Wowza and copies them to *multiple* Fastly data centers, which helps to provide scalability. *Finally*, each HLS viewer connects to the *nearest* Fastly data center using IP anycast, thus minimizing the last mile delay. In contrast, RTMP viewers always connect to the same Wowza data center that the broadcaster connects to, avoiding data transfer among Wowza data centers.

Wowza to Fastly Transmission. The CDN geolocation also directly affects the Wowza-to-Fastly delay, *i.e.* the latency for Wowza to distribute video chunks to different Fastly data centers. We estimate this latency from our crawled

[9]As discussed earlier, our crawl of real-world broadcasts captures the delay from broadcasters to CDNs but not those from CDNs to viewers.

[10]Here we assume that the per-viewer polling delay does not depend on the number of viewers. That is, the CDN servers have sufficient resources to handle all polling requests. We also assume the communication delay between the viewer and CDN is negligible compared to the polling delay.

[11]https://www.wowza.com/products/streaming-engine

[12]More than half of our crawled broadcasts contain the GPS coordinates of the broadcaster. Thus we directly compare each broadcaster's location to her associated Wowza data center location.

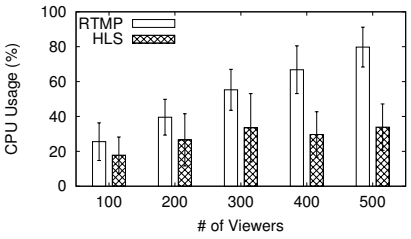


Figure 14: CPU usage of server using RTMP and HLS with different number of viewers.

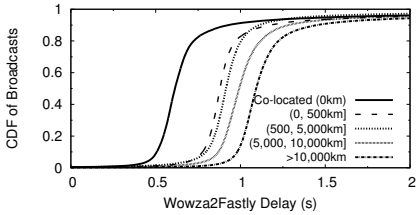


Figure 15: Wowza-to-Fastly Delay. We group datacenter pairs based on their geographic distance. Co-located pairs are located at the same city.

broadcast traces[13]. Figure 15 plots the CDF of the Wowza2Fastly delay across broadcasts, where we group Wowza and Fastly data center pairs based on their geographical distance (distance between the cities). Here co-located means that the two data centers are in the same city. As expected, pairs with longer distances experience larger delay.

Figure 15 also shows a clear distinction between co-located pairs and all other pairs. In particular, there is a significant gap (>0.25s) between the latency of co-located pairs and that of nearby city pairs (<500km), *e.g.*, San Francisco to Los Angeles. We suspect that this is because each Wowza server first transfers video chunks to the co-located Fastly data center, who then behaves as a gateway and distributes the data to other Fastly data centers. The extra delay is likely the result of coordination between this gateway and other data centers.

6. OPTIMIZING CLIENT-SIDE BUFFERING

Our delay analysis has shown that client-side buffering introduces significant delay to both RTMP and HLS viewers. The resulting video playback delay depends on the broadcast channel quality, buffering strategies and configurations at the mobile app. While buffering is necessary to ensure smooth video replay, setting reasonable parameters for the buffer can dramatically impact streaming delay. In this section, we perform trace-driven experiments to understand if (and by how much) Periscope can reduce this buffering delay, and the corresponding impact on end-user video quality.

Buffering Strategy. We start by describing the buffering strategy. For HLS, we were able to decompile Periscope's Android source code and analyze its media player settings. We found that Periscope users a naive buffering strategy. Specifically, when the live streaming starts, the client (viewer) first pre-buffers some video content (P seconds of video) in its buffer. During live streaming playback, the client inserts newly arrived video content into the buffer, which are organized and played by their sequence numbers to mitigate delay jitter and out-of-order delivery. Arrivals that come later than their scheduled play time are discarded. This smoothes out video playback but also introduces extra latency. We found from the source code that Periscope configures a sufficiently large memory to buffer video content and avoid dropping packets.

We were unable to identify the exact buffering setting for RTMP, because Periscope implements a customized RTMP protocol stack, and the source code is obfuscated. But results from our controlled experiments suggest that Periscope uses the same buffering strategy for RTMP and HLS.

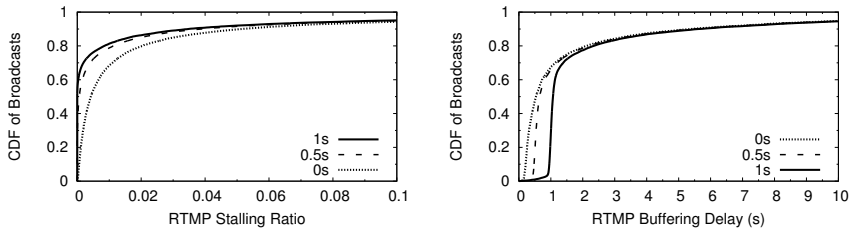
Next, we implement the above buffering strategy in trace-driven simulations for both RTMP and HLS clients. Our goal is to understand how Periscope's buffering parameters impact playback delay, and whether it can be optimized for better performance. For simplicity, we do not put any hard constraint on the physical buffer size, but vary the pre-buffer size (P) to study its impact.

Experiments. We perform trace-driven simulations using our measurements on our 16,013 real-world broadcasts. For each broadcast, we extract from our measurements a sequence of video frame/chunk arrival times (at the Wowza/Fastly server). This covers the path from the broadcaster to the CDN. We then simulate the path from the CDN to the viewers. For RTMP viewers, we implement client-side buffering, and for HLS viewers, we implement polling (at an interval of 2.8s) and buffering. While we cannot capture possible delay variances from last mile links, we assume it is under one second in our simulations.

We evaluate the viewer experience via two metrics. The first is the video play smoothness, represented by the *stalling ratio*, *i.e.* the duration of stalling moment (no video to play) divided by the duration of the broadcast. The second metric is the *buffering delay*, the time gap between a video frame/chunk arriving at the viewer till the time it got played. For each broadcast, we compute the average buffering delay across all the frames/chunks.

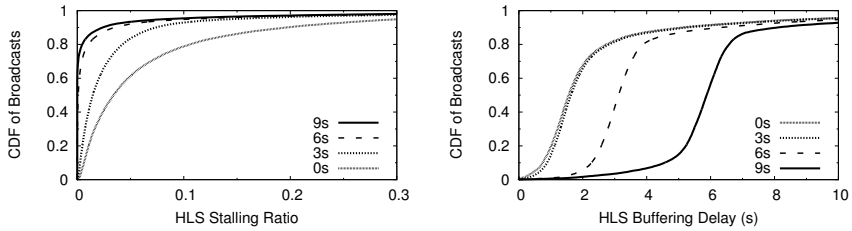
Figure 16(a)-(b) plot the stalling ratio and average buffering delay for RTMP users by varying P from 0s to 1s. As expected, pre-buffering more video content (*i.e.* larger P) increases the smoothness of video playing but also incurs

[13]Our HLS crawler (as a viewer) polls each Fastly server at a very high frequency of once per 0.1s, and thus can record the earliest time that a chunklist is updated at each Fastly server. Furthermore, operating at a polling frequency 20+ times faster than normal HLS viewers, our crawler will initiate the first poll on the Fastly server that triggers the server to poll Wowza and obtain a fresh chunklist. Thus we are able to estimate the Wowza2Fastly delay ⑪-⑨ (Figure 10) by measuring ⑪-⑦ and minimizing ⑨-⑦.



(a) Stalling Time Ratio (b) Buffering Delay

Figure 16: RTMP: the impact of different buffer size (for pre-download) to buffering delay and stalling.



(a) Stalling Time Ratio (b) Buffering Delay

Figure 17: HLS: the impact of different buffer sizes (for pre-download) to buffering delay and video stalling.

(slightly) higher buffering delay. Here the benefit of pre-buffering is not significant because RTMP streaming is already smooth. We also observe that in Figure 16(b), a small portion (10%) of broadcasts experience long buffering delay (>5s). This is caused by the bursty arrival of video frames during uploading from the broadcaster.

HLS viewers, on the other hand, experience much more stalling due to high variance in polling latency, and the system needs to pre-buffer much more video content (6-9s) to smooth out playback (Figure 17(a)). From analyzing our controlled experiment traces, we find that Periscope configures P=9s for HLS viewers. However, Figure 17 (a)-(b) show that a much less conservative value of P=6s already provides similar results on stalling, but reduces buffering delay by 50% (3s). These results suggest that Periscope's buffering configuration is conservative, and can be optimized today to significantly reduce buffering delay. Over time, we expect that buffering will increase as higher volume of broadcasts force servers to increase chunk size and decrease polling frequency.

Although we do not consider last-mile delay in our experiments, our result stands as long as the last mile delay is stable. In cases when viewers have stable last-mile connection, *e.g.,* good WiFi/LTE, smaller buffer size could be applied to reduce the buffering delay. In other cases of bad connection, Periscope could always fall back to the default 9s buffer to provide smooth playback.

7. SECURING LIVESTREAM BROADCASTS

During the course of our investigation into livestreaming services, we discovered that Periscope and Meerkat both shared a common security vulnerability. Neither service authenticated video streams after the initial connection setup, and it is not difficult for attackers to silently "alter" part or all of an ongoing broadcast for their own purposes. More specifically, the attacker can overwrite selected portions of an ongoing broadcast (video or audio or both), by tapping into the transmission at the source (in the broadcaster's network) or at the last-mile network of one or more viewers. In this section, we describe this vulnerability in the Periscope protocol, the results of our proof-of-concept experiments conducted on our own streams, and simple countermeasures to defend against this attack.

Ethics. Our primary concern was to the security of livestream broadcasters on these services. With this attack, any video stream could be altered in undetectable ways. For example, a broadcast coming from the White House can be altered in an unsecured edge wireless network in a foreign country, so that a selected group of users would see an altered version of the stream while any alterations remained invisible to the broadcaster.

Validating the vulnerability required that we perform experiments to alter a Periscope stream at both the broadcaster's local network and the viewer's network. We took all possible precautions to ensure our tests did not affect any other users. First, we conducted the proof-of-concept experiments on broadcasts created by ourselves, set up so that they were only accessible to our own viewer. All involved parties (attacker and victim) were our own Periscope accounts. Second, our Periscope accounts have no followers and thus our experiments did not push notifications to other users. Third, we made small changes that should have zero impact on any server side operations. Finally, once we confirmed the vulnerability, we notified both Periscope and Meerkat about this vulnerability and our proposed countermeasure (directly via

phone to their respective CEOs). We also promised any disclosures of the attack would be delayed for months to ensure they had sufficient time to implement and deploy a fix.

7.1 Validating the Attack

The broadcast tampering attack is possible because Periscope uses *unencrypted* and *unauthenticated* connections for video transmission. As shown in Figure 8(a), when a user starts a broadcast, she first obtains a unique broadcast token from a Periscope server via a HTTPS connection. Next, she sends the broadcast token to Wowza and sets up a RTMP connection to upload video frames. This second step introduces two critical issues: (1) the broadcast token is sent to Wowza via RTMP in plaintext; (2) the RTMP video itself is unencrypted. As a result, an attacker can easily launch a man-in-the-middle attack to hijack and modify the video content.

Tampering on Broadcaster Side. An attacker in the same edge network as the broadcaster can alter the stream before it reaches the upload server. This is a common scenario when users connect to public WiFi networks at work, coffee shop or airport. To launch the attack, the attacker simply connects to the same WiFi network and sniffs the victim's traffic. There is no need to take control of the WiFi access point.

Specifically, the attacker first performs a simple ARP spoofing attack to redirect the victim's traffic to the attacker[14]. The attacker then parses the unencrypted RTMP packet, replaces the video frame with arbitrary content, and uploads modified video frames to Wowza servers, which are then broadcast to all viewers. This attack can commence anytime during the broadcast, and is not noticeable by the victim because her phone will only display the original video captured by the camera.

Tampering at the Viewer Network. An attacker can also selectively tamper with the broadcast to affect only a specific group of viewers, by connecting to the viewers' WiFi network. When the viewer (the victim) downloads video content via WiFi, the attacker can modify the RTMP packets or HLS chunks using a similar approach. The broadcaster remains unaware of the attack.

Experimental Validation. We performed proof-of-concept experiments to validate both attack models. Since both attacks are similar, for brevity we only describe the experiment to tamper at the broadcaster. We set up a victim broadcaster as a smartphone connected to a WiFi network, an attacker as a laptop connected to the same WiFi, and a viewer as another smartphone connected via cellular. The broadcaster begins a broadcast and points the camera to a running stopwatch (to demonstrate the "liveness" of the broadcast). When the attack commences, the attacker replaces the video content with a black screen.

[14] ARP spoofing is a known technique to spoof another host in a local area network. This is done by sending out falsified ARP (Address Resolution Protocol) messages to manipulate the mapping between IP and MAC address in the local network.

Figure 18: Screenshots of the broadcaster and viewer before and after the attack. After the attack, the viewer sees a black screen (tampered), while the broadcaster sees the original video.

The attacker runs ARP spoofing to perform the man-in-the-middle attack (using the Ettercap library), and replaces video frames in the RTMP packet with the desired frames. Our proof of concept uses simple black frames. We wrote our own RTMP parser to decode the original packet and make the replacements. Finally, we open the broadcast as viewer from another phone to examine the attack impact. The viewer does not need connect to this WiFi network.

Figure 18 shows the screenshot results of the broadcaster and the viewer before and after the attack. Following the attack, the viewer's screen turns into a black frame while the broadcaster sees no change. In practice, comments from viewers may alert the broadcaster to the tampered stream, but by then the damage is likely done.

7.2 Defense

The most straightforward defense is to replace RTMP with RTMPS, which performs full TLS/SSL encryption (this is the approach chosen by Facebook Live). Yet encrypting video streams in real time is computationally costly, especially as smartphone apps with limited computation and energy resources. Thus for scalability, Periscope uses RTMP/HLS for all public broadcasts and only uses RTMPS for private broadcasts.

Another simple countermeasure would protect (video) data integrity by embedding a simple periodic signature into the video stream. After a broadcaster obtains a broadcast token from the Periscope server (via HTTPS), she connects to Wowza using this broadcast token and securely exchanges a private-public key pair (TLS/SSL) with the server. When uploading video to Wowza (using RTMP), the broadcaster signs a secure one-way hash of each frame, and embeds the signature into the metadata. The Wowza server verifies the signatures to validate video frames have not been modified. To mitigate viewer-side attacks, Wowza can securely forward the broadcaster's public key to each viewer, and they can verify the integrity of the video stream. Our solution is simple and lightweight, and we can further reduce overhead by signing only selective frames or signing hashes across multiple frames.

We reported this attack and countermeasure to the management teams at both Periscope and Meerkat in September

2015. To the best of our knowledge, Periscope is taking active steps to mitigate this threat.

8. DISCUSSION AND CONCLUSION

Our work shows a clear tension between scalability and delivery delay on today's personalized livestreams services. We also recognize that these systems are quite young in their development, and ongoing engineering efforts can significantly reduce per-broadcast overheads and improve scalability. Our results suggest, however, that services like Periscope are already limiting user interactions to ensure minimal lag between the audience and broadcaster. Moving forward, these services will have to make a difficult decision between maintaining hard limits on user interactivity (limiting comments to the first 100 users connected to RTMP servers), or addressing issues of scale in order to support more inclusive and richer modes of audience interaction.

One potential alternative is to build a dramatically different delivery infrastructure for interactive livestreams. To avoid the costs of managing persistent connections to each viewer, we can leverage a hierarchy of geographically clustered forwarding servers. To access a broadcast, a viewer would forward a request through their local leaf server and up the hierarchy, setting up a reverse forwarding path in the process. Once built, the forwarding path can efficiently forward video frames without per-viewer state or periodic polling. The result is effectively a receiver-driven overlay multicast tree (similar to Scribe [12] and Akamai's streaming CDN [34, 23]) layered on top of a collection of CDN or forwarding servers. We note that Akamai's CDN is focused on scalability, and uses a two-layer multicast tree that focuses on optimizing the transmission path from broadcaster to receiver [23]. Since its audience does not directly interact with the broadcaster, streams do not need to support real-time interactions.

Moving forward, we believe user-generated livestreams will continue to gain popularity as the next generation of user-generated content. Novel methods of interaction between broadcasters and their audience will be a differentiating factor between competing services, and issues of scalable, low-latency video delivery must be addressed.

Finally, following consultations with the Periscope team, we will make parts of our measurement datasets available to the research community at http://sandlab.cs.ucsb.edu/periscope/.

Acknowledgments

The authors wish to thank the anonymous reviewers and our shepherd Fabian Bustamante for their helpful comments. This project was supported by NSF grants CNS-1527939 and IIS-1321083. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

9. REFERENCES

[1] Accessing Fastly's IP ranges. https://docs.fastly.com/guides/securing-communications/accessing-fastlys-ip-ranges.

[2] Adobe RTMP specification. http://www.adobe.com/devnet/rtmp.html.

[3] Apple HLS specification. https://developer.apple.com/streaming/.

[4] Fastly. https://www.fastly.com/.

[5] Fastly network map. https://www.fastly.com/network.

[6] Huang, C., Wang, A., Li, J., Ross, K. W. Measuring and evaluating large-scale CDNs. http://dl.acm.org/citation.cfm?id=1455517 (2008).

[7] Periscope - live streaming with your gopro. https://gopro.com/help/articles/Block/Periscope-Live-Streaming-with-your-GoPro.

[8] Wowza stream engine. https://www.wowza.com/products/streaming-engine.

[9] Adhikari, V., Guo, Y., Hao, F., Hilt, V., and Zhang, Z.-L. A tale of three cdns: An active measurement study of hulu and its cdns. In *INFOCOM Workshops* (2012).

[10] Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *Proc. of INFOCOM* (2012).

[11] Bouzakaria, N., Concolato, C., and Le Feuvre, J. Overhead and performance of low latency live streaming using mpeg-dash. In *Proc. of IISA* (2014).

[12] Castro, M., et al. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC 20*, 8 (2002).

[13] Constine, J. Twitter confirms periscope acquisition, and here's how the livestreaming app works. TechCrunch, March 2015.

[14] Cresci, E., and Halliday, J. How a puddle in newcastle became a national talking point. The Guardian, January 2016.

[15] Dredge, S. Twitter's periscope video app has signed up 10m people in four months. The Guardian, August 2015.

[16] Hamilton, W. A., Garretson, O., and Kerne, A. Streaming on twitch: fostering participatory communities of play within live mixed media. In *Proc. of CHI* (2014), ACM.

[17] Hei, X., Liang, C., Liang, J., Liu, Y., and Ross, K. W. A measurement study of a large-scale p2p iptv system. *IEEE Transactions on Multimedia 9*, 8 (2007).

[18] Huang, T.-Y., Johari, R., McKeown, N., Trunnell, M., and Watson, M. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proc. of SIGCOMM* (2014).

[19] Jackson, R. How to avoid periscopes broadcast too full message. Phandroid Blog, August 2015.

[20] Jill, J. 'broadcast is too full'? how to share your periscope comments. Scope Tips Blog, October 2015.

[21] Kaytoue, M., Silva, A., Cerf, L., Meira Jr, W., and Raïssi, C. Watch me playing, i am a professional: a first study on video game live streaming. In *MSND@WWW* (2012).

[22] Khan, A. Broadcast too full & you can't comment? here are 3 ways to get your message out anyway. Personal Blog, August 2015.

[23] Kontothanassis, L., Sitaraman, R., Wein, J., Hong, D., Kleinberg, R., Mancuso, B., Shaw, D., and Stodolsky, D. A transport layer for live streaming in a content delivery network. *Proc. of the IEEE 92*, 9 (2004).

[24] Krishnan, R., Madhyastha, H. V., Srinivasan, S., Jain, S., Krishnamurthy, A., Anderson, T., and Gao, J. Moving beyond end-to-end path information to optimize CDN performance. In *Proc. of SIGCOMM* (2009).

[25] Kupka, T., Griwodz, C., Halvorsen, P., Johansen, D., and Hovden, T. Analysis of a real-world http segment streaming case. In *Proc. of EuroITV* (2013).

[26] Laine, S., and Hakala, I. H.264 qos and application performance with different streaming protocols. In *MobiMedia* (2015).

[27] Lederer, S., Müller, C., and Timmerer, C. Dynamic adaptive streaming over http dataset. In *Proc. of MMSys* (2012).

[28] Li, Y., Zhang, Y., and Yuan, R. Measurement and analysis of a large scale commercial mobile internet tv system. In *Proc. of IMC* (2011).

[29] Lohmar, T., Einarsson, T., Fröjdh, P., Gabin, F., and Kampmann, M. Dynamic adaptive http streaming of live content. In *Proc. of WoWMoM* (2011).

[30] Madrigal, A. C. The interesting problem with periscope and meerkat. Fusion, March 2015.

[31] Magharei, N., and Rejaie, R. Prime: Peer-to-peer receiver-driven mesh-based streaming. *IEEE/ACM TON 17*, 4 (2009), 1052–1065.

[32] Mediati, N. Twitter cuts off meerkat, won't let it import who you follow on twitter. PCWorld, March 2015.

[33] Müller, C., Lederer, S., and Timmerer, C. An evaluation of dynamic adaptive streaming over http in vehicular environments. In *Proc. of MoVid* (2012).

[34] Nygren, E., Sitaraman, R. K., and Sun, J. The akamai network: a platform for high-performance internet applications. *SIGOPS OSR 44*, 3 (2010).

[35] Perez, S. Live streaming app periscope touts 200 million broadcasts in its first year. TechCrunch, March 2016.

[36] Poblete, B., Garcia, R., Mendoza, M., and Jaimes, A. Do all birds tweet the same?: characterizing twitter around the world. In *Proc. of CIKM* (2011).

[37] Pramuk, J. Periscope ceo: How we're growing live-streaming. CNBC, December 2015.

[38] Pullen, J. P. You asked: What is the meerkat app? Time, March 2015.

[39] Siekkinen, M., Masala, E., and Kämäräinen, T. Anatomy of a mobile live streaming service: the case of periscope. In *Proc. of IMC* (2016).

[40] Silverston, T., and Fourmaux, O. Measuring p2p iptv systems. In *Proc. of NOSSDAV* (2007).

[41] Small, T., Liang, B., and Li, B. Scaling laws and tradeoffs in peer-to-peer live multimedia streaming. In *Proc. of MM* (2006).

[42] Sripanidkulchai, K., Ganjam, A., Maggs, B., and Zhang, H. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In *Proc. of SIGCOMM* (2004).

[43] Sripanidkulchai, K., Maggs, B., and Zhang, H. An analysis of live streaming workloads on the internet. In *Proc. of IMC* (2004).

[44] Su, A.-J., Choffnes, D. R., Kuzmanovic, A., and Bustamante, F. E. Drafting behind akamai (travelocity-based detouring). In *Proc. of SIGCOMM* (2006).

[45] Tang, J. C., Venolia, G., and Inkpen, K. M. Meerkat and periscope: I stream, you stream, apps stream for live streams. In *Proc. of CHI* (2016).

[46] Wilson, C., Boe, B., Sala, A., Puttaswamy, K. P. N., and Zhao, B. Y. User interactions in social networks and their implications. In *Proc. of EuroSys* (2009).

[47] Yin, X., Jindal, A., Sekar, V., and Sinopoli, B. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proc. of SIGCOMM* (2015).

[48] Zhang, C., and Liu, J. On crowdsourced interactive live streaming: a twitch. tv-based measurement study. In *Proc. of NOSSDAV* (2015).

[49] Zhang, X., Liu, J., Li, B., and Yum, T.-S. P. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *Proc. of INFOCOM* (2005).

[50] Zhao, X., Sala, A., Wilson, C., Wang, X., Gaito, S., Zheng, H., and Zhao, B. Y. Multi-scale dynamics in a massive online social network. In *Proc. of IMC* (2012), pp. 171–184.