LiveJack: Integrating CDNs and Edge Clouds for Live Content Broadcasting

Bo Yan[†*], Shu Shi[†], Yong Liu[*], Weizhe Yuan[*], Haoqin He[*],
Rittwik Jana[†], Yang Xu[*], H. Jonathan Chao[*]

New York University[*]
{boven.yan, yongliu, wy666, haoqin.he, yang, chao}@nyu.edu

AT&T Labs Research[†]
{shushi, rjana}@research.att.com

ABSTRACT

Emerging commercial live content broadcasting platforms are facing great challenges to accommodate large scale dynamic viewer populations. Existing solutions constantly suffer from balancing the cost of deploying at the edge close to the viewers and the quality of content delivery. We propose LiveJack, a novel network service to allow CDN servers to seamlessly leverage ISP edge cloud resources. LiveJack can elastically scale the serving capacity of CDN servers by integrating Virtual Media Functions (VMF) in the edge cloud to accommodate flash crowds for very popular contents. LiveJack introduces minor application layer changes for streaming service providers and is completely transparent to end users. We have prototyped LiveJack in both LAN and WAN environments. Evaluations demonstrate that LiveJack can increase CDN server capacity by more than six times, and can effectively accommodate highly dynamic workloads with an improved service quality.

1 INTRODUCTION

Modern content delivery systems for live broadcasting are facing unprecedented challenges. On the one hand, more traditional TV programs, such as nightly news and sports games, are now streamed online at a higher quality. Popular programs can easily attract millions of viewers [10, 28]. On the other hand, the emerging User-Generated Live Content (UGLC) are gaining tremendous popularity through various streaming platforms (such as Twitch, Facebook Live, and Youtube Live, etc.) and at the same time bringing new challenges. Any random UGLC may suddenly become viral on social media as the result of social cascading and recommender promotion, and cause a *flash crowd* of viewers to watch the same content within a few minutes (see Section 2.1 for details). Without knowledge of geographic and network distributions of the viewers, it is difficult to provision streaming resources to accommodate such unpredictable flash crowds beforehand. Moreover, many live streaming platforms encourage interactions between content generators and viewers. For instance, Twitch offers viewers a chat box to send feedbacks to the broadcasters, while Facebook Live enables viewers to click emoji buttons while watching a broadcast. Such interactive features require UGLC streaming to have the *minimized playback lag*.

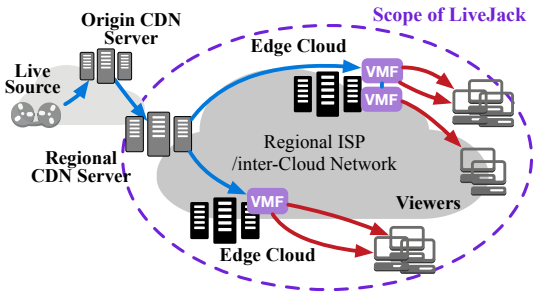


Figure 1: LiveJack Scope

Traditional content delivery network (CDN)-based live broadcasting systems are incapable of meeting all the new demands. CDN providers aggregate to one or multiple data centers to take advantage of the elasticity of Virtual Machine (VM) resources and the flexibility of routing inside data centers. However, the lack of edge presence makes the streaming vulnerable to long transmission delay and congestion fluctuations in Wide Area Networks (WAN). Only large content and CDN providers can afford to negotiate with ISPs to place compute resources at the ISP edge to improve QoE for regional customers [6, 15]. Nevertheless, due to the reduced degree of multiplexing at the network edge, provisioning enough edge presence to meet performance guarantee can be very cost-inefficient [22]. Measurement studies [29, 36, 40] have revealed that the streaming experience of leading live streaming platforms like YouTube and Twitch can suffer from occasional service interruption and unstable video quality.

Recently, major ISPs and cloud vendors have been investing heavily on developing integrated edge clouds. These edge clouds are deployed close to users and can provision virtual edge resources elastically from a centralized platform. For instance, AT&T has deployed 105 edge clouds at their provider edge sites in 2016 [1]. Cloud vendors and CDNs have proposed various methods to enhance the coordination between their data centers and edge resources [16, 24, 27]. This motivates us to rethink the design of live content delivery. Can ISP provide virtualized edge resources to assist CDNs to better serve flash crowds?

In this paper, we propose LiveJack, a transparent network service that enables CDN to seamlessly integrate edge clouds for live broadcast (Figure 1). The main idea is to elastically scale the serving capacity of CDN servers by dynamically deploying Virtual Media

MM'17, October 23–27, 2017, Mountain View, CA, USA.
 ISBN 978-1-4503-4906-2/17/10…$15.00
DOI: https://doi.org/10.1145/3123266.3123283

Functions (VMFs) at edge clouds. Compared with other live broadcasting solutions, LiveJack offers the following distinct features.

Centralized Infrastructure Scaling. Instead of pre-provisioning VMFs, LiveJack has a centralized controller to deploy VMFs on demand and optimize delivery paths in real-time with the global view of content sources, viewer groups, network, and cloud resources.

On-the-fly Session Migration. LiveJack employs layer-4 session hijacking techniques to transparently migrate streaming sessions to VMFs on-the-fly. Upon arrival, a new user can be immediately served by a CDN server and later seamlessly migrated to retrieve contents from a VMF. On-the-fly session migration enables all user sessions to be flexibly moved around to achieve better load balancing and VMF consolidation, which significantly improves LiveJack's response to flash crowds.

Dynamic Service Chaining. Recursive layer-4 session hijacking also enables service chaining of VMFs: a VMF can act as an end user and retrieve contents from another VMF. Dynamic multi-hop service chaining enables LiveJack to scale fast while maintaining efficient delivery paths among VMFs when facing a flash crowd.

Transparency and Compatibility. LiveJack is a layer-4 service and can support any layer-7 streaming applications. Applying LiveJack requires little modification on the server side and no modification on the client side. LiveJack is also compatible with any CDN optimization techniques.

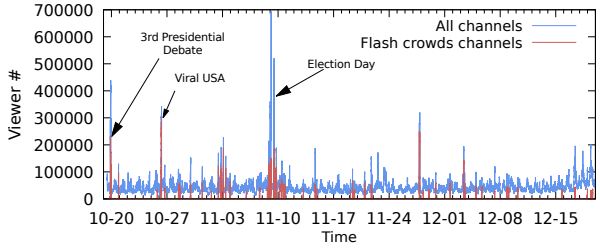
The **contributions** of this paper are four-fold: 1) We characterize large scale user traffic dynamics in commercial live streaming services (§2.1). 2) We introduce LiveJack, a novel network service that integrates CDNs and edge clouds for elastic and agile live content broadcasts (§3). 3) We implement VMF as a general Linux kernel component and develop a light-weight signaling protocol for streaming synchronization (§4). 4) We test the LiveJack prototype system in both LAN and WAN environments. Experimental results show great potential of LiveJack for commercial deployments (§5).

2 MOTIVATION AND RELATED WORK

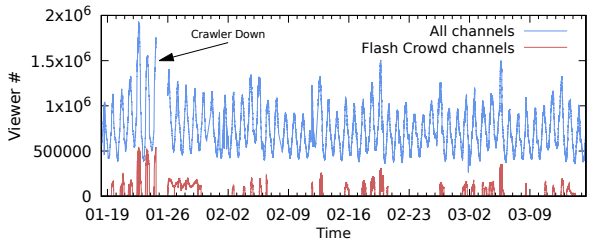
2.1 User Dynamics in Emerging Commercial Live Broadcasting Services

We have conducted a trace-driven study to profile the user dynamics of Facebook-Live and Twitch, both of which provide APIs to collect the viewer and channel statistics. We have collected traces consecutively from both Facebook-Live and Twitch for more than two months. Among all the channels, we identify the flash crowd channels based on two criteria: 1) the peak number of concurrent viewers; and 2) the peak viewer growth rate within a certain time window. In other words, a flash crowd channel is a mega-event [27, 40] with massive user arrivals in a short time period. For Facebook-live, we classify flash crowd channels as those with a peak viewer number of more than 20k and at least 10k viewer growth within 20min. For Twitch, flash crowd channels are defined as those with a peak viewer number of more than 100k and at least 50k viewer growth within 20min. The thresholds are determined by classifying the peak growth rates of all the channels. We do not rule out other threshold settings for data analysis, which lead to similar observations without loss of generality.

First, we analyze the evolution of viewer population in Figure 2a and Figure 2b for Facebook-live and Twitch, respectively. The

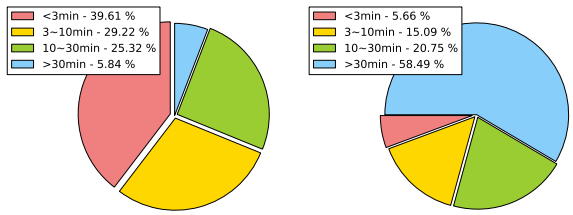


(a) Facebook-Live (Oct. 2016 - Dec.2016)



(b) Twitch (Jan. 2017 - Mar.2017)

Figure 2: Total viewer time series and flash crowd channels



(a) FB Live, 20k user growth (b) Twitch, 100k user growth

Figure 3: Percentage of flash-crowd channels with a certain user growth during 3 min/10 min/30 min

blue line depicts the total viewers in the system while the red line shows the viewers for the flash crowd channel with most viewers emerging at different time instants. The results reveal several interesting observations: first, for both systems, flash crowd channels significantly increase the total number of viewers in the system. Compared with the peak number of viewers in an average day without flash crowds, the flash crowds at 2016 Election Day increased the peak number by approximately ten times. When a flash crowd emerges, it tends to attract a lot more viewers than the average channels: for Facebook-live, the flash crowd channel by Viral USA attracted 86.4% of viewers of the entire system; for Twitch, the flash crowd channel for eSport attracts around 30% of the viewers of the system. Secondly, flash crowds can be relatively frequent. During our trace collection window, we have captured 154 flash crowd events for Facebook-live and 53 for Twitch with the aforementioned criteria. Thirdly, the events associated with the flash crowds can be either scheduled or random. We analyze the contextual information of the captured flash crowds. Besides the scheduled events, such as presidential elections and game competitions, we also noticed some sporadic accidental flash crowd events. For instance, on Oct. 26, 2016, a fake viral post of "Live NASA Space Walk" in Facebook-live by Viral USA accidentally generated a major flash crowd for the system. Even for the scheduled events, the actual viewer population evolution of flash crowd channels could differ from each other significantly.

Next we characterize how fast viewers enter flash crowd channels. Figure 3 shows the minimum time taken to reach a certain user growth for different channels in Facebook-Live and Twitch. In both platforms, the viewer population of some channels can reach a very high number within a very short period. For instance, 39.61% of the Facebook-Live channels can gather at least 20k viewers within three minutes. For Twitch, over 40% channels can grow more than 100k viewers within 30 minutes. By contrast, provisioning new resources (e.g. booting up virtual machines, changes routing and populate contents) can easily take longer than 10min [9].

2.2 Evolution of Live Content Broadcast

Much effort has been made to deliver live broadcast services over the past decade. In the early days, live content broadcast was built over IP multicast systems. However, as IP multicast faces practical deployment and management issues, it is only used in limited scenarios such as ISP-oriented IPTV services [11, 13]. During the early 2000, P2P-based live broadcasting systems won popularity to share video contents among end-user devices [8]. P2P live streaming system scales well under dynamic workloads, since each joining user acts as a reflected content source. However, prior research reported that P2P live streaming suffers from unstable video quality and severe playback lags up to minutes mainly due to peer churn and limited uplink bandwidth [25]. In addition, P2P systems introduce significant user-side complexities and no longer fit the modern requirment of lightweight client implementation. Even though recent effort in augmenting P2P with cloud and CDN (e.g., AngelCast [35]), LiveSky [39]) can effectively reduce the latency and improve the streaming quality, the P2P nature makes it difficult for these systems to attract users preferring to watch live streams in a browser or on mobile devices.

Today, most live broadcasting systems rely on CDN-based architectures to deliver live streams globally. By relaying the live content over one or multiple reflector sites, an overlay multicast tree is created to pass data from the origin content server to the edge servers in the regional data centers, which serve viewers directly through either HTTP or RTMP protocols [3, 22, 27]. Since the capacity and the egress bandwidth of each regional server can be very limited [36], most commercial streaming systems rely on the elasticity of the data center to handle varying traffic [31]. Some systems can provision just enough server resources to handle daily traffic and rely on the assistance of other CDN providers in the case of overflow, while others have to prepare for the worst case to maintain consistent experience for all viewers [2]. Prior research studies have also proposed various solutions to improve CDN performance for live broadcasting. For instance, VDN [27] developed a centralized streaming optimization and a hybrid control plane to reduce the start-up latency and improve routing choices across different CDN clusters. Footprint [24] shows the benefits of delivering streaming services by jointly optimizing the data center to provide the service, the WAN transport connectivity and the proxy selection. C3 proposes to improve video quality by helping clients to select better CDN sites through data-driven analysis[17].

Although the design philosophy and some technical details of LiveJack shares similarities with these related work, there are key differences. The scope of LiveJack is not to extend or optimize CDN, but facilitate the collaboration between CDN and ISP. Such CDN-ISP collaboration only exists nowadays for large content and CDN providers who are capable of deploying customized server hardware to the ISP edge [6, 15, 22]. In academia, NetPaaS [16] proposes to share ISP information with CDN to optimize user-to-server assignments and server allocations. Different from these infrastructure-sharing and information sharing approaches, LiveJack demonstrates a new way of collaboration: providing a network service to allow CDN servers to seamlessly leverage ISP edge resources to handle extreme viewer dynamics.

2.3 Session Hijacking

TCP session hijacking was originally developed as a penetration technique to take over a connection between the server and the client to eavesdrop or intervene the connection on behalf of the server or the client [18]. Recently, with the advance of Network Function Virtualization (NFV), transparent TCP proxies witness growing popularity. Through session hijacking, transparent proxies can perform various functionalities without disrupting an existing TCP connection or any application running on it. Various transparent HTTP proxies such as Squid can leverage transparent TCP proxying to deliver cached content to clients on behalf of the server behind an established HTTP session [32]. [21, 23] propose split TCP to improve the performance of mobile TCP connections. [34, 38] leverage session hijacking to accelerate the handshake and TCP slow start phase with improved handshake and forwarding mechanisms. In this paper, we utilize TCP session hijacking to seamlessly migrate users to a VMF.

3 SYSTEM DESIGN

3.1 LiveJack Overview

We start with an overview of a LiveJack live broadcasting scenario consisting of a streaming server, a logically centralized LiveJack controller, distributed VMFs, and multiple viewers that are requesting the same live content from the server. Figure 4 illustrates the architecture and how different modules interact with each other.

In LiveJack, the **streaming server** can serve each viewer directly via an individual *transport session*. We define the term **transport session** as the transport layer connection through which all session-dependent signaling messages and live content data are delivered. For most popular live streaming protocols (i.e., RTMP, HLS), the transport session refers to an established TCP connection[1]. Upon each viewer access, the server sends the central controller a request, which contains the detailed transport information of the established session (address and port of both server and viewer) along with the ID of the live content requested by the viewer. If the controller assigns a VMF in response to the request, the streaming server sets up a *footprint session* with the assigned VMF if such a footprint session does not exist. We define **footprint session** as a special transport session between a streaming server and a VMF. Once a footprint session is established, the streaming server only sends one copy of live content data to the footprint session, and only sends session-dependent signaling messages to corresponding transport sessions. A **VMF** is essentially a "hijacker" middle-box function for the transport sessions traversing through it. It can sniff and hijack any transport session (explained later in Section §4).

[1]Livejack can be easily modified to serve UDP based streaming protocols such as RTP/RTCP

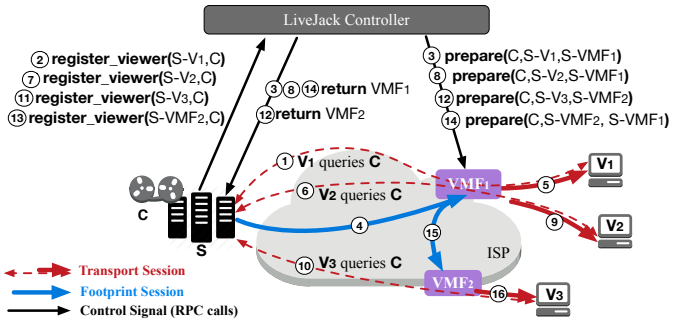


Figure 4: LiveJack service workflow.

After the footprint session is established, a VMF is responsible for replicating and injecting content data from the footprint session to the hijacked transport sessions. From the **viewer**'s perspective, it has no knowledge of the existence of any VMFs and receives all the signaling and data packets from the transport session set up with the server.

The **LiveJack controller** has three primary responsibilities. First, the controller tracks the evolving viewer demands, service quality, and resource availability, and interacts with the edge clouds to strategically deploy VMFs at optimized locations. Second, given the information in the server request, the controller determines which VMF to be assigned to assist each transport session. Last, when a VMF is ready for streaming, the controller configures the network to chain the transport session through the assigned VMF.

Note that: 1) a footprint session is also a TCP connection. The server can treat it the same way as a transport session and request the controller to assign a new VMF. Therefore, each VMF thinks it is talking directly to the server while the data is actually injected by an upstream VMF. Such a design enables LiveJack to construct an efficient footprint session tree by dynamically chaining VMFs together and significantly reduce the overall network bandwidth usage. 2) A VMF can be assigned to any transport session at any time. Therefore, after making the request to the controller, the server shall start streaming content data to the viewer directly through the established transport session immediately, and migrate the session on-the-fly to a VMF later as instructed by the controller. 3) the streaming server can detect a VMF failure or sessions experiencing poor performance through the signaling messages received from the transport sessions[2]. When such an event is detected, the streaming server terminates the affected transport sessions. Typically, in client implementation, the disconnected viewers would re-initiate new sessions with the server. The server has the option to serve them directly or through other available VMFs. VMFs affected by an upstream VMF failure are treated alike.

3.2 LiveJack Workflow

We now present an example in Figure 4 to explain how LiveJack works in detail. Initially, assume there is no viewer or VMF. The first viewer V_1 initiates a query for a live content C available at the stream server S. A transport session S-V_1 between S and V_1 is established using an application layer protocol (step ①). S sends a `register_viewer` request to the controller carrying the transport

[2]Failure handling can also be done by the controller to simplify streaming server in large-scale deployment

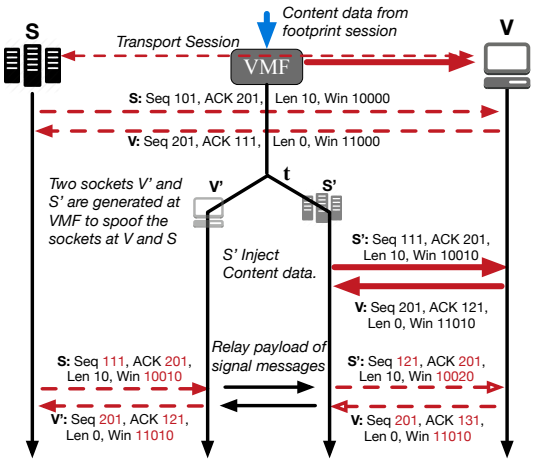


Figure 5: VMF session hijacking.

information of S-V_1 and the content ID for C (step ②). Assuming the controller decides to assign VMF_1 for this session, the controller prepares VMF_1 for streaming (step ③) by: a) routing the traffic of S-V_1 through VMF_1; b) providing the transport information of S-V_1 to VMF_1 for hijacking; and c) informing VMF_1 to expect content C from S. A `prepare` call from the controller to the helper VMF carries three arguments: the content ID, the targeting transport session, and the helper footprint session. Once VMF_1 is ready, the controller notifies S that VMF_1 is assigned to session S-V_1. Upon notification, S sets up the footprint session S-VMF_1 and starts sending content through S-VMF_1 (step ④). VMF_1 injects the received data to session S-V_1 (step ⑤). Note that S does not block to wait for VMF_1 to get ready. After making the `register_viewer` request, S may start streaming with V_1 using session S-V_1. S later switches to sending only signaling packets through S-V_1 when the footprint session S-VMF_1 is ready.

Similarly, when the second viewer V_2 queries the same content C, the transport session S-V_2 is established first (step ⑥), followed by the `register_viewer` request to the controller (step ⑦). Assume VMF_1 is selected and prepared to assist V_2 as well (step ⑧). In this case, since the footprint session S-VMF_1 is already active, VMF_1 can replicate the received content data and inject to both S-V_1 and S-V_2 (step ⑨). Assume later a third viewer V_3 is assigned to a different VMF_2 (step ⑫) after following similar steps (⑩⑪). When setting up the footprint session S-VMF_2, S treats VMF_2 the same way as a normal viewer, and send a `register_viewer` request to the controller (step ⑬). Assume the controller assigns VMF_1 for help. In this case, data from the footprint session S-VMF_1 is directly injected to the new footprint S-VMF_2. Subsequently, VMF_2 inject the data again into the session S-V_3 (steps ⑭⑮⑯).

4 IMPLEMENTATION

We implemented a prototype system of LiveJack for demonstration and evaluation. This section discusses the implementation details.

Development Environment. In our prototype system, the controller communicates with VMFs and streaming servers through RPC calls. VMFs are deployed on individual virtual machines (VMs). The session hijacking and content injection logic are implemented

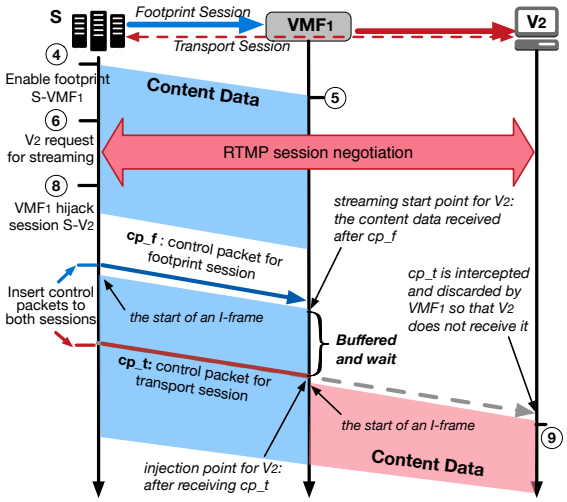


Figure 6: Control packets for video synchronization. The footprint session includes all the blue arrows or shape, while the transport session are marked red. Consistent with Figure 4, after VMF_1 hijacks the session S-V_2, cp_f is added before the first I-frame that is supposed to be sent to V_2 in the footprint session, while cp_t is added in the transport session to mark the instant to inject contents packets. In the example, cp_f arrives earlier than cp_t and VMF_1 buffers the received video and injects them to S-V_2 when cp_t arrives. Both control packets are intercepted and dropped by VMF_1.

using Netfilter and IPtables [33], which delegate packet processing to a user-space LiveJack-VMF program. We choose the RTMP-enabled Nginx web server as the streaming server. The RTMP protocol is currently a popular choice among live content providers such as Twitch and Facebook Live. We need to attach a lightweight LiveJack streaming plugin to the Nginx source code to subscribe to LiveJack services on the server side. All viewers are off-the-shelf video players that support RTMP streaming.

Session Chaining and Hijacking. LiveJack can leverage different techniques to steer any transport session through an assigned VMF. LiveJack controller managed by ISP can boot VMFs at the ISP provider edge (PEs) as PE components similar to a virtual firewall. When a transport session traverses the PE to reach the backbone network, the VMF has the chance to serve the session as a middlebox. Alternatively, if no VMF is available on the path, the ISP can configure IP tunnels or interact with SDNs controller to set up paths between any anchor points along the default routes and the VMFs. By dynamically mapping sessions to particular tunnels or paths, live sessions can be routed through the assigned VMFs. Various prior research on middle-box chaining serves our purpose [12, 14, 30, 37], on which we will not elaborate. In our prototype system as a proof-of-concept, we use Ryu OpenFlow [4] to install exact rules to steer the path between the streaming server and VMFs for each session.

We adapt "session hijacking" [18] to inject content packets received from the footprint session into the transport sessions, which are all TCP connections in our prototype system. Figure 5 shows one example of hijacking a transport session between S and V. We implement Netfilter hooks in a kernel module which can access any packet chained through VMF. A user-space program listens to RPC calls from LiveJack controller. Upon receiving `prepare` call, the program informs the kernel module to transparently sniff the packets of the transport session that needs to be hijacked, where the TCP session states such as SEQ/ACK numbers and timestamps can be tracked. At time t, VMF spoofs two TCP sockets, which breaks the original transport session into two subsessions[3]. The TCP socket S' at the VMF facing the viewer then spoofs the socket S, while V' facing the server spoofs socket V. Through the spoofed S', VMF can then inject content data received from the footprint session on behalf of the streaming server. The payloads of the signaling packets between S and V are relayed between V' and S'.

After hijacking, VMF applies constant offsets to SEQ/ACK numbers, modifies source/destination addresses and set local timestamps to proxy packets from the original transport session to the two subsessions and vice versa. Therefore, VMF only maintains in the order of tens of constants for each session with no time-varying states. The two subsessions work independently to handle transport functions such as congestion control and retransmission.

Control Packets for Video Synchronization. We notice a video synchronization problem in our implementation. In the previous workflow example (Figure 4), V_2 joins streaming and expects to receive video injected by VMF_1 at step ⑨. However, since VMF_1 has already started streaming with S and V_1 at the time, there should be a mechanism to inform VMF_1: of all the live video that VMF_1 is receiving from S, from which exact packet should VMF_1 start to inject into the session S-V_2. Failing to pinpoint the start point will cause V_2 to miss the first I-frame packet, which is required to decode subsequent frames predicted from the I-frame.

To address this problem, we propose to send *control packets* in both the transport session and the footprint session for video synchronization. Figure 6 illustrates how this control method works step-by-step. The control packet added to the footprint session (S-VMF_1) marks the first video packet needed by V_2 and the control packet added to the transport session (S-V_2) indicates when VMF_1 can start injecting video. Both control packets will be intercepted and dropped by the VMF_1 so that V_2 is not aware of them.

Although the control packets are currently designed for video synchronization, we want to mention that such mechanism can be extended to serve general session control functionalities. Compared to the RPC calls that has the LiveJack controller in the loop, the in-band control packets are more responsive and are in sync with the transport session. By sending control packets with different formats, the server may flexibly offload various application functionalities to its assigned VMFs (discussed later in Section §6).

5 EVALUATION

5.1 LAN Evaluation

Topology and Components: We have implemented a LAN testbed with four DELL R730 servers. Figure 7 shows the logical topology of the testbed. We deploy one streaming server in the network. There are two tiers of VM-based relay nodes in the testbed, each of which can be configured as a pure forwarding hop, a proxy server, or a VMF in different test scenarios. Viewers are generated

[3]The kernel module spoofs SYN-ACKs from V to S' and S to V' to fake the handshake for S' and V', respectively.

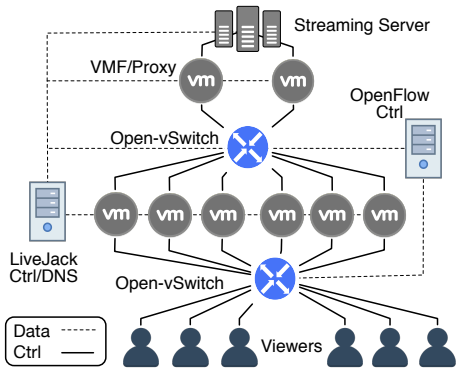


Figure 7: LAN testbed setup

at Docker containers with different IPs or TCP ports. Two Open vSwitches (OVS) connect different tiers. Through the Ryu OpenFlow controller [4], various routes are configured between the two tiers. There is an additional VM configured as the LiveJack controller or a DNS server.

Test Scenarios: The testbed can be configured into three test scenarios to compare LiveJack against different solutions:

- **Single Streaming Server:** Each relay node is a forwarding hop. This scenario serves as a benchmark of that viewers are directly served by the original content servers and/or CDN servers.
- **DNS-based Proxying:** A fixed number of Nginx-RTMP proxy servers are deployed in the relay nodes. A dynamic DNS server is configured to load balance viewers to different proxy servers. We configure a five-minute delay to emulate the real world latency introduced by populating DNS updates[24, 27]. A viewer cannot switch between proxy servers while it is being served.
- **LiveJack:** Viewers are initially served by the streaming server directly. When the live content becomes available on the VMF, OpenFlow controller routes the session through the VMF which then hijacks the session and seamlessly injects content into the session on behalf of the server. VMFs can be dynamically booted or shutdown based on the user workloads.

Metrics: When certain link becomes congested or certain VM gets overloaded, viewers experience high TCP retransmission rate and high jitter in received RTMP frames. Therefore, we use TCP goodput and average frame jitter as the metrics to evaluate QoE of viewers. Frame jitter is defined as the standard deviation of the inter-arrival time of frames. We randomly select some viewers and implement them using FFprobe to monitor per-frame statistics. All the other viewers are implemented using RTMPdump.

System Capacity Stress Test. In this test, we compare the maximum system capacities of LiveJack with VMFs enabled at all eight VMs and that of the single streaming server scenario. Each virtual link is rate limited to 200Mbps according to the typical VM network bandwidth in commercial clouds [5]. We generate 20 channels with the same 480p 600kbps Big Buck Bunny sample video. Viewers are randomly generated at different containers in this test. Figure 8 and Figure 9 show the results for single streaming server and eight-VMF LiveJack, respectively. A single streaming server can support only up to 250 viewers. Beyond 250 viewers the video frame jitter increases significantly and the goodput decreases, leading to an unstable system. By contrast, eight-VMF LiveJack supports up to 1,620 viewers with sub-100ms frame jitter and no observable goodput degradation. Beyond 1,620 viewers, the jitter exceeds 100ms due to the congestions in VMF egress bandwidth.

Individual VMF Performance. VMF can support more viewers with a higher egress bandwidth. In this test, we increase the egress bandwidth of a VMF to 3.3Gbps by accelerating OVS with DPDK [19].This setup mimics the real-world edge clouds that are close to viewers and have a high user-facing bandwidth. Figure 10a shows the goodput and frame jitter as we increase up to 500 viewers served by a single VMF with one 480p channel. The results show that a VMF can smoothly support all the viewers with 83ms frame jitter and no observable loss of video rate. At the 500-viewer workload, we further test two kinds of video sources (i.e. Big Buck Bunny and Elephant Dream) with different resolutions and bitrates. The source bitrates for 360p, 480p, 720p and 1080p are 300kbps, 600kbps, 1200kbps and 2400kbps, respectively. The results in Figure 10b show that a single VMF can safely support all 500 viewers with moderate bitrate for 360p/480p/720p video sources. Some viewers experience a higher rate variance for 1080p video while the average is moderate.

Adaptation to User Dynamics. In this test, we evaluate LiveJack's response to dynamic user demands. We compare LiveJack against DNS-based Proxying systems. We scale a 100min user trace collected from Facebook-Live to generate the workload. The streaming server cyclically plays a 480p Big Buck Bunny sample video with a length of 5min 30sec. Based on the geo-locations of the viewers in the real traces, each viewer is mapped to one of the containers. Since the trace does not contain the service time of individual viewers, the service time is configured to be exponentially distributed with a mean of 10min. All the links are configured to 200Mbps. For the DNS scenario, we create a two-proxy-server (Proxy-2) system and an over-provisioned four-proxy-server (Proxy-4) system. Every five minutes, the mapping of viewers to proxy servers are re-calculated according to the load of each proxy server in the previous five-min slot. The updated user mapping will be applied in the next time slot. LiveJack starts with one VMF serving up to 350 concurrent viewers. For every 350 more viewers, a new VMF is booted. For a fair comparison, LiveJack can boot up to four VMFs. A viewer is always chained and served by the VMF with the least load when it accesses the service.

Figure 11 shows the number of viewers, the average frame jitter and the goodput of LiveJack, Proxy-2, and Proxy-4 over time. Both Proxy-2 and Proxy-4 experience high frame jitter since the DNS-based mapping cannot keep up with the user dynamics. Even at lower workloads during 0-20min, Proxy-2 and Proxy-4 experience sporadic jitter bursts due to unbalanced user workloads. With over-provisioned proxy servers, Proxy-4 performs slightly better than Proxy-2 with fewer jitter bursts. However, Proxy-4's responses to flash crowds at 18min, 52min and 74min are still bad. For comparison, LiveJack maintains a low frame jitter and a smooth goodput throughout the test. With the on-the-fly session migration, VMFs achieve close to the optimal load balancing. The frame jitter increases by a few tens of milliseconds at 52min and 74min when viewers are served by the streaming server before new VMFs become ready. The result also shows that LiveJack can elastically scale

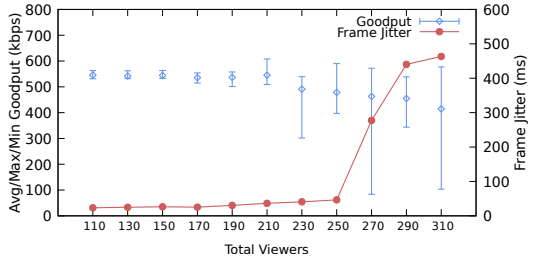


Figure 8: Single Streaming Server Capacity Benchmark

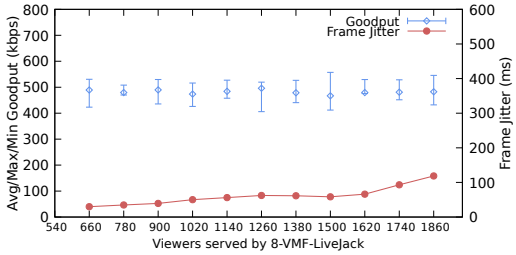
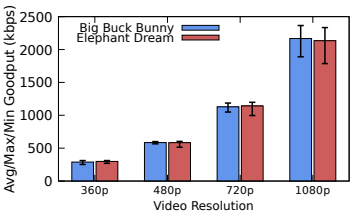


Figure 9: Eight-VMF LiveJack Capacity Benchmark

(a) Maximum viewer support



(b) Goodput v.s video resolution

Figure 10: Individual VMF benchmark

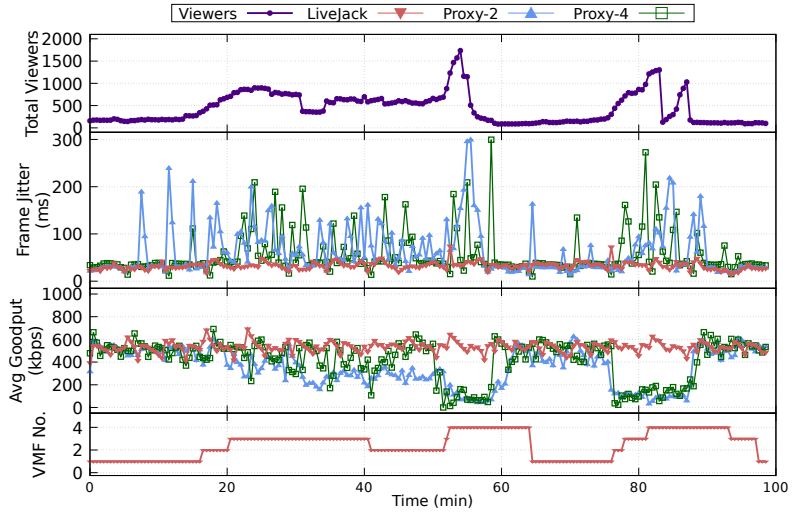


Figure 11: Response to user dynamics for LiveJack and DNS-based Proxying

and shrink VMFs in response to user dynamics. Note that VMFs are not shut down immediately as the workload shrinks (e.g. at 32min and 50min.) until all active viewers served by the VMFs fade out[4].

5.2 WAN Evaluation

To demonstrate LiveJack's performance in real-world wide area networks, we deploy a small-scale LiveJack prototype in GENI network. The WAN testbed includes a streaming server, multiple viewers, two VMFs, and the LiveJack controller deployed across four federal states as shown in Figure 12a. Two data paths for live content delivery are configured, where the server machine "srv" at UCLA broadcasts to viewers at "usr1" and "usr2" in Boston GPO via "relay1" at Wisconsin or "relay2" at GaTech. The end-to-end bandwidth from "srv" to "usr1" or "usr2" through the two data paths is only around 10.5Mbps. The bandwidth from "vmf1/2" to "usr1/2" less than 49.3Mbps. To accommodate the capacity, we generate only up to a total of 20 viewers at "usr1/2" and stream only one 256kbps sample video from "srv."

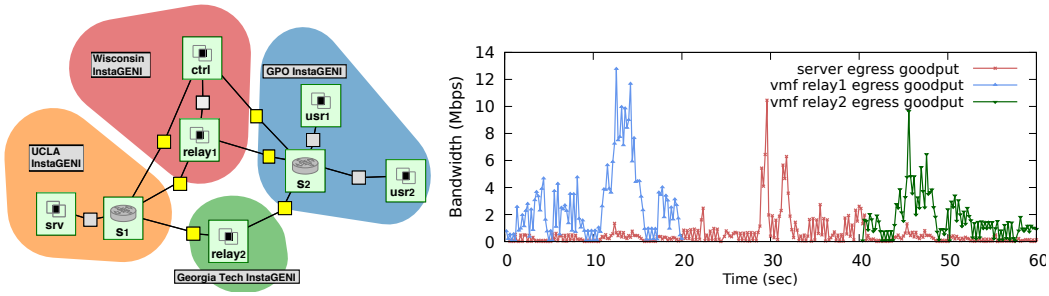
Service Latency. We first measure the service latency of both LiveJack and the proxy server solutions. We evaluate two kinds of service latency: the average end-to-end delivery lag between when a frame is sent by the server and when it is received at a viewer; the start-up latency between a viewer requests the content and the first video frame is received. We generate a total of 20 viewers at "usr1" and "usr2" in the GPO site in Boston. Ten of the viewers at "cli1" is served by VMF at "relay1" and the other ten at "cli2" are served by "relay2". For comparison, we also test the single-streaming-server scenario (i.e. all viewers are served by the server directly) and the proxying scenario (i.e. "relay1" and "relay2" configured as proxy servers). The average start-up latency, delivery lags and breakdown components of three scenarios are listed in Table 1.

Table 1: Start-up Latency and Delivery Lag

Setup	Itemize	Min/Avg/Max(ms)
S(Server)-V(Viewers)	E2E delivery lag	43.52/60.89/98.70
	Total start-up	795/1022/2072
S-VMF-V	E2E delivery lag	57.36/62.82/107.14
	E2E delay	53.35/54.84/69.20
	Total start-up	811/1080/1671
	RTMP handshake	680/850/1565
	RPC calls	65/176/260
	Session hijacking	143/257/698
S-S-V	E2E delivery lag	84/86.67/340.17
	E2E delay	53.24/56.40/77.19
	Total start-up	343/605/2017

The result reveals that LiveJack only introduces 1.93ms additional delivery lag compared to directly serving clients from the server. Most of the delivery lag is contributed by the end-to-end delay (E2E delay) from the west coast to the east coast. This greatly outperforms proxy-based solution which introduces 25.78ms for a single proxy relay. Since proxy server runs at the application

[4]To better show the elasticity of LiveJack, each viewer in our test is configured to leave in 20min. Theoretically, LiveJack can support a seamless consolidation of sessions from excessive VMFs, which is similar to migrating session from the streaming server to a VMF. The implementation of user consolidation in LiveJack is left for future work.



(a) GENI WAN testbed topology **(b) Egress Goodput of Nodes during failover.**

Figure 12: WAN evaluation of LiveJack

layer and implements streaming protocol logic, the lag is higher than VMF which purely duplicates and injects frames to viewers. Regarding start-up latency, LiveJack is comparable to the single-streaming-server setup. Most start-up latency is contributed by RTMP session negotiation, while the add-on logics of LiveJack such as RPC calls or session hijacking add little to the total start-up latency. The proxy-based solution enjoys the fastest start-up latency because the session negotiation is between the viewers and the proxy in their proximity. In practice, LiveJack will work with CDN streaming servers in edge clouds that are much closer to viewers than our current WAN setup, therefore, will achieve much shorter start-up latency. Meanwhile, the worst-case start-up latency of proxy-based-solution is much longer than LiveJack. When the content is not yet available at the proxy, it requires extra time to stream from the origin server and the viewers can be blocked before the contents are ready. In contrast, LiveJack does not block service even when VMFs are not yet ready.

VMF failover. In the second test we demonstrate how LiveJack reacts to VMF failover. Initially, ten viewers are served by a VMF at "relay1" and transport sessions traverse through "srv-s1-relay1-s2-cli1". At 20 sec, we shutdown the VMF at "relay1". Since viewers are not aware of VMFs, the affected viewers immediately try to re-connect and are then served by "srv" directly. The LiveJack controller detects that the VMF at "relay1" goes down, and controls the switches "s1" and "s2" to chain the user sessions through "vmf2". At 40sec, the VMF at "relay2" becomes ready, and seamlessly hijacks the sessions and serves the viewer on behalf of the server. The egress bandwidth of the server and VMFs are show in Figure 12b.

This result demonstrates that LiveJack is able to swiftly failover between VMFs in case of VMF failure or regional bandwidth congestion for certain viewers. The average down time between VMF at "relay1" goes down and all ten viewers being reconnected to the server is 1,436 ms, which is mostly contributed by re-negotiating RTMP channels. The handover from server to VMF at "relay2" introduces no down time or TCP reconnect. Therefore, LiveJack outperforms DNS or redirection-based solutions in case of site failures, which takes five min or more to populate user mapping. Although the failover in current LiveJack implementation is not as seamless as P2P solutions, we consider VMFs to be relatively reliable server-based entities, which do not introduce as many failures/churns as in P2P systems [25]. Meanwhile, LiveJack's robustness against VMF failures can be further enhanced by higher level client-side application solutions, such as buffering or multi-homing [7].

6 DISCUSSION

Content Encryption - Modern content services are generally encrypted from service provider to end users. In LiveJack, we assume content provider owns or share secure domain with the VMFs. As part of our ongoing work, we are implementing transport layer security (TLS) in VMF for content encryption. By extending the use of control packets, a streaming server can populate the negotiated key information to designated VMFs who encrypt user data. In this way, LiveJack provides content encryption as a service for CDN, which scales with VMFs to reduce the computation complexity of streaming servers.

DASH Support - Although we implement the system with RTMP protocol, the LiveJack framework can support HTTP-based streaming protocols. Since HTTP is pull-based, VMF needs to be adapted to recognize and serve the pull requests for contents from the clients. LiveJack can also support adaptive streaming. The streaming servers can simultaneously stream multiple bitrates to a same VMF. By extending the use of control packets, a server can notify VMFs to select bitrates for users. DASH support for LiveJack is listed as our on-going work.

VMF Placement and Session Mapping - The centralized control logic is highly simplified in current LiveJack implementation. Prior research in data center networking [20, 26] indicates that carefully engineering the placement of VMFs can effectively reduce network traffic and provide better streaming quality to viewers. Optimized deployment of VMFs may depend on the geo-locations of the streaming servers, the clouds and the viewers, the network topology and utilization, available cloud resources. Furthermore, we have noticed that the mapping from user requests to deployed VMFs affect the service quality. Joint optimization of VMF placement, session mapping, and fine-grain traffic engineering is also part of our ongoing work.

7 CONCLUSION

In this paper, we proposed a novel transport layer live content broadcasting system named LiveJack. LiveJack leverages edge cloud resources to create virtual media functions (VMFs) on demand. Through transparent "session hijacking" techniques, LiveJack enables live content delivery with high scalability, bandwidth efficiency, and flexible service options. Preliminary prototyping and evaluation in both LAN and WAN demonstrated that LiveJack can accommodate extreme user dynamics, increase the number of viewers served by streaming servers, react responsively to failures while maintaining very small latency overheads to the overall service.

REFERENCES

[1] 2016. AT&T Integrated Cloud to Include 105 Data Centers by Years End. https://www.sdxcentral.com/articles/news/att-integrated-cloud-count-100-data-centers-year-end/2016/04/. (2016).

[2] 2016. Justin.tv's Live Video Broadcasting Architecture. http://goo.gl/Q9c1KK. (2016).

[3] 2016. Twitch & Justin.tv Ingest Servers. https://bashtech.net/twitch/ingest.php. (2016).

[4] 2017. Ryu SDN framework. https://osrg.github.io/ryu/. (2017).

[5] Amazon AWS. 2017. Amazon EC2 Instance Configuration. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-ec2-config.html. (2017).

[6] Mike Axelrod. 2008. The Value of Content Distribution Networks and Google Global Cache. https://www.isoc.org/isoc/conferences/inet/08/docs/inet2008_kiagri.pdf. (2008).

[7] Suman Banerjee, Seungjoon Lee, Ryan Braud, Bobby Bhattacharjee, and Aravind Srinivasan. 2004. Scalable resilient media streaming. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*. ACM, 4–9.

[8] Salman A Baset and Henning Schulzrinne. 2004. An Analysis of the Skype Peer-to-peer Internet Telephony Protocol. *arXiv preprint cs/0412017* (2004).

[9] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. 2014. GENI: A federated testbed for innovative network experiments. *Computer Networks* 61 (2014), 5–23.

[10] YouTube Official Blog. 2012. Mission complete: Red Bull Stratos lands safely back on Earth. https://youtube.googleblog.com/2012/10/mission-complete-red-bull-stratos-lands.html. (2012).

[11] Meeyoung Cha, Pablo Rodriguez, Jon Crowcroft, Sue Moon, and Xavier Amatriain. 2008. Watching Television over an IP network. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. ACM, 71–84.

[12] Rafael Fernando Diorio and Varese Salvador Timóteo. 2015. A Platform for Multimedia Traffic Forwarding in Software Defined Networks. In *Proceedings of the 21st Brazilian Symposium on Multimedia and the Web*. ACM, 177–180.

[13] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. 2000. Deployment Issues for the IP Multicast Service and Architecture. *IEEE network* 14, 1 (2000), 78–88.

[14] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2014. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions using Flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 543–546.

[15] Ken Florance. 2008. How Netflix Works With ISPs Around the Globe to Deliver a Great Viewing Experience. https://goo.gl/joQtjz. (2008).

[16] Benjamin Frank, Ingmar Poese, Yin Lin, Georgios Smaragdakis, Anja Feldmann, Bruce Maggs, Jannis Rake, Steve Uhlig, and Rick Weber. 2013. Pushing CDN-ISP Collaboration to the Limit. *ACM SIGCOMM Computer Communication Review* 43, 3 (2013), 34–44.

[17] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. 2015. C3: Internet-scale Control Plane for Video Quality Optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 131–144.

[18] B Harris and R Hunt. 1999. TCP/IP Security Threats and Attack Methods. *Computer Communications* 22, 10 (1999), 885–897.

[19] DPDK Intel. 2014. Data Plane Development Kit. http://dpdk.org. (2014).

[20] Joe Wenjie Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. 2012. Joint VM Placement and Routing for Data Center Traffic Engineering. In *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2876–2880.

[21] Swastik Kopparty, Srikanth V Krishnamurthy, Michalis Faloutsos, and Satish K Tripathi. 2002. Split TCP for Mobile Ad-hoc Networks. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, Vol. 1. IEEE, 138–142.

[22] Federico Larumbe and Abhishek. Mathur. 2015. Under the Hood: Broadcasting Live Video to Millions. goo.gl/qpBAJM. (2015).

[23] Franck Le, Erich Nahum, Vasilis Pappas, Maroun Touma, and Dinesh Verma. 2015. Experiences Deploying a Transparent Split TCP Middlebox and the Implications for NFV. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 31–36.

[24] Hongqiang Harry Liu, Raajay Viswanathan, Matt Calder, Aditya Akella, Ratul Mahajan, Jitendra Padhye, and Ming Zhang. 2016. Efficiently Delivering Online Services over Integrated Infrastructure. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 77–90.

[25] Yong Liu, Yang Guo, and Chao Liang. 2008. A Survey on Peer-to-peer Video Streaming Systems. *Peer-to-peer Networking and Applications* 1, 1 (2008), 18–28.

[26] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. 2010. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *INFOCOM, 2010 Proceedings IEEE*. IEEE, 1–9.

[27] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. 2015. Practical, Real-time Centralized Control for CDN-based Live Video Delivery. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 311–324.

[28] Tim Peterson. 2017. Twitter's NFL Streams Averaged 265,800 Viewers per Minute across 10 Games. http://marketingland.com/twitters-nfl-streams-averaged-265800-per-minute-viewers-across-10-games-204824. (2017).

[29] Karine Pires and Gwendal Simon. 2015. Youtube Live and Twitch: A Tour of User-generated Live Streaming Systems. In *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 225–230.

[30] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement using SDN. In *ACM SIGCOMM computer communication review*, Vol. 43. ACM, 27–38.

[31] Dan Rayburn. 2010. For Video Delivery, It's not about A Distributed Vs. Non-Distributed Network. http://blog.streamingmedia.com/2010/04/for-video-delivery-its-not-about-distributed-versus-nondistributed.html. (2010).

[32] Alex Rousskov and Valery Soloviev. 1999. A Performance Study of the Squid proxy on HTTP/1.0. *World Wide Web* 2, 1-2 (1999), 47–67.

[33] Rusty Russell and Harald Welte. 2002. Linux Netfilter Hacking Howto. http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html. (2002).

[34] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM, 44–49.

[35] Raymond Sweha, Vatche Ishakian, and Azer Bestavros. 2012. Angelcast: Cloud-based Peer-assisted Live Streaming using Optimized Multi-tree Construction. In *Proceedings of the 3rd Multimedia Systems Conference*. ACM, 191–202.

[36] Bolun Wang, Xinyi Zhang, Gang Wang, Haitao Zheng, and Ben Y Zhao. 2016. Anatomy of a Personalized Livestreaming System. In *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM, 485–498.

[37] Matthias Wichtlhuber, Robert Reinecke, and David Hausheer. 2015. An SDN-based CDN/ISP collaboration architecture for managing high-volume flows. *IEEE Transactions on Network and Service Management* 12, 1 (2015), 48–60.

[38] Tilman Wolf, Shulin You, and Ramaswamy Ramaswamy. 2005. Transparent TCP acceleration through network processing. In *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, Vol. 2. IEEE, 5–pp.

[39] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. 2009. Design and Deployment of a Hybrid CDN-P2P System for Live Video Streaming: Experiences with LiveSky. In *Proceedings of the 17th ACM international conference on Multimedia*. ACM, 25–34.

[40] Cong Zhang and Jiangchuan Liu. 2015. On Crowdsourced Interactive Live Streaming: a Twitch.tv-based Measurement Study. In *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 55–60.