A scalable load generation framework for evaluation of video streaming workflows in the cloud

Roberto Ramos-Chavez
roberto@unified-streaming.com
Unified Streaming
Amsterdam, The Netherlands

Theo Karagkioules
theo@unified-streaming.com
Unified Streaming
Amsterdam, The Netherlands

Rufael Mekuria
rufael@unified-streaming.com
Unified Streaming
Amsterdam, The Netherlands

Abstract

HTTP Adaptive Streaming (HAS) is increasingly deployed at large, gradually replacing traditional broadcast. However, testing large-scale deployments remains challenging, costly and error-prone. Especially, testing with realistic streaming loads from massive numbers of users is challenging and costly. To improve this, we introduce an open-source load testing tool that can be deployed in the cloud or on-premise in a distributed manner, for load generation.

Our presented tool is an extension of an existing open-source web-application load-testing tool. In particular we have added functionality, that includes streaming load generation for a multitude of protocols (i.e. Dynamic Adaptive Streaming over HTTP (DASH) and HTTP-Live-Streaming (HLS)) and use-case implementations (e.g. live streaming, Video on Demand (VoD), bit-rate switching). The extension facilitates testing streaming back-ends at scale in a resource-efficient manner. We illustrate our tool's capabilities via a series of use-cases, designed to test, among others, how streaming deployments perform under different load scenarios, i.e. steep or gradual user ramp-up and stability testing over long periods.

CCS Concepts

• **Information systems → Multimedia content creation**; • **Applied computing → Event-driven architectures**; • **Computer systems organization → Cloud computing**.

Keywords

Performance testing, Cloud computing, Experimentation

ACM Reference Format:
Roberto Ramos-Chavez, Theo Karagkioules, and Rufael Mekuria. 2020. A scalable load generation framework for evaluation of video streaming workflows in the cloud. In *11th ACM Multimedia Systems Conference (MMSys'20), June 8–11, 2020, Istanbul, Turkey*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3339825.3394930

1 Introduction

During the last decade, video streaming has evolved to constitute a major fraction of today's Internet traffic [3]. However a closer look, reveals that a large fraction of this traffic is generated by a few popular Over-The-Top (OTT) streaming services, such as for instance YouTube, Netflix or Twitch [18]. A significant portion of the remaining consumed video traffic is still generated through traditional broadcast. Deploying video streaming at scale in a cost effective and reliable manner, i.e. one that can successfully replace broadcast television, remains challenging. There exist several widely available technologies for video content distribution over the Internet, e.g. media encoding, HAS, cloud computing, Content Delivery Networks (CDN) etc., while additional technologies for monitoring, ad-insertion and analytics are becoming increasingly popular. Nonetheless, combining many of these technologies, often results in complex workflows for streaming platforms. Thus, testing and deploying such composite platforms reliably and at scale, becomes a challenge for both operators and broadcasters. Further, HAS introduces, by design, more server resource demands, often variable when compared to broadcast.

Recently, looking at some high-profile examples, as for instance in the domain of prime-time live sports [15],[21], problems were reported on the match-day, in an otherwise stable platform. In such cases, the cloud or network deployment was not able to support the massive number of users and their behavior. There can be many reasons for failures, ranging from network failures and limitations to server capacity, to incorrectly configured clients, or even failing server nodes. Provisioning against such occurrences and failures, makes realistic load testing of streaming platforms, an absolute necessity. Typically, platforms have well defined requirements such as supporting a certain number of concurrent viewers, and resisting sudden changes in the number of users (e.g. flash clouds) or malicious/erroneous intent (e.g. distributed denial-of-service (DDoS) attacks). To test for such workloads, generating and applying them to the system under test is critical. However, deploying and emulating individual clients at scale is often impossible. One approach that is often deployed is based on A/B-based testing. However, A/B distribution requires that there already exists a well functioning deployment in place to compare with, which is not always feasible. As an alternative, we introduce a load testing approach based on an open-source load generator, that can emulate many end-users/clients on a single machine, and even more in a distributed setup with multiple machines (e.g. deployed in the cloud). Further, the proposed framework provides video streaming-specific extensions for testing video-streaming use-cases.

Our tool is not only useful for stress and load testing, but may also be used to study and evaluate cloud-based deployments, as performed in [13]. In this case, correlation of client and server metrics is used to develop server-side monitoring and server resource

MMSys'20, June 8–11, 2020, Istanbul, Turkey

ACM ISBN 978-1-4503-6845-2/20/06…$15.00
https://doi.org/10.1145/3339825.3394930

allocation. Another use-case includes economic analysis and optimization, in order to reduce deployment costs for cloud-based platforms. For instance, by stress-testing different cloud deployments at different levels, a more cost-effective solution may be chosen that still matches the target requirements.

In summary, we present a load generation framework for evaluation of complex video streaming workflows, deployed in the cloud. Our framework is based on Locust [11], which is a Python-based open-source tool for evaluation of the behavior of web applications with multiple concurrent users (swarms). The framework can generate adaptive streaming traffic load by creating a swarm of clients (users), that perform video streaming under the HAS principle. In part, it enables experimentation, allowing variation of:

- streaming protocols (DASH [8], HLS [16])
  - including client behavior,
- streaming client specific parameters (such as buffer size),
- streaming applications (VoD, live streaming),
- user hatch (new user joining) rate and
- maximum number of users (to emulate)

The load testing tool is open-sourced and made publicly available at [17]. We believe that the multimedia community can benefit from the extended functionality provided by our framework, that facilitates large-scale testing (experimentation) of cloud or network-media workflows. We hope this can also encourage more work on server-side deployment of HAS.

The remainder of the paper is organized in the following sections. In Section 2 we provide prior work on load testing methodologies and similar software tools and some target use-cases, whereas in Section 3 we specify the design principle that we followed and the main modules that constitute our framework, along with a description of our experimental design. Further, in Section 4 we detail two use-cases and provide indicative experimental results for each use-case. Last, in Section 5 we conclude the paper and list possible future extensions.

2 Background and motivation

2.1 Related Work

Server load testing generally aims to find out how a system under test can handle various loads and maintain an acceptable level of Quality of Service (QoS). Some of the system metrics that can be evaluated include response time, throughput, and availability [14]. To enable load testing, realistic load emulation is an important aspect. In that direction, Summers et. al. [22] provide methodologies for generating HTTP video workloads that try to accurately model the requested traffic. Similarly, Koskimies et al. [10] present a Quality of Experience (QoE) estimation–based server bench-marking system targeted on QoE-optimized resource provisioning of typical video delivery platforms. Draxler et al. [4] provide insights into the performance and resource demands of components of video streaming services by studying the behavior of service function chains in different load situations, while stretching the need for bench-marking tools and experimental data. In terms of network experiments, Frömmgen et al. [5] present a generic framework for the management, the scalable execution, and the interactive analysis of a large number of network experiments, that ensures reproducibility, and which when combined with our load test framework can produce even more realistic testing scenarios. Although load testing and resource usage profiling [2] are well studied subjects in the general context of web applications, our proposed framework is targeted towards addressing precisely the load generation of HTTP video streaming traffic at scale. A similar approach to ours, can be obtained via Apache JMeter [1] that extends its functionality to video streaming when paired with a licensed HLS plugin [24]. The tool provided in this paper can also support alternative protocols like MPEG-DASH and uses lib gevent to emulate many users on a single CPU core, which we believe will provide better support for large scale load generation. In addition it is open-source (without any licensed dependencies), generates DASH and HLS video streaming requests and enables deployment on multiple distributed nodes. In our design intent we focused primarily on the scale of load testing, as we aimed at testing not only servers or isolated components, but also more realistic, complex cloud deployments composed of multiple components.

2.2 Motivation

We briefly highlight some of the example use-cases that have motivated the development of our approach and how each is treated with the proposed framework.

- Live streaming. In such deployments, typical issues include: segment requests beyond the live edge, encoder discontinuities (both possibly resulting in HTTP 404), CDN misconfigurations leading to overload on the origin server, Solid State Disk (SSD) storage depletion leading to system failures, and other issues. Our tool-set contains specific scripts that emulate live streaming clients, in order to detect and mitigate such erroneous operations in live streaming back-ends.
- VoD streaming. Errors similar to the live streaming case may occur, yet the causes may be different. Accessing the storage back-end may introduce delay and performance overhead, leading to response delay and failures. Increased request rate attributed to augmentation of the user set may cause ineffective content caching. Thus, caching needs to be optimized based on content popularity rates. Our tool provides a specific script emulating VoD clients, that perform segment requests to any content items available in a catalog, not restricted to a live play-out edge.
- Provisioning and breaking point detection. When deploying a service, another use-case constitutes the detection of breaking points of provisioned cloud servers. In such cases the load may increase either gradually or abruptly, in order to detect the maximum users capacity of a specific deployment. The 'ramping' and 'hatch' rate functionality provided by our tool can support such tests.
- Long Term (Soak) testing. Another common scenario concerns testing a system for a prolonged time period, where failures like memory leaks, or slight misconfiguration (e.g. storage depletion) can be detected early. In Soak test typically a normal production load is applied, without breaching the boundaries of what the system can support. By running our tool in the cloud Soak testing can be supported as well.

3 Load Testing Framework

3.1 Architecture

The architecture of the load generation framework is illustrated in Figure 1, and is based on Locust [11], an open-source tool for load testing of web applications. Locust is optimized to support the emulation of a large number of users, without using a large number of threads by using an event-driven model. Locust allows writing specific user behavior for load testing through plain Python programming language scripts. Our tool provides example scripts for generating streaming loads based on client protocols such as HLS and MPEG-DASH for different use-cases. The streaming deployment under test is independent of the load testing framework. It can be any back-end providing streaming presentations by making manifests and segments available. When running in distributed mode on multiple machines, one of the Locust instances(nodes) is the 'master' node, which aggregates results of requests from each of the clients. A detailed list of the available client metrics can be found in Section 3.5. The results are stored and can be visualized from the 'master' node. The other nodes are so-called 'slave' nodes that generate additional requests and load to increase the number of users in the emulation.

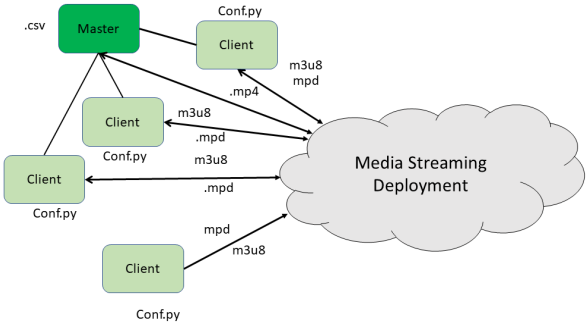

Figure 1: Load testing framework

3.2 Client model design

The framework emulates the behavior of multiple HAS clients. Typically, Hypertext Transfer Protocol (HTTP) video streaming clients. A basic operation of these clients is shown in Figure 2. Optimized clients may use a bit-rate adaptation algorithm to decide on the appropriate representation for each segment download, given network conditions. A performance evaluation of such bit-rate adaptation algorithms exists in [9]. Once a quality decision has been made, a segment is requested by the client. Once a segment is downloaded, it is temporarily stored in a finite-sized data queue, until played-out. It is worth mentioning here that the scheduling of each segment request by the client is performed as a function of the instantaneous buffer length. Every segment is requested immediately after the download of its previous, unless the instantaneous buffer length has reached the maximum buffer capacity; in which case a small inter-request delay is introduced. This application-level behavior is what gives streaming its characteristic bursty traffic pattern [23].

The client behavior is implemented in Python and extends the original functionality of the Locust framework. The proposed client

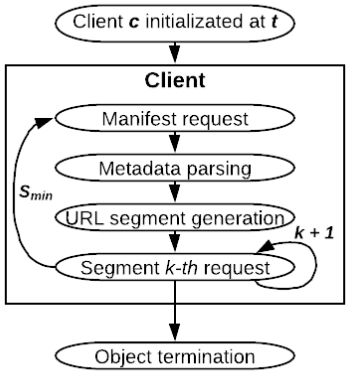

Figure 2: Client model

is used to build a Locust class object shown in Figure 2. The client life cycles are represented by the most common actions of a typical media player: **1)** Object initialization: the media player initializes the required resources such as buffer size and other media libraries to handle and download each of the video fragments. **2)** Manifest requests: it creates an HTTP GET request to retrieve the available media segments. **3)** Metadata parsing: Retrieves the required media information (e.g., quality representations, bit-rate, timing information, minimum manifest update time) for the desired segments to download. **4)** URI segment generation: it creates a URI segment timeline based on the retrieved metadata parameters. **5)** Segment request: it downloads the specified media segments. **6)** Object termination: used resources are removed from memory.

For the case of VoD streaming, the DASH's 'MPD' or HLS's 'Media playlist' are not frequently updated, but for live these are frequently updated. In Figure 2, it is assumed that the minimum update period S_{min} for VoD cases is equal to zero, while in live scenarios, it acquires the value $minimumUpdatePeriod$ for the case of DASH, and the Tag $EVENT$ and $EXT-X-ENDLIST$ for the case of HLS [7].

To parse the different manifest files for both HLS and DASH, we have used the two following open-source Python-based libraries: *m3u8* [6] for HLS and *python-mpegdash* [19] for DASH manifests.

3.3 Media streaming deployment

The media streaming deployment can be any type of streaming implementation deployment. It could be a server or a combination of functional entities providing a streaming service. In terms of additional cloud deployments, we find that the performance of cloud building blocks is relevant to streaming performance. For instance, an object storage containing video could be tested for its performance while serving VoD or live assets. Alternatively, it is worth testing the performance of different cloud instances or new cloud technologies such as server-less computing that have a different resource allocation mechanism.

3.4 Framework Configuration

In this section, we detail some of the configuration options available.

Table 1: Load generator configuration parameters

Variable	Values	Description
mode*	live vod	Set the player load generation mode
play_mode	only_manifest full_playback** random_segments	Requests only the Manifest file Creates playback of all the segments in the timeline Creates playback in available random segments
bitrate	highest_bitrate** lowest_birate random_bitrate	Selection of available bitrates
buffer_size	{0**,...,10}	Applies buffer algorithm
time_shift***	{-4,-3,-2,-1,0**,1}	Move away from the live edge(num. segments)

* Required variable and value to run.
** Default configuration if variable is not set.
*** Only applies if *mode=live* is set.

3.4.1 Configuration of the client: The following input parameters are available to create a load test:

- Type of load generated over time: increasing (ramp-up) or fixed number of users (predefined test-load).
- Ramp-up step-rate: new users added at each step. [1]
- Emulated streaming protocol: HLS & MPEG-DASH. Each user can emulate one or more DASH clients, one or more HLS clients, or both at the same time.
- Emulated media player buffer size: buffer size may affect player's download heuristic.
- HLS and MPEG-DASH distribution for emulated user: weight assignment for DASH or HLS client instantiation.
- Bit-rate adaptation heuristic: bit-rate per segment decisions, according to network conditions, i.e. high or low bit-rate. Additional bit-rate selection algorithms may be easily added.
- Streaming application: live or VoD.

3.4.2 Configuration of the Deployment

The deployment requires manifests that reference segments to be accessible through a URI. For successful deployment, both the URI and the segments must be accessible for the client. The configuration of the deployment is implementation-dependent, given the accessibility requirements.

3.4.3 Distributed configuration for multiple clients

To configure the load testing for a distributed deployment, we provide three examples that consist of two command lines(master & slave). Listing 1 initiates the 'master' Locust node and should be run first. Listing 2 is a 'slave' node and should be run after the 'master' has been initiated, and may be deployed many times at different locations to increase the test scale. The example connects to an origin URI that hosts manifests files and segments with some cloud processing involved. Listing 3 and 4 provides a command line example in the 'master' node to generate a VoD and live load respectively.

[1] https://docs.locust.io/en/stable/running-locust-in-step-load-mode.html.

```
#!/bin/bash
HOST_URL=http://${ORIGIN_HOST} \
  MANIFEST_FILE=tears-of-steel-en.ism/.m3u8 locust \
  -f hls_player.py --master \
  --expect-slaves=${EXPECTED_SLAVES} \
  --step-load --csv=ouput_example
```

Listing 1: Command line example 1 in the master node for a HLS clients.

```
#!/bin/bash
HOST_URL=http://${ORIGIN_HOST} \
  MANIFEST_FILE=tears-of-steel-en.ism/.m3u8 \
  locust  -f hls_player.py --slave \
  --master-host=${MASTER_CLIENT_IP} --step-load
```

Listing 2: Command line example 1 in a slave node for HLS clients.

```
#!/bin/bash
mode=vod bitrate=highest_bitrate \
  HOST_URL=http://${ORIGIN_HOST} \
  MANIFEST_FILE=tears-of-steel-en.ism/.m3u8 locust \
  -f hls_player.py --master \
  --expect-slaves=${EXPECTED_SLAVES} \
  --step-load  --csv=ouput_example
```

Listing 3: Command line example 2 in the master node for a VoD load generation of HLS clients with highest bit-rate selection.

```
#!/bin/bash
mode=live bitrate=highest_bitrate \
  HOST_URL=http://${ORIGIN_HOST} buffer_size=5 \
  MANIFEST_FILE=tears-of-steel-en.isml/.mpd locust \
  -f dash_sequence.py --master
  --expect-slaves=${EXPECTED_SLAVES} --step-load
```

Listing 4: Command line example 3 in the master node for a Live load generation of MPEG-DASH clients with buffer size and highest bit-rate selection.

3.5 Measurable metrics

The 'master' Locust node obtains the following HTTP request-response client-side metrics from each of the clients and makes them available as a graph or for download:

- Total number of accumulated requests
- Total number of accumulated failed requests
- Requests/second during time
- Requests Failed/second during time
- Average, median, min, max response time
- Average content size in bytes
- Number of failures for a particular request time

4 Use-cases and Experiments

4.1 Use-cases

Here are some use-cases that we have also used in our experimentation.

- **How many users can my deployment serve reliably without breaking?** By testing the deployment with the emulator at ramp-up mode, a failure-point can be identified.
- **How does a streaming platform react when many users are joining in short intervals?** The tool can emulate high peak loads in small intervals, and collect data to investigate the status.
- **How does a streaming platform behave over long periods under a normal load (e.g., soak testing)?**. Experiments show extended deployment of the emulator (<24h).

4.2 Experimental results

This section provides experimentation results of the potential load testing use-cases, as specified above. In the following experiments, we used a predefined video streaming setup running on Amazon Web Services EC2 Frankfurt region [20].

4.2.1 Client-side setup. The load generator setup consists of five client nodes running Locust software v0.14.5 [12]. The 'master' Locust node runs in one *t2.large*, and the rest of the four 'slave' nodes run in two separate *t2.large* instances (two nodes per instance). The following use-cases use the same setup but different video streaming load that will be explained in following sections.

4.2.2 Use-case 1: Cloud VoD – a gradual increase of users Figure 3 presents three performance metrics from the testbed setup: i.e. number of emulated users, number of failures of the DASH segment requests, and the average response time of the segment request during that period. The emulated users in this test were increased by ten every 10 seconds until reaching the maximum number of 1000 users. Figure 3 shows the correlation between the number of failures of the segment request to the average response time of each DASH segment request. According to the graph, the maximum number of emulated users before a failure (4XX HTTP status code) was 600 users, and the average response time found before a failure occurred was 600 milliseconds.

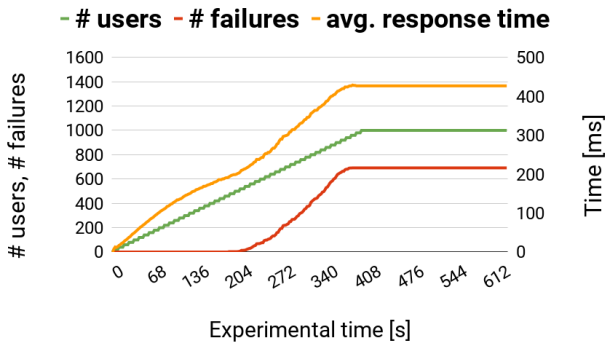

Figure 3: Use-case 1: Cloud VoD – increase of users.

In summary, this first use-case example shows an initial estimation of the maximum emulated users and the minimal response time that this specific VoD setup can handle.

4.2.3 Use-case 2: Users join a video stream abruptly. Figure 4 shows the second example use-case in which users join a VoD stream abruptly from 0 to 800 emulated users. The graph illustrates the correlation between the number of failures of DASH segments requests and their average response time. In comparison to the previous use-case, Figure 4 indicates that after nearly 120 seconds of experimentation time the TCP connections from the Origin server are being closed and errors start to occur. This shows that the deployment in this stage is not resilient to this behavior and further research is needed to improve the deployment.

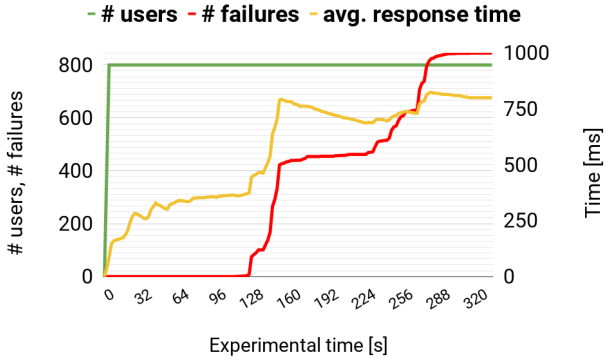

Figure 4: Use-case 2: Users join abruptly a video stream.

Figure 5 and Figure 6 shows the second use-case with lower increase of users compared to previous test. The step increment of users grows from 0 to 200, which leads to a heuristic estimation for the number of users that this VoD setup could handle. In this case, it yields a stable request rate of DASH segments and average response times under 120 milliseconds.

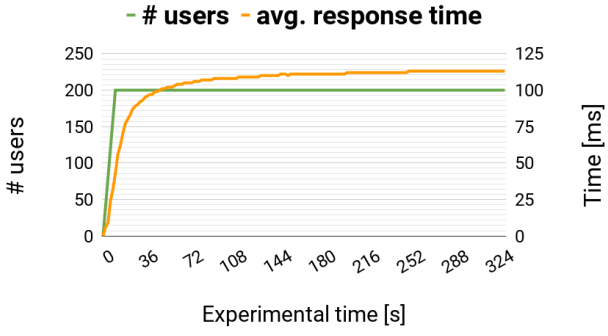

Figure 5: Video streaming setup with 200 users.

4.2.4 Use-case 3: Soak testing of a VoD streaming setup
Figure 7 and Figure 8 presents a third use-case example of a test with an abrupt increase of emulated users from 0 to 700 in the first few seconds. The system ran for 24 hours with a constant load. Nevertheless, the system started to degrade the segment request

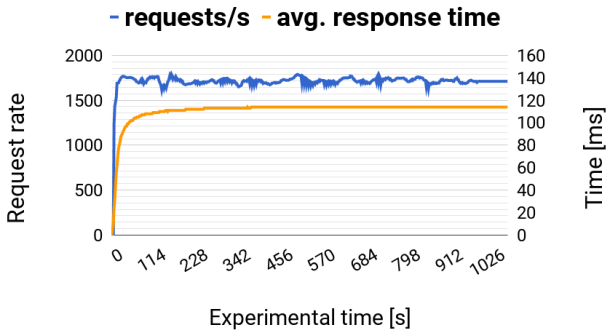

Figure 6: Throughput of the video streaming setup with 200 users.

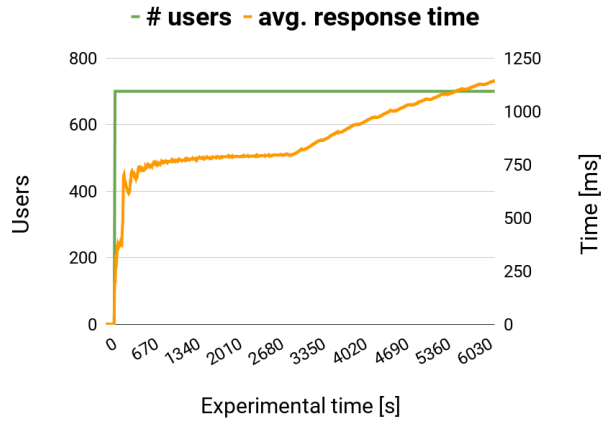

Figure 7: Use-case 3: Soak testing of a VoD streaming setup.

Figure 8: Use-case 3: Soak testing of a VoD streaming setup.

rate approximately 55 minutes after the start time. After this point, the response time of the system increased dramatically from 600 milliseconds to almost 1.2 seconds per request. This shows more investigation of the deployment to mitigate this would be needed.

5 Conclusion

We presented an open-source load testing tool for video streaming. Experiments showed the tool reports key performance metrics that can be used to further optimize and fine-tune video streaming cloud deployments.

References

[1] APACHE. 2019. JMeter. (Last visited: 03/03/2020). https://jmeter.apache.org/
[2] Varsha Apte, T.V.S. Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. 2017. AutoPerf: Automated Load Testing and Resource Usage Profiling of Multi-Tier Internet Applications. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*. ACM Press, L'Aquila, Italy, 115–126. https://doi.org/10.1145/3030207.3030222
[3] Cisco Visual Networking Index. 2019. Forecast and Trends, 2017–2022. *white paper* (Feb. 2019).
[4] Sevil Draxler, Manuel Peuster, Marvin Illian, and Holger Karl. 2018. Generating Resource and Performance Models for Service Function Chains: The Video Streaming Case. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, Montreal, QC, 318–322.
[5] Alexander Frömmgen, Denny Stohr, Boris Koldehofe, and Amr Rizk. 2018. Don't Repeat Yourself: Seamless Execution and Analysis of Extensive Network Experiments. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. 20–26.
[6] Globo. 2012. MIT License. Python m3u8 parser. (Last visited: 03.03.2020). https://github.com/globocom/m3u8
[7] Apple Inc. 2017. HTTP Live Streaming (HLS). (Last visited: 03.03.2020). https://tools.ietf.org/html/rfc8216#page-46
[8] ISO/IEC. 2014. Dynamic adaptive streaming over HTTP (DASH). *International Standard 23009-1:2014* (May 2014).
[9] T. Karagkioules, C. Concolato, D. Tsilimantos, and S. Valentin. 2017. A Comparative Case Study of HTTP Adaptive Streaming Algorithms in Mobile Networks. In *Proc. ACM NOSSDAV*.
[10] Lauri Koskimies, Tarik Taleb, and Miloud Bagaa. 2017. QoE estimation-based server benchmarking for virtual video delivery platform. In *2017 IEEE International Conference on Communications (ICC)*. IEEE, Paris, France, 1–6.
[11] Locust. 2019. Locust: An open source load testing tool. (Last visited: 03/03/2020). https://locust.io/
[12] Locust. 2020. MIT License. Locustio. (Last visited: 15.04.2020). https://docs.locust.io/en/0.14.5/
[13] Rufael Mekuria, Michael J. McGrath, Vincenzo Riccobene, Victor Bayon-Molino, Christos Tselios, John Thomson, and Artem Dobrodub. 2018. Automated profiling of virtualized media processing functions using telemetry and machine learning. In *Proc. ACM MMSys '18*.
[14] D.A. Menasce. 2002. Load testing of Web sites. *IEEE Internet Computing* (2002).
[15] NextTV Latin America Next. 2016. Sport Streaming Mexico. https://en.nextvlatam.com/chivas-tv-compensates-users-latest-service-outage//
[16] R. Pantos and W. May. 2011. HTTP live streaming. *IETF, Informational Internet-Draft 2582* (Sept. 2011).
[17] Ramos. 2020. MIT License. Locust-streaming. (Last visited: 03.03.2020). https://github.com/broyson/Locust-streaming
[18] Sandvine. 2019. Global Internet Phenomena Report. *report* (Sept. 2019).
[19] sangwonl. 2015. MIT License. python-mpegdash parser. (Last visited: 03.03.2020). https://github.com/sangwonl/python-mpegdash
[20] Amazon Web Services. [n. d.]. EC2 instance types. (Last visited: 15.04.2020). https://aws.amazon.com/ec2/instance-types/
[21] Advanced TV Italy Sports. 2018. Sport Streaming Italy. (Last visited: 03/03/2020). https://advanced-television.com/2018/08/20/dazn-sport-italian-launch-streaming-problems/
[22] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. 2012. Methodologies for Generating HTTP Streaming Video Workloads to Evaluate Web Server Performance. In *Proc. SYSTOR (SYSTOR '12)*.
[23] Dimitrios Tsilimantos, Theodoros Karagkioules, and Stefan Valentin. 2018. Classifying Flows and Buffer State for Youtube's HTTP Adaptive Streaming Service in Mobile Networks. In *Proc. ACM MMSys*.
[24] UBIK. 2019. UBIK LOAD PACK VIDEO STREAMING PLUGIN. (Last visited: 03/03/2020). https://www.ubik-ingenierie.com/blog/category/hls/