A Cloud-assisted Tree-based P2P System for Low Latency Streaming

Lucas Provensi, Frank Eliassen, and Roman Vitenberg
Department of Informatics, University of Oslo, Norway.
E-mail: {provensi,frank,romanvi}@ifi.uio.no

***Abstract*—Live media streaming applications are increasingly popular, with services such as Twitch.tv and YouNow being used by millions of people. Deploying such services on the cloud can be very expensive, as the cost is proportional to the amount of data transferred to the users. However, adopting a much less costly peer-to-peer (P2P) solution may reduce the overall quality-of-service (QoS) experienced by users, since there are no guarantees regarding resource availability. Therefore, hybrid P2P/Cloud solutions have been proposed to reduce the cost of using a cloud infrastructure while still providing QoS guarantees. Most existing P2P/cloud streaming solutions apply a pull-based data dissemination mechanism, and use the cloud to ensure that all users receive the data before playback deadline. Although push-based streaming trees can reduce the overall dissemination latency, they have been overlooked in these settings since they are less robust to user churn. In this paper we present a cloud-assisted P2P solution that is self-organizing, robust to user churn and leverages streaming trees to push data to users with low latency. We show through extensive simulations that the proposed solution can reduce playback latency considerably when compared to an alternative pull-based system, without compromising QoS or increasing the cost.**

***Keywords*-Cloud-assisted P2P, Media streaming, Self-organizing systems.**

I. Introduction

In recent years, a new class of media streaming applications have emerged: crowd-sourced live streaming [1]. Examples of such applications are Twitch.tv [1] and YouNow [2], which are services that allow users to broadcast their live video content to a potentially large number of viewers. In these applications, it is important that the average user receives the media content with as low latency as possible, so that the playback at its end does not lag too far behind from the time the content is produced at the broadcaster. This is specially important for live events, such as sports and e-sports matches, where users want to know about the match developments, such as scores, as soon as they happen.

Another characteristic of these applications, it that users can interact with broadcasters by using text chats (integrated in the system, such as in Twitch.tv, or using third-party applications such as Twitter), and the broadcasters can read comments from the chat and respond to them in real-time. The latency of live messaging using a chat application can be orders of magnitude lower than the latency of live media streaming [2], further increasing the need of providing the video content with as low latency as possible.

Commercial applications can achieve large-scale and guarantee quality-of-service (QoS) to the users by deploying their services in the cloud. The cost of leasing the infrastructure is calculated by the provider based on the amount of time, number of requests, amount of data transmitted to the users, etc. For video streaming applications this cost can be very high, and in many systems this cost is passed on to users by displaying advertisements or by enforcing subscription models. To reduce the cost of using an exclusive cloud-based service, recent works have proposed exploring the bandwidth capacity of the user devices at the edge of the network, with hybrid models that combine peer-to-peer (P2P) with cloud computing [3], [4].

There have been works on P2P/Cloud hybrid systems that aim at migrating the streaming service to the cloud and complement it by having a fraction of the bandwidth resources provided by the users, as in CALMS [5]. In these systems, the streaming service is still provided mainly by the cloud, and the cost reduction is proportional to the fraction of user resources the system is configured to use. Other works, such as CLive [6], instead of migrating the service to the cloud, propose a P2P solution assisted by cloud helpers. In these solutions, the cost is further reduced by utilizing more of the users bandwidth resources. However, the QoS can be reduced in this case, since disseminating most of the media through a multi-hop P2P overlay may increase the overall end-to-end latency.

Solutions such as CALMS and CLive utilize pull-based P2P meshes to exchange media chunks among users. Pull-based meshes are more reliable to user churn than the alternative streaming trees, but can increase the playback latency among peers considerably [7]. While there are a few works that use push-based streaming tree overlays in this settings, such as AngelCast [8], these works are not aimed at latency reduction and rely on centralized servers to maintain the tree structure and deal with node dynamics. Centralized solutions, however, do not scale well with the number of users.

In this paper we investigate the use of self-organizing tree structures to reduce the average playback latency in hybrid P2P/Cloud streaming systems. We propose a cloud-assisted solution that pushes the majority of the media content through low-latency steaming trees combined with a

[1] https://www.twitch.tv/
[2] https://www.younow.com/

pull-based mechanism to exchange data among users during temporary tree disconnections. Cloud helpers are used to alleviate the effect of bursts of nodes joining and leaving the network, and also to recover missing content close to the playback deadline. We compare this solution with a conventional cloud-assisted mesh-based approach and show that we can reduce playback latency considerably, without increasing playback discontinuity or the cost of using the cloud infrastructure. We evaluate our solution with extensive simulations, applying user churn traces collected from real Twitch.tv streaming sessions.

The rest of this paper is organized as follows. Section II describes the system model for the problem addressed by this paper. Section III presents our solution for constructing robust streaming trees on top of a P2P overlay. Section IV shows how to improve robustness and guarantee QoS by using a cloud helper. Section V discusses how our solution compares to existing hybrid P2P/Cloud streaming systems. Section VI presents the evaluation of the work through simulations and finally Section VII concludes the paper.

II. System Model

We consider a system composed of set of nodes that can communicate with each other by exchanging messages. This set is denoted as $N = b_s \cup V_s \cup H$, where b_s is a broadcaster that produces a media stream s, V_s is the set of viewers of s and H is the set of cloud helpers, which are not viewers but can be used to help relay s to all the nodes in V_s. Nodes in V_s are usually connected to hosts at the edge of the network, and can join and leave the network at any time, but nodes in H are always available an can be reached by all nodes.

When a node n send a message to a node q, this message is subject to the latency of the network path between n and q, denoted as $LT(n, q)$. If n and q are communicating directly, then we assume that $LT(n, q)$ is equal to half of the round-trip time of sending a package from n to q and receiving it back (including processing time at q and n). If the path between n and q consists of a sequence of nodes $\{n_0, n_1, ..., n_x\}$, where $n_0 = n$ and $n_x = q$, then the latency of the path is calculated as:

$$LT(n, q) = \sum_{i=0}^{x-1} LT(n_i, n_{i+1})$$

The node n also has limited upload bandwidth and, assuming serialized message distribution at constant rate, we denote by $UP(n)$ n's capacity divided by the rate of transmission required to send the stream s, or the number of upload slots that n can use to relay the stream s to other nodes per time unit (we can assume the time unit as *seconds* and the rate as *kbps*). Because of the limited bandwidth capacity, a data chunk c of size k (k is measured as a fraction of a slot and $k \leq UP(n)$) sent from n to q will be also subjected to a transmission delay $TD_c(n, q) = i * k/UP(n)$, where i is the position of c in the output queue of n.

Differently from nodes in V_s, a node $h \in H$ has a very high value on $UP(h)$, and therefore can serve the stream to a larger number of viewers with a low transmission delay. Cloud helpers, however, have a cost associate with them, denoted as $C(H)$, which is proportional to the volume of data transferred from the nodes in H to the rest of network.

Given that a viewer q consumes chunks from its playback buffer at a constant rate, and that the dissemination path from b_s to q contains the sequence of nodes $\{n_0, ..., n_x\}$, where $n_0 = b_s$ and $n_x = q$, the total playback latency of a chunk c at q can be estimated as:

$$PB_c(b_s, q) = BF_c(q) + LT(b_s, q) + \sum_{i=0}^{x-1} TD_c(n_i, n_{i+1})$$

where $BF_c(q)$ is the time that chunk c will stay in q's playback buffer before it is consumed.

In this paper we address the problem of disseminating s to all the viewers in V_s with the help of nodes in H, given that:

- Every viewer $v \in V_s$ wants to continuously receive s without suffering major interruptions of the data stream.
- For each media chunk c of a stream s, v wants to receive c with as low a $PB_c(b_s, v)$ value as possible, and $PB_c(b_s, v)$ should not be higher than a predefined upper bound lt_{bound}.
- In order to reduce the cost $C(H)$, the use of nodes in H should be restricted to ensure uninterrupted playback or to keep playback latency within bounds.

The minimum-delay multicast [9] and minimum cost delay-bounded multicast [10] problems are NP-hard, and difficult to solve even in centralized settings, where a planner have complete knowledge of the network. This is further aggravated in highly dynamic environments, where nodes join and leave the network frequently, making any previous solution found by a centralized planner invalid. We propose a decentralized solution, where the nodes will self-organize into streaming overlays with the goals of uninterrupted streaming, playback latency reduction and cost reduction.

III. P2P Overlay

To solve the problem described in Section II, we propose exploiting the resources of users connected to the edge of the network in addition to the highly-available (but costly) cloud resources. To this end, we build a P2P overlay with two main components: First, in Section III-A we propose the organization of peers in a low-latency streaming tree, used as the foundation to disseminate the stream s to all the nodes in V_s. Then, in Section III-B, we combine the streaming tree with a pull-based mechanism to recover missing media chunks close to playback deadline.

A. Streaming Tree Construction

Streaming trees can provide better end-to-end latencies than pull-based mesh overlays, but are prone to suffer major

playback discontinuity. This happens when nodes leave the network ungracefully, without notifying their parents and children, resulting in broken tree branches. Works such as [11] and [12] mitigate this problem by splitting the stream into a number of slices and disseminating each slice using a different tree. To further improve playback continuity, forward-error correction (FEC) can be combined with parent-disjoint streaming trees [13]. Splitting the stream, however, can only prevent discontinuity if the media can be decoded using incomplete data, and using multiple trees cannot guarantee continuity if several nodes leave the network at once (breaking branches in several trees). We aim at using a single robust streaming tree, which is less complex to build and maintain than multiple path-disjoint trees, and we introduce cloud helpers to guarantee playback continuity.

As a starting point, we use a decentralized protocol for constructing low-latency streaming trees from a previous work [14]. This protocol (referred to as *low-latency protocol* from now on, or *LLP* in short) is built on top of a peer sampling service [15] that provides every node with a partial view of the network (set of neighbors). The neighbor set is composed of a random subset of nodes, but the T-man protocol [16] is used in addition, to provide a subset of nodes ranked on their estimated latency to the stream source (so nodes can quickly find new low-latency parents). The partial view is used by the neighbors to periodically exchange meta-information about the session, such as estimated latency to the source and upload capacity. With this information, nodes can decide from which neighbors they should request the stream and to which neighbors they should provide the stream. Given a node n and its partial view (set of neighbors), at every cycle of the protocol n will:

1) Request s from a neighbor q, if the latency of the path from b_s to n passing through q is the lowest among all known neighbors of n.
2) Accept a request from a node p that has n in its view, if n has available bandwidth capacity or if there is a node i that is currently receiving s from n and $LT(n,p) < LT(n,i)$. In the latter case, n will replace i with p, and notify i so that it will try to find another parent.

Executing *LLP* at every node results in a self-organizing system that will evolve towards latency reduction.

In presence of user churn, nodes disconnected from the tree will suffer playback discontinuity, and will have to quickly find new parents to rejoin the tree. The higher the number of intermediate nodes between source and leaf nodes (tree depth), the higher the chances of disconnected tree branches. There are works, such as Sepidar [17], that aim at constructing minimal depth streaming trees to solve this problem, but these works do not consider the total path latency from the source to the leaf nodes in their solutions. Although there is indication of a relationship between the measured RTT and the offered bandwidth on the end-to-end path between nodes [18], reducing path latency does not necessarily imply reducing the number of intermediate nodes in the streaming tree. Therefore, in order to make the system more tolerant to churn, we have to change the tree construction protocol to prioritize reduced depth, while still aiming at building low-latency paths.

A simple mechanism for estimating the distance of a node to the broadcaster is to send this information down the tree and accumulating the results in intermediate nodes. If the node is the broadcaster, it will report its distance as zero to their children, which in turn will increment it by one and report to their children. The nodes then can use the distance information and their maximum node degree (upload capacity) to self-organize into streaming trees displaying short distances between root and leaf nodes.

To further reduce the chance of disconnected tree branches, interior nodes should be selected based on stability so that their children are less likely to be affected by churn. The age of a node is one metric that can be used to determine if the node is stable or not, as it is expected that nodes with higher age will stay in the session longer [19]. Stable nodes can also be determined by using prediction models [20], or in systems such as Twitch, by using the viewer history for a particular channel as an indication of which users tend to stay longer.

The work in [21] indicates that combining age-based and degree-based (upload capacity) selection will not improve performance considerably. We have observed that this holds true in our experiments as well, and that simply preventing intermittent nodes of climbing up the tree (becoming interior nodes) can reduce playback discontinuity considerably. Using the same notion that nodes with higher age tend to stay longer in the session, the nodes now will only replace their children with older nodes (the ones that have stayed longer in the session). Nodes will report their age as the amount of time that has passed since they joined the network, and it is assumed that all nodes have roughly synchronized times, which can be achieved by using NTP for instance (errors in the order of hundreds of milliseconds are acceptable).

To increase *LLP*'s robustness to churn, we modified it to take into consideration the two points discussed previously. First, prioritize reducing the depth of the tree, while aiming at latency reduction when depth cannot be reduced. Second, incorporate age-based peer selection into the protocol, to reduce the chances of using unstable nodes as interior nodes. Considering $D(b_s, n)$ as the distance between the broadcaster and a node n measured in number of hops and $A(n)$ the age of n, the modified tree construction protocol (referred to as *low-depth protocol* from now on, or *LDP* in short) will now work as follows:

1) A node n will request s from a neighbor q, if $D(b_s, q)$ is the smallest among all known neighbors of n. For the same distance, q will be the one with lowest $LT(b_s, q) + LT(q, n)$. However, n will not request s

from q, if the estimated worst case $PB_c(b_s, n)$ using q is higher than a predefined latency bound lt_{bound}.

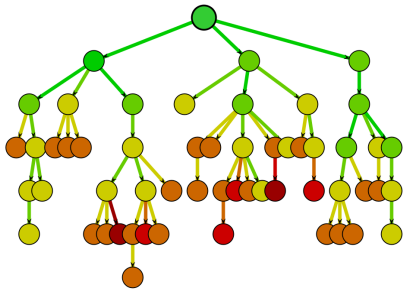
2) Accept a request from a node p, if n has available bandwidth capacity or if there is a node i that is currently receiving s from n and $UP(p) > UP(i)$ and $A(p) > A(i)$. If $UP(p) = UP(i)$, accept only if $A(p) > A(i)$ and $LT(n, p) < LT(n, i)$.

To estimate the worst case $PB_c(b_s, n)$ of a node n, we consider the worst case transmission delay for the path from b_s to n, passing through q. That is, all nodes between b_s and n are using all their upload slots and the chunks reaching n always occupy the last position in the output queues of all nodes. Therefore, given that path latency and playback latency are measured as a fraction of a time unit, the worst case transmission delay will be measured as 1 time unit per hop. Thus, n can estimate the worst case $PB_c(b_s, n)$ through q as $LT(b_s, q) + LT(q, n) + D(b_s, q) + 1$. If no neighbor can provide s with worst case $PB_c(b_s, n)$ lower than lt_{bound}, then the cloud infrastructure will have to be used, as we discuss in Section IV

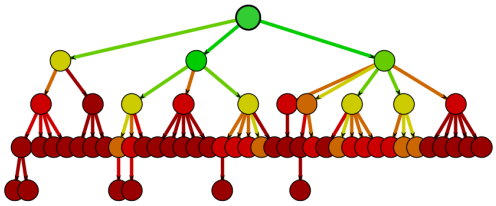
Every round of the protocol, nodes will exchange correct information with their neighbors. For a node n, this information includes $UP(n)$, $D(b_s, n)$, $LT(b_s, n)$ and $A(n)$. In practical settings, malicious nodes can tamper with this information to disrupt the protocol, become free-riders or to try to increase their own quality-of-service [22]. We consider auditing as an orthogonal problem (solutions include reputation systems for P2P [23]), and assume that nodes always send correct information to their neighbors.

Figure 1 shows examples of trees constructed using both protocols. These images where generated by taking a snapshot of a simulation with a network of 57 nodes, with randomly assigned bandwidth capacity and latencies from the Meridian Internet latency data set (further details about simulation settings in Section VI). The nodes are colored according to the accumulated latency of the path starting at the broadcaster, where nodes with low end-to-end latency are colored green and nodes with high end-to-end latency are colored red. The figures show that the tree constructed using *LLP* (Figure 1a) display lower end-to-end latencies from the broadcaster to all viewers. However, it is also deeper and has more intermediate nodes than the one constructed with *LDP* (Figure 1b), and therefore is less resilient to user churn.

Figure 2 shows an example of the amount of data skipped (data not in buffer by the playback time) in a simulation using different variants of the above protocols. For this simulation, a Twitch session dataset was used, with peak number of viewers of 600 and the streaming rate at the source was set to 700 kbps (medium quality broadcast). In this figure, *LDP* refers to the low-depth protocol without selecting stable nodes as interior nodes, *LDP (age)* refers to the same protocol using age to define stable nodes and *LDP (history)* refers to the protocol using the channel history to determine stable nodes. The graph also shows



(a) LLP tree



(b) LDP tree

Figure 1. Streaming trees constructed by using different protocols.

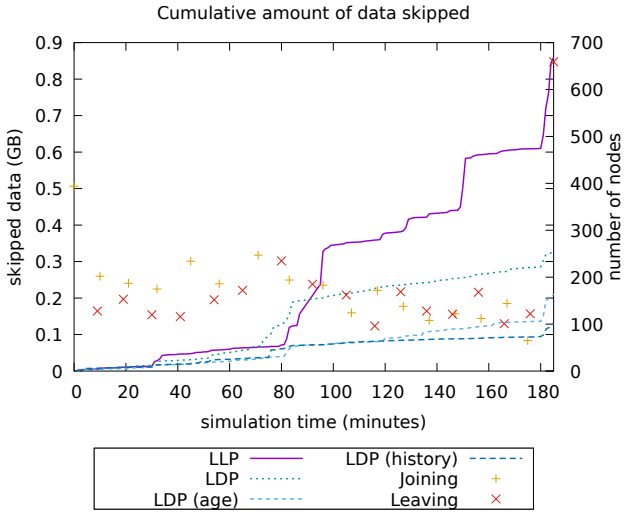


Figure 2. Amount of data skipped using different protocols

the number of nodes joining and leaving over time (vertical axis on the right). In this example, using *LLP* results in the highest amount of skipped data, while using *LDP* reduces the amount of data skipped by 61%. By selecting stable nodes as interior nodes, the amount of skipped data is further reduced, with a 75% decrease with *LDP (age)* and 85% decrease with *LDP (history)*.

Figure 3 shows the CDF of the stay times of the nodes in the Twitch session (Nodes that leave and rejoin later are counted as newly joined nodes). About 20% of the nodes stay in the session for less than 60 seconds, while only about 2% of nodes stay in the session for the whole session duration. By using the age of the nodes to determine

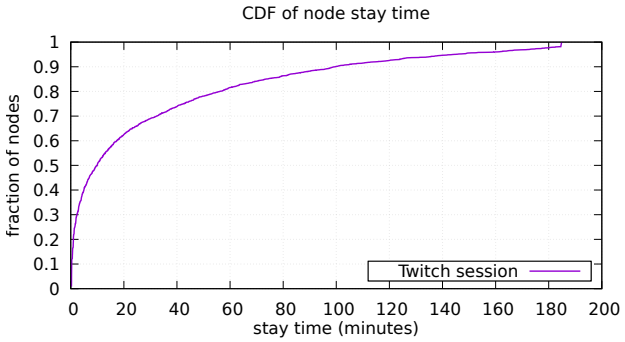


Figure 3. CDF of nodes stay times

stability, children nodes can only be replaced by nodes reporting higher ages, effectively preventing nodes with short stay times (such as the 20% with less than $60s$ stay times) of becoming interior nodes in the tree, and improving the chances of nodes with longer stay times (such as the 2% that stay for the whole session). As seen in Figure 2, this is sufficient to make the performance of *LDP (age)* close to the *LDP (history)* performance. Therefore, because the history of users might not be available in all sessions and to provide a more general solution, we use age comparison in the presented protocol.

B. Pulling missing data

There are hybrid P2P solutions that combine pull-based meshes with push-based trees, such as [24] and [25]. In these solutions, streaming trees are used for low-latency data dissemination while the overlay meshes are used to pull missing data chunks from neighbors. It is possible to use a pulling mechanism if the buffer is sufficiently big so there is enough time to request chunks and receive them before the playback deadline. These solutions can help reduce the number of skipped data, but also introduce higher traffic and control overhead because they require frequent exchange of buffer maps between neighbors [26].

We introduce a pulling mechanism to complement the main streaming tree in *LDP* by adding buffer maps in the same messages used by the protocol to exchange meta-information among neighbors. One problem is that if the exchange is not frequent enough, the number of chunks a peer can retrieve from its neighbors is too small. For instance, a node buffering 5 seconds of media and exchanging buffer maps also every 5 seconds will not find many neighbors to pull data from, as the time to the next exchange approaches, since most of the maps now are too old. To alleviate this problem without increasing the frequency of buffer map exchanges, we introduce a probabilistic pulling mechanism, in which nodes will try to predict which chunks will be in the buffer map from the time it was received to the time of the next buffer exchange.

We assume that a node n that is connected to the tree will continuously receive the data chunks from its parent, unless n or any other node in the path from b_s to n gets disconnected from the tree. As we discussed previously, the chances of disconnection increase with the distance $D(b_s, n)$. A node q that has n in its neighbor list has no guarantee that n will have completely filled its buffer from the time it first send its buffer map to q to the time of the next buffer exchange.

In our solution, we assume binomial probability distribution, where nodes have the same probability p of leaving the network before the next map exchange. For simplicity, we assume that the value of p remains the same throughout the session, and its value can be derived from the average viewer stay time from previous sessions, for instance. The node q, therefore, can decide to pull a chunk from n based on the probability $P_0 = P(X = 0)$ that zero nodes in the path from b_s to n leaves the network before the next map exchange (X is the expected number of nodes to leave the network). To calculate P_0 based on p, the node q will apply the binomial probability function $P_0 = (1 - p)^{D(b_s,n)}$.

With probabilistic pulling, q will first request the missing chunks of data from a node n that reported these chunks in its buffer map. If n did not report the missing chunks, but could have received them during the interval between buffer maps exchanges, q will request the chunks from n with the probability P_0.

Figure 4 shows the amount of data skipped when the pulling mechanism is introduced to *LDP*, for experiments with the same Twitch session from Figures 2 and 3. In this figure, *LDP* refers to the protocol using age comparison to define stable nodes, *LDP (BM)* is the same protocol with the addition of a mechanism to pull missing data from neighbors using buffer maps, and *LDP (BM + P)* introduces probabilistic pulling of chunks in addition to buffer maps. The figure shows that using a pulling mechanism in addition to the streaming tree improves playback continuity considerably, but the improvement is limited by the frequency of buffer maps exchanges. Using probabilistic pulling can improve it further, with the system displaying almost no data loss in this example.

IV. Cloud Assistance

Even if there is enough capacity in the network to carry the stream to all the nodes, it is not possible to guarantee playback continuity in presence of churn, specially during bursts of nodes joining and leaving the network. In order to provide guarantees in an otherwise best-effort P2P system, dedicated servers with high availability can be used to help the peers achieve their goals [27]. In a typical cloud-assisted P2P architecture, one or more servers are leased from a cloud provider and used for reliable storage and content delivery [28]. Using Amazon AWS [3] as an example, the

[3]https://aws.amazon.com/

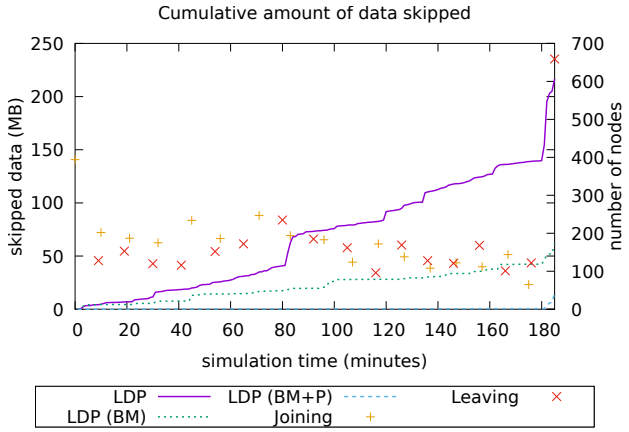


Figure 4. Amount of data skipped using LDP with pulling

source of a live media stream can connect to a media server hosted on Amazon EC2 and use CloudFront as a content delivery (CDN) and caching solution. The cloud architecture is elastic, and will scale up and down based on the demand, and the price payed is proportional to the resources used.

In our solution, we assume that the cloud provider will take care of scaling (in terms of the number of virtual machines), bootstrapping and replication in the edge locations, and focus on how a media server deployed in the cloud can assist the peers. The media server and the cloud infrastructure are abstracted as a single node $h \in H$ accessible to all peers. The broadcaster will use one of its upload slots to send its stream to h, so that the stream is stored (and possibly replicated in edge locations) and forwarded to the peers when needed.

A node n will use the cloud helper h by pulling chunks of media or by requesting the stream, which will then be actively pushed to n. The pulling mechanism will be used when one or more chunks that are close to the playback deadline are missing from n's buffer. n will first pull the chunks from other peers as described in Section III-B, but as the playback deadline approaches, n will pull from h since there is no guarantees that the peers will have the chunks in their buffer (in the case of probabilistic pulling) or that the peers will have available bandwidth to transmit the chunks before the deadline. We use a configurable threshold th_{dl} to define the range of chunks close to playback deadline that will be pulled from h in case they are not in buffer. The minimum value of th_{dl} can be selected based on the round-trip time from n to h, and the capacity of h. Higher values of th_{dl} will, however, increase the cost $C(H)$, as more nodes will pull chunks from h instead of pulling from neighbors.

n will request a stream s from h in the following cases:

1) **Preserve latency bounds**: n cannot find any neighbor to relay s, such that the worst case $PB_c(b_s, n)$ is lower than lt_{bound}. In this case, n will request the stream s from h.
2) **Reduce startup delay**: n requested the stream from its neighbors but could not find a parent: the request got consistently rejected, or no response was received because the neighbors left the network. In this case, to prevent n of waiting too long to start receiving the stream, n will request the stream from h. We define a maximum waiting time before the node request the stream from h.
3) **Reduce churn effect**: n is an interior node and got disconnected from its parent. As a consequence, all the nodes in the sub-tree starting at n will be disconnected and stop receiving the stream. To prevent the disconnection of its children, n will request the stream from h, and migrate its whole sub-tree to the cloud, so that streaming can be resumed as soon as possible.

Using h as a relay node increases the cost $C(H)$, since the cloud provider charges not only for the lease of the infrastructure but also for the amount of data transfered to the peers. To reduce the cost of using the cloud infrastructure, even when n is receiving the stream from h, it will continue to try to find a new parent using its neighbor set. However, n does not want to select a parent that is streaming chunks that are too close to n's playback deadline. Therefore, n will try to find a new parent node q based on $D(b_s, q)$, as nodes higher up in the tree can provide more recent chunks. If n was not connected to any other parent before connecting to h, $D(b_s, q)$ should be close to $D(b_s, h)$. If n was connected to another parent q', then $D(b_s, q)$ should be close to $D(b_s, q')$.

Although the tree construction protocol is aimed at reducing the number of interior nodes, having all disconnected interior nodes request the stream from h (reduce churn effect) may increase $C(H)$. Interior nodes are also closer to b_s than the leaf nodes, so they have less chances of finding new parents using their neighbor set, and will stay connected to h indefinitely. To reduce the number of requests that h receives, a disconnected interior node n will only request s from h if $D(n, q_{max})$ is greater than a depth threshold th_{depth}, where q_{max} is the leaf node in n's sub-tree with maximum depth. In order to estimate the maximum depth of tree branches, every node will send reports up the tree, and intermediate nodes will accumulate the results (maximum depth) before sending the reports up to their parents.

Upon receiving a request from a node, the helper will decide if the request will be served by the cloud infrastructure, or if it should be redirected to another node. The helper will start its operation with a minimum number of upload slots, and will increase or decrease this number based on the demand from the peers. Initially limiting the helper capacity will reduce $C(H)$, as requests will be redirected to peers whenever possible. Adapting the helper capacity based on demand will ensure high playback continuity even when there are bursts of nodes leaving the network, and peers

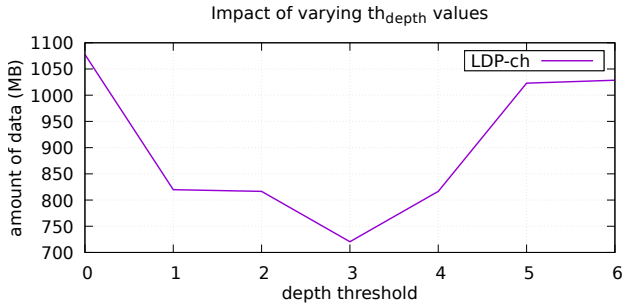


Figure 5. Amount of data retrieved by the nodes from the cloud helper, using varying values of th_{depth}

struggle to find parents.

In our solution, the helper counts the number of requests it receives every cycle of the protocol and if this number is greater than the number of slots, it will increase the number of slots. Otherwise, the helper will decrease the number of slots until it reaches the minimum again. If the helper has used all of its slots, it will redirect the requests to the nodes that are currently receiving the stream from the helper. A cloud helper h will select the node n that reported the lowest $D(n, q_{max})$ value to redirect a request. When n receives a redirection from h, n will accept the request if it has available upload slots, otherwise it will send the request to one of its own children, again selected based on the maximum depth of the children's sub-tree.

The depth threshold th_{depth} discussed previously also has an impact on the number of requests that the helper will receive, and consequently $C(H)$. Figure 5 shows an example of the amount of data retrieved from the cloud helper by the peers with varying th_{depth} values. In the figure, *LDP-ch* refers to the complete *LDP* using a cloud helper. For this experiment we used a shorter Twitch session (28 minutes) than the one used in Section III, but with more users (maximum 1000) and higher churn rate (average 150 peers joining per minute with mean stay time of 4 minutes). If th_{depth} is too small, the number of requests will increase, since the majority of nodes will be in lower levels of the tree and will suffer more disconnections in presence of churn. If th_{depth} is too big, only few interior nodes close to b_s will try to migrate their sub-trees to the cloud, and churn will affect more nodes.

Using the same Twitch session, Figure 6 shows the fraction of nodes receiving more than 99% of playback continuity. In these experiments, we used *LDP* with a cloud helper (*LDP-ch*) and without one (*LDP*). In this example, it was possible to achieve 99% of playback continuity in all nodes by using a cloud helper. Without the helper, the system may struggle to recover from churn events (specially at the end of the session, when there are bursts of users leaving), and playback continuity cannot be guaranteed.

Since the cloud helper is used to prevent playback dis-

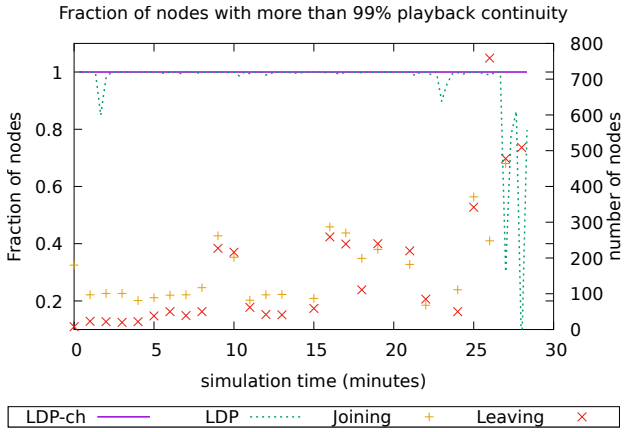


Figure 6. Fraction of nodes receiving 99% playback continuity, with and without cloud helper

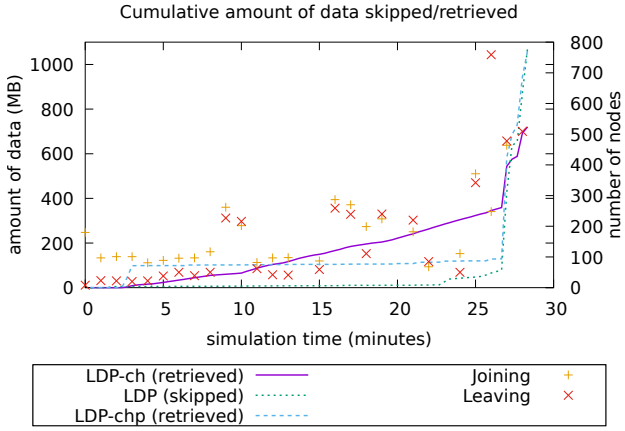


Figure 7. Comparison between the amount of data retrieved from the helper (*LDP-ch*) and the amount data skipped (*LDP*)

continuity, the amount of data transmitted from the cloud infrastructure to the peers might be higher than the amount of data that would actually be skipped from playback if no cloud helper were used. Figure 7 shows how the amount of data retrieved (*LDP-ch*) compares to the amount of data that would have been skipped (*LDP*), for the same session as in Figure 5 and 6. In this figure, *LDP-chp* refers to using the cloud helper only for pulling missing chunks, and not as a relay node (nodes will not request s from h). Although during most of the session the amount of data transferred from the cloud infrastructure is higher than the amount of data that would have been skipped, it will be significantly lower during demanding churn events (bursts of users leaving towards the end of the session). The same is true if the cloud helper is only used for pulling missing chunks.

V. Related Work

CALMS [5] aims at migrating the streaming service to the cloud, and only use the user resources to reduce the

lease cost. Different from our solution, CALMS makes the cloud infrastructure the main component of its architecture, and user capacity is exploited on demand. The P2P layer uses a swarming mechanism, where all the nodes have to report their resources to the cloud and the cloud acts as a centralized planner, providing each node with a set of neighbors to interact with. Unlike CALMS, in our solution the peers self-organize to form a low latency streaming tree.

CPPStreaming [29] is another example of migrating the streaming service to the cloud and uses the P2P layer in a similar way as in CALMS. CPPStreaming also address the problem of how to lease cloud servers in different regions to minimize the latency of P2P swarms. Different from this work, we aim at reducing the latency of the P2P layer, and consider content delivery and replication inside the cloud as an orthogonal problem. CPPStreaming also uses the cloud as a centralized planner to determine the neighbors that each node will use, while we advocate a decentralized solution.

CLive [6] extends a mesh-pull P2P overlay with active and passive cloud helpers. CLive does not prescribe any particular swarming solution, but CoolStreaming [30] chunk selection policy is used in their evaluation, where peers pull the missing chunks with the closest playback time first. CLive uses a decentralized P2P overlay, but unlike our solution, the peers are mainly organized in swarms with pull-based data transfer. Active cloud helpers will push the data to the swarms, which can potentially reduce the overall latency of the system, but once the data is in the swarms, it has to be disseminated with the pulling mechanism (increasing latency).

The work in [31] describes a mesh-pull P2P system assisted by a cloud infrastructure that, unlike CALMS and CLive, is aimed at reduced play-out latency. The main difference is the way the mesh is constructed: Instead of using a random mesh, it uses clusters of highly interconnected nodes. The advantage is that packages are highly available within the clusters, which will aim at homogeneous latencies relative to the media producer. However, clustering is known to reduce robustness of the P2P system in presence of churn [15], and the author does not present any evaluation of this system with demanding churn scenarios.

AngelCast [8] is a hybrid P2P/Cloud system for live streaming that is based on a multi-tree approach. This approach is similar to our in the use of streaming trees in its P2P layer, which can achieve lower end-to-end latencies, however its main focus is on uplink capacity utilization and not latency reduction. AngelCast splits a single stream into multiple disjoint trees (in the same way that SplitStream [11] works). Disseminating data over multiple paths can potentially increase latency, and increases the overhead of maintaining the tree structures. Moreover, the multi-tree structure is orchestrated by a centralized entity with knowledge of all nodes in the system, which can pose scalability challenges.

Table I
TWITCH STREAMING SESSION USED IN THE EVALUATION.

Session	**Duration (min)**	**peak number of nodes**
1	193	359
2	194	788
3	156	1165
4	158	3618
5	178	4264
6	234	5629
7	194	14657

VI. EVALUATION

We evaluate our solution with extensive experiments using the P2P event-driven simulator Peersim [32]. We selected CLive as a baseline to compare against, because as in our solution, it uses a descentrilized P2P overlay and its P2P layer is the main component of the system, assisted by a cloud infrastructure. We implemented CLive in Peersim following the system description in [6].

A. Settings

In the experiments, the underlying Internet topology is initialized using the Meridian Internet latency data set[4], containing 2500 hosts with their latencies. In the simulations, these hosts represent the ISP network that the nodes are connected to. Nodes are assigned hosts at random, and are subject to host-to-host latencies as well as node-to-host latencies. Every node n is assigned a random upload capacity in the range $1 \leq UP(n) \leq 5$ and we assume that every node will contribute at least 1 upload slot, as prerequisite to participate in the system.

To simulate peer dynamics, we collected user data from Twitch.tv live streaming sessions using Twitch API[5]. The API allows us to monitor which channel has a live streaming session, and which users are joining and leaving the chat room of this channel. Although the number of users in the chat room is not necessarily equal to the number of viewers in the channel (viewers might decide to not use the chat), it is representative of users who want to post comments about the stream in real time. Table I shows the sessions we selected for the experiments with their durations and peak number of users. Figure 8 shows the CDF of the node's stay time for all the streaming sessions used in the experiments. The experiments were executed for each of these sessions independently (no concurrent sessions).

We configured one node of the network to be the broadcaster, which will stream at a constant rate of 700 kbps (equivalent to a medium quality streaming session in Twitch). The peers in both systems are initially configured with a small buffer of 5 seconds, and will start playback by consuming data at the same constant rate once the buffer is

[4]http://www.cs.cornell.edu/People/egs/meridian/data.php
[5]https://dev.twitch.tv/

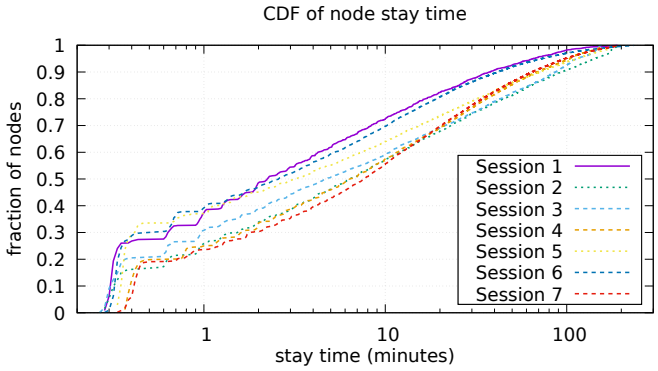


Figure 8. CDF of node stay times

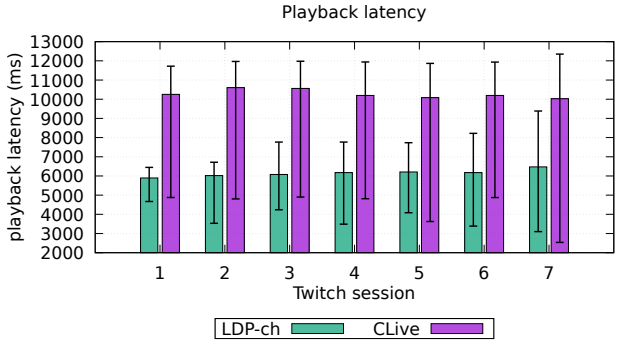


Figure 9. Playback latency of the Twitch sessions in Table I

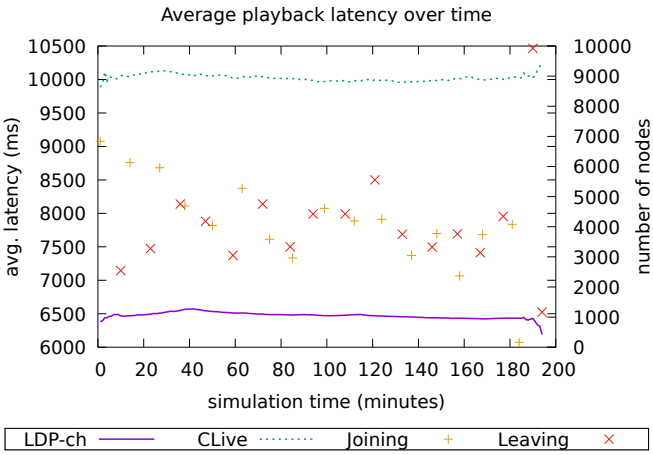


Figure 10. Average playback latency over time for Twitch session 7

filled. Both systems will also pull media chunks from the cloud helper if the chunks are not present in their buffers 2 seconds before playback (this is the th_{dl} value in our solution).

Throughout this section we refer to our cloud-assisted P2P protocol as *LDP-ch*, and it is configured as follows: The upper bound for latency lt_{bound} is 20 seconds, which is also used as upper bound for the CLive implementation. The peer sampling protocol is configured in both *LDP-ch* and CLive to keep a partial view of 30 neighbors selected at random, and runs every 3 seconds (to exchange neighbor sets). In *LDP-ch* buffer maps are exchanged every 3 seconds using the same peer sampling message, while in CLive buffer maps are exchanged every second. The depth threshold th_{depth} is 4, and the probability p of a node leaving before the next map exchange is 20%.

The cloud infrastructure is abstracted as a node h connected to the underlying network topology. This node is known by all users, is always available and has very high upload capacity. We define $UP(h) = 1000$ (In practice, the maximum capacity of a cloud service is in the order of tens of Gbps [6]), which is much higher then a normal peer capacity but the limit will effectively increase the transmission delay from the cloud to the peers when the cloud load increases. We assume that the latency of replicating the chunks received by the media server on the cloud helper is negligible, and all active helpers in CLive are also abstracted in the same helper.

B. Playback Latency

The first metric evaluated is playback latency. We assigned a timestamp to each media chunk generated by the broadcaster node (using the global simulation time), and the time difference is calculated at each node when the chunk is removed from the buffer for playback. Figure 9 shows the mean playback latency values, with average minimum and maximum, for the experiments conducted using the Twitch sessions shown in Table I. In these experiments, the average latencies achieved by *LDP-ch* are always lower than the ones achieved by CLive for all Twitch sessions (decrease of $38 - 42\%$). The use of the cloud helper in both systems ensures that the maximum latency is bounded, and the average maximum never reaches the threshold of 20 seconds.

Figure 10 shows the average playback latency over time in the experiments with session 7 (largest number of nodes). If streaming trees are used without the assistance of a cloud helper, the playback latency would grow unbounded as the number of nodes grows and the tree gets deeper. However, as the graph shows, in *LDP-ch* the average latency is stable throughout the session, and bursts of nodes joining and leaving the network do not have a big impact on the achieved latencies. Although the average latency in CLive is higher than in *LDP-ch*, it is also stable throughout the session.

C. Playback continuity

To measure playback continuity, we keep track of the chunks skipped from playback by each node, and compute the fraction of nodes that received more than 99% of the media content for the duration of the session. Figure 12

[6] http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html

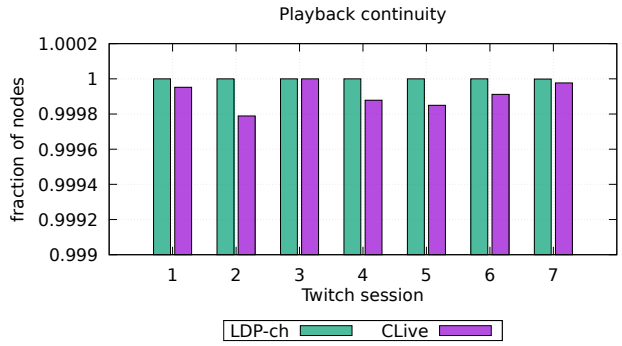


Figure 11. Fraction of nodes with more than 99% playback continuity

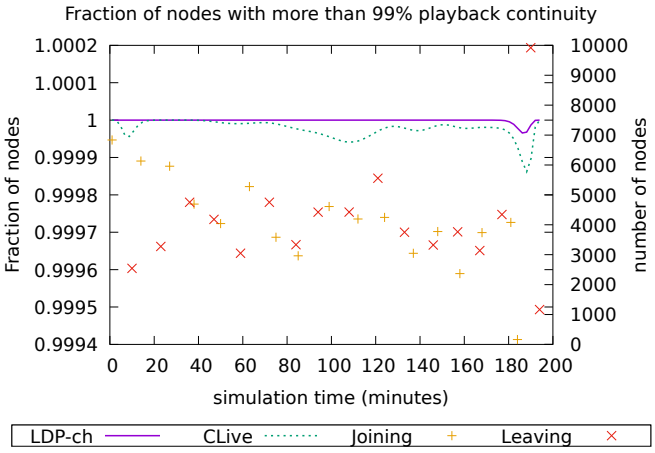


Figure 12. Fraction of nodes with more than 99% playback continuity over time for Twitch session 7

shows this fraction for the all Twitch sessions presented in table I. In both *LDP-ch* and CLive, more then 99.9% of the nodes displayed more than 99% playback continuity. Although the difference is not significant, *LDP-ch* performed better than CLive in most sessions.

One of the main criticism about using trees to disseminate content is that trees are not resilient to bursts of nodes leaving the network [33], since mending the streaming tree is challenging if orphan nodes cannot find new parents. Figure 12 shows how the fraction of nodes with more than 99% playback continuity changes over time in the experiments with the Twitch session 7. Towards the end of the session, when most of the nodes leave the network, both *LDP-ch* and CLive display a drop in continuity. However, *LDP-ch* manages to minimize the effect of churn by migrating broken tree branches to the cloud, so that the time it takes to mend the tree is reduced.

D. Cost

To evaluate the cost of using the cloud infrastructure, we keep track of the amount of data transmitted from the cloud to the peers. Although there are more factors that determine

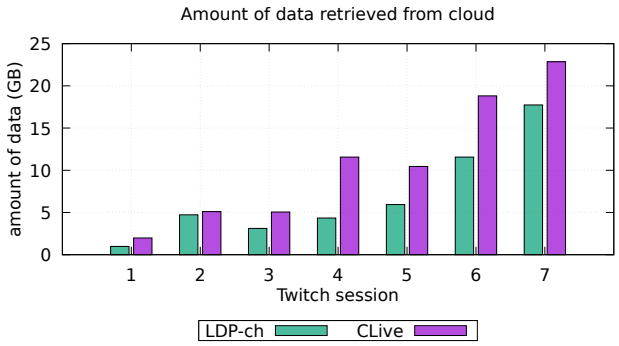


Figure 13. Amount of data retrieved from the cloud

the total cost of using a cloud service (such as number of requests, traffic into the cloud and inter-region traffic), the predominant cost of live video streaming will be based on the generated traffic to the peers. Figure 13 shows the total amount of data that the peers retrieved from the cloud for the Twitch sessions presented in table I. The amount of data is smaller with *LDP-ch* for all sessions, and the difference varies from session to session (as low as 7% and as high 60%).

Using the current pricing of Amazon CloudFront [7], the combined cost of data transfer for all the sessions would be $4.12 with *LDP-ch* and $6.45 with CLive. Even though these are small costs, we are only considering 7 sessions, which are only a small fraction of the number of sessions in popular live streaming applications. In Twitch, for instance, the monthly number of streamers is in the order of millions [8].

Figure 14 shows how the cumulative amount of data retrieved from the cloud changes over time for experiments with the Twitch session 7. Both CLive and *LDP-ch* will increase the number of peers actively served by the cloud helper to prevent the playback latency of growing out of bounds (and to reduce the churn effect and startup delay in *LDP-ch*). Actively streaming from the cloud implies a constant cost (per peer served) in addition to the cost of pulling missing chunks. The figure shows that *LDP-ch* keeps the cost lower than in CLive throughout the session, and there are no major spikes of increase caused by churn bursts.

VII. Conclusion and Future Works

In this paper we presented a cloud-assisted P2P streaming systems that uses trees to disseminate media content with low latency. We evaluated our solution with extensive simulations using churn traces collected from the Twitch.tv system, and we show that the achieved playback latency is significantly lower than in CLive, which is a mesh-based cloud-assisted P2P system. We also show that our approach can mitigate the effects of challenging churn scenarios

[7] https://aws.amazon.com/cloudfront/pricing/
[8] https://www.twitch.tv/p/about

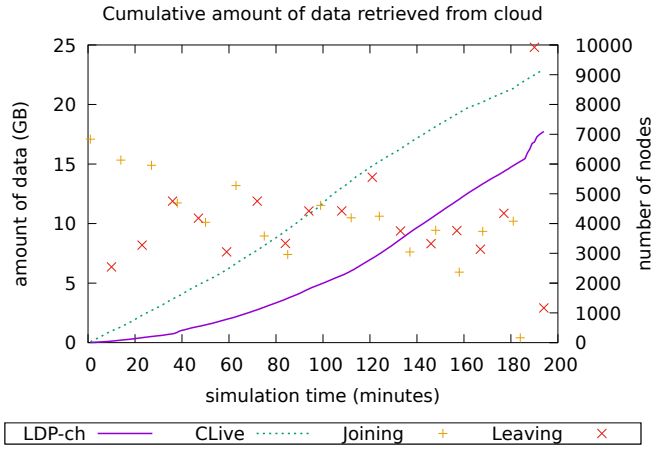


Figure 14. Cumulative amount of data retrieved from the cloud over time for Twitch session 7

on streaming tree structures, resulting in better playback continuity and lower cost when compared to the baseline.

As a future work, we want to explore streaming sessions with more than one source (multiple video feeds for the same event), where users can receive the streams from all sources, or select only a subset of sources to receive from. We also want to extend our solution to take into consideration different encoding qualities for the media being streamed.

Acknowledgments

This research was partially conducted in the framework of the Verdione project funded by the Research Council of Norway under grant 187828.

The authors would like to thank Stephan Reiter for his help developing the Twitch.tv session monitor. We thank Abhishek Singh for helpful discussions and suggestions.

References

[1] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, "Scalable crowd-sourcing of video from mobile devices," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pp. 139–152.

[2] C. Zhang and J. Liu, "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, 2015, pp. 55–60.

[3] I. Trajkovska, J. Salvachua Rodriguez, and A. Mozo Velasco, "A novel p2p and cloud computing hybrid architecture for multimedia streaming with qos cost functions," in *Proceedings of the 18th ACM international conference on Multimedia*. ACM, 2010, pp. 1227–1230.

[4] X. Jin and Y.-K. Kwok, "Cloud assisted p2p media streaming for bandwidth constrained mobile subscribers," in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE, 2010, pp. 800–805.

[5] F. Wang, J. Liu, M. Chen, and H. Wang, "Migration towards cloud-assisted live media streaming," *IEEE/ACM Transactions on networking*, vol. 24, no. 1, pp. 272–282, 2016.

[6] A. H. Payberah, H. Kavalionak, V. Kumaresan, A. Montresor, and S. Haridi, "Clive: Cloud-assisted p2p live streaming," in *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 79–90.

[7] S. Cho, J. Cho, and S.-J. Shin, "Playback latency reduction for internet live video services in cdn-p2p hybrid architecture," in *Communications (ICC), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–5.

[8] R. Sweha, V. Ishakian, and A. Bestavros, "Angelcast: cloud-based peer-assisted live streaming using optimized multi-tree construction," in *Proceedings of the 3rd Multimedia Systems Conference*. ACM, 2012, pp. 191–202.

[9] E. Brosh, A. Levin, and Y. Shavitt, "Approximation and heuristic algorithms for minimum-delay application-layer multicast trees," *IEEE/ACM Transactions on Networking*, vol. 15, no. 2, pp. 473–484, 2007.

[10] N. M. Malouch, Z. Liu, D. Rubenstein, and S. Sahu, "A graph theoretic approach to bounding delay in proxy-assisted, end-system multicast," in *Quality of Service, 2002. Tenth IEEE International Workshop on*. IEEE, 2002, pp. 106–115.

[11] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: high-bandwidth multicast in cooperative environments," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 298–313.

[12] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast," in *ICNP*. IEEE, 2006.

[13] D. Ren, W. Wong, and S.-H. G. Chan, "Toward continuous push-based p2p live streaming," in *Global Communications Conference (GLOBECOM), 2012 IEEE*. IEEE, 2012, pp. 1969–1974.

[14] L. Provensi, F. Eliassen, A. Singh, and R. Vitenberg, "Self-Organizing Media Streaming for Many-to-many Interaction," University of Oslo, Department of Informatics, Tech. Rep. 468, 2017. [Online]. Available: http://urn.nb.no/URN:NBN:no-58225

[15] S. Voulgaris, D. Gavidia, and M. Van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, 2005.

[16] M. Jelasity and O. Babaoglu, "T-man: Gossip-based overlay topology management," *Engineering Self-Organising Systems*, vol. 3910, pp. 1–15, 2005.

[17] A. H. Payberah, F. Rahimian, S. Haridi, and J. Dowling, "Sepidar: Incentivized market-based p2p live-streaming on the gradient overlay network," in *Multimedia (ISM), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[18] M. Mushtaq, T. Ahmed, and D.-E. Meddour, "Adaptive packet video streaming over p2p networks," in *Proceedings of the 1st international conference on Scalable information systems*. ACM, 2006, p. 59.

[19] K. Sripanidkulchai, B. Maggs, and H. Zhang, "An analysis of live streaming workloads on the internet," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 41–54.

[20] F. Wang, J. Liu, and Y. Xiong, "Stable peers: Existence, importance, and application in peer-to-peer live video streaming," in *INFOCOM 2008. The 27th Conference on Computer*

Communications. IEEE. IEEE, 2008, pp. 1364–1372.

[21] M. A. Bishop, S. G. Rao, and K. Sripanidkulchai, "Considering priority in overlay multicast protocols under heterogeneous environments." in *INFOCOM*, 2006, pp. 1–13.

[22] G. Gheorghe, R. L. Cigno, and A. Montresor, "Security and privacy issues in p2p streaming systems: A survey," *Peer-to-Peer Networking and Applications*, vol. 4, no. 2, pp. 75–91, 2011.

[23] S. Marti and H. Garcia-Molina, "Taxonomy of trust: Categorizing p2p reputation systems," *Computer Networks*, vol. 50, no. 4, pp. 472–484, 2006.

[24] F. Wang, Y. Xiong, and J. Liu, "mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast," in *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*. IEEE, 2007, pp. 49–49.

[25] S. Awiphan, Z. Su, and J. Katto, "Tomo: a two-layer mesh/tree structure for live streaming in p2p overlay network," in *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*. IEEE, 2010, pp. 1–5.

[26] F. Picconi and L. Massoulié, "Is there a future for mesh-based live video streaming?" in *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*. IEEE, 2008, pp. 289–298.

[27] R. Sweha, V. Ishakian, and A. Bestavros, "Angels in the cloud: A peer-assisted bulk-synchronous content distribution service," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 97–104.

[28] X. Cong, K. Shuang, S. Su, and F. Yang, "An efficient server bandwidth costs decreased mechanism towards mobile devices in cloud-assisted p2p-vod system," *Peer-to-Peer Networking and Applications*, vol. 7, no. 2, pp. 175–187, 2014.

[29] L. Cui, G. Li, X. Fu, and N. Lu, "Cppstreaming: A cloud-assisted peer-to-peer live streaming system," in *High Performance Computing and Communications (HPCC), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 7–13.

[30] X. Zhang, J. Liu, B. Li, and Y.-S. Yum, "Coolstreaming/donet: A data-driven overlay network for peer-to-peer live media streaming," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 3. IEEE, 2005, pp. 2102–2111.

[31] J. Chakareski, "Cost and profit driven cloud-p2p interaction," *Peer-to-Peer Networking and Applications*, vol. 8, no. 2, pp. 244–259, 2015.

[32] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*. IEEE, 2009, pp. 99–100.

[33] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr, "Chainsaw: Eliminating trees from overlay multicast," in *International Workshop on Peer-to-Peer Systems*. Springer, 2005, pp. 127–140.