Computing

CLive: Hybrid P2P-Cloud Live Streaming

--Manuscript Draft--

Manuscript Number:	
Full Title:	CLive: Hybrid P2P-Cloud Live Streaming
Article Type:	Original Research Article
Corresponding Author:	Amir Payberah SWEDEN
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	
Corresponding Author's Secondary Institution:	
First Author:	Amir Payberah
First Author Secondary Information:	
Order of Authors:	Amir Payberah
	Hanna Kavalionak
	Alberto Montresor
	Seif Haridi
Order of Authors Secondary Information:	
Abstract:	Peer-to-peer (P2P) overlays have lowered the barrier to stream live events over the Internet, and have thus revolutionized the media streaming technology. However, satisfying soft real-time constraints on the delay between the generation of the stream and its actual delivery to users is still a challenging problem. Bottlenecks in the available upload bandwidth, both at the media source and inside the overlay network, may limit the quality of service (QoS) experienced by users. A potential solution for this problem is to assist the P2P streaming network by a cloud computing infrastructure to guarantee a minimum level of QoS. In such approach, rented cloud resources (helpers) are added on demand to the overlay to increase the amount of total available bandwidth and the probability of receiving the video on time. Hence, the problem to be solved becomes minimizing the economical cost, provided that a set of constraints on QoS is satisfied. The main contribution of this paper is CLive, a cloud-assisted P2P live streaming system that demonstrates the feasibility of these ideas. CLive estimates the available capacity in the system through a gossip-based aggregation protocol and provisions the required resources from the cloud to guarantee a given level of QoS at low cost. We perform extensive simulations and evaluate CLive using large-scale experiments under dynamic realistic settings.
Suggested Reviewers:	Mark Jelasity University of Szeged jelasity@inf.u-szeged.hu His research field fits quite well on the subject of this paper.
	Spyros Voulgaris Vrije Universiteit Amsterdam spyros@cs.vu.nl

Noname manuscript No.
(will be inserted by the editor)

CLive: Hybrid P2P-Cloud Live Streaming

Amir H. Payberah · Hanna Kavalionak ·
Alberto Montresor · Seif Haridi

Received: date / Accepted: date

Abstract Peer-to-peer (P2P) overlays have lowered the barrier to stream live events over the Internet, and have thus revolutionized the media streaming technology. However, satisfying soft real-time constraints on the delay between the generation of the stream and its actual delivery to users is still a challenging problem. Bottlenecks in the available upload bandwidth, both at the media source and inside the overlay network, may limit the quality of service (QoS) experienced by users. A potential solution for this problem is to *assist* the P2P streaming network by a cloud computing infrastructure to guarantee a minimum level of QoS. In such approach, rented cloud resources (*helpers*) are added on demand to the overlay to increase the amount of total available bandwidth and the probability of receiving the video on time. Hence, the problem to be solved becomes minimizing the economical cost, provided that a set of constraints on QoS is satisfied. The main contribution of this paper is CLIVE, a cloud-assisted P2P live streaming system that demonstrates the feasibility of these ideas. CLIVE estimates the available capacity in the system through a gossip-based aggregation protocol and provisions the required resources from the cloud to guarantee a given level of QoS at low cost. We perform extensive simulations and evaluate CLIVE using large-scale experiments under dynamic realistic settings.

Keywords P2P Live Streaming · Gossip-based Algorithm · P2P-Cloud Overlay Network

1 Introduction

Nowadays, video streaming of live events is one of the hottest application of the peer-to-peer (P2P) paradigm. Unlike many other P2P services that failed to go beyond the

Amir H. Payberah and Seif Haridi
SICS, Isafjordsgatan 22, Box 1263, SE-164 29 Kista, Sweden
E-mail: {amir,seif}@sics.se
Hanna Kavalionak and Alberto Montresor
Dept. of Information Engineering and Computer Science, University of Trento, Italy
E-mail: {hanna.kavalionak,alberto.montresor}@unitn.it

lab of an university, the strong academic push towards this technology [48,49,35,34, 36] has been quickly paired by large investments from commercial companies [46,1, 47].

The main challenge of a decentralized video streaming service is to strike a good balance between *playback continuity* and *playback delay*. These two properties, which together with the video resolution define the quality of service (QoS) of a video streaming service, are linked by a trade-off: it is possible to increase the playback continuity, i.e., to obtain a smooth media playback, by adopting larger stream buffers, but at the expense of delay. On the other hand, improving playback delay requires that no bottlenecks are present in either the upload bandwidth of the media source and the aggregated upload bandwidth of all nodes in the *swarm*, i.e., the nodes forming the P2P streaming overlay [19].

Increasing the bandwidth at the media source may be difficult, as these systems are exactly designed to cope with situations where the source is an off-the-shelf machine provided with a standard Internet connection. Even when possible, bottlenecks in the swarm have proven to be much more disruptive [20]. The approach proposed in this paper to solve this issue is to add auxiliary *helpers* aimed at accelerating the content propagation. A helper could be an *active* computational node that participates in the streaming protocol, or a *passive* storage service that just provides content on demand. The helpers increase the total upload bandwidth available in the system, thus, potentially reducing the playback delay. Both types of helpers could be rented on demand from a cloud provider, e.g., Amazon AWS, with different prices.

This P2P-cloud hybrid approach is nothing new: a number of P2P content distribution systems [19,22,43,44] have already been pursued under the name of *cloud-assisted* P2P computing. What is novel here is the combination of two different types of helpers, active and passive, that are carefully selected and provisioned with respect to the dynamic behavior of the users. If not enough helpers are present, it is impossible to achieve the desired level of QoS. On the other hand, renting helpers is costly, and their number should be minimized.

The contribution of this paper is to model this problem as an optimization one, where the constraints are given by the desired QoS level with respect to playback continuity and playback delay, while the objective function is to minimize the total economic cost incurred in renting resources from the cloud. We provide CLIVE a decentralized on-line system that is adaptive to dynamic networks.

CLIVE guarantees the QoS at nodes by including a passive helper (PH), e.g., cloud storage, whose task is to provide a last resort for nodes that have not been able to obtain their video chunks through the swarm. Hence, if a node can not receive the required video chunks on time from other nodes in the swarm, it downloads them from the PH. We assume that the media source pushes the new generated video chunks to the PH, so it always contains the latest chunks. However, any interaction with the PH is associated with a cost, and since our objective function is to minimize the cost, we need to reduce the communication with the PH as much as possible.

A solution to reduce the PH cost is to take advantage of active helpers (AH), which are added/removed dynamically on demand. AHs are high upload bandwidth nodes that actively participate in the swarm. The more AHs are added to the system, the more chance nodes have to receive chunks from the swarm, thus, they refer to PH less often, and it consequently reduces the cost of communication with PH. However, the AHs are also costly themselves, therefore, a delicate balance between the amount of video chunks obtained from the PH and the number of AHs in the system must be found. In other word, AHs should be added to the system such that their cost does not exceed the reduced cost caused by less communication with the PH.

To solve this problem, CLIVE exploits a module called CLIVE manager (CM) that uses a gossip-based technique to monitor the state of the swarm, estimate the available upload bandwidth in the system, and based on that computes the proper number of required AHs. To summarized, CLIVE uses PH to guarantee the QoS, i.e., all nodes receive the video chunks on time, and it adds/removes AHs just to reduce the total cost (if possible).

To demonstrate the feasibility of CLIVE, we performed extensive simulations and evaluate our system using large-scale experiments under dynamic realistic settings. We show that we can save up to 45% of the cost by choosing the right number of active helpers compared to only using a passive helper to guarantee the predefined QoS.

2 System model and problem definition

We consider a network consisting of a dynamic collection of *nodes* that communicate through message exchanges. Nodes could be *peers*, i.e., edge computers belonging to users watching the video stream, *helpers*, i.e., computational and storage resources rented from a cloud, and the *media source* (*source* for short) that generates the video stream and starts its dissemination towards peers.

Each peer is uniquely identified by an ID, e.g., composed by IP address and port, required to communicate with it. We use the term *swarm* to refer to the collection of all peers. The swarm forms an *overlay network*, meaning that each peer connects to a subset of peers in the swarm (called *neighbors*). The swarm is highly dynamic: new peers may join at any time, and existing peers may voluntarily leave or crash. Byzantine behavior is not considered in this work. We assume that peers are approximately synchronized; this is a reasonable assumption, given that some cloud services, like Amazon AWS, are already synchronized and sometimes require the client machines to be synchronized as well.

The goal of CLIVE peers is to play the video with predefined *playback delay* (the time between the generation of the video and its visualization at the peer) and *playback continuity* (the percentage of chunks that are correctly streamed to users). A trivial solution to improve the users' viewing experience is to increase the buffering time at end points. The more buffering time users have, the higher is the playback continuity. On the other hand, a large buffering time increases the playback delay, which is not acceptable. Hence, the buffering time should be chosen such that it is long enough to compensate for the late received chunks, but not too long to disrupt the playback delay. Nevertheless, even if the buffering time is chosen properly, it still requires that no bottleneck exists in the swarm upload bandwidth, otherwise again users would not be able to download chunks on time. Since the high buffering time is not a suitable solution, and bottlenecks in the swarm upload bandwidth are very likely to happen [20], we use helpers, rented from the cloud, to guarantee playback continuity at users with a short playback delay.

There are two types of helpers: (i) *active helper* (AH), an autonomous virtual machine composed of one or more computing cores, volatile memory and permanent storage, e.g., Amazon EC2, and (ii) *passive helper* (PH), a simple storage service that can be used to store and retrieve arbitrary pieces of data, e.g., Amazon S3. We assume that customers of the cloud service are required to pay for computing time and bandwidth in the case of AHs, and for storage space, bandwidth and the number of interactions with the PH. This model follows the Amazon's pricing model [39,40].

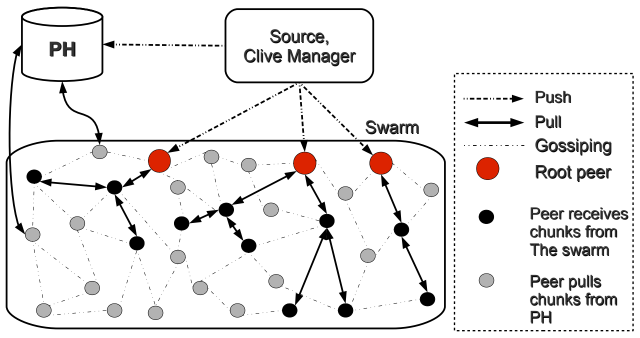
We assume the source generates a constant-rate bitstream and divides it into a number of *chunks*. A chunk c is uniquely identified by the real time $t(c)$ at which is generated. The generation time is used to play chunks in the correct sequence, as they can be retrieved in any order, independently from previous chunks that may or may not have been obtained yet.

Peers, helpers and the source are characterized by different bounds on the amount of available download and upload bandwidth. A peer can create a bounded number of download connections and accept a bounded number of upload connections over which chunks are downloaded and uploaded. We define the number of *download slots* and *upload slots* of a peer as the largest number of download and upload connections that it can sustain, respectively. Thanks to the replication strategies between different data centers currently employed in clouds [50], we assume that PH has an unbounded number of upload slots and can serve as many requests as it receives. Although the PH is logically one service, it could be implemented using a geographically replicated service. We have performed preliminary experiments using Amazon Cloudfront that enabled us to serve chunks from different geographical locations.

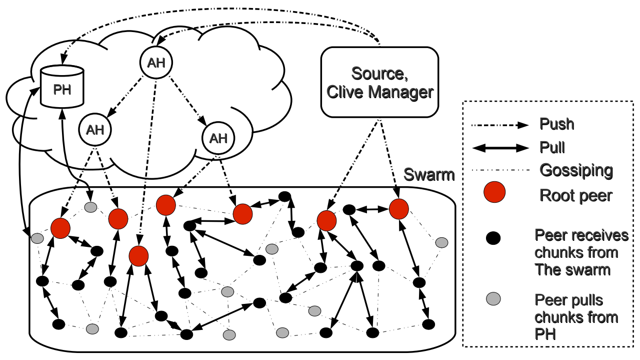
To reach the CLIVE goal, i.e., high playback continuity and short playback delay, CLIVE is allowed to rent PH and/or AH resources from the cloud. We model our problem as an optimization one, where the objective function to be minimized is the total economic cost incurred from renting active helpers and passive helpers from the cloud, subject to the following QoS constraints:

1. the maximum playback delay should be less than or equal to T_{delay}, meaning that if a chunk c is generated at time $t(c)$ at the source, no peers will show it after time $t(c) + T_{delay}$;
2. the maximum percentage of missing chunks should be less than or equal the predefined threshold.

Finding the optimum answer, i.e., lowest cost, of this problem requires a complete and accurate knowledge of the system, which is impossible in a dynamic network, where peers join and fail continuously. Even in a static network, where there is no churn, the optimum solution depends on the overlay structure. However, constructing the swarm overlay is out of the scope of this paper. In Section 4, we present our heuristic to reduce the total cost as much as possible, and in Section 5, we show that in a static and homogeneous network, where we have the accurate knowledge, we can achieve the minimum cost.


PH
Source, Clive Manager
Swarm
Push
Pull
Gossiping
Root peer
Peer receives chunks from The swarm
Peer pulls chunks from PH

(a) The baseline model.


PH
AH
AH
AH
Source, Clive Manager
Swarm
Push
Pull
Gossiping
Root peer
Peer receives chunks from The swarm
Peer pulls chunks from PH

(b) The enhanced model.

Fig. 1 The system architecture

3 System architecture

We present two architectural models, illustrated in Figures 1(a) and 1(b). The *baseline* model (Figure 1(a)) can be described as a P2P streaming protocol, where peers revert to PH whenever a chunk cannot be retrieved from other peers on time. The *enhanced* model (Figure 1(b)) builds upon the baseline, by considering AHs and by providing a distributed mechanism to provision their number and appropriately organizing them.

The main idea behind these models is as follows: to overcome the bottleneck in the swarm upload bandwidth, we rent extra resources from a cloud provider. In the baseline solution, only one PH, i.e., storage, is rented from a cloud provider. In this model, the source pushes chunks to the PH in addition to the swarm. Hence, when a peer cannot receive a chunk of data from other peers in the swarm, it pulls that chunk from the PH. This model guarantees that all peers receive the whole chunks on time. The PH is on a cloud, and clearly we cannot get 100% availability. However, we assume that the PH is always available in our system, because when the cloud fails, there is nothing we can do.

The baseline model could be costly, if the number of requests at the PH increases. A solution to reduce the PH load is to rent AHs in addition to the PH. AHs actively participate in data dissemination with other peers, and thus, reduce the PH usage.

Given that the cost of AH and PH are different, we select the optimal number of AHs that produce the minimum total cost while delivering the desired QoS.

To compute the number of AHs, we use a distributed aggregation protocol to estimate the available upload bandwidth in the system, as well as the number of peers that can receive chunks on time using the existing resources without the help of PH. Then, based on these estimations, CLIVE manager (CM) periodically adds/removes AHs to/from the system in rounds.

To prevent any noise and fluctuation in the number of AHs, only one AH is added/removed to/from the system in each round of the resource management, regardless of the total required AHs. Meanwhile AHs are added/removed, the peers can pull required chunks from the PH. Note that the lack of AHs in the system does not violate the QoS at peers, because although they help in data dissemination, their net effect is on the cost, and the QoS is guaranteed by the PH. When an AH joins the system, it just reduces the total cost by reducing the load on the PH.

3.1 The baseline model

The baseline model can be seen as a P2P streaming service associated by a PH (Figure 1(a)). The idea of augmenting a P2P video streaming application by renting cloud resources is general enough to be applied to several existing video streaming applications. We adopt a *mesh-pull* approach for data dissemination [23], meaning that peers are organized in an unstructured overlay and explicitly ask the missing chunks from their neighbors. Peers discover each other using a gossip-based peer-sampling service [30,29,28]; then, the random *partial views* created by this service can be used by any of the existing algorithms to build the streaming overlay [48,35,52,51].

In a mesh-pull model, neighboring peers exchange their data availability with each other, and the peers use this information to schedule and pull the chunks. There are a number of studies [33,32] on chunk selection policies, but here we use the *in-order* policy, as in COOLSTREAMING [31], where peers pull the chunks with the closest playback time first. The baseline model builds upon this P2P video streaming protocol by adding a PH. The source, apart from pushing newly created video chunks to the swarm, temporarily stores them on the PH.

In order to guarantee a given level of QoS, each peer is required to have a predefined amount of chunks buffered ahead of its playback time, which is called the *last chance window* (LCW), corresponding to a time interval of length T_{lcw}. If a peer p cannot obtain a chunk c from the swarm at least T_{lcw} time units before its playback time, it retrieves c directly from the PH.

3.2 The enhanced model

If the P2P substrate does not suffice, the baseline model represents the easiest solution, but as our experiments will show, this solution could be too expensive, as an excessive number of chunks could end up being retrieved directly from the PH. However, even if the aggregate bandwidth of the swarm may be theoretically sufficient to serve all chunks to all peers, the soft real-time constraints on the playback delay may prevent to exploit entirely such bandwidth. No peer must lag behind beyond a specified threshold, meaning that after a given time, chunks will not be disseminated any more. We need to increase the amount of peers that receive chunks in time, and this could be done by increasing the amount of peers that are served as early as possible. The enhanced model pursues this goal by adding a number of AHs to the swarm (Figure 1(b)).

AHs receive chunks from the source or from other AHs, and push them to other AHs and/or to peers in the swarm. To discover such peers, AHs join the peer sampling protocol [28] and obtain a partial view of the whole system. We use a modified version of CYCLON [28], such that peers exchange their number of upload slots along with their ID. AH chooses a subset of *root peers* (Figure 1(b)) from their partial view and establish a connection to them, pushing chunks as soon as they become available. Root peers of an AH are not changed over time, unless they fail or leave the system, or AH finds a peer with more upload slots than the existing root peers. Clearly, a peer could accept to be a root peer only for one AH, to avoid receiving multiple copies of the same chunk. The net effect of adding AHs is an increase in the number of peers that receive the video stream early in time. The root peers also participate in the P2P streaming protocol, serving a number of peers directly or indirectly. The PH still exists in the enhanced model to provide chunks upon demand, but it will be used less frequently compared to the baseline model.

Architecturally speaking, an important issue is how to organize multiple AHs and how to feed chunks to them. There are two possible models:

- *Flat*: the AHs receive all their chunks directly from the source and then push them to peers in the swarm, acting just as bandwidth multipliers for the source.
- *Hierarchical*: the AHs are organized in a tree with one AH at the root; the source pushes chunks to the root, which pushes them through the tree.

The advantage of the flat model is that few intermediary nodes cause a limited delay between the source and the peers. However, the source bandwidth could end up being entirely consumed to feed the AHs; and more importantly, any communication to the cloud is billed, including the multiple ones from the source to the AHs. We, thus, decided to adopt the hierarchical model, also considering that communication inside the cloud is (i) extremely fast, given the use of gigabit connections, and (ii) free of charge [53].

One important question in the enhanced model is: *how many AHs to add*? Finding the right balance is difficult; too many AHs may reduce the PH load, but cost too much, given that they are billed not only for bandwidth, but also for each hour of activity. Too few AHs increases the PH load as well, and as we show in the experiments, increases the cost. The correct balance dynamically depends on the current number of peers in the swarm, and their upload bandwidth.

The decision on the number of AHs to include in the system is taken by the CLIVE *manager* (CM), a unit that is responsible for monitoring the state of the system and organizing the AHs. By participating in a decentralized aggregation protocol [24], the CM obtains information about the number of peers in the system and the distribution of upload slots among them. Based on this information, it adds new AHs or removes existing ones, trying to minimize the economic cost. The CM role can be played either directly by the source, or by one AH. A detailed description of the CM is provided in the next section.

4 System management

Based on the swarm size and the available upload bandwidth in the swarm, the CM computes the number of AHs that have to be active to minimize the economic cost. Then, depending on the current number of AHs, new AHs may be booted or existing AHs may be shutdown.

The theoretical number of AHs that minimize the cost is not so straightforward to compute, because no node has a global view of the system and its dynamics, e.g., which peers are connected and how many upload slots each peer has. Instead, we describe a heuristic solution, where each peer runs a small collection of gossip-based protocols, with the goal of obtaining approximate aggregate information about the system. The CM joins these gossip protocols as well, and collects the aggregated results. It exploits the collected information to estimate a lower bound on the number of peers that can receive a chunk either directly or indirectly from AHs and the source, but not from PH. We call this set of peers as *infected peers*. The CM, then, uses this information to detect whether the current number of AHs is adequate to the current size of the swarm, or if correcting actions are needed by adding/removing AHs.

In the rest of this section, we first explain how the CM estimates the swarm size and the upload slot distribution, and then we show how it calculates the number of infected peers using the collected information. We also present how the CM manages the number of AHs, based on the swarm size and the number of infected peers, and finally we explain the effect of T_{lcw}, an important system parameter, on the cost and QoS.

4.1 The swarm size and upload slot distribution estimation

All peers in the system, including the CM, participate in the aggregate computation in Algorithm 1, in order to estimate (i) the current size of the swarm, (ii) the probability density function of the upload slots available at peers, and (iii) the T_{lcw} average (Section 4.4).

The size N_{swarm} of the current swarm is computed, with high precision, through the aggregation protocol [24]. On the other hand, knowing the number of upload slots of all peers is infeasible, due to the large scale of the system and its dynamism. However, we can obtain a reasonable approximation of the probability density function of the number of upload slots available at all peers.

Let ω be the actual upload slot distribution among all peers. We adopt ADAM2 [10] to compute $P_\omega : \mathbb{N} \rightarrow \mathbb{R}$, an estimate probability density function of ω. $P_\omega(i)$, then, represents the proportion of peers that have i upload slots w.r.t. the total number of peers, so that $\sum_i P_\omega(i) = 1$. ADAM2 is a gossip-based algorithm that provides an estimation of the cumulative distribution function of a given attribute across all peers.

For our algorithm to work, we assume that each peer is able to estimate its own number of upload slots, and the extreme values of such distribution are known to all and static. Otherwise, a simple mechanism proposed by Haridasan and van Renesse [9] can adjust the set of entries for the case where the extreme values of a variable are unknown. The maximum value is stored in `maxSlot` in Algorithm 1.

Our solution, summarized in Algorithm 1, is based on the gossip paradigm: execution is organized in periodic rounds, performed at roughly the same rate by all peers, during which a push-pull gossip exchange is executed [42]. In a round, each peer p sends a `REQ` message to a peer q, and waits for the corresponding `RES` message from q. Information contained in the exchanged messages are used to update the local knowledge about the entire system, which is composed by the following information:

- a *partial view*, or *view* for short, of the network, stored in variable `subview`, that represents a small subset of the entire population of peers,
- a *slot vector (SV)*, which is used to obtain an approximate and up-to-date data about the attribute distribution,
- a *local value (LV)*, which is used by peers to estimate the network size.

During the rounds, peers (i) update their views about other peers in the system, (ii) estimate the upload slot distribution among peers, and (iii) estimate the swarm size. In the following, we explain how peers update these information:

Updating the views: Partial views are needed to maintain a connected, random overlay topology over the population of all peers to allow the exchange of information. We manage the views through the CYCLON peer sampling service [28]. Each view contains a fixed number of *descriptors*, composed by a peer ID and a timestamp.

During each round, a peer p identifies the peer q with the oldest descriptor in its view, based on the timestamps through `selectOldest`. The corresponding descriptor is removed, and a subset of p's view is extracted through procedure `randomSubset`. This subset is sent to q through a `REQ` message. Peer q that receives the `REQ` message, replies with a `RES` message that similarly contains a number of descriptors randomly selected from its local view.

Whenever p receives a view from q, it merges its own view with the q's one through procedure `mergeView`. In this method, peer p iterates through the received view, and adds the descriptors to its own view. If a peer descriptor to be merged already exists in p's view, p updates its age, if it is newer, otherwise, if p's view is not full, it adds the peer to its view. If p' view is full, p replaces one of the peers it had sent to q with a peer in the received list. The `poll` method returns and removes the last peer from the list.

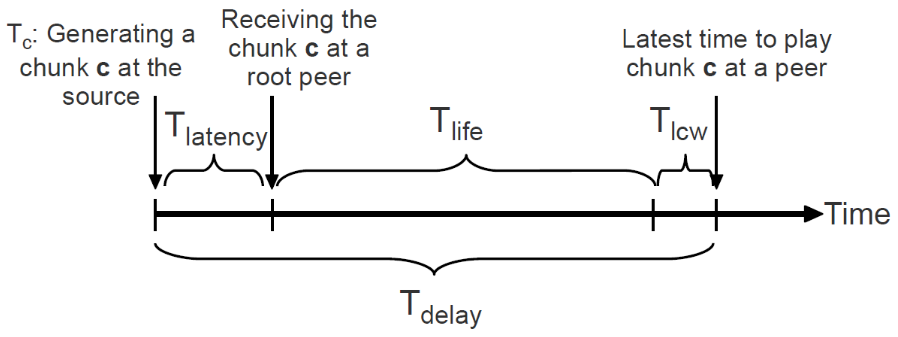
The net effect of this process is the continuous shuffling of views, removing old descriptors belonging to crashed peers and epidemically disseminating new descriptors generated by active ones. The resulting overlay network, where the neighbors of a peer are the peers included in the partial view, closely resembles a random graph, characterized by extreme robustness and small diameter [28].

Estimating the upload slot distribution: Each peer maintains a slot vector SV, which is a vector with `maxSlot` $+ 1$ entries, such that the index of each entry shows the number of slots, i.e., from 0 to `maxSlot`. Initially, all entries of SV at peer p are set

Algorithm 1: Updating the view at nodes and estimating the swarm size, upload slot distribution, and T_{lcw} average.

```
procedure init()
    int $T_{lcw} \leftarrow T_{rtt}$;
    int $slot \leftarrow$ the number of peer's slots;
    int $maxSlot \leftarrow$ maximum number of slots in the system;
    for $i \leftarrow 0$ to $maxSlot$ do
        if $i = this.slot$ then
            $this.SV[i] \leftarrow 1$;
        else
            $this.SV[i] \leftarrow 0$;
    if $this = CM$ then
        $this.LV \leftarrow 1$;
    else
        $this.LV \leftarrow 0$;

procedure round()
    $this.view.updateAge()$;
    $q \leftarrow selectOldest(this.view)$;
    $this.view.remove(q)$;
    $pSubview \leftarrow this.view.randomSubset(this.view)$;
    $pSubview.add(this)$;
    send $\langle \texttt{REQ}, pSubview, this.SV, this.LV, this.T_{lcw} \rangle$ to $q$;

on event receive $\langle \texttt{REQ}, pSubview, pSV, pLV, pT_{lcw} \rangle$ from $p$ do
    $qSubview \leftarrow this.view.randomSubset(this.view)$;
    send $\langle \texttt{RES}, qSubview, this.SV, this.LV, this.T_{lcw} \rangle$ to $p$;
    for $i \leftarrow 0$ to $maxSlot$ do
        $this.SV[i] \leftarrow \frac{this.SV[i]+pSV[i]}{2}$;
    $this.LV \leftarrow \frac{this.LV+pLV}{2}$;
    $this.T_{lcw} \leftarrow \frac{this.T_{lcw}+pT_{lcw}}{2}$;
    $mergeView(this.view, qSubview, pSubview)$;

on event receive $\langle \texttt{RES}, qSubview, qSV, qLV, qT_{lcw} \rangle$ from $q$ do
    for $i \leftarrow 0$ to $maxSlot$ do
        $this.SV[i] \leftarrow \frac{this.SV[i]+qSV[i]}{2}$;
    $this.LV \leftarrow \frac{this.LV+qLV}{2}$;
    $this.T_{lcw} \leftarrow \frac{this.T_{lcw}+qT_{lcw}}{2}$;
    $mergeView(this.view, pSubview, qSubview)$;

procedure $mergeView(view, sentView, receivedView)$
    forall the $n$ in $receivedView$ do
        if $this.view$ contains $n$ then
            $view.updateAge(n)$;
        else if $view$ has free space then
            $view.add(n)$;
        else
            $m \leftarrow sentView.poll()$;
            $view.remove(m)$;
            $view.add(n)$;
```

to 0, except the index equals p's slots, which is set to 1. For example, if $\texttt{maxSlot} = 5$ and the number of p's slots is 2, then p's SV would be $[0, 0, 1, 0, 0, 0]$.



T_c: Generating a chunk c at the source
Receiving the chunk c at a root peer
Latest time to play chunk c at a peer
T_latency
T_life
T_lcw
Time
T_delay

Fig. 2 Live streaming time model.

In each shuffle request, peer p sends its SV values, along with its view. When q receives a REQ message from p, it replies with a message containing its SV. Peer q, then, goes through the received SV and updates its own SV entries to the average of the values for each entry in both SVs, i.e., $q.SV[i] \leftarrow (q.SV[i] + p.SV[i])/2$. Likewise, peer p updates its SV, when it receives RES from q. After a few exchanges, all peers and the CM find the distribution of slots in their own SV, such that entry i in each SV shows the probability of peers with i slots.

Estimating the swarm size: LV is a local float value maintained at peers and at the CM. Initially, it equals zero in all peers, and equals one in the CM. In each shuffle request and reply, peers p and q exchange their LV values, and upon receiving the other peer's value update their local LV to $(q.LV + p.LV)/2$. They, then, compute the swarm size locally as:

$$N_{swarm} = 1/LV \tag{1}$$

4.2 The number of infected peers estimation

The number of peers that can receive a chunk from either the swarm, the source or one of the AHs is bounded by the time available to the dissemination process. This time depends on a collection of system and application parameters:

- T_{delay}: No more than T_{delay} time units must pass between the generation of a chunk at the source and its playback at any of the peers.
- $T_{latency}$: The maximum time needed for a newly generated chunk to reach the *root peers*, i.e., the peers that directly receive the chunks from either the source or the AHs, is equal to $T_{latency}$. While this value may depend on whether a particular root peer is connected to the source or to an AH, we consider it as an upper bound and we assume that the latency added by intermediate AHs is negligible.
- T_{lcw}: If a chunk is not available at a peer T_{lcw} time units before its playback time, it will be retrieved from the PH.

Therefore, a chunk c generated at time $t(c)$ at the source must be played at peers no later than $t(c) + T_{delay}$, otherwise the QoS contract will be violated. Moreover, the chunk c becomes available at root peers by time $t(c) + T_{latency}$, and it should be available in the local buffer of any peer in the swarm by time $t(c) + T_{delay} - T_{lcw}$, otherwise the chunk will be downloaded from the PH (Figure 2). This means that the lifetime T_{life} of a chunk at root peers is equal to:

$$T_{life} = (T_{delay} - T_{latency}) - T_{lcw} \tag{2}$$

Whenever a root peer r receives a chunk c for the first time, it starts disseminating it in the swarm. Biskupski et al. in [41] show that a chunk disseminated by a pull mechanism through a mesh overlay follows a tree-based diffusion pattern. We recursively define the *diffusion tree* $DT(r, c)$ rooted at a root peer r of a chunk c as the set of peers, such that a peer q belongs to $DT(r, c)$, if it has received c from a peer $p \in DT(r, c)$.

Learning the exact diffusion tree for all chunks is difficult, because this would imply a global knowledge of the overlay network and its dynamics, and each chunk may follow a different tree. Fortunately, such precise knowledge is not needed. What we would like to know is an estimate of the number of peers that can be theoretically reached through the source or the current population of AHs.

The chunk generation execution is divided into rounds of length T_{round}. Chunk uploaded at round i becomes available for upload to other peers at round $i + 1$. The maximum depth, $depth$, of any diffusion tree of a chunk over its T_{life} is computed as: $depth = \lfloor T_{life}/T_{round} \rfloor$. We assume that T_{round} is bigger than the average latency among the peers in the swarm. Given $depth$ and the probability density function P_ω, we define the procedure `size`$(P_\omega, depth)$ that executes locally at the CM and provides an estimate of the number of peers of a single diffusion tree (Algorithm 2). This algorithm emulates a large number of diffusion trees, based on the probability density function P_ω, and returns the smallest value obtained in this way.

Emulation of a diffusion tree is obtained by the recursive procedure `recSize`$(P_\omega, depth)$. In this procedure, variable n is initialized to 1, meaning that this peer belongs to the tree. If the depth of the tree is larger than 0, another round of dissemination can be completed. The number of upload slots is drawn randomly by function `random`() from the probability density function P_ω. Variable n is then increased by adding the number of peers that can be reached by recursive call to `recSize`(), where the depth is decremented by 1 at each step before the next recursion.

Algorithm 2: Lower bound for the diffusion tree size.

```
procedure size(DENSITY $P_\omega$, int $depth$)
    int $min \leftarrow +\infty$;
    repeat $k$ times
        $min \leftarrow$ min($min$, recSize($P_\omega$, $depth$));
    return $min$;

procedure recSize(DENSITY $P_\omega$, int $depth$)
    int $n \leftarrow 1$;
    if $depth > 0$ then
        int $slots \leftarrow$ random($P_\omega$);
        for $i \leftarrow 1$ to $slots$ do
            $n \leftarrow n +$ recSize($P_\omega$, $depth - 1$);
    return $n$;
```

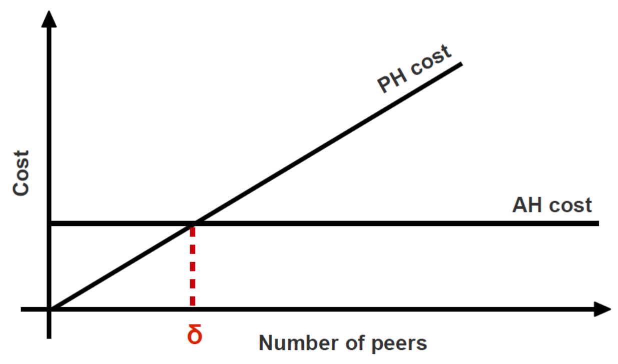


Fig. 3 Calculating the number of peers that is economically reasonable to serve with PH utilization instead of running an additional AH.

At this point, the expected number of infected peers, i.e., the total peers that can receive a chunk directly or indirectly from AHs and the source, but not from the PH, N_{exp}, is given by the total number of root peers times the estimated diffusion tree size, $N_{tree} = \texttt{size}(P_\omega, depth)$. The number of root peers is computed as the sum of the upload slots at the source, $Up(s)$, and AHs, $Up(h)$, minus the number of slots used to push chunks to the AHs themselves, as well as to the PH, which is equal to the number of AHs plus one. Considering $\mathcal{AH}$ as the set of all AHs, we have:

$$N_{exp} = \left(Up(s) + \sum_{h \in \mathcal{AH}} Up(h) - (|\mathcal{AH}| + 1) \right) \cdot N_{tree} \tag{3}$$

4.3 The management model

We define the cost C_{ah} of an AH in one round (T_{round}) as the following:

$$C_{ah} = C_{vm} + m \cdot C_{chunk} \tag{4}$$

where C_{vm} is the cost of running one AH (virtual machine) in a round, C_{chunk} is the cost of transferring one chunk from an AH to a peer, and m in the number of chunks that one AH uploads per round. Since we utilize all the available upload slots of an AH, we can assume that $m = Up(h)$. Similarly, the cost C_{ph} of pulling chunks from PH per round is:

$$C_{ph} = C_{storage} + r \cdot (C_{chunk} + C_{req}) \tag{5}$$

where $C_{storage}$ is the storage cost, C_{req} is the cost of retrieving (GET) one chunk from PH and r is the number of chunks retrieved from PH per round. C_{chunk} of PH is the same as in AH. Moreover, since we store only a few minutes of the live stream in the storage, $C_{storage}$ is negligible.

Figure 3 shows how C_{ah} and C_{ph} (depicted in Formulas 4 and 5) changes in one round (T_{round}), when the number of peers increases. We observe that C_{ph} increases linearly with the number of peers (number of requests), while C_{ah} is constant and independent of the number of peers in the swarm. Therefore, if we find the intersection of the cost functions, i.e., the point δ in Figure 3, we will know when is economically reasonable to add a new AH, instead of putting more load on PH.

$$\delta \approx \frac{C_{vm} + m \cdot C_{chunk}}{C_{chunk} + C_{req}} \tag{6}$$

The CM considers the following thresholds and regulation behavior:

- $N_{swarm} > N_{exp} + \delta$: This means that the number of peers in the swarm is larger than the maximum size that can be served with a given configuration, thus, more AHs should be added to the system.
- $N_{swarm} < N_{exp} + \delta - Up(h) \cdot N_{tree}$: Current configuration is able to serve more peers than the current network size, thus, extra AHs can be removed. $Up(h) \cdot N_{tree}$ shows the number of peers served by one AH.
- $N_{exp} + \delta - Up(h) \cdot N_{tree} \leq N_{swarm} \leq N_{exp} + \delta$: In this interval the system has adequate resource and no change in the configuration is required.

The CM periodically checks the above conditions, and takes the necessary actions, if any. In order to prevent temporary fluctuation, it adds/removes only single AH in each step.

4.4 Discussion

T_{lcw} is a system parameter that has an important impact on the quality of the received media at end users, as well as on the total cost. Finding an appropriate value for T_{lcw} is challenging. With T_{lcw} a too small, peers may fail to fetch chunks from PH in time for playback, while T_{lcw} a too large increases the number of requests to PH, thus, increases the cost. Therefore, the question is how to choose a value for T_{lcw} to achieve (i) the best QoS with a (ii) minimum cost.

4.4.1 Impact of T_{lcw} on the QoS

As we mentioned in Section 3, each peer buffers a number of chunks ahead of its playback time, to guarantee a given level of QoS. The number of buffered chunks corresponds to a time interval of length T_{lcw}. The length of T_{lcw} should be chosen big enough, such that if a chunk is not received through other peers, there is enough time to send a request to PH and retrieve the missing chunk from it in time for playback.

The required time for fetching a chunk from the PH depends on the round trip time (T_{rtt}) between the peer and the PH, and thus it is not the same for all the peers. Therefore, each peer measures T_{rtt} locally, which consists of the latency to send the request to the PH, plus the latency to receive the chunks at the peer's buffer. A peer should send a request for a missing chunk to PH no later than T_{rtt} time units before the playback time, otherwise, the retrieved chunk is useless. Therefore, each peer sets the minimum value of T_{lcw} to T_{rtt}.

While T_{lcw} is a local value at each peer, T_{life}, which is used by the CM to calculate the number of infected peers, depends on T_{lcw} (Equation 2). Therefore, the CM should be aware of the average T_{lcw} among peers. To provide this information to the CM, all peers, including the CM, participate in an aggregation protocol to get the average of T_{lcw} among all peers. Algorithm 1 shows that in each shuffle, a peer sends its local T_{lcw} to other peers, and upon receiving a reply it updates its T_{lcw} to the average of its own T_{lcw} and the received one.

4.4.2 Impact of T_{lcw} on the cost

Equation 5 shows that the cost of PH increases linearly with the number of requests in each round. On the other hand, increasing T_{lcw} increases the PH cost in a round, as peers send more requests to PH. To have a more precise definition of PH cost in Equation 5, we replace r, which is the number of received requests at PH, with $\delta \times l$, where l is the normalized value of T_{lcw} at the CM by the average T_{lcw} (achieved in the aggregation protocol), i.e., $l = \frac{T_{lcw}}{\text{avg}T_{lcw}}$, and δ is the number of peers sending requests to PH in a round. Therefore, Equation 6 can be rewritten as follows:

$$\delta \approx \frac{C_{vm} + m \cdot C_{chunk}}{l \times (C_{chunk} + C_{req})} \tag{7}$$

The CM uses the average T_{lcw} to tune T_{lcw} of the system. If the CM finds out that changing T_{lcw} can decrease the cost, without violating the QoS, it floods the new value of T_{lcw} to the peers. To do that, it sends the new T_{lcw} to the directly connected peers, and each peer forwards it to all its neighbors, expect the one that it receives the message from. In the flooding path, the peers with smaller T_{lcw} than the received one update their local T_{lcw}. However, if T_{lcw} at a peer is bigger than the received value, it does not change it, as its current T_{lcw} is the minimum required time to get a chunk from PH. Note that in Equation 6, we assumed the local CM T_{lcw} equals the aggregated average of T_{lcw}, thus, $l = 1$.

Figure 3 shows the relation between T_{lcw} and PH cost. The higher T_{lcw} is, the steeper the PH cost line is. This means that the cost of PH increases faster when T_{lcw} is large, since PH receives more requests in a shorter time. Moreover, smaller values for T_{lcw} push the PH cost line toward the x-axis. For example, if T_{lcw} is zero, i.e., peers never use the PH, the associated cost is zero and the line overlaps the x-axis. However, we cannot set T_{lcw} to zero, since we use PH as a backup of the chunks to guarantee the promised QoS.

Increasing T_{lcw} not only increases the PH cost, but also increases the total system cost. We see in Section 4.3, that the CM uses two parameters to manage the AHs, (i) the value of δ, and (ii) the number of infected peers. As Equation 7 shows, the higher T_{lcw} is, the smaller δ is. On the other hand, according to Equation 2, increasing T_{lcw} decreases T_{life}, and consequently, decreases the number of infected peers. Hence, increasing T_{lcw}, decreases both δ, and number of infected peers, and as a result the CM adds more AHs to the system, according to the management model in Section 4.3, which increases the total cost.

To summarize, we can say that the best value for T_{lcw} is the aggregated average T_{lcw}, where $l = 1$. Decreasing T_{lcw} below the average T_{lcw}, i.e., $l < 1$, decreases the

QoS at peers, as they may fail to fetch chunks from PH before their playback time. On the other hand, although increasing T_{lcw} may increase QoS, it also increases the cost ($l > 1$). Hence, the CM never broadcasts new value of T_{lcw} to the system.

5 Experiments

In this section, we evaluate the performance of CLIVE using KOMPICS [26], a framework for building P2P protocols that provides a discrete event simulator for testing the protocols using different bandwidth, latency and churn scenarios.

Table 1 Slot distribution in freerider overlay.

Number of slots	Percentage of peers
0	49.3%
1	18.7%
2	8.4%
3-19	5.2%
20	6.8%
Unknown	11.6%

5.1 Experimental setting

In our experimental setup, we set the streaming rate to 500kbps, which is divided into chunks of 20kb; each chunk, thus, corresponds to 0.04s of video stream. We define the *slot* as the unit of transferring the chunks, and based on that we define the upload bandwidth and download bandwidth at peers. The upload bandwidth of a peer is measured as $(\texttt{num of upload slots}) \times (\texttt{slot bandwidth})$. Similarly, we can measure the download bandwidth. For example, if the slot bandwidth is 100kbps, then, the upload bandwidth of a peer with 5 upload slots would be 500kbs.

Without loss of generality, we assume all peers have enough download bandwidth to receive the stream with the correct rate. In these experiments, all peers have 8 download slots, and we consider three classes of upload slot distributions: (i) *homogeneous*, where all peers have 8 upload slots, (ii) *heterogeneous*, where the number of upload slots in peers is picked uniformly at random from 4 to 13, and (iii) *real trace* (Table 1) based on a study of large scale streaming systems [43]. As it is shown in Table 1, around 50% of the peers in this model do not contribute to the data distribution. We set the slot bandwidth to 100kbps in our experiments. The media source is a single node that pushes chunks to 10 other peers. We assume PH has infinite upload bandwidth, and each AH can push chunks to 20 other peers.

There are a number of studies on the impact of chunk size on the playback continuity of peers that show that the smaller the chunk size is, the higher playback

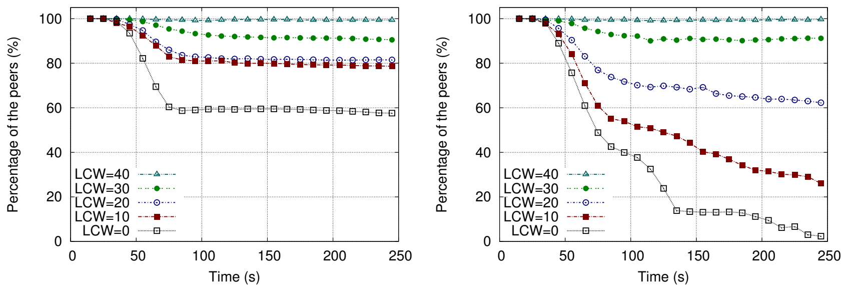


Fig. 4 The percentage of the peers receiving 99% playback continuity with different values of T_{lcw} (measured in number of chunks). Top: join scenario, bottom: churn scenario (1% churn rate).

continuity peers have [36,52]. In our experiments, we use the chunk size similar to the chunk size used in real systems, like BitTorrent, which is 16kb. However, since we assume that the slot bandwidth is 100kbps in our experiments, we chose 20kb instead of 16kb just to get whole (round) numbers after the division. Peers start playing the media after buffering it for 15 seconds, which is comparable to the buffering time in SopCast, the most widely deployed P2P live streaming system [46,57]. T_{delay} also equals 25 seconds. Latencies between peers are modeled using a latency map based on the King data-set [27].

In our experiments, we used two failure scenarios: *join-only* and *churn*. In the join-only scenario, 1000 peers join the system following a Poisson distribution with an average inter-arrival time of 10 milliseconds, and after joining the system they will remain till the end of the simulation. In the churn scenario, approximately 0.01%, 0.1% and 1% of the peers leave the system per second and rejoin immediately as newly initialized peers [45]. However, unless stated otherwise, we did the experiments with 1% churn rate to show how the system performs in presence of high dynamism. A continuous churn of more than 1% is simply unrealistic. Well, already 1% is unrealistic: it means that the average life time of nodes is less than 2 minutes, which means continuous zapping. At that point, no P2P network could help with that.

To the best of our knowledge, CLIVE is the only system that uses both PH and AH from a cloud provider for live streaming and studies their impact on the cost and performance. The existing systems consider only PH, as in CloudCast [22] and CloudMedia [21], or only AHs, as in AngelCast/CloudAngel [55], and they are designed for content distribution or video on demand. However, CLIVE is a live streaming system with real-time constraints on data delivery, which is not considered in the previous systems. Therefore, in this section we compare CLIVE with the baseline model, which is similar to the systems that use only PH.

5.2 The effect of T_{lcw} on system performance

In the first experiment, we evaluate the system behavior with different values for T_{lcw}, measured in number of chunks. In this experiment, we measure *playback continuity* and *playback delay*, which combined together reflect the QoS experienced by

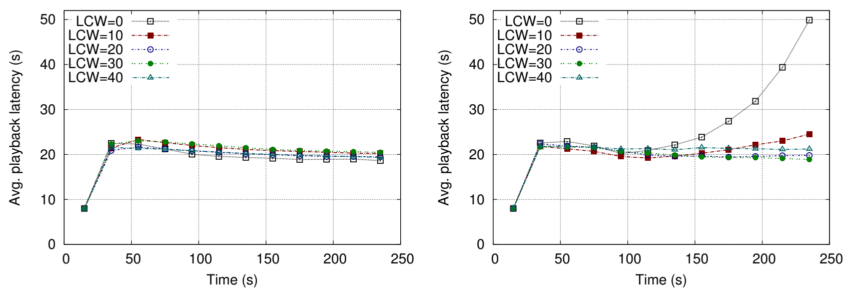
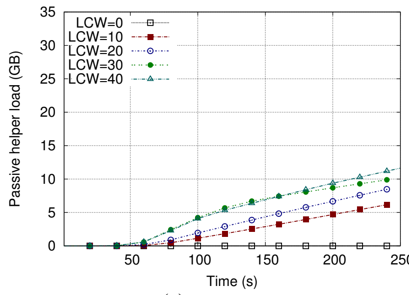
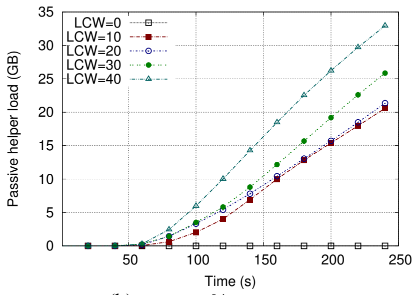


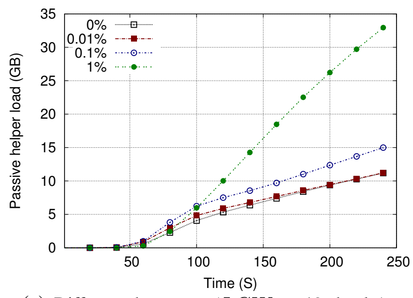
Fig. 5 Average playback delay across peers with different values of T_{lcw} (measured in number of chunks). Top: join scenario, bottom: churn scenario (1% churn rate).



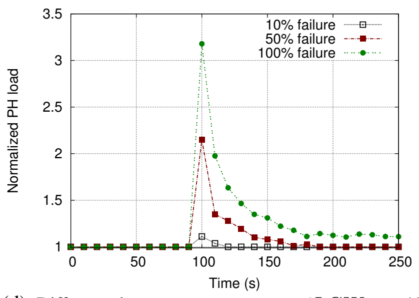
(a) Join-only.



(b) Churn (1% churn rate).



(c) Different churn rates ($LCW = 40$ chunks).



(d) Different churn rates at root peers ($LCW = 40$ chunks).

Fig. 6 The cumulative PH load with different values of T_{lcw} and churn rates.

the overlay peers. Playback continuity shows the percentage of chunks received on time by peers, and playback delay represents the difference, in seconds, between the playback point of a peer and the source.

For a cleaner observation of the effect of T_{lcw}, we use the homogeneous slot distribution in this experiment. Figure 4 shows the fraction of peers that received at least 99% of the chunks before their timeout with different T_{lcw} in the join-only and churn scenarios (1% churn rate). We changed LCW between 0 to 40 chunks, where zero means peers never use PH, and 40 means that a peer retrieves up to chunk $c + 40$

from PH, if the peer is currently playing chunk c. As we see, the bigger T_{lcw} is, the more peers receive chunks in time. Although for any value of $T_{lcw} > 0$ peers try to retrieve the missing chunks from PH, the network latency may not allow to obtain the missing chunk in time. As Figure 4 shows, all the peers retrieve 99% of the chunks on time when $LCW = 40$. Given that each chunk corresponds to 0.04 seconds, $T_{lcw} = 40$ implies 1.6 seconds.

The average playback delay of peers is shown in Figure 5. In the join-only scenario, playback delay does not depend on T_{lcw}, while in the churn scenario we can see a sharp increase when T_{lcw} is small.

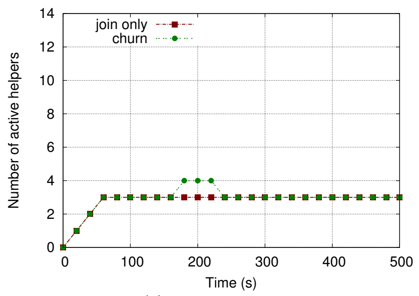
5.3 PH load in different settings

We then measured the PH load or the amount of fetched chunks from PH with different T_{lcw} values and churn rates. Figures 6(a) and 6(b) show the cumulative load of PH in the join-only and churn scenarios (1% churn rate), respectively. As we see in these figures, by increasing T_{lcw}, more requests are sent to PH, thus, increasing its load. Figure 6(c) depicts the cumulative PH load over time for four different churn rates and LCW equals 40 chunks. As the figure shows, there is no big change in PH load under low churn scenarios (0.01% and 0.1%), which are deemed realistic in deployed P2P systems [45]. However, it sharply increases in the presence of higher churn rates (1%), because peers lose their neighbors more often, thus, they cannot pull chunks from the swarm in time, and consequently they have to fetch them from PH.

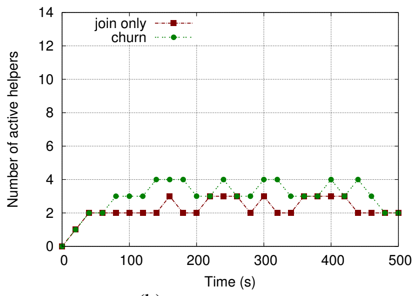
Moreover, we conducted an experiment to show the effect of failure of root peers (Figure 6(d)). We consider three scenarios, where 10%, 50% and 100% of root peers fail. We observed that in these failures, all the peers in the swarm receive chunks without any interruption, because in the worst case, when they lose all their partners, they pull the chunks from the PH. Figure 6(d) shows the PH load in these scenarios. As we see, when 10% and 50% of root peers fail, the PH load increases for a short time, but it goes back quickly to the situation before the failure. However, when all root peers fail, the PH load remains a bit more than the situation before the failure. In this scenario, many peers cannot receive chunks from the swarm at the right speed, thus, the number of buffered chunks at those peers decreases. These peers have shorter time to fetch chunks from other peers afterwards, therefore, they refer to the PH more often, even after resolving the failure.

5.4 Economic cost

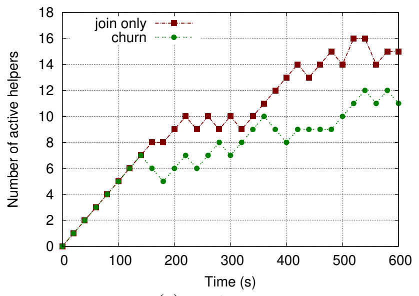
In this experiment, we measure the effect of adding and removing AHs on the total cost. Note, in these experiments we set LCW to 40 chunks, therefore, regardless of the number of AHs, all the peers receive 99% of the chunks before their playback time. In fact, AHs only affect the total cost of the service. In Section 4, we showed how CM estimates the required number of AHs. Figure 7 depicts how the number of AHs changes over time. In the join-only scenario and the homogeneous slot distribution (Figure 7(a)), the CM estimates the exact value of the peers that receive the



(a) Homogeneous.



(b) Heterogeneous.



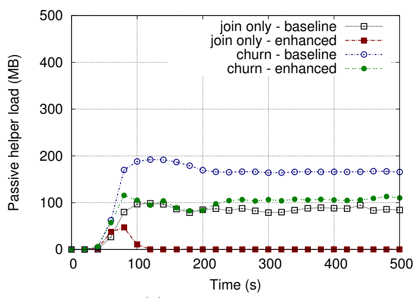
(c) Real trace.

Fig. 7 Number of AHs in different settings and scenarios.

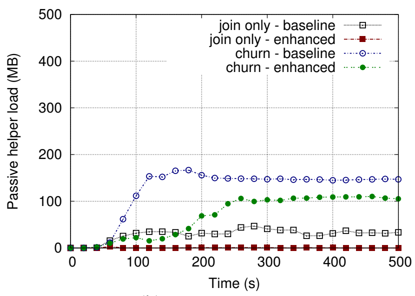
chunks on time using the existing resources in the system, and consequently the exact number of required AHs. Hence, as it is shown, the number of AHs will be fixed during the simulation time. However, in the heterogeneous and real trace slot distributions (Figures 7(b) and 7(c)), CM estimation changes over time, and based on this, it adds and removes AHs. In the churn scenario (1% churn rate), CM estimation also changes over the time, thus, the number of AHs fluctuates.

Relatively, we see how PH load changes in different scenarios in the baseline and enhanced models (Figure 8). Figure 7(a) shows that three AHs are added to the system in the join-only scenario and the homogeneous slot distribution. On the other hand, we see in Figure 8(a), in the join-only scenario, with the help of these three AHs (enhanced model), the load of PH goes down nearly to zero. It implies that three AHs in the system are enough to minimize PH load, while preserving the promised level of QoS. Hence, adding more than three AHs in this setting does not have any benefit and only increases the total cost. Moreover, we can see in the join-only scenario, if there is no AH in the system (baseline model), PH load is much higher than the enhanced model, e.g., around $90mb$, $40mb$, and $130mb$ per second in the homogeneous, heterogeneous, and real trace, respectively. The same difference appears in the churn scenario.

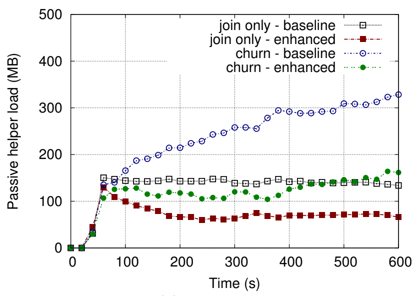
Figure 9 shows the cumulative total cost over the time in different scenarios and slot distributions. In this measurement, we use Amazon S3 as PH and Amazon EC2 as AHs. According to the price list of Amazon [39,40], the data transfer price of S3



(a) Homogeneous.



(b) Heterogeneous.



(c) Real trace.

Fig. 8 PH load in different scenarios with dynamic changes of the number of AHs.

is 0.12$ per GB, for up to 10 TB in a month. The cost of GET requests are 0.01$ per 10000 requests. Similarly, the cost of data transfer in EC2 is 0.12$ per GB, for up to 10 TB in a month, but since the AHs actively push chunks, there is no GET requests cost. The cost of a large instance of EC2 is 0.34$ per hour.

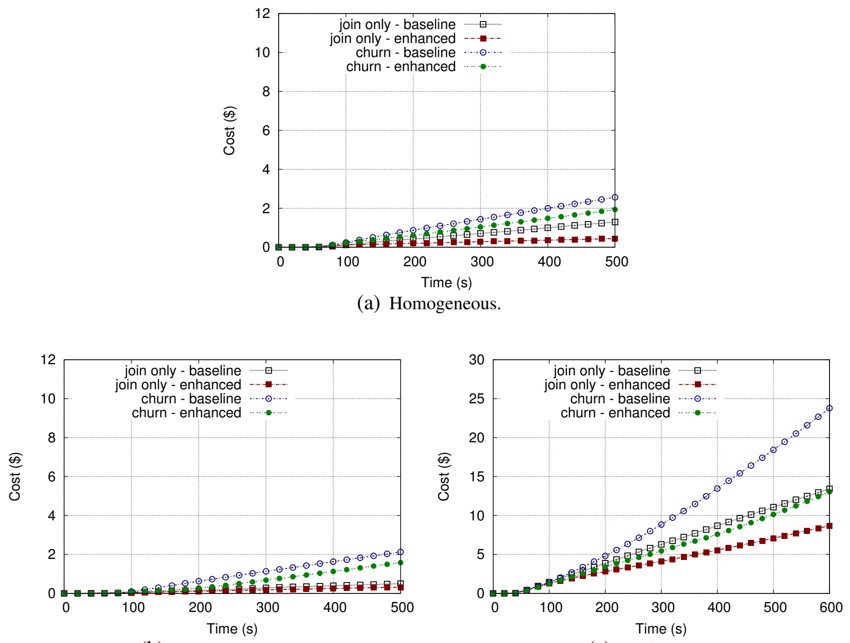
Considering the chunk size of $20kb$ $(0.02mb)$ in our settings, we can measure the cost of PH in Amazon S3 per round (second) according to the Formula 5:

$$\begin{aligned} C_{ph} &\approx r \cdot (C_{chunk} + C_{req}) \\ &\approx \frac{r \times 0.02 \times 0.12}{1000} + \frac{r \times 0.01}{10000} \end{aligned} \tag{8}$$

where r is the the number of received requests by PH in one round (second). The cost of storage is negligible. Given that each AH pushes chunks to 20 peers with the rate of $500kbps$ $(0.5mbps)$, then the cost of running one AHs in Amazon EC2 per second according to Formula 4 is:

$$\begin{aligned} C_{ah} &= C_{vm} + m \cdot C_{chunk} \\ &= \frac{0.34}{3600} + \frac{20 \times 0.5 \times 0.12}{1000} \end{aligned} \tag{9}$$

Figure 9 shows the cumulative total cost for different slot distribution settings. It is clear from these figures that adding AHs to the system reduces the total cost, while keeping the QoS as promised. For example, in the high churn scenario $(1\%$ churn



(a) Homogeneous.

(b) Heterogeneous.

(c) Real trace.

Fig. 9 The cumulative total cost for different setting and scenarios.

rate) and the real trace slot distribution the total cost of system after 600 seconds is $24$$ in the absence of AHs (baseline model), while it is close to $13$$ if AHs are added (enhanced model), which saves around 45% of the cost.

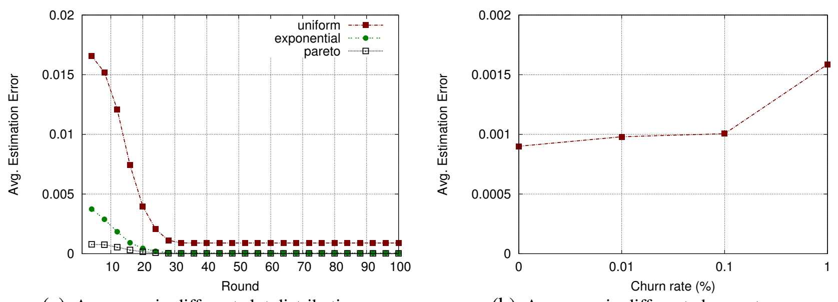
5.5 Accuracy evaluation

In this section we evaluate the accuracy of our estimations in form of evaluating the accuracy of upload slot distribution, and the accuracy of estimating the number of infected peers.

5.5.1 Upload slot distribution estimation

Here, we evaluate the estimation of upload slots distribution in the system. We adopt the Kolmogorov-Smirnov (KS) distance [54], to define the upper bound on the approximation error of any peer in the system. The KS distance is given by the maximum difference between the actual slot distribution, ω, and the estimated slot distribution, $E(\omega)$. We compute $E(\omega)$ based on P_ω for different number of slots. Since the maximum error is determined by a single point (slot) difference between ω and $E(\omega)$, it is sensitive to noise. Hence, we measure the average error at each peer as the average error contributed by all points (slots) in ω and $E(\omega)$. The total average error is then computed as the average of these local average errors.

We consider three slot distributions in this experiment: (i) the uniform distribution, (ii) the exponential distribution ($\lambda = 1.5$), and (iii) the Pareto distribution ($k = 5, x_m = 1$). Figure 10(a) shows the average error in three slot distributions, and Figure 10(b) shows how the accuracy of the estimation changes in different churn rates.



(a) Avg. error in different slot distributions. (b) Avg. error in different churn rates.

Fig. 10 Avg. estimation error.

5.5.2 Number of infected peers estimation

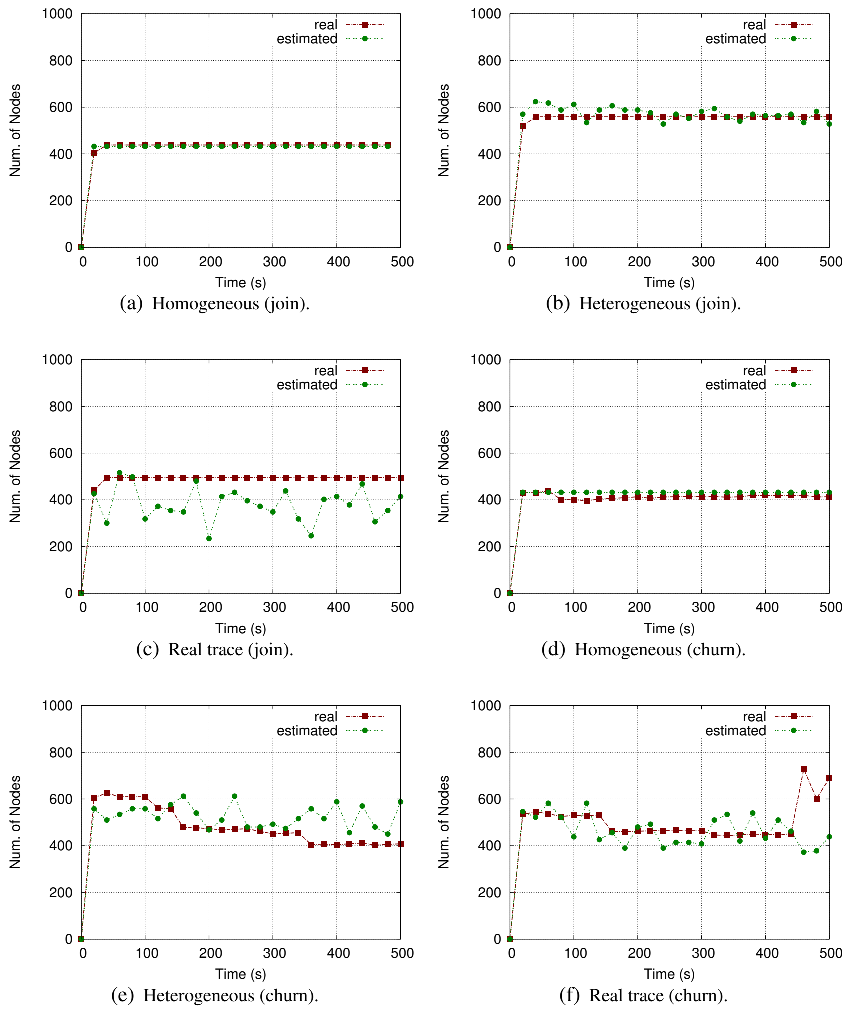
Finally, we evaluate the estimation accuracy of the number of infected peers. Figure 11 shows the real number of infected peers and estimated ones in three upload slot distributions and in join and churn scenarios. As shown in the homogeneous and heterogeneous slot distributions, our estimation of the number of infected peers closely fits the real number of such peer. However, in the real trace slot distribution, it may happen that a peer without upload slot connects directly to the source and prevents other peers to join the system, or on the other hand, a very high upload bandwidth peer joins close to the source and serves many other peers. That is why we see more difference between the real and estimated number of infected peers in the real trace slot distribution.

6 Related work

6.1 Hybrid approaches for content distribution

Although P2P algorithms are emerging as promising solutions for large scale content distribution, they are still subject to a number of challenges [23]. A typical problem is the bottleneck in the aggregated upload bandwidth in the overlay [20] that can lead to a low stream quality with disruptions. The bottleneck is caused by asymmetric bandwidth of the clients: the download bandwidth is usually much higher than upload.

In order to address these issues some recent research works propose to use an hybrid architecture combining Content Delivery Networks (CDN) and P2P overlays [2,



(a) Homogeneous (join). (b) Heterogeneous (join). (c) Real trace (join). (d) Homogeneous (churn). (e) Heterogeneous (churn). (f) Real trace (churn).

Fig. 11 The comparison between the real number of infected nodes and the estimated ones.

3]. Most of these works are focused on reducing the use of CDN servers via utilization of P2P available resources whenever it is possible. This kind of infrastructures is similar with the baseline model considered in this paper. Instead we propose an enhanced architecture that allows to regulate the amount of upload bandwidth in the swarm via the utilization of additional cloud resources.

The most relevant work with respect to ours is LiveSky, developed by Yin et al. [4]. The authors proposed a commercial deployment of the hybrid architecture, addressing several key challenges, including dynamic resource scaling while guaranteeing stream quality. However, while the authors consider redirection of the users according to available upload resources, we propose an approach for managing the amount of available upload bandwidth via swarm feedback processing.

One more possible approach to increase the upload capacity is to use helpers [15, 13]. The helper role can be played by *idle* [14] or *restricted* [12] users. Idle users are peers with spare upload capacity that are not interested in any particular data, while restricted users are users with limited rights to use the network service.

Nowadays, exploiting dedicated cloud servers to accelerate content distribution has become a very popular solution [19,55,56]. In these systems, the role of the cloud servers is to cache and forward content to other peers. ANGELCAST [19,55] is a cloud-based live streaming acceleration service that combines P2P and cloud technologies, where the cloud entities play the role of helpers to build a multi-tree overlay for content distribution. Montresor and Abeni [22] introduced CLOUDCAST as a combination of P2P entities and passive cloud storage to support information diffusion in large scale P2P networks. They used Amazon S3 as dedicated server.

In addition to these solutions, Wu et al. proposed CLOUDMEDIA [21] for video on demand (VoD) systems. The authors describe a queuing model to predict the dynamic demands of the users of a P2P VoD system providing elastic amounts of computing and bandwidth resources on the fly while minimizing the cost. Differently from CLIVE, the CLOUDMEDIA relies on a tracker server that maintains the list of peers with buffered data chunks.

Similarly, Jin et al. presented a cloud-assisted system for P2P media streaming for mobile peers [11]. In this system, mobile users located in the same area share bandwidth resources among each other, while the cloud is responsible for storage and computing demanding tasks.

Unlike all the described solutions, our work exploits cloud computing and storage resources as a collection of active and passive helpers. The combination of both types of helpers together with an effective resource management, distinguishes our approach from previous work.

6.2 Self-monitoring and self-configuration systems

Self-monitoring and *self-configuration* mechanisms are essential to manage large, complex and dynamic systems in an effective way. Self-monitoring detects the current states of system components, while self-configuration is aimed to adapt system configuration according to the received information.

Self-monitoring allows the system to have a view on its current utilization and state. One of the popular approaches for monitoring P2P overlays is decentralized aggregation [24]. For example, ADAM2 [10] presents a gossip-based aggregation protocol to estimate the distribution of attributes across peers. Similarly, Van Renesse and Haridasan [9] propose a distribution estimation mechanism, which can be used to aggregate not only the values of different peers but also how the values are ranked in relation with each others. Another system that uses aggregation is CROUPIER [30], which is a NAT-aware peer sampling service and use aggregation protocol to estimate the ratio of open peers in the network.

Self-configuration is the process of configuring components and protocols autonomously according to specified goals, e.g., reliability and availability [8,5,7,6]. A relevant example of a self-configuration mechanism is proposed by Kavalionak and

Montresor [6] that considers a replicated service on top of a mixed P2P and cloud system. This protocol is able to self-regulate the amount of cloud storage resources utilization according to available P2P resources. However, the main goal of the proposed approach is to support a given level of reliability, whereas in our work we are interested in an effective data dissemination that allows to self-configure the amount of active and passive cloud resources utilization.

7 Conclusions

The main contribution of this paper is CLIVE, a P2P live streaming system that integrates cloud resources (helpers), whenever the peer resources are not enough to guarantee a predefined QoS with a low cost. Two types of helpers are used in CLIVE, active helpers (virtual machines participating in the streaming protocol) and passive helpers (represented by a storage service that provides content on demand). CLIVE estimates the available capacity in the system through a gossip-based aggregation protocol and provisions the required resources (AHs/PHs) from the cloud provider w.r.t the dynamic behavior of the users. Moreover, despite of our previous CLIVE work, here we provide more deep analysis and evaluation of the system parameters.

One of the promising directions to extend the current work is to apply additional cloud computing services into the system architecture.

In particular, Amazon's Cloudfront extends S3 by replicating read-only content to a number of edge locations, in order to put clients closer to the data and reduce the communication latency. Altogether, these edge locations (currently there are 34 of them around the world) can be seen as a unique PH, from where chunks can be pulled. From the point of view of the source, the interface remain the same: content is originally pushed to S3 and from there is replicated across geographically dispersed data centers.

References

1. Roverso, R. and El-Ansary, S. and Haridi, S., SmoothCache: HTTP-Live Streaming Goes Peer-to-Peer, *Proc. of NETWORKING'12, 29–43, 2012, Springer.*
2. Lu, Z.H. and Gao, X.H. and Huang, S.J. and Huang, Y., Scalable and reliable live streaming service through coordinating CDN and P2P, *Proc. of ICPADS'11, 581–588, 2011, IEEE.*
3. Hu, C. and Chen, M. and Xing, C. and Xu, B., EUE principle of resource scheduling for live streaming systems underlying CDN-P2P hybrid architecture, *Peer-to-Peer Networking and Applications, 5(4), 1–11, 2012, Springer.*
4. Yin, Hao and Liu, Xuening and Zhan, Tongyu and Sekar, Vyas and Qiu, Feng and Lin, Chuang and Zhang, Hui and Li, Bo, Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with LiveSky, *Proc. of Multimedia'09, 25–34, 2009, ACM.*
5. Babaoglu, O. and Jelasity, M., Self-* properties through gossiping, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 366(1881), 3747–3757, 2008, The Royal Society.*
6. Kavalionak, H. and Montresor, A, P2P and Cloud: A Marriage of Convenience for Replica Management, *Proc. of IWSOS'12, 60–71, 2012, Springer.*
7. Jelasity, M. and Montresor, A. and Babaoglu, O., T-Man: Gossip-based fast overlay topology construction, *Computer Networks, 53(13), 2321–2339, 2009, Elsevier.*
8. Kephart, J.O. and Chess, D.M., The vision of autonomic computing, *Computer, 36(1), 41–50, 2003, IEEE.*

9. Haridasan, M. and van Renesse, R., Gossip-based distribution estimation in peer-to-peer networks, *Proc. of IPTPS'08, 2008, USENIX.*
10. Sacha, J. and Napper, J. and Stratan, C. and Pierre, G., Adam2: Reliable distribution estimation in decentralised environments, *Proc. of ICDCS'10, 697–707, 2010, IEEE.*
11. Jin, X. and Kwok, Y., Cloud Assisted P2P Media Streaming for Bandwidth Constrained Mobile Subscribers, *Proc. of IPDPS'10, 800–805, 2010, IEEE.*
12. Kumar, R. and Ross, K.W., Optimal peer-assisted file distribution: Single and multi-class problems, *Proc. of HOTWEB'06, 2006, IEEE.*
13. Zhang, H. and Wang, J. and Chen, M. and Ramchandran, K., Scaling peer-to-peer video-on-demand systems using helpers, *Proc. of ICIP'09, 3053–3056, 2009, IEEE.*
14. Wang, J. and Yeo, C. and Prabhakaran, V. and Ramchandran, K., On the role of helpers in peer-to-peer file download systems: Design, analysis and simulation, *Proc. of IPTPS'07, 2007.*
15. Wang, J. and Ramchandran, K., Enhancing peer-to-peer live multicast quality using helpers, *Proc. of ICIP'08, 2300–2303, 2008, IEEE.*
16. Mundinger, J. and Weber, R. and Weiss, G., Optimal scheduling of peer-to-peer file dissemination, *Journal of Scheduling, 11(2), 105–120, 2008, Springer.*
17. , Huang, F. and Khan, M. and Ravindran, B., On minimizing average end-to-end delay in P2P live streaming systems, *Principles of Distributed Systems, 459–474, 2010, Springer.*
18. Ezovski, G.M. and Tang, A. and Andrew, L.L.H., Minimizing average finish time in P2P networks, Proc. of INFOCOM'09, 594–602, 2009, IEEE.
19. Sweha, R. and Ishakian, V. and Bestavros, A., Angels in the cloud: A peer-assisted bulk-synchronous content distribution service, *Proc. of CLOUD'11, 97–104, 2011, IEEE.*
20. Kumar, R. and Ross, K.W., Peer-assisted file distribution: The minimum distribution time, *Proc. of HOTWEB'06, 1–11, 2006, IEEE.*
21. Wu, Y. and Wu, C. and Li, B. and Qiu, X. and Lau, F.C.M., Cloudmedia: When cloud on demand meets video on demand, *Proc. of ICDCS'11, 268–277, 2011, IEEE.*
22. Montresor, A. and Abeni, L., Cloudy Weather for P2P, with a Chance of Gossip, *Proc. of P2P'11, 250–259, 2011, IEEE.*
23. Yiu, W.P.K. and Jin, X. and Chan, S.H.G., Challenges and approaches in large-scale P2P media streaming, *MultiMedia, 14(2), 50–59, 2007, IEEE.*
24. , Jelasity, M. and Montresor, A. and Babaoglu, O., Gossip-based aggregation in large dynamic networks, *ACM Transactions on Computer Systems (TOCS), 23(3), 219–252, 2005, ACM.*
25. Montresor, A. and Ghodsi, A., Towards robust peer counting, *Proc. of P2P'09, 143–146, 2009, IEEE.*
26. Arad, C. and Dowling, J. and Haridi, S., Developing, simulating, and deploying peer-to-peer systems using the Kompics component model, *Proc. of ICST'09, 16, 2009, ACM.*
27. Gummadi, K.P. and Saroiu, S. and Gribble, S.D., King: Estimating latency between arbitrary internet end hosts, *Proc. of SIGCOMM Workshop'02, 5–18, 2002, ACM.*
28. Voulgaris, S. and Gavidia, D. and van Steen, M., CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays, *Journal of Network and Systems Management, 13(2), 197–217, 2005, Springer.*
29. Payberah, A.H. and Dowling, J. and Haridi, S., Gozar: NAT-friendly Peer Sampling with One-Hop Distributed NAT Traversal, *Proc. of DAIS'11, 1–14, 2011, Springer.*
30. Dowling, J. and Payberah, A.H., Shuffling with a Croupier: Nat-aware peer-sampling, *Proc. of ICDCS'12, 102–111, 2012, IEEE.*
31. Zhang, X. and Liu, J. and Li, B. and Yum, Y.S.P., CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming, *Proc. of INFOCOM'05, 2102–2111, 2005, IEEE.*
32. Zhao, B.Q. and Lui, J. and Chiu, D.M., Exploring the optimal chunk selection policy for data-driven P2P streaming systems, *Proc. of P2P'09, 271–280, 2009, IEEE.*
33. Carlsson, N. and Eager, D., Peer-assisted on-demand streaming of stored media using BitTorrent-like protocols, *Proc. of NETWORKING'07, 570–581, 2007, Springer.*
34. Payberah, A.H. and Dowling, J. and Rahimian, F. and Haridi, S., Gradientv: Market-based P2P live media streaming on the gradient overlay, *Proc. of DAIS'10, 212–225, 2010, Springer.*
35. Payberah, A.H. and Dowling, J. and Haridi, S., Glive: The gradient overlay as a market maker for mesh-based P2P live streaming, *Proc. of ISPDC'11, 153–162, 2011, IEEE.*
36. Payberah, A.H. and Rahimian, F. and Haridi, S. and Dowling, J., Sepidar: Incentivized market-based P2P live-streaming on the gradient overlay network, *Proc. of ISM'10, 1–8, 2010, IEEE.*
37. Armbrust, M. and Fox, A. and Griffith, R. and Joseph, A.D. and Katz, R.H. and Konwinski, A. and Lee, G. and Patterson, D.A. and Rabkin, A. and Stoica, I. and Zaharia, M., Above the Clouds: A Berkeley View of Cloud Computing, *EECS Department, University of California, Berkeley, 2009.*

38. Wang, L. and Tao, J. and Kunze, M. and Castellanos, A.C. and Kramer, D. and Karl, W., Scientific cloud computing: Early definition and experience, *Proc. of HPCC'08, 825–830, 2008, IEEE.*
39. Amazon Elastic Compute Cloud (Amazon EC2), *http://aws.amazon.com/ec2/, [Online; accessed 20-Nov-2012].*
40. Amazon Simple Storage Service (Amazon S3), *http://aws.amazon.com/s3/, [Online; accessed 20-Nov-2012].*
41. Biskupski, B. and Schiely, M. and Felber, P. and Meier, R., Tree-based analysis of mesh overlays for peer-to-peer streaming, *Proc. of DAIS'08, 126–139, 2008, Springer.*
42. Jelasity, M. and Voulgaris, S. and Guerraoui, R. and Kermarrec, A.M. and Van Steen, M., Gossip-based peer sampling, *ACM Transactions on Computer Systems (TOCS), 25(3), 8–8, 2007, ACM.*
43. Sripanidkulchai, K. and Ganjam, A. and Maggs, B. and Zhang, H., The feasibility of supporting large-scale live streaming applications with dynamic application end-points, *ACM SIGCOMM Computer Communication Review, 34(4), 107–120, 2004, ACM.*
44. Sripanidkulchai, K. and Maggs, B. and Zhang, H., An analysis of live streaming workloads on the internet, *Proc. of IMC'04, 41–54, 2004, ACM.*
45. Stutzbach, D. and Rejaie, R., Understanding churn in peer-to-peer networks, *Proc. of IMC'06, 25(27), 189–202, 2006.*
46. Lu, Y. and Fallica, B. and Kuipers, F.A. and Kooij, R.E. and Mieghem, P.V., Assessing the quality of experience of Sopcast, *International Journal of Internet Protocol Technology, 4(1), 11–23, 2009, Inderscience Publishers.*
47. Spoto, S. and Gaeta, R. and Grangetto, M. and Sereno, M., Analysis of PPLive through active and passive measurements, *Proc. of IPDPS'09, 1–7, 2009, IEEE.*
48. Fortuna, R. and Leonardi, E. and Mellia, M. and Meo, M. and Traverso, S., QoE in pull based P2P-TV systems: overlay topology design tradeoffs, *Proc. of P2P'10, 1–10, 2010, IEEE.*
49. Frey, D. and Guerraoui, R. and Kermarrec, A.M. and Monod, M., Boosting gossip for live streaming, *Proc. of P2P'10, 1–10, 2010, IEEE.*
50. Goel, S. and Buyya, R., Data replication strategies in wide area distributed systems, *2006, Idea Group Inc., Hershey, PA, USA.*
51. Pai, V. and Kumar, K. and Tamilmani, K. and Sambamurthy, V. and Mohr, A., Chainsaw: Eliminating trees from overlay multicast, *Proc. of IPTPS'05, 127–140, 2005, Springer.*
52. Li, B. and Xie, S. and Qu, Y. and Keung, G.Y. and Lin, C. and Liu, J. and Zhang, X., Inside the new Coolstreaming: Principles, measurements and performance implications, *Proc. of INFOCOM'08, 1031–1039, 2008, IEEE.*
53. Armbrust, M. and Fox, A. and Griffith, R. and Joseph, A.D. and Katz, R. and Konwinski, A. and Lee, G. and Patterson, D. and Rabkin, A. and Stoica, I. and others, A view of cloud computing, *Communications of the ACM, 53(4), 50–58, 2010, ACM.*
54. Schay, G., Introduction to probability with statistical applications, *2007, Birkhäuser.*
55. Sweha, R. and Ishakian, V. and Bestavros, A., AngelCast: cloud-based peer-assisted live streaming using optimized multi-tree construction, *Proc. of MMsys'12, 191–202, 2012, ACM.*
56. Michiardi, P. and Carra, D. and Albanese, F. and Bestavros, A., Peer-assisted content distribution on a budget, *Computer Networks, 56(7), 2038–2048, 2012, Elsevier.*
57. Fallica, B. and Lu, Y. and Kuipers, F. and Kooij, R. and Van Mieghem, P., On the quality of experience of sopcast, *Proc. of NGMAST'08, 501–506, 2008, IEEE.*