LiveShift: Mesh-Pull Live and Time-Shifted P2P Video Streaming

Fabio V. Hecht*, Thomas Bocek*, Richard G. Clegg†, Raul Landa†, David Hausheer*‡, Burkhard Stiller*
* University of Zurich, Department of Informatics (IFI), Zurich, Switzerland
† Department of Electronic and Electrical Engineering, University College London
‡ Currently on leave at Department of EECS, University of California, Berkeley, CA 94720, USA
Email: {hecht,bocek,hausheer,stiller}@ifi.uzh.ch, {rlanda,rclegg}@ee.ucl.ac.uk

***Abstract*—The popularity of video sharing over the Internet has increased significantly. High traffic generated by such applications at the source can be better distributed using a peer-to-peer (P2P) overlay. Unlike most P2P systems, LiveShift combines both live and on-demand video streaming – while video is transmitted through the peer-to-peer network in a live fashion, all peers participate in distributed storage. This adds the ability to replay time-shifted streams from other peers in a distributed and scalable manner. This paper describes an adaptive fully-distributed mesh-pull protocol that supports the envisioned use case and a set of policies that enable efficient usage of resources, discussing interesting trade-offs encountered. User-focused evaluation results, including both channel switching and time shifting behavior, show that the proposed system provides good quality of experience for most users, in terms of infrequent stalling, low playback lag, and a small proportion of skipped blocks in all the scenarios studied, even in presence of churn.**

I. Introduction

The demand for video streaming on the Internet is increasing [10]. Further, the growth in deployment of Fiber to the Home (FTTH) technology has increased upload capacity at the edge, making the peer-to-peer (P2P) content distribution paradigm especially attractive, both to increase scalability and to decrease media publishing costs. While P2P video distribution is quickly becoming a mature technology, with wide-scale deployments already providing video stream distribution through user collaboration, storage and subsequent distribution of content using this same P2P infrastructure remains an open research challenge.

To address this issue, *LiveShift* allows collaboration beyond the current state of the art, enabling peers to store received video streams in order to distribute them in the future, thus allowing time shifting (TS) or – if the combined storage is large – even video-on-demand (VoD) functionality.

A use case of this functionality allows a user, without having previously prepared any local recording, to watch a program from the start and jump over uninteresting parts until seamlessly catching up with the live stream. Watching pre-recorded or live content does not require different protocols and system architectures; LiveShift seamlessly provides these two functionalities. Similarly, live transmission may be used for the premiere of a movie, TV show, or news program, when several users might watch it at the same time. Instantly and automatically, the content is made available – since the starting time of the premiere – to every user joining at a later time. Content providers may also benefit from the proposed approach, since, besides general P2P properties that reduce bandwidth costs at the provider side, moving the storage to the end-user results in content automatically being replicated at a factor proportional to its popularity and distributed to locations where it is popular. The protocol does not mandate particular architectural requirements, as it can be deployed on computers, set top boxes, or servers provided by ISPs or CDN operators; providers may include large as well as small home broadcasters.

The extra challenges that the proposed functionality introduces are several, but two are distinguished as the main drivers of our protocol. First, and in stark difference with a live video streaming system, users may switch not only between channels, but also within various time positions in a given channel, over a potentially large time scale. Second, the asymmetry of interest inherent in such a scenario demands a flexible protocol based on policies that do not require that peers be simultaneously interested in data each other has.

The functionality of LiveShift for real-time streaming was presented as a demonstration in [9]. The two main contributions of this paper are a) to propose a new, fully-distributed, P2P streaming, mesh-pull protocol, which is suitable both for live streaming *and VoD*, and b) to propose, discuss, and analyze initial policies to be used with the protocol. While this paper does not claim that the presented policies are optimal, it introduces the main challenges and trade-offs involved when designing those in a system in which live and on-demand streaming are indistinguishable. Protocol and policies are evaluated using traces from a real IPTV system to model peer behavior, including channel switching. Results are measured in terms of quality of experience (QoE) metrics, such as playback lag, that are important for the end user. As far as the authors are aware, this is the first fully-decentralized, mesh-pull protocol designed for both live streaming and VoD, and which has been tested to include the effects of peer channel browsing behavior.

The rest of this paper is organized as follows. Section II presents related work in the area of time-shifted P2P video streaming. The protocol is described in Section III, while policies are presented in Section IV. Section V shows evaluation results, and Section VI contains conclusions and suggests future work.

II. Related Work

The idea of a P2P live video streaming system that supports time shifting is relatively new. Many previous works [5], [6], [14], [15] have relied, instead, on the separate and independent distribution of live video and time-shifted streams. The seamless transition between live and time-shifted operation, as provided by LiveShift, has remained elusive. Further differences between these works and the present one can be found in the evaluation: [5] is interested in evaluating the distribution of data availability and expected number of available peers, not the QoE obtained by users; [6] is very short and presents no insights or evaluation; and in both [14] and [15], evaluation is very limited regarding QoE, since the impact of a higher number of failed requests is not clear – in addition, in this work peers need to store streams they never watched, only to be able to serve them to other peers.

The works presented in [18] and [16] unify live and time-shifted video streaming as envisioned in LiveShift, but suggest a hybrid P2P approach, with a central entity holding a complete view of which stream segments every peer has. This may become a single point of failure and result in scalability problems when the number of peers is very high, there are many streams, there is a flash crowd, or churn is high. In addition, it requires additional infrastructure (the central tracker) to be in place, which is not desirable in a scenario where users broadcast from their homes. The first of these works, [18], adopts a multiple-tree approach, which has been shown to perform badly with dynamic network conditions and churn [17], when compared compared with a mesh approach. Such conditions, though, are exactly what is expected when users are allowed to both switch channels and rapidly change their media playback position. Further, the evaluation of [18] relies on simulations in which every peer is always able to upload at a rate twice the bit rate of the video stream. This kind of reliable over-provisioning, while simplifying many problems, is unrealistic in a P2P environment where bandwidth availability is expected to fluctuate. The second of these works, [16], presents a model based on multiple-interval graphs and several optimization strategies that can be used by a central tracker with a global view. Peers are assumed to always transfer complete 1-minute-long chunks to each other, which is problematic when peers need to download different chunks from different peers, each at speed lower than the bit rate of the video stream, and combine them on time for playback. It is probably due to this fact that their evaluation only considers peer capacities which are integer multiples of the video stream bitrate. Finally, the results presented concern only the reduction of bandwidth at the provider side, which are a natural consequence of using a P2P approach, but overlook the issue of user experience.

This paper is heavily influenced by other mesh-pull (also known as *swarming*) systems and protocols. In particular, many concepts present in the BitTorrent [4] file-sharing system are part of the LiveShift protocol. The mesh-pull approach has also been successfully employed to provide live streaming by systems such as CoolStreaming [13], as well as VoD, *e.g.* in [11]. LiveShift, in contrast, unites live and on-demand video streaming under a single protocol and policies.

III. Protocol Design

LiveShift's protocol design objectives are the following: (1) **Free Peercasting:** Any peer is able to publish a channel, therefore becoming a *peercaster*; (2) **Scalability:** The approach shall scale to a high number of peers, even when several of them are only able to upload at a fraction of the bit rate of the video stream; (3) **Robustness:** The system must tolerate churn; (4) **Full decentralization:** In order to allow any peer to publish a channel without deploying a large infrastructure and to fully benefit from P2P properties, no central entities shall be present – except peercasters, a single point of failure for the live stream of the channel they originate; and (5) **Low overhead:** Video streaming is very bandwidth-consuming and sensitive to delay, therefore network overhead introduced must be low.

A. Segments and Blocks

The proposed use case gives users the possibility of switching channels and time shifting. LiveShift adopts the mesh-pull approach [10], which adapts better to dynamic network conditions and churn when compared to tree protocols [17]. Mesh-pull divides the stream into chunks that are exchanged between peers with no fixed structure. Two levels of chunking are used – a *segment* is an addressing entity, which is made up of several smaller *blocks*.

LiveShift addresses and identifies segments and blocks based solely on time – they are both of well-known fixed time-based length. This makes it trivial to discover which block and segment contain the part of the video stream to be played at a given time. This is especially useful for live streaming, since it may be difficult to predict at which bit rate the video will be generated in the future. It also makes it simple for peercasters to change the bitrate of the offered channel, or even offer variable bit rate (VBR) streams to save bandwidth in sequences that can be better compressed. Each segment is uniquely identified by a *SegmentIdentifier*, which is a pair $(channelId, startTime)$ announced on the tracker by peers which offer video blocks within a segment. Blocks are small-sized, fixed-time video chunks, and are the video unit exchanged by peers.

Differentiating segment and blocks is important to reduce the number of tracker operations (due to larger segments), while allowing peers to download blocks from several other peers, recovering quickly if any fails (due to smaller blocks). Thus, only segment availability is announced on trackers; block availability within a segment is queried directly between peers, considering that it is more sensitive to timing and synchronization issues.

B. Distributed Hash Table and Distributed Tracker

LiveShift uses a distributed hash table (DHT) to store the channel list and individual channel information. There

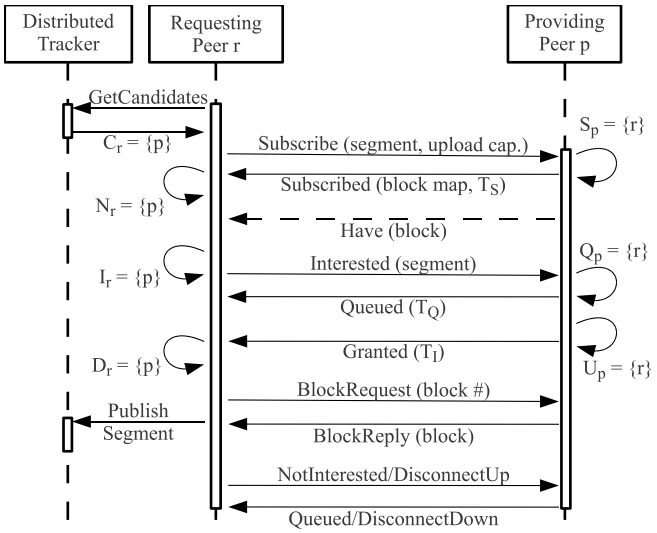

Fig. 1. LiveShift protocol example sequence diagram

are three DHT operations available: `GetChannelList` retrieves a list with all available channels and $channelIds$, `PublishChannel` and `UnpublishChannel` creates and removes a channel, respectively. A possible enhancement would be adding an electronic program guide (EPG) to map programs to $(channelId, startTime)$, achieving VoD-style program browsing.

The tracker is responsible for mapping segments to a set of providers – peers that hold at least one block in the segment. LiveShift uses a fully-distributed tracker (DT), maintained by all peers in the system, improving scalability and load balancing, avoiding a single point of failure, and reducing the infrastructure deployed by peers. There are three DT operations: `PublishSegment` is invoked by providers of a segment, `UnpublishSegment` is called by peers that have removed all blocks in a segment from their local storage, and `GetCandidates` retrieves a set with peers that provide a particular segment. The DT differs from the DHT, since its operations are invoked much more frequently. Therefore, every peer that is a provider for a certain segment caches the provider set obtained, which can then be supplied to other peers on request. This improves load balancing, which is of especial importance on popular segments. More information on the DT can be found on [8].

Since peers may leave the system unexpectedly, each tracker entry has a time-out value; thus, peers need to periodically refresh their content availability. A timeout value of 30 minutes is currently used. The timeout value is a compromise between overhead and chance of holding outdated information in the tracker, especially in the presence of churn. Even if a central tracker were in place, such a timeout value would still be important to remove outdated information.

C. Protocol Overview

The protocol is designed to be flexible by allowing the implementation of different policies for specific functions; a set of basic policies is proposed in Section IV. A key difference between LiveShift and other existing P2P systems and protocols regards symmetry of interest. Supporting VoD as envisioned in LiveShift requires that peers serve stored video blocks without necessarily being simultaneously interested in what the peer receiving the download has to offer, thus the separation between neighbors and subscribers as described below. This has implications with respect to incentive mechanisms which, although out of scope of this paper, have been successfully approached in the literature [2], [12].

Figure 1 shows, in a simplified sequence diagram, the steps taken by a peer to locate and download content (see Table I for nomenclature). A peer r, when entering the system, retrieves the channel list from the DHT. After having chosen a $channelId$ and a $startTime$ to tune into, r consults the DT to retrieve a set C_r of candidate peers (providers) that have advertised blocks in the corresponding segment. Another way of obtaining candidates is receiving `PeerSuggestion` messages, which contain suggestions for new candidates, directly from any peer. Peer r then contacts a number of candidates $p \in C_r$ by sending each a `Subscribe` message, containing the $SegmentIdentifier$ and a declared upload capacity. Verifying the correctness of the upload capacity is out of the scope of this paper.

When a peer $p \in C$ receives a `Subscribe` message from a peer r, it attempts to place r in its subscribers set S_p. If $|S_p| < \overline{S}_p$, the subscribers set is not full yet, and peer r is sent a `Subscribed` message, with a block map indicating which blocks in the requested segment p holds and a timeout value T_S. Peer r will then be subscribed to receive updates to the corresponding block map via `Have` messages. If $|S_p| = \overline{S}_p$, p checks if there is another peer $q \in S_p$ that has lower priority than r (according to the policy used, *c.f.* Section IV-D). If so, it will be preempted immediately and removed from the set. Thus, either q or r will receive a `NotSubscribed` message. Limiting $|S_p|$ is important because if each peer has $|S|$ subscribers for a particular segment, the total number of `Have` messages sent for each new block in the whole P2P network is $|S|^2 - |S|$. The constraint that a peer will not send `Have` to a peer which has reported to hold the block only reduces it to $(|S|^2 - |S|)/2$. Note that the proposed algorithm contrasts with existing mesh-pull P2P streaming ones in two important aspects; first, that the relationship between peers is asymmetric, and second, that immediate preemption causes a quicker reaction compared to re-evaluating peer selection at fixed time slots, reducing overall playback lag.

When peer r receives `Subscribed`, it adds p to the neighbor set N_r and needs to verify interest periodically by computing the intersection between blocks scheduled for downloading and blocks announced by p via its initial block map and following `Have` messages. If the intersection is not empty, r adds p to I_r and sends it an `Interested` message, which makes p add r in $Q_p \subset S_p$, the queue for peers waiting for an upload slot, and reply a `Queued` message, with a timeout value T_Q. On the contrary, when p has no

TABLE I
IMPORTANT QUANTITIES

C_r	set of candidate peers that announce at least one block in a segment at which r seeks for blocks
N_r	set of peers in which r is subscribed to block map updates
I_r	set of peers in which r is interested, waiting for a slot
D_r	set of peers which are granting r an upload slot
S_p	set of peers that subscribed to block map updates from p
T_S	time limit a peer is allowed to stay in S
Q_p	set of peers that have manifested interest and are in the upload queue waiting to get an upload slot from peer p
T_Q	time limit a peer is allowed to stay in Q
U_p	set of peers a peer p is granting an upload slot to
T_I	time limit a peer is allowed to stay in U inactive (*i.e.* maximum time between two `BlockRequests`)
$\overline{\bullet}$	maximum allowed size of set $\bullet$
$\lvert\bullet\rvert$	current size of set $\bullet$

more interesting blocks, r sends it `NotInterested` to be removed from Q_p.

Peer p has a number of upload slots $\overline{U}_p$, each of which is granted to an interested peer $r \in Q_p$. When peer r is granted an upload slot, it receives a `Granted` message, with a inactivity time-out value T_I, such that an upload slot that has not been used for T_I seconds is granted to another peer. Similarly to what happens in S_p, peers with higher priority may immediately preempt other peers from upload slots.

When r is granted an upload slot from p, it is allowed to send `BlockRequest` messages to p and receive video blocks in `BlockReply` messages. Each upload slot queues up to two `BlockRequests` at a time, to fully utilize its upload capacity, with no delays between sending a `BlockReply` and receiving the next `BlockRequest` message. This happens until a) r sends either a `NotInterested` or `DisconnectUp` message, b) p sends either a `Queued` message (if r is preempted) or a `DisconnectDown` message, or c) r times out.

The two different types of disconnect messages reflect the asymmetry of interest present in the system. `DisconnectUp` is issued by a requesting peer r which is not anymore interested in downloading a particular segment. A peer p that receives `DisconnectUp` stops uploading to r, removing it from S_p, Q_p, and U_p. `DisconnectDown`, in contrast, is issued by a providing peer p that is leaving the system. It is similar to the `NotSubscribed` message, in the sense that it communicates that r has been removed from S_p, Q_p, and U_p, so it must remove p from C_r, N_r, I_r, and D_r. The difference is that the retry behavior of r towards p should take into account that it has actually left the system. Differentiating disconnect messages allows a peer that is switching channels or time shifting to change peers it is downloading from without breaking its uploads.

D. Peer Departure and Failure

Three mechanisms are present so the system reacts quickly to peers leaving unexpectedly or failing, namely timeout values, reporting of DHT routing errors, and ping messages. The timeout values T_S, T_Q, and T_I are defined to impose a limit on the resources taken by peers that leave the system unexpectedly. The latest received timeout value always overwrites all previously received ones. Additionally, the DHT is able to inform LiveShift about routing errors. When a routing error occurs, a moving average for the failing peer is incremented. When the moving average exceeds a threshold, the peer is removed from all sets, leaving space for other peers. The moving average absorbs temporary or intermittent peer or network failures. Finally, `PingRequest` messages may be used to test if peers are on-line. Peers must reply with a `PingReply` whenever they receive a `PingRequest`, otherwise they are considered as having failed.

IV. CURRENT POLICIES

As stated in section III-C, LiveShift may be used with different sets of policies. Our focus will be on the discussion of the engineering trade-offs embodied by the policies described in this section. These policies are simple enough to produce reliable results, yet complete enough to give an accurate representation of the design space for streaming protocols capable of both VoD and real time operation.

A. Length of Segments and Blocks

Larger segments mean less entries in the DT and less `Subscribe` and `(Not)Subscribed` messages, but reduce the chance of locating interesting peers. LiveShift currently uses 10-minute-long segments, which have shown to produce good results.

To minimize delay, blocks must have a small size, since they can only be uploaded after they are completely downloaded. Yet, the smaller they are, the larger is the overhead with headers, block maps, and `Have`, `BlockRequest`, and `BlockReply` messages. A block length of one second has shown to provide a good compromise, since a peer is still able to download each one quickly from different peers and combine them in a short time period.

B. Block Selection

Another important decision is how many video blocks peers try to download ahead of the playing time. Downloading many blocks ahead may decrease the probability of a block not being present at playback time. It may not, though, be always desirable to read ahead as much as possible, since doing so may consume resources from other peers that could be used to send blocks to the community. LiveShift selects the next 15 missing blocks for downloading, counting from the current playback position, with two limitations: (1) at most 30 blocks ahead of the playback position are selected, and (2) blocks that were not yet produced because their timestamp is in the future are not scheduled for download. Peer r downloads each selected block in ascending chronological order from each peer $p \in D_p$ if p announced to hold the missing block. Another option would be downloading rarest blocks first, but since a

short time period between playing time and current time is a possibility, the policy is not used at the moment.

Since peers may fail unpredictably, `BlockRequest` messages that are not answered in a timely fashion need to be sent to another peer in D_r. While a short time-out value causes the number of duplicate blocks received to increase, wasting resources, a longer time out makes the system react too slowly, increasing the number of blocks that do not meet their playback deadline. LiveShift tackles this problem by keeping a moving average of response time of each peer in D_r. When a requested block takes longer than twice the moving average of the last five block requests, the block is rescheduled. A default value of four seconds is used for the first block download attempt, since the average is not yet known.

C. Candidate and Neighbor Selection

Initially, peer r retrieves 40 random peers from the DT to be added to C_r. Peer r may also receive candidates from a peer p via `PeerSuggestion` messages, which should be sent following `DisconnectDown` messages, containing all known peers that are providers for the segment r was subscribed. This is important to lessen disruption of the overlay, which can then be quickly repaired. Furthermore, peers add senders of `Subscribe` messages for interesting segments as candidates, since these may be newly arrived peers.

Every peer tries to have up to 15 other peers in N_r by choosing from C_r in the following order: (1) least amount of `Subscribe` messages sent, (2) highest amount of blocks provided, (3) random. This selects peers that provided successfully blocks in the past, while allowing for rotation in case downloading is not successful.

Peers stop looking for members of C_r, N_r, and I_r when receiving video blocks at a rate sufficient to keep up with normal playback, that is, at least one block per second. This is to reduce overhead and avoid needlessly preempting other peers.

As a peer advances its playback position, it eventually reaches the segment boundary and needs to start downloading the next segment. For the new segment, a peer r adds to C_r and first tries to obtain upload slots at the peers from which it has successfully downloaded blocks in the recent past. This results in a more stable segment transition, since the peer does not have to begin again looking for peers to reach D_r.

Peers are removed from C_r after exceeding a threshold (currently 5) in number of `Subscribe` messages sent without having provided any blocks. This allows new peers to be added to C_r from the DT. Peers in N_r that report to only hold blocks too far (currently 8s) behind playback position are removed from N_r, since they are unlikely to have any interesting blocks soon.

D. Subscribers and Upload Slot Selection

It is intuitive that peers should prioritize uploading to peers with a high upload capacity, because these are able to serve many other peers, amplifying the upload capacity of the peercaster earlier in the distribution process [19]. By doing so, the average application-level hop count to the peercaster through the overlay is reduced, thus reducing average playback lag.

To this end, members of S_p and U_p are primarily chosen by peer p according to their upload capacity. $\overline{S_p}$ is defined as 5 peers per upload slot, that is, $5 \cdot \overline{U_p}$. This value should not be too large, since it is only worthwhile to keep subscribed peers that are likely to get an upload slot. In case there are peers with the same upload capacity, the peers which have been given more blocks in the recent past have preference, which increases overlay stability, avoiding unnecessary changes to the overlay. A peer may only grant a single upload slot to a peer, in order to distribute streams to more peers.

Using a fixed number of upload slots has disadvantages. Possibly, the more upload slots a downloading peer is granted, the less often it will request blocks from each of them. It is difficult for an uploading peer to know precisely how much its upload slots will be used at any moment. The solution is having peers dynamically adjusting the number of upload slots according to the used upstream bandwidth. When a peer p detects that its upstream is underused by the granted upload slots, it creates a new upload slot and grants it to a peer in Q_p. When, however, the used upstream bandwidth reaches the maximum capacity of the peer and each slot is providing, on average, less than the full stream, a slot is shut down by sending a `Queued` message to the peer to which it is granted. By adjusting $\overline{U_p}$ to be able to provide at least the full stream at full rate to each slot, it avoids many peers receiving the stream at a rate too low to be played properly.

When selecting a slot to shutdown, an intuitive policy would be using the same ranking used to grant upload slots. But since the last granted slot is possibly the lowest-ranked one, the system could return to the previous state of underusing upstream. Thus, the method currently used is to shutdown the least used slot in a moving average of 3s.

Concerning timeout values, T_S is set to 5s, and T_Q to 10s. T_I defines that a peer may remain only up to 4s inactive (not downloading blocks) while granted an upload slot. Such low values are to promote rotation.

E. Playback Policy

The *playback position* is the position on the time scale that refers to the block currently being played, and is advanced by 1 block per second if the peer holds the corresponding block. Due to overlay network problems such as jitter, churn, upstream boundaries, and the limited view of resources in other peers, some blocks may not be found, or they may not be downloaded on time to be played. The *playback policy* is the decision on, when a block is missing for playback, whether to *skip* it, or *stall* waiting for it. While skipping has a negative effect on image quality, stalling increases playback lag.

LiveShift's playback policy is to skip n contiguous missing blocks if and only if the peer holds at least rn contiguous blocks immediately afterward. A ratio $r = 2$ is used on the evaluations; in practice, though, it may be more adequate to let the user adjust the ratio to express its preference on whether

TABLE II
EVALUATION SCENARIOS

Scenario	*Number PC*	*Number HU*	*Number LU*	*Churn*
s1	6	15	60	0
s2	6	15	90	0
s3	6	15	120	0
s2c30	6	15	90	30%

to skip or stall more often. The proposed policy and ratio are nevertheless effective at not letting peers stall for long in case only a few blocks are rare, achieving a low number of skipped blocks, which cause severe image quality degradation with most video encoding methods.

F. Storage Policy

The selection of which blocks peers keep in the local storage in case they run out of space is an interesting aspect and impacts data availability. The currently used storage policy is storing all received blocks until the maximum capacity – currently two hours of video – is reached. When storage capacity is full, blocks with oldest download time are deleted to make up space. Although simple, the strategy achieves good results, since blocks that are currently more popular naturally get more replicated in the system. The peercaster may choose to offer larger storage capacity in order to assure that at least one copy of past video streams is accessible in the system, though this is not assumed on this paper's evaluation section.

V. EVALUATION

LiveShift has been fully implemented, in Java version 1.6, to allow the investigation of the interdependencies among the different policies. The evaluation was made using 16 physical machines from the CSG Testbed – a detailed description of the testbed can be found at [1]. This section presents the subset of our evaluation results which pertains to the validation of the proposed protocol and policies, the testing of its scalability limits, and the investigation of its improvement opportunities. Evaluation includes both channel browsing behavior and churn to produce highly realistic results. The reader is directed to [7] for additional evaluation results.

A. Evaluation Scenarios and Peer Behavior

Table II describes the four scenarios addressed. Peers were divided in classes regarding their maximum upload capacities. High upload capacity (HU) peers and peercasters (PC) have upload limit of 500%, *e.g.*, for the video stream bit rate of 500 kbit/s defined for our experiments, HU and PC nodes have an upload capacity of 2.5 Mbit/s. This value is not particularly high, considering that these nodes may be running at universities or connected via FTTH (Fiber to the Home) technology. Low upload capacity (LU) peers may be, for example, running DSL (Digital Subscriber Line) or cable connections, and have only 50% upload capacity (250 kbit/s). Peers are not limited in download bandwidth. All results were obtained over 10 runs of 1 hour each.

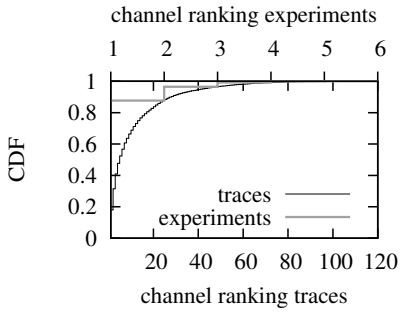

Fig. 2. Channel popularity

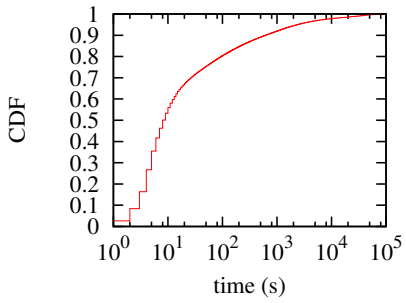

Fig. 3. Channel holding time

Peer behavior was modeled using traces from a real IPTV system [3]. Peers were created with an inter-arrival time of 1s and loop through the following two steps: (1) choose a channel and starting time, (2) hold to the channel, locating and downloading content from other peers. The distribution used to determine which channel peers switch to is displayed in Figure 2, including the original distribution with 120 channels, and the one used on these experiments with six channels. The channel holding time distribution is pictured in Figure 3. Since there is no system supporting both live and time-shifted streaming as described on this paper, there are no traces available documenting how long behind real time will a newly arriving peer join. Hence, it was assumed that the probability of a peer tuning to a time nearer the current *live* time is exponentially larger than the probability of it tuning to some other point in the past. The distribution used was the one pictured in Figure 3, with the starting time ranging from the current playback position, and all the way back to the playback time at the start of the experiment.

In the scenario with churn, when a peer chooses a channel and time to tune to, it has a nonzero probability of going offline. While offline, a peer does not react to any incoming message. Peers remain offline for an amount of time given by the channel holding time distribution before having again the same chance of remaining offline or going back on-line. Peers disconnect cleanly, that is, they follow the protocol properly by sending `PeerSuggestion`, `DisconnectUp`, and `DisconnectDown` messages.

B. Quality of Experience and Scalability

The main Quality of Experience (QoE) metric used is the playback lag experienced by users during holding time, from the point a $(channelId, startTime)$ was selected. Figures 4-7 show the playback lag experienced as users hold on (watch) a channel in the different proposed scenarios. The x-axis represents the time (in minutes) for which a user holds on a channel. The y-axis represents the playback lag (in seconds), which is the difference between the current playback position, according to the defined playback policy, and the time of the block that should be playing if there were no interruptions in playback. A lower playback lag means lower start-up delay, less interruptions, and more closeness to what the user initially intended to watch, thus better user experience. Since blocks in LiveShift are transmitted via reliable connections (TCP), there is no risk of losing data within a block. For example, the 95% LU line designates the maximum playback lag which 95% of low upload capacity users achieve. In other words, it is the worst case lag for 95% of LU peers. As can be seen in Figures 4-7, our results show that:

a. playback lag increases only slightly as users continue to view a channel, which shows that users do not experience frequent stalling;
b. even in the worst case scenarios investigated, 95% of HU peers experience playback lag of less than 10 seconds, which is acceptable performance (indeed many live TV broadcasts may have similar lag); and
c. LU peers are much more susceptible to high lag and especially in scenarios with churn or less available bandwidth. For instance, Figure 7 shows that 5% of LU peers exhibit a playback lag larger than 25 seconds after watching for long periods of time, while in Figure 6, the worst 5% of LU peers have playback lag just above 50 seconds after watching a channel for more than 40 minutes.

In Scenario s3, the system shows signs of being saturated, with several LU peers exhibiting playback lags surpassing 30s. This happens because the average delay to the peercaster through the overlay increases, since the blocks need to travel through relatively more hops. In addition, peers take a longer than average time to obtain upload slots, which are more disputed in this scenario.

Overall, average playback lag is 5.45s in s1, 7.70s in s2, 14.31s in s3, and 8.93s in s2c30, showing that 30% churn increases average playback lag by about 15% in s2.

C. Skipped Blocks and Failed Playback

According to the playback policy defined in Section IV-E, some blocks may be skipped, therefore avoiding an increase in playback lag (it actually *decreases*). The proportion of skipped blocks, on average, 2.61% of the total blocks played in s1, 2.41% in s2, and 1.86% in both s3 and s2c30. Interestingly, relatively less blocks are skipped in more bandwidth-constrained scenarios, due to fewer concurrent downloads.

The availability of content is affected by the fact that peers change their interest frequently. In the worst case, a peer may not be granted an upload slot from any of the peers which hold the blocks that it seeks. This may happen even when the system has spare bandwidth, due to the unbalance in content popularity: peers that have unused upload capacity may only hold unpopular content, leading to available overlay resources remaining unused. If playback stalls for a long time, it is not realistic to assume that the user will wait forever. Thus, when a peer, in a sliding window of the last 30s of playback, is able to play less than half the blocks it should have been able to play, playback is considered failed, that is, the user is considered to have given up and switched to another $(channel, time)$. Failed playback events account for 0.08% of the channel switches in s1, 0.60% in s2, 1.87% in s2c30, and 4.64% in s3.

D. Upload Capacity Utilization

The upload capacity utilization of a peer is its percentage of upload capacity used, on average, and is obtained by dividing its *used* upload capacity by its *total* upload capacity. An efficient P2P system is able to discover unused bandwidth and react quickly to peers changing interest, which is challenging in a fully-decentralized system.

Figure 8 shows the average upload capacity utilization, per peer. The presented protocol and policies are able to achieve on average 63.2% upload capacity utilization in s1, 65.8% in s2, 82.4% in s3, and 64.7% in s2c30. HU peers use more of their upload capacity, since they are placed nearer the PCs (due to the policy in Section IV-D), receiving (and announcing) having blocks sooner than LU peers. There is little variation in upload capacity utilization of PCs in the different scenarios because they do not react to channel popularity – in s2, while the PC for channels 1 and 2 average higher than 98% upload capacity utilization, the PC for channel 6 averages only 4.0% simply because the channel is unpopular. In addition, due to the difference in popularity of channels and the dynamic behavior of peers, some swarms may be more or less provided with bandwidth. This can explain why some LU capacity is used while HUs are not fully loaded: some swarms may not have enough HUs, and need to resort to LUs. Because of the small block size, peers can combine the upstream capacity of several LU peers.

E. Overhead

Since both blocks and segments are fully time-based, the absolute system overhead does not depend on the bit rate of the video stream, but rather, on the length of blocks and segments.

Figure 9 gives an insight into the running protocol, displaying the average number of messages a peer sends per second, by type, in Scenario s2. Some messages `Have` messages are the most common ones, which is expected from the protocol design. The number of `Have` messages sent per second mean that, on average, $|S| = 4.88$ in s2. All other messages have negligible size when compared to the `BlockReply` message, which carries the video block itself. The overhead relative to a stream of 500kbit/s is, for any scenario, at most 2.01% on average, excluding DT and DHT traffic, as can be seen in Figure 11. Figure 12 includes DT and DHT

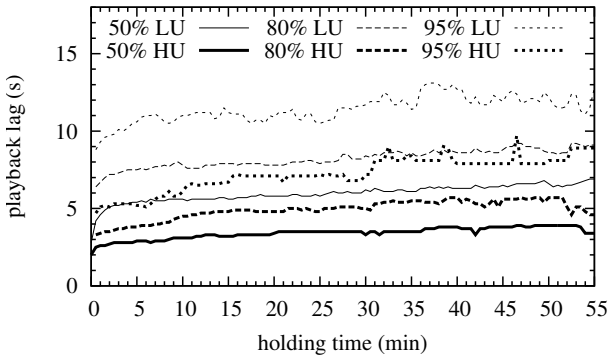

Fig. 4. Playback lag in Scenario s1

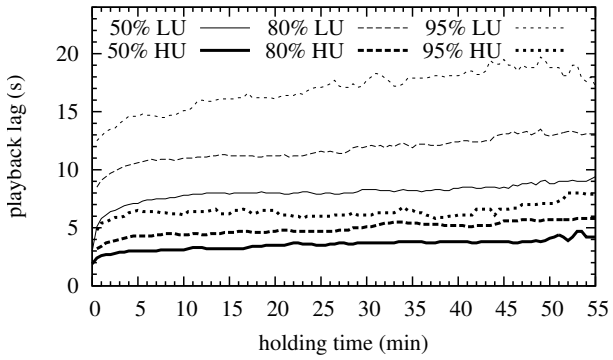

Fig. 5. Playback lag in Scenario s2

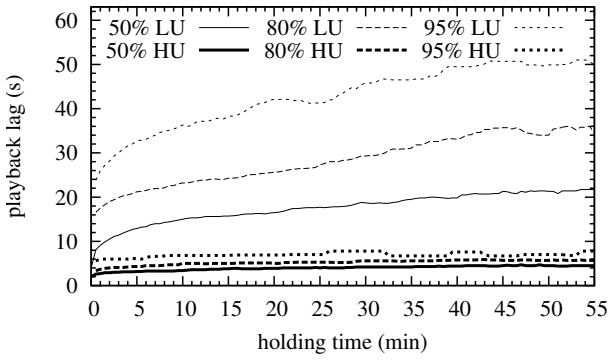

Fig. 6. Playback lag in Scenario s3

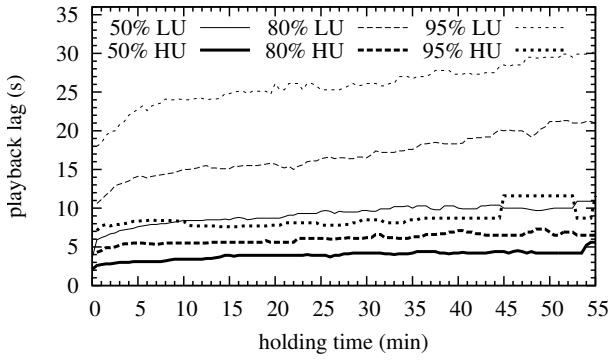

Fig. 7. Playback lag in s2c30

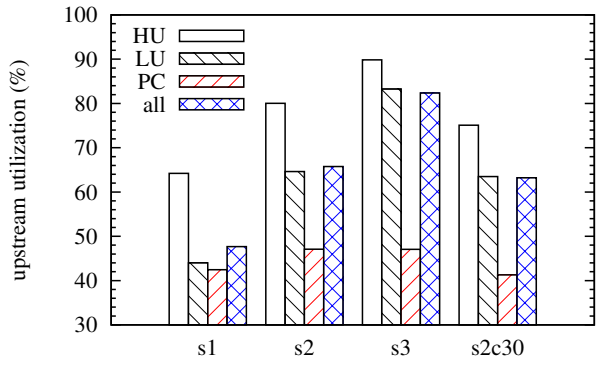

Fig. 8. Upstream utilization

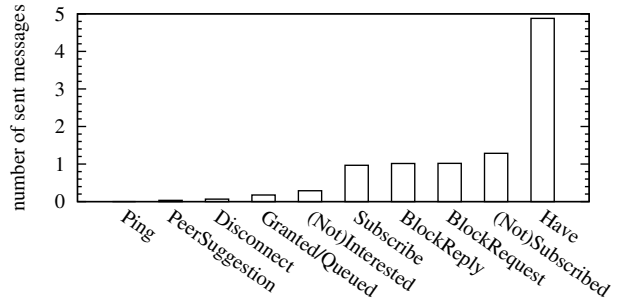

Fig. 9. Sent messages per peer per second in Scenario s2

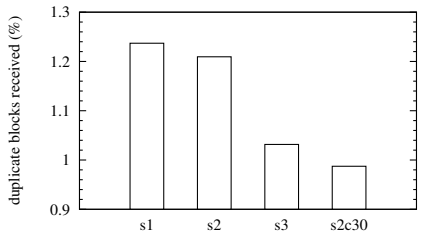

Fig. 10. Duplicate blocks received

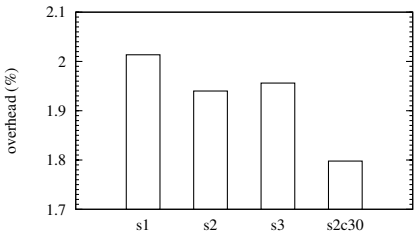

Fig. 11. Overhead not including DHT+DT

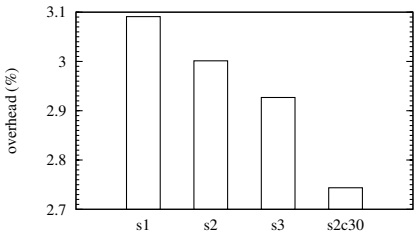

Fig. 12. Overhead including DHT+DT

traffic, and shows that overhead is at most 3.09% on average. Interestingly, Scenario s1 has slightly higher overhead than the other scenarios, which is credited to the higher number of messages allowed by the higher available bandwidth.

Figure 10 shows that the proportion of duplicate blocks received is below 1.3% on average, for all of the analyzed scenarios. Similarly to the number of skipped blocks, the decrease in more bandwidth-restricted scenarios is explained by a smaller $|D|$, which means a restricted choice of peers to download video chunks from. This validates the rescheduling policy presented in Section IV-B.

VI. Conclusions and Future Work

This paper presents a flexible and fully-decentralized mesh-pull P2P protocol for locating and distributing both live and time-shifted video streams in an integrated manner. It also presents policies that can be used with the new protocol, revealing and discussing the main trade-offs encountered in building such a system, as well as evaluation results.

Trace-driven evaluations focus in scenarios with many channel switches and an increasing number of peers with upstream capacity lower than the rate of the video stream being transmitted. The system shows its ability to find content and upstream capacity quickly enough to sustain a low playback lag relatively to the holding time. LiveShift also shows ability to combine the upload capacity of several low upstream peers on time. In the scenarios studied, the system supports a low (less than 10s) playback lag for 80% of users with high bandwidth, even in the presence of churn – in form of channel switching, time shifting, and peers disconnecting. Users with low upstream bandwidth are negatively affected in scenarios with high churn or overall low upload capacity, but playback lag remains within 60 seconds of transmission for over 95% of the peers in the investigated scenarios. Finally, in the scenarios studied, overhead remains below 3.1% of the original bitrate of the video stream being transmitted.

While the definition and evaluation of a protocol and initial policies represent an important first step into supporting the proposed use case of integrating both live and time-shifted video streaming in a fully-decentralized environment, this paper identifies many open questions. Future work includes further analyzing and finding optimal policies, developing an effective incentive mechanism to verify the upload capacity of peers that may be applied in the proposed use case, and running longer experiments on a global scale with a higher number of users and channels. Further important issues to be considered concern overall system security and commercial aspects.

Acknowledgment

This work has been performed partially in the framework of the EU ICT STREP SmoothIT (FP7-2008-ICT-216259).

References

[1] "Testbed Infrastructure for Research Activities – CSG," http://www.csg.uzh.ch/services/testbed/, last visited: 26.06.2010.

[2] T. Bocek, Y. El-khatib, F. V. Hecht, D. Hausheer, and B. Stiller, "CompactPSH: An Efficient Transitive TFT Incentive Scheme for Peer-to-Peer Networks," in *Proceedings of the 34th IEEE Conference on Local Computer Networks*, Zurich, Switzerland, October 2009.

[3] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatriain, "Watching Television over an IP Network," in *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, New York, NY, USA, 2008, pp. 71–84.

[4] B. Cohen, "Incentives Build Robustness in BitTorrent," Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003.

[5] S. Deshpande and J. Noh, "P2TSS: Time-Shifted and Live Streaming of Video in Peer-to-Peer Systems," in *2008 IEEE International Conference on Multimedia and Expo*, April 2008, pp. 649 –652.

[6] D. Gallo, C. Miers, V. Coroama, T. Carvalho, V. Souza, and P. Karlsson, "A Multimedia Delivery Architecture for IPTV with P2P-Based Time-Shift Support," in *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, Piscataway, NJ, USA, 2009, pp. 447–448.

[7] F. Hecht, T. Bocek, R. G. Clegg, R. Landa, D. Hausheer, and B. Stiller, "LiveShift: Mesh-Pull P2P Live and Time-Shifted Video Streaming," University of Zurich, Department of Informatics, Tech. Rep. IFI-2010-0009, September 2010.

[8] F. Hecht, T. Bocek, and B. Stiller, "B-Tracker: Improving Load Balancing and Efficiency in Distributed P2P Trackers," in *To appear in 11th IEEE International Conference on Peer-to-Peer Computing (P2P'11)*, Kyoto, Japan, September 2011.

[9] F. V. Hecht, T. Bocek, C. Morariu, D. Hausheer, and B. Stiller, "LiveShift: Peer-to-peer Live Streaming with Distributed Time-Shifting," in *8th International Conference on Peer-to-Peer Computing*, Aachen, Germany, September 2008.

[10] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross, "A Measurement Study of a Large-Scale P2P IPTV System," *IEEE Transactions on Multimedia*, vol. 9, no. 8, pp. 1672–1687, December 2007.

[11] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang, "Challenges, Design and Analysis of a Large-Scale P2P-VoD System," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 375–388, 2008.

[12] R. Landa, D. Griffin, R. G. Clegg, E. Mykoniati, and M. Rio, "A Sybilproof Indirect Reciprocity Mechanism for Peer-to-Peer Networks," in *Proceedings of INFOCOM 2009*, Rio de Janeiro, Brasil, April 2009.

[13] B. Li, S. Xie, Y. Qu, G. Keung, C. Lin, J. Liu, and X. Zhang, "Inside the New Coolstreaming: Principles, Measurements and Performance Implications," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, Apr. 2008, pp. 1031 –1039.

[14] Y. Liu and G. Simon, "Peer-to-Peer Time-shifted Streaming Systems," *ArXiv e-prints*, Nov. 2009.

[15] Y. Liu and G. Simon, "Distributed Delivery System for Time-Shifted Streaming Systems," in *2010 IEEE 35th Conference on Local Computer Networks (LCN 2010)*, Oct. 2010, pp. 276 –279.

[16] ——, "Peer-Assisted Time-shifted Streaming Systems: Design and Promises," in *Communications (ICC), 2011 IEEE International Conference on*, Jun. 2011.

[17] N. Magharei and R. Rejaie, "Mesh or Multiple-Tree: A Comparative Study of Live P2P Streaming Approaches," in *Proceedings of IEEE INFOCOM 2007*, 2007, pp. 1424–1432.

[18] J. Noh, A. Mavlankar, P. Baccichet, and B. Girod, "Time-Shifted Streaming in a Peer-to-Ppeer Video Multicast System," in *GLOBECOM'09: Proceedings of the 28th IEEE conference on Global telecommunications*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 6025–6030.

[19] M. Piatek, A. Krishnamurthy, A. Venkataramani, R. Yang, D. Zhang, and A. Jaffe, "Contracts: Practical Contribution Incentives for P2P Live Streaming," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 6–6. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855711.1855717