University of Zurich
Department of Informatics

Fabio Hecht
Thomas Bocek
Richard G. Clegg
Raul Landa
David Hausheer
Burkhard Stiller

LiveShift: mesh-pull P2P live and time-shifted video streaming

TECHNICAL REPORT – No. IFI-2010.0009

September 2010

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

F. Hecht
T. Bocek
R. Clegg
R. Landa
D. Hausheer
B. Stiller:
Technical Report No. IFI-2010.0009, September 2010
Communication Systems
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland

Chapter 1

Introduction

The demand for video streaming on the Internet is increasing [8]. The growth in deployment of Fiber to the Home (FTTH) technology increases upload capacity at the edge, making the peer-to-peer (P2P) paradigm especially attractive to increase scalability and decrease cost for the publisher.

Since users of a P2P system already successfully collaborate on distributing video streams, the contribution of LiveShift is to allow further collaboration by having peers store received video streams in order to distribute them in the future, thus allowing time shifting or – if the combined storage is large – even video-on-demand (VoD). This enables the use case of 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. Similarly, the live transmission may be used for the premier of a movie, TV show, or news program, when several users might watch it at the same time. Instantly and automatically it will be available – since the starting time of the premier – to every user joining at a later time. The system may be deployed on PCs, set top boxes, or servers at the providers.

The extra challenges that the proposed functionality introduces are several. Differently from a live video streaming system, users may switch not only channels but also positions in a potentially large time scale. This, added to the asymmetry of interest inherent in such a scenario, requires a protocol and policies that do not require that peers need to be simultaneously interested in data each other has.

LiveShift functionality was presented as a demonstration [7]. This technical report defines an architecture that supports the envisioned use case, describes a flexible mesh-pull protocol and a set of preliminary policies to be used with the protocol, finally presenting evaluation results. A full implementation is used, allowing interdependencies among different policies to be investigated, revealing interesting properties and trade-offs. Chapter 2 presents related work, while the overall design, containing the architecture, protocol and policies, is shown in Chapter 3. Chapter 4 shows evaluation results, and Chapter 5 contains conclusion and future work.

Chapter 2

Related Work

The idea of a P2P live video streaming system that supports time shifting is relatively new. There are P2P applications that support either live or on-demand video streaming, and papers that propose P2P time-shifted video streaming with no integration to the live stream. While supporting integration as proposed in LiveShift is not supported by any application, there is related work on how to achieve such integration using a multiple-tree approach.

Tribler [13] is a modified BT client that supports both live and on-demand P2P video streaming. In order to provide VoD, it alters the client to download pieces in sequential order. To support live streaming, it introduces special mechanisms to bypass problems with an unknown video length, unknown future video content, unsuitable piece-selection policy, and the lack of seeders [12]. Live video streaming and VoD are implemented as completely separate modes – a user may not switch from one to the other. In fact, when watching a live stream, it is not possible to delay the reception arbitrarily.

[10] and [6] propose that the distribution of live video streams is done separately and independently of the time-shifted streams. The use case of seamlessly switching from live to time-shifted is therefore not supported. The work presented in [14] comes closest to LiveShift – it adopts, though, a multiple-tree approach, with each video source maintaining a complete view of the swarm.

LiveShift takes a different approach and defines a unified mesh-pull protocol and policies to distribute and locate efficiently both live and time-shifted video streams in a fully-distributed manner.

Chapter 3

System Architecture

This chapter describes, at a high level, which components are part of LiveShift's architecture, as well as the protocol design and a set of policies.

3.1 Design Objectives

LiveShift's 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;
3. **Robustness:** The system must tolerate churn;
4. **Full decentralization:** In order to fully benefit from P2P properties, no central entities shall be present – except peercasters, which are 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 by the protocol and mechanisms must be low.

3.2 Main Components

Figure 3.1 shows the components of the top-level LiveShift architecture, which are split into three layers. The Network layer contains the Distributed Tracker (DT) and the Distributed Hash Table (DHT) to store network-wide persistent information in a fully-decentralized manner, and a Signaling component to handle direct communication between peers. The DT and DHT are provided by the TomP2P [3] library. The Signaling

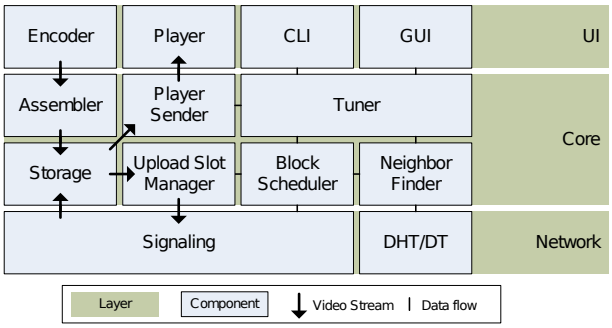


Figure 3.1: LiveShift top-level architecture

component handles direct communication between peers. Peers use it to send unicast messages to other peers.

The Core layer contains all main software functionality. The Storage stores video blocks, which are small-sized video chunks. Blocks are received from the Assembler if the peer is a peercaster, otherwise, they come from other peers via the Signaling component. The Upload Slot Manager component manages creation, granting, and verification of upload slots. Its policies are of fundamental importance to the distribution of the video stream, since it selects how many and which peers will be granted upload slots, thus receive video blocks, and distributes the available upload bandwidth among upload slots. The Block Scheduler selects next blocks to be played and sends messages requesting blocks to appropriate peers. The Neighbor Finder maintains and manages sets of candidates C and neighbors N, querying the distributed tracker when necessary. The Assembler receives the encoded video stream from the Encoder and partitions it into blocks and segments. The Player Sender obtains blocks from the Storage, joins the video data contained in them, and sends it to the Player at the right rate. The Tuner receives commands from the UI about which channel and starting time to tune into. It, then, initializes and coordinates the Neighbor Finder, the Block Scheduler, and the Player Sender.

The User Interface (UI) layer manages machine-human interfaces, encodes and decodes video. The Encoder is responsible for receiving signal from a video capture device or file, encode them properly, and output them to Segment Assembler. LiveShift uses the VLCJ library [2] as encoder. The player is responsible for decoding the video stream and playing it back to the user. VLCJ is also used for this purpose. The CLI (Command-Line Interface) is used to control automated experiments from the command line. The Graphical User Interface (GUI) is operated by users in demonstrations and trials.

3.3 Protocol Design

This section describes the protocol used by LiveShift, justifying design decisions based on the use case and design objectives.

3.3.1 Segments and Blocks

The proposed use case gives users the possibility switching channels and time shifting. Hence, LiveShift adopts the mesh-pull approach, which is the most used P2P video streaming paradigm at present [8] due to its flexibility, adapting better to dynamic network conditions and churn when compared to tree protocols [11]. 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 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 entries in the tracker, while allowing peers to download blocks from several other peers, recovering quickly if a peer fails.

3.3.2 Distributed Hash Table and Distributed Tracker

LiveShift uses a distributed hash table (DHT) to store the channel list and individual channel information. There are three DHT operations available: `GetChannelList` retrieves a list with all available channels and $channelIds$, `PublishChannel` and `UnpublishChannel` create and remove a channel, respectively. A possible enhancement would be adding an electronic program guide (EPG) to map programs to $(channelId, startTime)$.

The tracker is responsible for mapping content to peers. LiveShift uses a distributed tracker (DT), maintained by all peers in the system, improving scalability and avoiding a single point of failure. There are three DT operations: `PublishSegment` is invoked by peers that possess at least one block in a particular segment, `UnpublishSegment` is called by peers that have removed all blocks in a segment, and `GetPeerList` retrieves a list of peers that have published a particular segment. 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.

3.3.3 Protocol Overview

Figure 3.2 shows in a simplified sequence diagram how a peer locates and downloads content. Table 3.1 shows important quantities used. The protocol is designed to be

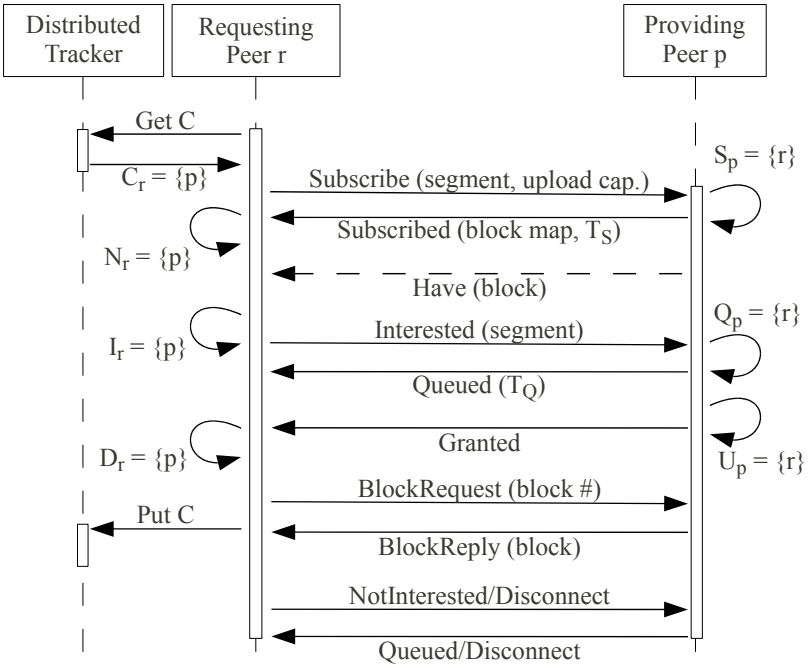


Figure 3.2: LiveShift protocol example sequence diagram

flexible by allowing implementation of different policies. A set of policies *e.g.* for selecting peers is described in Section 3.4. A key difference from LiveShift to existing P2P systems and protocols regards symmetry of interest. Supporting VoD as envisioned in LiveShift requires that peers may serve stored video blocks without necessarily being simultaneously interested in what the other has to offer. This causes implications with respect to incentive mechanisms; although incentives are out of scope of this work, it is possible to provide incentives in P2P systems with asymmetry of interest [4, 9].

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 that advertised blocks in the corresponding segment. Another way of obtaining candidates is receiving `PeerSuggestion` messages, which contain suggestions for new candidates. Peer r then contacts a number of candidates $p \in C$ by sending each a `SubscribeRequest` message, containing the $SegmentIdentifier$ and a declared upload capacity. Verifying the correctness of the upload capacity is out of the scope of this technical report.

When a peer $p \in C$ receives a `SubscribeRequest` 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. 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. If so, it will be preempted 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

Table 3.1: Important quantities

C_r	set of candidate peers that announce blocks in a segment in which r is interested
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
$\overline{\bullet}$	maximum allowed size of set $\bullet$
$\lvert\bullet\rvert$	current size of set $\bullet$

number of `Haves` sent for each new block in the system is $|S|^2 - |S|$. The constraint that a peer will not send `Have` to a peer which has reported to hold the block may only reduce it to $(|S|^2 - |S|)/2$.

When r receives `Subscribed`, it adds p to the neighbor set N_r and needs to verify interest periodically by computing the intersection between scheduled blocks and blocks announced by p. If p reports to have blocks that r needs to request, r sends p 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 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. Similarly to what happens in S_p, peers with higher priority may preempt peers from upload slots.

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

3.3.4 Peer Departure and Failure

The system needs to react quickly to peers leaving the system unexpectedly, or failing, temporarily or permanently. Thus, two mechanisms are present to address this. The DHT notifies the application when a routing error occurs, incrementing a moving average

for the failing peer. 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 failures. Also, `PingRequest` messages may be used to test if peers are on-line. Peers are expected to reply with a `PingReply` whenever they receive a `PingRequest`. A peer which has failed recently has a reduced chance of getting in N and U.

3.4 Current Policies

The defined protocol may be used with different policies. While optimal policies have not been studied, this section discusses trade-offs and shows policies used for evaluations.

3.4.1 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 acceptable 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 1s has shown to produce good results, since a peer is still able to download each one quickly from different peers and combine them in a short time period.

3.4.2 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. Currently, LiveShift schedules the next 15 missing blocks, starting from play position, with a limit of 30 blocks ahead of the play position.

Since peers may fail unpredictably, `BlockRequest` messages that are not answered in a timely fashion need to be sent to another peer in D. 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. When a requested block takes longer than twice the moving average of the last five block requests, the block is rescheduled.

3.4.3 Candidate and Neighbor Selection

Initially, 40 random peers are retrieved from the DT to be added to C. Peers also receive candidates from other peers in `PeerSuggestion` messages when preempted from an upload slot, or when the peer disconnects, lessening disruption to the overlay. Furthermore, peers add to C senders of `Subscribe` messages for interesting segments.

Every peer tries to have up to 15 peers in N_p by choosing from C_p 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 contact is not successful.

Peers stop looking for members of C, N, I when receiving video blocks at a rate sufficient to keep up with normal playback. This is to reduce the number of messages exchanged and avoid needlessly preempting other peers.

As a peer advances its play position, it eventually reaches the segment boundary and needs to start downloading the next segment. For the new segment, a peer adds to C and first tries to obtain upload slots at the peers from which it has successfully downloaded blocks in the recent past. This aims to make the transition smoother, since the peer does not have to begin again looking for peers that will grant blocks.

Peers are removed from C 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. Peers in N that report to only hold blocks too far (currently 8s) behind play time are also removed, since they are not likely to have interesting blocks soon.

3.4.4 Subscribers and Upload Slot Selection

Members of S and U are chosen according to their upload capacity. $\overline{S}$ is defined as 5 times the upload capacity of the peer (in full streams). It should not be too large, since it is only worthwhile to keep peers that are likely to get an upload slot. In case there are more peers with same upload capacity, the peers which have been given more blocks in the past have preference, which increases overlay stability. 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. In general, the more upload slots a peer is granted, the less often it will request blocks from each of them. It is difficult for a peer to know 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 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. When, however, the used upstream bandwidth reaches the maximum capacity of the peer and each slot is providing, in average, less than the full stream, it shuts down a slot by sending a `Queued` message to the peer to which it is granted. By adjusting $\overline{U}$ to be able to provide at the least the full stream at full rate to each slot, it avoids getting the system to a state in which many peers receive 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 used currently 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. A peer in U may remain only 4s inactive (not downloading). Such low values are to encourage rotation.

3.4.5 Playback Policy

Due to *e.g.* jitter, churn, upstream boundary, and limited view of peers, some blocks may not be downloaded on time to be played, or not be found at all. 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 current playback policy is to skip n contiguous missing blocks if and only if the peer holds $2n$ contiguous blocks immediately afterward. Although conservative, the proposed policy is effective in not letting peers stall for long in case only a few blocks are rare, and displays very low number of skipped blocks, which cause severe image quality degradation with most video encoding methods. Even with the use of Scalable Video Coding (SVC) [15], arbitrary parts of the stream may not be lost.

3.4.6 Storage Strategy

The selection of which blocks peers keep in the local storage in case a they run out of space is an interesting aspect and impacts data availability. This is, however, left for future work. Peers currently store all received blocks in their local storage until the maximum capacity – currently 2 hours of video – is reached; then, the oldest blocks are deleted.

Chapter 4

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 the CSG Testbed [1], a distributed network of several real machines, in which 17 machines were used to run up to 141 instances of LiveShift. . This chapter presents an extract of evaluation results, done to validate proposed protocol and policies, test scalability limits, and investigate improvement opportunities.

4.1 Evaluation Scenarios and Peer Behavior

The defined evaluation scenarios and peer behavior include both channel browsing behavior and churn to produce highly realistic results.

4.1.1 Evaluation Scenarios

Table 4.1 describes the four scenarios used. Peers were divided in classes regarding their maximum upload capacities. High Upstream (HU) peers and Peercasters (PC) have upload limit of 500% of the bit rate of the video stream, *e.g.*, the bit rate of the video stream is 500 kbit/s and their capacity is 2.5 Mbit/s. They may be running at universities or connected via FTTH (Fiber to the Home) technology. Low Upstream (LU) peers may be, for example, running DSL (Digital Subscriber Line) or cable connections, and have only 50% upstream capacity. Peers are not limited in download bandwidth. All results were obtained over 10 runs of 1 hour each for each scenario.

4.1.2 Peer Behavior

Peer behavior was modeled using traces from a real IPTV system [5]. Peers are created with an inter-arrival time of 1s and loop through the following two steps: (1) choose a

Table 4.1: 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%

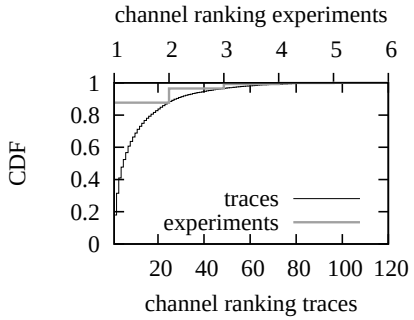


Figure 4.1: Channel popularity

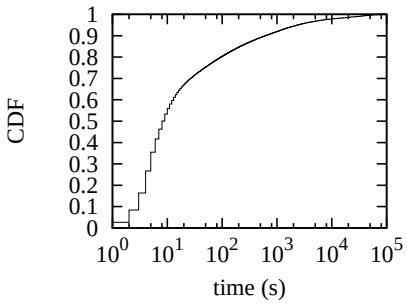


Figure 4.2: Channel holding time

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 4.1, including the original distribution with 120 channels, and the one used on these experiments with six channels. Since there is no system supporting both live and time-shifted streaming as described on this work, there are no traces available on starting time. Hence, it was assumed that the probability of a peer tuning to time nearer the current time (live stream) is exponentially larger than it tuning to some point in the past. The distribution used was the one pictured in Figure 4.2, with the starting time ranging from the current time to the start of the experiment. The channel holding time distribution is pictured in Figure 4.2.

In the scenario with churn, peers, when choosing a channel and time to tune to, have a certain chance of going offline. While offline, they do 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. Peer disconnect cleanly, that is, they follow the protocol and send `PeerSuggestion` and `Disconnect` messages properly.

4.2 Evaluation Results

This section presents the evaluation results obtained by applying the described policies to the protocol using the defined scenarios.

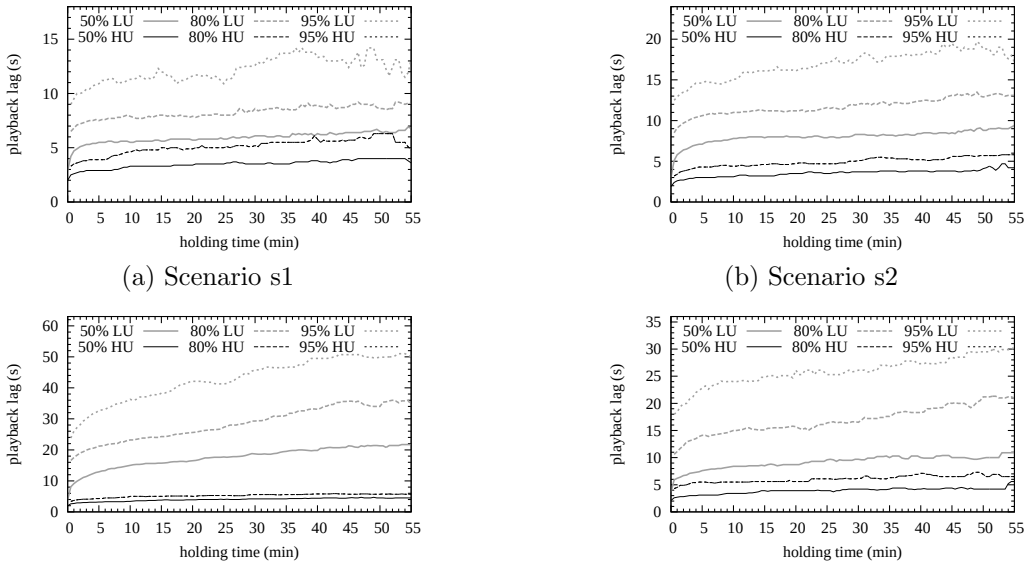
4.2.1 Quality of Experience and Scalability

The Quality of Experience (QoE) metric used is the playback lag experienced by users, during holding time, from the point a $(channelId, startTime)$ was selected. The playback lag is the difference between the time of the video block that is playing, according to the defined play policy, and the time of the block that should be playing if there were no interruptions at all 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.

Figures 4.3 and 4.4 show the average playback lag experienced as users hold to a channel in the different proposed scenarios. Playback lag has a skewed distribution and tends to form a longer tail as the bandwidth gets more restricted. The lines for 50th and 95th percentiles for HU peers have collapsed into a single line for all scenarios, showing that there is no long tail in playback lag for the respective peer class. 95% of HU peers are able to maintain good (*i.e.* less than 7 seconds) playback lag in all scenarios, due to the priority used in S and U, which places them closer to the peercaster in the overlay. The increasing number of LU peers in scenarios s2 and s3 hinder only the QoE of peers in that class. In Scenario s3, the system shows signs of being saturated, as playback lag for several LU peers already go above 30s. This happens because the average distance to the peercaster increases and the stream needs to go through many hops; besides, peers take longer on average to obtain upload slots from other peers. While churn has little impact on HU peers, since they are placed near the peercaster and have the power to preempt LU peers, it does impact LU peers, which still display playback lag below 30s for 95% of peers. Overall average playback lag is 5.45s in s1, 7.70s in s2, 14.31s in s3, and 8.93s in s2c30.

According to the play policy defined in Section 3.4.5, some blocks may be skipped, not causing playback lag to increase, but in fact to decrease. Figure 4.5 displays the average share of skipped blocks in the different scenarios. It is interesting relatively less blocks are skipped in more bandwidth-constrained scenarios. This may be due to less concurrent downloads happening, so the probability is lower that a block due to be played immediately takes longer to be downloaded than a block further ahead in the time scale, causing it to skip.

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 any upload slot from peers which hold the blocks sought after simply because all peers holding the desired block are busy serving other peers. This may happen even when the system has spare bandwidth, due to the unbalance in content popularity – some peers may have unused upstream capacity due to holding only unpopular content. In case playback stalls for a long time, it is considered that the user gave up and switched to another $(channel, time)$, since it is not realistic to assume that the user will wait forever on a blank screen. The criterion used was the following. When a peer, in a sliding window of the last 30s of playback, is able to play an amount equal or less than half the blocks it should (15 blocks), playback is considered failed. Figure 4.6 shows that failed playbacks represent a very low share of switches in s1, but increases significantly in Scenario s3.



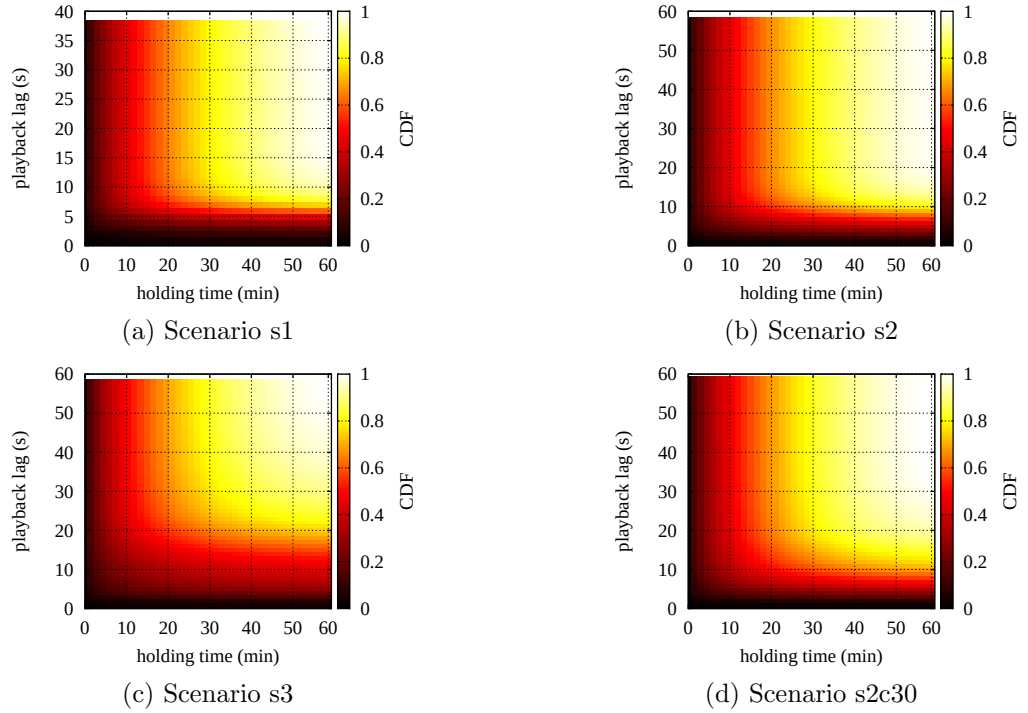
(a) Scenario s1

(b) Scenario s2

(c) Scenario s3

(d) Scenario s2c30

Figure 4.3: Playback lag in all scenarios



(a) Scenario s1

(b) Scenario s2

(c) Scenario s3

(d) Scenario s2c30

Figure 4.4: Playback lag and holding time in all scenarios

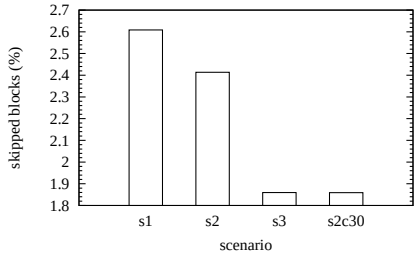


Figure 4.5: Skipped blocks

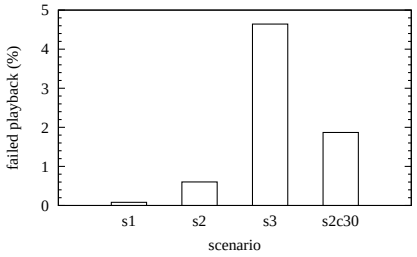


Figure 4.6: Failed playback

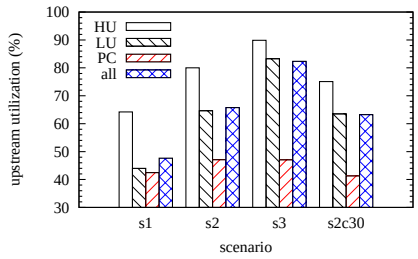


Figure 4.7: Upstream utilization

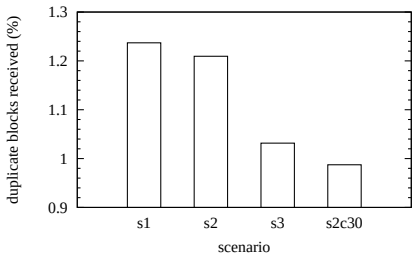


Figure 4.8: Duplicate blocks received

4.2.2 Upstream Utilization

Upstream utilization of a peer is the percentage of upload capacity used, on average, and is obtained by dividing its used upstream 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 4.7 shows the average upstream utilization, per peer. HU use more of their upstream capacity, since they are placed nearer the PCs, receiving (and announcing) having blocks sooner than LU peers. There is little variation in upstream utilization of PCs in the different scenarios because they do not react to channel popularity – in s2, for example, while the PC for channels 1 and 2 average higher than 98% upstream utilization, the PC for channel 6 averages only 4.0% simply because the channel is unpopular. Also due to the difference in popularity of the channels and the dynamic behavior of peers, some swarms may be more or less provided with bandwidth, which explains why some LU capacity is used while HUs are not fully loaded – while some swarms may be entirely served by HU peers, some may not have enough HUs and need to resort to LU peers. This explains why, in Scenario s3, the upstream utilization is not at 100% and the playback lag and failed playback share are high.

4.2.3 Overhead

Since the protocol is fully time-based, the absolute overhead is constant, that is, it does not depend on the bit rate of the video stream, but rather on the length of blocks and segments.

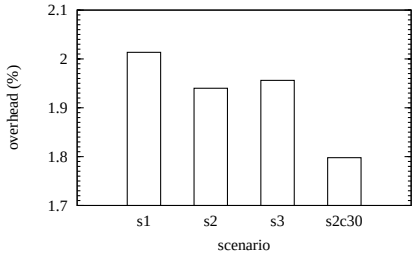


Figure 4.9: Overhead without DHT/DT

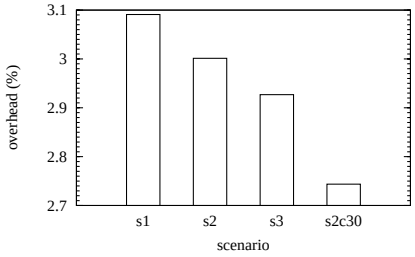
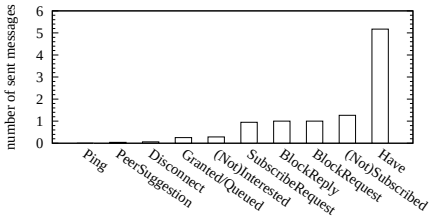
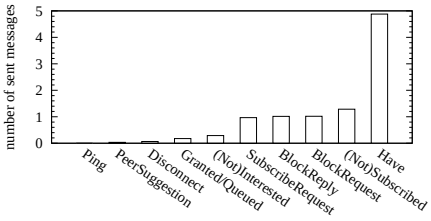


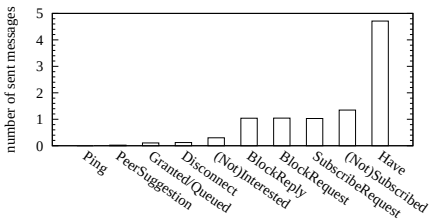
Figure 4.10: Overhead with DHT/DT



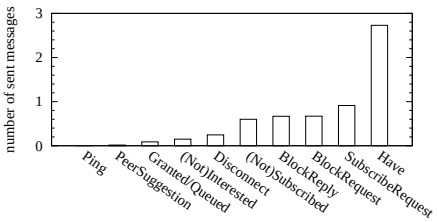
(a) Scenario s1



(b) Scenario s2



(c) Scenario s3



(d) Scenario s2c30

Figure 4.11: Sent messages in all scenarios

Figure 4.11 gives an insight into the running protocol, displaying the average number of messages a peer sends per second, by type. `Have` messages are the most common ones, which is expected from the protocol design, but it is interesting to see that it is around 5 in scenarios s1, s2, and s3, even with the increasing number of peers. The average number of `Have` messages sent per second corresponds to the average $|S|$, which is limited according to Section 3.4.4.

All messages have negligible size (a few bytes) when compared to the `BlockReply`, since it carries the video block (several kilobytes). 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 4.9. Figure 4.10 including DT and DHT traffic, overhead is at most 3.09% on average. It is, at a first sight, counter-intuitive that Scenario s2c30 shows smaller overhead than all other scenarios. This is probably due to the average number of on-line peers being lower than in other scenarios, therefore the chance that $|S|$ reaches $\overline{S}$ is lower. This issue needs to be further investigated.

Concerning duplicate blocks received, the policy proposed in Section 3.4.2 keeps the level relatively low. Figure 4.8 shows that the share of duplicate blocks received is, similarly to the number of skipped blocks, decreasing in more bandwidth-restricted scenarios. This may be explained by peers having a smaller $|D|$, which in practice means less choice of

other peers to download from, thus less concurrent downloads from different peers and less probability of receiving duplicate blocks.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This technical report presents the architecture and 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 may be used with the new protocol, as well as evaluation results, revealing and discussing the main trade-offs encountered in building such a system. Results show that the proposed protocol and policies are able to maintain good QoE for the end user and respond well to churn, although QoE degrades in scenarios with more bandwidth restriction.

5.2 Future Work

While this represents an important first step into supporting the proposed use case, several questions remain open. Future work includes finding optimal policies, improving overall security, and developing an effective incentive mechanism to verify upload capacity of peers that may be applied in the proposed use case.

Acknowledgment

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

Bibliography

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

[2] vlcj. http://code.google.com/p/vlcj/, 2008.

[3] T. Bocek. TomP2P - A Distributed Multi Map. http://www.csg.uzh.ch/publications/software/TomP2P, 2009.

[4] 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.

[5] 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*, pages 71–84, New York, NY, USA, 2008.

[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*, pages 447–448, Piscataway, NJ, USA, 2009.

[7] 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.

[8] 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*, 9(8):1672–1687, December 2007.

[9] 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.

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

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

[12] J. J. D. Mol, A. Bakker, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips. The Design and Deployment of a BitTorrent Live Video Streaming Solution. *Intl. Symposium on Multimedia*, pages 342–349, 2009.

[13] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H. J. Sips. Give-to-Get: Free-Riding Resilient Video-on-Demand in P2P Systems. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6818, January 2008.

[14] J. Noh, A. Mavlankar, P. Baccichet, and B. Girod. Time-shifted streaming in a peer-to-peer video multicast system. In *GLOBECOM'09: Proceedings of the 28th IEEE conference on Global telecommunications*, pages 6025–6030, Piscataway, NJ, USA, 2009. IEEE Press.

[15] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103 –1120, sep. 2007.