Performance Analysis of the Libswift P2P Streaming Protocol

Riccardo Petrocco
Technische Universiteit Delft
Delft, The Netherlands
r.petrocco@gmail.com

Johan Pouwelse
Technische Universiteit Delft
Delft, The Netherlands
peer2peer@gmail.com

Dick H. J. Epema
Technische Universiteit Delft
Delft, The Netherlands
d.h.j.epema@tudelft.nl

***Abstract*—Video distribution is nowadays the dominant source of Internet traffic, and recent studies show that it is expected to reach 90% of the global consumer traffic by the end of 2015. Peer-to-peer assisted solutions have been adopted by many content providers with the aim of improving the scalability and reliability of their distribution network. While many solutions have been proposed, virtually all of them are at the overlay level, and so rely on the standard functionality of the transport layer. The Peer-to-Peer Streaming Protocol workgroup of the IETF has adopted the Libswift transport-layer protocol that is targeted at P2P traffic. In this paper we describe the design features and a first implementation of the Libswift protocol, and a piece-picking protocol that uses the transport features of Libswift in an essential way. We investigate its performance on both high-end and power-constrained low-end devices, comparing it to the state-of-the-art in P2P protocols.**

I. Introduction

Over the last years, the demand for video over the internet has dramatically increased. A recent white paper released by Cisco [10] predicts that by the end of 2015, 90% of global consumer internet traffic will be video. In facing this growing demand, techniques that guarantee a good quality of service (QoS) to the end users are required. A popular choice for this has been peer-to-peer (P2P) assisted systems [25, 25, 28, 29], in which each end user contributes to other interested viewers by sharing his downloaded content. P2P assisted systems offer an efficient way of increasing the scalability of a system by reducing the load on the distribution infrastructure, and therefore, on the content provider. To satisfy the need for a standard for distributing video content over P2P networks, the Internet Engineering Task Force (IEFT) has been working towards defining a new Peer-to-Peer Streaming Protocol (PPSP), which has resulted in the Libswift protocol being adopted as the working group reference. In this paper we present the first implementation of current draft, and a performance comparison with existing BitTorrent-based P2P clients.

The most popular way of sharing content in a P2P fashion is currently the BitTorrent protocol, which is a natural candidate for the design of P2P assisted systems. Originally designed for *file downloading* with the goal of retrieving the *entire* content as soon as possible, much work has been done on adapting BitTorrent-like protocols to fit the needs of video-on-demand and video streaming [4, 14, 17, 18, 21]. Most of the proposed changes have affected the download scheduler (usually called the piece picker), the upload policies, and the overlay networks, but have relied on the standard transport protocols in the internet.

In this paper we provide a performance comparison of the Libswift protocol with popular BitTorrent-based clients on both high-end and power-limited devices. We propose Libswift [19, 23] as an appropriate candidate for *media streaming* applications. Libswift is a content-centric multi-party transport protocol that has been designed with flexibility and lightness in mind. It allows NAT traversal, small message-passing overhead, and has a novel data structure that allows a very small per-connection cost. It differs from most of the existing P2P systems as it lies at the transport layer rather than at the application layer. While most of the existing solutions use standard transport layer protocols designed for client-server networks such as TCP, Libswift provides a protocol specifically crafted for P2P networks, removing features that are not needed for video distribution such as in-order retrieval. Furthermore, we present an algorithm for selecting peers from which to request time-critical data, an algorithm for ordering data requests that guarantees upload fairness to all requesting peers that applies a form of Weighted Fair Queuing (WFQ), and a downloading algorithm that takes peer locality and robustness into account. Each of these three algorithms exploits information available at the transport layer of Libswift.

We have implemented a realistic testing framework that executes the P2P clients that we compare on a multicluster machine, which gives us a chance to evaluate Libswift's behaviour in a real world environment, rather than relying on simulations. Furthermore, we provide a comparison of the Libswift client with popular BitTorrent-based clients, providing more insight into its strengths and weaknesses.

The main contributions of this paper are the following:

- We describe the design features of the Libswift protocol (Section II).
- We present a peer selection and upload request ordering policy for video-on-demand that use Libswift's transport-layer features in an essential way (Section III).
- We perform an experimental evaluation of Libswift's performance on high-end and power-constrained devices, and compare it with popular, state-of-the-art BitTorrent-based protocols (Sections IV and V).

II. LIBSWIFT

This section describes several design characteristics of Libswift which are required to understand the algorithms and experiments presented in this paper. A full description of the protocol is available at [7].

A. Design Features

The Libswift protocol has been designed with simplicity and lightness in mind. It can be run over any transmission protocol [19], allowing each provider to choose its own mechanism depending on the specific needs. Libswift operates at the transport layer rather than the application layer, and given its small footprint, it can be easily integrated into the operating-system kernel as an additional transport protocol. It features very short start-up times and it has been designed for in-order and out-of-order download with a particular focus on real-time streaming.

For congestion control, the Low Extra Delay Background Transport (LEDBAT) [27] approach has been implemented and integrated into our reference implementation. LEDBAT has been designed to provide the best possible "background" transmission, avoiding interference with TCP foreground traffic on the same network link. The algorithm has proven to be successful and has already being integrated into some BitTorrent-based clients [6]. This behaviour allows Libswift to be run in the background, without the side effects of a noticeable delay when traffic generated by other applications, such as a web browser, needs access to the link. LEDBAT is the ideal congestion control mechanism for file-sharing and video-on-demand (VoD) content, while for live-streaming a different mechanism might be applied.

Content integrity is provided by employing Merkle hash trees [8, 24]. All the hashes needed to validate the received packets are sent along with the actual content, allowing for immediate verification. In our implementation, the Merkle tree scheme is applied for every download scenario, offline file downloading, VoD, and live streaming, but the modularity of Libswift allows for an easy integration of different mechanisms. As a last note, Libswift has been designed to support both push and pull strategies, but for the scope of the experiments presented in this paper, we implement Libswift as a push protocol, providing consistent and reliable results.

The reference implementation currently consists of about 8K lines of code, and has not yet been fully optimised. It uses the UDP protocol because of its simplicity, lack of connection set-up delays, and retransmission. UDP also fulfils the requirements of time critical applications, such as video streaming, and we therefore consider it to be the best candidate for our reference implementation.

B. Binmap

Every P2P system needs a way of representing the local availability of the downloaded content to other peers participating in the swarm. BitTorrent, which splits the content into a number of data blocks, usually about a thousand, addresses this issue by using a bitmap in which every bit represents a successfully downloaded block. Peers exchange their bitmaps to detect who already retrieved any missing blocks. While this approach guarantees a constant and easy way of sharing information, it does not scale and is not flexible enough to provide any level of aggregation of information. Being a plain array of bits, the cost of sharing this information is constant and does not scale over time.

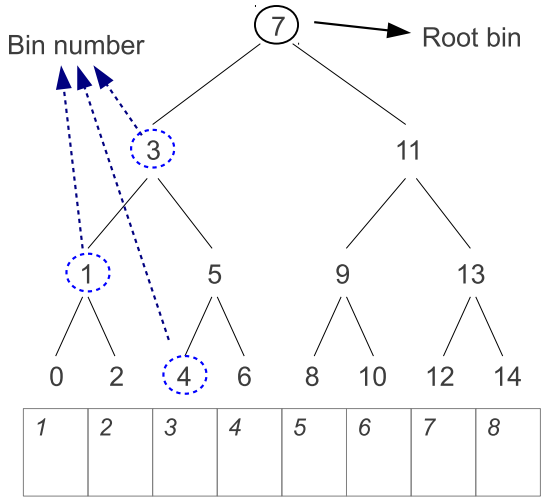

Fig. 1: The binmap for a file divided into 8 blocks.

In Libswift, to prevent the propagation to every node of redundant information, a binary tree is built on top of the bitmap every node of which represents a specific data range. The nodes in this tree, which is called a *binmap* [31], represent the availability of their left and right child nodes. If a consecutive range of data has the same status, either available or not, this information can be aggregated in a node in the tree that is at a higher layer, offering an easy way to identify missing ranges of data without having to inspect the entire binmap. Once the information of a node, also called binary interval or *bin*, is aggregated in a parent node, it is removed from the tree, thus reducing the size of the tree and space in memory. Each bin in the binmap is represented by a 16-bit value, therefore, while the leaves of the binmap have a value of either all 0 or all 1, their information is propagated to their parent bins, up to the fourth layers. This structure is repeated at the higher levels of the tree, with the root of the tree indicating whether all blocks are available or not. The novel binmap data structure merges the concept of bitmaps with the one of binary interval, and represents an efficient way of storing information about the data availability of each connection, reducing the per-connection costs. This approach reduces the overhead in peer communication, allowing to request data and announce the local availability with a single number, the bin number.

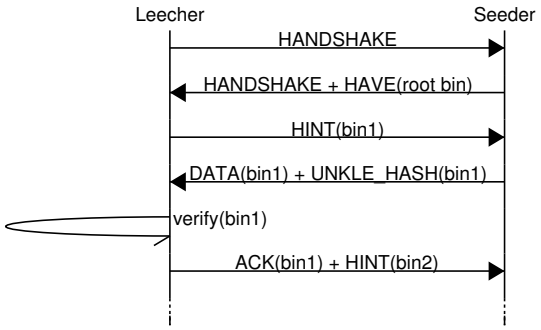

Fig. 2: Sequence diagram of the initial interaction.

An example of such a binmap for a file consisting of 8 blocks is presented in Figure 1. In this example bin 7, called the root bin, represents the entire content from block 1 to block 8, while for instance bin 3 represents blocks 1 to 4. The default block size in Libswift is 1 KByte, but the system can easily increase the size depending on the environment and size of the content. The choice of 1 KByte data packet allows to avoid fragmentation while adding to the message the local availability, acknowledgements for previously received packets and requests for new data blocks, all represented by bin numbers.

C. Peer Communication

As previously stated, Libswift operates at the transport layer, and as it is specifically targeted at video distribution in P2P systems, it only has messages for the discover of new peers, for establishing a connection, and for requesting content.

Peer discovery in Libswift is done through Peer exchange (PEX), enabling peers to exchange peer addresses with each other in order to reduce the load on the (optional) trackers. The communication between peers is performed with a small set of messages. These represent the initial handshake (HANDSHAKE), the announcement of local availability of a piece (HAVE), acknowledgements of received packets (ACK), packets containing data (DATA) with its uncle hashes needed for verification (HASH), and requests for new content (HINT). The quality of the connection with each peer is constantly being monitored and predicted, based on the arrival of data, the round trip time (RTT), and the stability of the end-to-end link. The binmap data structure used to record the content availability presented in the previous section allows for a very low per-peer communication cost. The request for hardware resources is highly dynamic and is directly influenced by the level of data fragmentation in the swarm.

Figure 2 shows the initial interaction between a peer and a seeder. The key characteristic is the aggregation of information within the same message. This design allows peers to start requesting data already at their second interaction (HINT), greatly reducing the warm-up time and therefore leading to short start-up times in streaming environments.

III. Download Scheduling

In this section we present the design principles of Libswift's download algorithm, which has been designed with a focus on providing video-on-demand capabilities, such fast pre-fetching or buffering, and low start-up delays. The download algorithm, presented in Section III-C, selects *fast* peers from which the high-priority data is requested. Section III-A presents the ranking system used to select fast peers and Section III-B presents the upload strategy introduced to achieve fairness, to have the opportunity to limit the upload capacity, and to provide a better bandwidth estimation.

A. Peer Ranking

P2P clients usually select peers to which to send requests for data randomly. The importance of selecting the right peers when playing a video stream retrieved through a P2P network has been proven by many existing systems [13, 21]. Many alternative upload strategies have been proposed [18, 30] that allow requesting *urgent* data, which is prioritised over other requests, from a selected group of peers. This behaviour guarantees that urgent data will be sent before their deadline, but it assumes an environment without selfish peers. This can however not be assumed for the Libswift system, because as the code and its policies are easily accessible to the users, selfish peers might modify their policies and mark all of their requests as urgent. While most of the solutions design an overlay network to organise the peers, we have direct access to the transport layer, and therefore, we can take advantage of direct information about the actual received bandwidth and other values such as the packet loss rate. Exploiting this information, we can select peers that provide the best end-to-end links, reducing the probability of requesting urgent data from slow or unreliable peers.

The quality of a link is calculated based on the measured bandwidth speed as derived from the RTT and the packet inter-arrival time, and the packet loss rate. The information we need is already available as every peer continuously monitors the acknowledgements for the data it sends to other peers in the swarm, and, in order to apply the LEDBAT congestion control [27], the average RTT, the bandwidth capacity, and the packet loss rate are constantly updated.

Every peer then ranks all its neighbour peers according to the value of

$$R_i = S_i - k \cdot S_i \cdot L_r \quad \text{, where} \quad S_i = P \cdot \frac{1 - RTT_i}{DIP_i}, \qquad (1)$$

where R_i is defined as the expected bandwidth of peer i, DIP represents the packet inter-arrival time, P represents the packet size, L_r represents the loss ratio defined as the ratio between the numbers of lost and received packets, and k is a variable parameter that determines the influence of the packet loss rate on the bandwidth estimation. The variable k is initialised at 1, and incremented if the peer shows to have an unstable connection, resulting in big variations of the derived RTT and DIP values. Peers then sort their neighbour peers according

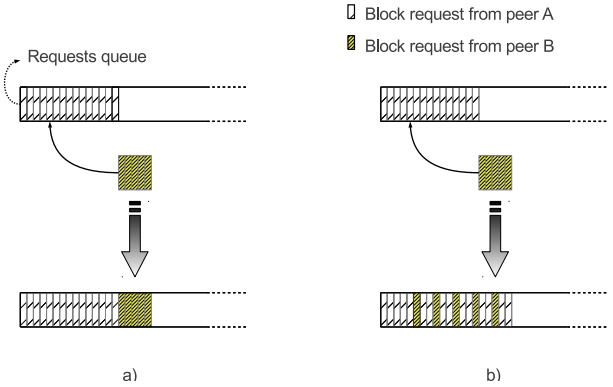

Fig. 3: Scheduler for incoming requests. a) The original scheduler. b) The new scheduler for fairness and load distribution.

to decreasing expected bandwidth, and select the first n peers for which

$$\sum_{i=1}^{n} R_i \geq x \cdot BR_v, \tag{2}$$

where BR_v is the average video bit-rate and x is a dynamic factor that varies based on the experienced playback behaviour, e.g., the number of playback stalls. These n peers represent the set of *fast peers*, to which the download algorithm will turn when requesting urgent data.

B. Upload Policy

The design of Libswift [7] does not specify a specific upload strategy when serving incoming requests. The requesters decide how much data to request based on the measured network delay, following the LEDBAT approach [27]. The initial design of LEDBAT did not take fairness in serving requests into account, but recently its authors have presented [9] several strategies to do so. The proposed solutions provide a fair distribution of the available bandwidth to concurrent LEDBAT streams, always aiming to fully utilise the available bandwidth. While this approach is well suited for high-end devices, maximising the upload stream is not always the best approach for low-end devices, such as mobile phone and set-top boxes, where resources are usually expensive. In order to be able to limit the upload capacity, we approach the problem of fairness from a different perspective. One of the main goals of Libswift is to provide a protocol that, given its small footprint, can be easily integrated on low-power devices such as mobile phones and set-top boxes. To limit the upload bandwidth utilisation in environments where resources are scarce, either because of network costs or to offload the hardware during a video playback, we present a new approach that schedules the stream of outgoing packets equally among the connected peers.

We replace the original upload queue management that was serving requests following a FIFO approach, see Figure 3-a, with a more elaborate one to provide fairness and a good level of bandwidth estimation. We apply the well known Weighted Fair Queuing (WFQ) scheduling technique [12], that fairly distributes the available bandwidth to all the connected peers. The weight used in the WFQ algorithm is assigned per peer, depending on the amount of outstanding requests. If the upload speed of the sender is limited, and the limit is reached, the sender will delay serving the requests, introducing a delay specific to each peer, and proportional to the amount of his outstanding requests. The requester, on the other hand, will notice an increasing delay in the responses and will decrease the rate of his requests. Ideally, the algorithm reaches a stable state in which all requests are served with the same delay, and all peers sends the same amount of requests. If the requester is expecting urgent data, it will cancel his outstanding requests, that might be asked from other connected peers, and reduce the size of the next request to guarantee its delivery on time.

In this way it becomes trivial to predict load changes on the sending peers, and it gives a more accurate bandwidth estimation needed for the peer scheduler, see above.

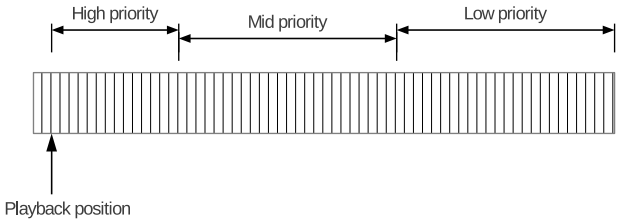

Fig. 4: Content divided into three different priority sets.

C. The download algorithm

For the download algorithm, each peer maintains for every swarm it participates in a so-called *availability tree* in which it keeps track of its view of the global data availability in the swarm. In fact, the availability tree of a peer represents the data availability among all peers it is connected to. It has the same structure and size as the binmap in its fully expanded form, and it keeps track of the percentage of availability of each bin, or node, of the binmap tree. The cost of maintaining this tree is very small as it is only updated when new availability information is received from a neighbour peer. The availability tree allows the download algorithm to detect, and target for download, bins that have a low level of replication. This approach will balance the data availability in a swarm, and will facilitate data exchange between peers.

The download algorithm divides the video stream into three sets, a high-, a medium-, and a low-priority set [16], see Figure 4. The high-priority set starts at the current playback position and selects data in-order, as it represents the most urgent data in a video stream. If nothing can be downloaded from the high-priority set, the algorithm selects data from the mid- and low-priority sets in a rarest first fashion to increase their availability in the swarm. In the high-priority sets data are retrieved as soon as possible, requesting content form the set of *fast peers*

defined in the previous section, which provide the best end-to-end link quality. The size of the requests depends on the number of fast peers currently in the set and the queue delay calculated given by LEDBAT. In particular it is defined as:

$$\min\left(\frac{W_{ledbat}}{2}, \frac{H}{F}\right), \quad (3)$$

where W_{ledbat} is the size of the congestion window, H is the size of the high-priority window, and F is the number of fast peers currently in the set. The algorithm keeps track of each request that has been sent, and assigns a deadline to it based on the current playback position and the loss ratio. The deadline for packets from the high priority is always assigned with a safety margin, allowing to forward the same request to a different peer in case the it is not retrieved on time.

The higher the speed and the reliability calculated from previous data transmissions, the bigger the requested data blocks. This is a similar approach to the windowing system in the TCP protocol. Libswift constantly monitors several parameters, such as packet losses and average round trip times (RTT), trying to fully utilise the available link capacity. When downloading from the low-priority set, the algorithm iterates through the local binmap to identify missing data, through the remote peers' binmaps to see what data can be retrieved, and through its availability tree, selecting the rarest bin in the swarm.

IV. Experimental set-up

In this section we present the environment, the scenarios, and the metrics we use to evaluate the performance of Libswift in comparison to existing solutions. Section IV-A describes the experimental set-up for assessing the download algorithm presented in Section III, Section IV-B does so for the comparison of Libswift with BitTorrent-based protocols, and Section IV-C does so for the evaluation of the performance of Libswift on power-limited devices.

A. Algorithm Comparison

In this section we describe the environment and scenarios we use to evaluate the impact of the newly introduced scheduling algorithm. We perform a series of experiments in which we analyse the behaviour of Libswift in terms of efficiency, reliability, and its capabilities of providing deadline-based streaming content. Table I presents the scenarios we use to investigate Libswift's behaviour, characterized by different proportions of seeders and leechers, with the size of the swarm ranging from 32 to 208 peers. We assume peers to be selfish, leaving the swarm as soon as they complete the download or reach the end of the playback. We compare the algorithms in this scenario, since swarm that provide an over-supply of bandwidth don't allow to easily identify misbehaving algorithms. The only seeder in the swarm is the initial content provider, which distributes a video stream encoded with an average bitrate of about 1Mbit/s.

To simulate the content playback, we implement a video player that starts consuming the content provided by the download engine as soon as the initial 5-second buffer has been filled. This initial set of data is requested by the media player in an aggressive way in order to reduce time till playback, thus stressing the download engine. After the buffer has been filled, the player requests the content at its average bitrate, simulating normal playback. The media player stops consuming content only when it reaches the end of the stream or when the engine cannot provide the requested content, in which case it will pause and resume once the buffer has been filled again. This behaviour completely decouples the media player from the transport layer, as the download engine has no control over the rate at which the content is consumed. Furthermore, it simulates a real world scenario, in which existing mediaplayers can use Libswift as their transport protocol.

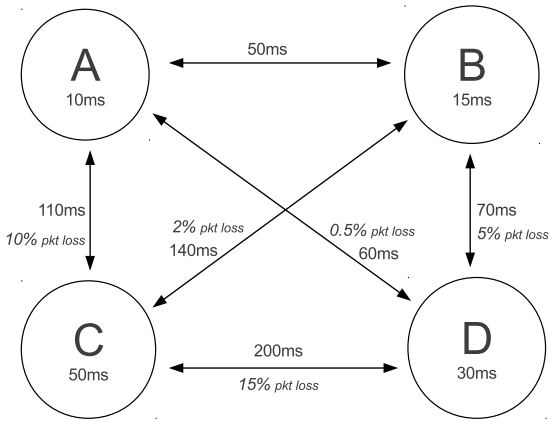

Fig. 5: Artificial delay and packet loss rate introduced between group of peers.

We compare our VoD algorithm with two alternative approaches, a baseline algorithm, referred to as *linear*, and a standard priority-based VoD algorithm, referred to as *std-vod*. The first retrieves data in order, while the second divides the content into the same priority sets as our algorithm, retrieving data in order from the high-priority set and in a rarest-first fashion from the mid- and low-priority sets. Both algorithms do not apply any kind of peer selection, and forward the requests to random connected peers.

To evaluate the performance of our VoD algorithm presented in the previous section, we divide the leechers into four groups distinguished by different connection speeds. For each group we introduce an artificial delay and a packet loss rate for each outgoing network packet, except for the seeder for which no packet loss rate is introduced. Packets sent by the seeder are characterised by the peer's delay and an additional 5ms latency. The delays assigned to each group, and introduced between the groups, are presented in Figure 5. The seeder distributes the content over a 20Mbit/s network link, while the upload rate of the leechers is limited to 1.2Mbit/s.

Our experiments are performed on the Distributed ASCI supercomputer [15] (DAS4), which has dual quad-core 2.5

TABLE I: Experiment scenarios showing the number of leechers in the swarm.

Libswift vs. BitTorent. 4 Leechers per Node										
4	8	16	24	32	48	64	96	128	176	208
Libswift VoD algorithm comparison. 8 Leechers per Node										
		32	48	64	96	128	176	208		

GHz machines connected with Gbps links, to guarantee a stable environment providing consistent results and avoiding bottlenecks introduced by limited network capabilities and lack of hardware resources. Depending on the specific experiment presented in Table I, we execute up to 8 clients simultaneously on each DAS node, considering that our current prototype implementation is single threaded and each process fits well within the boundaries of a single core's capabilities.

B. Libwift versus BitTorrent

To evaluate the performance of the Libswift protocol we compare it to the most popular P2P protocol, BitTorrent [11]. For our comparison we have selected the uTorrent client [6], as the most widely used P2P client [32], and the libtorrent library [2], which is currently in use by several popular projects (e.g., Limewire [3], Deluge torrent [1] and Miro [4]) for its lightness and good performance. For our experiments we select the latest Windows version of uTorrent, as it proves to be more stable and provides more consistent results than the outdated Linux version of the client, and we implement a small client composed of few lines of code as an interface to the libtorrent library. For both BitTorrent-based clients we disable optional features, such as DHT support or the decline of connections from clients running the same host, where possible. Furthermore, we enable features such as prioritising partial pieces, to increase sharing.

In the experiments we measure the time needed to complete the download and the bootstrap time, that is, the initial start-up delay until the playback starts, which has a great impact on the quality of experience (QoE). Furthermore, we investigate the capacity of the clients to provide consistent results across all the peers in the swarm. During this set of experiments we do not implement a player that consumes the content in real-time, as those functionalities are not easily accessible on all clients. Libswift downloads data according to the algorithm of Section III, with the high-priority window set to 10 second, which corresponds to 11.4MB, while the remaining content is retrieved in a rarest-first fashion. Libtorrent has policies to download the entire content either sequentially or in rarest-first mode. Even through we could set the download policy to retrieve the initial set of pieces in order, and than change it to rarest first, we decide to retrieve the entire content in a rarest first fashion, as this represents libtorrent's default behaviour. UTorrent, on the other hand, offers a streaming feature, but unfortunately it cannot be enabled from the integrated web interface we use to retrieve the download progress. Thus, when measuring the start-up delay for uTorrent and libtorrent, we consider the first 11.4MB downloaded from the *entire* content, giving the clients a clear advantage over Libswift, as the retrieved content doesn't need to be at the beginning of the file.

Also for these experiments, we use the DAS4. For the BitTorrent clients we execute the content provider and seeder on the same node as the tracker, using the popular opentracker [5], while the clients are executed on remaining nodes following the scenarios presented in Table I. For Libswift, even through a tracker is considered to be optional and can be used to speed up the peer discovery phase, we consider PEX to be sufficient for our needs and rely on the initial seeder for the dissemination of peer addresses to new-comers.

We run each of the scenarios of Table I applying two inter-arrival patterns. In the first, peers join the swarm roughly at the same time, simulating a flashcrowd scenario. In the second arrival pattern, the peers are started following a Poisson interinter-arrival process with a rate of 1 peer per second, simulating a steady-state scenario. In all the presented scenarios, peers are initialised only once the seeder and the tracker are ready to receive incoming requests. The testing framework schedules the arrival of the first peer 30 seconds after the seeder and tracker have been initialised, giving the seeder enough time to perform the needed integrity checks of the 1GB seeding content. We sample the progress with 1 second accuracy, starting once the .torrent file is added to the client in the case of uTorrent, or at execution time for Libswift and Libtorrent as they receive it as an execution argument.

C. Libswift on Power Limited devices

In this section we present the metrics and scenarios we use to evaluate Libswift when executed on power-limited devices. In order to do so, we analyse its performance when running Libswift on a 450Mhz set-top box developed by Pioneer in the context of the P2P-Next project [26], and on a Samsung Galaxy Nexus smart phone running the Android platform.

During the P2P-Next project, the Pioneer Digital Design Center in London developed a low end set-top box (STB),called NextShareTV, featuring HD streaming provided through a P2P network. The initial implementation used a BitTorrent based download engine to provide the stream to the decoder, and ultimately to an HD TV. Later versions adopt Libswift as the main P2P client. To evaluate both download engines, we compare the ability to reach high download bitrates given the hardware constraints. Experiments are run by the Pioneer R&D team in an isolated environment where 7 STB peers receive the stream from the initial content provider, represented by a PC client. All the peers are connected to each other over a 1Gbit/s link, and, as for our previous experiments, clients are started within 1 second to each other. Furthermore, different from the previously presented scenarios, the retrieved stream is actually sent to a player and displayed on the connected TV, increasing the load on the hardware.

On the Android platform we evaluate the performance of Libswift in comparison to the most popular VoD application, YouTube. We implement a small interface between the Libswift protocol and the default media player available on

the Android platform, to compare Libswift's capabilities to download and simultaneously playback.To evaluate the performance of the compared applications, we measure the power consumption when downloading, while playing, a 58 second 720p video content composed of about 14MByte over the same wireless network link. To provide an equal comparison of Libswift with the YouTube application, we chose the same video encoded in mp4 format. This allows Libswift to stream the content to the integrated media player using the same native decoder used by the YouTube application. Considering the fact that in the Android environment each application runs independently in its own private virtual machine, measuring the CPU and memory load would not give us sufficient inside to the actual resource utilization. We therefore measure the power consumption, physically bypassing the battery, and consider it to be the most reliable metric to analyse the resources needed to execute each application. We apply a 0.33 Ohm high side shunt providing a 99% accuracy [20] for our measurements, and use a NI USB-6009 data acquisition card to retrieve the battery variations. Our results are averaged over 3 consecutive measurements, where between each measurement we reboot the device and wait 30 seconds after the application is initialised, starting the download only once it provides a constant reading. Even through the battery voltage readings are known to degrade over time, and can not be avoided [22], we consider the difference not significant enough to draw our conclusions.

V. Experimental results

In this section we present the experimental results of Libswift. Section V-A presents the results of retrieving streaming content when different download algorithms are applied. Section V-B presents the results of the comparison with existing BitTorrent-based P2P solutions, while Section V-C presents the performances on Libswift when run on hardware-limited devices.

A. VoD Algorithm Comparison

In this section we discuss the effect that different download policies have on the QoE of the final viewer. We consider the number of stalls experienced during playback as the most important metric to evaluate the final QoE. Figure 6 presents the average number of stalls that occurred during the playback. The worst QoE is obviously caused by the linear approach. This approach, despite it's simplicity, causes a very low level of scalability and information exchange between peers, as only peers further on in the playback are able to provide useful data to newcomers. The main reason for failing in establishing a fully scalable distributed network, is the lack of global knowledge about data availability in the swarm. The standard VoD algorithm, on the other hand, holds a global knowledge about data availability in the swarm, using the same availability tree data structure presented in Section III-C. The effect of retrieving data in a rarest first fashion, to increase its availability in the swarm, becomes quite evident in scenarios with more than 96 leechers. Libswift's download

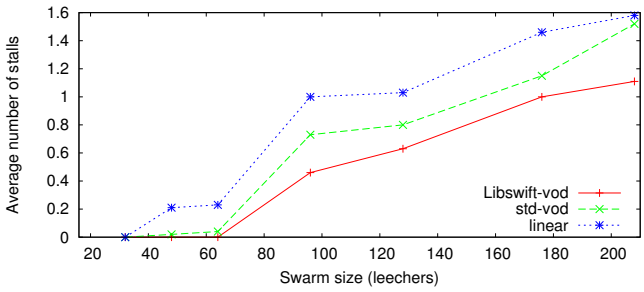

Fig. 6: Average number of stalls experienced during playback.

algorithm clearly outperforms the other approaches, selecting the best candidate peers to which request urgent data before its deadline, and in general, increasing the scalability of the system. As faster peers, belonging to group A and B of Figure 5, retrieve the content from the high priority set, they are selected by slow peers, from group C and D, as the best candidates for providing urgent data and increasing their QoE. We do not show the start-up time, as it proves to be very similar for all three algorithm, ranging from 3 to 7 seconds on average. This experiment also clearly demonstrates how P2P networks represent an efficient solution to increase the scalability of a system, taking fully advantage of the viewers available bandwidth.

B. Libswift vs. BitTorrent

For each of the figures presented in this section, the time refers to the relative time of each client. The time starts once the torrent file is scheduled for download, for the BitTorrent-based clients, or once the client has been launched, for Libswift. The presented results are averaged over 10 runs for each scenario and for each client.

Figure 7 shows the average time needed to complete the download in a flashcrowd scenario. On average Libswift performs right between the two compared clients, and all three clients present a similar level of scalability. For swarms that are characterised by a leecher to seeder ratio smaller than 32:1, Libswift presents better performance than Libtorrent. Both clients fully utilise the available bandwidth link, and the slightly faster download speed presented by Libswift is given by the smaller overhead in peer communication, as messages are sent within the same packet containing the actual data. For swarms that are characterised by a ratio of leechers to seeders higher than 48:1, libtorrent clearly outperforms Libswift, given its highly optimised code and algorithm that select rare pieces for download.

Figure 8 presents the interquartile range and the average start-up delay for the six biggest scenarios presented in Table I. We only investigate the scenarios with swarms composed of more than 48 peers, as those present more interesting results. The vertical axis of Figure 8 presents the start-up

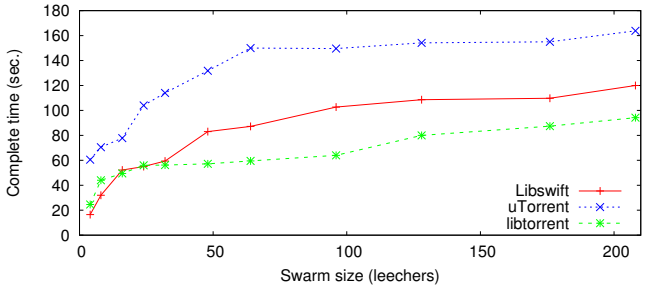

Fig. 7: The completion time in the flashcrowd scenario.

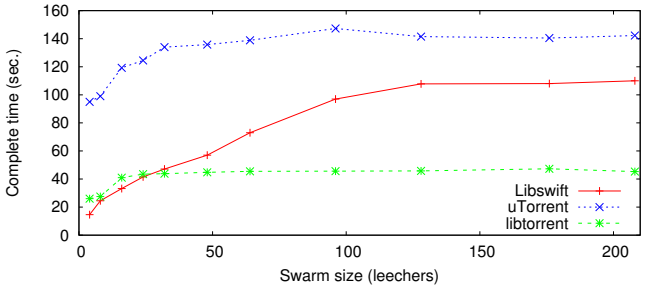

Fig. 9: The completion time in the steady-state scenario.

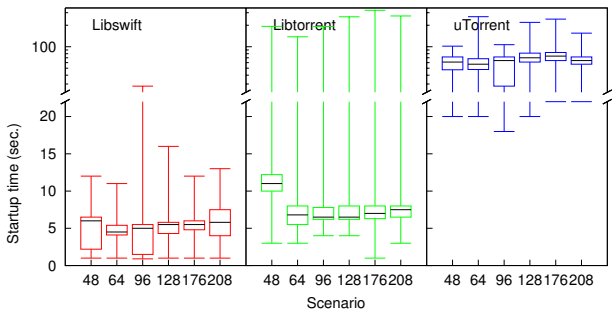

Fig. 8: The start-up time in the flashcrowd scenario.

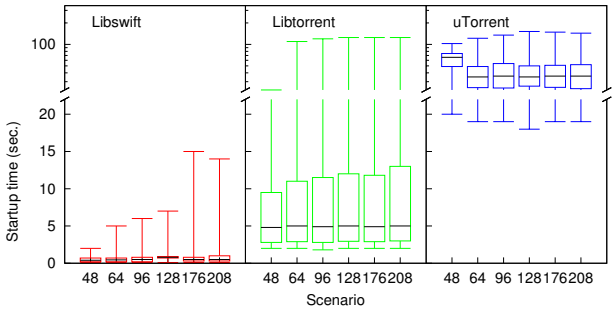

Fig. 10: The start-up time in the steady-state scenario.

delay using two different scales. The first 20 seconds are represented linearly, while the remainder is displayed on a logarithmic scale. This allows us to show in great detail the most interesting results, providing a higher QoE to the final viewer, while keeping worst results on the same graph.

It is clear that on average, Libswift and Libtorrent appear to have similar performance, but there is a great difference in the variability of the results. While all Libswift clients start almost always within the first 15 seconds, a small percentage of Libtorrent clients will have to wait more than one minute before being able to start the playback. UTorrent clients, on the other hand, experience a lower QoE, having to wait more than a minute on average before retrieving the first 10 seconds of video and being able to start their playback.

Figure 9 shows the average download time when peers join the swarm following a Poisson inter-arrival rate of 1 peer per second. In this scenario, Libswift presents slightly better results than in the previous one. This is clearly caused by the VoD oriented download scheduler. In the absence of a flashcrowd, peers are able to download all the data from the high-priority window, prioritised over the rest of the content, and start requesting data in a rarest first fashion, increasing the cooperation between peers. This behaviour is more evident in Figure 10, where the vast majority of Libswift peers retrieve the initial data needed to start the playback within the first second from their execution. As for the results presented in Figure 8, Libtorrent peers fill the initial buffer within the first 10 seconds, but a small minority still needs more than one minute for the buffer to fill. UTorrent clients also perform better when their arrival is linearly distributed over time, reducing by ∼20 seconds their average start-up delay, reaching a more acceptable value of 40 second.

Finally, Figure 11 presents the CDF of the completion time in the steady state scenario with 128 leechers and 1 seeder. Clearly, the average values presented in Figure 9 do not always reflect the real behaviour, as 25% of Libtorrent peers will experience a quite higher complete time. As a last note, while uTorrent appears in our results to be the worst client, it is also the more mature one, offering a quite more extensive set of functionalities compared to Libtorrent or Libswift. When only one client per node is executed, libswift is the fastest, retrieving the 1GB file in only 9 seconds and fully utilising the Gbit connection, but its performance degrades a bit when the link is shared with other clients or applications. Obviously, our implementation being just an initial prototype, we can not compete with the maturity of Libtorrent and uTorrent. Nevertheless, we analyse the suitability of the clients in a highly demanding streaming context with the sole aim of retrieving the content as soon as possible.

C. *Libswift on Power Limited devices*

The low footprint of Libswift allows for an easy integration on hardware-limited devices.

Figure 12 presents the results of running Libswift as the transport protocol for the NextShareTV STB. During the P2P-

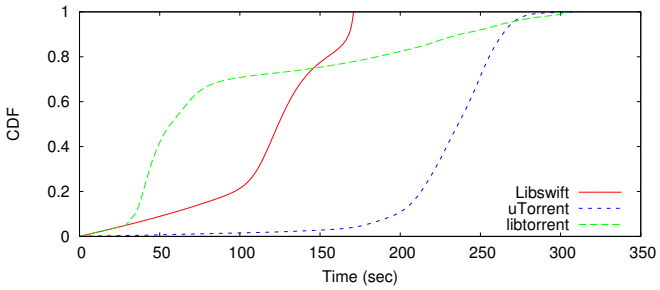

Fig. 11: CDF of the completion time in a steady-state scenario with 128 leechers

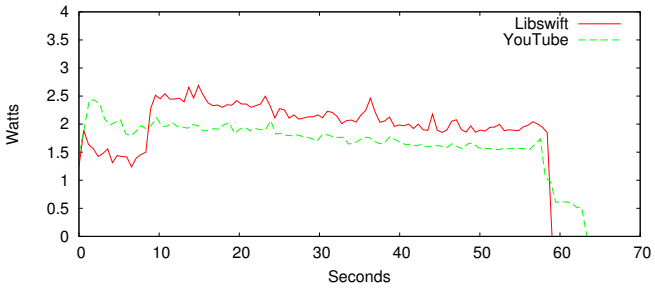

Fig. 13: The power consumption of Libswift and YouTube when playing a 720p video stream on an Android smart phone.

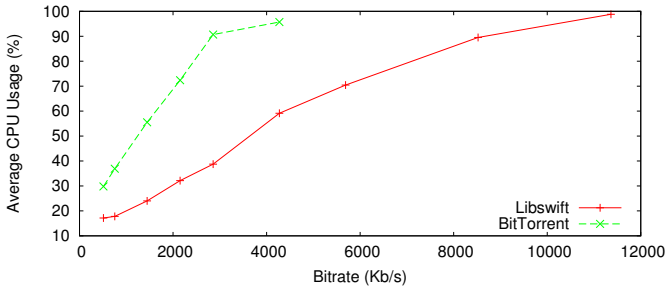

Fig. 12: A comparison of the NextShareTV STB running Libswift and a python-based BiTorrent client as download engine.

Next project, also real-world experiment have been conducted, but unfortunately at that time only the original python-based BitTorrent client had been integrated into the STB. It is obvious that the performance of Libswift is far superior than the original BitTorrent client, even though much has been gained by moving from python code, needing more resources, to C++ code. Nevertheless, the integration of Libswift allows the STB to reach higher bitrates, offering a footprint of only 1MB for the code, and 0.5% of the content size for the Merkle hashes.

Figure 13 presents the power consumption when running the Libswift and the YouTube Android applications on the same hardware. the measurements are taken while downloading and playing the same 720p content in the same network conditions. The experiment shows that the YouTube application maintains a lower, more constant, power consumption, which is due to the fact that the application retrieves the content at a lower rate. On the other hand, the Libswift application is just a proof-of-concept, and has not been optimised for such a use. Therefore, it retrieves the content in an aggressive way, using fewer resources, but leading to a higher power consumption during playback. The aim of this experiment is not to prove the superiority of Libswift over the commercial application; instead, we want to show that Libswift, if optimised, can be a valuable alternative to provide the same QoE while distributing the content over a P2P network.

VI. Conclusions and Future Work

It this paper we have introduced the new P2P Streaming Protocol Libswift, which is a proposed IETF standard, explaining its design characteristics and goals. Furthermore, we have presented a download algorithm specifically crafted for retrieving streaming content that reaches a high level of QoE by exploiting the information available at the transport layer. We have validated the efficiency of our download algorithm by running realistic experiments on a multicluster machine, simulating a media player that consumes the content in real time. Results show that requesting urgent data to peers that provide a good end-to-end network link leads to a better QoE, reducing the occurrence and duration of stalls during playback. To investigate the performance of Libswift in general terms of content retrieval and streaming capabilities, we provide a comparison with two of the most popular BitTorrent-based clients, uTorrent and Libtorrent, in several scenarios. We show that Libswift outperforms the compared clients when the ratio of seeders to leechers is relatively low, and in general provides more consistent results among all peers of the swarm. On the other hand, it is yet not fully optimised, and it does not reach the same level of scalability as the highly optimised libtorrent. Future work will focus in improving the scalability of Libswift, by investigating different strategies to keep track of the availability of data in the swarm in high churn scenarios. We also analyse the time it takes to start retrieving the content, and provide detailed results that prove the low start-up time required by Libswift, making it an ideal candidate for time-critical applications.

We demonstrate that Libswift is a valuable alternative to standard P2P systems in terms of speed and efficiency, while still providing the key functionalities of a streaming protocol such as low warm-up times. Finally, we provide an evaluation of Libswift's performance when run on hardware-limited devices, such as mobile phones and set-top boxes, demonstrating its adaptability as a transport protocol in different environments.

ACKNOWLEDGMENT

The authors would like to thank Victor Grishchenko for the initial design and implementation of the Libswift protocol, Arno Bakker for comments and suggestions, and the Pioneer Digital Design Center in London for the results of the experiments regarding the STB. This work is supported in part by the European Commission in the context of the P2P-Next project (FP7-ICT-216217) [26].

REFERENCES

[1] Deluge torrent. http://deluge-torrent.org/.

[2] Libtorrent. http://www.libtorrent.org/.

[3] Limewire. http://www.limewire.com/.

[4] Miro. http://www.getmiro.com/.

[5] Opentracker. http://erdgeist.org/arts/software/opentracker.

[6] the utorrent bt client. http://www.utorrent.com/.

[7] A. Bakker and R. Petrocco. Peer-to-peer streaming peer protocol (ppspp), June 2012. IETF draft.

[8] Arno Bakker. Merkle hash torrent extension. BitTorrent Enhancement Proposal 30, Mar 2009. http://bittorrent.org/beps/bep_0030.html.

[9] Giovanna Carofiglio, Luca Muscariello, Dario Rossi, and Silvio Valenti. The quest for ledbat fairness. *CoRR*, abs/1006.3018, 2010.

[10] Cisco. Cisco visual networking index: Forecast and methodology, 2010-2015. White paper, June 2011. http://www.cisco.com.

[11] B. Cohen. Bittorrent protocol 1.0. http://www.bittorrent.org.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19:1–12, August 1989.

[13] T T Do, K A Hua, and M A Tantaoui. P2vod: providing fault tolerant video-on-demand streaming in peer-to-peer environment. *2004 IEEE International Conference on Communications IEEE Cat No04CH37577*, 3pp(c):1467–1472, 2004.

[14] A. Sentinelli et al. A survey on P2P overlay streaming clients. In *Towards the Future Internet - A European Research Perspective*, pages 273–282, 2009.

[15] Henri Bal et al. The distributed asci supercomputer project. *SIGOPS Oper. Syst. Rev.*, 34(4):76–96, October 2000.

[16] J.J.D. Mol et al. Give-to-Get: Free-riding-resilient video-on-demand in P2P systems. In *Multimedia Computing and Networking*, volume 6818, San Jose, USA, 2008.

[17] N. Capovilla et al. An architecture for distributing scalable content over peer-to-peer networks. In *MMEDIA'10*, pages 1–6, June 2010.

[18] Z. Liu et al. LayerP2P: Using layered video chunks in P2P live streaming. *IEEE Trans. on MM*, 11(7):1340–1352, August 2009.

[19] V. Grishchenko et all. On the design of a practical information-centric transport. Technical report, TUDelft, Deft, The Netherlands, 2011.

[20] P. M. Glatz, L. B. Hoermann, C. Steger, and R. Weiss. A System for Accurate Characterization of Wireless Sensor Networks with Power States and Energy Harvesting System Efficiency. In *Proceedings of the Sixth IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing*, page 468 473, 2010.

[21] Yang Guo, Saurabh Mathur, Kumar Ramaswamy, Shengchao Yu, and Bankim Patel. Ponder: Performance aware p2p video-on-demand service. In *GLOBECOM*, pages 225–230, 2007.

[22] L. B. Hoermann, P. M. Glatz, K. B. Hein, and R. Weiss. State-of-Charge Measurement Error Simulation for Power-Aware Wireless Sensor Networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference Mobile and Wireless Track*, 2012.

[23] Libswift. http://libswift.org/.

[24] Ralph Charles Merkle. Secrecy, authentication, and public key systems. Stanford, CA, USA, 1979. Stanford University. Ph.D. Thesis.

[25] PPSTream. http://www.ppstream.com.

[26] The P2P-Next Project. Fp7-ict-216217. http://www.p2p-next.org.

[27] S. Shalunov. Low extra delay background transport (ledbat). In *IETF Draft*, March 2009.

[28] SOPCast. http://www.sopcast.org.

[29] TVAnts. http://www.tvants.com.

[30] G. Simo U. Abbasi and T. Ahmed. Differentiated chunk scheduling for p2p video-on-demand system. In *8th IEEE International Consumer Communication & Networking Conference*, Las Vegas, USA, January 2011.

[31] Johan Pouwelse Victor Grishchenko. Binmaps: Hybridizing bitmaps and binary trees. Technical report, TUDelft, Deft, The Netherlands, 2011.

[32] Chao Zhang, Prithula Dhungel, Di Wu, and Keith W. Ross. Unraveling the bittorrent ecosystem. *IEEE Trans. Parallel Distrib. Syst.*, 22(7):1164–1177, July 2011.

Summary Review Documentation for "Performance Analysis of the Libswift P2P Streaming Protocol"

Authors: Riccardo Petrocco, Johan A. Pouwelse, Dick Epema

REVIEWER #1

Comments essentially relate to presentation issues.

Strengths: The strengths are (1) the importance of the work to the very relevant task of streaming media; and (2) the presentation of the paper, which is well written and generally easy to understand.

Weaknesses: The weaknesses are (1) the incomplete evaluation as authors admittedly compare a resource hungry implementation of BitTorrent to Libswift, and there does not seem to be a clear conclusion from the energy usage comparison on the mobile phone; (2) the fact that the paper compares Libswift to standard BitTorrent clients, and not those adapted/modified for use in video streaming (ce.g., ompletion time is not directly relevant to media streaming); and (3) the lack of comparison of Libswift to other media streaming systems in the literature.

REVIEWER #2

Comments expand with additional details on the weaknesses of the paper.

Strengths: The main strength is (1) the implementation of a complete system with a realistic evaluation.

Weaknesses: The weaknesses are (1) the unclear benefits with respect the current state of the art (engineering details vs. conceptual contribution); (2) the too narrative style of the paper; and (3) the lack of characterization of what are the conceptual innovations brought by the paper.

REVIEWER #3

The comparison with UTorrent is unfair because the streaming mode was off, avoiding the use of a piece selection algorithm specific for streaming.

Strengths: The strengths are (1) the importance of the paper, which proposes a downloading scheme fundamental for VoD; (2) the prototype used to show experimentally that the VoD implementation provides reasonable performance; and (3) the fact that Libswift is associated with standardization work within IETF PPSP WG, which may broaden the applicability of results.

Weaknesses: The weaknesses are (1) the poor separation between novel content and what has been /proposed elsewhere, e.g., in the draft describing PPSP; (2) the focus of performance evaluation that does not study VoD streaming as would be expected, but instead mixes different goals of streaming and file sharing; and (3) the lack of analysis and comparison with related work.

REVIEWER #4

The paper does a good job at discussing the design rationale of Libswift and, most importantly, evaluating it and comparing it against BitTorrent implementations. The power consumption study is particularly interesting. Some points are not sufficiently explained. For instance, parameters and values in the 3 equations shown in the paper, as well as their rationale, should be clarified. The assumption that all peers are selfish is too strict. It would be more realistic to have peers stay for a given time distribution (e.g., many leave immediately, some stay longer). Considering additional experimental scenarios would greatly improve the paper (distribution of arrivals/departures, larger number of leechers, peers with heterogeneous bandwidth, behavior under churn, comparison with other P2P protocols specifically designed for streaming).

Strengths: The strengths are (1) the sound design and performance of Libswift; (2) the experimental evaluation of a real implementation; and (3) the comparison with BitTorrent and study of the power consumption.

Weaknesses: The weaknesses are (1) the incomplete discussion of related work; and (2) the small number of scenarios that have been used for evaluation.

REVIEWER #5

The paper is well written and the evaluation is pleasantly through. One detail that was unclear is how the availability tree is maintained under churn. Doing so is likely to be very challenging. When all is said and done, the work is incremental in nature; it does push down known functionality into lower layers but there is obviously no reason this could not exist higher up (as it does today). What does this mechanism enable while being at the transport layer? That was entirely unclear. It would be helpful to see the bandwidth received (or continuity or stall periods) over time, it would paint a more thorough picture than just the number of stalls.

Strengths: The strengths are (1) the very good startup times; (2) the performance that is faster than state-of-the-art in some cases; and (3) the fairly thorough evaluation.

Weaknesses: The weaknesses are (1) the incremental nature of the work; (2) the fact that it is not strictly better than state-of-the-art (not sure it is lack of optimization to blame); and (3) lack of details on how the availability tree is maintained under churn

RESPONSE FROM THE AUTHORS

We thank the reviewers for their useful comments that helped improve the camera-ready version of this paper. We now state the novelty introduced by this work more clearly and extend the experimental results of the VoD comparison. Furthermore, we will address the lack of optimization and scalability of Libswift, which is mostly due to the management of the availability tree in high churn, as part of our future work.

In this paper, we aimed to present a comparison of Libswift with different P2P streaming systems, but we encountered several problems during the process. The first one is the

lack of availability of the open-source code of popular P2P streaming systems presented in the literature. The solution of reverse engineering those systems, as done in previous work, would not give us sufficient insight to draw valid conclusions. The main open-source P2P VoD system available is the Tribler client, which has the same download engine as the NextShareTV. The nature of our testing infrastructure, in which we coded the network limitations needed to replicate the realistic environment setting presented in Section 5-a, does not allow us to impose limitations on the network link itself. Therefore, for P2P systems such as PULSE, non-trivial modifications to the source code would be needed to provide a fair comparison with Libswift. Those modifications include an implementation of a player simulating the consumption of video content, and several changes to the code needed to apply the network link limitations.

The problems introduced by non-optimized code can be noticed in the experimental evaluation, where Libswift outperforms the compared clients in swarms with a small proportion of leechers to seeders. However, Libswift does not scale as well with high churn, mostly because of the non-optimized management of the availability tree.

With regard to the fairness of the comparison of the start-up times presented in Section 5-b, we clearly explain how disabling the streaming feature in the uTorrent and Libtorrent clients gives them a big advantage over Libswift. While Libswift retrieves the data for the initial player's buffer in-order, thus greatly increasing the competition between leechers of the same swarm, uTorrent and Libtorrent are free to retrieve data pieces from the entire content in any order, thus filling the player's buffer with data that does not represent the initial content needed for playback. Furthermore, our experiments show that the performance of Libtorrent dramatically decreases if the in-order data retrieval policy is enforced.