SmoothCache: HTTP-Live Streaming Goes Peer-to-Peer

Roberto Roverso, Sameh El-Ansary, Seif Haridi

SmoothCache: HTTP-Live Streaming Goes Peer-To-Peer

Roberto Roverso[1,2], Sameh El-Ansary[1], and Seif Haridi[2]

[1] Peerialism Inc., Stockholm, Sweden
[2] The Royal Institute of Technology (KTH), Stockholm, Sweden
{roberto, sameh}@peerialism.com, haridi@kth.se

Abstract. In this paper, we present SmoothCache, a peer-to-peer live video streaming (P2PLS) system. The novelty of SmoothCache is threefold: *i*) It is the first P2PLS system that is built to support the relatively-new approach of using HTTP as the transport protocol for live content, *ii*) The system supports both single and multi-bitrate streaming modes of operation, and *iii*) In Smoothcache, we make use of recent advances in application-layer dynamic congestion control to manage priorities of transfers according to their urgency. We start by explaining why the HTTP live streaming semantics render many of the existing assumptions used in P2PLS protocols obsolete. Afterwards, we present our design starting with a baseline P2P caching model. We, then, show a number of optimizations related to aspects such as neighborhood management, uploader selection and proactive caching. Finally, we present our evaluation conducted on a real yet instrumented test network. Our results show that we can achieve substantial traffic savings on the source of the stream without major degradation in user experience.

Keywords: HTTP-Live streaming, peer-to-peer, caching, CDN

1 Introduction

Peer-to-peer live streaming (P2PLS) is a problem in the Peer-To-Peer (P2P) networking field that has been tackled for quite some time on both the academic and industrial fronts. The typical goal is to utilize the upload bandwidth of hosts consuming a certain live content to offload the bandwidth of the broadcasting origin. On the industrial front, we find successful large deployments where knowledge about their technical approaches is rather limited. Exceptions include systems described by their authors like Coolstreaming [16] or inferred by reverse engineering like PPlive [4] and TVAnts [12]. On the academic front, there have been several attempts to try to estimate theoretical limits in terms of optimality of bandwidth utilization [3][7] or delay [15].

Traditionally, HTTP has been utilized for progressive download streaming, championed by popular Video-On-Demand (VoD) solutions such as Netflix [1] and Apple's iTunes movie store. However, lately, adaptive HTTP-based streaming protocols became the main technology for live streaming as well. All companies who have a major say in the market including Microsoft, Adobe and Apple have adopted HTTP-streaming as the main approach for live broadcasting. This shift to HTTP has been driven by a number of advantages such as the following: *i*) Routers and firewalls are more permissive to HTTP traffic compared to the

RTSP/RTP *ii*) HTTP caching for real-time generated media is straight-forward like any traditional web-content *iii*) The Content Distribution Networks (CDNs) business is much cheaper when dealing with HTTP downloads [5].

The first goal of this paper is to describe the shift from the RTSP/RTP model to the HTTP-live model (Section 2). This, in order to detail the impact of the same on the design of P2P live streaming protocols (Section 3). A point which we find rather neglected in the research community (Section 4). We argue that this shift has rendered many of the classical assumptions made in the current state-of-the-art literature obsolete. For all practical purposes, any new P2PLS algorithm irrespective of its theoretical soundness, won't be deployable if it does not take into account the realities of the mainstream broadcasting ecosystem around it. The issue becomes even more topical as we observe a trend in standardizing HTTP live [8] streaming and embedding it in all browsers together with HTML5, which is already the case in browsers like Apple's Safari.

The second goal of this paper is to present a P2PLS protocol that is compatible with HTTP live streaming for not only one bitrate but that is fully compatible with the concept of adaptive bitrate, where a stream is broadcast with multiple bitrates simultaneously to make it available for a range of viewers with variable download capacities (Section 5).

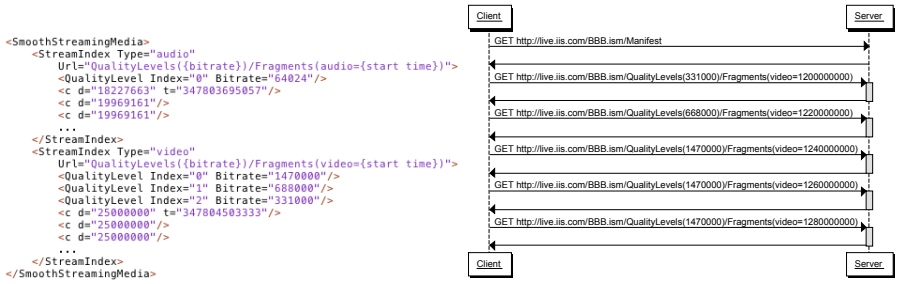
The last goal of this paper is to describe a number of optimizations of our P2PLS protocol concerning neighborhood management, uploader selection and peer transfer which can deliver a significant amount of traffic savings on the source of the stream (Section 6 and 7). Experimental results of our approach show that this result comes at almost no cost in terms of quality of user experience (Section 8).

2 The Shift from RTP/RTSP to HTTP

In the traditional RTSP/RTP model, the player uses RTSP as the signalling protocol to request the playing of the stream from a streaming server. The player enters a receive loop while the server enters a send loop where stream fragments are delivered to the receiver using the RTP protocol over UDP. The interaction between the server and player is stateful. The server makes decisions about which fragment is sent next based on acknowledgements or error information previously sent by the client. This model makes the player rather passive, having the mere role of rendering the stream fragments which the server provides.

In the HTTP live streaming model instead, it is the player which controls the content delivery by periodically pulling from the server parts of the content at the time and pace it deems suitable. The server instead is entitled with the task of encoding the stream in real time with different encoding rates, or qualities, and storing it in data fragments which appear on the server as simple resources.

When a player first contacts the streaming server, it is presented with a metadata file (*Manifest*) containing the latest stream fragments available at the server at the time of the request. Each fragment is uniquely identified by a time-stamp and a bitrate. If a stream is available in n different bitrates, then



(a) (b)

Fig. 1: a) Sample Smooth streaming Manifest, b) Client-Server interactions in Microsoft Smooth Streaming

this means that for each timestamp, there exists n versions of it, one for each bitrate.

After reading the manifest, the player starts to request fragments from the server. The burden of keeping the timeliness of the live stream is totally upon the player. The server in contrast, is stateless and merely serves fragments like any other HTTP server after encoding them in the format advertised in the manifest.

Manifest Contents. To give an example, we use Microsoft's Smooth Streaming manifest. In Figure 1a, we show the relevant details of a manifest for a live stream with 3 video bitrates (331, 688, 1470 Kbps) and 1 audio bitrate (64 Kbps). By inspecting one of the streams, we find the first (the most recent) fragment containing a d value which is the time duration of the fragment in a unit of 100 nanoseconds and a t value which is the timestamp of the fragment. The fragment underneath (the older fragment) has only a d value because the timestamp is inferred by adding the duration to the timestamp of the one above. The streams each have a template for forming a request url for fragments of that stream. The template has place holders for substitution with an actual bitrate and timestamp. For a definition of the manifest's format, see [5].

Adaptive Streaming Protocol. In Figure 1b, we show an example interaction sequence between a Smooth Streaming Client and Server [5]. The Client first issues a HTTP GET request to retrieve the manifest from the streaming server. After interpreting the manifest, the player requests a video fragment from the lowest available bitrate (331 Kbps). The timestamp of the first request is not predictable but in most cases we have observed that it is an amount equal to 10 seconds backward from the most recent fragment in the manifest. This is probably the only predictable part of the player's behavior.

In fact, without detailed knowledge of the player's internal algorithm and given that different players may implement different algorithms, it is difficult to make assumptions about the period between consecutive fragment requests, the time at which the player will switch rates, or how the audio and video are interleaved. For example, when a fragment is delayed, it could get re-requested at the same bitrate or at a lower rate. The timeout before taking such action is one thing that we found slightly more predictable and it was most of the time

around 4 seconds. That is a subset of many details about the pull behavior of the player.

Implications of Unpredictability. The point of mentioning these details is to explain that the behavior of the player, how it buffers and climbs up and down the bitrates is rather unpredictable. In fact, we have seen it change in different version of the same player. Moreover, different adopters of the approach have minor variations on the interactions sequence. For instance, Apple HTTP-live [8] dictates that the player requests a manifest every time before requesting a new fragment and packs audio and video fragments together. As a consequence of what we described above, we believe that a P2PLS protocol for HTTP live streaming should operate as if receiving random requests in terms of timing and size and has to make this the main principle. This filters out the details of the different players and technologies.

3 Impact of the Shift on P2PLS Algorithms

Traditionally, the typical setup for a P2PLS agent is to sit between the streaming server and the player as a local proxy offering the player the same protocol as the streaming server. In such a setup, the P2PLS agent would do its best, exploiting the peer-to-peer overlay, to deliver pieces in time and in the right order for the player. Thus, the P2PLS agent is the one driving the streaming process and keeping an active state about which video or audio fragment should be delivered next, whereas the player blindly renders what it is supplied with. Given the assumption of a passive player, it is easy to envisage the P2PLS algorithm skipping for instance fragments according to the playback deadline, i.e. discarding data that comes too late for rendering. In this kind of situation, the player is expected to skip the missing data by fast-forwarding or blocking for few instants and then start the playback again. This type of behavior towards the player is an intrinsic property of many of the most mature P2PLS system designs and analyses such as [13, 15, 16].

In contrast to that, a P2PLS agent for HTTP live streaming can not rely on the same operational principles. There is no freedom in skipping pieces and deciding what is to be delivered to the player. The P2PLS agent has to obey the player's request for fragments from the P2P network and the speed at which this is accomplished affects the player's next action. From our experience, delving in the path of trying to reverse engineer the player behavior and integrating that in the P2P protocol is some kind of black art based on trial-and-error and will result into very complicated and extremely version-specific customizations. Essentially, any P2PLS scheduling algorithm that assumes that it has control over which data should be delivered to the player is rather inapplicable to HTTP live streaming.

4 Related Work

We are not aware of any work that has explicitly articulated the impact of the shift to HTTP on the P2P live streaming algorithms. However, a more relevant

topic to look at is the behavior of the HTTP-based live players. Akhshabi et. al [2], provide a recent dissection of the behavior of three such players under different bandwidth variation scenarios. It is however clear from their analysis that the bitrate switching mechanics of the considered players are still in early stages of development. In particular, it is shown that throughput fluctuations still cause either significant buffering or unnecessary bitrate reductions. On top of that, it is shown how all the logic implemented in the HTTP-live players is tailored to TCP's behavior, as the one suggested in [6]. That in order to compensate throughput variations caused by TCP's congestion control and potentially large retransmission delays. In the case of a P2PLS agent acting as proxy, it is then of paramount importance to not interfere with such adaptation patterns.

We believe, given the presented approaches, the most related work is the P2P caching network LiveSky [14]. We share in common the fact of trying to establish a P2P CDN. However, LiveSky does not present any solution for supporting HTTP live streaming.

5 P2PLS as a Caching Problem

We will describe here our baseline design to tackle the new realities of the HTTP-based players. We treat the problem of reducing the load on the source of the stream the same way it would be treated by a Content Distribution Network (CDN): as a *caching problem*. The design of the streaming protocol was made such that every fragment is fetched as an independent HTTP request that could be easily scheduled on CDN nodes. The difference is that in our case, the caching nodes are consumer machines and not dedicated nodes. The player is directed to order from our local P2PLS agent which acts as an HTTP proxy. All traffic to/from the source of the stream as well as other peers passes by the agent.

Baseline Caching. The policy is as follows: any request for manifest files (metadata), is fetched from the source as is and not cached. That is due to the fact that the manifest changes over time to contain the newly generated fragments. Content fragments requested by the player are looked up in a local index of the peer which keeps track of which fragment is available on which peer. If information about the fragment is not in the index, then we are in the case of a P2P cache *miss* and we have to retrieve it from the source. In case of a cache *hit*, the fragment is requested from the P2P network and any error or slowness in the process results, again, in a fallback to the source of the content. Once a fragment is downloaded, a number of other peers are immediately informed in order for them to update their indices accordingly.

Achieving Savings. The main point is thus to increase the cache hit ratio as much as possible while the timeliness of the movie is preserved. The cache hit ratio is our main metric because it represents savings from the load on the source of the live stream. Having explained the baseline idea, we can see that, in theory, if all peers started to download the same uncached manifest simultaneously, they would also all start requesting fragments exactly at the same time in perfect alignment. This scenario would leave no time for the peers to advertise and exchange useful fragments between each others. Consequently a perfect

Strategy	Baseline	Improved
Manifest Trimming (MT)	Off	On
Partnership Construction (PC)	Random	Request-Point-aware
Partnership Maintenance (PM)	Random	Bitrate-aware
Uploader Selection (US)	Random	Throughput-based
Proactive Caching (PR)	Off	On

Table 1: Summary of baseline and improved strategies.

alignment would result in no savings. In reality, we have always seen that there is an amount of intrinsic asynchrony in the streaming process that causes some peers to be ahead of others, hence making savings possible. However, the larger the number of peers, the higher the probability of more peers being aligned. We will show that, given the aforementioned asynchrony, even the previously described baseline design can achieve significant savings.

Our target savings are relative to the number of peers. That is we do not target achieving a constant load on the source of the stream irrespective of the number of users, which would lead to loss of timeliness. Instead, we aim to save a substantial percentage of all source traffic by offloading that percentage to the P2P network. The attractiveness of that model from a business perspective has been verified with content owners who nowadays buy CDN services.

6 Beyond Baseline Caching

We give here a description of some of the important techniques that are crucial to the operation of the P2PLS agent. For each such technique we provide what we think is the simplest way to realize it as well as improvements if we were able to identify any. The techniques are summarized in Table 1.

Manifest Manipulation. One improvement particularly applicable in Microsoft's Smooth streaming but that could be extended to all other technologies is manifest manipulation. As explained in Section 2, the server sends a manifest containing a list of the most recent fragments available at the streaming server. The point of that is to avail to the player some data in case the user decides to jump back in time. Minor trimming to hide the most recent fragments from some peers places them behind others. We use that technique to push peers with high upload bandwidth slightly ahead of others because they have they can be more useful to the network. We are careful not to abuse this too much, otherwise peers would suffer a high delay from live playing point, so we limit it to a maximum of 4 seconds. It is worth noting here that we do a quick bandwidth measurement for peers upon admission to the network, mainly, for statistical purposes but we do not depend on this measurement except during the optional trimming process.

Neighborhood & Partnership Construction. We use a tracker as well as gossiping for introducing peers to each other. Any two peers who can establish bi-directional communication are considered neighbors. Each peer probes his neighbors periodically to remove dead peers and update information about their last requested fragments. Neighborhood construction is in essence a process to create a random undirected graph with high node arity. A subset of the edges in the neighborhood graph is selected to form a directed subgraph to establish

partnership between peers. Unlike the neighborhood graph, which is updated lazily, the edges of the partneship graph are used frequently. After each successful download of a fragment, the set of (*out-partners*) is informed about the newly downloaded fragment. From the opposite perspective, it is crucial for a peer to wisely pick his *in-partners* because they are the providers of fragments from the P2P network. For this decision, we experiment with two different strategies: *i*) Random picking, *ii*) Request-point-aware picking: where the in-partners include only peers who are relatively ahead in the stream because only such peers can have future fragments.

Partnership Maintenance. Each peer strives to continuously find *better* in-partners using periodic maintenance. The maintenance process could be limited to replacement of dead peers by randomly-picked peers from the neighborhood. Our improved maintenance strategy is to score the in-partners according to a certain metric and replace low-scoring partners with new peers from the neighborhood. The metric we use for scoring peers is a composite one based on: *i*) favoring the peers with higher percentage of successfully transferred data, *ii*) favoring peers who happen to be on the same bitrate. Note that while favoring peers on the same bitrate, having all partners from a single bitrate is very dangerous, because once a bit-rate change occurs the peer is isolated. That is, all the received updates about presence of fragments from other peers would be from the old bitrate. That is why, upon replacement, we make sure that the resulting in-partners set has all bit-rates with a gaussian distribution centered around the current bitrate. That is, most in-partners are from the current bit rate, less partners from the immediately higher and lower bit rates and much less partners from other bitrates and so forth. Once the bit-rate changes, the maintenance re-centers the distribution around the new bitrate.

Uploader Selection. In the case of a cache hit, it happens quite often that a peer finds multiple uploaders who can supply the desired fragment. In that case, we need to pick one. The simplest strategy would be to pick a random uploader. Our improved strategy here is to keep track of the observed historical throughput of the downloads and pick the fastest uploader.

Sub-fragments. Up to this point, we have always used in our explanation the fragment as advertised by the streaming server as the unit of transport for simplifying the presentation. In practice, this is not the case. The sizes of the fragment vary from one bitrate to the other. Larger fragments would result in waiting for a longer time before informing other peers which would directly entail lower savings because of the slowness of disseminating information about fragment presence in the P2P network. To handle that, our unit of transport and advertising is a sub-fragment of a fixed size. That said, the reality of the uploader selection process is that it always picks a set uploaders for each fragment rather than a single uploader. This parallelization applies for both random and throughput-based uploader selection strategies.

Fallbacks. While downloading a fragment from another peer, it is of critical importance to detect problems as soon as possible. The timeout before falling back to the source is thus one of the major parameters while tuning the system.

We put an upper bound (T_{p2p}) on the time needed for any P2P operation, computed as: $T_{p2p} = T_{player} - S * T_f$ where T_{player} is the maximum amount of time after which the player considers a request for a fragment expired, S is the size of fragment and T_f is the expected time to retrieve a unit of data from the fallback. Based on our experience, T_{player} is player-specific and constant, for instance Microsoft's Smooth Streaming waits 4 seconds before timing out. A longer T_{p2p} translates in a higher P2P success transfer ratio, hence higher savings. Since T_{player} and S are outside of our control, it is extremely important to estimate T_f correctly, in particular in presence of congestion and fluctuating throughput towards the source. As a further optimization, we recalculate the timeout for a fragment while a P2P transfer is happening depending on the amount of data already downloaded, to allow more time for the outstanding part of the transfer. Finally, upon fallback, only the amount of fragment that failed to be downloaded from the overlay network is retrieved from the source, i.e. through a partial HTTP request on the range of missing data.

7 Proactive Caching

The baseline caching process is in essence reactive, i.e. the attempt to fetch a fragment starts *after* the player requests it. However, when a peer is informed about the presence of a fragment in the P2P network, he can trivially see that this is a future fragment that would be eventually requested. Starting to prefetch it early before it is requested, increases the utilization of the P2P network and decreases the risk of failing to fetch it in time when requested. That said, we do not guarantee that this fragment would be requested in the same bitrate, when the time comes. Therefore, we endure a bit of risk that we might have to discard it if the bitrate changes. In practice, we measured that the prefetcher successfully requests the right fragment with a 98.5% of probability.

Traffic Prioritization. To implement this proactive strategy we have taken advantage of our dynamic runtime-prioritization transport library DTL [9] which exposes to the application layer the ability to prioritize individual transfers relative to each other and to change the priority of each individual transfer at run-time. Upon starting to fetch a fragment proactively, it is assigned a very low-priority. The rationale is to avoid contending with the transfer process of fragments that are reactively requested and under a deadline both on the uploading and downloading ends.

Successful Prefetching. One possibility is that a low-priority prefetching process completes before a player's request and there is no way to deliver it to the player before that happens, the only option is to wait for a player request. More importantly, when that time comes, careful delivery from the local machine is very important because extremely fast delivery might make the adaptive streaming player mistakenly think that there is an abundance of download bandwidth and start to request the following fragments a higher bitrate beyond the actual download bandwidth of the peer. Therefore, we schedule the delivery from the local machine to be not faster than the already-observed average download rate. We have to stress here that this is not an attempt to control the player

to do something in particular, we just maintain transparency by not delivering prefetched fragments faster than not prefetched ones.

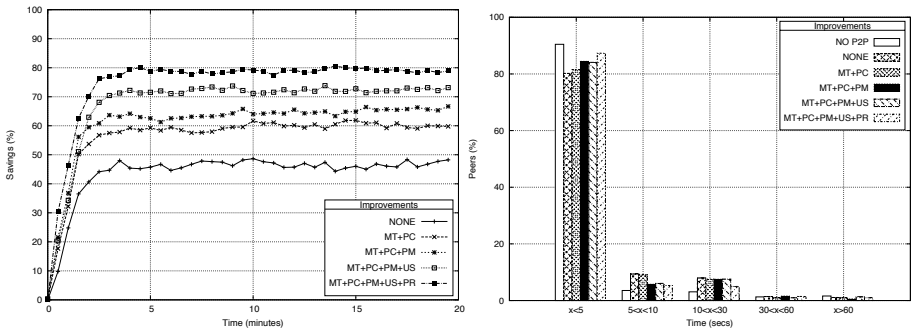
Interrupted Prefetching. Another possiblity is that the prefetching process gets interrupted by the player in 3 possible ways: *i*) The player requests the fragment being prefetched: in that case the transport layer is dynamically instructed to raise the priority and T_{player} is set accordingly based on the remaining amount of data as described in the previous section. *ii*) The player requests the same fragment being prefetched but at a higher rate which means we have to discard any prefetched data and treat the request like any other reactively fetched fragment. *iii*) The player decides to skip some fragments to catch up and is no longer in need of the fragment being prefetched. In this case, we have to discard it as well.

8 Evaluation

Methodology. Due to the non-representative behaviour of Planetlab and the difficulty to do parameter exploration in publicly-deployed production network, we tried another approach which is to develop a version of our P2P agent that is remotely-controlled and ask for volunteers who are aware that we will conduct experiments on their machines. Needless to say, that this functionality is removed from any publicly-deployable version of the agent.

Test Network. The test network contained around 1350 peers. However, the maximum, minimum and average number of peers simultaneously online were 770, 620 and 680 respectively. The network included peers mostly from Sweden (89%) but also some from Europe (6%) and the US (4%). The upload bandwidth distribution of the network was as follows: 15% : $0.5Mbps$, 42% : $1Mbps$, 17% : $2.5Mbps$, 15% : $10Mbps$, 11% : $20Mbps$. In general, one can see that there is enough bandwidth capacity in the network, however the majority of the peers are on the lower end of the bandwidth distribution. For connectivity, 82% of the peers were behind NAT, and 12% were on open Internet. We have used our NAT-Cracker traversal scheme as described in [11] and were able to establish bi-directional communication between 89% of all peer pairs. The unique number of NAT types encountered were 18 types. Apart from the tracker used for introducing clients to each other, our network infrastructure contained, a logging server, a bandwidth measurement server, a STUN-like server for helping peers with NAT traversal and a controller to launch tests remotely.

Stream Properties. We used a production-quality continuous live stream with 3 video bitrates (331, 688, 1470 Kbps) and 1 audio bitrate (64 Kbps) and we let peers watch 20 minutes of this stream in each test. The stream was published using Microsoft Smooth Streaming traditional tool chain. The bandwidth of the source stream was provided by a commercial CDN and we made sure that it had enough capacity to serve the maximum number of peers in our test network. This setup gave us the ability to compare the quality of the streaming process in the presence and absence of P2P caching in order to have a fair assessment of the effect of our agent on the overall quality of user experience. We stress that, in a real deployment, P2P caching is not intended to eliminate the need for a



(a) (b)

Fig. 2: (a) Comparison of traffic savings with different algorithm improvements, (b) Comparison of cumulative buffering time for source only and improvements.

CDN but to reduce the total amount of paid-for traffic that is provided by the CDN. One of the issues that we faced regarding realistic testing was making sure that we are using the actual player that would be used in production, in our case that was the Microsoft Silverlight player. The problem is that the normal mode of operation of all video players is through a graphical user interface. Naturally, we did not want to tell our volunteers to click the "Play" button every time we wanted to start a test. Luckily, we were able to find a rather unconventional way to run the Silverlight player in a headless mode as a background process that does not render any video and does not need any user intervention.

Reproducibility. Each test to collect one data point in the test network happens in real time and exploring all parameter combination of interest is not feasible. Therefore, we did a major parameter combinations study on our simulation platform [10] first to get a set of worth-trying experiments that we launched on the test network. Another problem is the fluctuation of network conditions and number of peers. We repeated each data point a number of times before gaining confidence that this is the average performance of a certain parameter combination.

Evaluation Metrics. The main metric that we use is *traffic savings* defined as the percentage of the amount of data served from the P2P network from the total amount of data consumed by the peers. Every peer reports the amount of data served from the P2P network and streaming source every 30 seconds to the logging server. In our bookkeeping, we keep track of how much of the traffic was due to fragments of a certain bitrate. The second important metric is *buffering delay*. The Silverlight player can be instrumented to send debug information every time it goes in/out of buffering mode, i.e. whenever the player finds that the amount of internally-buffered data is not enough for playback, it sends a debug message to the server, which in our case is intercepted by the agent. Using this method, a peer can report the lengths of the periods it entered into buffering mode in the 30 seconds snapshots as well. At the end of the stream, we calculate the sum of all the periods the player of a certain peer spent buffering.

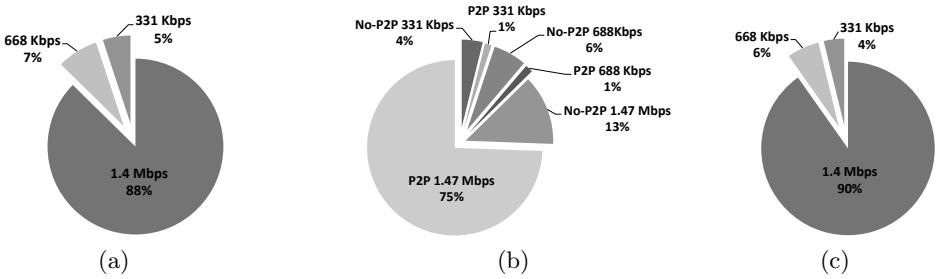


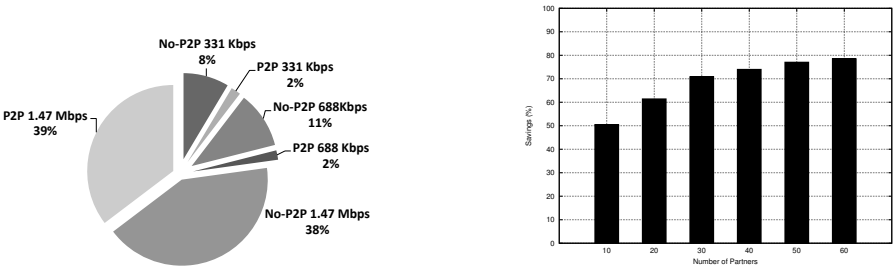
Fig. 3: Breakdown of traffic quantities per bitrate for: (a) A network with P2P caching, Source & P2P traffic summed together. (b) The same P2P network with source & P2P traffic reported separately, and (c) A network with no P2P.

8.1 Deployment Results

Step-by-Step Towards Savings. The first investigation we made was to start from the baseline design with all the strategies set to the simplest possible. In fact, during the development cycle we used this baseline version repeatedly until we obtained a stable product with predictable and consistent savings level before we started to enable all the other improvements. Figure 2a shows the evolution of savings in time for all strategies. The naive baseline caching was able to save a total of 44% of the source traffic. After that, we worked on pushing the higher-bandwidth peers ahead and making each partner select peers that are useful using the request-point-aware partnership which moved the savings to a level of 56%. So far, the partnership maintenance was random. Turning on bit-rate-aware maintenance added only another 5% of savings but we believe that this is a key strategy that deserves more focus because it directly affects the effective partnership size of other peers from each bitrate which directly affects savings. For the uploader selection, running the throughput-based picking achieved 68% of savings. Finally, we got our best savings by adding proactive caching which gave us 77% savings.

User Experience. Getting savings alone is not a good result unless we have provided a good user experience. To evaluate the user experience, we use two metrics: First, the percentage of peers who experienced a total buffering time of less than 5 seconds, i.e. they enjoyed performance that did not really deviate much from live. Second, showing that our P2P agent did not achieve this level of savings by forcing the adaptive streaming to move everyone to the lowest bitrate. For the first metric, Figure 2b shows that with all the improvements, we can make 87% of the network watch the stream with less than 5 seconds of buffering delay. For the second metric, Figure 3a shows also that 88% of all consumed traffic was on the highest bitrate and P2P alone shouldering 75% (Figure 3b), an indication that, for the most part, peers have seen the video at the highest bitrate with a major contribution from the P2P network.

P2P-less as a Reference. We take one more step beyond showing that the system offers substantial savings with reasonable user experience, namely



(a) (b)

Fig. 4: (a) Breakdown of traffic quantities per bitrate using baseline, (b) Comparison of savings between different in-partners number.

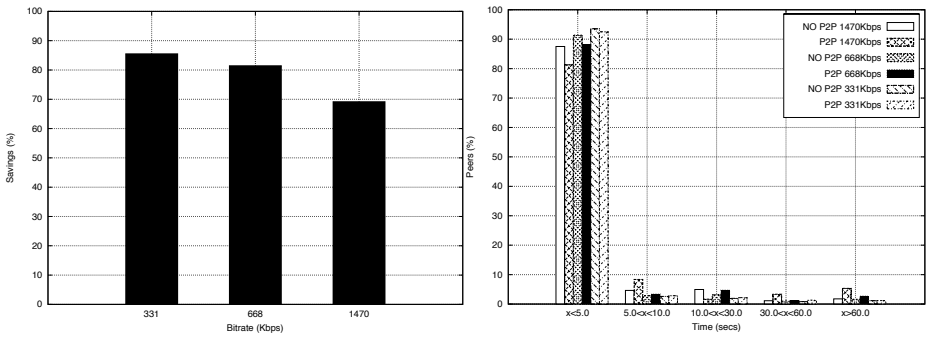
to understand what would be the user experience in case all the peers streamed directly from the CDN. Therefore, we run the system with P2P caching disabled. Figure 2b shows that without P2P, only 3% more (90%) of all viewers would have a less than 5 seconds buffering. On top of that, Figure 3c shows that only 2% more (90%) of all consumed traffic is on the highest bitrate, that is the small price we paid for saving 77% of source traffic. Figure 4a instead describes the lower performance of our baseline caching scenario, which falls 13% of the P2P-less scenario (77%). This is mainly due to the lack of bit-rate-aware maintenance, which turns out to play a very significant role in terms of user experience.

Partnership Size. There are many parameters to tweak in the protocol but, in our experience, the number of in-partners is by far the parameter with the most significant effect. Throughout the evaluation presented here, we use 50 in-partners. Figure 4b shows that more peers result in more savings; albeit with diminishing returns. We have selected 50-peers as a high-enough number, at a point where increasing the peers does not result into much more savings.

Single Bitrate. Another evaluation worth presenting as well is the case of a single bitrate. In this experiment, we get 84%, 81% and 69% for the low, medium and high bitrate respectively (Figure 5a). As for the user experience compared with the same single bitrates in a P2P-less test, we find that the user experience expressed as delays is much closer to the P2P-less network (Figure 5b). We explain the relatively better experience in the single bitrate case by the fact that all the in-partners are from the same bitrate, while in the multi-bitrate case, each peer has in his partnership the majority of the in-partners from a single bitrate but some of them are from other bitrates which renders the effective partnership size smaller. We can also observe that the user experience improves as the bitrate becomes smaller.

9 Conclusion

In this paper, we have shown a novel approach in building a peer-to-peer live streaming system that is compatible with the new realities of the HTTP-live. These new realities revolve around the point that unlike RTSP/RTP streaming, the video player is driving the streaming process. The P2P agent will have a



(a) (b)

Fig. 5: (a) Savings for single bitrate runs, (b) Buffering time for single bitrate runs.

limited ability to control what gets rendered on the player and much limited ability to predict its behaviour. Our approach was to start with baseline P2P caching where a P2P agent acts as an HTTP proxy that receives requests from the HTTP live player and attempts to fetch it from the P2P network rather the source if it can do so in a reasonable time.

Beyond baseline caching, we presented several improvements that included: a) Request-point-aware partnership construction where peers focus on establishing relationships with peers who are ahead of them in the stream, b) Bit-rate-aware partnership maintenance through which a continuous updating of the partnership set is accomplished both favoring peers with high successful transfers rate and peers who are on the same bitrate of the maintaining peer, c) Manifest trimming which is a technique for manipulating the metadata presented to the peer at the beginning of the streaming process to push high-bandwidth peers ahead of others, d) Throughput-based uploader selection which is a policy used to pick the best uploader for a certain fragment if many exist, e) Careful timing for falling back to the source where the previous experience is used to tune timing out on P2P transfers early enough thus keeping the timeliness of the live playback.

Our most advanced optimization was the introduction of proactive caching where a peer requests fragments ahead of time. To accomplish this feature without disrupting the already-ongoing transfer, we used our application-layer congestion control [9] to make pre-fetching activities have less priority and dynamically raise this priority in case the piece being pre-fetched got requested by the player.

We evaluated our system using a test network of real volunteering clients of about 700 concurrent nodes where we instrumented the P2P agents to run tests under different configurations. The tests have shown that we could achieve around 77% savings for a multi-bitrate stream with around 87% of the peers experiencing a total buffering delay of less than 5 seconds and almost all of the peers watched the data on the highest bitrate. We compared these results with

the same network operating in P2P-less mode and found that only 3% of the viewers had a better experience without P2P which we judge as a very limited degradation in quality compared to the substantial amount of savings.

References

1. Netflix inc. www.netflix.com
2. Akhshabi, S., Begen, A.C., Dovrolis, C.: An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In: Proceedings of the second annual ACM conference on Multimedia systems. MMSys (2011)
3. Guo, Y., Liang, C., Liu, Y.: AQCS: adaptive queue-based chunk scheduling for P2P live streaming. In: Proceedings of the 7th IFIP-TC6 NETWORKING (2008)
4. Hei, X., Liang, C., Liang, J., Liu, Y., Ross, K.W.: Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System. In: Proc. of IPTV Workshop, International World Wide Web Conference (2006)
5. Microsoft Inc.: Smooth Streaming. http://www.iis.net/download/SmoothStreaming
6. Liu, C., Bouazizi, I., Gabbouj, M.: Parallel Adaptive HTTP Media Streaming. In: Computer Communications and Networks (ICCCN), Proc. of 20th International Conference on. pp. 1 –6 (31 2011-aug 4 2011)
7. Massoulie, L., Twigg, A., Gkantsidis, C., Rodriguez, P.: Randomized Decentralized Broadcasting Algorithms. In: INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE. pp. 1073 –1081 (may 2007)
8. Pantos, R.: HTTP Live Streaming (Dec 2009), http://tools.ietf.org/html/draft-pantos-http-live-streaming-01
9. Reale, R., Roverso, R., El-Ansary, S., Haridi, S.: DTL: Dynamic Transport Library for Peer-To-Peer Applications. In: In Proc. of the 12th International Conference on Distributed Computing and Networking. ICDCN (Jan 2012)
10. Roverso, R., El-Ansary, S., Gkogkas, A., Haridi, S.: Mesmerizer: A effective tool for a complete peer-to-peer software development life-cycle. In: In Proceedings of SIMUTOOLS (March 2011)
11. Roverso, R., El-Ansary, S., Haridi, S.: NATCracker: NAT Combinations Matter. In: Proc. of 18th International Conference on Computer Communications and Networks. ICCCN '09, IEEE Computer Society, SF, CA, USA (2009)
12. Silverston, T., Fourmaux, O.: P2P IPTV measurement: a case study of TVants. In: Proceedings of the 2006 ACM CoNEXT conference. pp. 45:1–45:2. CoNEXT '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1368436.1368490
13. Vlavianos, A., Iliofotou, M., Faloutsos, M.: BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In: INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. pp. 1 –6 (april 2006)
14. Yin, H., Liu, X., Zhan, T., Sekar, V., Qiu, F., Lin, C., Zhang, H., Li, B.: Livesky: Enhancing cdn with p2p. ACM Trans. Multimedia Comput. Commun. Appl. 6, 16:1–16:19 (August 2010), http://doi.acm.org/10.1145/1823746.1823750
15. Zhang, M., Zhang, Q., Sun, L., Yang, S.: Understanding the Power of Pull-Based Streaming Protocol: Can We Do Better? In: Selected Areas in Communications, IEEE Journal on. vol. 25, pp. 1678 –1694 (december 2007)
16. Zhang, X., Liu, J., Li, B., Yum, Y.S.P.: CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies