

file, it can continue to *seed* it by using its upload capacity to serve the file to others for free.

Mature BitTorrent clients extend the basic BitTorrent protocol, for instance by including support for a global DHT ring, which allows peers to locate each other without requiring to contact the tracker. This DHT ring in particular could be used as a basis for an application-level multicast tree. However, such a multicast tree would not benefit from any tit-for-tat mechanism, and would thus be vulnerable for abuse.

B. Related Work

Many of the P2P live streaming algorithms proposed in the literature are designed to operate in a cooperative environment. Abusing such algorithms on the Internet is easy [4]. The BAR Gossip [5] and Orchard [6] algorithms do not assume full cooperation from the peers. BAR Gossip is a gossip algorithm in which the broadcaster can evict malicious peers after malicious behaviour has been reported by other peers. The Orchard algorithm divides a video stream into several substreams, which are exchanged by the peers in a fair way. Unlike our approach, neither of these algorithms considers the underlying P2P framework or is measured in a deployed system.

Deployed live streaming P2P solutions have been measured before [7]–[9] and reveal a decent performance. Ali et al. [7] and Agarwal et al. [8] measure the performance of commercial P2P live streaming systems, but they do not describe the P2P distribution algorithm that was used. Xie et al. [9] measure a deployed and open protocol called Coolstreaming [10]. Our approach is different in that we extend BitTorrent, which makes our algorithm easy to develop, and easy to deploy, also in non-cooperative environments. We will compare our results with those found by Agarwal et al. [8] and Xie et al. [9] to provide insight into our results.

III. EXTENSIONS FOR LIVE STREAMING

Our live streaming extensions to BitTorrent use the following generic setup. The injector obtains the video from a live source, such as a DV camera, and generates a *tstream* file, which is similar to a torrent file but cannot be used by BitTorrent clients lacking our live streaming extensions. An end user (peer) which is interested in watching the video stream obtains the *tstream* file and joins the swarm (the set of peers) as per the BitTorrent protocol. We regard the video stream to be of infinite length, which is useful when streaming a TV channel or a webcam feed.

We identify four problems for BitTorrent when streaming live video, which are unknown video length, unknown future video content, a suitable piece-selection policy, and the lack of seeders, which we will discuss in turn.

A. Unlimited Video Length

The BitTorrent protocol assumes that the number of pieces of a video is known in advance. This number is used throughout a typical implementation to allocate arrays with an element for every piece, and therefore, it is not practical to simply

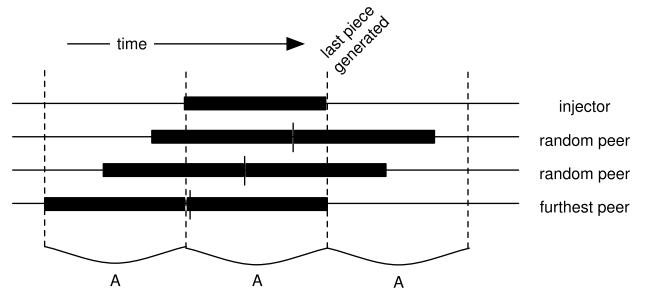


Figure 1. The sliding windows of valid pieces for the injector, two random peers, and the peers furthest from the injector in terms of delay. The vertical markers on each bar indicate the peer's playback position.

increase the number of pieces to a very large value that will not be reached (for instance, 2^{32}). Instead, each peer maintains its own sliding window that rotates over a fixed number of pieces at the speed of the video stream. Each peer deletes pieces that fall outside of its sliding window, and will consider them to be deleted at its neighbours as well, thereby avoiding the need for additional messages. Within its sliding window, each peer barter for pieces according to the BitTorrent protocol.

The injector defines its sliding window as the A most recently generated pieces. The value of A is included in the *tstream* file and is thus known to all peers. Any other peer is not necessarily connected to the injector, and synchronises its sliding window with the pieces available at its neighbours. For any peer, let piece p be the youngest piece available at more than half of its neighbours. Then, its sliding window is $(p - A, p + A]$. Figure 1 illustrates the relationship between the sliding windows of the peers. We allow the playback position, which we will define in Section III-C, of any peer is to be no more than A pieces behind the latest generated piece. A peer can verify this by comparing its clock to the timestamps which we will include in the pieces. The playback position of any peer thus falls within the sliding window of the injector, which allows the injector to serve any peer if all others depart. Furthermore, the sliding window of one peer cannot overlap with overlap with the sliding window of another peer due to a wrap around over the total set of pieces.

A peer thus has to be able to determine which piece is the youngest available. In order to make that possible, we let the total number of pieces be $4A$. Each neighbour has a sliding window of size at most $2A$, and will subsequently report at least $2A$ consecutive missing pieces. The last piece reported before a missing range of at least $2A$ pieces is thus the youngest piece owned by that neighbour. Neighbours that report to have no pieces at all, are ignored. We use majority voting to be able to ignore malfunctioning and malicious clients, which may report pieces that are outdated, or that they do not own.

B. Data Validation

Once a peer completes the download of any piece, it proceeds to validate the data. The ordinary BitTorrent download protocol computes a hash for each piece and includes these hashes in the torrent file. In the case of live streaming, these

hashes have to be filled with dummy values or cannot be included at all, because the data are not known in advance and the hashes can therefore not be computed when the torrent file is created. The system would thus be left prone to injection attacks and data corruption.

We prevent data corruption by using asymmetric cryptography to sign the data, which has been found by Dhungel et al. [11] to be superior to several other methods for data validation in live P2P video streams. The injector publishes its public key by putting it in the torrent file. Each piece includes an absolute sequence number, starting from zero, and a time stamp at which the piece was generated. The SHA1 hash of each piece is signed by the injector, and the signature is appended to the piece. Any peer can thus check whether a piece was generated by the injector. The sequence number allows a peer to confirm that the piece is recent, since the actual piece numbers are reused by the rotating sliding window. The time stamp allows a peer to determine the propagation delay from the injector. We use 64-bit integers to represent both the sequence numbers and the timestamps, so a wrap around will not occur within a single session.

In our case, the signature adds 64 bytes of overhead to each piece, and the sequence number and timestamp add 16 bytes in total. If a peer downloads a piece and detects an invalid signature or finds it outdated, it will delete the piece and disconnect from the neighbour that provided it. A piece is outdated if it was generated A pieces or longer ago.

C. Live Playback

Before a peer starts downloading any video data, it has to decide at which piece number it will start watching the stream, which we call the *hook-in point*. We assume each peer desires to play pieces as close as possible to the latest piece it is able to obtain. However, if the latest pieces are available at only a small number of neighbours, downloading the stream at the video bitrate may not be sustainable. We therefore let a peer start downloading at B pieces before the latest piece that is available at at least half of its neighbours. The *prebuffering phase* starts when the peer joins the network and ends when it has downloaded 90% of these B pieces. We do not require a peer to obtain 100% of these pieces, to avoid waiting for pieces that are not available at its neighbours or that take too long to download. The *prebuffering time* is defined as the length of the prebuffering phase.

The pieces requested from a neighbour are picked on a rarest-first basis, as in BitTorrent. This policy encourages peers to download different pieces and subsequently exchange them. If peers would request their pieces in-order, as is common in video streaming algorithms, peers would rarely be in the position to exchange data, which would conflict with the tit-for-tat incentives in BitTorrent.

A buffer underrun occurs when a piece i is to be played but has not been downloaded. Since pieces are downloaded out of order, the peer can nevertheless have pieces after i available for playback. If a peer has more than $B/2$ pieces available after i , the peer will drop missing piece i . Otherwise, it will

stall playback in order to allow data to be accumulated in the buffer. Once more than $B/2$ pieces are available, playback is resumed, which could result in dropping piece i after all if it still has not been downloaded. We will use a value of B equivalent to 10 seconds of video in our trial, a value which we derive through simulation in Section IV-C.

We employ this trade off since neither dropping nor stalling is a strategy that can be used in all situations. For instance, if a certain piece is lost because it was never available for download, it should be dropped, and playback can just ignore the missing piece. On the other hand, a peer's playback position can suddenly become unsustainable due to network dynamics. A different set of neighbours may only be able to provide the peer with older data than it needs. In that case, a peer should stall playback in order to synchronise its playback position with its new neighbours.

D. Seeders

In BitTorrent swarms, the presence of seeders significantly improves the download performance of the other peers (the leechers). However, such peers do not exist in a live streaming environment as no peer ever finishes downloading. We redefine a *seeder* in a live streaming environment to be a peer which is always *unchoked* by the injector and is guaranteed enough bandwidth in order to obtain the full video stream. The injector has a predefined list of peers it trusts to act as seeders if they connect to the injector. The identity of the seeders is not known to the other peers to prevent malicious behaviour.

The injector and seeders behave like a small Content Delivery Network (CDN). Even though the seeders diminish the distributed nature of the BitTorrent algorithm, they can be required if the peers cannot provide each other with enough bandwidth to receive the video stream in real-time. The seeders boost the download performance of other peers as in regular BitTorrent swarms. Also, the seeders increase the availability of the individual pieces, which reduces the probability of an individual piece not being pushed into the rest of the swarm. We will measure the effect of increasing the number of seeders through simulation in Section IV-E.

IV. SIMULATIONS

We have written a BitTorrent simulator and extended it with our algorithm to evaluate the performance of our extensions. In this section, we will explain the setup of our simulations, followed by the evaluation of the impact of four parameters: the capacity of the uplink of the peers, the hook-in point, the piece size, and the number of seeders.

A. Simulation Setup

We do not aim to reproduce the exact same conditions in the simulations that will occur in our trial. Instead, we use the simulator to implement a reasonable scenario in order to test the impact of several parameters defined throughout this paper. In each simulation run, we simulate 10 minutes of time, and let peers arrive with an average rate of one per second (using a Poisson distribution) to share a video

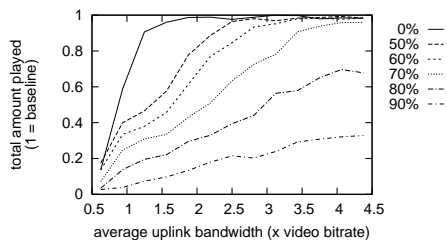


Figure 2. The total amount of data played versus the average uplink bandwidth, for several percentages of firewalled peers.

stream of 512 Kbit/s using BitTorrent pieces of 32 Kbyte, which amounts to two pieces per second. Each peer has a residence time of 0 – 10 minutes. For each set of parameters (data point) we evaluate, we performed ten simulation runs. The peers have an asymmetric connection with an uplink of 1–1.5 Mbit/s and a downlink of four times the uplink. The latency between each pair of peers is 100 – 300 ms. We have shown in previous work [3] that the percentage of peers that cannot accept incoming connections (due to the presence of a firewall or NAT) can have a significant impact on the performance, especially if this percentage is higher than 50%. The BitTorrent protocol does not define any firewall puncturing or NAT traversal techniques, and due to their complexity we consider both techniques to be outside the scope of our extension. Therefore, we repeat our simulations, with every peer having a probability of 0%, 50%, 60%, 70%, 80%, or 90% of not accepting incoming connections. All peers can, however, initiate outgoing connections and transfer data in both directions if the connection is accepted. To keep our model simple and for ease of discussion, we will consider any peer which cannot accept incoming connections to be firewalled, and vice versa. Peers not behind a firewall will be called *connectable*.

B. Uplink Bandwidth

The amount of available upload capacity of the peers is one of the critical factors for the performance of a P2P live video streaming algorithm. Regardless of the streaming algorithm, the peers (including the injector and the seeders) need at least an average upload bandwidth equal to the video bitrate in order to serve the video stream to each other without loss. In practice, more upload bandwidth is needed as not all of the available upload bandwidth in the system can be put to use. Even though one peer may have the data and the capacity to serve another peer that needs those data, the peers may not know about each other, or may be barred from connecting to each other due to the presence of firewalls.

Using our simulator, we tested the impact of the average available uplink bandwidth. For each simulation, we define an average uplink bandwidth u for the peers. Each peer is given an uplink between $0.75u$ and $1.25u$. As our performance metric, we use the total amount of data played by all peers, instead of the more common metrics such as piece loss and prebuffering time. We do this for two reasons. First, we need a single metric to compare the performance between simulations. The piece loss in a system can be lowered by increasing

the prebuffering time, and vice-versa, so performance cannot be accurately described by either metric alone. The total amount of played data is a single metric which avoids such a trade-off, since both an increased piece loss and an increased prebuffering time lower the amount of data played. Secondly, describing the piece loss figures for a set of peers in a single number is problematic, as the fraction of loss is easily skewed by peers which remain in the system for only a brief period of time. When considering the total amount of data played by all peers, peers which remain in the system only briefly have a lower impact.

Our metric has two downsides as well. First of all, for piece loss and prebuffering time figures, their optimal values are clear. However, for the total amount of data played, the optimal value depends on the arrival and departure rates of the peers. We therefore use the same arrival and departure patterns when testing different parameters. The best average performance over the ten patterns we see across our tests is used as our base line, and assign it a value of 1. In that case, we found 4.3 seconds of prebuffering time and $\ll 0.01\%$ pieces loss. The results of all other tests are given relative to this base line. Secondly, since our metric captures both piece loss and prebuffering time, the difference between them cannot be distinguished. If the total amount of data played is low, one cannot tell whether there was a lot of piece loss, or whether peers took a long time before they started playing. Also, for long simulations, an acceptable piece loss can accumulate to high values which would not be acceptable if the same amount of loss was caused by an exceedingly long prebuffering time.

Figure 2 plots the total amount of data played as it depends on the available uplink bandwidth normalised with regard to the video bitrate (512 Kbit/s). As expected, the performance is poor when the peers have an average uplink smaller than the video bitrate. If there are no firewalled peers, the peers need roughly 1.5 times the video bitrate as their uplink in order to obtain the highest performance. However, once more than half of the peers are firewalled, the performance drops significantly and more bandwidth is needed. This result is consistent with our earlier findings on P2P data exchange in general [3].

In the remaining simulations, we give peers an uplink of 1 – 1.5 Mbit/s, which is equal to 2.5 times the video bitrate, on average. We thus expect to provide acceptable performance if at most 60% of the peers is behind a firewall.

C. Hook-in Point

As mentioned in Section III-C, every peer hooks in on the live stream by downloading B pieces before the latest piece they see at at least half of their neighbours. A small value of B results in a short prebuffering time. But since B also functions as a buffer against fluctuations in the download speed, choosing a too small value for B will lead to peers losing pieces or stalling. We measured the effect of varying B in a system with no firewalled peers. Figure 3 plots the measured performance. The average required prebuffering time is plotted against the buffer size B , as well as the average percentage of time a peer spent either losing pieces or stalling.

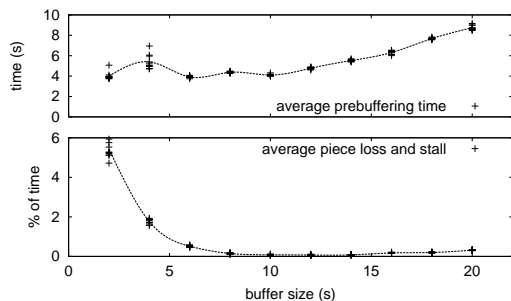


Figure 3. The average prebuffering time (top) and the average percentage of time spent losing pieces or stalling (bottom) versus the size of the buffer of the prebuffering phase.

As B increases, peers experience a better playback quality (fewer losses and stalls), but have to wait longer for playback to start. A value for B of 10 seconds (= 20 pieces) is a good trade off, and this is the value we will use in our trial (see Section V).

D. Piece Size

Figure 4 plots the performance obtained in our simulations when the size of the BitTorrent pieces varies. Each line represents a different percentage of peers behind a firewall. Again, the performance of the system degrades as the percentage of firewalled peers increases. Furthermore, both small and large piece sizes provide poor performance. Very small pieces introduce too much overhead, and large pieces require too long to download, or may never even be downloaded completely before their playback deadline. According to our simulations, the best performance is obtained using pieces of 16 Kbyte or 32 Kbyte in size. In our trial, we use pieces of 32 Kbyte, because the piece-size simulations were performed after our trial.

E. Number of Seeders

Finally, we test the impact of the number of seeders. The results of these tests are shown in Figure 5. We performed simulations with 1 to 20 seeders per swarm (against approximately 600 peers in total as one peer arrives every second for a period of 10 minutes), and with varying percentages of peers behind firewalls. Figure 5 indicates that increasing the number of seeders significantly increases performance, in case the performance is poor due to a high percentage of peers behind firewalls.

V. PUBLIC TRIAL

In this section, we will first describe how our trial was set up. Then, we will discuss the main performance characteristics: the size of the swarm, the amount of piece loss and stalling experienced by the peers, the prebuffering time they needed, and their sharing ratios. The sharing ratio of a set of peers is the ratio of the number of uploaded bytes and the number of downloaded bytes, which is a metric for their resource contribution and for the scalability of our solution.

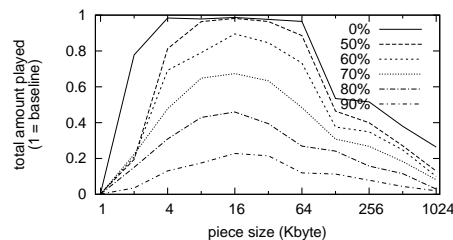


Figure 4. The total amount of played data versus the piece size, for several percentages of firewalled peers.

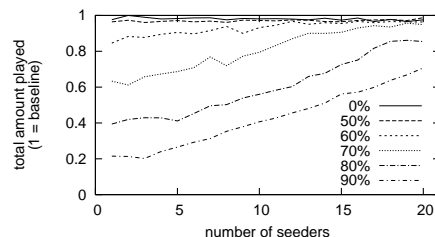


Figure 5. The total amount of played data versus the number of seeders, for several percentages of firewalled peers.

A. Trial Setup

We have developed a P2P video player called the *Swarm-Player*, which is based on the Tribler [2] core and the VideoLan Client (VLC) [12] video player. It runs a mature BitTorrent implementation plus our proposed extensions. In a public trial on 17th – 26th July 2008, we invited users to watch a specific live video stream using the SwarmPlayer. Over the course of 9 days, we saw 4555 users from around the globe tune in.

We were not able to obtain the rights to stream popular copyrighted content to a world-wide audience, which would attract a lot of viewers. Instead, we used an outdoor DV camera, aimed at the Leidseplein, which is a busy square in the center of Amsterdam. We transcoded the live camera feed to a 512 Kbit/s MPEG-4 video stream. The video stream was wrapped inside an MPEG Transport Stream (TS) container to make it easier for clients to start playing at any offset within the byte stream, and cut into 32 KByte BitTorrent pieces. Our extensions for live streaming were configured with a value of A (the window size of the injector) of 7.5 minutes, and with a value of B , the size of the buffer in the prebuffering phase, of 10 seconds. We deployed an injector as well as five seeders. The performance results in the following sections will exclude these peers.

To participate in our trial, a user had to download the SwarmPlayer and the tstream file representing our DV camera. Once the SwarmPlayer was started, it sent status updates to our logging server at 30 second intervals. The logging server checked for each peer whether it is firewalled by trying to connect to it. UPnP remote firewall control is supported.

We will compare the results of our trial with two other measurements of deployed live P2P streaming systems. The first, by Xie et al. [9], measures a deployment of the open protocol Coolstreaming [10] on up to 40,000 concurrent peers spread over an unknown number of swarms, delivering a 768 Kbit/s stream. The second, by Agarwal et al. [8], measures the

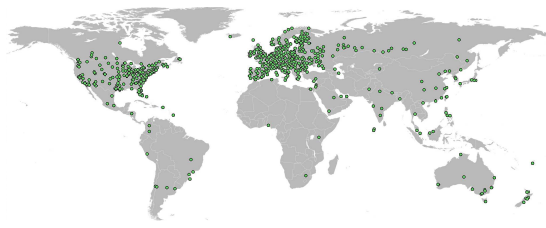


Figure 6. The locations of all users.

deployment of a closed protocol on up to 60,000 concurrent peers within a single swarm, delivering a 759 Kbit/s stream. These measurements assume a different set up and measure a different set of users. Nevertheless, they will provide a source of comparison that allows us to interpret our results.

B. Performance over Time

During the trial, users from 4555 different IP addresses joined our swarm 6935 times in total. Figure 6 shows the locations of all users. These locations were derived from the IP addresses using publicly available databases. Our injector and seeders are located in the Netherlands.

Figure 7 plots the size of the swarm over time (subfigure (a)) as well as the average performance of the peers (subfigures (b)–(d)). Each data point is the average for an hour to improve readability. All time stamps are GMT.

The public trial ran from Thursday July 17th 2008 until Saturday July 26th 2008. Several events occurred during the course of the trial which have been marked in Figure 7. The trial was announced through several media (Ars Technica, BBC News, and others) before and after the first weekend. As the time line shows, the trial was not without problems. On Saturday 19th, a new SwarmPlayer had to be released which fixed a few bugs that we will mention later. On Tuesday 22nd, our DV camera went briefly off line, causing all peers to freeze playback. Once the camera was plugged back in, the system resumed without requiring any intervention. The injector itself ran uninterrupted during the whole trial.

The size of the swarm (Figure 7(a)) varies between 0 and 86 peers, although only up to 76 peers stayed long enough to be readable in the figure. Although we would have liked to see more concurrent users, repeating the trial was not feasible for us. Most of the peers arrived after the press release on Friday 18th, and a boost in the arrival rate can be clearly seen on Monday 21st, when the second press release was made. On most of the days, the swarm is at its smallest at night time. The contrast between the high number of arrivals (6935) and the rather small swarm indicates that most visits are brief. Indeed, Figure 8 plots the duration of all sessions from long to short. The median session duration is 100 seconds, with 560 sessions lasting longer than an hour. The minimum session duration we could measure is 30 seconds, because that is the interval with which peers send status reports to our logging server.

Figure 7(b) shows the average bitrate of pieces that were received on time. Two aspects are of interest. First, the low bit rates at night and the spikes at dawn are artefacts of our video encoder. Second, there is a drop in playback rate on Saturday

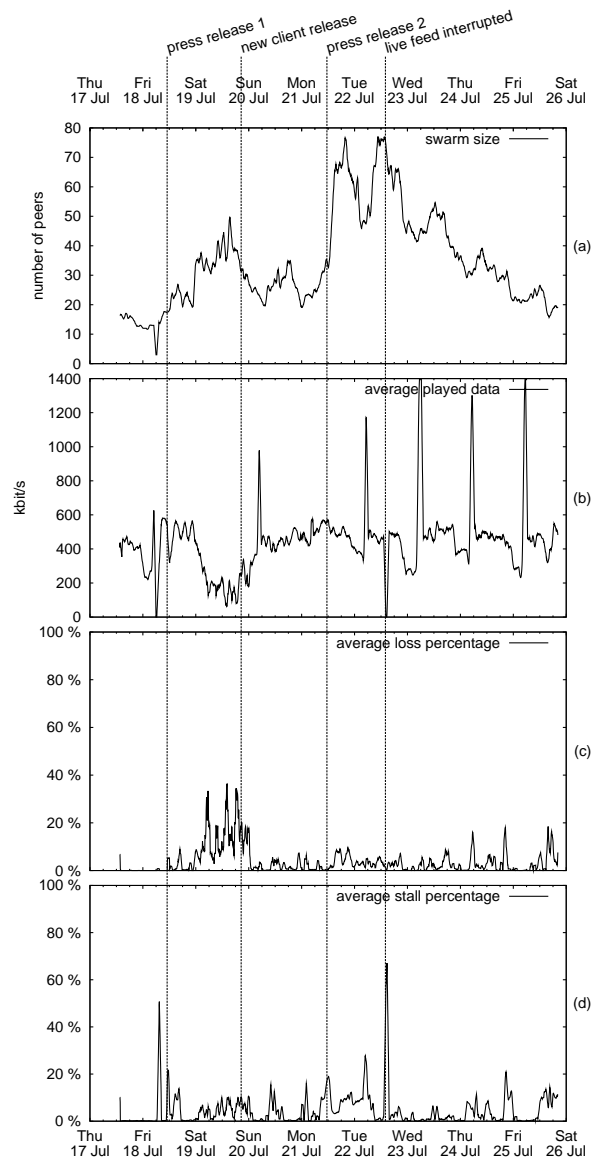


Figure 7. The performance during the trial.

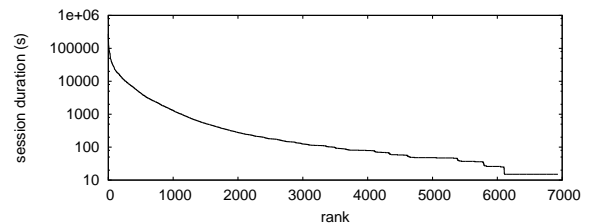


Figure 8. The duration of the user sessions, ranked according to length.

19th, which was caused by client malfunction. We released a new version of the SwarmPlayer the same day, and the system resumed its operation.

Figure 7(c) shows the percentage of pieces lost. Most of the piece loss occurred on Saturday 19th as is to be expected. Overall, the rest of the piece loss is low, and is mostly concentrated on a few peers, as is shown in Figure 9, which shows the percentage of pieces lost within the individual sessions. A small subset of the peers experience a high percentage of

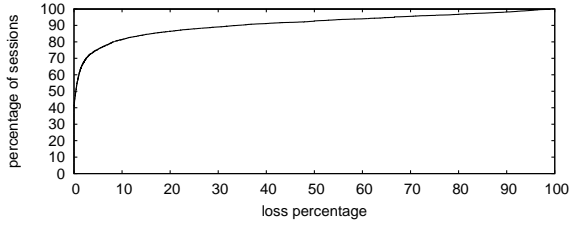


Figure 9. The cumulative distribution of the percentage of pieces lost per user session.

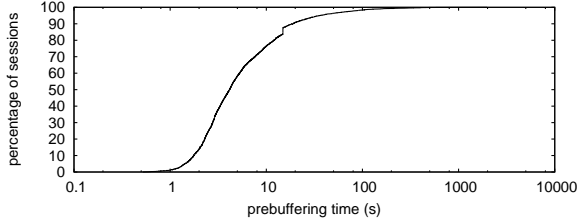


Figure 10. The cumulative distribution of the prebuffering time per user session.

piece loss, but most peers experience almost no loss. There are at least two reasons why a peer may lose many pieces. First and foremost, since peers barter for pieces using the BitTorrent protocol, our extensions inherit any inefficiencies present in BitTorrent. Second, some peers may not have the necessary bandwidth to watch the video stream in the first place, due to having a narrow up- or downlink or due to side traffic. Xie et al. [9] measure an average piece loss of 1%, which is less than our average of 6.6%. On the other hand, Agarwal et al. [8] measure a median piece loss of 5%, while we measured 0.4%.

Figure 7(d) shows the average percentage of time the peers were stalled. The peak on Tuesday 22nd is caused by our DV camera being disconnected. Note that the peak of 50% on Friday 18th is actually measured over only two peers, one of them stalling. Overall, the stall time typically fluctuates between 0% and 10%. We found most of this stall time to be present just after a peer started playback, which is an artifact of the SwarmPlayer not being able to predict how much data the video playback module (VLC) will discard before playback is started. If VLC discards too much data, the peer will have to stall in order to stay synchronised with its neighbours. Neither Xie et al. [9] nor Agarwal et al. [8] mention their peers stalling video playback. We assume their algorithms use a longer prebuffering time, to create a large enough buffer which can handle most scenarios.

C. Prebuffering Time

The prebuffering time determines how long it takes for a peer to start playback. Even though a long prebuffering time allows the player to accumulate a large buffer and therefore minimise piece loss and stalling, we regard a short prebuffering time to be more desirable since it reduces the amount of time the user will have to wait before viewing the first frame. The distribution of prebuffering times required in our trials is shown in Figure 10. The median prebuffering time was 3.6 seconds, with 67% of the peers requiring less than 10 seconds. A few peers require substantially more prebuffering

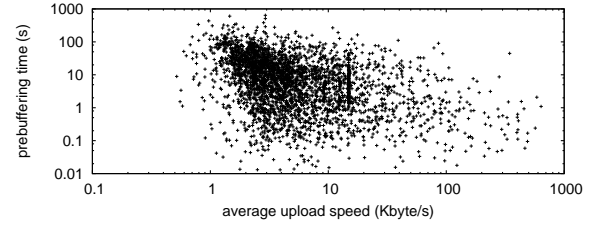


Figure 11. The prebuffering time against the average upload speed for all user sessions.

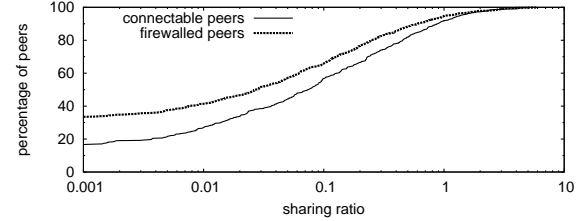


Figure 12. The cumulative distribution of the sharing ratios of the connectable and firewalled peers.

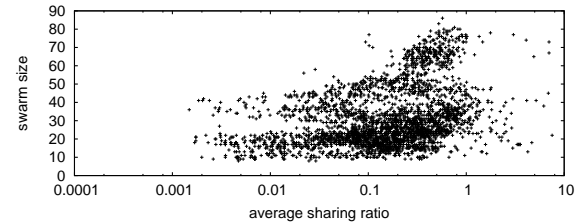


Figure 13. The swarm size against the average sharing ratio for every five-minute interval.

time. A possible explanation is that such peers do not provide enough upload bandwidth for BitTorrent to perform the tit-for-tat bartering. The relation between the prebuffering time and the average upload speed in each session is shown in Figure 11. A negative correlation is clearly visible, in which peers with low average upload speeds tend to require longer prebuffering times. Average upload speeds higher than the video bitrate are possible for short sessions, since all pieces can be downloaded simultaneously in the prebuffering phase.

Xie et al. [9] measure a median prebuffering time between 10 and 20 seconds, and Agarwal et al. [8] find a median prebuffering time of 27 seconds. Both of these figures are significantly larger than our 3.6 second median.

D. Sharing Ratios

The data that are not uploaded by our injector and our seeders is uploaded by the peers. We aim to distribute the burden of uploading fairly among the peers, to avoid having to rely on a subset of altruistic peers. As a measure of fairness, we use the *sharing ratio* of a peer (or a group of peers), which is defined as the number of uploaded bytes divided by the number of downloaded bytes. Peers with a sharing ratio smaller than 1 are net consumers, those with a sharing ratio larger than 1 are net producers.

In our trial, we found 61% of the IP addresses to be firewalled, and firewalled peers accounted for 52% of the on-line time of all peers. Figure 12 plots the cumulative

distribution of the sharing ratios for both the firewalled and the connectable (non-firewalled) peers. The difference in sharing ratios is clearly visible. Since the firewalled peers can only form connections with connectable peers, they are limited in the number of peers they can upload to. On the other hand, the connectable peers can become connected to any other peer. The connectable peers are thus able to upload much more data than the firewalled peers, which is consistent with the findings in our previous work [3]. In total, the connectable peers upload 0.41 as much as they download, while the firewalled peers upload only 0.18 times as much.

Our injector and seeders provided the rest of the data, which amounts to 74% of the pieces. Even though we employed five seeders, they were hardly used. The injector supplies 72% of the pieces, and the seeders only 1.6%. One reason for this bias is the fact that the injector obtains and announces each piece slightly before any seeders do. Any peer connected to both the injector and a seeder will thus see each piece appear at the injector first, and will request it immediately if possible. Apparently, the injector provided enough bandwidth so that the peers did not have to fall back on and request pieces from the seeders. The seeders were thus not needed to serve our limited number of users, but also did not have to provide more than a token amount of resources.

Figure 13 shows the sharing ratios of all peers for each five-minute interval. For small swarms, most peers receive their data from the injector and the seeders as they are the first to obtain each piece, and they have enough upload bandwidth to serve all peers. When the swarm grows, the peers start to forward the stream to each other. The average sharing ratio clearly improves for larger swarms, which is an indication for the scalability of our approach. Note that sharing ratios higher than 1 are due to artefacts of measurement, in which data is uploaded in a different five-minute interval than it is downloaded, or one of the peers disconnects before reporting the number of downloaded bytes.

In Xie et al. [9], 70% of the peers were firewalled, and the other 30% of the peers provided 80% of the bandwidth exchanged by the peers, but they do not mention how much data are actually coming from their central servers. Their figures do however, like ours, indicate a significant skew in the resource contribution towards the connectable peers. The peers measured by Agarwal et al. [8] have roughly 10% of the data coming from their central servers. They report 84% of their peers to be firewalled; since these firewalled peers have to obtain the data from the 16% connectable peers, the latter will have to provide an amount of upload bandwidth equal to several times the video bitrate, which correlates with the sharing ratio distribution given in [8].

The performance of all measured systems, including ours, depends on the upload bandwidth the injector and the connectable peers are able and willing to provide. Since a high percentage of firewalled peers in a deployed system seems unavoidable, the injector and the connectable peers may not be able to provide all of them with the video stream. Firewall traversal techniques will have to be deployed in order to

increase the contribution of the firewalled peers.

VI. CONCLUSIONS

We presented extensions to the BitTorrent protocol which add live streaming functionality. Among other things, we added a rotating sliding window over a fixed set of pieces in order to project an infinite video stream onto a fixed number of pieces. When our extensions as well as Video-on-Demand extensions [13], [14] are added to a BitTorrent client, a single solution is created for streaming live and pregenerated video, as well as for off-line viewing of high-definition content. The synergy between these components reduces the time to implement them, and allows them all to benefit from future improvements to BitTorrent.

We used simulations to estimate the optimal values of several parameters. Also, we found that the percentage of peers behind a firewall can be a decisive factor for the performance. We implemented our extensions into our BitTorrent client called Tribler, and launched a public trial inviting users to tune into our DV camera feed. Indeed, thousands of people around the globe downloaded our video player and joined the swarm. The performance we measured varies across the peers as is to be expected, but for most peers lies within bounds we find acceptable. Most clients could start playing within four seconds after the client was loaded, which is significantly faster than the results found by others in different systems.

REFERENCES

- [1] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems*, 2003.
- [2] J. Pouwelse and et al., "Tribler: A Social-Based Peer-to-Peer System," in *Concurrency and Computation: Practice and Experience*, vol. 20, 2008, pp. 127–138.
- [3] J. Mol, J. Pouwelse, D. Epema, and H. Sips, "Free-riding, Fairness, and Firewalls in P2P File-Sharing," in *Proc. of the 8th IEEE Intl. Conf. on Peer-to-Peer Computing*, 2008, pp. 301–310.
- [4] L. Mathy, N. Blundell, V. Roca, and A. El-Sayed, "Impact of Simple Cheating in Application-Level Multicast," in *Proc. of IEEE INFOCOM*, vol. 2, 2004, pp. 1318–1328.
- [5] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," in *Proc. of the 7th USENIX OSDI*, 2006, pp. 191–206.
- [6] J. Mol, D. Epema, and H. Sips, "The Orchard Algorithm: Building Multicast Trees for P2P Video Multicasting Without Free-riding," *IEEE Transactions on Multimedia*, vol. 9, no. 8, pp. 1593–1604, 2007.
- [7] S. Ali, A. Mathur, and H. Zhang, "Measurement of Commercial Peer-To-Peer Live Video Streaming," in *Workshop in Recent Advances in Peer-to-Peer Streaming*, 2006.
- [8] S. A. et al., "Performance and Quality-of-Service Analysis of a Live P2P Video Multicast Session on the Internet," in *Proc. of IwQoS*, 2008.
- [9] S. Xie, G. Y. Keung, and B. Li, "A Measurement of a Large-Scale Peer-to-Peer Live Video Streaming System," in *Proc. of the IEEE Intl. Conf. on Parallel Processing Workshops*, 2007, p. 57.
- [10] X. Zhang, J. Lieu, B. Li, and T.-S. P. Yum, "DONet/Coolstreaming: A Data-driven Overlay Network for Live Media Streaming," in *Proc. of IEEE INFOCOM*, 2005.
- [11] P. Dhungel, X. Hei, K. Ross, and N. Saxena, "The Pollution Attack in P2P Live Video Streaming: Measurement Results and Defenses," in *Proc. of the Workshop on Peer-to-Peer Streaming and IP-TV*, 2007, pp. 323–328.
- [12] VideoLan Client (VLC), "<http://videolan.org/>."
- [13] J. Mol, J. Pouwelse, M. Meulpolder, D. Epema, and H. Sips, "Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems," in *Proc. of SPIE*, vol. 6818, Article 681804, 2008.
- [14] A. Vlavianos, M. Iliofotou, and M. Faloutsos, "BiToS: Enhancing BitTorrent for Supporting Streaming Applications," in *IEEE Global Internet Symposium*, 2006.