Delft University of Technology
Parallel and Distributed Systems Report Series

The Design and Deployment of a BitTorrent Live Video Streaming Solution

J.J.D. Mol, A. Bakker, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips

{J.J.D.Mol, A.Bakker, J.A.Pouwelse, D.H.J.Epema, H.J.Sips}@tudelft.nl

report number PDS-2009-002

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
`reports@pds.twi.tudelft.nl`

Information about Parallel and Distributed Systems Section:
`http://pds.twi.tudelft.nl/`

© 2009 Parallel and Distributed Systems Section, Faculty of Information Technology and Systems, Department of Technical Mathematics and Informatics, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

Abstract

The BitTorrent protocol is by far the most popular protocol for offline peer-to-peer video distribution on the Internet. BitTorrent has previously been extended to support the streaming of recorded video, that is, Video-on-Demand (VoD). In this paper, we take this support for video streaming a step further by presenting extensions to BitTorrent for supporting live video streaming, which we have implemented in our BitTorrent client called Tribler. We have tested our extensions both by running simulations, and by deploying our implementation in a public trial in the Internet, using the optimal values of several parameters as found in the simulations. We analyse the performance of Tribler in systems with varying values for the percentage of peers behind a firewall or NAT, which we consider to be one of the key parameters in the performance of deployed P2P systems.

Our public trial lasted 9 days, during which 4555 unique peers participated from around the globe. We found 61% of the peers to be behind a firewall, a level at which our simulations still indicate acceptable performance. Most of the peers indeed obtained good performance, with for instance a very low prebuffering time before playback starts, indicating the feasibility of our approach. The median prebuffering time we measured was 3.6 seconds, which is 3–10 times shorter than the prebuffering time measured in other deployed peer-to-peer systems.

Contents

1 Introduction **4**

2 Background **5**
2.1 BitTorrent . . . 5
2.2 Related Work . . . 5

3 Extensions for Live Streaming **5**
3.1 Unlimited Video Length . . . 6
3.2 Data Validation . . . 7
3.3 Live Playback . . . 7
3.4 Seeders . . . 8

4 Simulations **9**
4.1 Simulation Setup . . . 9
4.2 Uplink Bandwidth . . . 9
4.3 Hook-in Point . . . 10
4.4 Piece Size . . . 10
4.5 Number of Seeders . . . 10

5 Public Trial **11**
5.1 Trial Setup . . . 12
5.2 Performance over Time . . . 12
5.3 Prebuffering Time . . . 15
5.4 Sharing Ratios . . . 15

6 Conclusions **17**

List of Figures

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. . . . 6

2 The injector, seeders and leechers in a live streaming setting. The bold nodes are seeders. The bold edges are connections which are guaranteed to be unchoked. . . . 8

3 The total amount of data played versus the average uplink bandwidth, for several percentages of firewalled peers. . . . 9

4 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. . . . 11

5 The total amount of played data versus the piece size, for several percentages of firewalled peers. 11

6 The total amount of played data versus the number of seeders, for several percentages of firewalled peers. . . . 11

7 The locations of all users. . . . 13

8 The performance during the trial. . . . 14

9 The duration of the user sessions, ranked according to length. . . . 15

10 The cumulative distribution of the percentage of pieces lost per user session. . . . 15

11 The cumulative distribution of the prebuffering time per user session. . . . 16

12 The prebuffering time against the average upload speed for all user sessions. . . . 16

13 The cumulative distribution of the sharing ratios of the connectable and firewalled peers. . . 16

14 The swarm size against the average sharing ratio for every five-minute interval. . . . 16

List of Tables

1 Introduction

The BitTorrent protocol [5] has been a popular method for P2P file sharing since its introduction in 2001. It has originally been designed to allow a group of peers to download the same file, to be viewed only after the download is completed. The BitTorrent protocol has many implementations, and has probably been deployed millions of times. We thus consider the BitTorrent protocol to be a proven technology which has several proven implementations. With BitTorrent, files are received out of order, making the BitTorrent protocol unsuitable for streaming purposes. In this paper, we propose extensions to BitTorrent to support live video streaming. Our approach is to take advantage of the proven BitTorrent file-sharing technology for video streaming by extending the former to support the latter, allowing future improvements of the BitTorrent protocol to improve the performance of our extensions as well.

By incorporating the extensions presented in this paper, any BitTorrent client can become a single platform for both downloading high quality videos and for streaming content. We have created such a client by incorporating the extensions proposed in this paper into Tribler [14], which already supports Video-on-Demand (VoD) based on a BitTorrent core. The main difference between VoD and live streaming is the moment a video becomes available. In VoD, a video is made available for download only after it has been generated completely, while in live video streaming, a video is made available while it is being generated.

Although the difference between VoD and live video streaming may seem minor from a user's perspective, the technological differences are actually fundamental when considering a BitTorrent-like system. First, both the original BitTorrent protocol and its VoD extensions assume all video data to be known beforehand. The total length of the data as well as hashes of it are generated before the file is offered for download. Such information is not available in a live streaming session. The hashes of the data can only be calculated when all of the data have been generated, but the length of the stream is unknown in many scenarios (for example, when broadcasting TV channels). Secondly, in contrast to VoD, peers in live streaming are always playing approximately the same part of the stream. All peers thus require roughly the same data, and cannot download faster than the stream is generated. Finally, when in VoD peers finish downloading the complete stream, they can share it with others for free (called *seeding*), which considerably increases the download performance of the other peers. We have previously observed BitTorrent systems in which 90% of the peers are seeding. In live streaming, no peer is ever done downloading, so no seeds exist. In this paper we will provide solutions to these three issues, and we will present extensions to BitTorrent that implement these solutions, thus enabling live video streaming.

We will evaluate the feasibility of our approach by using simulations as well as a deployed system. Because it is difficult to capture the full dynamics of a deployed P2P system through simulations, we mainly used simulations to estimate some of the values for the parameters that are used in our extensions (e.g., the prebuffering time, which is the time used for the initial downloading of video data before the playback starts). We also focus on varying the percentage of peers in the system that are behind firewalls or NATs. Previous research has shown [11] this percentage to be a key factor in the performance of P2P content distribution systems. The deployment of our extensions consisted of a public trial using our Tribler BitTorrent client. We deployed an outdoor DV camera, and invited users to tune in using our BitTorrent client. Over the course of 9 days, 4555 unique users participated in our trial.

This paper is organised as follows. First, we will briefly describe BitTorrent and discuss related work in Section 2. Then, we will present our extensions to BitTorrent in Section 3. Section 4 contains the set up and results of our simulations. Section 5 will describe the set up and results of our public trial. Finally, we will draw conclusions in Section 6.

The research leading to this contribution has received funding from the European Community's Seventh Framework Programme in the P2P-Next project under grant agreement no 216217.

2 Background

In this section, we will present the background of our live video streaming extensions of BitTorrent. First, we will briefly describe the BitTorrent protocol as it forms the basis of our work; a more extensive description can be found in [5]. Then, we will discuss the differences between our approach and those of other live video streaming systems.

2.1 BitTorrent

The BitTorrent protocol [5] operates as follows. The injector creates a *torrent* meta-data file, which is shared with the peers through, for example, an HTTP server. The file to be shared is cut into fixed-size pieces, the hashes of which are included in the torrent file. A central server called a *tracker* keeps track of the peers in the network. The peers exchange data on a tit-for-tat basis in the *swarm* of peers interested in the same file: the neighbours of a peer that provide the most data are allowed to request pieces in return (they are *unchoked*). Once a peer has completed the download of a file, it can continue to *seed* it by using its upload capacity to serve the file to others for free.

2.2 Related Work

Many P2P live streaming algorithms have been proposed in the literature [4, 7, 10, 13, 15]. Most of these algorithms are designed to operate in a cooperative environment, which makes them easy to abuse when used on the Internet [8]. The BAR Gossip [7] and Orchard [10] algorithms do not assume full cooperation from the peers, but the P2P framework in which they operate is out of the scope of either algorithm, nor do they include measurements of a deployed system.

Deployed live streaming P2P solutions have been measured before [2, 3, 18] and reveal a decent performance. Ali et al. [3] and Agarwal et al. [2] measure the performance of commercial P2P live streaming systems, but they do not describe the P2P distribution algorithm that was used. Xie et al. [18] measure a deployed and open protocol called Coolstreaming [19]. Our approach is different in that we extend BitTorrent, which can be deployed in environments in which peers do not necessarily behave well. We will compare our results with those found by Agarwal et al. [2] and Xie et al. [18] to provide insight into our results.

3 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. First, the length of the video, and therefore the number of pieces that will be generated, is unknown, but this number is required throughout the BitTorrent algorithm. Secondly, the pieces themselves are not known in advance, which precludes the calculation of their hashes beforehand, which is required for the torrent file and for data validation by the peers. Thirdly, peers want to download only the latest generated pieces, for which BitTorrent does not define a suitable piece-selection policy. Finally, seeders do not exist in a live streaming environment, even though they significantly boost the download performance in regular BitTorrent swarms. We will discuss these problems and their solutions in the following sections.

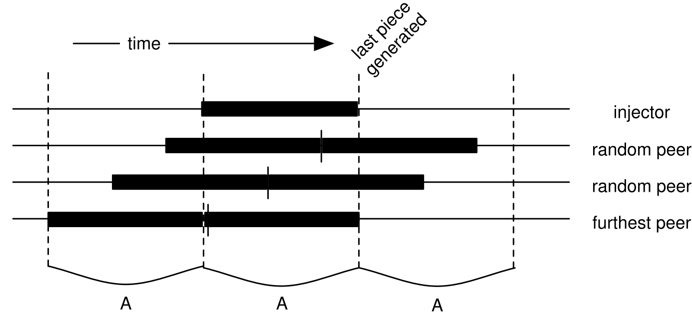

Figure 1: The sliding windows of valid pieces for the injector, two random peers, and the peers furthest from the injector in terms of delay. The vertical markers on each bar indicate the peer's playback position.

3.1 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 increase the number of pieces to a very large value that will not be reached (for instance, 2^{32}). Instead, we use a sliding window that rotates over a fixed number of pieces. Pieces with numbers outside this window will be considered to be out-dated. Each peer deletes out-dated pieces, and will consider them to be deleted by its neighbours as well, thereby avoiding the need for additional messages. If a piece is out-dated by the time its downloaded finishes, it will not be offered for download to other peers. Within its sliding window, each peer barters for pieces according to the BitTorrent protocol.

The sliding window of the injector is defined as the sequence of the A pieces that have been generated last, for some value of A. Since the injector has to be able to serve any peer in case all other peers depart, we base the definition of the sliding windows of the other peers on the assumption that their playback positions are not more than A pieces behind the latest generated piece. This means that the playback position of any peer falls within the sliding window of the injector. A peer that is not connected to the injector has no way of knowing where its playback position is with respect to the injector. When it is (almost) synchronized with the injector, of course it makes sense to have its sliding window extend A pieces behind its current playback position. On the other hand, when it is lagging (almost) A pieces behind the injector, the A pieces beyond its current playback position are already available in the system, and it makes sense to have its sliding window extend A pieces beyond this position. Therefore, every peer (except the injector) maintains a sliding window of both A pieces ahead of and A pieces behind its current playback position. Figure 1 illustrates the concept of the sliding window by showing a time line of each peer and a horizontal bar denoting the sliding window. Note that not all of the pieces within the sliding window of a peer may actually already exist, as the sliding windows may extend beyond the latest generated piece. We will use a rather large value of A equivalent to 7.5 minutes of video in our trial, which allows the system to operate in a broad range of networks.

When a peer joins the network, its neighbours will inform him about the pieces they own as per the BitTorrent protocol. Then, the peer has to decide which of these pieces are young, and which are old, in order to synchronise its sliding window with the rest of the system. Even though all the reported pieces will fall a $2A$ piece range, as is shown in Figure 1, the piece numbers do wrap around. In order for a peer to be able to tell which pieces are the youngest, it must be able to distinguish the beginning and end of the $2A$ piece range. If there are at least $4A$ pieces over which the sliding windows rotate, there will be at least $2A$ consecutive piece numbers that will not be reported by any neighbour. The last reported piece before this empty range has to be the youngest one available. A peer will realign its sliding window when new information becomes available, for example if it connects to more neighbours. Malfunctioning and malicious clients may claim to have pieces outside the range that is considered valid by the other peers. To avoid basing the sliding window on such clients, a peer uses majority voting to determine the correct current position of its sliding window: only the pieces that are available at more than half of its neighbours are considered to

be valid.

3.2 Data Validation

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. However, the lack of hashes makes it impossible for the peers to validate the data they download, leaving the system 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. [6] 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 is assigned an absolute sequence number, starting from zero. Each piece, along with its absolute sequence number and time stamp, is signed by the injector. Such a scheme allows any peer to validate any piece as originating from 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. As a bonus, the included time stamps allow a peer to estimate the delay between the injector and its own playback position. We use 64-bit integers to represent the sequence numbers and the timestamps, so a wrap around will not occur within a single session. Also note that the major difference between sequence numbers and the BitTorrent piece numbers is a matter of implementation. Many BitTorrent implementations allocate arrays that span all possible piece numbers. The use of 64-bit integers for the piece numbers would thus require refactoring of the whole implementation.

The payload of each piece is slightly reduced to make room for the metadata and the signature. 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 disconnect from the neighbour that provided the piece. A piece is outdated if the timestamp indicates it was generated A pieces or longer ago.

The validity of a piece can only be verified once it has been downloaded completely, and only verified pieces are offered to others and to the video player. As a result, small pieces reduce the delay of each hop traversed by the piece. On the other hand, if pieces are too small, the overhead of dealing with an increased number of pieces increases as well. In Section 4.4, we will show through simulation that the common piece sizes in BitTorrent of 256 Kbyte to 2 MByte are too big for live streaming. Piece sizes of 16 and 32 KByte provide better performance. We will use 32 KByte pieces in our trial.

3.3 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.

During playback, a peer maintains a playback buffer of pieces to send to the video player in-order. The first amount of data sent to the video player is discarded as the video player seeks to the first complete picture to display. Since our algorithm deals with byte streams and does not know any frame boundaries,

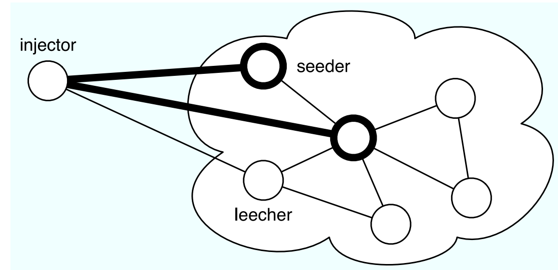

Figure 2: The injector, seeders and leechers in a live streaming setting. The bold nodes are seeders. The bold edges are connections which are guaranteed to be unchoked.

the amount of data discarded is hard to predict accurately. As a result, it is theoretically possible for a peer to have a buffer underrun immediately after playback has commenced.

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 4.3.

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 never reached a peer's neighbours, 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 if the neighbourhood of the peer changes. The new 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.

3.4 Seeders

In BitTorrent swarms, some of the peers may have completed their download and seed the content to others. 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 has ever finished downloading the video stream.

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 list of peer identifiers (for example, IP addresses and port numbers) representing trusted peers which are allowed to act as seeders if they connect to the injector. The relationship between the injector, seeders and leechers is shown in Figure 2. The seeders and leechers use the exact same protocol, but the seeders (bold nodes) are guaranteed to be unchoked by the injector (bold edges). The identity of the seeders is not known to the other peers to prevent malicious behaviour targeted at the seeders.

The injector thus controls the set of seeders, and has two incentives for maintaining that set. First, the seeders can provide their upload capacity to the network, taking load off the injector. The seeders behave like a small Content Delivery Network (CDN), which boosts the download performance of other peers as in regular BitTorrent swarms. Second, the seeders increase the availability of the individual pieces. All pieces are pushed to all seeders, 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 4.5.

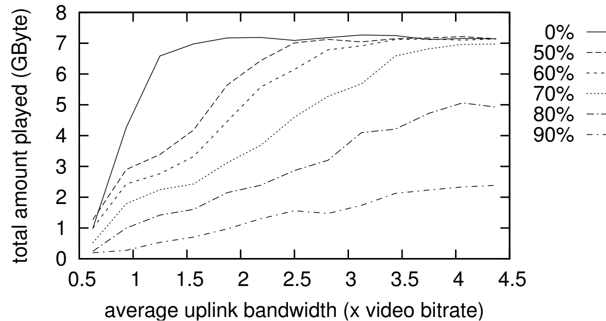

Figure 3: The total amount of data played versus the average uplink bandwidth, for several percentages of firewalled peers.

4 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.

4.1 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 stream of 512 Kbit/s using BitTorrent pieces of 32 KByte. Each peer departs after 5 minutes, on average, based on a uniform distribution. 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 between 100 and 300 ms. We have shown in previous work [11] 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%. 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 an outgoing 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*.

4.2 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 $1.25u$, and a downlink of four times their uplink. We use the total amount of data played by all peers together as our performance metric instead of the piece loss, for two reasons. First, piece loss figures give a

skewed result if little data could be played by the peers but nevertheless with little loss. Secondly, the piece loss figures would not include the time a peer has to wait before it starts playback. By using the amount of played data as a performance metric, both of these problems are avoided. Both an increase in piece loss and an increase in prebuffering time will result in a decrease in the total amount of data played. Note that the maximum amount of data that can be played for each data point is the same, because the arrival and departure patterns as well as the video bitrate, are equal between data points. We can thus compare the amounts of played data between data points. The best average over the ten arrival patterns we measured is 7.13 GByte of total data played by all of the peers, with for the peers an average piece loss of $<< 0.01\%$ and an average prebuffering time of 4.3 seconds. We thus consider the 7.13 GByte mark as a realistic optimal value for our setup.

Figure 3 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, in the presence of firewalled peers, the performance drops significantly and more bandwidth is needed. This result is consistent with our earlier findings on P2P data exchange in general [11]. If two peers cannot connect to each other because both are behind a firewall or NAT, they cannot exchange data. Such poor performance can only be avoided by the use of firewall puncturing and NAT traversal techniques. However, these techniques are highly complex and firewall/NAT-specific in nature. The BitTorrent protocol does not define such techniques. We therefore consider firewall puncturing and NAT traversal techniques to be outside the scope of this paper.

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.

4.3 Hook-in Point

We measure the effect of changing the hook-in point by varying the size B of the buffer of the prebuffering phase introduced in Section 3.3 in a system with no firewalled peers. Figure 4 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. As the size of the buffer B increases, the prebuffering time increases as more pieces have to be downloaded before playback can start. The amount of time spent by a peer losing pieces or stalling decreases when B increases, which is explained by the fact that a larger buffer provides each peer with more time to obtain missing pieces. A value for B of 10 seconds is a good trade off, and this is the value we will use in our trial (see Section 5).

4.4 Piece Size

Figure 5 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 will use pieces of 32 Kbyte.

4.5 Number of Seeders

Finally, we test the impact of the number of seeders. The results of these tests are shown in Figure 6. 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.

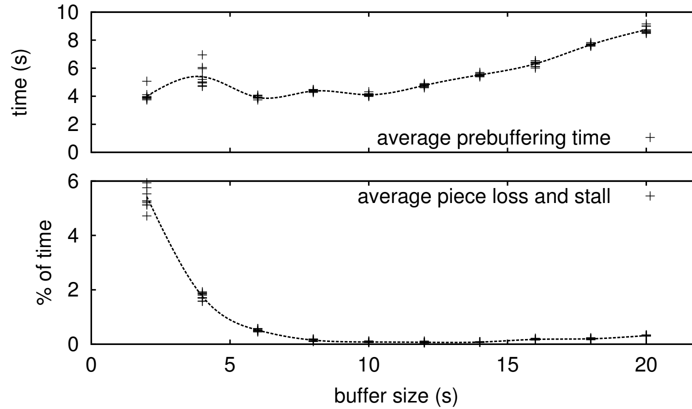

Figure 4: The average prebuffering time (top) and the average percentage of time spent losing pieces or stalling (bottom) versus the size of the buffer of the prebuffering phase.

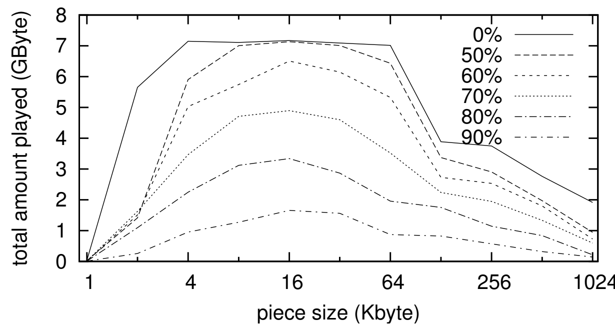

Figure 5: The total amount of played data versus the piece size, for several percentages of firewalled peers.

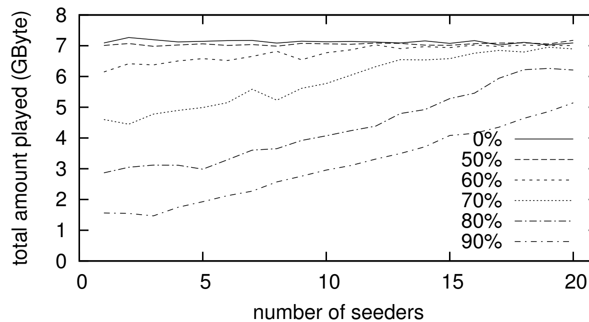

Figure 6: The total amount of played data versus the number of seeders, for several percentages of firewalled peers.

Figure 6 indicates that increasing the number of seeders significantly increases performance, in case tha performance is poor due to a high percentage of peers behind firewalls.

5 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.

5.1 Trial Setup

We have developed a P2P video player called the *SwarmPlayer*, which is based on the Tribler [14] core and the VideoLan Client (VLC) [16] video player. It runs the BitTorrent protocol with 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 unique IP-addresses 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) [1] 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 15 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.

The SwarmPlayer client was distributed for the Windows and Linux platforms. 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 stored these updates in a database. The logging server checked for each peer whether it is connectable (i.e., not behind a firewall) by trying to connect to it, since the fraction of unconnectable peers in a swarm bounds the amount of data that can be uploaded by the unconnectable peers [11]. The SwarmPlayer supports UPnP [9] remote firewall control, making any peer that supports it connectable.

We will compare the results of our trial with two other measurements of deployed live P2P streaming systems. The first, by Xie et al. [18], measures a deployment of the open protocol Coolstreaming [19] on up to 40,000 concurrent peers spread over an unknown number of swarms. The second, by Agarwal et al. [2], measures the deployment of a closed protocol on up to 60,000 concurrent peers within a single swarm. 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.

5.2 Performance over Time

During the trial, users from 4555 different IP addresses joined our swarm 6935 times in total. Figure 7 shows the locations of all the users that participated. These locations were derived from the IP addresses using publicly available databases. Our injector and seeders are located in the Netherlands.

Figure 8 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 8. 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 DV 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 8(a)) varies between 0 and 86 peers, although only up to 76 peers stayed long enough to be readable in the figure. 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.

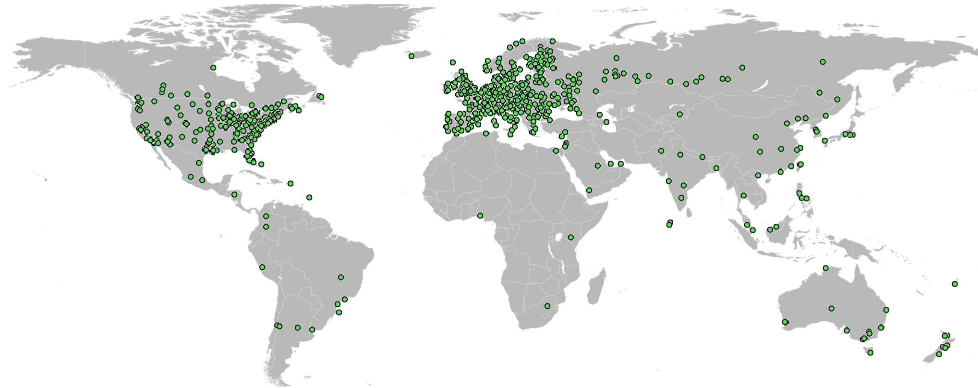

Figure 7: The locations of all users.

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 9 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 8(b) shows the average bitrate of pieces that were received on time and transferred to the video player. Two aspects are of interest. First, spikes in the data rate appear at dawn (4 am – 6 am), which are caused by our transcoder being unable to generate a 512 Kbit/s stream from a scene in which the lighting is slowly changing. These spikes are preceded by periods of lower bitrates, corresponding to night time at which compression is more efficient. Second, there is a drop in playback rate on Saturday 19th, which was caused by client malfunction. A programming glitch caused some players to freeze playback permanently, which caused confusion for arriving peers with respect to the current playback position. The small amount of data that is played on Saturday 19th mostly consists of arriving peers downloading and playing pieces they obtained from malfunctioning peers. We repaired the system by releasing a new version of the SwarmPlayer the same day, which introduced the majority voting for determining the playback position on arrival, as described in Section 3.1. Subsequent arrivals therefore were not easily fooled by the presence of a few malfunctioning peers, and the system resumed its operation.

Figure 8(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 10, which shows the percentage of pieces lost within the individual sessions. A small subset of the peers experience a high percentage of piece loss, but most peers experience almost no loss. There are at least two reasons why a peer may loose many pieces. First and foremost, since peers barter for pieces using the BitTorrent protocol, our extensions inherit any inefficiencies present in BitTorrent. Second, 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. Nevertheless, the measured piece losses are realistic as they are measured in a deployed system. A simulation is likely to underestimate this amount of loss, even though it is one of the key metrics for measuring the performance of a live video streaming system. Xie et al. [18] measure an average piece loss of 1%, which is less than our average of 6.6%. On the other hand, Agarwal et al. [2] measure a median piece loss of 5%, which is more than our median of 0.4%.

Figure 8(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. [18] nor Agarwal et al. [2] mention their peers stalling

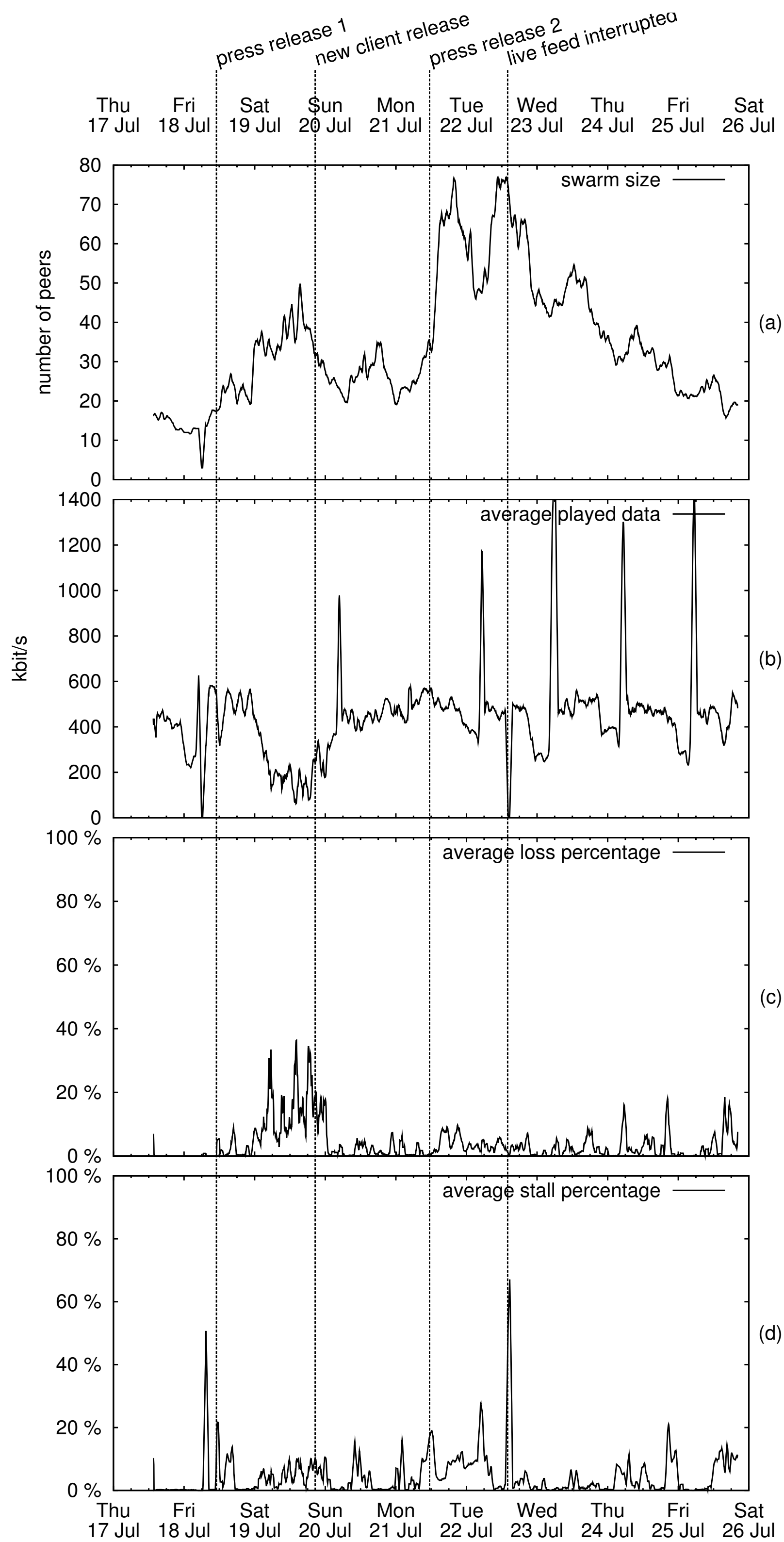

Figure 8: The performance during the trial.

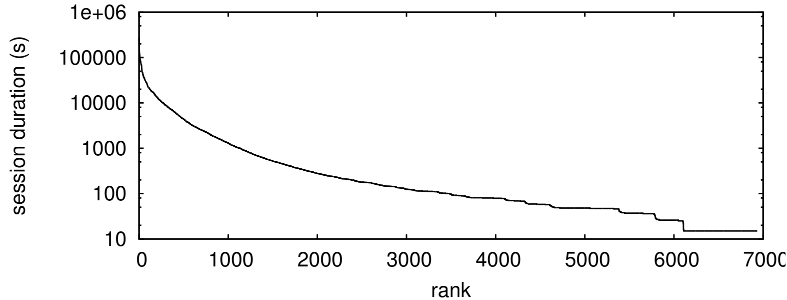

Figure 9: The duration of the user sessions, ranked according to length.

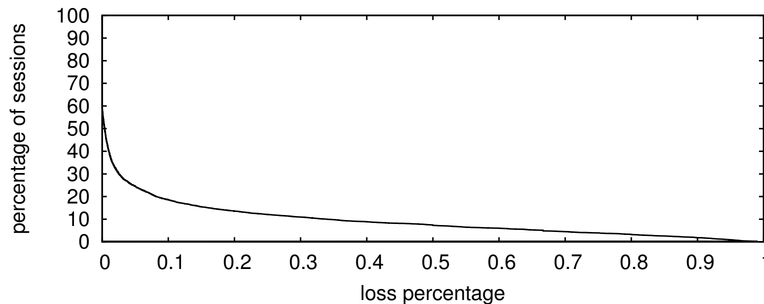

Figure 10: The cumulative distribution of the percentage of pieces lost per user session.

video playback. We assume their algorithms use a large enough buffer to accommodate for most scenarios, a fact that will be reflected in the differences in prebuffering times.

5.3 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 11. 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 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 12. 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 when the session is short, since all pieces can be downloaded simultaneously in the prebuffering phase.

Xie et al. [18] measure a median prebuffering time between 10 and 20 seconds, and Agarwal et al. [2] find a median prebuffering time of 27 seconds. Both of these figures are significantly larger than our 3.6 second median.

5.4 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

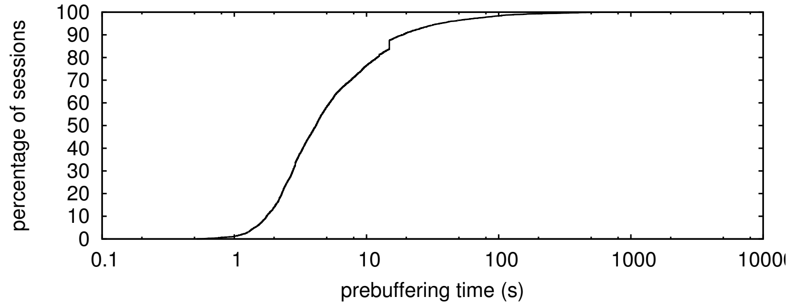

Figure 11: The cumulative distribution of the prebuffering time per user session.

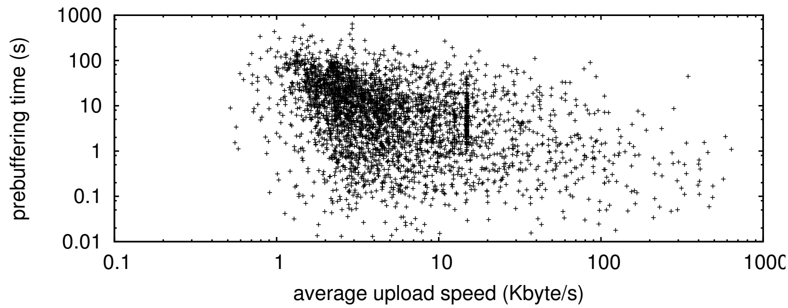

Figure 12: The prebuffering time against the average upload speed for all user sessions.

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.

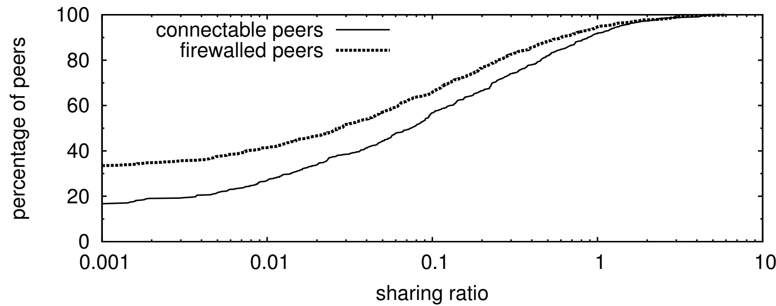

Figure 13: The cumulative distribution of the sharing ratios of the connectable and firewalled peers.

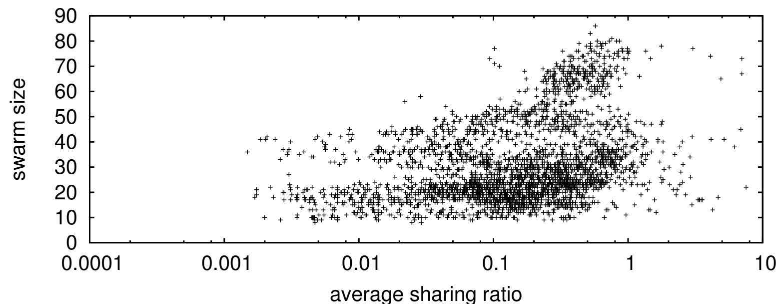

Figure 14: The swarm size against the average sharing ratio for every five-minute interval.

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 13 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. The connectable peers are able to upload much more data than the firewalled peers, which is consistent with the findings in our previous work [11]. 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.

Figure 14 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 scalibility 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. [18], 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. [2] 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 [2].

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.

6 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 [12, 17] 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] ISO/IEC international standard 13181-1; information technology - generic coding of moving pictures and associated audio information, 2000.

[2] S. Agarwal, J. P. Singh, A. Mavlankar, P. Baccichet, and B. Girod. Performance and Quality-of-Service Analysis of a Live P2P Video Multicast Session on the Internet. In *Proc. of the 16th IEEE Intl. Workshop on Quality of Service (IwQoS)*, 2008.

[3] 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.

[4] S. Banerjee, S. Lee, R. Braud, B. Bhattacharjee, and A. Srinivasan. Scalable Resilient Media Streaming. Technical Report UMIACS TR 2003-51, University of Maryland, 2003.

[5] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems*, 2003.

[6] 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*, pages 323–328, 2007.

[7] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *Proc. of the 7th USENIX OSDI*, pages 191–206, 2006.

[8] L. Mathy, N. Blundell, V. Roca, and A. El-Sayed. Impact of Simple Cheating in Application-Level Multicast. In *Proc. of IEEE INFOCOM*, volume 2, pages 1318–1328, 2004.

[9] Microsoft Corporation. Universal Plug and Play Internet Gateway Device v1.01. 2001.

[10] J. Mol, D. Epema, and H. Sips. The Orchard Algorithm: Building Multicast Trees for P2P Video Multicasting Without Free-riding. *IEEE Transactions on Multimedia*, 9(8):1593–1604, 2007.

[11] 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*, pages 301–310, 2008.

[12] 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*, volume 6818, Article 681804, 2008.

[13] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *Proc. of the 4th IPTPS*, pages 127–140, 2005.

[14] J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. van Steen, and H. Sips. Tribler: A Social-Based Peer-to-Peer System. In *Concurrency and Computation: Practice and Experience*, volume 20, pages 127–138, 2008.

[15] D. Tran, K. Hua, and T. Do. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. In *Proc. of the 22nd IEEE INFOCOM*, pages 1283–1292, 2003.

[16] VideoLan Client (VLC). `http://videolan.org/`.

[17] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In *IEEE Global Internet Symposium*, 2006.

[18] 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*, page 57, 2007.

[19] 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.