Spotify – Large Scale, Low Latency, P2P Music-on-Demand Streaming

Gunnar Kreitz
KTH – Royal Institute of Technology, and Spotify
Stockholm, Sweden
Email: gkreitz@kth.se

Fredrik Niemelä
KTH – Royal Institute of Technology, and Spotify
Stockholm, Sweden
Email: niemela@kth.se

***Abstract*—Spotify is a music streaming service offering low-latency access to a library of over 8 million music tracks. Streaming is performed by a combination of client-server access and a peer-to-peer protocol. In this paper, we give an overview of the protocol and peer-to-peer architecture used and provide measurements of service performance and user behavior.**

The service currently has a user base of over 7 million and has been available in six European countries since October 2008. Data collected indicates that the combination of the client-server and peer-to-peer paradigms can be applied to music streaming with good results. In particular, 8.8% of music data played comes from Spotify's servers while the median playback latency is only 265 ms (including cached tracks). We also discuss the user access patterns observed and how the peer-to-peer network affects the access patterns as they reach the server.

I. Introduction

Spotify is a streaming music service using peer-to-peer techniques. The service has a library of over 8 million tracks, allowing users to freely choose tracks they wish to listen to and to seek within tracks. Data is streamed from both servers and a peer-to-peer network. The service launched in October 2008 and now has over 7 million users in six European countries.

The service is offered in two versions: a free version with advertisement, and a premium, pay-per-month, version. The premium version includes some extra features such as the option to stream music at a higher bitrate, and to synchronize playlists for offline usage. Both versions of the service allow unlimited streaming, and a large majority of users are on the free version. The music catalog is the same for both free and premium users with the exception of some pre-releases exclusive to premium users. However, due to licensing restrictions, the tracks accessible to a user depends on the user's home country.

One of the distinguishing features of the Spotify client is its low playback latency. The median latency to begin playback of a track is 265 ms. The service is not web-based, but instead uses a proprietary client and protocol.

A. Related Services

There are many different on-demand music streaming services offered today. To our knowledge, all such services but Spotify are web-based, using either Adobe Flash or a web browser plug-in for streaming. Furthermore, they are pure client-server applications without a peer-to-peer component. Among the more well-known such services are Napster, Rhapsody, and We7.

The application of peer-to-peer techniques to on-demand streaming is more prevalent when it comes to video-on-demand services. Such services include Joost, PPLive, and PPStream. These vary between supporting live streaming (usually of a large number of channels), video-on-demand access, or both. While there are many similarities between video-on-demand and music-on-demand streaming, there are also many differences; including user behavior, the size of streaming objects, and the number of objects offered for streaming.

A service offering on-demand streaming has many things in common with file-sharing applications. For instance, the mechanisms for locating peers in Spotify are similar to techniques from BitTorrent and Gnutella.

B. Related Work

Leveraging the scalability of peer-to-peer networks to perform media streaming is a well-studied area in the academic literature. Most such systems are concerned with live streaming, where viewers watch the same stream simultaneously. This setting is different in nature from the Spotify application, where a user has on-demand access to a large library of tracks.

The peer-to-peer live streaming literature can be roughly divided into two general approaches [1]: tree-based (e.g. [2]), and mesh-based (e.g. [3], [4]), depending on whether they maintain a tree structure in the overlay. While both techniques have their advantages, intuitively it seems that a mesh structure is a better fit for on-demand streaming applications.

There have been several studies measuring the performance and behavior of large peer-to-peer systems, describing and measuring both on-demand streaming [5], [6], and file-sharing [7], [8] protocols. We believe that there is high value in understanding how peer-to-peer techniques perform in today's networks.

Huang *et al.* [5] describe the PPLive video-on-demand streaming system, and also present measurements on its performance. To the best of our knowledge, their work is the only other detailed study of an on-demand streaming system of this size, with a peer-to-peer component. While there are naturally many similarities between PPLive and Spotify, there are also many differences, including the overlay structure and Spotify's focus on low latency techniques.

C. Our Contribution

In this paper we give an in-depth description and evaluation of Spotify. We discuss the general streaming protocol in Section II, and go into more details on the peer-to-peer parts in Section III.

Furthermore, in Section IV, we present and comment on data gathered by Spotify while operating its service. We give detailed measurements on many aspects of the service such as latency, stutter, and how much data is offloaded from the server by the peer-to-peer protocol. Some measurement data is also presented in Sections II and III; that data was collected as described in Section IV-A.

II. Spotify Overview

The Spotify protocol is a proprietary network protocol designed for streaming music. There are clients for OS X and Windows as well as for several smartphone platforms. The Windows version can also be run using Wine. The smartphone clients do not participate at all in the peer-to-peer protocol, but only stream from servers. Since the focus of this paper is the evaluation of peer-to-peer techniques we will ignore the smartphone clients in the remainder of this paper.

The clients are closed-source software available for free download, but to use a client, a Spotify user account is needed. Clients automatically update themselves, and only the most recent version is allowed to access the service.

The user interface is similar to those found in desktop mp3 players. Users can organize tracks into playlists which can be shared with others as links. Finding music is organized around two concepts: searching and browsing. A user can search for tracks, albums, or artists, and she can also browse—for instance, when clicking on an artist name, the user is presented with a page displaying all albums featuring that artist.

Audio streams are encoded using Ogg Vorbis with a default quality of q5, which has variable bitrate averaging roughly 160 kbps. Users with a premium subscription can choose (through a client setting) to instead receive Ogg Vorbis in q9 quality, averaging roughly 320 kbps. Both types of files are served from both servers and the peer-to-peer network. No re-encoding is done by peers, so a peer with the q9 version of a track cannot serve it to one wanting the q5 version.

When playing music, the Spotify client monitors the sound card buffers. If the buffers underrun, the client considers a *stutter* to have occurred. Stutters can be either due to network effects, or due to the client not receiving sufficient local resources to decode and decrypt data in a timely manner. Such local starvation is generally due to the client host doing other (predominantly I/O-intensive) tasks.

The protocol is designed to provide on-demand access to a large library of tracks and would be unsuitable for live broadcasts. For instance, a client cannot upload a track unless it has the whole track. The reason for this is that it simplifies the protocol, and removes the overhead involved with communicating what parts of a track a client has. The drawbacks are limited, as tracks are small.

While UDP is the most common transport protocol in streaming applications, Spotify instead uses TCP. Firstly, having a reliable transport protocol simplifies protocol design and implementation. Secondly, TCP is nice to the network in that TCP's congestion control is friendly to itself (and thus other applications using TCP), and the explicit connection signaling helps stateful firewalls. Thirdly, as streamed material is shared in the peer-to-peer network, the re-sending of lost packets is useful to the application.

Between a pair of hosts a single TCP connection is used, and the application protocol multiplexes messages over the connection. While a client is running, it keeps a TCP connection to a Spotify server. Application layer messages are buffered, and sorted by priority before being sent to the operating system's TCP buffers. For instance, messages needed to support interactive browsing are prioritized over bulk traffic.

A. Caching

Caching is important for two reasons. Firstly, it is common that users listen to the same track several times, and caching the track obviates the need for it to be re-downloaded. Secondly, cached music data can be served by the client in the peer-to-peer overlay. The cache can store partial tracks, so if a client only downloads a part of a track, that part will generally be cached. Cached content is encrypted and cannot be used by other players.

The default setting in the clients is for the maximum cache size to be at most 10% of free disk space (excluding the size of the cache itself), but at least 50 MB and at most 10 GB. The size can also be configured by the user to be between 1 and 100 GB, in 1 GB increments. This policy leads to most client installations having large caches (56% have a maximum size of 5 GB or more, and thus fit approximately 1000 tracks).

Cache eviction is done with a Least Recently Used (LRU) policy. Simulations using data on cache sizes and playback logs indicate that, as caches are large, the choice of cache eviction policy does not have a large effect on cache efficiency. This can be compared to the finding of Huang *et al.*[5] that the PPLive system gained much efficiency by changing from LRU to a more complex, weight-based evaluation process. However, in their setting, the objects in the cache are movies, and their caches are of a size such that a client can only cache one, or a few movies.

B. Random Access to a Track

The easiest case for a streaming music player is when tracks are played in a predictable order. Assuming sufficient bandwidth available, this allows the player to begin fetching data needed to play upcoming tracks ahead of time (*prefetching*). A more difficult, and perhaps interesting, case is when the user chooses a new track to be played, which we refer to as a *random access*. We begin by describing the random access case and then proceed to discuss prefetching in Section II-C.

Approximately 39% of playbacks in Spotify are by random access (the rest start because the current track finished, or because the user clicked the forward button to skip to the

next track). Unless the client had the data cached, it makes an initial request to the server asking for approximately 15 seconds of music, using the already open TCP connection. Simultaneously, it searches the peer-to-peer network for peers who can serve the track, as described in Section III-C.

In most cases the initial request can be quickly satisfied. As the client already has a TCP connection to the server no 3-way handshake is needed. Common TCP congestion avoidance algorithms, such as TCP New Reno [9] and CUBIC [10], maintain a congestion window limiting how large bursts of data can be sent. The congestion window starts out small for a new connection and then grows as data is successfully sent. Normally, the congestion window (on the server) for the long-lived connection between client and server will be large as data will have been sent over it. This allows the server to quickly send much, or all, of the response to the latency-critical initial request without waiting for ACKs from the client.

The connection to the server is long-lived, but it is also bursty in nature. For instance, if a user is streaming from a popular album, a very large fraction of the traffic will be peer-to-peer traffic, and the connection to the server will then be almost unused. If the user then makes a random access playback there is a sudden burst of traffic. RFC 5681 [11] states that an implementation should reduce its congestion window if it has not sent data in an interval exceeding the retransmission timeout. Linux kernels can be configured to disable that reduction, and when Spotify did so the average playback latency decreased by approximately 50 ms. We remark that this was purely a server-side configuration change.

If a user jumps into the middle of a track (*seeks*), the client treats the request similarly to a random access, immediately requesting data from the server as described above. The Ogg Vorbis format offers limited support for seeking in a streaming environment, so Spotify adds a custom header to all their files to better support seeking.

C. Predictable Track Selection

Most playbacks (61%) occur in a predictable sequence, i.e. because the previous track played to its end, or because the user pressed the forward button. Clients begin prefetching the next track before the currently playing track has been played to completion. With prefetching there is a trade-off between cost and benefit. If clients begin prefetching too late the prefetching may not be sufficient to allow the next track to immediately start playing. If they prefetch too early the bandwidth may be wasted if the user then makes a random request.

The clients begin searching the peer-to-peer network and start downloading the next track when 30 seconds or less remain of the current track. When 10 seconds or less remain of the current track, the client prefetches the beginning of the next track from the server if needed.

We did not have data to directly measure how good the choice of these parameters are. But, we can measure how often a user who listens to a track for the duration of the track minus t seconds continues to the next track. If a user seeks within a track, the measured event does not correspond to playback coming within t seconds of the end of the track. For $t = 10, 30$, the next scheduled track was played in 94%, and 92% of cases, respectively. This indicates that the choice of parameters is reasonable, possibly a bit conservative.

During a period of a few weeks, all clients had a bug where prefetching of the next track was accidentally disabled. This allows us to directly measure the effects of prefetching on the performance of the system. During a week when prefetching was disabled the median playback latency was 390 ms, compared with the median latency over the current measurement period of 265 ms. Furthermore, the fraction of track playbacks in which stutter occurred was 1.8%, compared to the normal rate of 1.0%.

D. Regular Streaming

While streaming, clients avoid downloading data from the server unless it is necessary to maintain playback quality or keep down latency. As discussed in Section II-B, when the user makes a random access, an initial request for data is sent to the server. Clients make local decisions about where to stream from depending on the amount of data in their play-out buffers. The connection to the server is assumed to be more reliable than peer-connections, so if a client's buffer levels are low, it requests data from the server. As long as the client's buffers are sufficiently full and there are peers to stream from, the client only streams from the peer-to-peer network.

There is also an "emergency mode" where, if buffers become critically low (less than 3 seconds of audio buffered during playback), the client pauses uploading data to its peers. The reason for this is that many home users have asymmetric connection capacity, a situation where ACK compression can occur and cause degradation of TCP throughput [12]. The "emergency mode" has been in the protocol since the first deployment, so we have not been able to evaluate its effects.

A given track can be simultaneously downloaded from the server and several different peers. If a peer is too slow in satisfying a request, the request is resent to another peer or, if getting the data has become too urgent, to the server.

While streaming from a server, clients throttle their requests such that they do not get more than approximately 15 seconds ahead of the current playback point, if there are peers available for the track. When downloading from the peer-to-peer network, no such throttling occurs and the client attempts to download the entire currently playing track. If a user changes tracks, requests relating to the current track are aborted.

Files served within the peer-to-peer network are split into chunks of 16 kB. When determining which peers to request chunks from, the client sorts peers by their expected download times (computed as the number of bytes of outstanding requests from the peer, divided by the average download speed received from that peer) and greedily requests the most urgent chunk from the peer with the lowest estimated download time (and then updates the expected download times). This means that chunks of a track are requested in sequential order. As all peers serving a file have the entire file, requesting blocks in-order does not affect availability or download speeds.

A client can at most have outstanding requests from a given peer for data it believes the peer can deliver within 2 seconds. An exception to this is that it is always allowed to have requests for 32 kB outstanding from a peer. If the estimated download time for a block exceeds the point in time at which the block is needed, that block is not requested.

E. Play-out Delay

Streaming applications need to employ some mechanism to combat the effects of packet loss and packet delay variation. Several different options have been suggested in the literature; for a survey see [13]. Spotify clients do not drop any frames or slow down the playout-rate, and are thus *non delay-preserving* in the nomenclature of [13]. As TCP is used as transport protocol, all data requested will be delivered to the application in-order, but the rate at which data is delivered is significantly affected by network conditions such as packet loss. If a buffer underrun occurs in a track, the Spotify client pauses playback at that point, re-performing latency adjustment.

As discussed by Liang *et al.*[14], there is a trade-off between initial playback latency, receiver buffer size, and the stutter free probability. Spotify clients do not limit the buffer size, and thus the crux of the problem is the appropriate modeling of the channel and using that information to adjust the initial playback latency. As a simplification, the client only considers the channel to the server for latency adjustment.

Spotify clients use a Markovian model for throughput as observed by the client (i.e., affected by packed delay variation, packet loss, and TCP congestion control). Clients make observations of throughput achieved while it is downloading from the server to estimate a Markov chain. Only data collected during the last 15 minutes of downloading is kept and used. The states of the Markov chain is the throughput during 1 second, discretized to 33 distinct levels between 0 and 153 kBps (more granular at lower throughputs).

The model is not used to compute an explicit playback latency. Instead, before playback has commenced, the client periodically uses the Markov chain to simulate the playback of the track, beginning with the current amount of buffered data, and the current data throughput. Each such simulation is considered as failing or passing, depending on if an underrun occurred or not. The client performs 100 simulations and if more than one of them fails, it waits longer before beginning playback. During these simulations the client makes the simplifying assumption that data is consumed at a constant rate despite the fact that the codec used has a variable bitrate encoding.

III. Spotify's Peer-to-Peer Network

Spotify's protocol has been designed to combine server- and peer-to-peer streaming. The primary reason for developing a peer-to-peer based protocol was to improve the scalability of the service by decreasing the load on Spotify's servers and bandwidth resources. An explicit design goal was that the usage of a peer-to-peer network should not decrease the performance in terms of playback latency for music or the amount of stutter. While that design goal is addressed by the reliance on a server for latency-critical parts, it puts demands on the efficiency of the peer-to-peer network in order to achieve good offloading properties.

We discussed in Sections II-B and II-D how the clients combine streaming from the peers and servers. In this section, we give an overview of the peer-to-peer network.

A. General Structure

The peer-to-peer overlay used is an unstructured network, the construction and maintenance of which is assisted by trackers. This allows all peers to participate in the network as equals so there are no "supernodes" performing any special network-maintenance functions. A client will connect to a new peer only when it wishes to download a track it thinks the peer has. It locates peers likely to have a track it is looking for through the mechanisms described in Section III-C.

As discussed in Section II-A, clients store (relatively large) local caches of the tracks they have downloaded. The content of these caches are also what the clients offer to serve to their peers. As tracks are typically small and as live streaming is not supported, a simplification made in the protocol is that a client only offers to serve tracks which it has completely cached. This allows for a slightly simpler protocol, and keeps the protocol overhead down.

There is no general routing performed in the overlay network, so two peers wishing to exchange data must be directly connected. There is a single message forwarded on the behalf of other peers, which is a message searching for peers with a specific track. The rationale for the lack of routing in the overlay is to keep the protocol simple and keep download latencies and overhead down.

B. A Split Overlay Network

The service is currently run from two data centers, one in London and one in Stockholm. A peer uniformly randomly selects which data center to connect to, and load is evenly spread over both data centers. Each data center has an independent peer-to-peer overlay. Thus, the peer-to-peer overlay is in fact split into two overlays, one per site.

The split is not complete since if a client loses its connection to the server, it reconnects to a new server. If it reconnected to the other site it keeps its old peers but is unable to make any new connections to peers connected to servers at its old site. For simplicity of presentation, we will describe the protocol as having a single overlay network.

C. Locating Peers

Two mechanisms are used to locate peers having content the client is interested in. The first uses a tracker deployed in the Spotify back-end, and the second a query in the overlay network.

The problem of locating peers is somewhat different in music-on-demand streaming compared to many other settings. As tracks are small, a client generally only needs to find one, or a few peers to stream a track from. However, as

tracks are also short in duration, downloading new tracks is a very frequent operation, and it is important to minimize the overhead. Furthermore, the lookup time becomes a big issue, which is one of the reasons for Spotify not using a Distributed Hash Table (DHT) to find peers. Other reasons for not implementing a DHT include keeping the protocol simple and keeping overhead down.

The functionality of the tracker is similar, but not identical, to that of a tracker in the BitTorrent protocol [15]. It maintains a mapping from tracks to peers who have recently reported that they have the track. As a peer only offers to serve a track if it has the whole track cached, peers listed in the tracker have the whole track.

As two complementary mechanisms are used, the tracker can be simplified compared to many other system. In particular, the tracker only keeps a list of the 20 most recent peers for each track. Furthermore, clients are only added to the tracker when they play a track and do not periodically report the contents of their caches, or explicitly notify the tracker when content is removed. This helps in keeping overhead down, and simplifies the implementation of the tracker. As clients keep a TCP connection open to a Spotify server, the tracker knows which clients are currently online. When a client asks the trackers for peers who have a track, the tracker replies with up to 10 peers who are currently online. The response is limited in size to minimize overhead.

In addition to the tracker-based peer searches, clients also send search requests in the overlay network, similar to the method used in Gnutella [16]. A client in search of a track sends a search request to all its neighbors in the overlay, who forward the request to all their neighbors. Thus, all peers within distance two of the searcher in the overlay see the request, and send a response back if they have the track cached. Search queries sent by clients have a query id associated with them, and peers remember the 50 most recent searches seen, allowing them to ignore duplicate messages. This limited message forwarding is the only overlay routing in the Spotify peer-to-peer protocol.

When a client is started, how does it get connected to the peer-to-peer network? If it was still listed in the tracker for some tracks then it is possible that other clients will connect to it asking for those tracks. If the user starts streaming a track, it will search the peer-to-peer network and connect to peers who have the track, thus becoming a part of the overlay.

D. Neighbor Selection

Keeping the state required to maintain a large number of TCP connections to peers is expensive, in particular for home routers acting as stateful firewall and Network Address Translation (NAT) devices. Thus, each client has a maximum number of peers it may be connected to at any given time. Clients are configured with both a soft and a hard limit, and never go above the hard limit. The client does not make new connections above the soft limit and periodically prunes its connections to keep itself below the soft limit (with some headroom for new connections). These limits are set to 50 and 60, respectively.

When a client needs to disconnect one or more peers, it performs a heuristic evaluation of the utility of each connected peer. The intention is for the heuristic to take into account both how useful the connection is to the evaluating peer, as well as how useful the link is to the overlay as a whole.

The client sorts all its connected peers according to 6 criteria: bytes sent in the last 10 minutes, bytes sent in the last 60 minutes, bytes received in the last 10 minutes, bytes received in the last 60 minutes, the number of peers found through searches sent over the connection in the last 60 minutes, and the number of tracks the peer has that the client has been interested in in the last 10 minutes. For each criterion, the top scoring peer in that criterion gets a number of points, the second peer a slightly lower number, and so on (with slightly different weights for the different criteria). Peers with a raw score of 0 for a criterion do not get any points for that criterion. The peers points are then summed over all the criteria, and the peers with the least total scores are disconnected.

The client simultaneously uploads to at most 4 peers. This stems from the fact that TCP congestion control gives fairness between TCP connections, so many simultaneous uploads can have adverse effects on other internet usage, in particular for a home user with small uplink bandwidth.

E. State Exchanged Between Peers

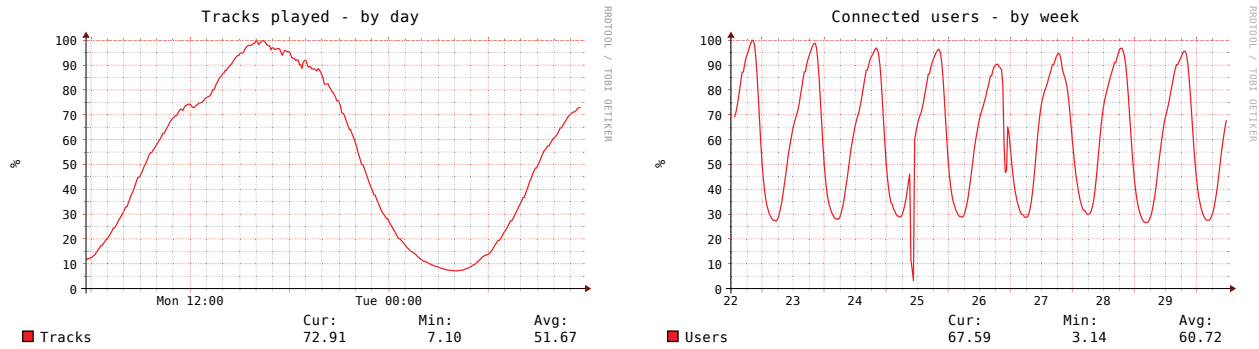
A client wanting to download a track will inform its neighbors of its interest in that track. The interest notification also contains a priority, where the client informs its peer of the urgency of the request. Currently, three discrete levels (in falling order of priority) are used: currently streaming track, prefetching next track, and offline synchronization.

A serving client selects which peers to service requests from by sorting them by the priority of the request, and previously measured upload speed to that peer, and then offer to service the requests of the top 4 peers. Peers are informed whenever their status changes (if their requests become sufficiently prioritized to be serviced, or if their requests no longer are).

F. NAT Traversal

All traffic in the peer-to-peer network uses TCP as transport protocol so the most common protocols for NAT traversal, e.g. STUN [17], are not immediately applicable as they are designed for UDP. While there are techniques for performing TCP NAT traversal as well [18], Spotify clients currently do not perform any NAT traversal.

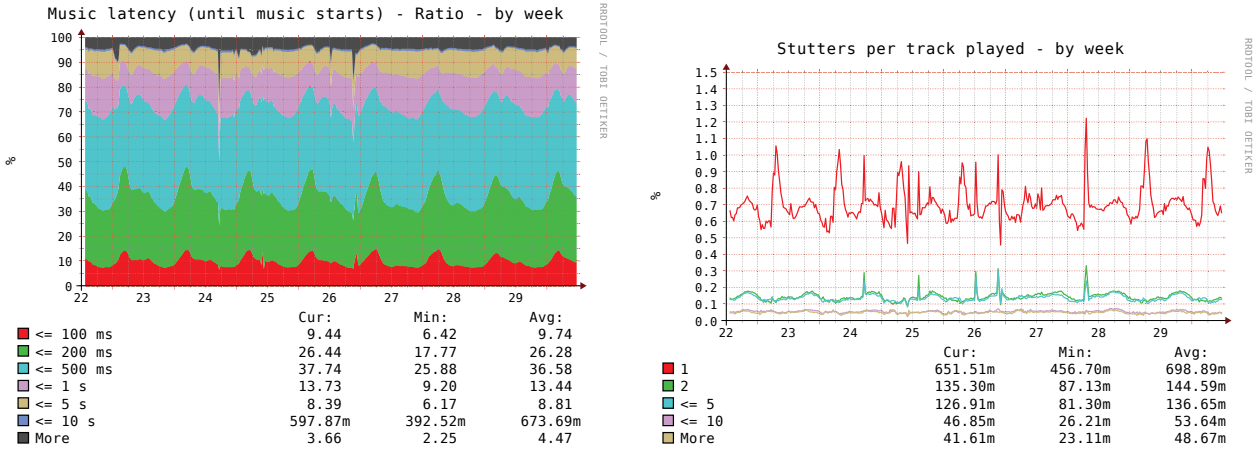
This lack of NAT traversal is mitigated by two factors. Firstly, when a client wishes to connect to a peer a request is also forwarded through the Spotify server asking the connectee to attempt a TCP connection back to the connecter. This allows the connection to be established provided one of the parties can accept incoming connections. Secondly, clients use the Universal Plug n' Play (UPnP) protocol to ask home routers for a port to use for accepting incoming connections.



(a) Tracks played

(b) Users connected

Figure 1. The weekly usage pattern of the Spotify service. Data has been normalized to a 0-1 scale.



(a) Playback latency

(b) Stutter during playback

Figure 3. Playback latency and music stutter over a week.

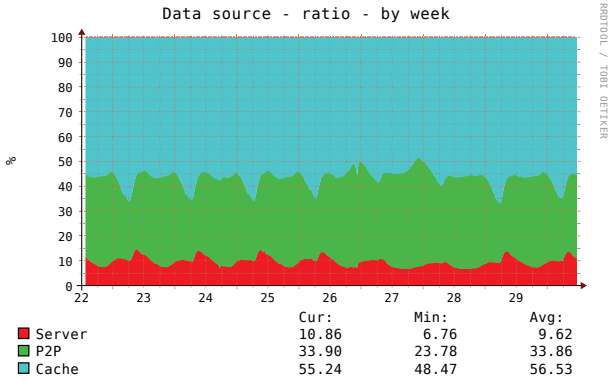


Figure 2. Sources of data used by clients

IV. Peer-to-Peer Evaluation

In this section, we will present and discuss measurements indicating the performance of the Spotify system with focus on the peer-to-peer network performance. For business reasons, some data is presented as ratios rather than absolute volumes.

A. Measurement Methodology

Both Spotify clients and servers perform continuous instrumentation and monitoring of the system. Most client measurements are aggregated locally before being sent to the server. For instance, reports on connection statistics are sent every 30 minutes to the server.

The raw log messages are collected and stored on log servers and in a Hadoop cluster (an open-source map-reduce and distributed storage implementation), where they are available for processing. There is also a real-time monitoring system, based on the open-source Munin monitoring system, storing aggregated data and generating graphs based on the log messages and instrumentation of Spotify servers. Most of our graphs are based on the aggregated Munin databases, while most aggregate statistics (e.g., median playback latency) were computed from the raw log files.

In the graphs presented in this paper, min and avg gives the mimimum and average (per time unit) values taken over the measurement period, and cur denotes the current value when the measurement was made. Values below 1 will be denoted with units m or u, denoting milli (10^{-3}) and micro (10^{-6}), respectively.

We collected log messages and monitoring databases for a measurement period of the week between Tuesday 23 March and Monday 29 March 2010 (inclusive). During the measurement period there was a brief disruption (lasting approximately 1.5 hours) in the service during the evening of Friday March 26th. Times are given in UTC.

B. Periodic Variations

When looking at measurements of the Spotify network it quickly becomes apparent that there are significant periodic effects. As users are located in Western Europe, the effect of nighttime is clearly visible. We remark that usage is high throughout the workday, as Spotify is legal and music-listening can be done while working.

There is also a clear weekly pattern with a distinct difference in the shape of data between weekdays and weekends. We believe these effects can likely be attributed to both a difference in user behavior and in the difference of computer setup and network architecture between corporate and home networks. At home, we would expect most users to have machines with much free hard disk space, connected to the Internet through a NAT device speaking UPnP. At work, we would expect a larger variation with many users being behind more restrictive firewalls. In Figure 1, we show the variation over a week in the number of users connected, and the number of tracks played, where the data was normalized to a 0-1 scale. As the two curves have almost the same shape we use different time frames for the two graphs, and show tracks played during a single day in Figure 1(a), and users connected over a week in Figure 1(b). The dip during the early morning hours of March 25th appears to be due to an error in the collection of the data, while the dip during the 26th is due to the outage mentioned above.

C. Data Sources

We now turn to the question of how effectively the servers are offloaded. Figure 2 shows the sources of track data for the client and its variation throughout a week. All (non-duplicate) data downloaded from servers and peers is counted, even if it is not played (due to the user skipping the track). Data from the cache is counted when the client reads the data from its cache for playback, meaning that it corresponds reasonably closely, but not exactly, to cached data played by the client.

Some periodic effects are clearly visible. During nighttime, a significantly larger fraction of data played comes from the cache. This effect is less pronounced on weekend nights, when users play more new music. There is a distinct decrease in the fraction offloaded by the peer-to-peer overlay while the logged in population is increasing during the morning, and there is a lesser, but more prolonged decrease when users log off in the evening. There is also a difference between weekdays and weekends, with peer-to-peer data offloading a larger fraction of data during the weekends.

In total, during the measurement period, 8.8% of data came from servers, 35.8% from the peer-to-peer network, and the remaining 55.4% were cached data. Thus, we can see that the large caches and peer-to-peer network together significantly decrease the load on Spotify's servers. Our results are comparable to the measurements performed by Huang *et al.* [5] where PPLive was shown to have 8.3% data downloaded from servers.

D. Playback Latency and Stutter

Two important measures for on demand streaming services are the playback latency and the stutter frequency. As discussed by Liang *et al.* [14], streaming applications must make a trade-off between the two quantities. We are not aware of any studies discussing in detail how user's satisfaction with a streaming service depends on these two factors, but we believe them to be very important. We think that studying the impact of streaming performance on user satisfaction would be an interesting subject for future research.

When combining peer-to-peer and server streaming, there is also a trade-off between server load and latency and stutter frequencies. Spotify has designed to prioritize the latter two.

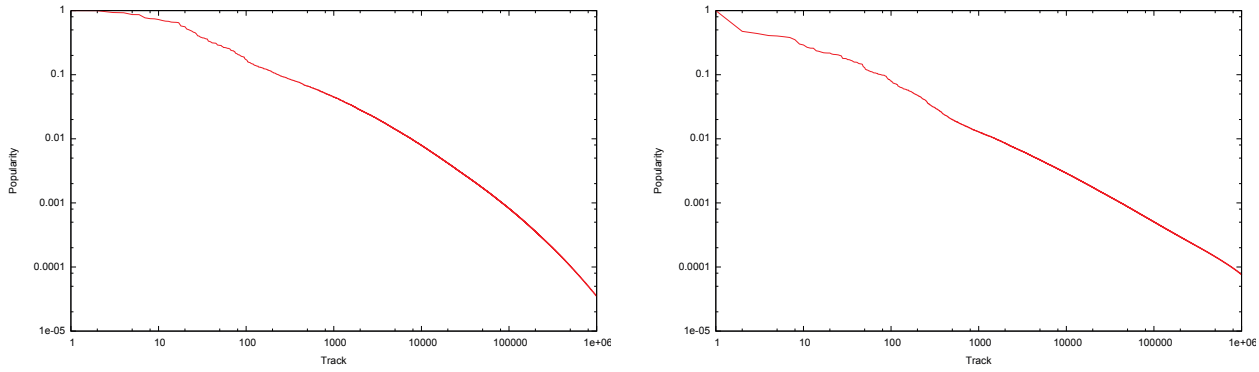
The Spotify client measures the playback latency as the time between when a track should start, either due to user action or due to the previous track ending, and the time the OS is instructed to begin playback. Thus, the latency measured not only includes network latency but also the time needed on the client's host to decrypt and decompress downloaded data. We remark that, due to digital rights management (DRM), even fully cached material must wait for a reply over the network before playback can commence and such a network request is sent when the track should start, unless the track has been synchronized for offline playback. Thus, the latency almost always includes at least one RTT to the Spotify servers.

In Figure 3(a), we show how playback latency varied over the measurement period. The large fraction of playbacks shown as having a latency of more than 10 seconds was discovered to be due to an error in latency reporting, the true value is significantly lower, and was below 1% throughout the measurement period.

We observe that the latency is significantly lower during the night, when the number of tracks played is also low. This is mainly due to two reasons. Firstly, more data is played from the cache during the night. Secondly, users are less interactive during the night, allowing the prefetching discussed in Section II-B to be much more efficient.

Throughout the measurement period the median latency was 265 ms, the 75th percentile was 515 ms, and the 90th percentile was 1047 ms. As over half the data during the measurement period came from the cache, it is likely that the median playback latency measures the time needed for DRM together with local processing time.

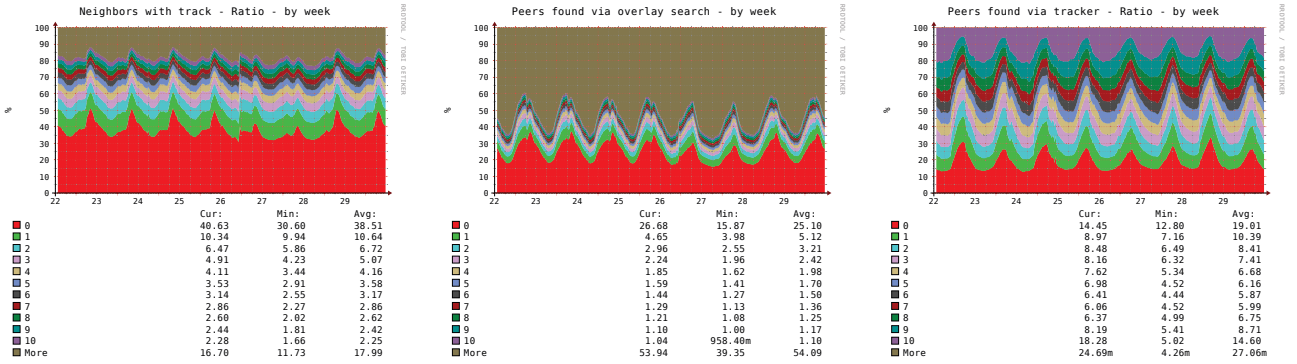
Intimately related to the playback latency is the amount of stutter occurring upon playback. In Figure 3(b), we show the fraction of playbacks during which one or more stutters occurred. During the measurement period, less than 1% of all playbacks stuttered. However, in 64% of stutters, the client had more than 20 kilobytes of compressed music data available. This indicates that stutters are often due to local effects (e.g.,



(a) Track playback frequencies (normalized), log-log scale

(b) Track server request frequencies (normalized), log-log scale

Figure 4. Frequency of track accesses, both as played by clients and as blocks requested from the Spotify servers



(a) Neighbors who had track

(b) Peers with track found in overlay

(c) Peers returned by tracker

Figure 5. Peers found through broadcasting searches in the overlay and through the tracker

not receiving sufficient CPU time), rather than the network. While we think a rate of 1% is acceptably low, there seems to be room for further improvement.

E. Distribution of Track Accesses

There has been a considerable amount of discussion on how an on-demand system affects usage patterns [19], [20], [21]. In particular, it has been discussed whether all-you-can-eat on-demand access leads users to access content according to a long-tail distribution. A full discussion of the access pattern of music in the Spotify service is out of scope for this paper, but we will briefly mention some findings on the access pattern.

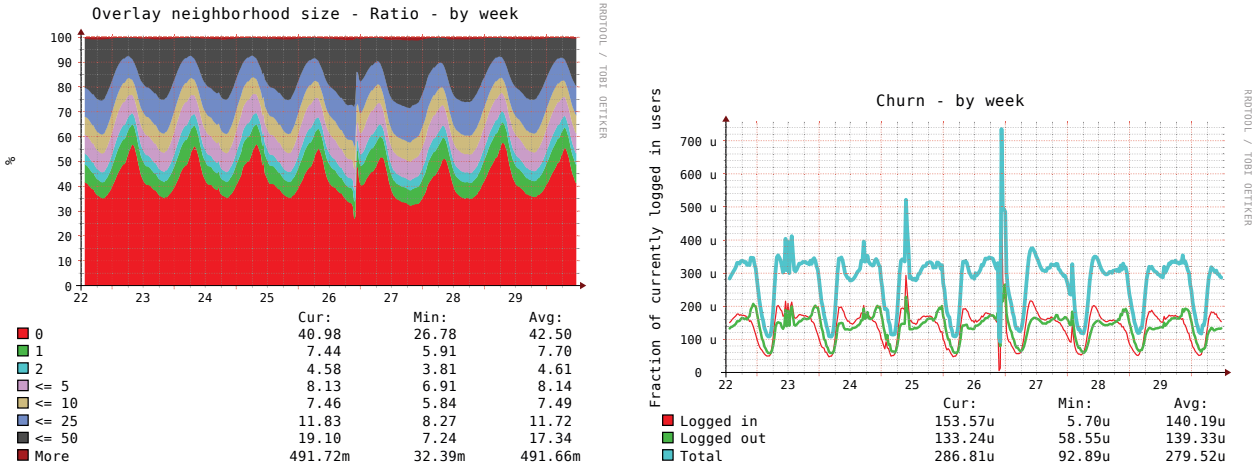
In Figure 4(a), we show the shape of the probability density function of track playbacks in the Spotify system, normalized so the most popular item has frequency 1. During the measurement period 88% of the track accesses were within the most popular 12% of the library. While much weight is on the most popular tracks, it turns out that a large fraction of the catalog is played. During our week-long measurement period, approximately 60% of the content available was accessed at least once.

Given the distribution of track accesses one could wonder how well the peer-to-peer network works at varying degrees of popularity. In Figure 4(b), we show the shape of frequencies of tracks as requested from the Spotify servers. The graph is based on data collected during February 21–23 2010, which is outside our measurement period. This graph gives the relative frequencies of tracks requested at servers. Clients normally make between zero and ten server requests for a track, depending on how much of it they download from a server. The size of blocks requested varies, with the mean size being 440 kB. The distribution of requests to the servers is significantly less top-heavy than that of track playbacks with 79% of accesses being within the most popular 21%.

F. Locating Peers

Spotify provides two separate mechanisms for locating peers having a piece of content, as described in Section III-C. In Figure 5, we show diagrams indicating how well the mechanisms work. We have divided the neighborhood search into two types of responses, peers who were already neighbors (Figure 5(a)), and new peers found who were at distance 2 (Figure 5(b)). We observe that locating peers through the overlay appears to be essentially binary in nature—most requests receive either none or many replies. In contrast, the number of peers located through the tracker mechanism is more varied.

The fact that a client so often finds that many of its overlay neighbors have the track it is interested in indicates that a certain amount of clustering by interest is occurring in the overlay. We believe it would be an interesting topic for future



(a) Neighborhood size

(b) Churn

Figure 6. Overlay properties — sizes of peer's neighborhoods, and churn (fraction of currently connected clients connecting/disconnecting per second)

Table I
SOURCES OF PEERS

Sources for peers	Fraction of searches
Tracker and P2P	75.1%
Only Tracker	9.0%
Only P2P	7.0%
No peers found	8.9%

Table II
DISTRIBUTION OF APPLICATION-LAYER TRAFFIC IN OVERLAY NETWORK

Type	Fraction
Music data, used	94.80%
Music data, unused	2.38%
Search overhead	2.33%
Other overhead	0.48%

work to further explore the extent of such clustering, and if it can be further enhanced by the protocol design, or if it can be leveraged in other peer-to-peer systems.

Recall that the tracker mechanism is necessary for bootstrapping a newly joined client into the overlay network (as without a tracker, she would never get connected to anyone and could not use searches in the overlay to find new peers). Referring to Figure 5(c), it seems as if once the bootstrapping was done it is possible that the searching within the overlay could be a sufficiently effective mechanism on its own.

When designing the two complementary mechanisms, a reason for having both mechanisms was that the tracker would allow for "long jumps" within the overlay. Assuming that some amount of clustering is occurring, if a user's mood changes and she now feels like listening to Jazz instead of the Hip Hop she has been listening to for the last hour, intuitively it seems that searching locally in her overlay would not yield much, if anything. A reason for having overlay searches given that there is a tracker is to make the system more robust to tracker failures, and to allow for a simpler and cheaper tracker implementation.

We analyzed the traces for those tracks where the client had downloaded at least one byte. Thus, tracks played from the cache, or skipped before the client received any data were ignored in this analysis. The results of a tracker query and a neighborhood search can overlap, and a client may learn of a given peer from both sources simultaneously. Such peers are reported by the client as having been found in the peer-to-peer network (only).

The results are summarized in Table I. We note that, for a majority of requests, both mechanisms are successful in locating new peers. We believe that the relatively high frequency (7%) of the event that peers are found only through the overlay is mostly an artifact of the reporting of overlapping search results.

G. Peer-to-Peer Protocol Overhead

An issue in all peer-to-peer networks is to keep the overhead cost of the overlay at a minimum. A potential problem in Gnutella-like systems in particular is the overhead incurred by searches in the peer-to-peer network.

Clients periodically report the total number of bytes received on peer-to-peer TCP sockets as well as the total number of bytes of useful song data they have downloaded over the peer-to-peer network. Song data is considered useful if the request had not been cancelled before the data arrived, and if it is not duplicate to data the client has received from other peers or the server. Our measurements do not include overhead caused by lower-layer protocols such as TCP and IP.

Over our measurement period, 5.20% of the traffic received from the peer-to-peer network was not useful song data. In Table II we break down the overhead into three categories. Most of the overhead, 2.38% of received traffic, comes form song data downloaded which was not useful to the client. Closely following is the overhead coming from searching for peers. The measurements on protocol overhead can be compared with experimental measurements on the BitTorrent protocol by Legout *et al.* [7] where they found the BitTorrent protocol

overhead to be, in most of their measurements, below 2%. We remark that the two protocols are quite different in nature, and there is a difference in the measurement methodology.

H. Properties of the Overlay Network

As discussed in Section III-D, each peer has a limit to the number of neighbors it retains connections to in the overlay network. In Figure 6(a), we show what the distribution of node degrees in the overlay looks like. A surprisingly large fraction (between 30%-50%) of clients are disconnected from the overlay. While we have not fully investigated the reasons for this, we believe a partial explanation may be that the neighbor selection and tracker mechanisms cause idle users to become disconnected from the overlay.

In Section III-F, we discussed the absence of NAT traversal techniques in the Spotify clients. Intuitively, one would expect that this would constitute a problem as users are likely to be situated behind various NAT devices. Our data shows that it indeed is a problem and that the fraction of successful connection attempts was 35% during our measurement period.

An important factor for how well a peer-to-peer network can function is the amount of churn. While we do not have any direct measurements of churn, we do have access to data showing the fraction of currently connected clients connecting to and disconnecting from servers, shown in Figure 6(b). It can happen that a client reconnects to a new Spotify server while still being active in the peer-to-peer network, e.g. in case the server is overloaded. This should be sufficiently rare that Figure 6(b) gives a realistic measurement of the churn in the overlay.

We note that the total churn rate is roughly even throughout the day with a sharp decrease during nighttime. During the morning it is dominated by logins while it is dominated by logouts towards the evening. Comparing the churn to Figure 2 we observe that, seemingly, the efficiency of the overlay in data delivery is not severely impacted by clients logging out. On the other hand, there is a daily dip in the fraction of data delivered by the overlay during the morning when many new users are logging on.

V. Conclusion

We have given an overview of the protocol and structure of the Spotify on-demand music streaming service, together with many measurements of the performance of the system. In particular, we note that the approach that Spotify uses to combine server-based and peer-to-peer streaming gives very good results, both with respect to user-relevant performance measures, and in reducing server costs. Furthermore, this is done using TCP as a transport protocol, indicating that streaming over TCP is a very viable option. The data collected shows also shows that a simplified tracker coupled with overlay broadcasts can be an efficient design for locating peers.

We believe on-demand streaming will continue to grow rapidly in the coming years, and that many interesting problems remain in further developing such services. Among these are (1) development of user satisfaction measures for on-demand streaming; (2) improved playout strategies, adapted to peer-to-peer data delivery; (3) efficient peer-to-peer overlays exploiting the overlap in interests between users.

Acknowledgements

We would like to express our gratitude to everyone working at Spotify for their help. We would like to thank Sonja Buchegger, Pehr Söderman, and Cenny Wenner for reading and commenting on draft versions.

References

[1] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live P2P streaming approaches," in *Proc. of IEEE INFOCOM'07*, 2007, pp. 1424–1432.

[2] H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming live media over peers," Stanford InfoLab, Technical Report 2002-21, 2002. [Online]. Available: http://ilpubs.stanford.edu:8090/863/

[3] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming," in *Proc. of IEEE INFOCOM'05*, 2005, pp. 2102 – 2111.

[4] X. Liao, H. Jin, Y. Liu, L. M. Ni, and D. Deng, "AnySee: Peer-to-peer live streaming," in *Proc. of IEEE INFOCOM'06*, 2006, pp. 1–10.

[5] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang, "Challenges, design and analysis of a large-scale P2P-VoD system," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 375–388, 2008.

[6] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross, "A measurement study of a large-scale P2P IPTV system," *Multimedia, IEEE Transactions on*, vol. 9, no. 8, pp. 1672–1687, dec. 2007.

[7] A. Legout, G. Urvoy Keller, and P. Michiardi, "Understanding BitTorrent: An experimental perspective," Technical Report, 2005. [Online]. Available: http://hal.inria.fr/inria-00000156/en/

[8] S. Saroiu, K. P. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Multimedia Computing and Networking (MMCN)*, January 2002.

[9] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782 (Proposed Standard), Internet Engineering Task Force, Apr. 2004.

[10] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[11] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681 (Draft Standard), Internet Engineering Task Force, Sep. 2009.

[12] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan, "Improving TCP throughput over two-way asymmetric links: analysis and solutions," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 78–89, 1998.

[13] N. Laoutaris and I. Stavrakakis, "Intrastream synchronization for continuous media streams: a survey of playout schedulers," *Network, IEEE*, vol. 16, no. 3, pp. 30–40, may/jun 2002.

[14] G. Liang and B. Liang, "Jitter-free probability bounds for video streaming over random VBR channel," in *Proc. of QShine'06*. New York, NY, USA: ACM, 2006, p. 6.

[15] B. Cohen, "The bittorrent protocol specification," BEP 3 (Standard), 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[16] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," *Proc. of IEEE P2P'01*, pp. 99–100, 2001.

[17] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008.

[18] S. Perreault and J. Rosenberg, "TCP candidates with interactive connectivity establishment (ICE)," Internet-Draft (work in progress), draft-ietf-mmusic-ice-tcp-08, Oct. 2009. [Online]. Available: http://tools.ietf.org/html/draft-ietf-mmusic-ice-tcp-08

[19] C. Anderson, *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, July 2006. [Online]. Available: http://www.worldcat.org/isbn/1401302378

[20] A. Elberse, "Should you invest in the long tail?" *Harvard Business Review*, vol. 86, no. 7/8, pp. 88–96, 2008.

[21] S. Goel, A. Broder, E. Gabrilovich, and B. Pang, "Anatomy of the long tail: ordinary people with extraordinary tastes," in *WSDM '10: Proceedings of the third ACM international conference on Web search and data mining*. New York, NY, USA: ACM, 2010, pp. 201–210.