Comparing Hybrid Peer-To-Peer Networks with Pure Peer-To-Peer Networks: Midterm Update

In this update, I provide a summary of my research on Spotify's hybrid peer-to-peer and client server architecture. Future updates will focus on drawing comparisons with other case studies.

Introduction

Spotify is a music-on-demand streaming service that has gained a lot of popularity since it launched in October 2008. During that time there were many different streaming services like Rhapsody and We7. All but Spotify were web-based, using either Adobe Flash or a web browser plug-in for streaming. Further, they all employed purely a client server architecture. Instead, Spotify used a combined server-client and peer-to-peer network architecture to provide a low-latency service for its users while saving money on server bandwidth. Leveraging the scalability of peer-to-peer networks allowed Spotify to provide a library of over 8 million tracks, to over 7 million users in six European countries. Since then, Spotify has grown quite a lot, and has moved away from this hybrid model in favor for a more traditional client-server architecture as they were able to afford enough coverage with their servers.

Introduction to Spotify's Protocol

The protocol is designed to provide on-demand access to a large library of tracks and has been designed to keep the protocol as simple as possible. This in turn, ensures that overhead is low. For instance, a peer in this network cannot share a file unless it has the whole track. This removes any overhead involved with communicating what parts of a track a client has. Spotify's protocol also uses TCP instead of UDP. The reasoning for this is three-fold:

- TCP’s congestion control is friendly to itself (and hence other applications using TCP)
- Having a reliable transport protocol simplifies protocol design.
- As streamed material will be shared in the peer-to-peer network, lost packets need to be resent.

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. Cache eviction is done with a Least Recently Used (LRU) policy.

How Spotify Handles Streaming a Random Track

The easiest case for a streaming music player is when tracks are played in a pre-determined order. This allows the player to begin fetching data needed to play upcoming tracks before the current track has finished playing.

A more difficult case is when the user chooses a new track to be played (referred to as “random access”). Approximately 39% of playback in Spotify are by random access. The following will detail how Spotify handles random access. The client will first check its cached data if it already has the track downloaded. If not, it makes an initial request to the server asking for approximately 15 seconds of music. At the same time, it searches the peer-to-peer network for peers who can serve the track. In most cases, the initial request can be quickly satisfied by the server. The client already has an open TCP connection to the server so no 3 way handshake is required.

How Spotify Handles Streaming a Predictable Track

Most playback occurs in predictable sequence: i.e. the current tracks plays to the end and the next track start playing or the user hits the next track button and the next track starts playing. To handle this the Spotify client begins 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 it could not find a peer to serve the track.

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 the P2P protocol was to reduce the load on Spotify's servers when scaling their service.

An important goal for this protocol was to maintain performance in terms of playback latency for music. While the design goal is addressed by relying on the server for latency-critical tasks, it requires the peer-to-peer network to be efficient in order to effectively share some of the load of the server.

The P2P overlay used is an unstructured network- maintained and created by trackers. There are no "super nodes" performing any special maintenance functions and all peers are treated as equals. A client will only connect to a peer when it wants to download a track that it thinks the peer has. The mechanism used to locate peers are described later in this paper.

Clients store (relatively) large caches of downloaded tracks. As tracks are relatively small, a simplification is made to the protocol to keep overhead low: tracks are only shared by the peer if the peer has the whole track.

There is no general routing performed in the overlay network, so two peers wishing to exchange data must be directly connected. The rationale for the lack of routing in the overlay is to keep the protocol simple and keep download latencies and overhead down.

Future Work

- Finish research on how Spotify’s protocol locates peers. Start write up for the same
- Compare some of the design decisions made in Spotify’s application with that of Gnutella and BitTorrent
- Start research on making a hybrid p2p chat application, using some of the lessons learnt from the comparison.