MusicNode: A Music File and Playback Synchronization Tool

Clint Riley - clintr@stanford.edu

Introduction

There are a number of reasons for wanting to share music across devices. The ability to have access to files that originate on different devices without explicitly issuing a copy command for them can be very convenient. With the ubiquity of Internet connections that are a reasonable speed, this can not only be done over the local network, but over a wide area network as well.

MusicNode is a system for allowing sharing of files across devices and synchronizing playback.

Features

File Sharing

Nodes can specify directories that contain files that will be made available to any node joining the group. No further user action is required to share a file.

Synchronized Playback

Nodes can request synchronized playback with all nodes that are currently active in the group. When a node receives a synchronized playback request from the master, it will download the file if required and then attempt to play the file, synchronized with all the other nodes.

Deduplication

Files are logically split into chunks when they are discovered by the system. These chunks are then fingerprinted and added to the file's metadata in the database. When a node requests a file, it will only request the chunks that are not already present on its local file system.

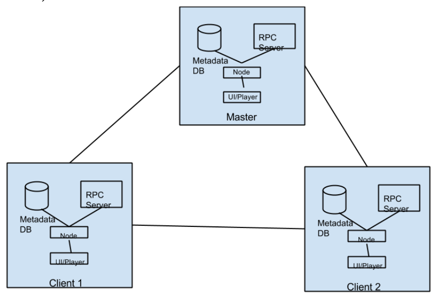
Design

Each MusicNode consists of a database, which is responsible for holding data about the files the node is currently tracking (both remote and local), an RPC server, a simple audio player, and a user interface.

Each node is designated as either a master or a client. The master node is responsible for maintaining the authoritative central database for files, storing various metadata that will allow for it or client nodes to retrieve the files in their entirety. Additionally, the master tracks currently connected clients to sync playback.

The database that each node maintains information about files as well as chunks. A file entity holds relevant metadata for a file (e.g. title, artist, available locally or not, chunks making up the file). A chunk entity holds metadata for the chunk (e.g. fingerprint, size in bytes, where it can be found locally). In most cases, a chunk is only a logical entity and not a ”physical” one. That is, a chunk only exists as some byte offset within some local file (the exception to this is of course when a file download is in flight).

Each node (of both master and client varieties) runs an RPC server. RPCs sent to these servers are the means by which nodes communicate with one another. There are several types of RPCs implemented to facilitate metadata transfer, file data transfer, and playback synchronization. Certain RPCs are implemented only on the server (such as heartbeat and library info). Others are implemented on all nodes (e.g. chunk request).



Implementation

Heartbeat

Nodes initially join the group by sending a heartbeat to the master node. The master responds and assigns the node a unique node ID, which will be used for the duration of the session. The master's response contains the current status and membership of the group. This information will be used by the client to establish connections with non-master nodes for file transfer.

The master will time out nodes that it has not heard from in some configurable period of time. Once a timeout is triggered, the node will be removed from the master's group table and this information will be propagated to the other clients at the next heartbeat. Clients will then remove the defunct node from their own group table and will no longer attempt to communicate with it.

Library Synchronization

In addition to the periodic heartbeat with the master, nodes send periodic library synchronization messages. These messages inform the master what files are currently available from each client. The master responds to these messages by sending a delta of its (authoritative) library compared to the client's. These synchronizations are relatively lightweight since they do not contain all of the required metadata to download a file (i.e. the list of hosting nodes is not sent, nor are the chunk fingerprints).

Non-serving nodes

Mobile devices are clearly a very desirable platform for MusicNode to run on. The ability to seamlessly play music files from a home device while away from home is rather convenient. That being said, a mobile node cannot act precisely like a typical node. First, serving as a source for other nodes to download from is quite likely not desired. With battery and data usage considerations, it's unlikely that a user would be happy using their phone as a file server.

Additionally, it's not practical for some clients to serve files under this architecture. If a device is behind a NAT and port forwarding is unavailable or undesirable, external nodes will not be able to (easily) create a direct connection to it.

To accommodate these scenarios, an option exists to turn off serving functionality for a node. To enable this, the node simply stops advertising itself as a host for any file. As a result, no other nodes will attempt to make a connection to it.

Example: Downloading a file

The first step to downloading a remote file is to obtain the metadata for the file if it is not already present. This can be obtained via a RequestMetadata call to the master. This will provide the information necessary to acquire the chunks for the file and to reassemble those chunks into the original file.

Once the metadata is present, the node needs to determine which chunks are already present. It consults its database and strikes the already present chunks from the required list. Next, the node determines the "best" way to download the missing chunks. This portion is very much a work in progress and currently does a simple load balancing across available nodes. If downloads of chunks time out, or otherwise fail, the node will attempt to contact another node that has the required chunk and ask for it before retrying an already failed node.

After all the chunks are present on local disk, the node assembles the file in its entirety, deletes no longer needed temporary files, and notifies the master that the chunks are now available from it.

Example: Synchronized Playback

Synchronized playback proceeds in several steps. Any node may issue a playback synchronization request. This request is sent to the master. The master then sends SynchronizedControlRequests to each connected client.

When a client receives a SynchronizedControlRequest that indicates a file play, it checks to see if it already has the file available locally. If it does, it will respond to the server immediately, indicating it is ready for playback. If it does not, it needs to proceed with a file download before it can respond. Once it successfully completes its download, it responds.

After all active clients have responded to the requests, then master can then issue a SynchronizedPlaybackCommand. This is an indicator to the clients that they should begin playback of the specified file at their earliest convenience.

RPC System

Protocol Buffers

Underlying the RPC system are Google Protocol buffers. These messages are specified in .proto files in a C-like language. The proto files are then converted into Java files (or other languages). Protocol buffers are known to be relatively fast (compared to Java?s native serialization/deserialization).

Notably absent from the publicly available implementation of protocol buffers is an RPC implementation, so a simple version was created for this project. RPCs can be executed synchronously or asynchronously and calls may be retried a configurable number of times. It should be noted that even the "synchronous" RPCs are actually asynchronous RPCs with a block on the calling thread, rather than obstructing the entire connection.

Additionally, a code generator for RPC files is in progress (that is, it reads the .proto file and outputs skeleton Java files).

File chunking

Fingerprinting

In the current implementation, fingerprints are calculated by simply taking an MD5 hash of the chunk. This will be updated to use Rabin fingerprinting. The main reason for this is performance. Rabin fingerprints are considerably cheaper to compute.

Chunk Size

Files are broken up into chunks of 32KB (with the final chunk being smaller, unless the file happens to be a perfect multiple of 32KB). Using variable sized chunks was explored, but given the expected usage pattern of media files, this was not necessary (i.e. media will typically be write once, read many times).

File Assembly

Once a node acquires all chunks required for a file, it is a relatively trivial matter to assemble the chunks in the proper order to recreate the original file. The node can then mark that file as locally available and begin serving its chunks (if applicable).

Handling Failure

Master Failure

In the current implementation, a master failure leads to a number of features being unavailable, including playback synchronization and library synchronization. File transfers can continue as long as nodes are aware of which client node holds the necessary chunks. Clients will attempt to reconnect to the master with an exponential backoff. Once the reconnect succeeds, operation will resume normally.

A desirable feature to add would be automatic master failover to nodes that are capable (i.e. a mobile node would be ineligible to become master).

Client Failure

Client failures or disconnects are much simpler to deal with. Clients send heartbeat messages to the master at a regular, configurable interval. The master will time out clients that it does not hear from within some, also configurable, multiple of the heartbeat interval. Upon a heartbeat timeout, the master will remove the client from the synchronized play group and will no longer advertise chunks as being available from the timed out node. It will also broadcast a message to other clients indicating that the node has timed out.

Future Work

Streaming Playback

As the implementation stands today, a file must be completely downloaded before playback can begin. There is no reason we need a complete file before we can begin playback. The download queue would need to be updated to aggressively download chunks that are needed ?soon? instead of proceeding in a relatively haphazard manner. Additionally, a streaming player would need to be implemented.

Tighter Playback Synchronization

While playback is relatively tightly synchronized over a local area network, it is much less so when the master is far from the nodes. There are a number of interesting techniques involving round trip time estimation that could allow rather tight control of the playback. This would also be useful for local area network playback.

Related Work

LBFS

This work draws on a number of ideas from LBFS while leaving several out. The main idea from LBFS that has been implemented here is the breaking up of files into chunks [4]. The difference is that files are broken up into static size chunks, rather than variable size chunks. This decision was made because MusicNode is not intended to be used as a general purpose file system, but for media files, which will rarely (if ever) change, so defense against a prepend to a file shifting most of the chunks (and changing the fingerprints) is much less important.

Napster

There is a resemblance to Napster and other peer to peer file sharing systems in MusicNode. Notably, MusicNode also uses a central server to hold the file metadata as did Napster, which clients connect to in an effort to determine which peers to download data from. A significant difference, however, is that the "master" in Napster is controlled by a third party, Napster itself, while the clients are controlled by the users.

Evaluation

Bits saved

An analysis of a small corpus of MP3s (67 files of an average size of approximately 4.5MB) shows that the number of duplicated chunks is quite small, although clearly increasing with a smaller chunk size (which is expected). While smaller chunk sizes lead to more reuse, this incurs overhead in metadata exchange as well as local storage.

Size	**Distinct**	**Duplicates**
2K	74142	32
4K	37100	10
8K	18577	5
16K	9307	3
32K	4674	0

Code

The source code is available from Github. The RPC server is here: https://github.com/cjriley/rpcv2 MusicNode itself is here: https://github.com/cjriley/cs244b-project-fall2014.

References

[1] Annapureddy, S., Freedman, M. J., and Mazieres, D. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 129–142.

[2] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.

[3] Mashtizadeh, A. J., Bittau, A., Huang, Y. F., and Mazieres, D. Replication, history, and grafting in the ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 151–166.

[4] Muthitacharoen, A., Chen, B., and Mazieres, D. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 174–187.

[5] Wikipedia - napster. `http://en.wikipedia.org/wiki/Napster`. Accessed: 2014-12-01.