Fun With Chords – A Distributed Music Player

Lawrence Xing
lxing@stanford.edu

Mark Ulrich
mulrich@stanford.edu

Omar Diab
odiab@stanford.edu

ABSTRACT

Fun With Chords (FWC) is a distributed music player designed to replicate media content and playback commands. We employ techniques from existing distributed systems to make our implementation scalable and robust. From an architectural standpoint, FWC is a very simple distributed file system with a time synchronization module to simultaneously execute commands on multiple nodes.

1. INTRODUCTION

Digital music playback has evolved from disjoint computers playing files on physical media to highly available streaming services that give users access to a huge on-demand library. These services largely cater to the use case of a single computer playing music in isolation. In reality, people typically have multiple computers or devices that can output media, but there are few ways of synchronizing playback between them.

Existing solutions for playing media on multiple speaker systems are limited. Companies like Sonos [1] sell speakers that coordinate with each other, but this is expensive and requires specialized hardware. Another option is to use a central playback server wired to several speaker systems, but this is time consuming and expensive to set up.

As an alternative, we present Fun With Chords (FWC) as a decentralized media system that offers synchronized media playback with minimal required user knowledge. FWC requires no specialized hardware, no software setup, and automatically replicates local media files to other machines.

FWC synthesizes several techniques from existing distributed systems. It provides replication and availability for uploaded media files using a ring-keyspace distributed hash table. It also synchronizes application-level playback commands with a dedicated clock module on each machine, similar to Google Spanner's TrueTime [2].

Section 2 discusses related work that motivated our design. Section 3 presents the data model and API and some justification for the design. Section 4 presents the system architecture. Section 5 discusses implementation. Section 6 discusses performance testing and results. Section 7 exposes implementation details. Section 8 goes over consistency guarantees.

2. RELATED WORK

Napster, one of the earliest large-scale music sharing networks, used a centralized index that kept track of every user and file on the network. Users fetched content by querying the central index for appropriate peers, then downloading the file from those peers.

Spotify [4] functions similarly to Napster by providing a centralized peer indexing service, except it also replicates copies of each music track on a DHT-based backend. To fetch music, clients alternate between requesting chunks from peers and from the Spotify replication service.

Outside of the music space, there exist various systems for scalable data replication and lookup. Chord [5] consistently hashes nodes and files onto a one-dimensional keyspace and assigns files to nodes by keyspace successor. Amazon's Dynamo [6] and Facebook's Cassandra [7] build on this by replicating

data across multiple Chord nodes, providing conflict resolution across replicas, and using alternative key lookup mechanisms.

For synchronization, Google's Spanner guarantees bounds on clock drift by polling servers against atomic clocks, using those guarantees to ensure external transaction consistency.

Sonos sells speakers and amplification hardware that can play music streamed from a remote source. The hardware coordinates simultaneous playback via a proprietary wireless mesh network called SonosNet, in which speakers communicate with each other speaker by using intermediate speakers and bridges as proxies.

3. API

FWC nodes serve as both clients and replicas, and each node is capable of executing API methods. The API is pretty simple:

```
- insert(file)
- play(fileName)
```

The FWC API does not support file deletion – once uploaded, a file remains in the system until the last replica is shut down. It also does not support modifications: once uploaded, a file's data is never changed. There are limited cases where one would want to modify a music file anyways (mostly metadata-related), and we ignore those cases.

4. System Architecture

4.1. Identifiers

Nodes on FWC have a unique human-readable identifier. Because FWC doesn't employ a master server to coordinate naming, each node generates a random identifier on startup. This can result in collisions, and limits scalability as the Birthday Paradox becomes a problem. We're going to modify our protocol so that bootstrapping nodes wait until they have full knowledge of node membership before picking an id.

FWC generates the identifier for each file as the hash of the file's contents, as described in Shark [8]. In the original paper, this was to ensure security so that an adversary could not guess the file contents. Here we use it to prevent uploading of duplicate files. Since file contents are never modified, the identifier will correctly prevent collisions on the file for its lifetime.

4.2. Partitioning and Replication

FWC must support dynamic partitioning as nodes join or leave the system. We reuse the Chord approach of consistent hashing. Each node is assigned to a random position on a circular keyspace (a "ring"). To locate a file id on this keyspace, FWC hashes the id into a key and selects the node immediately succeeding the key on the ring.

To provide robustness, the system must replicate data across multiple nodes. We use the simplest, locality-unaware solution of assigning file keys to the next R successors on the ring, where R is a replication factor configured per instance of FWC. We refer to these successor nodes as *designated replicas*. We don't have any mechanism to prevent load imbalance induced by the pseudorandom assignment, although rebalancing solutions exist.

Replicating data to targeted nodes is not sufficient, though. In order to fully use the FWC API, a node must have knowledge of every file uploaded to the network, even if the node is not a designated replica. To handle this, we introduce multiple levels of replication for files. A file record on a node can have one of three states:

- Remote: The file record contains only metadata. Globally replicated.

- Cached: The file record contains metadata and the file contents. Present when the file is on local disk but not pinned.

- Pinned: The file record contains metadata and the file contents, and the current node is a designated replica for the file.

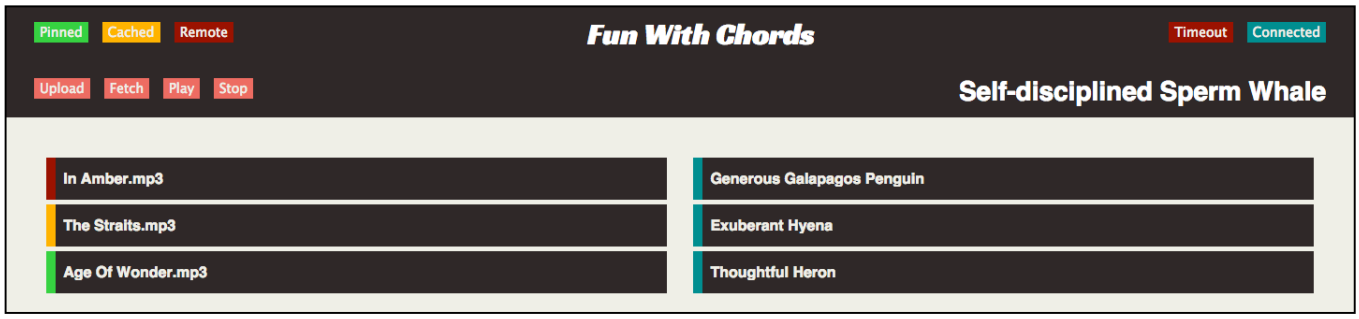

Figure 1: The FWC client. The local file index consists of one remote file record, one cached file record, and one pinned file record (left). There are three other nodes in the group (right).

When a file is uploaded through a FWC node, the node copies the file to each designated replica, creating pinned file records on those replicas. In addition, the uploading node broadcasts a remote file record to every other node on the ring. This allows non-replica nodes to learn about and issue play commands for files that aren't locally available.

Cached file records exist to preserve bandwidth by keeping file data at nodes other than designated replicas. They can arise in several ways. First, if a file is uploaded from a node but the node is not a designated replica, the node will keep a cached copy of the file. Secondly, when a node joins the ring, the rebalancing operation may lead to another node losing designated replica status on a file. In this case, the node won't discard the file, but will instead downgrade its pinned record to a cached record. Lastly, nodes will fetch files and cache them in anticipation of or in response to queued plays. Unlike distributed sloppy hash trees such as Coral [9], nodes with cached but unpinned file records don't advertise their cached contents. This simplifies the routing protocol at the expense of download parallelizability.

Pinned records are like cached records, but their file data cannot be evicted from node-local storage. This ensures the file is constantly available until the record becomes unpinned.

4.3. Membership and Consistency

Each node subscribes to a global communication channel, which provides a mechanism for efficient broadcast. The channel's implementation also usefully provides notifications to subscribers when other nodes join or leave/timeout from the channel, which allows each node to maintain a full list of nodes in the system. We assume that this channel is reliable and does not permit network partitions, which in turn lets us assert that the nodes maintain a consistent view of the group's membership.

Nodes only use the global channel to exchange membership information and for broadcasts. They use peer-to-peer connections to mediate data exchange. We do not make any reliability assumptions about these peer-to-peer channels, so there is no guarantee the file system will be globally consistent. Since files are immutable and only created, not destroyed, the worst consequence of this is that a node will be missing a file record. This is resolved in two ways. When a node receives a play command for a file record it doesn't have, it fetches the file from a designated replica. Alternatively, a node may be missing a file record when it is designated replica for that file. When it receives a request for the file, it redirects it to a different replica, then fetches the file record for itself.

The global channel removes the need for a master or coordinator to trigger membership change operations. Instead, each node reacts to group membership notifications from the channel by itself. These operations are briefly described below.

A new node will bootstrap itself into the ring by contacting a random group member and requesting remote copies of its file records and membership data. The new node requests pinned copies of remote files for which it is now the designated replica. Meanwhile, those displaced replicas will respond to the new node's join event by downgrading their pinned file records to cached.

When a node leaves, every other node scans its file index for files where it might have inherited

designated replica status, then fetches those files as pinned records to rebalance the dropped node's load. This is one disadvantage of not using a coordinator, since every node must perform the scan regardless of whether or not they need to participate in rebalancing. However, the extra work is primarily computation, and does not consume any more bandwidth than a coordinated rebalancing operation.

4.4. Playback Synchronization

To handle a `play(filename)` operation, a FWC node selects a play time and broadcasts the operation to the other nodes. A recipient node that only possesses a remote file record for the file (or that lacks the file record entirely) will (1) buffer the play command (2) request the file data from a random designated replica (3) store it as a cached file record and (4) execute the buffered play. Nodes that possess a pinned or cached copy of the file data will immediately execute playback.

Since the envisioned use case of this project is for audio playback in a cluster of machines within hearing distance of one another, synchronizing the timing of playback is extremely important. Instead of trying to accommodate communication latencies between machines, each machine stays in sync with a global clock as closely as possible.

The implementation mirrors Google's Spanner, which maintains a number of GPS and atomic-clock based *time masters* that are nearly perfectly synchronized with one another. Each Spanner machine polls against the multiple masters, keeping high bounds on uncertainty as a function of clock drift and time since last poll.

FWC polls against a single time master. Unlike Spanner, the protocol cannot use waits to mitigate uncertainty since it wants to ensure every machine plays at the same time, not that one machine definitively plays before another. Instead, each node collects the round trip time for server polls. It uses this round trip time to estimate the absolute offset between the local clock and the master. Using a moving window of these estimated offsets, the node generates an aggregated offset in a number of ways:

- Mean
- Exponentially weighted moving average
- Mean, with high round-trip times discarded (since these incur high uncertainty)

Using this aggregated offset, the node estimates the master time by adjusting the local time by the offset. All playback operations on a node are synchronized against the locally estimated master time.

5. Implementation

For hardware interoperability, our client runs as a web application. The implementation relies on WebRTC, an open web standard for direct peer-to-peer communication between web browsers. The protocol was designed for bandwidth-heavy applications like video calls and file sharing, making it an appropriate choice for this project. We heavily leverage the WebRTC DataChannel API, which enables transmission of arbitrary data packets [10].

WebRTC requires a "signaling server" to coordinate metadata exchange between peers [11]. To expedite our implementation we used a third-party service called PubNub [12]. In addition to working as a WebRTC signaling server, PubNub provides two important tools. First, it supports a time() endpoint that acts as a time master for the synchronization protocol described in Section 4.4. Secondly, PubNub exposes an abstraction called channels, which serve as the global communication channel in Section 4.3 used to propagate group membership information.

Our implementation does not write any data to disk, instead taking advantage of most modern browsers' generous memory allocations to cache all file data in memory. This means all file data is lost when the last node in the ring is shut down. We intend to write a serialization mechanism that lets us write files to local browser storage in encoded chunks. Alternatively, the caching layer is flexible enough that it can be adapted to flush to a disk-based storage with minimal effort and an appropriate interface.

The entire client is implemented in Javascript, with some HTML/CSS for the user interface. The code is 2200 lines.

Aggregation Function	Standard deviation of computed offset (ms)
Mean	53
EWMA	65
Filtered Mean	60

Table 1: Variance in computed offsets from synchronization modules on the same machine.

6. Evaluation

6.1. Playback Synchronization

We benchmarked the accuracy of the FWC time synchronization module by running five instances of a FWC from the same machine, Google Chrome on a MacBook Pro with a low-latency WiFi connection (ping to Google of 2-5 ms). We could not figure out how to compare data from separate machines. Each instance used a poll time of 2 seconds, a moving window of 60 estimated offsets, and ran for 60 intervals (until the moving window was fully populated). At the end of the last interval, each instance computed its aggregate offset and we computed the standard deviation of the offsets.

This procedure was replicated for each of the three algorithms. Exponentially weighted moving average (EWMA) used a decay factor of 0.9. Filtered mean discarded all offsets from the moving window except those computed with the lowest 20% of round-trip times, effectively shrinking the moving window to 12 entries. This yielded the data in Table 1. The results represent an acceptable delay for synchronized music playback from different nodes – in practice, playbacks on separate machines produced a strong reverb but no perceptible desynchronization.

6.2. Replication

FWC replicates files not only to provide robustness in the face of node failure, but also to provide multiple endpoints for nodes to retrieve file data during playback. Figure 2 presents benchmarks verifying this feature. We connected eight iMacs to a single FWC group via Ethernet, in addition to a

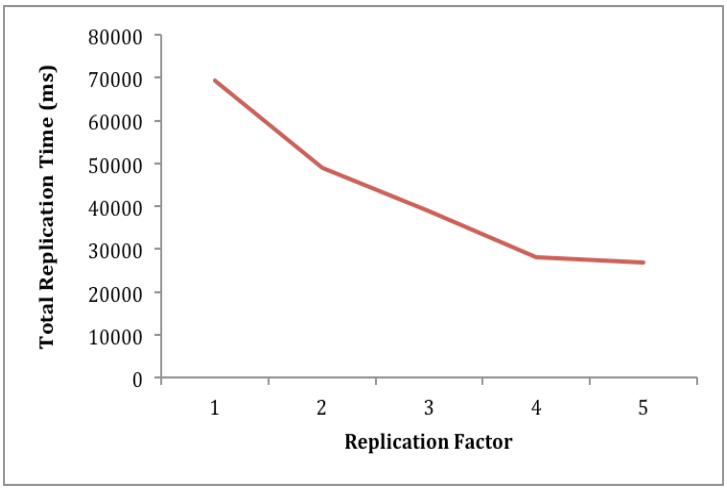

Figure 2: Replication times for various replication factors.

MacBook Pro connected via WiFi. All machines used Google Chrome to run the client. The laptop was used only to upload files to the network. No files were pinned to the laptop, and it always kept a cached copy of each file, ensuring that it never participated in any file data transfers. We initiated playbacks of an 80.2 MB .wav file from the laptop with varying replication levels and collected the total time for the file to be fully replicated (pinned or cached) at each node.

7. Conclusions

To summarize, Fun With Chords implements a simplified distributed hash table with replication and synchronized broadcast operations. It's easily configurable and scales to more nodes than is necessary for a home-play environment, as demonstrated in 6.2 when we initialized FWC on eight Stanford Library computers with no dependencies other than an internet connection.

We've outlined possibilities for future iterations throughout this paper. Virtual node ids or smart node insertion for load balancing, cache-aware routing protocols, and persistent local storage are features we intend to implement next.

We're grateful to the CS244b instructors for giving us the tools to implement this project.

References

[1] Sonos. http://www.sonos.com/system?r=1.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. To appear in *Proceedings of OSDI*, page 1, 2012.

[3] Sad but true: Napster '99 still smokes Spotify 2012. https://news.ycombinator.com/item?id=3716948.

[4] M. Goldmann and G. Kreitz, "Measurements on the Spotify peer-assisted music-on-demand streaming system," in *2011 IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2011, pp. 206 –211.

[5] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek and H. Balakrishnan, 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, 2003.

[6] G. de Candia, D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220. ACM, 2007.

[7] A. Lakshman and P. Malik: Cassandra: a decentralized structured storage system. *Operating Systems Review* 44(2): 35-40 (2010).

[8] S. Anapureddy, M.J. Freedman and D. Mazieres. Shark: Scaling File Servers via Cooperative Caching. *Proc. 2nd Symp. Network Systems Design and Implementation*, pp.129-142 2005.

[9] M. J. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters. In *IPTPS*, 2003.

[10] WebRTC, http://www.webrtc.org/reference/architecture

[11] HTML5 Rocks, http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

[12] PubNub, http://www.pubnub.com/blog/what-is-webrtc/