YourTunes File System

Sean Collins, Tim Farley, Robert Hensey

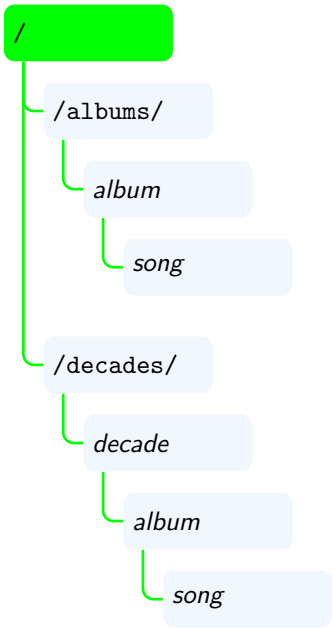
CS 3210

1 Introduction

The **YourTunes File System** is a FUSE-based implementation of a filesystem tailored towards music files. It is built on top of a remote backend server, which stores file data and metadata on a database server with a RESTful web server frontend. The client is responsible for transparently querying the remote server for resident files and constructing a metadata-based abstraction based on the parsed server response.

1.1 Directory structure

The directory structure on the local filesystem is based on audio metadata - specifically, the `Album`, `Title`, `Track`, and `Year` ID3 fields.



Each song is placed as a leaf in two separate directory trees: one based on albums directly, and one based on albums categorized by the decade that they were released in (parsed from the `Year` field). In the case of missing metadata, a song is placed in a "Unknown" folder for both cases (`/albums/Unknown/`*file*, `/decades/Unknown/Unknown/`*file*). Files with no metadata are also put into these buckets.

Given the same metadata, the file pointers through both directory hierarchies are pointers to the same file data.

1.2 Example

Given an audio file with the following metadata:

Track	4
Title	Jesus, Take the Wheel
Artist	Carrie Underwood
Album	*Some Hearts*
Year	2005

The filesystem will present the following abstraction:

```
/
├── albums/
│   └── Some Hearts/
│       └── 4-Jesus, Take the Wheel
└── decades/
    └── 2000s/
        └── Some Hearts/
            └── 4-Jesus, Take the Wheel
```

2 Client

The client program is built on FUSE, which allows specification of function pointers that implement core filesystem I/O functions through a `fuse_operations` struct (e.g. `read()`, `write()`, `statfs()`...). When the filesystem is mounted to a local directory, metadata is requested from the server via a `HTTP GET` to `/ls` on the server[1]. The metadata is cached into a local cache directory, where it is parsed and used to build the structure detailed above.

When a file is moved into the mountpoint, the client performs an `HTTP POST` to `/upload`. File accesses are handled via an `HTTP GET` to `/get_file`, and removals are handled with an `HTTP GET` to `/delete_file`. Upon any of these filesystem changes, `/ls` is queried again to download updated metadata from the server.

2.1 Setup

The following packages are required: `libfuse-dev` `pkg-config`, `mp3info`, `curl`, `jq`. In addition, to run the test performance scripts, the `python3` package is required, along with the `stagger` Python library. Assuming a Debian-based system, these can be installed by running
`./install-deps.sh` or running the following command manually:

```
sudo apt-get install -y libfuse-dev pkg-config mp3info curl jq
    python3
```

[1]The client uses the `curl` utility for HTTP requests.

```
sudo pip3 install stagger
```

Once the dependencies are installed, run `make` from the project root to build the client. Finally, run `./yourtuneslib <mountpoint>` to mount the filesystem to a local directory, with `<mountpoint>` being the path to an existing directory.

3 Server

The backing server serves as the file and metadata store for the filesystem. It provides an interface to list, add, and remove files as well as query for metadata. It runs on a MySQL database and Python web server under the Flask framework.

Upon an upload, the server computes a SHA1 hash of the file data and uses the hash value as the raw filename in the data directory. The filename is used to uniquely identify an individual file. The metadata is then parsed and inserted along with the filename into the MySQL database, which is queried upon a call to `/ls` (see below).

3.1 Setup

Run `./install.sh` in the `server` directory. You will need sudo privileges as well as a root MySQL database password, and will be prompted for these when required. If MySQL is not installed, it will be installed for you, with the root password being set to `team14`.

Once the script completes, start the server by running `python3 server.py`.

3.2 API

3.2.1 `GET /ls`

Parameters: *None.*

Returns a recursive listing of all files on the filesystem, represented in JSON format.

Example output

```
{
  "albums": {
    "Project 3": [
      {
        "filename": "93fb5d4840ea88f4174e5d03b63a577e4bc7fbaf.mp3",
        "filesize": 43054,
        "title": "Test Song",
        "track": 1
      }
    ]
```

```
  },
  "decades": {
    "2010": {
      "Project 3": [
        {
          "filename": "93fb5d4840ea88f4174e5d03b63a577e4bc7fbaf.mp3",
          "filesize": 43054,
          "title": "Test Song",
          "track": 1
        }
      ]
    }
  }
}
```

3.2.2 `GET /get_file/`*filename*

Parameters: `filename` The name of the file (as output by `/ls` above).

Serves a raw file given by the filename through HTTP.

3.2.3 `GET /delete_file/`*filename*

Parameters: `filename` The name of the file (as output by `/ls` above).

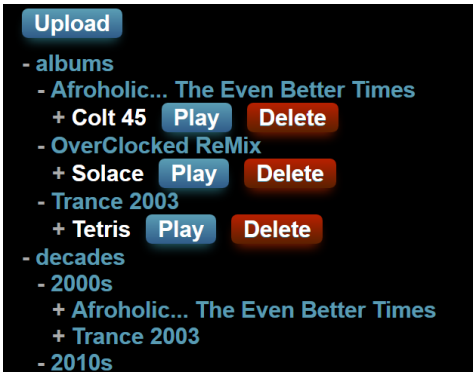
Deletes a file given by the filename from the server.

3.2.4 `POST /upload`

Parameters: `file_data` The raw file data.

Uploads a file to the file store.

3.3 Web interface



In addition, the server includes a web interface accessible at `/web.html`. In a use case which does not use the client FUSE binary (mobile devices, for example), the web interface allows users to access, delete, and upload files on the filesystem through HTTP. It parses the same metadata as the base client, but constructs an HTML tree from the parsed JSON to represent the filesystem structure. File access, deletion, and upload are performed using the same API detailed above.

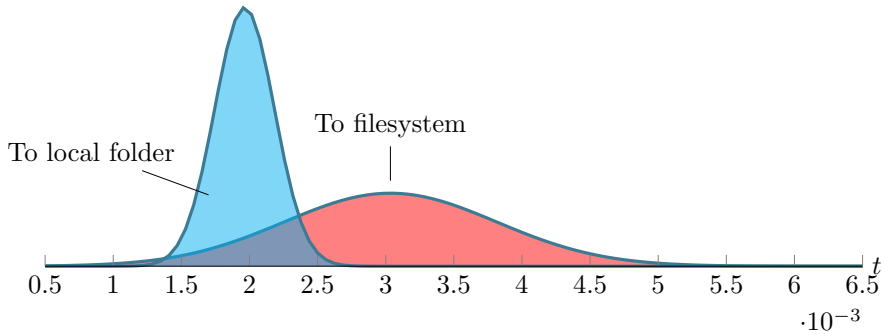
4 Performance benchmarks

Two series of tests were performed to measure performance: file write latency and mount delay based on number of files existing on the server. Both tests measure nanosecond-precision delay of atomic operations (file write, filesystem mount), and are run using a local backend server.

4.1 Write latency

Tests were performed with the included `perf-test-no-fuse.sh` and `perf-test-with-fuse.sh` test programs, which measure performance copying a test file from a local source to destination. For the purposes of comparison, the first test copies the file to a local folder, and the second test copies the file to a local FUSE mountpoint (a remote destination).

Run them with `./perf-test-no-fuse.sh` and `./perf-test-with-fuse.sh`, respectively.



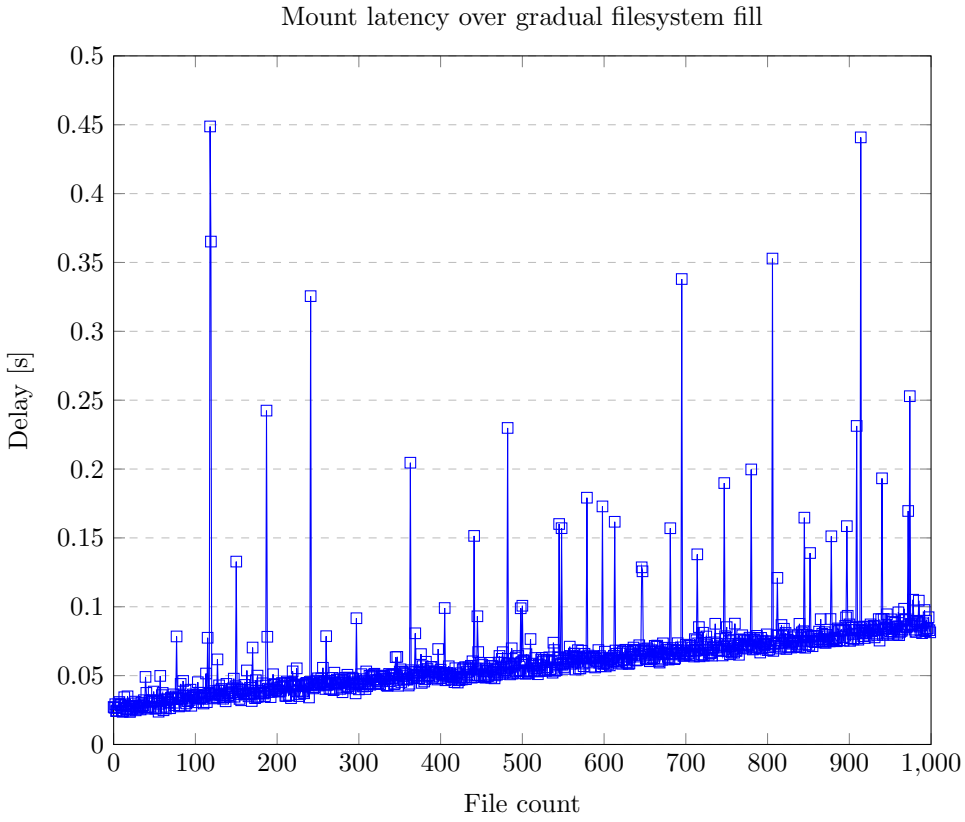
	Min	Max	Mean	Std. Div.
To local folder	0.001754 s	0.002715 s	**0.001964 s**	**0.000220 s**
To filesystem	0.002236 s	0.006568 s	**0.003034 s**	**0.000779 s**
Distance	+0.000482 s	+0.003853 s	**+0.001070 s**	**+0.000559 s**

Analysis
Copying a file to the FUSE filesystem results in slightly higher latency, as there are additional costs associated with network I/O and post-copy metadata parsing.

4.2 Mount delay

Tests were performed with the included `perf-test-fuse-mount.sh`, which generates audio files with distinct metadata and measures the delay in mounting the filesystem over gradual fill with the test files.

To run the test, run `./perf-test-fuse-mount.sh`.



Analysis
Mount latency is linearly tied to the amount of files resident on the file system. This is likely due to the increased size of the metadata returned by the server on each mount, resulting in greater parse time.