Abstract

In order to allow users to persistently share mutable media between both themselves and others, the ability to transparently manage current copies of content over several electronic devices has become a must. True transparency, however, is an incredibly difficult task due to a variety of technical constraints. The greatest issue with maintaining consistency between files is that this sort of concurrency requires software software must, to some degree, understand which files have logical dependencies with others. Given a general collection of files it is impossible to try to comprehend information they contain, it would be wasted effort to try to grasp the meaning behind individual files. The goal in this project was to use a hierarchy of torrent files to hash the directory, including its children directories, in such a way that would both allow for the distribution of file storage and bandwidth over a collection of hosts as well as allow users to maintain data consistency in directories without intruding on their work flow.

Overview

The service I designed operates like a decentralized Dropbox: users identify directories that they want to be maanged across several electronic devices the service transparently keeps the files consistent over all of the devices. However, there are several key issues that must be dealt with by such a system, the foremost issue being a standard concurrency issue: data consistency. I decided to explore trying to decentralize file storage in order to meet several curiousities as to what Bittorrent could be applied to. In particular, I wanted to see how much of resources could be amortized between different machines.

The motivation for this approach is that my family has quite a bit of media (on the scale of several gigabytes) that I'm in charge of sharing with others, but I don't have the resources to simply serve several gigabytes of data to others due to various resource constraints such as bandwidth. Therefore I wanted to have a mechanism to allow family members to corroborate a large quantity of media among each other while trying to amortize resource costs without forcing anyone to maintain their own dedicated servers. I wanted to be able to provide a transparent service that would be able to automatically load media onto others' computers.

Though the service that I currently have implemented is fairly rudimentary and admittedly lackluster, I have aspirations to continue working on it to see the advantages that I have envisioned can be realized.

Methodology

The basic responsibility of this service is to synchronize directories over a network while being able to keep file consistency. In order to do this I wound up making several design decisions that I will describe bellow.

Language Choice

The choice to use Python was less one for the intent of the project but rather one to allow for rapid prototyping which is convenient for rapidly testing code especially when using unfamiliar libraries or for introspecting into binary files especially with all of the possible confusion caused by trying to keep consistency over a distributed system. Python provided a simple method for storing any data I wanted to keep through a built in library called Pickle. Pickle provides a simple mechanism for storing and extracting the dictionaries that I used to represent the versions of different directories. When I was looking for tools to help me develop a Bittorrent client rasterbar's libtorrent library was mentioned to be a well known Bittorrent library written in C with Python bindings. I preferred to not use C because of the incremental development style I chose so using Python almost became a necessity.

Bittorrent

Bittorrent was a convenient and natural solution to some of the behaviors I wanted my implementation to have. The first goal that Bittorrent met was that it reduced the load required by a single server to a couple of HTTP requests and the storage / serving of a few torrent files, usually less than 100KB. In fact, for my use case I could see users not even using the server to transfer files, but rather emailing torrent files around to tell each other what to look at.

One of my initial goals was to have a mechanism to detect detect changes in whole directory trees and a way to represent the changes in the versions of child directories to maintain snapshots of whole directory trees, which led me to the conclusion that I should represent directory versions as hashes and represent directories as hashes of their contents, as one sees in a Merkle tree. This would allow for users to select a particular version of a directory and give the system a unique way to propagate the desired state to all of child directories while minimizing the amount of information on the children directories that it needs to store. Thus Bittorrent theoretically allows for a highly

localized compact notation for the state of a whole subtree of directories in a single hash, which I wanted to take advantage of.

My other main concern was that I wanted to have a robust way to transfer files between the various clients in a decentralized peer-to-peer fashion. Bittorrent provides precisely that by keeping hashes of file contents and by being almost completely decentralized apart from the tracker. A neat trick that I realized about Bittorrent is that no client necessarily needs to host all of the data, and theoretically Bittorrent inherently allows for a series of servers for which no individual user has the whole repository, but anyone who wants to fetch the whole repository could do so. This is possible because Bittorrent allows for users to fetch incomplete files as long as the blocks are available. It's possible that sharing could be done in such a way that no server would need to have a complete copy of any individual file! Though I do not intend to distribute servers so no individual file is fully contained on a server, I do imagine my users to only want to have preferences over which files they are willing to share or not, so I do imagine seeing partial copies of repositories. However, because of my choice to use Bittorrent, file storage turned out to be much trickier than I initially imagined.

File Storage

Within each directory being managed I stored a `.btv` directory locally that contained all of the book keeping information required to both store all of the versions of files and make them ready to be published over Bittorrent at the same time. The precise details on storage turned out to be the most challenging part of this task due to the various constraints that some of my major goals provided on how I could store and manage the book keeping information that I wanted.

I chose to let each torrent file represent a file version, so if one wanted to quickly download a specific version of a directory they could simply fetch the torrent for that directory and obtain only the files from that directory. However, the choice of Bittorrent left some restrictions on the way I could store files if I wanted to minimize the amount of extra data store. After all, if I were to store a copy of each file for each torrent, I would end up with n copies of a file in order to have n versions of a file. Therefore I chose to store an initial copy of a file as well as the series of delta files so no file in the versioning system would ever be modified. Therefore, if I only had one file being managed, for each of the n versions of the file would be represented by a torrent file containing the first initial copy, and the first n-1 delta files.

Bittorrent left me with the more general challenge that I could not store any files without making sure that the filenames were all unique. Therefore I stored every file with a SHA1 hash of the file contents. SHA1 hashes don't leave any ordering information I sound that I also needed to store the trees of changes for each file and store them in. Therefore for each version file stored a `state` file that was named with the same name as the torrent file, which was a pickled dictionary of file metadata, including modification times, the current file version, and an adjacency list of the graph of file versions. The modification times were not necessary but were used while I was prototyping and can be seen as a local optimization on checking for file changes, as creating hashes for whole files is a fairly expensive task compared to running stat on a file and comparing the modification times.

Usage

Prerequisites

Both the server and the client depend on Python version 2.7, but the particular version should not matter if it's >2.5 and <3.0.

Server

The server requires a running ssh daemon and the Python API of the Google App Engine (version 1.6 was used). Google App Engine can be easily found on the Google App Engine website.

A vanilla instance of sshd will suffice, and any restrictions on it are appropriate as long as one can scp and list directories in one's home directory. Currently ssh public/private keys are required in order to not have to type in passwords twice every 5 seconds.

Client

The client was designed and run on Python version 2.7 running the rasterbar libtorrent library version 0.15.7.0 and Bash. Python and Bash should already exist on any modern Linux system and libtorrent by rasterbar can be found

as python-libtorrent on Ubuntu using apt, or on rasterbar's website. The program can be run as long as all of the py files are collocated.

Execution

Server

The ssh daemon can simply be run as one would normally run one on a standard Unix machine using `/etc/init.d/sshd start` or `/etc/rc.d/sshd start` depending on the distribution. The tracker component is designed to be run on hte Google App Engine so one could upload it by creating an app called `csc2209svr` and upload it using `python appcfg.py .` in the directory where the tracker source is located. An easier way to run the tracker though, is to run it in dev mode by the command `python dev_appserver.py .` in the directory where the tracker source is located.

Client

To start an instance of this service on a directory one simply enters the directory and runs the python file `main.py` with `python main.py`. Right now there are no inputs for the program. The script `change_state.sh` is a simple mechanism for changing between versions of a directory. The script is run without any arguements with the command `sh change_state.sh`, and it will output the 30 most recent versions of the directory in the format "`index) version`". To select a version enter the index of the given version and press return. Apart from changing versions the application runs as a background service, automatically updating files along the network.

Implementation Specifics

At brief, my implementation uses a client and server, but because of my choice to use Bittorrent, the server utilizes an infinitesimal amount of resources. The server provides almost precisely the same services one would expect from a standard torrent site: a mechanism for fetching .torrent files and a Bittorrent tracker. It's up to the client to interpret the torrent files and the contents that the torrents collect to maintain the various snapshots of directories being tracked.

Server

The server is easily composed of two parts: one service for serving files to users and one service for letting users know the other users that have files that they want. The former service can easily be implemented by using any standard file service such as ftp, and I chose to use ssh/scp to a specified repository directory to require authentication and give this component some semblance of security. The torrents for the directory `$PATH` from the remote directory `server:$REPOS/DIR` directory . My choice of using Bittorrent was defined how to provide a service to let users know where other users are, and I chose to let the client do enough of the processing such that I was able to use an out-of-the box Bittorrent tracker. I chose to use the tracker called Attrack because of its simplistic nature and because of my comfort with python. Though I haven't implemented it yet, I plan to apply some security features which I will mention later.

Client

The client is run by three main classes that separate checking for file changes, writing file changes, and networking which are split into the lurker, the patcher, and the file manager. This separation was used to distinguish between the permissions of various components of the system: the lurker never modifies any torrents or files directly, the patcher only creates new files in the `.btv` directory or modifies the original directory, and the version manager only has access to the network and creating states and torrents. Quite a bit of functionality, however, is built into a utility library `util.py` including a few global variables.

Everything is run in a main loop, located in `main.py`. The program initializes itself by checking through the whole directory tree for file versions and initializes the necessary torrents for each of the directories being managed. The lurker is called once every five seconds in a main loop to scan for file changes in the file hierarchy and if changes are detected locally, the file manager creates a new torrent, seeds it, and publicizes the new torrent on the server. If the lurker feels confident that it can update the version or the user specifies that they want to revert to a particular version of the directory it calls the patcher to change the version. Since the file manager automatically produces new branches before checking for other branches to try to follow it will only update to a remote version if no edits have been made recently.

Lurker

The operation of the lurker is fairly simple, though it gave me the most trouble during development. It begins by trying to load the state of the system if one exists so it can scan through each file and directory for changes. The files are checked through a quick mtime check and if the mtime check succeeds a SHA1 hash of the file is generated to, with a high level of confidence, guarantee that the file has changed. For the directories I first run the lurker on them to detect changes and if the state of the directory is changed after lurking through it it's marked as modified. The files and directories are marked as new, modified, or deleted to specify if I need to create a new base version for the file, a patch, or simply mark the file in the file state dictionary that keeps track of all of the versions each for the files has gone through. The file state dictionary is used by other tools such as the patcher to recompose files as the hashes for the files become the names of the newest patches for the files. After scanning for changes the lurker checks if a `.btv/current` file has exists, and if it does it checks the contents of the file to discern a new user-designated version that the system should be on. If such a file exists the system calls on the patcher to set the version of the entire directory tree to be at the specified version. If the version manager reports that the system is sufficiently stable, there are newer versions in the current branch of the current directory, and the directory wasn't manually set, the system automatically updates the version of the system to be the most current one in its branch. / The difficulties of this task was largely a concurrency issue. Most combinations of when to scan for changes, set the current state, write the current state, and creating new states turned out to be bad. I spent a significant amount of time developing on a single client and playing with how to version one client because of the issues that developed when I introduced a second client. The most annoying issue was that both clients would create new states for themselves as updates to the state of the other, which would result in both clients producing a new state every five seconds without any proper automatic updates occurring in the directories themselves.

File Manager

The file manager has the primary responsibility of keeping track of the various versions available to the system and in the network. It operates under two main modes: one for loading versions and one for creating versions.

To load versions it recurively checks through directories to find torrents and opens the torrents in each directories respective `.btv` directory as well as store some information on each of the torrents. Each torrent is named after the hash for its version and contains the version of its parent as a comment. Therefore, in order to maintain a graph of the parent relationships between versions I read the comments of each torrent file to create a graph used by the lurker to discern what versions to advance the directories to.

The creation of torrents was fairly straightforward. The file manager receives a list of new, modified, or deleted files which it defers to the patcher for creating patches or changing a current state for, and then creates a new torrent file containing all of the data of previous versions.

I ran into a few issues because the library I used had a bug that made the torrents created not as compact as I wanted. I would have preferred to add files to the contents of the torrent one by one by the set of files newly created or modified, but the `add_files` method created corrupted torrent files that appear to be half way between single file mode and multi file mode. In single file mode the info-¿name entry is the file name, and under the library I used the most recent file name was always the value of info-¿name, but at the same time the files were being appended to the info-¿files list though the entries were being written as empty strings. Empty strings aren't supported by bencoding (the encoding Bittorrent uses for torrent files) so the info-¿files[i] entries, which are dictionaries, were malformed creating corrupted torrents. The process of creating the torrent files and fixing the torrent files after the fact looked far more troublesome than simply dumping everything in a `.btv/data` directory and throwing the directory to libtorrent in bulk. This meant that any extra files added to the `.btv/data` directory would be permanently added to all of the versions and couldn't be removed. The pickled version states and file data were all put in the `.btv/data` directory.

Patcher

There were four main tasks that the patcher had to deal with: adding new files, patching existing files, deleting files, and resetting the system to version states. For new files it copies the file to `.btv/data/base` and renames it with a hash to keep file uniqueness, for modified files it creates patches and copies the patches to `.btv/data/patches` with a hashed name, and for deletion it actually writes nothing on the file system, which initially seems worrisome. However, each of these three methods return the name of the new file to the file manager which sets the current file names to be the returned values. Deleted files are noted in the states with a `d.<filename>` to virtually represent a deleted file.

The final task of the patcher was to fully revert files to previous versions. This task was fairly straightforward for files, as one simply removes the current files and adds the reformed files from the file repository, but for directories this proved tricky. Directories contain far too much information to be simply deleted, so they should be moved to somewhere invisible. I decided that they should be moved to the `.btv` directory but didn't get to implementing this due to time constraints and there's quite a bit of bookkeeping to keep track of in this task, for torrents would need to be uninstantiated in some directories and re-instantiated in other directories across the whole subtree for the directory removed. Nevertheless, for child directories without deletion the state of those children seen in the state should be set to be the versions in the children and by recursively calling the unpacking method the children should all be changed to being the appropriate versions.

I originally intended to do a differential patcher, but never got to writing a differential format. However, I wrote the patcher in such a way that obviously wants to have patches written, but whenever I have a "patch" I simply store the whole file.

The patcher actually could be run without the rest of the code, and used to help users quickly fetch files without having to use my custom torrent mechanism or lurking through directories. In the case where a user stores all of their files in a flat directory it would be fairly simple to share the torrent file and open the torrent file in any directory, and use the patcher to recompose the file that the version represents.

Restrictions and Further Work

The current program is a bit rigid with some hardcoded directory locations, but it could easily extended. Right now there is a single hard coded repository server and repository name being used, `localhost` and `myrepos`. That means that the host machine must have a directory `/btv/myrepos` available to be written to. All that is required to fix this issue is time. However, beside that one limitation there are a few other additions I wish I had time to implement.

Initially my hope was to use the inotify kernel subsystem to detect file changes (I believe Dropbox uses it) to not have to scan every file in a loop and the lurker process that is currently in place was supposed to be only temporary. By using inotify I could quickly only scan the directories that had been modified by recursing up directories parent by parent and only then generating new versions. This would turn my active scanning process into a passive scanning process, which would certainly be preferable.

Of course I want to soon implement a true differential patching system. I plan on implementing two patching systems: a text patching system which would be a wrapper around the GNU diff/patch programs and maybe a binary one which would compare patches. The latter method is of course rather limited but I am interested in looking at it because in my own use case I can imagine users using a fairly limited set of operations on their files that might make patch comparisons reasonable.

Security

As I have mentioned, I plan on using this software for a personal project with private information, which means I will someday have to implement security features to keep outsiders from obtaining personal data. The main place where data can be obtained are at that foreign users can join the torrent swarm to fetch files. However, this can be prevented through an authentication system with the tracker. Trackers operate through HTTP GET requests, so appending authentication information to the GET request should not be too difficult. In fact many trackers do implement some level of authentication security by storing authentication credentials in the torrent files themselves, and my plan is to move the storage of authentication material to the client itself. Currently the fetching of the torrent files is already relatively secure as the current implementation requires users to have ssh rsa key pairs to fetch the files at all.

Conclusion

For both this course and for myself I have developed a file synchronization service that is distributed and decentralized in essentially every possible aspect. It can successfully synchronize directory trees across a network in a way that automatically branches when discrepancies are produced by concurrent edits to a directory and will automatically update files when there's a only a single branch to follow. Though this implementation isn't doesn't support some of the features I wanted to, it is certainly a strong step towards a complete product in that all of the base functionality has already been written and all future improvements will be incremental. The code that should be replaced in the future has been written in a way that provides a sufficient proof of concept for what I want to do in the future.