CS4532 – Distributed Systems

Distributed File System Project

Adam K. Schmidt – 12312907

Overview

This implementation of a distributed file system consists of 5 main components:

1. File Servers – these are responsible purely for the storage of file contents
2. The Directory Server – responsible for mapping from the client-side directory hierarchy to the flat structure in the File Servers, as well as for managing the locking of files
3. The Authentication Server – responsible for creating, maintaining, and authorising users
4. The Client – responsible for providing a user interface with which to interact with the file system, as well as for caching accessed files for reduced network load
5. A Common API – used by all the preceding components, this allows for the API to be defined, maintained, and improved in a central location

Note: This project does not provide replication or transaction services.

Setup

The project depends on both Haskell-Stack and MongoDB. Installation is as follows:

For MongoDB: https://www.ostechnix.com/install-mongodb-linux/

For Haskell, stack, etc: https://bitbucket.org/esjmb/teaching/wiki/Developing_on_Docker_with_Stack_Emacs_Yesod_and_Intero

For the project: git clone https://github.com/qizen/DistFileSys.git

To build and run the system locally:

```
chmod u+x build.sh

chmod u+x run.sh

./build.sh

./run.sh
```

This will start the mongo service (possibly requiring superuser authentication) and run the server set.

1. The File Servers

API

Function	Inputs	Outputs

getFile	?name=<pathToFile>	DfsFile file
postFile	[JSON] DfsFile	Bool success
listFiles		[DfsDateName] files

Overview

The file servers act as purely flat storage units. They simply store every file passed to them in a single predetermined folder (by default, “/FileStore/” under the FileServer directory). They have no conception of directories themselves.

Upon starting, a file server will attempt to register itself with the directory server by calling its registerFileServer endpoint. Once the directory server receives a register command, it records the IP and port of the server and sends it a listFiles command, so that it can build up its database of files.

After this, the file server simply responds to get- and postFile requests from the directory server.

2. The Directory Server

API

Function	**Inputs**	**Outputs**
registerFileServer	(RemoteHost) plus ?port=<portNumber>	String (Success|Failure)
ls	?token=<token>&path=<path>	[DfsDirContents] conts
createFile	“	Bool success
openFile	“	Maybe DfsFile
lockFile	“	String reason
unlockFile	“	String reason

Overview

The directory server is the part of the system where higher-level file management functions happen. It stores a mapping for each file to the file server on which its hosted. It also understands folder structure. File names are stored as their full paths (with “/” replaced by “#”) so that folder structure can be deduced by looking at the start of each filename. Since MongoDB indexes its documents for faster search, this should allow for relatively efficient access without the overhead of providing a hierarchical collection structure (which Mongo is not that strong at). Further, this allows simple reconstruction of the hierarchy in the event that file servers go down or the directory db gets corrupted – since the file servers store these full file names, the hierarchy can always be reconstructed.

The directory server also provides a locking function. Since the token passed to each of its functions contains the name of the user that owns the token, locks can be stored simply as a filename and an owner. This allows only the owner of the lock to subsequently release the lock.

3. The Authentication Server

API

Function	Inputs	Outputs
CreateUser	?username=<username>&password=<password>	Bool (success)
Login	“	Either String (error) DfsToken (token)

Overview

The authentication server handles the management of users and login tokens. In an ideal world, these messages would be encrypted, rather than sent in cleartext, but hopefully for a proof of concept this suffices. Each token has an expiry date, 30 minutes into the future. After this point a user must log in again to use any part of the service.

4. The Common API

Overview

This file simply serves to define common data types, the APIs of each server, and some helper functions for MongoDB access. It probably stands as its own description.

5. The Client

Overview

Function	Description
createUser	Prompts for a username and password
login	As above, plus stores the resulting token for the session
openFile	Prompts for the filepath, displays the file

writeFile	Prompts for the filepath, allows the file contents to be entered directly, line by line, terminated by two blank lines.
uploadFile	Prompts for both a local and server filepath, uploads the file in the former to the latter
lockFile	Prompts for the path, displays whether successful or not
unlockFile	“
ls	Prompts for a path, lists the folders and files in that path

The client is also where caching takes place. The original idea was to use MongoDB's capped collections, which are effectively a circular buffer into which writes are made, with the oldest making way for the newest. However, with the latest version of MongoDB it is no longer possible to change the length of a document in a capped collection. This is a key component of the cache design, and so a regular collection is used instead. This has the unfortunate side effect that there is no upper bound on the amount stored in the cache. Again this was deemed unnecessary to fix in this project, but would need to be fixed in a real world application.