Distributed File System Project

Alexis Daboville

December 11, 2011

Introduction

There's many choice which should be done while designing a distributed file system (DFS). The first choice is which functionalities our DFS should provide, I decided to implement the following ones:

- Basic functionality (i.e. being able to open/read/write files within the DFS)
- A directory service (see Name server) which allow the users to access to the files without knowing where they are located.
- Caching on the client-side, chance are if a client access a file he will access it again.
- Locking, it's mandatory to avoid that multiple users modify the same file at the same time, and leave the file in a inconsistent state.

Even if this DFS doesn't implement things that would be necessary in a *true* DFS (understand a DFS used in production) it's a good idea to design a system as loosely coupled as possible to be able to extend the system easily in the future.

The source code may be found as annexe or online (HTTPS://GITHUB.COM/ALEXIS-D/DFS).

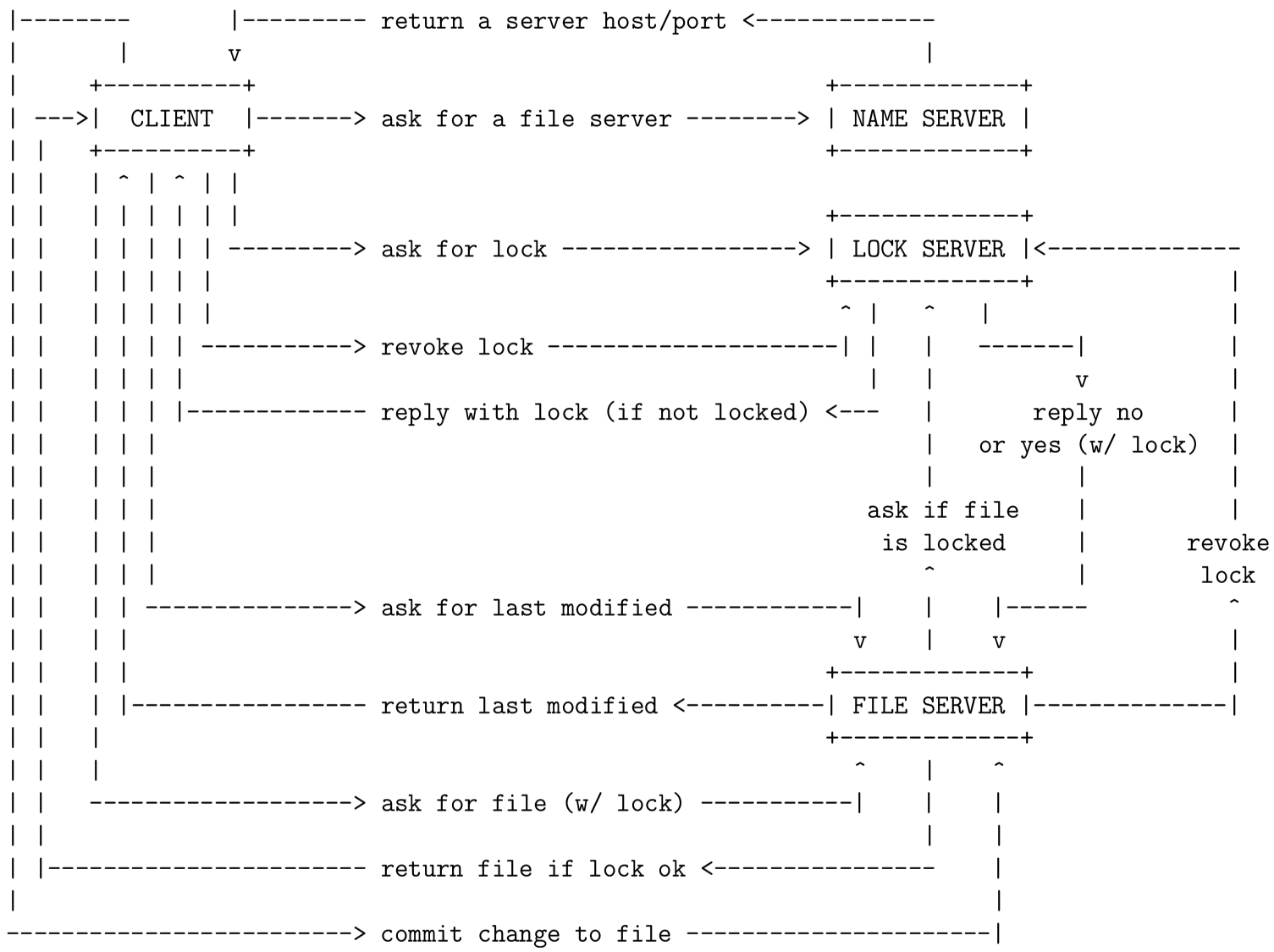
Overall Architecture

I choose to use an upload/download model over an NFS system because when you access a file you often need the whole file. As explained in The Google File System by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung this kind of decision highly depends on the use of the FS (if you are doing data mining, or you just store files you'll probably want an upload/download schema).

Another important choice was to use a REST API. I choose it because it's well known and use a common protocol (HTTP). Moreover using a REST API it's easy to separate each part of the DFS and also to develop new client API for different target languages. Also a REST API is easy to test: from the browser, using curl and so on.

This DFS was developed in Python 2, my first idea was to use Python 3 but as I use web.py (HTTP://WEBPY.ORG) which isn't compatible with Python 3 I had to change my initial plans. web.py is just used for the servers to handle all the routing stuff. Obviously we could have achieved the same thing using the standard module `BaseHTTPServer`, it would just have need a little bit more work.

The global design of the system is pretty simple, it's composed of a client, a name server, a lock server and one or more file servers. As a graph worth a thousand words here's one which describe pretty much everything about the DFS.


return a server host/port
CLIENT
ask for a file server
NAME SERVER
ask for lock
LOCK SERVER
revoke lock
reply with lock (if not locked)
reply no
or yes (w/ lock)
ask if file is locked
revoke lock
ask for last modified
FILE SERVER
return last modified
ask for file (w/ lock)
return file if lock ok
commit change to file

The servers and the client are configured through a simple JSON file, e.g. a file server may be configured like this:

```
{
    "nameserver": "localhost:8000",
    "lockserver": "localhost:8001",
    "srv": "localhost:8002", /* serve files on srv */
    "directories": ["/data", "/src/linux", "/src/vim"],
    "fsroot": "fs1/" /* the directory that's used as FS root */
}
```

A last thing to note about lock & name servers is that they store their state using the shelve module, it means that even if a server is killed, or crashed it may be restored in the same state as it was before crash.

Client API

One of the goal of the DFS was to provide a really easy to use API to the client; in consequence the API is the same as standard files, except that you need to use `dfs.client.open` instead of `open`. However the `dfs.client.open` propose a non-standard flag for the mode: 'c', which means that the file should be stored in the cache (even if the file is closed). A file stored in the cache may be retrieved through `dfs.client.File.from_cache` if the cache is outdated the method return `None`.

The object return by a call to `dfs.client.open` is a file which inherit from `tempfile.SpooledTemporaryFile` which may be used like a normal file. A great thing about `SpooledTemporaryFile` is that while it's not too big – the value is controlled by `max_size` in the client config file (`client.dfs.json`) – it's stored in the RAM and once its size exceed the maximum size it's stored on the disk (in `/tmp` by default on UNIX-like).

If a file is opened in 'a' or 'w' mode, the client API automatically request a lock. If a lock can't be granted, the user will see a `dfs.client.DFSIOError`.

An example use of the client API may look like this:

```
import dfs.client

filename = '/whatever/foo/bar'

with dfs.client.open(filename, 'wc') as f:
   f.write('hello, world')

f = dfs.File.from_cache(filename) # retrieve file from cache
f.read() # 'hello, world'
```

```
# dfs.client.rename(filename, '/foo/bar/baz')
dfs.client.unlink(filename) # delete the file
```

Name Server

The role of the name server is to map filenames to servers. When it starts it knows nothing about filenames & servers (except if it loads the last config file `names.db`). So how can it map filenames to servers? That's pretty easy, when the file servers are started the first thing they do is to contact the name server and tell which directory they served. So once this is done the name server is able to do is job.

A name server may be created like this (`names.py`):

```
#!/usr/bin/env python2
#-*- coding: utf-8 -*-

import web

import dfs.nameserver

urls = ('(/.*)', 'dfs.nameserver.NameServer',)

app = web.application(urls, globals())


if __name__ == '__main__':
    app.run()

# to launch it:
# $ chmod +x names.py
# $ ./names.py 8000 # launch server on localhost:8000
```

Lock Server

The role of the lock server is simply to grant/revoke lock on files, nothing really special. Each lock is associated with a random number (this is used by the client to show that he's the owner of the lock). Locks have a maximum lifetime defined by the `lock_lifetime` settings in `lockserver.dfs.json` so if a client lost it's connection or simply doesn't release the lock it may be invalidated if another client request a lock on the file at least `lock_lifetime` seconds after the creation of the lock. Obviously a lock may be used indefinitely if no one else request it.

The lock server is launched in the same fashion than the name server.

File Servers

The file servers are used to serve files (`GET`), create or modify files (`PUT`), or delete files (`DELETE`). Each request to a file server may be accompanied with an optional parameter `lock_id` which is used if the file is locked to be sure that the user is authorized to see/update/delete the file.

The file servers are launched in the same way than the name server.

Possible improvements

There's some obvious improvements that can be made:

- Authentication: this would use an authentication server
- Encryption: this could be coupled with the authentication the system
- Replications: this would require a few changes in the name server (to be able to map a filename to multiple servers) and also the file servers to handle the replication.
- Transactions: this would be easy to implement as the lock server is already able to grant multiple lock in one time (`POST` request on / with multiple filenames).
- Another possible improvement would be to use the `difflib` Python module to decrease the size of change that are send to the file servers (we would just upload the diff and not the whole file).

Conclusion

Although it's not perfect and not aimed to be used in a real-world environment this DFS works pretty well and does what we want.