# Scalable Computing Project – Distributed File System

## Introduction

This project involves the development of a distributed file system. The following components have been implemented

- Distributed File System
- Directory Server
- Locking Server
- Caching

## Requirements and Setup

The Project was written in Python 3.6 and Flask 0.12. All server side files are in the /dfs directory. In this directory we can start the three servers by issuing the commands:

```
python directory_server.py --host=[HOST_IP] --port=[PORT]
```

```
python file_server.py --host=[HOST_IP] --port=[PORT] --config=[some_config.json]
```

```
python locking_server.py --host=[HOST_IP] --port=[PORT]
```

A quick shell example of spinning these server up is provided in /tests/run.sh.

The clients examples are in the /test folder. The client imports the api.py from the /dfs directory to interface with the whole distributed file system.

1. Run up the directory server with host/port of your choice
2. Run up one or more files servers with host/ports of your choice. Each file server requires a .json config file where the directory server is defined (make sure it is the same as set before). In addition the serving directories of the file system are defined here as well.
3. Run up the locking server with host/port of your choice

Testing:

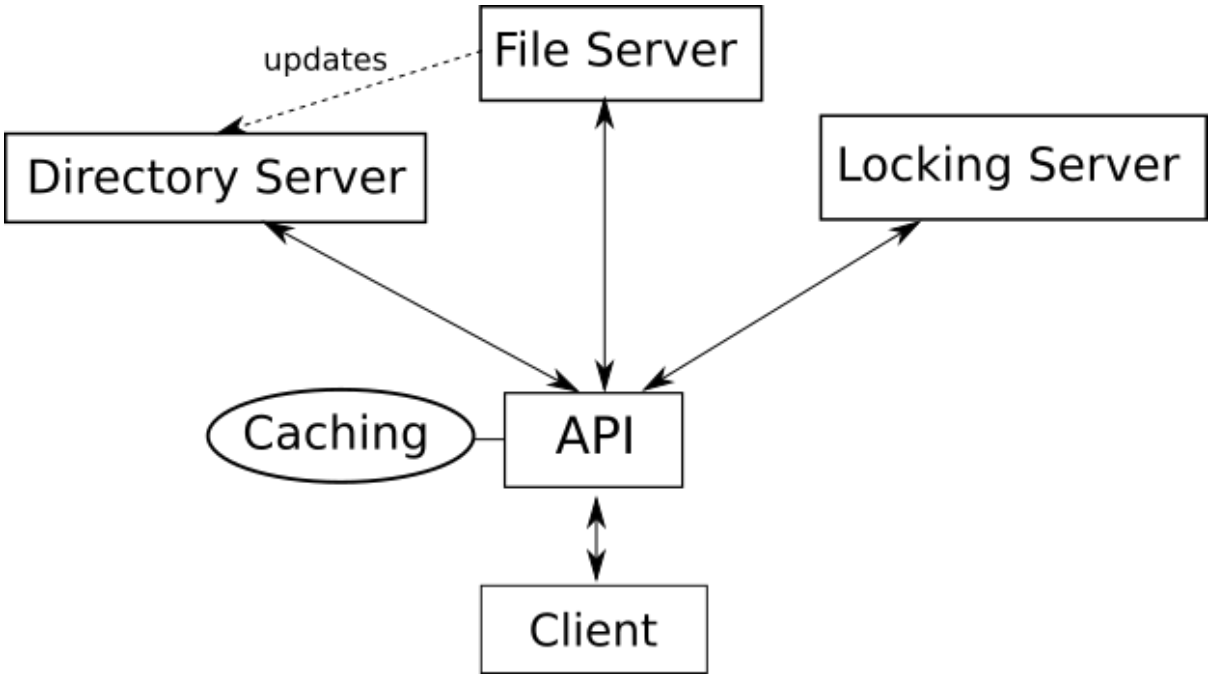
1. Execute client1.py, which tests read/write operations. In the same directory is a server.json config file which provides the client with the addresses of the directory and locking servers.

Remarks:

- The deployed servers require file read/write access to the disk of the node.

## Application Design

The following diagram shows an overview of the implementation:




- The Client interfaces with an API which provides the open, and close functions. A file can be opened in write or read mode. The client is unaware on which file server a new file will be written to.
- An AFS approach was implemented
- When a file server is started, it updates the directory server of the address and which directories it is serving. Otherwise all servers act independently and only communicate via the API.

## Technical Implementation

Client:

- The API exposes the `open()`. The function takes the file path and modes 'w' for write and 'r' for read. The API creates a temporary file in memory on which read and write operations can be applied. The copy is sent to the file server after the `close()` function is called.
- The API requests the address of the correct file server for the file path provided by the client.
- Caching is done on the client side. The client choses manually by `close(cached=True)` if the client wants a file to be cached before closing and pushing to the file system.
- The cached copy is automatically opened if it exists when using the `open()` function.
- When a cached copy is accessed, it checks if the copy is the most recent on in the file server. This creates extra overhead

File Server:

- The file server writes the files onto the disk of the node.
- Each file is written into a flat hierarchy under a root folder with a unique name
- The file server keeps track of the file path to the saved filed using a one-to-one mapping in the `fpath_to_fpathi` variable.

Locking Server:

- The locking server keeps track of all files that are currently locked
- For each file that is locked a queue is created with the client ID trying to access the file. If multiple users try to access the same file they are added to the queue and provided access once all previous clients have closed the file.
- If the locking server denies access to the client, it will keep polling the server until the lock is revoked.

## Examples

For the following examples import the random and dfs.api modules

```
import random
import dfs.api
```

- Writing Files:

  `Open()` function returns a temporary file object
  `write()` modifies the temporary object
  `close()` pushes changes to a file server

```
f1 = dfs.api.open('/etc/aag', 'w')
f1.write(str(random.randint(0, 100000)) * 5)
f1.close()
```

  With the file server output

```
File /etc/aag written to file server
```

- Reading Files:

  Similar to writing. `Read()` returns a string of the read file.

```
f2 = dfs.api.open('/etc/aag', 'r')
read_content = f2.read()
f2.close()
```

- Locking Service

  In client1.py It writes to a file but only closes it after set amount of time.

```
---------
testing locking
---------
writing to file '/etc/blub' without closing
File not in cache
Try to execute client2.py now, for simultaneous access
```

  Trying to access the file will show following output for client2 (client2.py):

```
writing to file '/etc/blub'
File not in cache
File /etc/blub is locked. Polling again
File /etc/blub is locked. Polling again
File /etc/blub is locked. Polling again
File /etc/blub is locked. Polling again
File /etc/blub is locked. Polling again
File /etc/blub is locked. Polling again
success
```

  Client2 tries to access a file currently being opened by client1. The locking server refuses the access until it is closed by client1.

  The output from the locking server prints current locking status of files. If requested it prints `Is /etc/blub locked for 7? True` while locked and `Is /etc/blub locked for 7? False` when lock has been lifted for that client.

- Caching

  Caching is done using `close(cached=True)` as follows. The default is set to False.

```
f = dfs.api.open('/etc/blub2', 'w')
f.write(str(random.randint(0, 100000)) * 1000)
f.write(str(random.randint(0, 100000)) * 1000)
f.write(str(random.randint(0, 100000)) * 1000)
f.close(cached=True)
```

  Printing the cache which maps file paths to local file objects shows our written file.

```
Current Cache after close:  {'/etc/blub2': <dfs.api.File object at 0x0000018B1227BD68>}
```

  Reading from the same file fetches now the cached version and prints:

```
Accessing cache directory...
Returning cached version.
```

  After t has checked with the file server if the cached version is the most up-to-date version on the file server.