

# Distributed File System

---

**Amber Higgins, M.A.I. Computer Engineering, 13327954**

A distributed Network File System (NFS) implementation with:

1. Distributed transparent file access
2. Directory service
3. Locking service
4. Client-side caching

Implemented in Python 2.7 using the Flask-Restful framework. All servers are run on *localhost* (hostname **127.0.0.1**).

This project is [available on Github](#) to the *demonstrators* user for the CS7NS1 course.

## Launch Instructions

---

The application can be run on either Windows or Linux environments. A number of shell scripts are provided for running this file-system in a **Linux environment**. Each of these will need to be given *execute permissions*, which can be assigned using the shell command '`chmod +x <script_name>`'. For ease, just **copy and paste the following into a terminal session** to give execute permissions to all scripts in the repository:

```
chmod +x install_dependencies.sh ; chmod +x launch_locking_server.sh ; chmod +x  
launch_directory_server.sh ; chmod +x launch_client.sh ; chmod +x launch_file_servers.sh
```

To launch the entire distributed file-system in the least time possible, the launch order 1-5 below should be followed.

1. **install\_dependencies.sh**

This script should be run first, since it will install all project dependencies specified in the *requirements.txt* file.

2. **launch\_directory\_server.sh**

This script launches a directory server on localhost, at URL <http://127.0.0.1:5000>.

3. **launch\_locking\_server.sh**

This script launches a locking server on localhost, at URL <http://127.0.0.1:5001>.

#### 4. **launch\_file\_servers.sh**

This script launches a number of file-servers on localhost, which will connect to the directory server if available. The number of file-servers should be specified by the user as **\$1**. Each file-server will then be launched on sequential ports, starting from port **45678**. This means that the first file-server to be launched will be available at <http://127.0.0.1:45678>.

**Note:** this script will also kill any existing processes running on these ports, before attempting to attach. This was deemed to be acceptable since this range of ports is not reserved for any specific application.

#### 5. **launch\_client.sh**

This script launches a client application, which will connect to the directory and locking servers if they are available at the URLs specified. Since the client requires user input, each client should be launched individually in a separate session.

## Dependencies

---

- Python 2.7.9
- flask
- flask-restful
- requests
- werkzeug

## Additional Notes

---

- Originally, I had implemented the fileserver communications with the client application using sockets. I was also starting to work on a locking server. However I realised after some time that the Restful approach was a lot simpler, abstracting away the low-level implementation details and allowing me to get on with the feature implementation.
- The commit at which I scrapped working with sockets can be found [here](#).

# Documentation

## Distributed Transparent File Access

---

The system operates according to the Network File System (NFS) model. It can support multiple connected clients, and multiple file-servers. It deals *exclusively* with .txt files.

### Client Library

All file accesses are made through a client library called a client library called ***ClientApi.py***. This library is the interface exposed to the client-side application for manipulation of the local and remote file-systems (also the cache: see later sections). By using the library, the under-the-hood implementation details of the distributed file-system are hidden from the client application.

Clients can:

- Access remote copies of files stored on file-servers through the *read* and *write* function calls. They can also create new files, and open existing files locally to view the contents.
- Interact with file contents through the system editor, which is launched before every remote write and after every remote read. In Windows, this is **notepad.exe**, and in Linux it is **nano** which is used.

**Note on opening files:** in order to 'open' a file in read-only mode, the contents of the file are displayed in the console window. Opening the file in the text editor (as is done with the *read* and *write* operations) would give an individual the option to edit the file when they should really just be able to view the contents - an undesirable behaviour that is resolved by this approach.

### Client application

This application, called ***Client.py***, provides a simple console-based UI offering options to manipulate files. This includes reading, writing, opening and creating of text files.

- In the background, the user's choices when interacting with the system are routed through the client library to the appropriate services (e.g. cache, locking server, directory server, file server), providing abstraction from the underlying technical implementation.

- When a text file is created for the first time, the specified directory is automatically created if it doesn't already exist. Error handling is included to prevent overwriting of existing file contents.
- All files on the local filesystem are visible to every active client using the system, meaning that concurrent accesses to files can easily occur. However, each client implements its own cache, allowing them to keep their own most frequently accessed content close at hand.
- Upon termination of the client, the contents of the cache are erased: however, no changes are made to the files in the local file-system.

## Directory Service

---

A directory service in any distributed file-system has the function of converting human-readable file names displayed to a client, into non-verbose file identifiers on remote file-servers. It acts as a namespace for the system that maps a unique filename (from which the client can be identified) to a unique file identifier (from which the remote file server can be identified) – an intermediary between the client and the remote file-servers.

## Directory Server

The system has one directory server located at a known address of <http://127.0.0.1:5000/>, called **DirectoryServer.py**. This server acts as a management application for the distributed file-system. It maintains a record of the mappings of full client-side filenames to enumerated file identifiers on the remote file-servers.

Amongst other responsibilities, it:

- Provides a registration mechanism for -
  - Client applications, at URL endpoint [http://127.0.0.1:5000/register\\_client](http://127.0.0.1:5000/register_client)
  - File-servers, at URL endpoint [http://127.0.0.1:5000/register\\_fileserver](http://127.0.0.1:5000/register_fileserver)

including the assignment of unique ids to each;

- Records connection details for clients and file-servers that have registered with it;
- Load-balances connected file-servers, such that at any time the least-loaded server will be given any new loading;

- Maintains records of client-fileserver file-directory mappings, where clients provide a full file path (e.g. ../Client0/hello.txt) and this is mapped to a unique server-side identifier in the server's root directory (e.g. ../Server10/18.txt).
- Maintains versioning of the files using integer values, according to the number of times they have been updated by clients. This is a crucial enabling element of the client-side caching. After a client-fileserver write-update has occurred successfully, the client updates the record of the file with the directory server at URL endpoint [http://127.0.0.1:5000/update\\_file\\_version](http://127.0.0.1:5000/update_file_version).

## Remote File Server

It is possible to have as many file servers as desired in this distributed file-system, since each file server registers with (and is subsequently managed by) the directory server when it is launched.

The file-servers are implemented as a flat-file system, with each storing files in a single directory. This directory uses the naming pattern **ServerX**, where X is an id assigned to the server when it registers with the directory server. All files stored on a file server follow a simple numerical naming system: for example, *0.txt* for the first file created on the server.

Each server accepts *get()* and *post()* requests from clients. It can be reached at any available host address and port specified by the user, which are provided as **sys.argv[1]** and **sys.argv[2]**. Each fileserver should be started on a *different port number*, and this mechanism is provided in the accompanying launch scripts.

- A client wishing to **read** a remote copy of a file will send a *get()* request. The client must provide JSON parameters:
  - 'file\_id': file\_id
  - 'file\_server\_id': server\_id
- A client wishing to **write** to a remote copy of a file will send a *post()* request. The client must provide JSON parameters:
  - 'file\_id': file\_id
  - 'file\_contents': file\_contents

### Notes:

- The fileserver does not hold any versioning information about the files that it stores: it is the directory server which handles this. Versioning is used as part of the caching mechanism.

- The client is able to identify the values of 'file\_id' and 'file\_server\_id' by consulting the directory server for the file mapping it requires.
- The fileserver, upon startup, also erases any previous contents of a root directory under the same name, e.g. if a directory called 'Server0/' exists from a previous run of the application, when a new Server0 is launched it will erase the contents of this file.

## Locking Service

---

A locking service provides concurrency control for multiple clients requiring access to the same files. It allows clients exclusive access to files under particular conditions.

A client must request and successfully acquire a single lock for a file in order to perform a restricted-access operation (such as a write). This means that for all writes in the distributed file-system, the request must first be routed through a **locking server** before access to the remote copy is granted.

## Locking Server

There is one locking server in this distributed file-system, called **LockingServer.py**. It manages the operation of locking and unlocking files as requested by clients. It is accessible at URL endpoint <http://127.0.0.1:5001/>. Any requests for a read or write on a remote file copy must first be routed through this service for approval. However, **only write-locks may be acquired**: there is no concept of a read-lock in this file-system.

Each file which exists on a remote fileserver will have a corresponding record with the locking server. This record exists as a lookup-table, where each entry is a [file\_id, is\_locked] pair.

The locking server also maintains a record of the clients that have registered with it. Clients must register with the locking server when they start up, connecting to URL endpoint [http://127.0.0.1:5001/register\\_new\\_client](http://127.0.0.1:5001/register_new_client). By doing this, it is ensured that:

- Only validated clients may lock or unlock a file, and not just anyone who can access the locking server URL;
- Since any client who is granted a lock will have their client\_id stored with the locking server's records for that file, we prevent a client who didn't lock the file originally, from releasing the lock on the file.

There are a number of other conditions that can occur in the locking and unlocking of a file in the distributed file-system. Some of these scenarios are as follows:

- A client wishing to write to a file must request the lock for that file. They will not be granted the lock until the file is no longer locked by another client. Until they are granted the lock, the requesting client essentially **polls** the locking server.
- A client wishing to read a file, will not be able to read it until it is unlocked. However, no read operation requires the acquisition of a lock - the file simply should not be readable until it is no longer locked by another client.
- To guard against infinite waiting for a lock to be released by a client, a safety mechanism in the form of a timeout (**60 seconds**) is used. After the timeout has elapsed, if the file is still locked, it will be assumed that the client who locked it has died and the lock is released by the server itself.

#### Notes:

- **Assumption:** when a remote copy is created for the first time on a fileserver, it doesn't need to be locked (nobody will try to concurrently access the file until after it has been created).

## Client-side Caching

---

A caching solution in a system allows for quicker access time to files if used effectively. It involves the local storage of up-to-date copies of files, so that if the remote copy of a file hasn't changed since the last time it was accessed, it can be read locally rather than requiring a request over the network. By using caching, the performance of a system can be enhanced since the servers are likely to have less loading at a given time. However, if not implemented appropriately on the client-side it can introduce unnecessary runtime overheads.

In general, it is most efficient to use caching on the client-side to reduce the number of relatively-slow calls required over the network – particularly in the case of file reads. Concerns such as cache invalidation for out-of-date copies of cached files, and an eviction policy in order to keep the contents of the cache relevant and small in number, are important to consider.

### Cache

A custom-implemented cache called **ClientCache.py** is used to provide client-side caching in this distributed file-system. An individual cache is created for each client

during initialisation: this allows each client to cache the files that they access most often both close at hand and up-to-date. The cache uses a look-up table with a maximum number of entries, whose contents are managed using (i) file versioning, and (ii) a Least-Recently-Used (LRU) cache eviction policy.

Operations implemented in the cache include: adding entries, updating entries, finding entries, evicting entries, and clearing the cache upon exit. Each cache entry looks similar to:

```
cache[file_id] = {  
    'file_contents': file_contents,  
    'file_version': file_version,  
    'file_name': file_name,  
    'timestamp' timestamp  
}
```

The timestamp is used to determine the age of the file, and the version is used to determine whether the entry is up-to-date with the remote copy at any given time.

The benefit of the cache is the reduction in volume of traffic going over the network, as well as more instantaneous access to files in many cases. As an NFS implementation, the benefits of this are most clearly seen when performing read operations.

1. **Read:** The cache is checked for an entry corresponding to the file to be read.
  - If there exists a corresponding entry, then the version is checked against that recorded with the directory server.
    - If the cache copy is of an outdated version of the file, then a request is made to the file-server on which the remote copy is stored to provide the most up-to-date copy of the file.
    - If the copy in the cache is up-to-date with the remote copy, the copy from the cache is used. This results in two fewer network accesses: one saving by not needing to request the file from the file-server, and another by the file-server not needing to service an extra request.
2. **Write:** The cache is updated after a successful write to a remote file copy.



- When a client writes to a file, the cache will not be updated until the remote copy has been updated. If it did, the cache copy and remote copy could easily become out of sync if the remote write failed.
  - Once the remote write has occurred successfully, a cache copy is either (i) created, or (ii) updated for that file in the particular client's cache.
    - If a cache copy is created, then an initial file-version, a timestamp and all of the specific file details are stored in a cache entry.
    - If a cache copy already existed, this cache copy is simply updated. This involves updating the version (compared to that previously recorded on the directory server), updating the timestamp to the current time, and updating the file contents with the new contents.
  - Thanks to the implementation of the locking service for remote write operations, concurrent updates to a remote file are sequenced such that any cache update will be correct at the time that its corresponding remote-write update occurs.
  - The updating of the version of the file on the directory server occurs during the write operation, before the updating of the cache. This ensures that any references that the cache makes to the remote copy's version, will be based on the latest update of record for that file on the directory server.
3. **Open:** since the file-system is implemented to replicate the NFS model, there are no calls across the network for the open operation, and the file as it exists in the cache is simply displayed in read-only mode. **Note on opening files:** in order to 'open' a file in read-only mode, the contents of the file are displayed in the console window. Opening the file in the text editor (as is done with the *read* and *write* operations) would give an individual the option to edit the file when they should really just be able to view the contents - an undesirable behaviour that is resolved by this approach.

**LRU Eviction Policy:** The Least-Recently Used cache eviction policy enforces a maximum number of cache entries at any particular time. This allows the cache to be indexed more quickly, and keeps the contents maximally relevant to the needs of the client it is associated with.

- In this implementation, **a maximum of 10 entries** can be stored in the cache at any given time. This empirically selected value could easily be increased or decreased depending on the requirements of this cache in practise - however, for this experimental design a small value is more than sufficient.

- Once the cache is full, the client will not be able to add another entry until the least-recently used entry has been evicted. The least-recently used entry is determined by the timestamp of the entry, which corresponds to the last time at which the cache entry was updated.

Upon termination of the client, the contents of the cache are erased. The copies of the corresponding files in the local file-system however, are persisted.

## Operation Examples

In order to best explain the operation of the distributed file-system in practise, some sequences of screenshots corresponding to different scenarios are provided below with brief explanations.

### Launch and Registration of Directory Server, Locking Server, Multiple Clients, Multiple Fileservers

- Client Output: Launching and registering two clients (Client0, Client1)

```

macneill.scss.tcd.ie - PuTTY
Cache0: Cache is full. Will evict LRU file.
Cache0: Oldest entry has key: 0
Cache0: Evicted LRU file from cache successfully.
Cache0: Added entry successfully at key 0

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

x
-----TERMINATING-----
Client0: Closing connection to server. Terminating the client.
Goodbye world!
Client0: Cleaning up after client(0)'s cache
Cache0: Clearing cache for client 0
amhiggin@macneill:~/Documents/DistributedFileSystem$ ./launch_client.sh
Client: Hello world from client!
Client0: Response to request for new client id: 0
Client0: Sending request to register client 0 with locking server..
Client0: Registered with locking server.
Cache: Setting up new cache.
Client0: New cache for client 0 created

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

macneill.scss.tcd.ie - PuTTY
copies of data, or unauthorised exports of data held on the system.

Authorised users must not disclose logon credentials to any third parties.
Third party access to this system must be facilitated and documented as
per the terms of the College Third party access agreement policy.
See http://www.tcd.ie/ITSecurity/policies for further information.

amhiggin@macneill.scss.tcd.ie's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Dec 15 09:43:19 2017 from lg35x23.scss.tcd.ie
amhiggin@macneill:~/Documents/DistributedFileSystem$ nano launch_file_servers.sh
amhiggin@macneill:~/Documents/DistributedFileSystem$ nano launch_file_servers.sh
amhiggin@macneill:~/Documents/DistributedFileSystem$ ./launch_client.sh
Client: Hello world from client!
Client1: Response to request for new client id: 1
Client1: Sending request to register client 1 with locking server..
Client1: Registered with locking server.
Cache: Setting up new cache.
Client1: New cache for client 1 created

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

```

- Fileserver Output: Launching and registering multiple filesevers (FileServer0, FileServer1)

```

amhiggin@macneill:~/Documents/DistributedFileSystem$ ./launch_file_servers.sh 2
* Running on http://127.0.0.1:45679/ (Press CTRL+C to quit)
* Restarting with stat
* Running on http://127.0.0.1:45678/ (Press CTRL+C to quit)
* Restarting with stat
FileServer: Hello, I'm a Fileserver! My address is 127.0.0.1:45679
FileServer: Connecting to directory server...
FileServer: Hello, I'm a Fileserver! My address is 127.0.0.1:45678
FileServer: Connecting to directory server...
FileServer: Connected.
FileServer0: Successfully registered as server 0
FileServer: Connected.
FileServer1: Successfully registered as server 1
FileServer1: Removed old Server1 directory
FileServer1: Created new Server1 root directory
FileServer0: Removed old Server0 directory
FileServer0: Created new Server0 root directory
* Debugger is active!
* Debugger PIN: 254-363-214
* Debugger is active!
* Debugger PIN: 254-363-214

```

### 3. Directory Server Output: Launching and registering multiple clients and file servers

```

amhiggin@macneill:~/Documents/DistributedFileSystem$ ./launch_directory_server.sh
Launching directory server on 127.0.0.1:5000
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 254-363-214
DirectoryServer: NEW FILE SERVER REGISTERED AT: 127.0.0.1:45678/ WITH ID 0
127.0.0.1 - - [15/Dec/2017 10:18:02] "POST /register_fileserver HTTP/1.1" 200 -
DirectoryServer: NEW FILE SERVER REGISTERED AT: 127.0.0.1:45679/ WITH ID 1
127.0.0.1 - - [15/Dec/2017 10:18:02] "POST /register_fileserver HTTP/1.1" 200 -
DirectoryServer: NEW CLIENT 0 REGISTERED
127.0.0.1 - - [15/Dec/2017 10:18:15] "GET /register_client HTTP/1.1" 200 -
DirectoryServer: NEW CLIENT 1 REGISTERED
127.0.0.1 - - [15/Dec/2017 10:20:43] "GET /register_client HTTP/1.1" 200 -

```

### 4. Locking Server Output: Launching and registering multiple clients

```

amhiggin@macneill:~/Documents/DistributedFileSystem$ ./launch_locking_server.sh
Launching locking server on 127.0.0.1:5001
* Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 254-363-214
LockingServer: RECEIVED REQUEST FROM CLIENT 0. REGISTERED SUCCESSFULLY.
127.0.0.1 - - [15/Dec/2017 10:18:15] "GET /register_new_client HTTP/1.1" 200 -
LockingServer: RECEIVED REQUEST FROM CLIENT 1. REGISTERED SUCCESSFULLY.
127.0.0.1 - - [15/Dec/2017 10:20:43] "GET /register_new_client HTTP/1.1" 200 -

```

## Creating a New File Locally

1. Client creating a new file successfully

```
-----  
Select option:  
1 = Read a file  
2 = Open a file  
3 = Write to a file  
4 = Create a new, empty file  
x = Kill client  
  
4  
  
Enter file path: Goodbye  
  
Enter file name at this path: goodbye.txt  
Directory Goodbye doesn't exist: do you want to create it now? (enter y/n): y  
Client0: Created directory Goodbye successfully.  
Client0: Created new file goodbye.txt successfully.  
-----
```

2. Incorrect attempt to create a non '.txt' file

```
-----  
Select option:  
1 = Read a file  
2 = Open a file  
3 = Write to a file  
4 = Create a new, empty file  
x = Kill client  
  
4  
  
Enter file path: Hello  
  
Enter file name at this path: hello.md  
Client0: Invalid file-name entered: must be a .txt file.  
Enter file name at this path: █
```

3. Incorrect attempt to re-create existing file

```
-----  
Select option:  
1 = Read a file  
2 = Open a file  
3 = Write to a file  
4 = Create a new, empty file  
x = Kill client  
  
4  
  
Enter file path: Hello  
  
Enter file name at this path: hello.txt  
Client0: File hello.txt already exists - will not overwrite contents.  
-----
```

## Request Remote Read with No Fileservers Launched

1. Client error message when no filesystems exist

```
-----  
Select option:  
1 = Read a file  
2 = Open a file  
3 = Write to a file  
4 = Create a new, empty file  
x = Kill client  
  
3  
  
Enter file path: Hello  
  
Enter file name at this path: hello.txt  
Client0: Writing to Hello/hello.txt  
Client0: Launching Linux system text editor  
Client0: Posting Hello/hello.txt to directory server and extracting mapping from  
response.  
Client0: There are no file servers registered with the directory server: cannot  
store remotely.  
Please register at least one filesystem!  
-----
```

2. Directory server error message when no filesystems exist

```

Launching directory server on 127.0.0.1:5000
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 254-363-214
DirectoryServer: NEW CLIENT 0 REGISTERED
127.0.0.1 - - [15/Dec/2017 09:41:49] "GET /register_client HTTP/1.1" 200 -
DirectoryServer: Request to post file Hello/hello.txt and return mapping.
DirectoryServer: File Hello/hello.txt isn't recorded in the directory server.
DirectoryServer: Remote copy of file Hello/hello.txt wasn't found on any server.
Will create a remote copy on the least-loaded server.
DirectoryServer: There are no file-servers registered - cannot service the load
balance request.
127.0.0.1 - - [15/Dec/2017 09:42:47] "POST / HTTP/1.1" 200 -

```

## Attempt to Read Non-Existent Remote Copy

1. Client application output

```

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

1

Enter file path: Hello

Enter file name at this path: hello.txt
Client0: Client requested to read Hello/hello.txt
Client0: Getting Hello/hello.txt mapping from directory server.
Client0: Hello/hello.txt doesn't exist as a remote copy.
-----

```

2. Directory server response to request

```

127.0.0.1 - - [15/Dec/2017 09:44:59] "POST /register_fileserver HTTP/1.1" 200 -
DirectoryServer: Request to get Hello/hello.txt file mapping.
DirectoryServer: File Hello/hello.txt isn't recorded in the directory server.
127.0.0.1 - - [15/Dec/2017 09:48:37] "GET / HTTP/1.1" 200 -

```

## Create New Remote File Copy (First Write)

1. Client input for writing to a new remote copy of file *Hello/hello.txt*

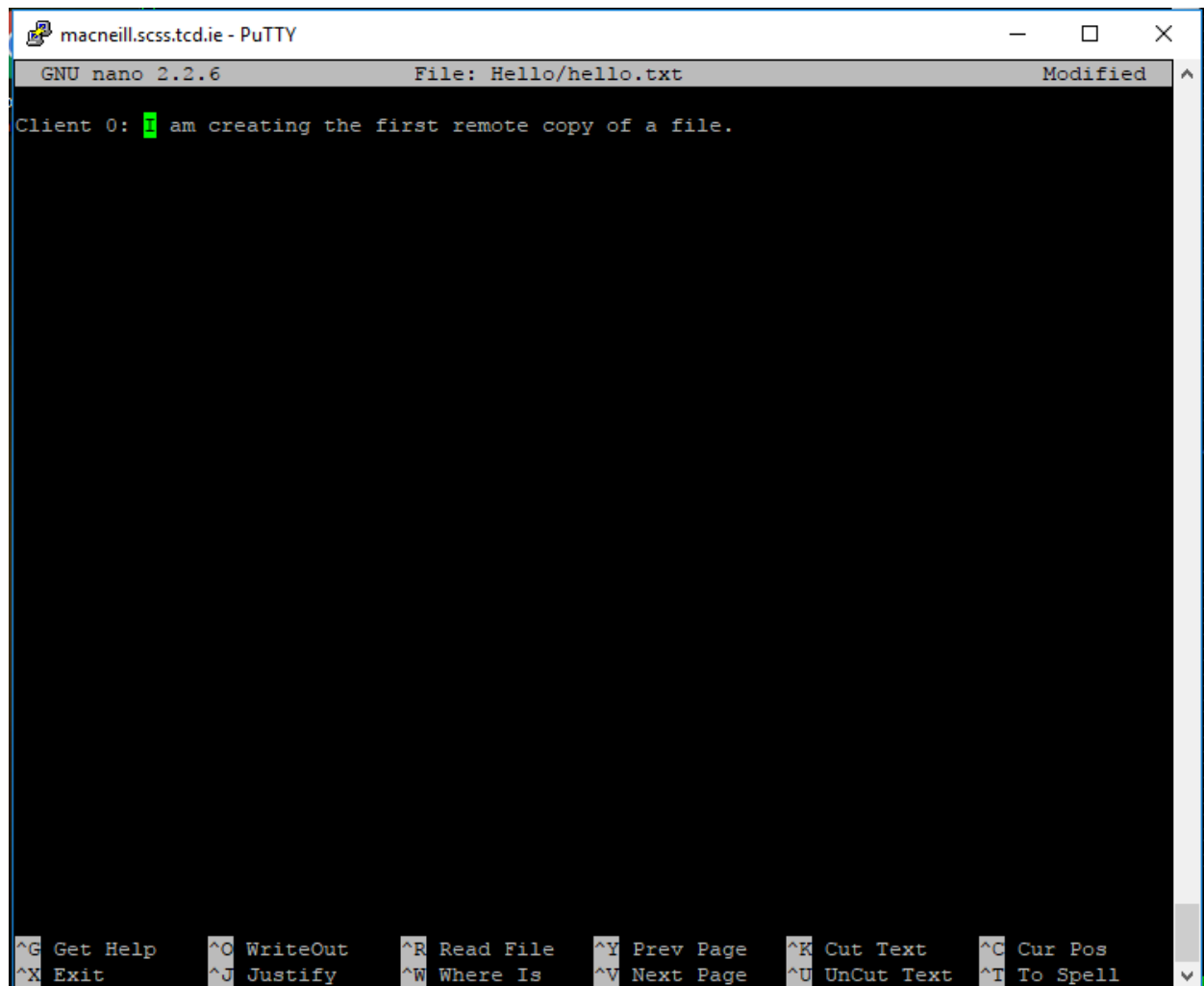
```
-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

3

Enter file path: Hello

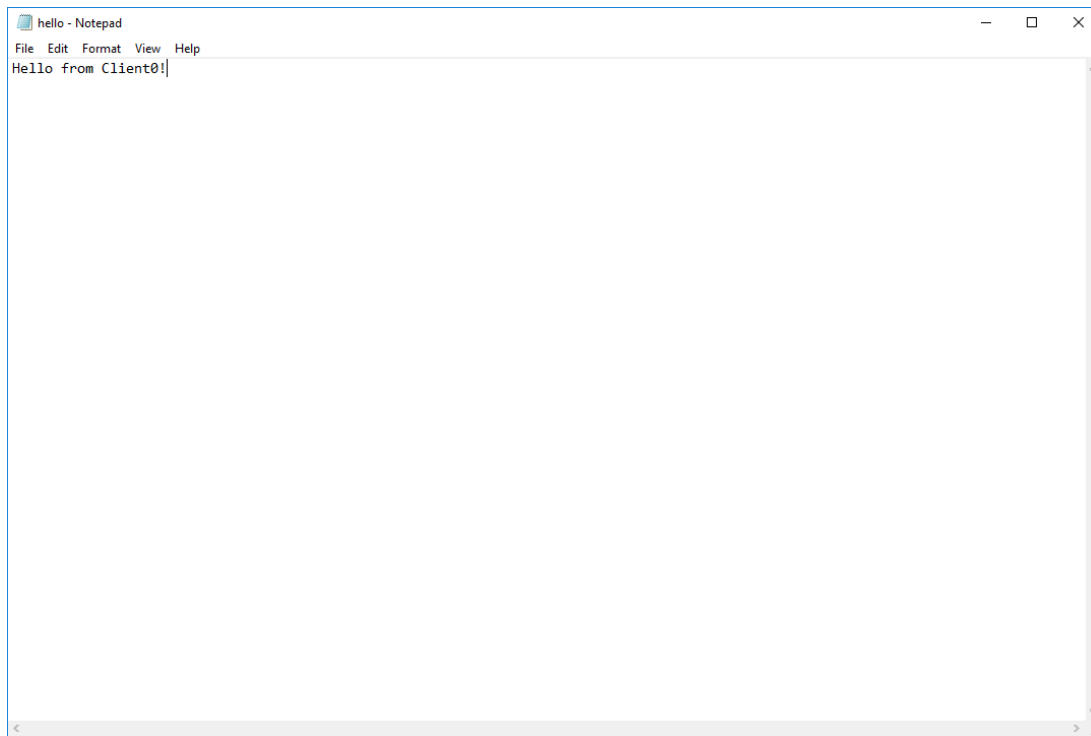
Enter file name at this path: hello.txt
Client0: Writing to Hello/hello.txt
Client0: Launching Linux system text editor
Client0: Posting Hello/hello.txt to directory server and extracting mapping from response.
Client0: Created new remote copy of Hello on file server 0.
Cache0: Adding/updating entry for Hello/hello.txt
Cache0: Hello/hello.txt does not exist in the cache. Adding to the cache.
Cache0: Added entry successfully at key 0
-----
```

## 2. Writing to *Hello/hello.txt* in Nano (Linux) and Notepad (Windows)



```
macneill.scss.tcd.ie - PuTTY
GNU nano 2.2.6      File: Hello/hello.txt      Modified
Client 0: I am creating the first remote copy of a file.

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```



### 3. Directory server output for creating new remote copy of file

```
DirectoryServer: Request to post file Hello/hello.txt and return mapping.  
DirectoryServer: File Hello/hello.txt isn't recorded in the directory server.  
DirectoryServer: Remote copy of file Hello/hello.txt wasn't found on any server.  
Will create a remote copy on the least-loaded server.  
DirectoryServer: The least loaded file server is 0.  
DirectoryServer: Successfully created remote copy.  
127.0.0.1 - - [15/Dec/2017 09:52:23] "POST / HTTP/1.1" 200 -
```

### 4. File server output for creating new remote copy of file

```
FileServer0: Request received to create new file Server0/0.txt  
FileServer0: Remote copy of file id 0 successfully created.  
127.0.0.1 - - [15/Dec/2017 09:52:23] "POST /create_new_remote_copy HTTP/1.1" 200 -
```

## Reading Up-to-Date Cached Copy of File

### 1. Client reading up-to-date cache copy of file *Hello/hello.txt*



```
-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

1

Enter file path: Hello

Enter file name at this path: hello.txt
Client0: Client requested to read Hello/hello.txt
Client0: Getting Hello/hello.txt mapping from directory server.
Cache0: File Hello/hello.txt is cached
Cache0: File Hello/hello.txt is up-to-date with remote copy
Cache0: Fetching cache entry.
Client0: Opening local copy to update with response contents: Just created a client. testing to
see what happens when there are no connected filesystems.
Now, the client will attempt to write this file to the file server.

Client0: Launching Linux system text editor
-----
```

## 2. Directory Server response to request for version of file *Hello/hello.txt*

```
DirectoryServer: Request to get Hello/hello.txt file mapping.
DirectoryServer: Hello/hello.txt on record
DirectoryServer: The file 0 (with version 0) is stored on file-server 0. The cor
responding server address is 127.0.0.1:45679
127.0.0.1 - - [15/Dec/2017 09:54:52] "GET / HTTP/1.1" 200 -
```

## Writing to Existing Remote Copy

### 1. Client output when writing to existing remote copy

```

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

3

Enter file path: Hello

Enter file name at this path: hello.txt
Client0: Writing to Hello/hello.txt
Client0: Launching Linux system text editor
Client0: Posting Hello/hello.txt to directory server and extracting mapping from response.
Client0: Updating existing remote copy of Hello/hello.txt with new changes...
Client0: Client0 has locked file 0
Client0: Updated version on directory server successfully.
Client0: File 0 has been unlocked
Cache0: Adding/updating entry for Hello/hello.txt
Cache0: Updated entry for Hello/hello.txt successfully.
-----

```

## 2. Directory server response to request for remote copy mapping

```

DirectoryServer: Request to post file Hello/hello.txt and return mapping.
DirectoryServer: Hello/hello.txt on record
DirectoryServer: The file 0 (with version 0) is stored on file-server 0. The corresponding server address is 127.0.0.1:45679
DirectoryServer: A remote copy of Hello/hello.txt with version 0, exists.
127.0.0.1 - - [15/Dec/2017 09:58:36] "POST / HTTP/1.1" 200 -
Successfully updated version of Hello/hello.txt to version = 1
127.0.0.1 - - [15/Dec/2017 09:58:36] "POST /update_file_version HTTP/1.1" 200 -

```

## 3. Locking server allowing lock of remote copy for writing

```

LockingServer: Client 0 has successfully locked file 0
127.0.0.1 - - [15/Dec/2017 09:58:36] "PUT / HTTP/1.1" 200 -
LockingServer: Client 0 has unlocked file 0
127.0.0.1 - - [15/Dec/2017 09:58:36] "DELETE / HTTP/1.1" 200 -

```

## 4. Fileserver performing update on remote copy

```

FileServer0: Wrote update to file 0 as requested.
127.0.0.1 - - [15/Dec/2017 09:58:36] "POST / HTTP/1.1" 200 -

```

# Load-Balancing Multiple File Servers

## 1. Directory Server Output: Load-Balancing based on least-loading

```
DirectoryServer: Request to post file Hello/hello.txt and return mapping.
DirectoryServer: File Hello/hello.txt isn't recorded in the directory server.
DirectoryServer: Remote copy of file Hello/hello.txt wasn't found on any server.
Will create a remote copy on the least-loaded server.
DirectoryServer: The least loaded file server is 0.
DirectoryServer: Successfully created remote copy.
127.0.0.1 - - [15/Dec/2017 10:25:16] "POST / HTTP/1.1" 200 -
DirectoryServer: Request to post file Goodbye/goodbye.txt and return mapping.
DirectoryServer: File Goodbye/goodbye.txt isn't recorded in the directory server.
DirectoryServer: Remote copy of file Goodbye/goodbye.txt wasn't found on any server. Will create a remote copy on the least-loaded server.
DirectoryServer: The least loaded file server is 1.
DirectoryServer: Successfully created remote copy.
127.0.0.1 - - [15/Dec/2017 10:26:01] "POST / HTTP/1.1" 200 -
```

## Timeout of File Lock

1. Client2 places write-lock on file *Hello/hello.txt* and never releases it

```
-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

3

Enter file path: Hello

Enter file name at this path: hello.txt
Client2: Writing to Hello/hello.txt
Client2: Launching Linux system text editor
Client2: Posting Hello/hello.txt to directory server and extracting mapping from response.
Client2: Updating existing remote copy of Hello/hello.txt with new changes...
Client2: Client2 has locked file 0
```

2. Client0 requests and obtains write-lock on file *Hello/hello.txt*

```

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

3

Enter file path: Hello

Enter file name at this path: hello.txt
Client0: Writing to Hello/hello.txt
Client0: Launching Linux system text editor
Client0: Posting Hello/hello.txt to directory server and extracting mapping from response.
Client0: Updating existing remote copy of Hello/hello.txt with new changes...
Client0: Client0 has locked file 0
Client0: Updated version on directory server successfully.
Client0: File 0 has been unlocked
Cache0: Adding/updating entry for Hello/hello.txt
Cache0: Updated entry for Hello/hello.txt successfully.

```

### 3. Locking server releases lock on *Hello/hello.txt* after timeout elapses

```

LockingServer: RECEIVED REQUEST FROM CLIENT 0. REGISTERED SUCCESSFULLY.
127.0.0.1 - - [15/Dec/2017 12:51:00] "GET /register_new_client HTTP/1.1" 200 -
LockingServer: RECEIVED REQUEST FROM CLIENT 1. REGISTERED SUCCESSFULLY.
127.0.0.1 - - [15/Dec/2017 12:51:44] "GET /register_new_client HTTP/1.1" 200 -
LockingServer: Client 1 has successfully locked file 0
127.0.0.1 - - [15/Dec/2017 12:54:15] "PUT / HTTP/1.1" 200 -
LockingServer: Client 1 has unlocked file 0
127.0.0.1 - - [15/Dec/2017 12:54:15] "DELETE / HTTP/1.1" 200 -
LockingServer: File 0 is not locked
127.0.0.1 - - [15/Dec/2017 12:54:36] "GET / HTTP/1.1" 200 -
LockingServer: RECEIVED REQUEST FROM CLIENT 2. REGISTERED SUCCESSFULLY.
127.0.0.1 - - [15/Dec/2017 12:58:07] "GET /register_new_client HTTP/1.1" 200 -
LockingServer: Client 2 has successfully locked file 0
127.0.0.1 - - [15/Dec/2017 12:58:48] "PUT / HTTP/1.1" 200 -
LockingServer: Timeout reached on file 0: unlocking
LockingServer: Client 0 has successfully locked file 0
127.0.0.1 - - [15/Dec/2017 13:00:32] "PUT / HTTP/1.1" 200 -
LockingServer: Client 0 has unlocked file 0
127.0.0.1 - - [15/Dec/2017 13:00:32] "DELETE / HTTP/1.1" 200 -

```

## Invalidation of File's Cache Copy

1. Client1 performing a write-update on remote copy of *Hello/hello.txt* created by Client0

```

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

3

Enter file path: Hello

Enter file name at this path: hello.txt
Client1: Writing to Hello/hello.txt
Client1: Launching Linux system text editor
Client1: Posting Hello/hello.txt to directory server and extracting mapping from response.
Client1: Updating existing remote copy of Hello/hello.txt with new changes...
Client1: Client1 has locked file 0
Client1: Updated version on directory server successfully.
Client1: File 0 has been unlocked
Cache1: Adding/updating entry for Hello/hello.txt
Cache1: Hello/hello.txt does not exist in the cache. Adding to the cache.
Cache1: Added entry successfully at key 1
-----

```

## 2. Client0 invalidating and updating its cache copy of *Hello/hello.txt* on a read

```

-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

1

Enter file path: Hello

Enter file name at this path: hello.txt
Client0: Client requested to read Hello/hello.txt
Client0: Getting Hello/hello.txt mapping from directory server.
Cache0: File Hello/hello.txt is cached
Cache0: File Hello/hello.txt is NOT up-to-date with remote copy
Client0: File 0 isn't locked
Client0: Updating file with response contents...
Cache0: Adding/updating entry for Hello/hello.txt
Cache0: Updated entry for Hello/hello.txt successfully.
Client0: Launching Linux system text editor
-----

```

## LRU Cache Eviction

### 1. Client console output for cache eviction

```
-----
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

3

Enter file path: Assignments

Enter file name at this path: distributed_file_system.txt
Client0: Writing to Assignments/distributed_file_system.txt
Client0: File Assignments/distributed_file_system.txt does not exist for writing: will create a
new empty file locally.
Client0: Created new file distributed_file_system.txt successfully.
Client0: Launching Linux system text editor
Client0: Posting Assignments/distributed_file_system.txt to directory server and extracting mapp
ing from response.
Client0: Created new remote copy of Assignments on file server 0.
Cache0: Adding/updating entry for Assignments/distributed_file_system.txt
Cache0: Assignments/distributed_file_system.txt does not exist in the cache. Adding to the cache
.
Cache0: Cache is full. Will evict LRU file.
Cache0: Oldest entry has key: 0
Cache0: Evicted LRU file from cache successfully.
Cache0: Added entry successfully at key 0
-----
```

## Termination of Client Application

1. Console output showing the termination of a client

```
Select option:
1 = Read a file
2 = Open a file
3 = Write to a file
4 = Create a new, empty file
x = Kill client

x

-----TERMINATING-----
Client0: Closing connection to server. Terminating the client.
Goodbye world!
Client0: Cleaning up after client0's cache
Cache0: Clearing cache for client 0
amhiggin@macneill:~/Documents/DistributedFileSystem$
```