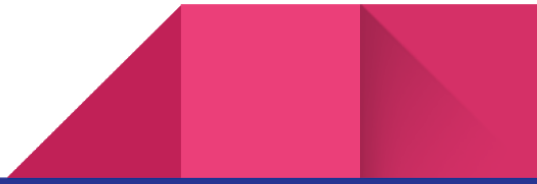
Maxwell Lincoln Dantas da Silva

Creating a multi-threaded TCP server

Cruz das almas - BA

2020

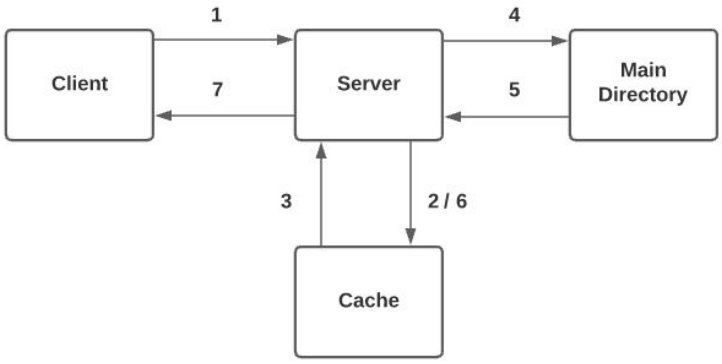


Introduction

In this work a TCP client and a multi-threaded TCP server were implemented using python 3. The client is able to request files from the server and the server is able to send files to the client. Also, the client can consult a list of all files in the server's cache.

Architecture Details

The communication between the client and the server when transferring a file follows the steps below:



1. The client requests a file from the server
2. The server checks whether the file is in the cache
3. If the file is in the cache, prepare the file to be sent to the client and skip to seven step
4. If the file is not in the cache, search the main directory
5. If the file exists, brings the file from the directory

6. If the total file size is less than the maximum cache size (64MB), add the file to the cache
7. sends the file to the client

Note: The Main Directory is defined through the server input parameters

Implementation Details

The system is divided into 2 files:

- client.py -> contains the client code
- server.py -> contains the server code

The client can perform two actions, request a file from the server or request a list of files in the server's cache. The type of action that will be taken depends on the parameters passed on the command line. when the server starts it listens on a port waiting for client connections, all communication is done through sockets.

Cache Implementation

When starting the server, a folder called 'cache' will be created in the same directory as 'server.py'. If the folder already exists, it will be deleted and a new empty 'cache' will be created. This folder is the server's cache and the sum of the size of your files should never exceed 64MB.

Cache Replacement Policy

The replacement policy chosen was FIFO (First In First Out). Every time a file smaller than 64MB is requested by the client it will be stored in the cache, if there is not enough space the files will be removed one by one until there is space for the requested file. The removal order is always from the oldest file to the newest file, files larger than 64MB will never be stored in the cache. Although the FIFO policy is not very efficient, it was chosen for its easy implementation.

Concurrency Problems

As the server is multi-threaded, concurrency problems must be addressed. For this, some tools from the python threading library were used. First a lock object was created, this object has access to 2 important functions: acquire() and release(), they are used to manage parallel processes. A lock has two states, "locked" or "unlocked", when the state is unlocked lock.acquire() is used to change state to locked and lock.release() is used to change state to unlock.

The big problem with the multi-threaded server is that it can allow multiple parallel processes to manipulate the cache at the same time, if a control is not done, information can be lost.

As soon as a new connection is created, the acquire() method is invoked by the lock object, the possible conditions for a release() to be invoked are listed below:

- **The client requested listing of files in the cache**
- **The client requested a file larger than 64MB**
- **The client requested a file that can be stored in the cache without having to delete any other files (filesize < cacheAvailableMemory)**

If none of these cases occur, the release() will only be invoked when the current process has already taken the file from the main directory, stored in the cache and sent to the client. To test this functionality, some sleep methods were added to the code

Type of Socket Used

SOCK_STREAM: The connection is established and the two parties have a conversation until the connection is terminated by one of the parties or by a network error.

Note: More implementation details can be found in the code comments