A TCP multi-thread client-fileserver

Student: Jordan Silva Oliveira

Professor: Ramon Pereira Lopes

Course: Distributed Systems

University: UFRB

Description

The objective of this project was to build a TCP server that should have at least two functionalities:

1. As asked by the client send a list of all files present inside its cache memory;
2. As asked by the client send a file present inside of its cache memory or once its doesn't exist on cache, the file should be sended from a specifc folder and then loaded to the cache memory, respecting its max size of 64MB(where this value should not be surpassed, being the programmers task to limit and make sure space is available before the file be loaded to the cache).

   Some other requists where asked, but above are listed the most important ones.

Project Structure

```
.
├── client
│   ├── client_data
│   │   ├── cat.jpg
│   │   ├── file1.txt
│   │   └── video.mp4
│   └── tcp_client.py
├── README.md
├── REQUISTOS.md
└── server
    ├── server_data
    │   ├── bigFile.txt
    │   ├── cat.jpg
    │   ├── file1.txt
    │   ├── file2.txt
    │   ├── file3.txt
    │   ├── file4.txt
    │   ├── file5.txt
    │   ├── file6.txt
    │   ├── file7.txt
    │   ├── file8.txt
    │   ├── music.mp3
    │   ├── texto.txt
    │   └── video.mp4
    └── tcp_server.py
```

The project has the above structure, two main folders client and server where its names are self explanatory in the context of this project. Inside of each folder we have an folder where the data of either the server (to be sended or stored in cache) or the client (to be received after being requested), aside of those folders there are the .py file where the code actualy resides.

Server

We have bellow the code of the server.py file:

```python
import os
import socket
import threading
import sys
from threading import Thread

# List of constants
IP = "localhost"
PORT = 4457 if sys.argv[1] == "" else int(sys.argv[1])
ADDR = (IP, PORT)
SIZE = 1024
FORMAT = "utf-8"
SERVER_DATA_PATH = "server_data"
BUFFER_SIZE = 4096
# constant with the max size of the cache memory, this value in bytes is
equivalent to 64mb
CACHE_MAXSIZE = 67108864

# Cache memory is created as a python dictionary, where the key is the filename
and the value is the data of the file
cache = dict()

# Current size of the cache memory
cache_currentsize: int


def handle_client(conn, addr, lock):
    print(f"[NEW CONNECTION] {addr} connected.")
    conn.send("OK@Welcome to the File Server.".encode(FORMAT))

    while True:
        data = conn.recv(SIZE).decode(FORMAT)
        data = data.split("@")
        cmd = data[0]

        # List of all files inside the server_data directory
        files = os.listdir(SERVER_DATA_PATH)

        if cmd == "list":
            send_data = "OK@"

            if len(files) == 0:
                send_data += "The server directory is empty"
            else:
                send_data += "\n[FILES INSIDE THE SERVER DIRECTORY]:\n"
                send_data += "\n".join(f for f in files)
```

```
                send_data += "\n[FILES IN CACHE]:\n"
                send_data += "\n".join(k for k in cache.keys())

            conn.send(send_data.encode(FORMAT))
            break

        elif cmd == "help":
            data = "OK@"
            data += "[HELP]"
            data += "list: List all the files from the server (on cache and on
a server local directory).\n"
            data += "help: List all the commands.\n"
            data += "file: [HOST](default is localhost) [PORT](default is 4457)
file [FILENAME] (to request a file from the server).\n"

            conn.send(data.encode(FORMAT))
            break

        elif cmd == "file":
            filename = data[1]
            if filename in files or filename in cache:
                conn.send(("OK@").encode(FORMAT))
                if filename in cache:
                    for data in cache[filename]:
                        conn.send(data)

                    print("file sent from cache")
                    break
                else:
                    lock.acquire()
                    with open(os.path.join(SERVER_DATA_PATH, filename), 'rb')
as file:
                        while True:
                            bytes_read = file.read(BUFFER_SIZE)
                            if not bytes_read:
                                break
                            conn.send(bytes_read)
                    print("file sent")

                    with open(os.path.join(SERVER_DATA_PATH, filename), 'rb')
as file:
                        data = file.readlines()
                        dataSize = Size(data)

                        cache_currentsize = cacheCurrentSize(cache)

                        keys = []
                        if not dataSize > CACHE_MAXSIZE:
```

```
                            while (cache_currentsize+dataSize) > CACHE_MAXSIZE:
                                for k in cache:
                                    keys.append(k)

                                cache_currentsize -= Size(cache[keys[0]])
                                cache.pop(keys[0])
                                keys.pop(0)

                            cache.update({filename: data})

                            print("Adding file to cache")
                        lock.release()
                        break
                else:
                    conn.send(("FNF@").encode(FORMAT))
                    break

    print(f"[DISCONNECTED] {addr} disconnected")
    conn.close()


def cacheCurrentSize(cache: dict):
    size = 0
    for key in cache:
        size += Size(cache[key])

    return size


def Size(data):
    size = 0
    for line in data:
        size += line.__sizeof__()

    return size


def main():
    print("[STARTING] Server is starting")
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(ADDR)
    server.listen()
    lock = threading.Semaphore()
    print(f"[LISTENING] Server is listening on {IP}:{PORT}.")
    while True:
        conn, addr = server.accept()
        thread = threading.Thread(
            target=handle_client, args=(conn, addr, lock))
```

```
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.active_count() - 1}")


if __name__ == "__main__":
    main()
```

The main function starts the server through the method socket from the socket library, them the bind method is called receiving the ADDR tuple formed by the values from the IP and PORT constants declared on the top of the code. The server then is started by the listen method, listening on the localhost and the defined port. When running the file, the user can choose to pass an custom port (making sure its available), if no port is passed as an argument the server will use the 4457 port as default. As the title of this report says, this is a multi-thread server, so it should be capable of receiving multiple connections, and this is what the while loop does, it accepts new connections through the accept method and starts a new thread passing the conn, addr and lock variables as argumentd and the *handle_client* function as target.

The *handle_client* function is responsible for handling the clients requests, such as return a list of all the files in cache and inside the directory and actually return the files when they are requested. The control of the cache memory is done in the block between the lines 82 and 102, before we talk about what this block actually does, we should first talk about two auxilliary functions *Size* and *cacheCurrentSize*, where *Size* receives an array of data and returns its total size in bytes and *cacheCurrentSize* receives an dictionary (in this case the cache memory) and returns is size in bytes, now that those two are explained we can talk about the cache control. First of all the file is readed and stored inside and viariable, then its size is measured through the *Size* function and saved in another variable, then the same is done to the current size of the cache memory using the *cacheCurrentSize* function, after that we created the keys list, later the keys of the cache will be stored on if, so we can iterate throught the files and manipulate them. The next step is actually compare the size of the file with the size of the cache, if the file is bigger then it can't be stored and the program skips the rest of the block, if thats not the case, then we now compare the size of the file plus the size of cache memory already in use, if its bigger then we need to clean the cache until the file can be stored inside it, and to do this we choosed to use the FIFO(First in/First out) strategy where the first to have been stored is removed first and so on, by doing that we make sure to have in cache the most recent file to be requested, we could choose to use other techniques like the LRU(Least Recently Used) but to this project this could be seen as an overcomplication, since the python decorator didn't works so well when you need to limit the cache by its size in bytes and not by the ammount of files on it.

As a plus an help functionality was implemented, where the client can send the "help" word as an argument like in the example bellow:

```
python tcp_client.py localhost 3000 help
```

And receive an message with the accepted commands and a little description for each one.

Client

```python
import socket
import os
import sys

IP = sys.argv[1]
PORT = int(sys.argv[2])
ADDR = (IP, PORT)
FORMAT = "utf-8"
SIZE = 1024
CLIENT_DATA_PATH = "client_data"
BUFFER_SIZE = 4096


def main():
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(ADDR)

    data = client.recv(SIZE).decode(FORMAT)
    res, msg = data.split("@")

    if res == "DISCONNECTED":
        print(f"[SERVER]: {msg}")

    elif res == "OK":
        print(f"{msg}")

    cmd = sys.argv[3]

    if cmd == "help":
        client.send(cmd.encode(FORMAT))
        data = client.recv(SIZE).decode(FORMAT)
        res, msg = data.split("@")
        print(msg)

    elif cmd == "list":
        client.send(cmd.encode(FORMAT))
        data = client.recv(SIZE).decode(FORMAT)
        res, msg = data.split("@")
        print(msg)

    elif cmd == "file":
        filename = sys.argv[4]
        send_data = f"{cmd}@{filename}"
        client.send(send_data.encode(FORMAT))
        asn = client.recv(SIZE).decode(FORMAT)
        if asn == "FNF":
```

```
            print("File not found!")
        else:
            with open(os.path.join(CLIENT_DATA_PATH, filename), 'wb') as file:
                while True:
                    filedata = client.recv(BUFFER_SIZE)
                    if not filedata:
                        break
                    file.write(filedata)
            print(f"File {filename} received")

    else:
        print("[ERROR - invalid command]")

    client.close()
    print("[Disconnected from the server.]")


if __name__ == "__main__":
    main()
```

In the code block above we have the client code, first of all the sys library helps us to receive the arguments that are passed on the run of the .py file, bellow we have a little example of how to properly start this code:

```
#To request an file
python tcp_client.py localhost [port] file [filename]


#To resquest the list of files
python tcp_client.py localhost [port] list


#To request the help functionality
python tcp_client.py localhost [port] help
```

Have in mind that the arguments between brackets must be replaced by properly ones that matche those passed and present on the server.

Final Considerations

This project was built using python 3.10.4 in a Fedora linux 64bit machine, so should work on any higher version of the language and probably any other SO with python installed.

The project is public and was an part of the course of Distributed Systems, in the Federal University of the Recôncavo of Bahia (Universidade Federal do Recôncavo da Bahia - Brasil), so feel free to clone and imporove it if you fell like.

Thanks for reading.