Peer-to-peer networks for file exchange

Abstract

I this paper, I present you an implementation of a peer to peer (P2P) network for file exchange between users. All the nodes were realised locally using flask and a set of port numbers. To analyse this implementation, I will first focus on the different aspects of a peer-to-peer communication as designed by the subject, how the file exchange procedure works and then how the exchanged data was encrypted. Finally, I will adopt a critical point of view on the obtained results to discuss what could be improved and conclude of the effectiveness of this secure communication

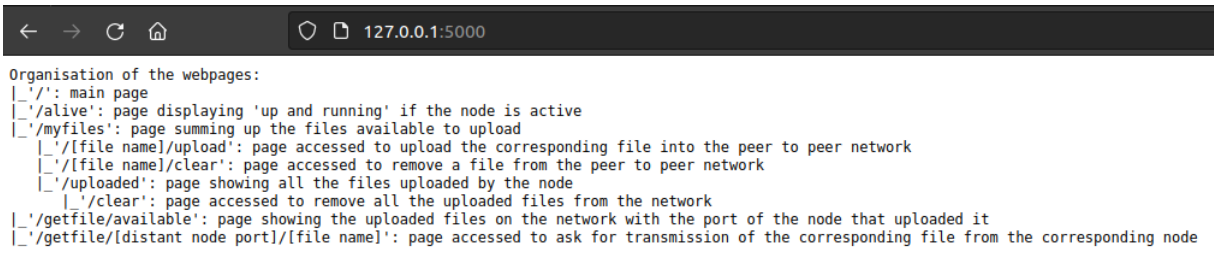
1. Introduction

Either for professional or personal use, a P2P mesh network seems to be the easiest and fastest way to share files with others in the same local network. But this kind of networks presents major threats for the user, one being the protection of the data that passes over the medium. For this reason, files must be encrypted during the exchange and decrypted only upon reception by the distant node. Thus, the objective for this assignment was to first create a working file exchange system on a P2P network simulated with flask to then encrypt the exchange using the encryption method realised in the previous assignment.

2. Design and Implementation

2.1 Creating the P2P network

As the objective in this assignment is to permit users exchanging, I had to keep in mind that the user interacts with the system through the webpages he can access. To make it user-friendly, I made the organisation of the accessible pages as clear as possible. Furthermore, on the home page in my flask server (meaning the one at the address: '127.0.0.1:[port number]/') I decided to make an organisational diagram for the user to learn how to access the information he needs.



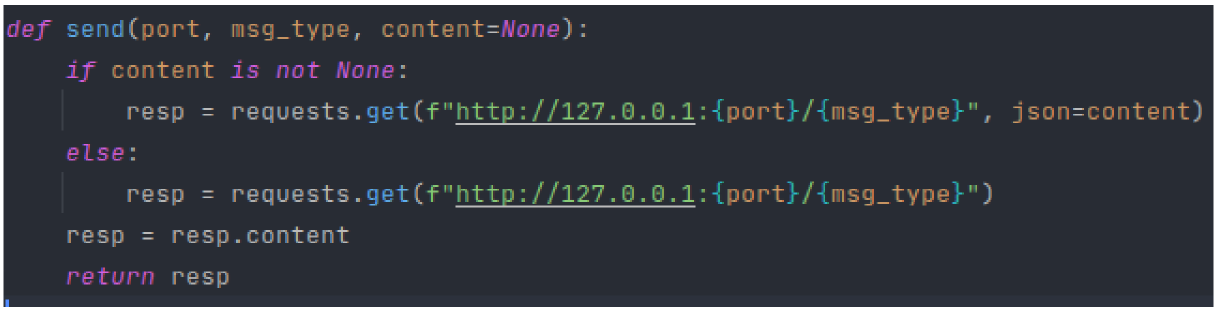
```
Organisation of the webpages:
|_'/': main page
|_'/alive': page displaying 'up and running' if the node is active
|_'/myfiles': page summing up the files available to upload
   |_'/[file name]/upload': page accessed to upload the corresponding file into the peer to peer network
   |_'/[file name]/clear': page accessed to remove a file from the peer to peer network
   |_'/uploaded': page showing all the files uploaded by the node
      |_'/clear': page accessed to remove all the uploaded files from the network
|_'/getfile/available': page showing the uploaded files on the network with the port of the node that uploaded it
|_'/getfile/[distant node port]/[file name]': page accessed to ask for transmission of the corresponding file from the corresponding node
```

fig.1. organisation of the useful pages for the user

Note that this diagram does not sum up all the existing webpages of the flask server, but only the ones made for the user to access (and get data).

2.1.1 Sending requests

To be able to communicate with other nodes in the network and to get information from them, the program needs to use the function ‘requests.get’. In order to simplify its use, I created a file name ‘send.py’, containing only one function: ‘send’.



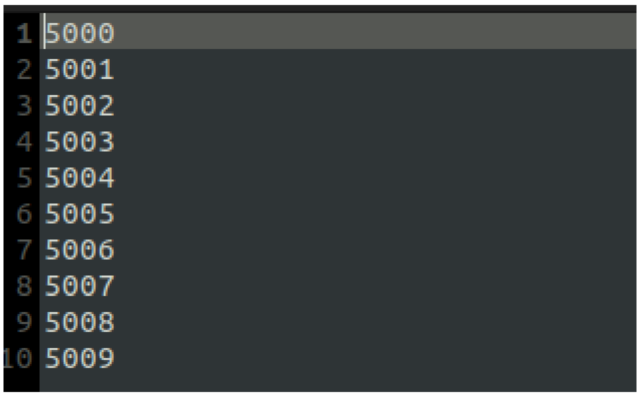
```python
def send(port, msg_type, content=None):
    if content is not None:
        resp = requests.get(f"http://127.0.0.1:{port}/{msg_type}", json=content)
    else:
        resp = requests.get(f"http://127.0.0.1:{port}/{msg_type}")
    resp = resp.content
    return resp
```

fig.2.send function

This function verifies if there is some data to transmit along the message. Whether the case, it sends the request with the right parameters and returns the response of the other node.

2.1.1 Discovery phase

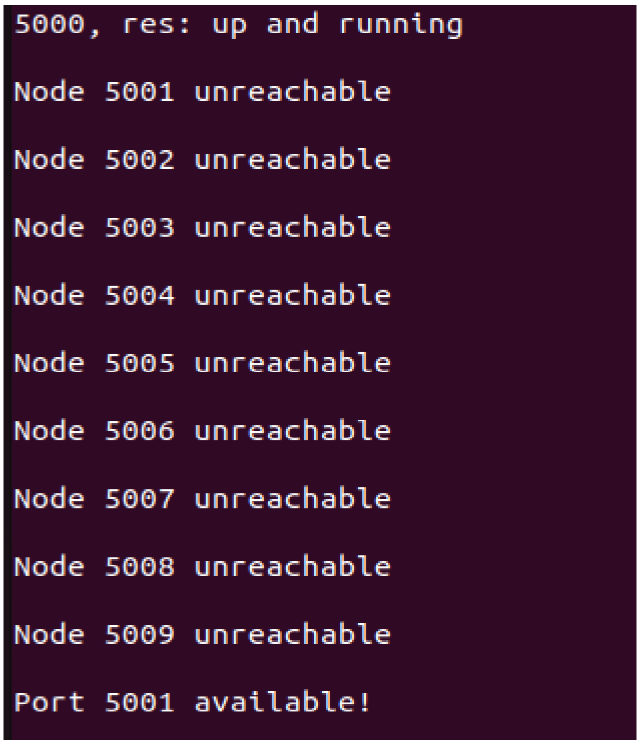
The natural first step is to first create a P2P network. As it is a local implementation, the IP address of all the instances would be “127.0.0.1”, so instead of defining IP addresses, I had to define port numbers to differentiate each node. To do so, a simple .txt file was created and is accessed during the initialisation.



```
5000
5001
5002
5003
5004
5005
5006
5007
5008
5009
```

fig.3. ‘ports.txt’ containing the list of authorized ports for the application

For each possible port number, the program tries to access the node with a ‘requests.get’ to ‘127.0.0.1:[port number]/alive’. The webpage (available when the node is still alive) just returns a string “up and running”. If the sender got an answer, the port number is considered as used and if there is any remaining port not responding to the request, it would be then used as the port number for this instance of the program.

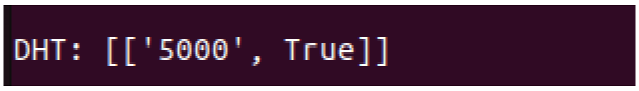


```
5000, res: up and running

Node 5001 unreachable

Node 5002 unreachable

Node 5003 unreachable

Node 5004 unreachable

Node 5005 unreachable

Node 5006 unreachable

Node 5007 unreachable

Node 5008 unreachable

Node 5009 unreachable

Port 5001 available!
```

fig.4. discovery phase of the program printed in the shell (logs)

Here, on fig.2, we can see that the ports 5001 to 5009 are available. Once the program defined its own port number, it starts answering to the requests of the other nodes (in the figure there is one other node using the port 5000).

In order to get the actual alive nodes at any time to communicate with them, a DHT (standing for 'Distributed Hash Table') containing all the responding nodes is created. One of the requirements being to keep in the table all the nodes that were once available but are not anymore, a Boolean being True if the node responds and False on the contrary is added to each port number of the table.



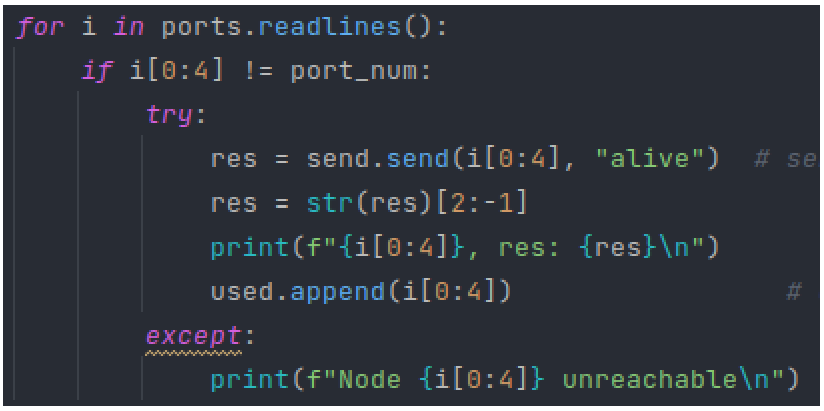
```
DHT: [['5000', True]]
```

fig.5. DHT created by our node upon launch

A similar DHT is created by the node on port 5000 but containing port 5001 instead.

2.1.2 Keep alive

To keep this table updated, a thread is created to send similar requests than the one sent in the discovery phase to all the port numbers.

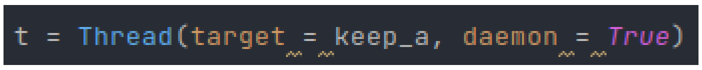


```python
for i in ports.readlines():
    if i[0:4] != port_num:
        try:
            res = send.send(i[0:4], "alive")  # se
            res = str(res)[2:-1]
            print(f"{i[0:4]}, res: {res}\n")
            used.append(i[0:4])                  #
        except:
            print(f"Node {i[0:4]} unreachable\n")
```

fig.6. function that lists all the used ports (called by the thread every ten seconds)

Once the thread gets the list of all the ports used by a node, it compares it to the DHT. If a new node appeared, it is added to the DHT with a true Boolean. If a node stopped responding, its Boolean goes to false and finally if a node started to respond again its Boolean goes to true.

Finally, this thread is designed with the parameter "daemon", making it stop if the main thread is shut down.



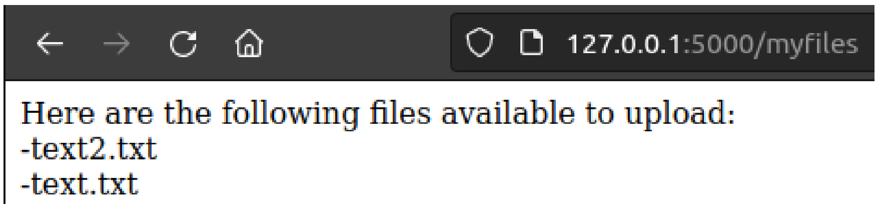
```python
t = Thread(target = keep_a, daemon = True)
```

fig.7. thread definition

2.2 Exchanging files

2.2.1 Available files

To exchange files with other users, a directory named '/files/' was created inside the program directory. All the files needed to be potentially exchanged, must be placed inside of it beforehand. To know all the files actually inside this directory, one of the webpages created and accessible by the user is '127.0.0.1:[port number]/myfiles'.


127.0.0.1:5000/myfiles

Here are the following files available to upload:
-text2.txt
-text.txt

fig.8. list of all the available files to upload

The function of this webpage basically just cycles through the directory (with the help of 'os.listdir()') and returns it.

2.2.2 Uploading a file

The system is designed so that not all available files are accessible by other nodes. It seems quite logical according to protection of personal data. So the user has to specify which files are to be uploaded. It can be done by accessing the webpage at url: '127.0.0.1:[port_number]/myfiles/[file name]/upload'. It will automatically execute the associated function 'upload_file'. This function opens a file, to get its content, and then calculates the content's hash value.

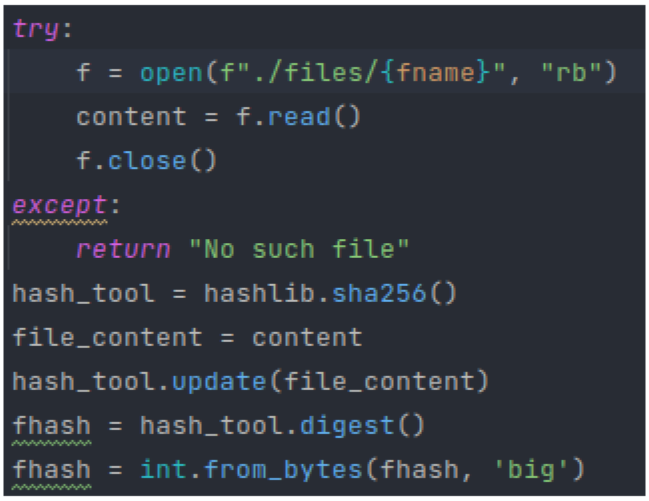
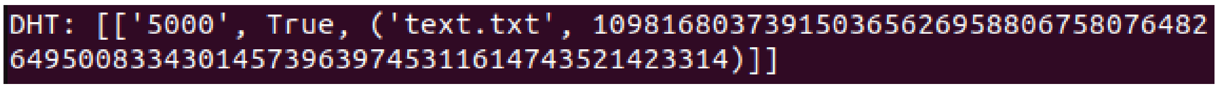

```
try:
    f = open(f"./files/{fname}", "rb")
    content = f.read()
    f.close()
except:
    return "No such file"
hash_tool = hashlib.sha256()
file_content = content
hash_tool.update(file_content)
fhash = hash_tool.digest()
fhash = int.from_bytes(fhash, 'big')
```

fig.9. hash value computation

Once the hash value is calculated, the function checks if a file with the same hash value (meaning a file with the same content) is already uploaded. In fact, each time a file is indeed uploaded, a tuple containing the file name and its hash value are added to the list already containing the port number and the Boolean in the DHT of the other nodes.



```
DHT: [['5000', True, ('text.txt', 109816803739150365626958806758076482
6495008334301457396397453116147435214233 14)]]
```

fig.10. DHT from node with port 5001 with one file uploaded by node with port 5000

So, to check if the file has already been uploaded, the function just goes through the DHT. If no file has the same hash value in the DHT, the function checks if the file is already uploaded (with the global variable ‘uploaded’ containing their names and hash values) on its own node. If it still does not find anything, it can proceed with html requests to the other nodes to notify them all the files that have been uploaded.

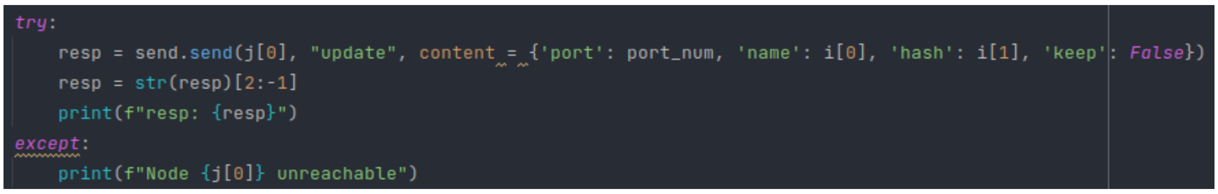
The webpage ‘127.0.0.1:[port number]/myfiles/uploaded’ permits to show the different files uploaded by the node with the corresponding port number.

Upon receiving a message from another node stating that a file was uploaded, the function ‘update’ handles the modification of the DHT according to the parameter of the message received.

2.2.3 Removing a file

Even though the subject did not ask about it, with concerns for private data protection, I decided to create two functionalities: to remove a specified uploaded file and to remove all of them. It goes without saying they can only affect the files uploaded by the user’s node and not the ones uploaded by other nodes.

To remove a specific file, the user needs to access the webpage ‘127.0.0.1:[port number]/myfiles/[file name]/clear’. The function behind this page first removes the files from the ‘uploaded’ global variable and then sends a message to other nodes with the file information in the data (name, hash value and port of the node that uploaded the file).



```
try:
    resp = send.send(j[0], "update", content = {'port': port_num, 'name': i[0], 'hash': i[1], 'keep': False})
    resp = str(resp)[2:-1]
    print(f"resp: {resp}")
except:
    print(f"Node {j[0]} unreachable")
```

fig.11. removing the file on other DHT

Please do note that on figure 11, the ‘keep’ variable in the data transmitted is set to false (contrarily to the messages sent after uploading files). This variable determines if the receiving node should try to add or remove it from the DHT.

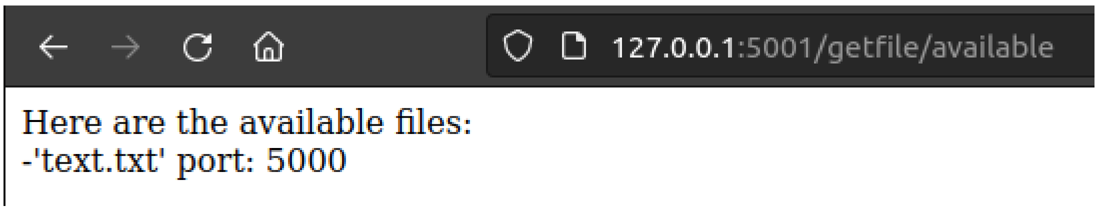
To remove all the files a user uploaded, he needs to access the webpage with address: ‘127.0.0.1:[port number]/uploaded/clear’ which does the same actions as the function to remove one file, but for all the files inside the ‘uploaded’ global variable list.

2.2.4 Accessing files

Now that the files are uploaded, the objective is to send them to other nodes when requested, or more precisely, to return them on our webpage so that when another nodes requests it, the answer will be the content of the file.

By accessing the webpage ‘127.0.0.1:[port number]/getfile/[port number of the other node]/[file name]’, the user executes the function ‘get_file’, that tries to send a request to the webpage ‘/myfiles/[file name]’ of the other node. This webpage tries to access the specified file and return it to the requester.

To permit the user knowing which files are uploaded on the network, the page ‘127.0.0.1:[port number]/getfile/available’ lists all the uploaded files from the other nodes along with the port number of the node that uploaded it.



```
Here are the available files:
-'text.txt' port: 5000
```

fig.12. list of uploaded files over the network

2.3 Encrypting the exchange

According to the work realised in assignment 2, the encryption process will follow a key generation according to the principle of Diffie-Hellman technique. The only change here in the algorithm is that the public key of the other node is recovered with a request directly to the other node through a webpage at the address ‘127.0.0.1:[port number]/getkey’.

With the computed shared key, a pseudo-random key is generated following BBS (Blum Blum Shub) algorithm. This key is then used as seed for the AES algorithm.

Instead of returning the content of the file on the page ‘127.0.0.1:[port number]/myfiles/[file name]’, it returns the encrypted version. Then, the get_file function decrypts it to get the content.

3. Observations

After tests, the program seems to work perfectly fine. In order to verify that the data exchanged is indeed the encrypted version of the file, I added some prints in order to get information inside the shell as administration data (and not data seen by the user). According to the requirements, this implementation respects both resilience of the network to changes of architecture (nodes becoming unavailable for instance) and security of the data transmitted. I also added some functionalities for ease of use for the customer which should permit anyone to know how to use the application in a real case scenario.

4. Conclusion

We succeeded in creating a secured way of exchanging files between users. Adding the functions to remove files from the network enhance the security of the private datas. But if we wanted to improve it, we could change the choice of encryption method. As pointed out in the previous assignment, AES and BBS are not the most secure algorithms (even though they still are correct at what they do). We must also keep in mind that this implementation is local, and not over the Internet. If we had to implement distant communications, it would be good to create a VPN between the nodes to secure even further the information (by implementing security on other communication layers, here the VPN secures layer 3).