Report - Programming Assignment #1

1. Design documentation

1.1. Implementation

To implement this assignment, I chose to use Python for its high level and its possibility to program with sockets. All the communications are based on TCP.

The code is broken down in three parts:

1. the peer
2. the indexing server
3. the communication protocol

In this section, I will start by describing the communication protocol as it is used by both the peer code and the server code. Then, I will define what data structures are maintained by the indexing server and each peer. Finally, I will describe the overall behavior of the system.

1.1.1. The communication protocol

The communication protocol is built over the python sockets. It is made of one class: the `MessageExchanger` class. Each `MessageExchanger` object is built by passing a socket as an argument. The functions of this class hide the protocol to the user. This way, the communication is easier to implement. Amongst the properties of the communication protocol, here are the more important ones:

- it uses a buffer of size 4096 bytes;
- the end of a message is coded using `\r\r\n`;
- one can send an acknowledged message such that the message is sent, and then the function waits for an acknowledgement;
- python objects can be sent using data serialization;
- files (text and binary) can be sent from one side of the `MessageExchanger` and saved to disk on the other side of it.

This layer is important as it allows the peer and the indexing server to be developed without paying attention to the end of the messages, serializing objects, or managing a file transfer. In summary, this protocol allow abstraction of the process. It can also be useful if we decide later to communicate through UDP or MPI instead of TCP as this would reduce the amount of code to modify. Finally, it also allows us avoid code repetition.

1.1.2. The indexing server

The indexing server maintains the following data structures:

- `peers_info` is a hashtable that associates a peer identifier to the peers' informations such as its listening public IP addresses and its listening port.
- `file2peers` is a hashtable that maps a file identifier (currently its name) to the list of peers that have registered this file.
- `files_info` is a hashtable that associates a file identifier to its infos such as its absolute path and its size.

When launching the indexing server, the program is waiting for connections from peers. When a connection is accepted, a new process is created that handles the communication with the peer. This way, our server is multithreaded (which is

actually multiprocessing in python).

The new process is a message handler that loops until the peer ends the connection. Each message received by the handler is processed and the correct function is consequently launched. The functions handled by the indexing server are the functions defined in section **1.2.** of the README file.

1.1.3. The peer

The peer instances maintain the following data structure:

- `files_dict` is a hashmap associating a file identifier (its name) to its informations such as its size and its absolute path.

When launching the peer process, a server process is run in the background. This server waits for connections from other peers. Once a connection is accepted by the server, a new process is created to handle the connection with the other peer to manage the file transfer. In the first plan, a command line interface runs to handle the user commands. The supported handle commands are defined in section **1.2.** of the README file.

1.1.4 Overall behavior

Here we explain the behavior of the system. To start, we let IS be the indexing server and name two peers P1 and P2. The details on how to launch an indexing server and peers is explained in the README file.

1. When a peer is launched, it connects to the IS and send its information: its id (created using the id function in python), its listening IP address, and its listening port. Using this information, the IS can update its `peers_info` hashtable.
2. When a peer registers, the information of each file is send to the IS so that it can fill the different hashtables. The information is sent using data serialization in order to avoid creating a specific protocol to send files' information. The information of every file is sent one by one and when this is done, a "poison pill" is send to indicate to the server that all files have been registered for this peer.
3. When the user asks to lookup for a file, the request is forwarded by the peer P1 to the IS. The IS goes through its `files2peer` map to access the list of peers that contains this file. This list contains the identifier of every peer that contains the requested file. For each of these peers, the peer information (IP and port) can be accessed by the IS through the `peers_info` data structure. The IS sends back to P1 a list with the IP address and port of every peer that contains the requested file. P1 then waits for a user decision on the peer P2 to select for the file transfer. Once the user selected P2 amongst the list of available peers, a connection is created between P1 and P2 and the file transfer can begin through the communication protocol as it supports file transfers (either text or binary). Once the file transfer is done, the connection between P1 and P2 is automatically closed.
4. When a peer P1 exits, a message is sent to the IS so that P1's information and the information related to P1's files are removed from the IS. Once this is done, the connection between the IS and P1 is closed on both sides and the processes handling the connections terminate

Note that the connection between a peer and the indexing server is always open whereas the connection between two peers stay open only for the time of the file transfer.

1.2. Possible improvements

In the actual implementation, one process is created every time a connection is accepted. This part of the code could be improved by created a static pool of processes that get the accepted connections from a shared queue. This would improve the performance in the case of a very large number of connections.

Another possible improvement would be to identify a file by its hashed value and not by its name. This would allow the indexing server to index files with the same name but different contents. Computing the hash value of each file could be done on the peer side at registering time to avoid the IS to be overloaded.

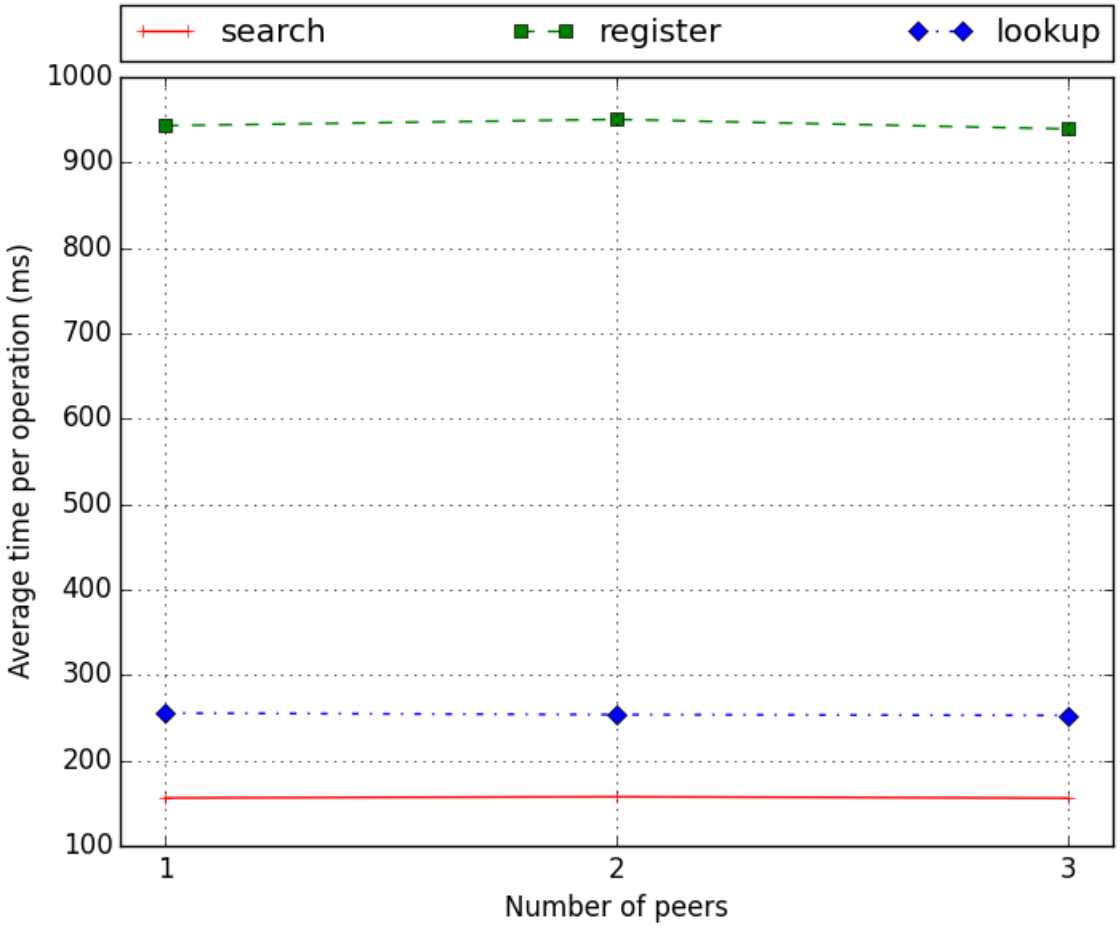
Finally, the registering process could be improved as one message is sent for every file to register. Doing bulk registering by sending one message for all the files to register would probably improve the performance by reducing the number of exchanges between the peer and the indexing server.

2. Perfomance evaluation

In order to evaluate the performance of our system, we ran the following benchmark:

- 1000 search operations;
- 200 register operations;
- 800 lookup operations.

In order to see how the system scales, we have increased the number of peers running this benchmark concurrently from 1 to 3 peers. We ran these benchmarks using one t2.medium instances for the server and one for every peer. These instances have 2 vCPUs.



The above figure plots the average time per operation for the search, lookup, and register benchmarks. We can see that the performance does not decrease even though we increase the concurrency level. Even though I was expecting the average response time to drop when reaching 3 concurrent peers, I examined the overall CPU usage of the indexing server and noticed that even though 3 processes were communicating with it concurrently, only at most 10% of the CPU is used. Let's also note that the requests are all succeeding as the benchmark stops if a request fails.

In conclusion, I have implemented and benchmarked an indexing server and peers as the first step into a distributed file sharing system. Even though the system can still be improved, every required function (register, search, lookup) have been implemented and is fully functional. Moreover, the system supports the transfer of binary file.