

RELAZIONE PROGETTO

Sistemi distribuiti e Cloud Computing

D'Atri Fulvio 235344

Unical LM Ing. Informatica

Prof. Domenico Talia

Prof. Loris Belcastro

07/09/2022

Sommario

1. Introduzione	1
2. Applicazione Python	3
2.1. Classi di utilità.....	3
2.1.1. Configs	3
2.1.2. Messages.....	3
2.1.3. Utils	4
2.2. Classe Server	4
2.3. Classe Node.....	6
3. Docker Engine	11
3.1. Immagini	11
3.2. Container	12
3.3. Rete	13
3.4. Volumi.....	14
3.5. Esecuzione dei container	14
3.6. Reset dei container	16
4. Esecuzione	17
5. Codici sorgenti.....	22
5.1. server.py	22
5.2. node.py	24
5.3. utils.py	34
5.4. configs.py.....	37
5.5. message.py.....	38
5.6. node2node.py.....	38
5.7. node2server.py	39
5.8. server2node.py	39
5.9. chunk_sharing.py.....	39
5.10. docker-compose.yml	39
5.11. config.sh	42
5.12. reset.sh	42
5.13. run.sh	42

1. Introduzione

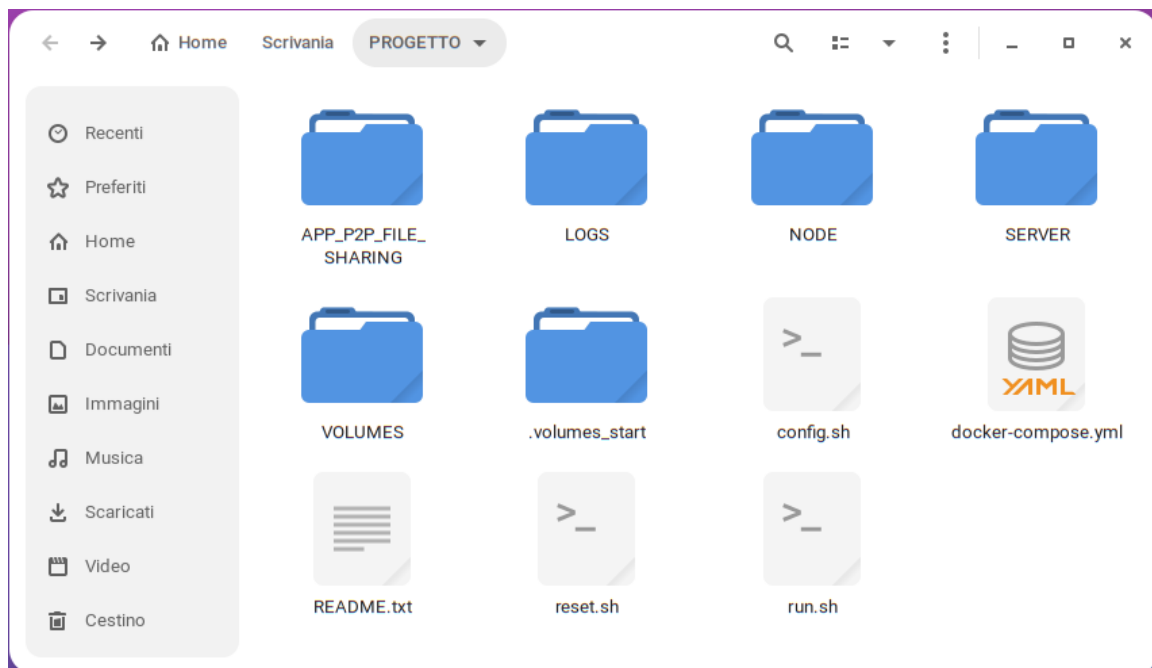
La presente relazione descrive i passi fondamentali dello sviluppo del progetto e della sua esecuzione. Il progetto di riferimento riguarda la realizzazione di un'applicazione che implementi un servizio distribuito di **file sharing peer-to-peer**, utilizzando soluzioni di calcolo, storage e virtualizzazione messe a disposizione da **Docker Engine**.

Gli strumenti principali adottati per lo sviluppo del progetto sono stati:

- **Docker Engine**: come richiesto esplicitamente dalla traccia del progetto; per "simulare" l'intera rete, dove saranno collegati i diversi nodi ed il singolo server.
- **Python**: per lo sviluppo del software del server e dei nodi; si è fatto uso della libreria `socket`, per lo scambio di dati tra le macchine connesse nella rete.

Il risultato finale del progetto corrisponde all'esecuzione simultanea di più nodi nella stessa rete e del server, dove tali macchine saranno totalmente isolate tra loro essendo diversi *container* eseguiti sullo stesso sistema operativo. In questo modo si potrà testare il funzionamento dell'applicazione per il download di file da parte dei nodi.

La cartella finale prodotta come output del progetto è la seguente:



All'interno della cartella sono presenti i seguenti file:

- **APP_P2P_FILE_SHARING**: è la cartella contenente tutti i codici sorgenti python, ovvero quello per il server, per il singolo nodo e altri di utilità.
- **NODE**: è la cartella contenente il *Dockerfile* per la creazione dell'immagine dei container che rappresenteranno i nodi.

- *SERVER*: è la cartella simmetrica per quanto riguarda però l'immagine del container che rappresenta il server.
- *VOLUMES*: è la cartella contenente un'ulteriore cartella per ogni nodo della rete; ogni cartella interna è un volume condiviso quindi con il rispettivo container. Tale strategia è stata utile per visualizzare in tempo reale i file attuali e quelli appena scaricati dai diversi nodi.
- *LOGS*: è la cartella come quella in precedenza, contiene all'interno ulteriori volumi. L'utilità di quest'altra invece è quella di poter controllare i log sia del server che dei diversi nodi in caso di errori e debug.
- *docker-compose.yml*: è il file per l'esecuzione di *Docker Compose*, ovvero saranno specificati all'interno tutte le caratteristiche dei servizi e quindi dei container che saranno eseguiti (es. rete, volumi, ecc.).
- *config.sh*: è il file per la configurazione delle immagini docker; in particolare rimuove e riesegue la *build* dei due *Dockerfile*.
- *reset.sh*: è il file per il reset dell'intera applicazione; ovvero cancella i log, termina ed elimina tutti i container in esecuzione, elimina file temporanei di configurazione ed infine se specificato con "-v" effettua un ripristino dei file nei nodi della rete.
- *run.sh*: è il file per l'avvio dell'intera applicazione; lancia in esecuzione su ogni terminale diverso, il server e i nodi. Il numero di nodi può essere specificato come parametro con "-n", oppure direttamente il nome del nodo con "-nn";
nota importante: attualmente il sistema ha un limite massimo di nodi in esecuzione pari a 10.

Nel proseguo della relazione verranno approfonditi gli aspetti appena citati, in particolare la struttura è la seguente:

- 2. *Applicazione python*: è il capitolo in cui verranno approfonditi alcuni aspetti lato codice sorgente del server e dei nodi.
- 3. *Docker Engine*: è il capitolo in cui verrà approfondito tutto l'aspetto che riguarda la realizzazione della rete di container, approfondendo quindi il tipo di *network* adottata, le immagini, i volumi e anche *docker compose*.
- 4. *Esecuzione*: è il capitolo in cui si spiegherà meglio la fase di esecuzione dell'intero progetto con alcuni esempi; si vedrà l'esecuzione dei diversi container, con relative spiegazioni dei file *.sh* per il reset e il run dell'applicazione.
- 5. *Codici sorgenti*: è il capitolo finale in cui sono presenti tutti i codici sorgenti.

Da precisare che per il funzionamento dell'intera applicazione si è seguita la traccia del progetto; oltre allo standard richiesto dalla traccia si è fatta un'aggiunta nel funzionamento dell'applicazione: una modalità di scaricamento dei file "extra". Questa modalità è simile a quella che avviene nel protocollo *BltTorrent*, ovvero il nodo che richiede il file, invece di scegliere un singolo peer per il download di quel file, effettua il download in parallelo su tutti i peers che posseggono quel file, dove ognuno di essi si occuperà a scaricare una porzione del file (*chunk*), e sarà responsabilità del nodo che richiede il file a riordinare i singoli chunk provenienti dai diversi nodi e ricreare il file originale.

2. Applicazione Python

In questo capitolo verranno dettagliate le parti più importanti dei **codici sorgenti** dell'applicazione, che come detto in precedenza, sono scritti in linguaggio **python**. Si inizieranno con alcune classi di utilità, come ad esempio le classi per definire oggetti per i messaggi scambiati, oppure per la configurazione generale; si continuerà con il codice del server ed infine quello del nodo.

2.1. Classi di utilità

Le classi di utilità sono quelle per i messaggi scambiati, quella per la configurazione delle costanti ed infine quella di metodi di utilità generale.

2.1.1. Configs

È una classe che crea un oggetto a partire da un **json**. In tale classe sono presenti tutte le costanti che saranno usate nei codici sorgenti; in particolare si dividono in **4 macrogruppi**:

- *directory*: contiene i percorsi della cartella dei logs e della cartella che contiene i file posseduti dal singolo nodo.
- *constants*: contiene tutte le costanti di configurazione, come ad esempio l'indirizzo ip e la porta del server, le dimensioni del buffer per lo scambio dati TCP e UDP, i tempi di intervallo in cui il server controlla i nodi connessi e i nodi inviano un segnale per informare che sono ancora connessi al server, ecc.
- *server_request_mode*: contiene la tipologia di dati che vuole inviare il nodo al server. La modalità *REGISTER*, il nodo invia tale pacchetto al server sia al primo avvio che a cadenza di tempo prestabilita per avvisare il server che è ancora connesso nella rete; e la modalità *EXIT*, quando l'utente vuole uscire esplicitamente, allora il nodo invia tale pacchetto al server per notificare la sua uscita dalla rete.
- *node_request_mode*: contiene la tipologia di dati che i nodi si scambiano tra loro. La modalità *DOWNLOAD*, quando il nodo richiede il download diretto ad un altro nodo presente in rete di un file che possiede quest'ultimo; la modalità *SIZE*, quando un nodo chiede ad un altro nodo la dimensione di un file; la modalità *CHUNKS*, quando il nodo richiede agli altri nodi che possiedono quel file il download diviso in chunks, ed ogni nodo ricevente si occuperà di inviare una sola porzione del file. Da notare che le ultime due tipologie vengono adottate solamente se si vuole effettuare uno scaricamento del file per chunks.

2.1.2. Messages

Sono tutte quelle classi che permettono la creazione di oggetti adatti per la trasmissione in rete.

```
1  from __future__ import annotations
2  import pickle
3
4  class Message:
5      def __init__(self):
6          pass
7
8      def encode(self) -> bytes:
9          return pickle.dumps(self.__dict__)
10
11     @staticmethod
12     def decode(data: bytes) -> dict:
13         return pickle.loads(data)
```

In particolare la classe *Message* è la classe madre che espone i metodi *encode* e *decode* grazie l'ausilio della libreria *pickle*, in modo da serializzare l'oggetto che si vuole trasmettere in rete.

Le altre classi sono una specializzazione di *Message*; è presente una classe per ogni tipo di scambio che avviene tra le macchine:

- *Node2Node*: serve per lo scambio di dati tra peers; l'oggetto creato avrà come campi il nome del file (*filename*), il tipo di richiesta (*mode*), la dimensione del file (*size*), il range della dimensione del file da inviare (*range*) ed infine la porta UDP del server in ascolto per il download del file diviso in chunks (*portUDP*). Da notare che questi ultimi tre campi verranno usati solamente in caso si vorrà scaricare il file diviso in chunks parallelamente dai diversi nodi che lo possiedono.
- *Node2Server*: serve per lo scambio di dati del nodo verso il server; i campi sono il tipo di richiesta da effettuare al server (*mode*), la porta TCP in ascolto del nodo per le richieste della presenza file locale (*request_port_TCP*) ed infine la porta TCP in ascolto per il download effettivo del file (*download_port_TCP*).
- *Server2Node*: serve per lo scambio di dati del server verso il nodo; l'unico campo è la lista risultato (*search_result*) che corrisponde alla lista degli altri nodi connessi in rete che il server invia al nodo.

Infine è presente un'altra classe che è utile solamente nel caso in cui si volesse scaricare il file parallelamente per chunk dai diversi nodi che lo detengono. Tale classe è *ChunkSharing* e come campi ha il nome del file (*filename*), il range di byte da scaricare (*range*), l'identificativo del pacchetto all'interno del chunk che si sta inviando (*idx*) ed infine l'effettivo pacchetto di byte che si sta inviando (*chunk*).

2.1.3. *Utils*

È una classe in cui sono presenti alcuni dei metodi di utilità, in particolare i seguenti:

- *set_socket_TCP(port, addr)*: metodo per la creazione di un socket TCP con porta *port* e indirizzo *addr*.
- *set_socket_UDP(port, addr)*: metodo per la creazione di un socket UDP con porta *port* e indirizzo *addr*.
- *free_socket(sock)*: metodo per effettuare la *close()* del socket e rimuovere la porta usata dalla lista di porte usate in esecuzione.
- *generate_random_port()*: metodo per la generazione casuale di una porta tra quelle ammissibili e disponibili, quindi nel range [1024, 65535] e che non sia già presente nella lista delle porte usate.
- *parse_command(command)*: metodo per la lettura del comando scritto dall'utente nell'applicazione del nodo che restituisce il comando seguito eventualmente dal nome del file se è un comando di download.
- *log(content: str, is_server=False, printing=False)*: metodo che effettua la scrittura su file e su schermo se esplicitato di una stringa; utile per scrivere i log dell'applicazione.

2.2. Classe Server

Prima di entrare nel dettaglio del codice, si descrive brevemente la **logica di funzionamento** del server.

All'avvio del server, esso lancia un thread periodico che andrà a controllare ad intervalli regolari quali nodi sono ancora connessi in rete; soluzione importante nel caso in cui un nodo dovesse uscire dalla rete involontariamente senza avvisare il server. In questo modo

se il thread rivela l'assenza del nodo dalla rete, lo elimina dalla lista di tutti i nodi attualmente connessi. Nel frattempo il thread principale del server è in ascolto sulla porta TCP (definita nella classe *Configs*) sui nodi che lo vogliono contattare. Il server lancia un nuovo thread per ogni connessione che stabilisce con un nodo. Quest'ultimo thread si occupa di controllare la tipologia di richiesta che il nodo invia al server, ed eseguire azioni in base ad essa:

- *REGISTER*: il thread aggiorna la presenza del nodo nella rete, ed invia a quest'ultimo la lista dei nodi connessi.
- *EXIT*: il thread elimina dalla lista dei nodi connessi il nodo che invia la richiesta di uscita.

Entrando nel dettaglio del codice, sono presenti **3 variabili** di istanza e **4 metodi** principali.

Le **variabili** sono:

- *server_socket*: è il socket TCP che il server espone verso i nodi.
- *peers_status*: è un dizionario, in cui la chiave è l'indirizzo ip del nodo, mentre il valore è un booleano; esso indica se *True* che quel nodo è ancora connesso in rete, altrimenti che non è più connesso.
- *peers*: è un dizionario, in cui la chiave è l'indirizzo ip del nodo, mentre il valore è una tupla di due elementi; il primo è la porta TCP per le richieste di ricerca del file (*request_port_TCP*) del rispettivo nodo, mentre il secondo è la porta TCP per le richieste di download del file (*download_port_TCP*).

I **metodi** sono:

- *run(self)*: è il primo metodo che viene eseguito; ha il compito di lanciare il thread che esegue il controllo periodico dei peer connessi con il metodo *check_peers_periodically*; inoltre all'interno di un *while True* effettua il metodo *accept()* sul *server_socket*, per accettare connessioni dai nodi. Una volta stabilita la connessione lancia un altro thread che esegue il metodo *handle_node_request* per la gestione della richiesta del nodo, così che il server è subito disponibile per richieste di altri nodi.
- *check_peers_periodically(self, interval)*: è il metodo che si occupa del controllo periodico dei peers connessi in rete. Viene passato come argomento un intero (*interval*) che stabilisce la frequenza di tempo in cui il server deve ogni volta rieseguire tale thread e quindi ricontrollare i nodi connessi in rete. La strategia adottata per controllare se un nodo non è più presente in rete è semplice: si controlla il valore booleano associato a quel particolare nodo dal dizionario *peer_status*, se tale valore è *False* significa che il nodo non ha aggiornato tale stato ricontattando il server, allora non è più presente nella rete e lo si elimina dalla lista dei nodi *peers*; altrimenti se il valore è *True* significa che il nodo ha aggiornato il suo stato recentemente e perciò è ancora presente in rete, quindi l'unica operazione da fare è quella di modificare lo stato in *False*, in modo che se il nodo non ricontatterà il server per notificare la sua presenza, tale valore rimarrà non aggiornato a *False* e quindi al prossimo controllo il nodo verrà eliminato dalla lista dei nodi.
- *handle_node_request(self, conn, addr: tuple)*: è il metodo che si occupa di gestire la richiesta del singolo nodo. Ha due argomenti *conn*, ovvero il socket che viene restituito dopo la chiamata *accept()*, che rappresenta la connessione con il nodo; e *addr* che rappresenta il secondo attributo restituito da *accept()*, ossia l'indirizzo ip e la porta del nodo che si è appena connesso al server. In tale metodo si chiama il metodo *recv()* sull'oggetto *conn* prelevando il dato inviato dal nodo; lo si decodifica con il metodo *Message.decode(data)*; e si estrapola il campo *mode*, così da poter controllare il suo valore e sulla base di questo richiamare il metodo

register_and_send_peers se il valore di *mode* è *REGISTER*, altrimenti rimuovere il nodo dai due dizionari se il valore di *mode* è *EXIT*.

- *register_and_send_peers(self, conn, msg: dict, addr: tuple)*: è il metodo che viene richiamato per effettuare la prima volta il log in del nodo verso il server, oppure quando il nodo notifica periodicamente la sua presenza al server. Sono presenti gli argomenti *conn*, gli stessi descritti in precedenza; *msg*, rappresenta il messaggio già decodificato che proviene dal nodo; *addr*, è lo stesso descritto in precedenza. Le istruzioni che esegue il metodo sono semplici: elimina il nodo dalla lista dei nodi *peers* se presente (in realtà si usa il metodo *pop* sul dizionario, quindi elimina il nodo se presente altrimenti non esegue niente), questa operazione è necessaria per non inviare la lista dei nodi compreso l'indirizzo del nodo stesso; si crea un oggetto *Server2Node* passandogli come *search_result* la lista dei nodi (*peers.items()*), si usa il metodo *items()* per inviare insieme agli indirizzi ip dei nodi anche le due rispettive porte di ascolto TCP; il server invia l'oggetto appena creato e codificato (*encode()*) con il metodo *send()* sulla connessione con il nodo *conn*; ed infine viene aggiunto il nodo insieme alle sue porte TCP nella lista dei nodi *peers* e aggiornato il suo stato a *True* nel dizionario *peers_status*.

2.3. Classe Node

Prima di entrare nel dettaglio del codice, si descrive brevemente la **logica di funzionamento** del nodo.

All'avvio del nodo, vengono lanciati tre thread: il primo è un thread che si occupa di notificare al server la presenza del nodo nella rete periodicamente; il secondo è un thread che sta in ascolto alle richieste di ricerca di file da parte degli altri nodi; ed infine il terzo è un thread che sta in ascolto sulle richieste di download di file interi da parte degli altri nodi. Il thread principale invece rimane in ascolto dei comandi che l'utente passa all'applicazione. Tali comandi sono 5:

- **help**: l'applicazione restituisce all'utente la lista dei comandi disponibili.
- **getip**: l'applicazione restituisce all'utente l'indirizzo ip della macchina in cui è in esecuzione.
- **show**: l'applicazione restituisce la lista dei file locali che la macchina possiede.
- **exit**: l'applicazione invia il messaggio al server dell'uscita dalla rete, attende eventuali download o upload, ed infine termina.
- **download nomeFile**: è il comando più importante; l'applicazione, nel caso in cui il file *nomeFile* non è presente localmente, restituisce la lista dei nodi che posseggono tale file e chiede nuovamente all'utente da quale nodo scaricare il file, oppure se avviare la modalità di scaricamento per chunks. L'utente allora seleziona il nodo da cui scaricare il file (o tutti i nodi nel caso di download per chunks) ed avvia il download del file da quel nodo (o quei nodi); nel frattempo l'utente può digitare altri comandi, e quindi effettuare altri download. Alla fine dello scaricamento del file l'utente riceverà una notifica.

Sebbene la descrizione del funzionamento possa sembrare semplice, il codice sorgente del nodo è più lungo e complesso di quello del server; proprio perché in questo caso il nodo svolge operazioni caratteristiche sia di un server che di un client, essendo un sistema peer-to-peer.

Entrando nel dettaglio del codice, sono presenti **9 variabili** di istanza e **20 metodi**.

Le **variabili** sono:

- *ip_addr*: è l'indirizzo ip del nodo in cui è in esecuzione l'applicazione; è un valore importante, perché tutti i socket in ascolto del nodo saranno associati a tale indirizzo ip; in questo modo il socket è raggiungibile all'interno della rete dei container.
- *request_port_TCP*: è la porta TCP che viene assegnata al socket *request_socket*; tale porta sarà scelta casualmente ed inviata agli altri nodi.
- *download_port_TCP*: è la porta TCP che viene assegnata al socket *download_socket*; anch'essa sarà scelta casualmente.
- *request_socket*: è il socket TCP che rimane in ascolto delle connessioni degli altri nodi per la richiesta di ricerca dei file.
- *download_socket*: è il socket TCP che rimane in ascolto delle connessioni degli altri nodi per la richiesta di download di un particolare file per intero.
- *peers*: è la lista dei nodi connessi in rete; un elemento è dato dalla forma (*peer_addr*, (*peer_req_port_TCP*, *peer_download_port_TCP*)) dove *peer_addr* è l'indirizzo ip del nodo, *peer_req_port_TCP* è la porta TCP del socket del nodo che ascolta le richieste di ricerca file, ed infine *peer_download_port_TCP* è la porta del socket del nodo che ascolta le richieste di download di un intero file.
- *download_threads*: è la lista dei thread che stanno effettuando uno scaricamento di un file; tale lista è utile quando l'utente digita il comando *exit* e quindi in questo modo si conoscono i thread in fase di scaricamento da attendere prima di terminare l'applicazione.
- *upload_threads*: è la lista dei thread che stanno effettuando un caricamento di un file; questa lista è simmetrica a quella precedente perché fa sapere quali thread in fase di caricamento attendere in caso di uscita intenzionale da parte dell'utente.
- *downloaded_files*: è un dizionario, dove la chiave è il nome del file, il valore è una lista di chunks del file; tale variabile viene usata solo per il download del file per chunks.

I **metodi** sono:

- *run(self)*: è il primo metodo che viene eseguito all'avvio dell'applicazione, il suo compito è quello di avvisare il server dell'accesso alla rete del nodo, di lanciare i tre diversi thread descritti nel funzionamento dell'applicazione e di stare in ascolto dei comandi che l'utente invia.
- *enter_network(log_content)*: è il metodo che viene chiamato sia al primo avvio che periodicamente per avvisare la presenza del nodo nella rete al server. Il parametro *log_content* è una stringa che serve solamente per questioni di debug e log. Il compito del metodo è quello di creare un messaggio con la tipologia *REGISTER*, stabilire una connessione verso il server, ed inviare tale messaggio; la risposta del server sarà la lista dei nodi attualmente connessi in rete, che sarà salvata nella variabile di istanza *peers*.
- *inform_server_periodically(interval: int)*: è il metodo del thread periodico lanciato dal metodo *run*, che serve per notificare la presenza del nodo nella rete al server; altro non fa che invocare il metodo *enter_network* descritto precedentemente.
- *receive_request_search_from_nodes()*: è il metodo del secondo thread lanciato dal metodo *run*; il suo compito è quello di stare in ascolto sul socket *request_socket*, e non appena viene stabilita una connessione viene lanciato un ulteriore thread che si occuperà di gestire la richiesta con il metodo *check_file*; in questo modo all'interno di un ciclo *while*, il thread corrente è già disponibile a soddisfare richieste provenienti da altri nodi.
- *check_file(conn, addr: tuple)*: è il metodo invocato per la ricerca del file locale richiesto da un nodo che ha appena stabilito la connessione *conn*, e avente indirizzo

addr. Tale metodo esegue semplici istruzioni: legge e decodifica il messaggio proveniente dal nodo, preleva il nome del file che ha richiesto, ed effettua una ricerca nel file system locale; infine risponde al nodo richiedente con uno stesso messaggio, dove il nome del file è vuoto nel caso in cui il file non sia presente localmente.

- *search_file_owner(filename: str)*: è il metodo che dato il nome del file *filename*, restituisce la lista dei nodi che posseggono quel file. Il funzionamento consiste nella creazione di un socket client e di una lista vuota che diventerà il risultato; viene eseguita un'iterazione su tutti i componenti della lista dei nodi connessi *peers*, e per ogni nodo, viene creato un messaggio con il solo nome del file (*filename*); si invia tale messaggio codificato all'indirizzo del nodo e alla porta che è in ascolto sulle richieste di ricerca dei file (*peer_req_port_TCP*) e si attende la risposta. Arrivata la risposta dall'altro nodo, si controlla se il nome del file non è vuoto e in questo caso si aggiunge alla lista risultato l'indirizzo ip del nodo e la porta di ascolto del nodo dei download (*peer_addr*, *peer_download_port_TCP*); in caso contrario significa che quel nodo non possedeva il file *filename* e quindi si passa al nodo successivo.
- *receive_request_download_from_nodes()*: è il metodo che viene eseguito dal terzo thread avviato dal metodo *run*, ovvero il thread che è in ascolto sulle richieste di download di un particolare file. All'interno di un *while True*, si è in ascolto di connessioni da parte degli altri nodi; stabilita la connessione con un nodo, si legge il messaggio ed in base alla tipologia di messaggio si lancia un thread diverso per la sua gestione. Esistono tre tipologie:
 - *SIZE*: il nodo richiedente domanda la dimensione del file; viene avviato un thread sul metodo *tell_file_size*.
 - *CHUNKS*: il nodo richiede il download del file per chunks; viene avviato un thread sul metodo *send_chunk*.
 - *DOWNLOAD*: il nodo richiede il download del file intero; viene avviato un thread sul metodo *send_file*.

Infine viene aggiunto il thread appena creato nella lista dei thread che stanno effettuando un caricamento verso un altro nodo (*upload_threads*).

- *tell_file_size(msg, conn)*: metodo che si occupa dell'invio della dimensione del file con nome file *msg['filename']*. Viene inviata la dimensione sul socket *conn*; infine viene rimosso il thread corrente dalla lista *upload_threads*.
- *ask_file_size(filename: str, file_owners)*: metodo che restituisce la dimensione del file *filename*. Crea un messaggio con tipologia *SIZE*, ed itera su ogni nodo presente nella lista *file_owners* passata come argomento; si invia al nodo corrente il messaggio codificato, se quest'ultimo risponde, allora si interrompe il ciclo e si restituisce la risposta, ovvero la dimensione del file; altrimenti in caso di errore di connessione si procede con il prossimo nodo nella lista *file_owners*.
- *send_file(conn, msg)*: è il metodo che si occupa dell'invio del file *msg['filename']* al nodo che ha stabilito una connessione con il socket *conn*. Per l'invio si legge ad ogni iterazione un gruppo di byte pari alla dimensione massima del buffer TCP (*TCP_BUFFER_SIZE*) e la si invia sul socket *conn*; terminata la lettura del file si chiudono i flussi di byte e si elimina il thread corrente dalla lista *upload_threads*.
- *start_download(file_owner, filename)*: metodo che si occupa al contrario di quello in precedenza di scaricare il file *filename*, dal nodo *file_owner*. Viene creato un messaggio con il nome del file e con tipologia di messaggio *DOWNLOAD*, viene codificato ed inviato sul socket delle richieste di download in ascolto di *file_owner*. La risposta di *file_owner* corrisponde alle sequenze di byte dell'invio effettivo del file; quindi simmetricamente al metodo *send_file*, all'interno di un ciclo *while* si effettua

la lettura. Infine si chiudono i flussi di byte aperti e viene rimosso il thread corrente dalla lista dei thread che sono in fase di download di file (*download_threads*).

- *fetch_owned_files()*: è un metodo banale che si occupa di restituire la lista dei nomi dei file presenti localmente nel path *node_files_dir*. Tale metodo viene richiamato ogni qual volta si deve controllare la lista dei file locali posseduti.
- *download(self, filename: str)*: metodo che si occupa di scaricare il file *filename*. Innanzitutto si verifica se il file non è già presente localmente, in tal caso si informa l'utente che il file è già presente, altrimenti, si preleva la lista dei possessori (*file_owners*) di quel file connessi in rete con il metodo *search_file_owners*; se la lista è vuota, significa che nessun nodo possiede quel file, e quindi viene notificato l'utente; nel caso in cui esiste almeno un nodo in rete che possiede quel file, viene mostrato all'utente la lista degli indirizzi ip dei nodi possessori del file e viene chiesto all'utente di digitare un numero *x* che sia tra i seguenti:
 - *x == -1*: l'utente ha intenzione di annullare il download, allora riceve una notifica di annullamento e viene effettuato un *retrun*.
 - *x == -2*: l'utente ha intenzione di effettuare un download in parallelo su tutti i thread diviso il file per chunk. Viene lanciato un thread sul metodo *start_download_chunks* e viene aggiunto il thread nella lista *download_threads*.
 - *x >= 0 and x < len(file_owners)*: *x* indica la posizione del nodo scelto per il download intero del file nella lista *file_owners*. Viene lanciato un thread sul metodo *start_download* e viene aggiunto il thread nella lista *download_threads*.
- *exit_network()*: è il metodo che viene eseguito ogni qual volta l'utente digita il comando *exit*, ovvero ha intenzione di uscire dalla rete. Si effettua la copia delle liste dei thread che sono in fase di download e upload (*download_threads* e *upload_threads*), perché altrimenti potrebbero esserci delle inconsistenze in fase di iterazione di queste ultime; per ogni thread nelle liste si effettua una *join*, in modo da attendere la loro terminazione; dopo di ciò viene inviato un messaggio al server con tipologia *EXIT*, per notificare il server dell'uscita del nodo dalla rete. Infine viene eseguito *exit(0)* per terminare l'applicazione.
- *start_download_chunks(file_owners, filename)*: metodo che si occupa del download del file per chunks da diversi nodi in parallelo. È suddiviso in 5 fasi:
 1. Richiesta della dimensione del file con l'invocazione del metodo *ask_file_size*.
 2. Divisione della dimensione del file equamente tra i diversi nodi possessori del file, calcolando i range di byte per ogni nodo.
 3. Lancio di un thread per ogni nodo, che si occupa di scaricare il suo chunk di byte del file, con il metodo *receive_chunk*. Viene effettuata la *join* su ogni thread per aspettare che tutti i nodi terminano l'invio del chunk.
 4. Ordinamento dei chunks arrivati dai nodi con il metodo *sort_downloaded_chunks*.
 5. Riassemblaggio del file, andando man a mano ad appendere i pezzi di chunks ordinati con il metodo *reassemble_file*.

Infine viene rimosso il thread corrente dalla lista *download_threads*.

- *receive_chunk(filename: str, range: tuple, file_owner: tuple)*: è il metodo che si occupa di ricevere il rispettivo chunk assegnato al nodo. Consiste nell'inviare inizialmente un messaggio al nodo possessore del file (*file_owner*) con tipologia *CHUNKS*, così che il *file_owner* può avviare l'invio del chunk. Quindi viene creato un socket UDP su una porta random, e all'interno di un ciclo *while* vengono letti i byte inviati su tale socket ed appesi alla lista dei pezzi di chunk associata al file *filename* (*downloaded files*). Se il messaggio che arriva ha come campo *idx* il valore *-1*, il nodo

capisce che la trasmissione del chunk è terminata; allora invia un ack TCP al possessore del file per avvisare la ricevuta di fine ricezione chunk.

- *send_chunk(msg, ip_addr_dest, conn)*: è il metodo che si occupa di inviare il chunk del file al nodo richiedente. Inizialmente si divide il chunk in pezzi perché la trasmissione UDP ha un limite massimo del buffer di byte di invio, pari a *UDP_BUFFER_SIZE*; la divisione del chunk avviene con il metodo *split_file_to_chunks* che ritorna la lista dei pezzi di byte del chunk. Dopo di ciò si crea il socket UDP con una porta casuale per l'invio dei pezzi di chunks al nodo richiedente. L'invio di questi avviene all'interno di un ciclo *for* che itera sulla lista dei pezzi di chunk, e ad ogni iterazione crea un messaggio di tipo *ChunkSharing*, con valore del campo *idx* pari all'indice del pezzo di chunk che si sta inviando; si invia quindi il messaggio. Alla fine dei pezzi da inviare, il nodo invia un messaggio con *idx* pari a -1 per notificare la fine dell'invio del chunk; tale invio viene ripetuto ogni 2 secondi, finché non viene ricevuto l'ack TCP da parte del nodo ricevente; per concludere viene rimosso il thread corrente dalla lista *upload_threads*.
- *split_file_to_chunks(file_path: str, range: tuple)*: è un metodo semplice, dato in input il percorso del file (*file: path*) e il range che definisce il chunk del file da inviare (*range*), ritorna la lista dei pezzi del chunk. Per fare ciò viene aperto il file, e presi tutti i byte del chunk definito da *range*, vengono a loro volta suddivisi in pezzi di dimensione pari alla dimensione massima dei pezzi di chunk (*CHUNK_PIECES_MAX*), che è sicuramente minore della dimensione del buffer UDP (*UDP_BUFFER_SIZE*).
- *sort_downloaded_chunks(filename: str)*: è il metodo per riordinare l'insieme dei chunk compresi i pezzi di chunk provenienti dai diversi nodi; questo perché non si ha alcuna garanzia con la connessione UDP. Il dizionario *downloaded_files* con chiave *filename* restituisce la lista dei messaggi che sono stati inviati per quel file. Il metodo consiste nel riordinare tale lista in base all'attributo *range*, attributo che identifica un singolo chunk. Infine, a sua volta, si ordinano i pezzi identificati dal campo *idx*, facenti parte dello stesso *range* e quindi dello stesso chunk.
- *reassemble_file(chunks: list, file_path: str)*: è il metodo finale che riassembla il file a partire dalla lista di byte (*chunks*). Viene semplicemente aperto il file definito nel percorso *file_path*, e viene effettuata la *write* su tale file di ogni byte presente nella lista *chunks*.

3. Docker Engine

L'utilizzo di **Docker Engine** era richiesto dalla traccia del progetto; come detto nell'introduzione, il suo utilizzo è servito per realizzare una rete tra container, dove nei container saranno in esecuzione i codici dei nodi e del server. Nel seguito saranno approfonditi alcuni aspetti quali: immagini, container, rete, volumi, esecuzione e reset dei container.

3.1. Immagini

Le **immagini docker** create sono due, una per quanto riguarda il **server** e l'altra per il **nodo**. La creazione delle immagini viene effettuata attraverso l'esecuzione del file `config.sh` scritto in `bash`. Tale file effettua innanzitutto la rimozione di tutte le immagini presenti in *Docker Engine* attraverso l'istruzione: `docker rmi $(docker images -a -q)`. Dopo di ciò crea le due immagini con i seguenti comandi:

- `docker build --rm -t server -f SERVER/Dockerfile .` : per creare l'immagine del server; con `-t` si specifica il nome che deve avere l'immagine (in questo caso `server`), mentre con `-f` si specifica il path del `Dockerfile`.
- `docker build --rm -t node -f NODE/Dockerfile .` : per creare l'immagine del nodo; in questo caso l'immagine è stata giustamente denominata `node`.

Come si vede dai comandi, la creazione delle immagine avviene grazie a due *Dockerfile*, rispettivamente uno per il server e uno per il nodo.

Il **Dockerfile** del **server**:

```
1 FROM python:latest
2 LABEL Maintainer="flv"
3 WORKDIR /usr/app/src
4 COPY APP_P2P_FILE_SHARING ./
5 CMD [ "python", "./server.py"]
```

- `FROM python:latest`: istruzione per iniziare la costruzione dell'immagine server a partire da un'immagine python già esistente, presente nel docker hub.
- `WORKDIR /usr/app/src`: istruzione per impostare la cartella `src` come cartella corrente all'interno del container; in modo che le prossime istruzioni saranno eseguite su tale cartella.
- `COPY APP_P2P_FILE_SHARING .`: istruzione per copiare il contenuto della cartella `APP_P2P_FILE_SHARING` (cartella contenente tutti i sorgenti python) all'interno della cartella corrente nel container (quindi la cartella `src`).
- `CMD ["python", "./server.py"]`: istruzione che esegue da linea di comando del container `pyhon server.py`; altro non fa che eseguire all'avvio del container il codice del server.

Il **Dockerfile** del **nodo**:

```
1 FROM python:latest
2 LABEL Maintainer="flv"
3 WORKDIR /usr/app/src
4 COPY APP_P2P_FILE_SHARING ./
5 CMD [ "python", "./node.py"]
```

Molte istruzioni sono identiche alle istruzioni nel *Dockerfile* del server; cambia solamente l'ultima ovvero: `CMD ["python", "./node.py"]` che esegue il file *node.py* all'avvio del container.

3.2. Container

Per quanto riguarda l'esecuzione delle immagini e quindi i container, si è fatto uso di **docker-compose**, dovendo gestire una molteplicità di container. Per tale motivo si è creato il file di configurazione di tali container: *docker-compose.yml*.

All'interno del *docker-compose.yml* sono definiti 11 servizi ed una rete. In particolare 1 servizio per quanto riguarda il container del server, mentre gli altri 10 servizi per i container dei nodi. Il motivo per cui sono stati creati 10 servizi per i nodi e non solamente uno è dovuto al semplice fatto di dover aprire per ciascuno un terminale separato; altrimenti se i container dei nodi avrebbero dovuto lavorare in background, la scelta migliore sarebbe stata quella di creare un singolo servizio per il nodo, e mandare in esecuzione 10 istanze di quel servizio grazie al parametro `---scale node=10` di *docker-compose*. La soluzione adottata, se da un lato permette l'esecuzione dei nodi su più terminali interattivi nello stesso sistema operativo, dall'altro ha come limitazione il fatto di poter eseguire al massimo 10 nodi ed inoltre aumenta la lunghezza del file *docker-compose.yml*. Se si volessero eseguire più di 10 nodi, va modificato quest'ultimo file e aggiungere più servizi.

Il **servizio** del **server**:

```
4 server:
5   image: server:latest
6   tty: true           # docker run -t
7   networks:
8     | docker-network-test:
9     | | ipv4_address: 172.20.0.2
10  volumes:
11  | - ./LOGS/server:/usr/app/src/logs
```

- *image: server:latest*: istruzione per specificare l'immagine del container da eseguire; in questo caso giustamente l'immagine *server* descritta in precedenza.
- *tty: true*: istruzione che permette la visualizzazione dell'output del codice *server.py* anche sul terminale del sistema operativo host.
- *ipv4_address: 172.20.0.2*: **istruzione importante**; serve per applicare manualmente un indirizzo ip al container del server. Questa operazione è necessaria, perché si suppone che il server abbia un indirizzo ip fisso e conosciuto da tutti gli altri nodi; infatti è lo stesso indirizzo di quello descritto nelle costanti della classe *Configs*.

- `./LOGS/server:/usr/app/src/logs`: istruzione che permette di costruire un volume in cui sono condivise la cartella `LOGS/server` nel sistema operativo host con la cartella `/usr/app/src/logs` del container. Ovvero la cartella in cui saranno salvati i log del server.

Il **servizio** del **nodo** (si veda ad esempio il *node1*, perché gli altri sono simili):

```
12     node1:
13         image: node:latest
14         stdin_open: true # docker run -i
15         tty: true        # docker run -t
16         networks:
17         |   - docker-network-test
18         volumes:
19         |   - ./VOLUMES/node1:/usr/app/src/node_files
20         |   - ./LOGS/node1:/usr/app/src/logs
```

- `image: node:latest`: istruzione uguale a quella simmetrica descritta in precedenza, soltanto che adesso l'immagine è quella per il nodo.
- `tty: true`: istruzione che permette la visualizzazione dell'output del codice `node.py` anche sul terminale del sistema operativo host.
- `stdin_open: true`: istruzione che permette l'utilizzo dello stdin da parte del sistema operativo host. Aspetto importante in quanto si vuole lanciare un terminale dal sistema operativo host per mandare comandi all'applicazione del nodo.
- `./LOGS/node1:/usr/app/src/logs`: istruzione per il volume identica a quella già descritta per il server.
- `./VOLUMES/node1:/usr/app/src/node_files`: istruzione per la creazione di un secondo volume; questo mette in condivisione la cartella del sistema operativo host `VOLUMES/node1` con la cartella del container `/usr/app/src/node_files`. Tale volume è utile, perché in questo modo si possono visualizzare direttamente nel sistema operativo host i file posseduti e scaricati dal nodo; inoltre è possibile copiare dei file locali nel file system del container, semplicemente trasferendoli all'interno del volume.

Da notare che a differenza del servizio server, i servizi dei nodi non hanno bisogno di specificare un indirizzo ip statico, proprio per simulare un comportamento reale, in cui l'indirizzo ip non è statico ma può cambiare al riavvio della macchina.

3.3. Rete

La **rete** è uno degli aspetti principali per cui si è utilizzato *Docker Engine*. Sono disponibili diverse tipologie di rete; nel caso in esame, la tipologia che più si addice al progetto è quella anche più usata in *Docker Engine*, ossia la rete **Bridge**.

Tale rete consiste in una sorta di rete LAN tra container, dove ogni container ha un indirizzo di rete prelevato da una prefissata sottorete; e quindi ogni container collegato a tale rete può raggiungere l'altro. Da notare che il sistema operativo host invece è isolato dai container.

La configurazione della rete è stata effettuata anch'essa all'interno del file `docker-compose.yml`:

```
112 networks:
113     docker-network-test:
114         driver: bridge
115         ipam:
116             driver: default
117             config:
118                 - subnet: 172.20.0.0/16
119                   gateway: 172.20.0.1
```

- `docker-network-test`: è il nome della rete da creare.
- `driver: bridge`: specifica la tipologia di rete, che come appena detto è `bridge`.
- `subnet: 172.20.0.0/16`: specifica la subnet della rete; in particolare una subnet che va dall'indirizzo 172.20.0.0 all'indirizzo 172.20.255.255; quindi possono collegarsi un numero di container pari a $2^{16}-2$.
- `gateway: 172.20.0.1`: indirizzo che può servire per la comunicazione con il sistema operativo host.

Da notare che al server è stato assegnato l'indirizzo 172.20.0.2, che è il primo disponibile nella rete; e tutti gli altri nodi prenderanno un indirizzo nella subnet 172.20.0.0/16.

3.4. Volumi

I **volumi** creati permettono di visualizzare all'interno del file system del sistema operativo locale i file presenti all'interno del file system dei container. In particolare sono presenti la cartella `LOGS` e la cartella `VOLUMES`, con all'interno tante cartelle quanti sono i nodi e una cartella per il server. All'interno di queste cartelle è possibile modificare e visualizzare file presenti all'interno dei container in base al mapping nel file system del container.

Per tale motivo, per ogni servizio, esiste un volume che mette in condivisione la cartella dei logs dei container con una cartella del sistema operativo; così da poter leggere i log direttamente dal sistema operativo host. Inoltre, solamente per i nodi, è stato creato un altro volume che mette in condivisione una cartella del file system locale del sistema operativo con la cartella dei container che contiene i files del nodo; così facendo è possibile inserire file all'interno di un nodo direttamente dal sistema operativo host, oppure si possono visionare i file scaricati senza entrare nel container del nodo.

La creazione dei volumi avviene direttamente nel file `docker-compose.yml`, nella sezione `volumes` dei servizi. Nel dettaglio l'istruzione è stata già spiegata nel paragrafo 3.2. *Container*.

3.5. Esecuzione dei container

Per eseguire i container è stato creato un file apposta per l'esecuzione dell'applicazione: **`run.sh`**.

Il file `run.sh` prevede l'uso dei seguenti argomenti:

- `-s`: specifica l'esecuzione del container del server con il numero (`-s 1`).
- `-n`: specifica il numero dei container dei nodi da eseguire (es. `-n 5`).

- `-nn`: specifica il nome del container del nodo da eseguire (es. `-nn node5`).

Il file `run.sh` fa uso di un file nascosto `.conf`, in cui salva il nome dei nodi in esecuzione, così da non rieseguire gli stessi nodi, e da non eseguire un numero di nodi totali che supera 10 nodi.

Per l'esecuzione del **server** si usa la seguente istruzione:

```
konsole --geometry 500x350+0+0 -p tabtitle='SERVER' -e docker compose run --rm --name server server &
```

konsole: è il comando che permette di avviare un terminale; esso prende i seguenti argomenti:

- `--geometry`: per specificare la dimensione del terminale e la sua posizione nel desktop (es. `--geometry 500x350+0+0` specifica un terminale di dimensione (500x350)px posizionato in alto a sinistra).
- `-p tabtitle`: per definire un nome al terminale (es. `-p tabtitle='SERVER'` specifica il nome SERVER al terminale).
- `-e`: per specificare il comando da eseguire appena viene avviato il terminale; in questo caso si vuole avviare il container del server con il comando `docker compose run --rm --name server server`.
- `&`: per eseguire il comando **konsole** in background e non bloccare il resto del programma `run.sh`.

Per l'esecuzione dei **nodi** si usa la seguente istruzione richiamata all'interno di un ciclo for che itera per il numero di nodi stabilito con `-n`:

```
konsole --geometry 480x350+$x+$y -p tabtitle='node'$i -e docker compose run --rm --name node$i node$i &
```

I comandi sono gli stessi di quelli usati per il server, da notare che si fa uso di `$i` che indica il contatore del nodo corrente; e anche si fa uso di `$x` e `$y`. Queste ultime due variabili sono width ed height rispettivamente della posizione del terminale i-esimo; infatti grazie ad esse, che vengono calcolate in modo opportuno, si potranno vedere i terminali dei nodi uno di fianco l'altro ordinati come in figura.



3.6. Reset dei container

Per terminare l'esecuzione, visto che può essere sgradevole chiudere manualmente ogni terminale di ogni nodo, è stato scritto il file **reset.sh** che permette la terminazione di tutti i container in esecuzione e la loro cancellazione.

Il file prende come argomento opzionale "-v"; se specificato viene rimpiazzata la cartella *VOLUMES* con la cartella nascosta *.volumes_start*. Questa operazione è utile se si vuole ripristinare lo stato dei file che possiedono i nodi, infatti la cartella *.volumes_start* contiene i file iniziale dei nodi, e andando a sostituirla alla cartella *VOLUMES*, in automatico i nodi riavranno i loro file iniziali.

Dopo di ciò vengono cancellati tutti i log nella cartella *LOGS*, ripristinandola semplicemente con una cartella vuota (perché le cartelle per ogni nodo vengono create in automatico quando si creano i volumi).

Viene eliminato il file *.conf*, così che il file *run.sh* alla prossima esecuzione saprà che non ci sono altri nodi in esecuzione.

Infine vengono eseguiti i seguenti comandi docker:

- `docker kill $(docker ps -q)`: comando per terminare tutti i container in esecuzione.
- `docker rm $(docker ps -a -q)`: comando per eliminare tutti i container.

Lanciati questi due comandi, i terminali ancora aperti si chiuderanno in automatico non essendoci al loro interno alcun programma in esecuzione (se il flag *—hold* del comando *konsole* era attivo, allora i terminali sarebbero rimasti aperti).

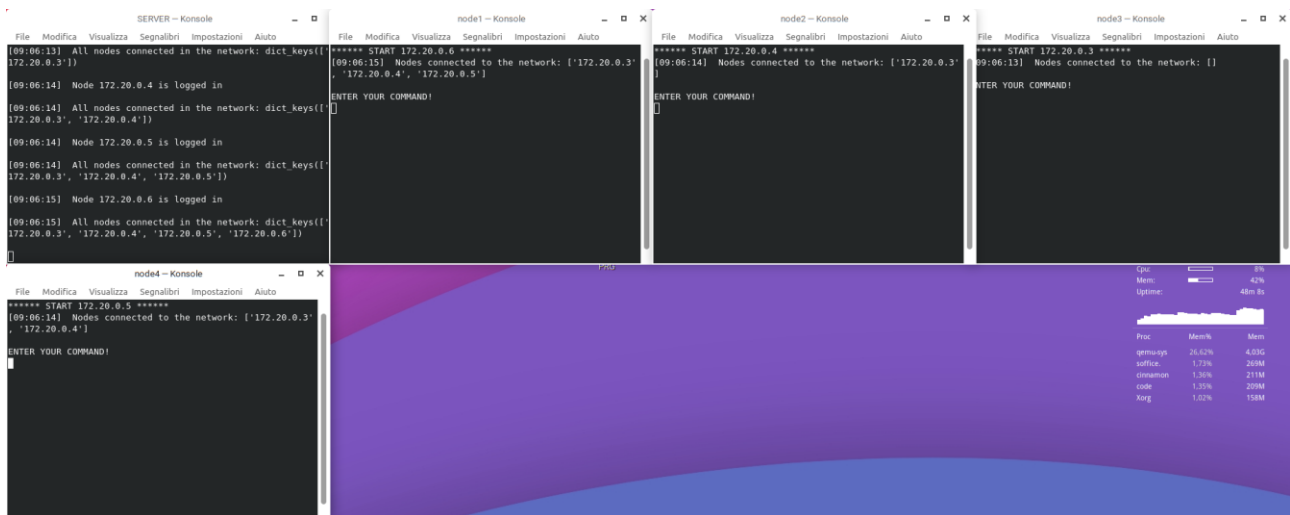
4. Esecuzione

In questo capitolo verranno mostrati alcuni **esempi di esecuzione** dell'applicazione. Innanzitutto è presente il file **README.txt** in cui sono descritte brevemente le linee guida per eseguire l'intera applicazione, eccole riportate:

```
1 1. RUN file 'config.sh' to build image of server and node from dockerfile.
2 2. RUN file 'run.sh':
3   2.1. run.sh -s 1 --> mandatory starting server.
4   2.2. run.sh -n x --> x is the number of node from 1 to 10.
5   2.3. run.sh -nn N --> N is the name of a node (e.g. node5).
6 3. PLAY and ENJOY with P2P File Sharing APP!
7 4. RUN file 'reset.sh' to remove temporary file and to stop and delete docker containers.
```

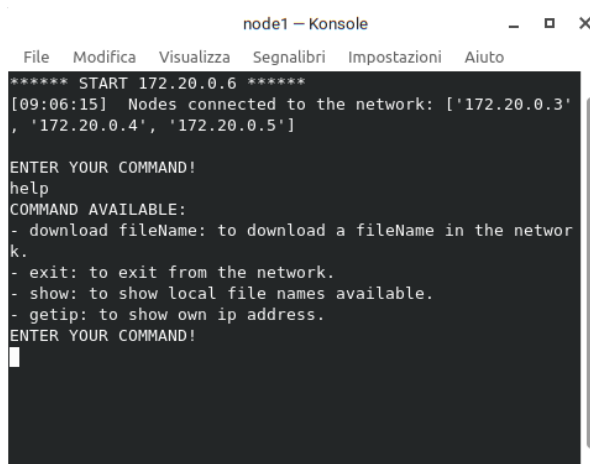
Verrà mostrato il caso ad **esempio** in cui si vogliono eseguire **4 nodi**; ecco i comandi da eseguire:

- `./reset.sh -v`: per sicurezza si effettua un reset compreso dei volumi.
- `./config.sh`: per la creazione delle due immagini.
- `./run.sh -s 1 -n 4`: vengono mandati in esecuzione un server e 4 nodi, per tale motivo si apriranno in modo ordinato 5 terminali come in figura.

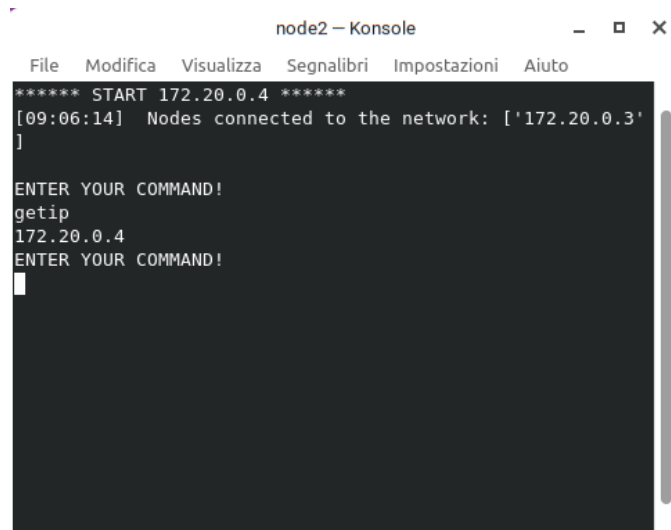


Da questo momento in poi si può testare l'applicazione, lanciando alcuni comandi sui nodi. Eseguendo ad esempio i comandi:

- **help** su uno qualsiasi dei nodi, restituisce la lista dei comandi che è possibile usare.

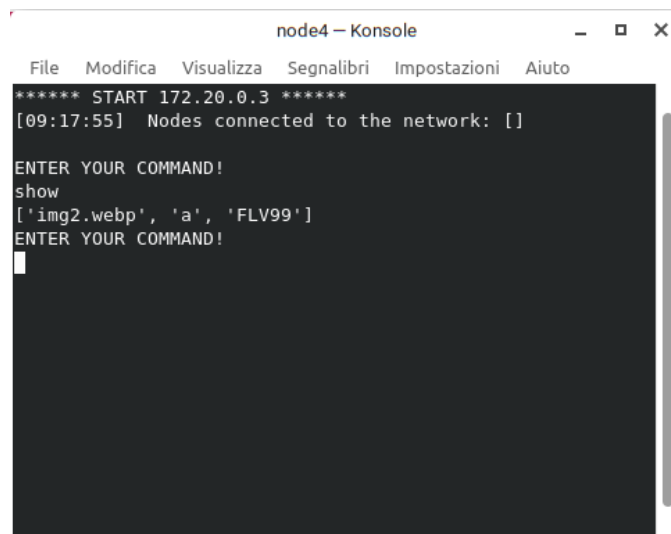


- **getip** sul nodo1 si può vedere che restituisce l'indirizzo ip di quel nodo (corrisponde all'indirizzo ip presenta nella prima riga, appena avviato il terminale).

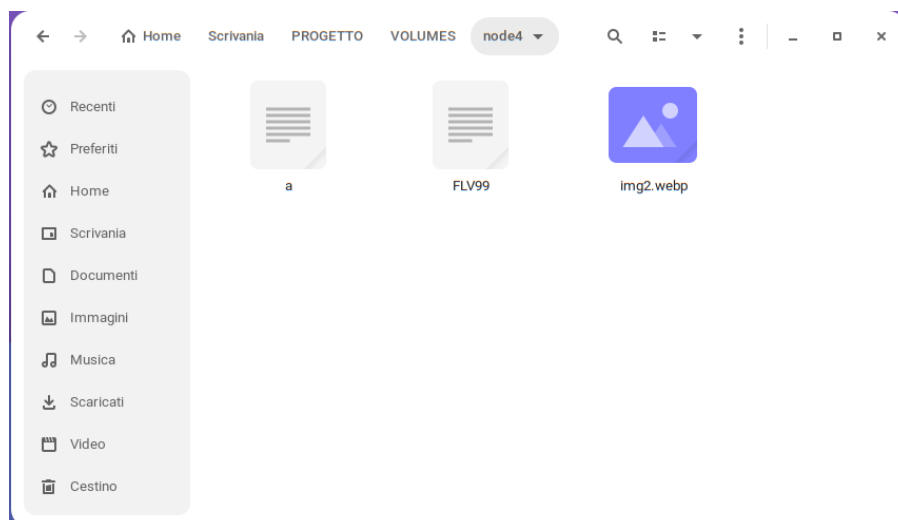


```
node2 - Konsole
File Modifica Visualizza Segnalibri Impostazioni Aiuto
***** START 172.20.0.4 *****
[09:06:14] Nodes connected to the network: ['172.20.0.3']
ENTER YOUR COMMAND!
getip
172.20.0.4
ENTER YOUR COMMAND!
```

- **show** sul nodo1 si possono vedere i file che esso possiede; si può effettuare la verifica che siano esatti, vedendo che combaciano con i file presenti nella cartella *VOLUMES/node1*.

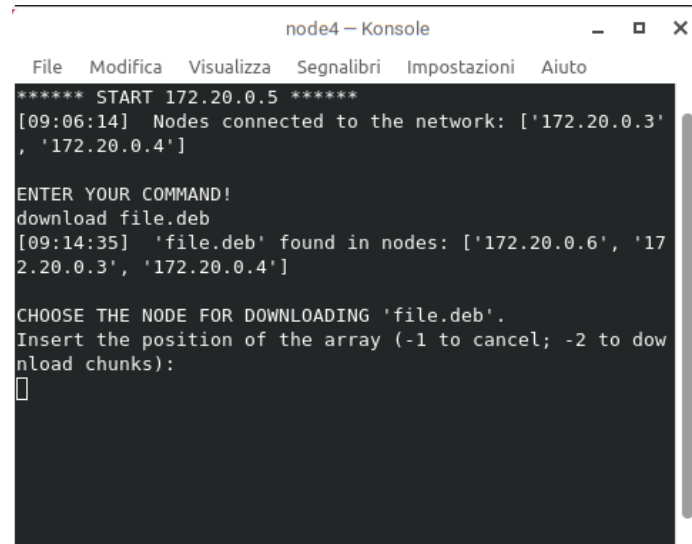


```
node4 - Konsole
File Modifica Visualizza Segnalibri Impostazioni Aiuto
***** START 172.20.0.3 *****
[09:17:55] Nodes connected to the network: []
ENTER YOUR COMMAND!
show
['img2.webp', 'a', 'FLV99']
ENTER YOUR COMMAND!
```



È possibile effettuare il comando `show` anche per i nodi 2, 3 e 4, o di conseguenza vedere i file in `VOLUMES/node2`, `VOLUMES/node3` e `VOLUMES/node4`.

- Come si vede, il file "`file.deb`" è presente nei nodi 1, 2 e 3 ma non nel nodo 4; si può pensare di scaricarlo nel nodo4: all'interno del terminale del nodo4 si esegue il seguente comando → **download file.deb**.



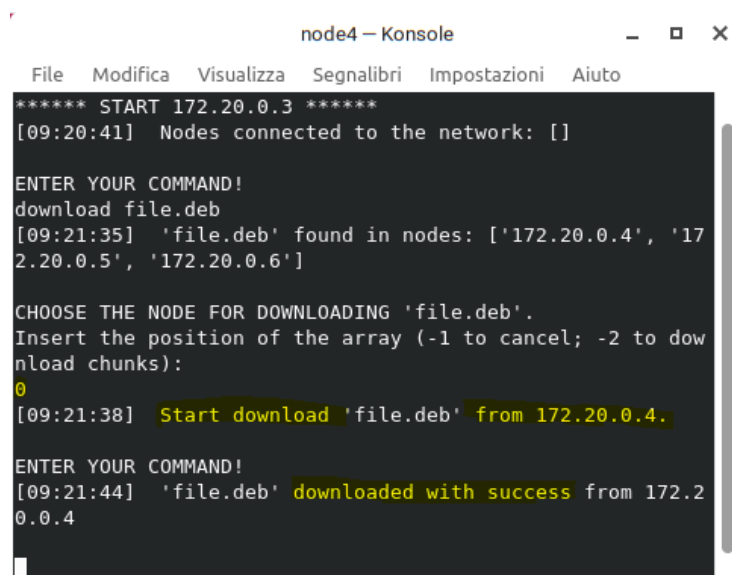
```
node4 - Konsole
File  Modifica  Visualizza  Segnalibri  Impostazioni  Aiuto
***** START 172.20.0.5 *****
[09:06:14] Nodes connected to the network: ['172.20.0.3', '172.20.0.4']

ENTER YOUR COMMAND!
download file.deb
[09:14:35] 'file.deb' found in nodes: ['172.20.0.6', '172.20.0.3', '172.20.0.4']

CHOOSE THE NODE FOR DOWNLOADING 'file.deb'.
Insert the position of the array (-1 to cancel; -2 to download chunks):
█
```

Viene mostrata a schermo la lista degli indirizzi ip dei nodi che possiedono già il file "`file.deb`", ovvero i nodi 1,2 e 3; inoltre chiede di scegliere il nodo da cui scaricare il file attraverso l'**indice di array** della lista (0,1 o 2), oppure **-1** per annullare il download, o ancora **-2** per effettuare lo scaricamento simultaneo dai diversi 3 peer, ognuno di essi invia un chunk del file.

- Prima di tutto si verifica il download classico, ovvero l'intero file da un singolo nodo: ad esempio si digita il numero **0**, e si attende la fine del download (da notare che il download viene gestito da un altro thread, quindi volendo si potrebbero eseguire più download in parallelo. Alla fine del download, si può verificare l'avvenuto scaricamento o con il solito comando `show`, oppure direttamente nella cartella `VOLUMES/node4`, così da poter aprire il file nel caso o controllarne la dimensione.

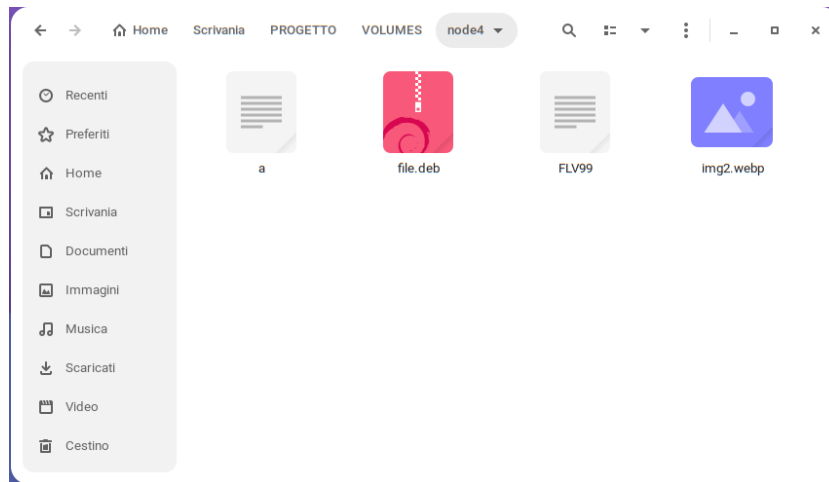


```
node4 - Konsole
File  Modifica  Visualizza  Segnalibri  Impostazioni  Aiuto
***** START 172.20.0.3 *****
[09:20:41] Nodes connected to the network: []

ENTER YOUR COMMAND!
download file.deb
[09:21:35] 'file.deb' found in nodes: ['172.20.0.4', '172.20.0.5', '172.20.0.6']

CHOOSE THE NODE FOR DOWNLOADING 'file.deb'.
Insert the position of the array (-1 to cancel; -2 to download chunks):
0
[09:21:38] Start download 'file.deb' from 172.20.0.4.

ENTER YOUR COMMAND!
[09:21:44] 'file.deb' downloaded with success from 172.20.0.4
█
```



Per testare anche il download parallelo sui diversi nodi del file diviso in chunk, si può effettuare il test sullo stesso file, andando però prima ad eliminarlo manualmente nella cartella *VOLUMES/node4*, altrimenti l'applicazione dirà che il file è già presente localmente.

- Si riesegue il comando *download file.deb*; e questa volta quando viene mostrata la lista dei peer che posseggono il file, si digita il numero **-2**, esplicitando quindi, il download parallelo dai diversi nodi. Alla fine del download si possono effettuare gli stessi test descritti in precedenza sul controllo del file scaricato.

```
node4 - Konsole
File Modifica Visualizza Segnalibri Impostazioni Aiuto

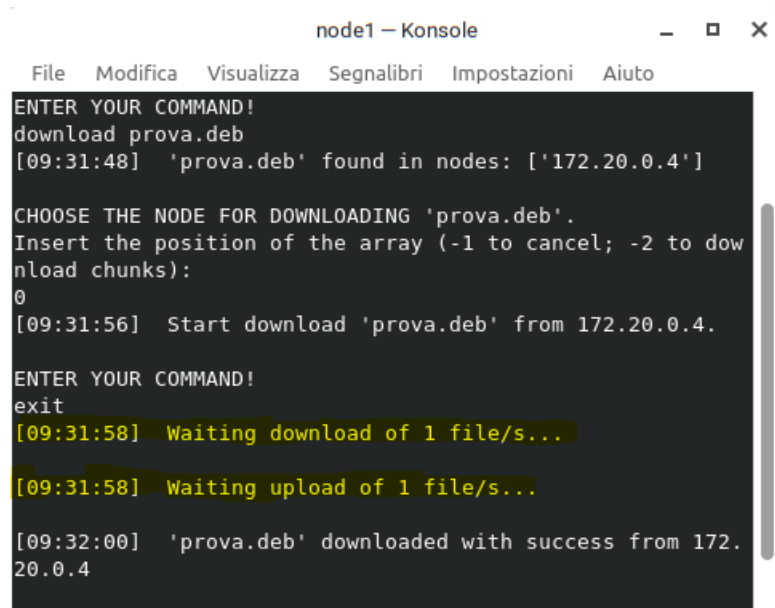
ENTER YOUR COMMAND!
download file.deb
[09:26:28] 'file.deb' found in nodes: ['172.20.0.6', '172.20.0.5', '172.20.0.3']

CHOOSE THE NODE FOR DOWNLOADING 'file.deb'.
Insert the position of the array (-1 to cancel; -2 to download chunks):
-2
[09:26:32] Start download chunks of 'file.deb' from ['172.20.0.6', '172.20.0.5', '172.20.0.3'].

ENTER YOUR COMMAND!
[09:26:32] The file 'file.deb' which you are about to download, has size of 393775740 bytes

[09:26:47] 'file.deb' has successfully downloaded and saved in my files directory.
```

- Si possono fare tanti ulteriori test, come ad esempio scaricare più file contemporaneamente, oppure eseguire il comando *exit* per uscire dalla rete; in questo caso, un aspetto importante, è che se il nodo che sta uscendo sta inviando uno o più file, oppure sta scaricando uno o più file, quest'ultimo aspetterà la loro terminazione prima di uscire, come si vede in figura:



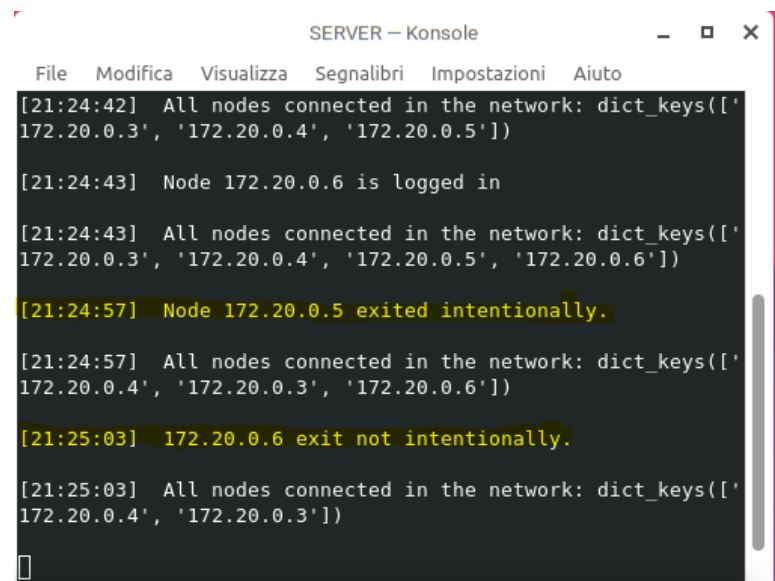
```
node1 - Konsole
File Modifica Visualizza Segnalibri Impostazioni Aiuto
ENTER YOUR COMMAND!
download prova.deb
[09:31:48] 'prova.deb' found in nodes: ['172.20.0.4']

CHOOSE THE NODE FOR DOWNLOADING 'prova.deb'.
Insert the position of the array (-1 to cancel; -2 to download chunks):
0
[09:31:56] Start download 'prova.deb' from 172.20.0.4.

ENTER YOUR COMMAND!
exit
[09:31:58] Waiting download of 1 file/s...
[09:31:58] Waiting upload of 1 file/s...

[09:32:00] 'prova.deb' downloaded with success from 172.20.0.4
```

- Ulteriori test possono essere le uscite intenzionali o non intenzionali dei nodi, e visualizzare i messaggi del server che notifica l'uscita di un nodo intenzionalmente o non. Per ulteriori debug visualizzare i log all'interno della cartella LOGS.



```
SERVER - Konsole
File Modifica Visualizza Segnalibri Impostazioni Aiuto
[21:24:42] All nodes connected in the network: dict_keys(['172.20.0.3', '172.20.0.4', '172.20.0.5'])
[21:24:43] Node 172.20.0.6 is logged in
[21:24:43] All nodes connected in the network: dict_keys(['172.20.0.3', '172.20.0.4', '172.20.0.5', '172.20.0.6'])
[21:24:57] Node 172.20.0.5 exited intentionally.
[21:24:57] All nodes connected in the network: dict_keys(['172.20.0.4', '172.20.0.3', '172.20.0.6'])
[21:25:03] 172.20.0.6 exit not intentionally.
[21:25:03] All nodes connected in the network: dict_keys(['172.20.0.4', '172.20.0.3'])
```

Alla fine dell'esecuzione dell'intera applicazione utilizzare il comando `./reset.sh` per effettuare una pulizia ed interrompere i container in esecuzione.

5. Codici sorgenti

5.1. server.py

```
1. # built-in libraries
2. from threading import Thread, Timer
3. from collections import defaultdict
4. import json
5. import datetime
6. import time
7. import warnings
8. warnings.filterwarnings("ignore")
9.
10. # implemented classes
11. from utils import *
12. from messages.message import Message
13. from messages.server2node import Server2Node
14. from configs import CFG, Config
15. config = Config.from_json(CFG)
16.
17. next_call = time.time()
18.
19. class Server:
20.     def __init__(self):
21.         self.server_socket = set_socket_TCP(config.constants.SERVER_ADDR[1],
22.                                             addr=config.constants.SERVER_ADDR
23.                                             [0])
24.         self.peers_status = defaultdict(bool)
25.         self.peers = defaultdict(tuple)
26.     def check_peers_periodically(self, interval: int):
27.         global next_call
28.         alive_nodes_ids = set()
29.         dead_nodes_ids = set()
30.         try:
31.             for node, has_informed in self.peers_status.items():
32.                 # it means the node has informed the server that is still
33.                 connected
34.                 if has_informed:
35.                     self.peers_status[node] = False
36.                     alive_nodes_ids.add(node)
37.                 else:
38.                     dead_nodes_ids.add(node)
39.                     self.peers_status.pop(node, None)
40.                     self.peers.pop(node, None)
41.                 # the dictionary size maybe changed during iteration, so we check
42.                 nodes in the next time step
43.         except RuntimeError:
44.             pass
```



```
43.
44.     if len(dead_nodes_ids) != 0:
45.         log_content = f"{node} exit not intentionally."
46.         log(content=log_content, is_server=True, printing=True)
47.         log_content = f"All nodes connected in the network:
{self.peers.keys()}"
48.         log(content=log_content, is_server=True, printing=True)
49.
50.     if not (len(alive_nodes_ids) == 0 and len(dead_nodes_ids) == 0):
51.         log_content = f"Node(s) {list(alive_nodes_ids)} is in the network
52.                        and node(s){list(dead_nodes_ids)} have left."
53.         log(content=log_content, is_server=True)
54.
55.     datetime.now()
56.     next_call = next_call + interval
57.     Timer(next_call - time.time(), self.check_peers_periodically,
args=(interval,)).start()
58.
59.     def register_and_send_peers(self, conn, msg: dict, addr: tuple):
60.         first_time = False
61.         if addr[0] not in self.peers.keys():
62.             first_time = True
63.             log_content = f"Node {addr[0]} is logged in"
64.             log(content=log_content, is_server=True, printing=True)
65.
66.             self.peers.pop(addr[0] , None)
67.             #send list of connected peers
68.             server_response = Server2Node(search_result =
list(self.peers.items()))
69.             conn.send(server_response.encode())
70.             #add new peer to the list of peers
71.             self.peers[addr[0]] = (msg['request_port_TCP'],
msg['download_port_TCP'])
72.             self.peers_status[addr[0]] = True
73.             if first_time:
74.                 log_content = f"All nodes connected in the network:
{self.peers.keys()}"
75.                 log(content=log_content, is_server=True, printing=True)
76.
77.
78.     def handle_node_request(self, conn, addr: tuple):
79.         data = conn.recv(config.constants.TCP_BUFFER_SIZE)
80.         msg = Message.decode(data)
81.         mode = msg['mode']
82.         if mode == config.server_requests_mode.REGISTER:
83.             self.register_and_send_peers(conn, msg=msg, addr=addr)
84.         elif mode == config.server_requests_mode.EXIT:
85.             self.peers.pop(addr[0] , None)
86.             self.peers_status.pop(addr[0] , None)
```

```
87.         log_content = f"Node {addr[0]} exited intentionally."
88.         log(content=log_content, is_server=True, printing=True)
89.         log_content = f"All nodes connected in the network:
{self.peers.keys()}"
90.         log(content=log_content, is_server=True, printing=True)
91.         conn.close()
92.
93.     def run(self):
94.         log_content = f"***** Server program started just right
now! *****"
95.         log(content=log_content, is_server=True)
96.
97.         #timer thread to check if peers are connected
98.         timer_thread = Thread(target=self.check_peers_periodically,
args=(config.constants.SERVER_TIME_INTERVAL,))
99.         timer_thread.setDaemon(True)
100.        timer_thread.start()
101.
102.        while True:
103.            self.server_socket.listen()
104.            conn, addr = self.server_socket.accept()
105.            t = Thread(target=self.handle_node_request, args=(conn,
addr))
106.            t.start()
107.
108.        if __name__ == '__main__':
109.            t = Server()
110.            t.run()
```

5.2. node.py

```
1. # built-in libraries
2. from cmath import e
3. from multiprocessing import Event
4. from socket import timeout
5. from messages.chunk_sharing import ChunkSharing
6. from utils import *
7. import argparse
8. from threading import Thread, Timer
9. from operator import itemgetter
10. import datetime
11. import time
12. from itertools import groupby
13. import mmap
14. import warnings
15. warnings.filterwarnings("ignore")
16. from threading import current_thread
17.
```

```
18.# implemented classes
19.from configs import CFG, Config
20.config = Config.from_json(CFG)
21.from messages.message import Message
22.from messages.node2server import Node2Server
23.from messages.node2node import Node2Node
24.
25.next_call = time.time()
26.
27.class Node:
28.    def __init__(self, request_port_TCP: int, download_port_TCP: int):
29.        self.request_port_TCP = request_port_TCP
30.        self.ip_addr = socket.gethostbyname(socket.gethostname())
31.        self.request_socket = set_socket_TCP(request_port_TCP,
        addr=self.ip_addr)
32.        self.download_port_TCP = download_port_TCP
33.        self.download_socket = set_socket_TCP(download_port_TCP,
        addr=self.ip_addr)
34.        self.peers = []
35.        self.download_threads = []
36.        self.upload_threads = []
37.        self.downloaded_files = {}
38.
39.
40.    def receive_chunk(self, filename: str, range: tuple, file_owner: tuple):
41.        udp_port = generate_random_port()
42.        msg = Node2Node(filename=filename,
        mode=config.node_requests_mode.CHUNKS, range=range, portUDP=udp_port)
43.        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
44.        client.connect(file_owner)
45.        client.send(msg.encode())
46.        #client.close()
47.
48.        log_content = f"I sent a request for a chunk of {filename} for node
        {file_owner[0]}"
49.        log(content=log_content)
50.
51.        temp_sock = set_socket_UDP(udp_port, addr=self.ip_addr)
52.
53.        while True:
54.            data, addr = temp_sock.recvfrom(config.constants.UDP_BUFFER_SIZE)
55.            msg = Message.decode(data)
56.            if msg["idx"] == -1: # end of the file
57.                client.send(data)
58.                free_socket(temp_sock)
59.                client.close()
60.                return
61.            self.downloaded_files[filename].append(msg)
62.
```

```
63.     def split_file_to_chunks(self, file_path: str, rng: tuple) -> list:
64.         with open(file_path, "r+b") as f:
65.             mm = mmap.mmap(f.fileno(), 0)[rng[0]: rng[1]]
66.             # we divide each chunk to a fixed-size pieces to be transferable
67.             piece_size = config.constants.CHUNK_PIECES_SIZE
68.             return [mm[p: p + piece_size] for p in range(0, rng[1] - rng[0],
piece_size)]
69.
70.     def send_chunk(self, msg, ip_addr_dest, conn):
71.         filename = msg['filename']
72.         range = msg['range']
73.         dest_port = msg['portUDP']
74.         file_path = f"{config.directory.node_files_dir}{filename}"
75.         chunk_pieces = self.split_file_to_chunks(file_path=file_path,
rng=range)
76.
77.         client = set_socket_UDP(generate_random_port(), addr=self.ip_addr)
78.         for idx, p in enumerate(chunk_pieces):
79.             msg = ChunkSharing(filename=filename, range=range, idx=idx,
chunk=p)
80.             client.sendto(Message.encode(msg), (ip_addr_dest, dest_port))
81.             # tell the node that sending has finished (idx = -1)
82.             msg = ChunkSharing(filename=filename, range=range)
83.             conn.settimeout(2.0)
84.             while True:
85.                 client.sendto(Message.encode(msg), (ip_addr_dest, dest_port))
86.                 try:
87.                     data = conn.recv(config.constants.TCP_BUFFER_SIZE)
88.                     break
89.                 except timeout:
90.                     pass
91.
92.             log_content = f"The process of sending a chunk to node
'{ip_addr_dest}' of file {filename} has finished!"
93.             log(content=log_content)
94.             conn.close()
95.             free_socket(client)
96.             self.upload_threads.remove(current_thread())
97.
98.     def sort_downloaded_chunks(self, filename: str) -> list:
99.         sort_result_by_range = sorted(self.downloaded_files[filename],
key=itemgetter("range"))
100.        group_by_range = groupby(sort_result_by_range, key=lambda i:
i["range"])
101.        sorted_downloaded_chunks = []
102.        for key, value in group_by_range:
103.            value_sorted_by_idx = sorted(list(value),
key=itemgetter("idx"))
104.            sorted_downloaded_chunks.append(value_sorted_by_idx)
```

```
105.
106.         return sorted_downloaded_chunks
107.
108.     def reassemble_file(self, chunks: list, file_path: str):
109.         with open(file_path, "bw+") as f:
110.             for ch in chunks:
111.                 f.write(ch)
112.                 f.flush()
113.                 f.close()
114.
115.     def start_download_chunks(self, file_owners, filename):
116.         # 2. Ask file size
117.         file_size = self.ask_file_size(filename=filename,
118. file_owners=file_owners)
119.         if file_size == 0:
120.             content_log = f"ERROR to get file size of '{filename}'"
121.             log(content=content_log, printing=True)
122.             return
123.             log_content = f"The file '{filename}' which you are about to
124. download, has size of {file_size} bytes"
125.             log(content=log_content, printing=True)
126.
127.         # 2. Split file equally among nodes to download chunks of it
128.         from them
129.         step = file_size / len(file_owners)
130.         chunks_ranges = [(round(step*i), round(step*(i+1))) for i in
131. range(len(file_owners))]
132.
133.         # 3. Create a thread for each node to get a chunk from it
134.         self.downloaded_files[filename] = []
135.         neighboring_peers_threads = []
136.         for idx, obj in enumerate(file_owners):
137.             t = Thread(target=self.receive_chunk, args=(filename,
138. chunks_ranges[idx], obj))
139.             t.setDaemon(True)
140.             t.start()
141.             neighboring_peers_threads.append(t)
142.         for t in neighboring_peers_threads:
143.             t.join()
144.
145.         log_content = f"All the chunks of '{filename}' has downloaded
146. from neighboring peers. But they must be reassembled!"
147.         log(content=log_content)
148.
149.         # 4. Sort chunks of file.
150.         sorted_chunks = self.sort_downloaded_chunks(filename=filename)
151.         log_content = f"All chunks of the '{filename}' is now sorted
152. and ready to be reassembled."
153.         log(content=log_content)
```

```
147.
148.     # 5. Assemble the chunks to re-build the file
149.     total_file = []
150.     file_path = f"{config.directory.node_files_dir}{filename}"
151.     for chunk in sorted_chunks:
152.         for piece in chunk:
153.             total_file.append(piece["chunk"])
154.     self.reassemble_file(chunks=total_file, file_path=file_path)
155.     log_content = f"'{filename}' has successfully downloaded and
saved in my files directory."
156.     log(content=log_content, printing=True)
157.
158.     self.download_threads.remove(current_thread())
159.
160. def start_download(self, file_owner, filename):
161.     peer_addr, peer_download_port_TCP = file_owner
162.     msg = Node2Node(filename=filename)
163.     client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
164.     client.connect((peer_addr, peer_download_port_TCP))
165.     client.send(msg.encode())
166.
167.     file_path = f"{config.directory.node_files_dir}{filename}"
168.     file = open(file_path, "wb")
169.
170.     # Receive any data from client side
171.     recv_data = client.recv(config.constants.TCP_BUFFER_SIZE)
172.     while recv_data:
173.         file.write(recv_data)
174.         recv_data = client.recv(config.constants.TCP_BUFFER_SIZE)
175.     file.close()
176.     client.close()
177.     self.download_threads.remove(current_thread())
178.     content_log = f"'{filename}' downloaded with success from
{file_owner[0]}"
179.     log(content=content_log, printing=True)
180.
181. def get_pos_node(self, file_owners):
182.     try:
183.         pos = int(input())
184.     except ValueError:
185.         pos=-10
186.     while pos<0 or pos>= len(file_owners):
187.         if pos== -1 or pos== -2:
188.             return pos
189.         print("Error! Insert the position again of the array (-1 to
cancel; -2 to download chunks):")
190.     try:
191.         pos = int(input())
192.     except ValueError:
```

```
193.             pos=-10
194.         return pos
195.
196.     def download(self, filename: str):
197.         file_path = f"{config.directory.node_files_dir}{filename}"
198.         if os.path.isfile(file_path):
199.             log_content = f"You already have this file!"
200.             log(content=log_content, printing=True)
201.             return
202.         else:
203.             log_content = f"You just started to download {filename}.
Let's search it in the network!"
204.             log(content=log_content)
205.             file_owners = self.search_file_owners(filename=filename)
206.             if len(file_owners)>0:
207.                 log_content = f"{filename}' found in nodes: {[ip for
(ip,_) in file_owners]}]"
208.                 log(content=log_content, printing=True)
209.                 print(f"CHOOSE THE NODE FOR DOWNLOADING '{filename}'")
210.                 print("Insert the position of the array (-1 to cancel;
-2 to download chunks):")
211.
212.                 pos = self.get_pos_node(file_owners)
213.                 if pos == -1:
214.                     log_content = f"Downloading canceled!"
215.                     log(content=log_content, printing=True)
216.                     return
217.
218.                 if pos == -2:
219.                     log_content = f"Start download chunks of
'{filename}' from {[ip for (ip,_) in file_owners]}."
220.                     log(content=log_content, printing=True)
221.                     t = Thread(target=self.start_download_chunks,
args=(file_owners, filename,))
222.                     t.setDaemon(True)
223.                     t.start()
224.                     self.download_threads.append(t)
225.                     return
226.
227.                     log_content = f"Start download '{filename}' from
{file_owners[pos][0]}."
228.                     log(content=log_content, printing=True)
229.                     t = Thread(target=self.start_download,
args=(file_owners[pos], filename,))
230.                     t.setDaemon(True)
231.                     t.start()
232.                     self.download_threads.append(t)
233.                 else:
234.                     log_content = f"{filename}' not found in the network."
```

```
235.             log(content=log_content, printing=True)
236.
237.     def search_file_owners(self, filename: str):
238.         file_owners = []
239.         for (peer_addr, (peer_req_port_TCP, peer_download_port_TCP)) in
self.peers:
240.             msg = Node2Node(filename=filename)
241.             try:
242.                 client = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
243.                 client.connect((peer_addr, peer_req_port_TCP))
244.                 client.send(msg.encode())
245.                 data = client.recv(config.constants.TCP_BUFFER_SIZE)
246.                 server_msg = Message.decode(data)
247.                 if server_msg['filename'] != '':
248.                     file_owners.append((peer_addr,
peer_download_port_TCP))
249.                 client.close()
250.             except ConnectionRefusedError:
251.                 log_content = f"Unable to connect to {peer_addr}"
252.                 log(content=log_content, printing=True)
253.
254.         return file_owners
255.
256.     def fetch_owned_files(self) -> list:
257.         files = []
258.         node_files_dir = config.directory.node_files_dir
259.         if os.path.isdir(node_files_dir):
260.             _, _, files = next(os.walk(node_files_dir))
261.         else:
262.             os.makedirs(node_files_dir)
263.
264.         return files
265.
266.     def check_file(self, conn, addr: tuple):
267.         data = conn.recv(config.constants.TCP_BUFFER_SIZE)
268.         msg = Message.decode(data)
269.         filename = msg['filename']
270.         if filename not in self.fetch_owned_files():
271.             content_log = f"You don't have '{filename}' request from
{addr}"
272.             log(content=content_log)
273.             filename = ''
274.
275.             msg = Node2Node(filename=filename)
276.             conn.send(msg.encode())
277.             conn.close()
278.
279.     def send_file(self, conn, msg):
```



```
280.         filename = msg['filename']
281.         file_path = f"{config.directory.node_files_dir}{filename}"
282.         file = open(file_path, "rb")
283.
284.         send_data = file.read(config.constants.TCP_BUFFER_SIZE)
285.         while send_data:
286.             conn.send(send_data)
287.             send_data =
288.                 file.read(config.constants.TCP_BUFFER_SIZE)
289.             file.close()
290.             conn.close()
291.             self.upload_threads.remove(current_thread())
292.
293.         def ask_file_size(self, filename: str, file_owners) -> int:
294.             msg = Node2Node(filename=filename,
295.                 mode=config.node_requests_mode.SIZE)
296.             size=0
297.             for file_owner in file_owners:
298.                 try:
299.                     client = socket.socket(socket.AF_INET,
300.                         socket.SOCK_STREAM)
301.                     client.connect(file_owner)
302.                     client.send(msg.encode())
303.                     data = client.recv(config.constants.TCP_BUFFER_SIZE)
304.                     server_msg = Message.decode(data)
305.                     size = server_msg['size']
306.                     client.close()
307.                 except ConnectionRefusedError:
308.                     log_content = f"Unable to connect to {file_owner[0]}
309.                         for request of file size."
310.                     log(content=log_content)
311.             return size
312.
313.         def tell_file_size(self, msg, conn):
314.             filename = msg['filename']
315.             file_path = f"{config.directory.node_files_dir}{filename}"
316.             file_size = os.stat(file_path).st_size
317.             response_msg = Node2Node(filename=filename, size=file_size)
318.             conn.send(response_msg.encode())
319.             conn.close()
320.             if current_thread() in self.upload_threads:
321.                 self.upload_threads.remove(current_thread())
322.
323.         def inform_server_periodically(self, interval: int):
324.             global next_call
325.             self.enter_network(f"I informed the server that I'm still
326.                 alive!")
327.
328.             datetime.datetime.now()
```

```
323.         next_call = next_call + interval
324.         Timer(next_call - time.time(), self.inform_server_periodically,
args=(interval,)).start()
325.
326.     def receive_request_download_from_nodes(self):
327.         self.download_socket.listen()
328.         while True:
329.             conn, addr = self.download_socket.accept()
330.             data = conn.recv(config.constants.TCP_BUFFER_SIZE)
331.             msg = Message.decode(data)
332.             if msg['mode'] == config.node_requests_mode.SIZE:
333.                 t = Thread(target=self.tell_file_size, args=(msg,
conn,))
334.             elif msg['mode'] == config.node_requests_mode.CHUNKS:
335.                 t = Thread(target=self.send_chunk, args=(msg, addr[0],
conn,))
336.             else:#download
337.                 t = Thread(target=self.send_file, args=(conn, msg,))
338.                 t.start()
339.                 self.upload_threads.append(t)
340.
341.
342.     def receive_request_search_from_nodes(self):
343.         self.request_socket.listen()
344.         while True:
345.             conn, addr = self.request_socket.accept()
346.             t = Thread(target=self.check_file, args=(conn, addr,))
347.             t.start()
348.
349.
350.     def exit_network(self):
351.         wait_download_threads = self.download_threads.copy()
352.         wait_upload_threads = self.upload_threads.copy()
353.         if len(self.download_threads) > 0:
354.             log_content = f"Waiting download of
{len(self.download_threads)} file/s..."
355.             log(content=log_content, printing=True)
356.         if len(self.upload_threads) > 0:
357.             log_content = f"Waiting upload of
{len(self.upload_threads)} file/s..."
358.             log(content=log_content, printing=True)
359.         for t in wait_download_threads:
360.             t.join()
361.         for t in wait_upload_threads:
362.             t.join()
363.
364.         msg = Node2Server(mode=config.server_requests_mode.EXIT,
request_port_TCP=self.request_port_TCP,
download_port_TCP=self.download_port_TCP)
```

```
367.
368.         client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
369.         client.connect(tuple(config.constants.SERVER_ADDR))
370.         client.send(msg.encode())
371.         client.close()
372.
373.         log_content = f"You exited the torrent!"
374.         log(content=log_content)
375.
376.     def enter_network(self, log_content):
377.         msg = Node2Server(mode=config.server_requests_mode.REGISTER,
378.                             request_port_TCP = self.request_port_TCP,
379.                             download_port_TCP=self.download_port_TCP)
380.         client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
381.         client.connect(tuple(config.constants.SERVER_ADDR))
382.         client.send(msg.encode())
383.
384.         log(content=log_content)
385.
386.         data = client.recv(config.constants.TCP_BUFFER_SIZE)
387.         server_msg = Message.decode(data)
388.         self.peers = server_msg['search_result']
389.         if log_content == "You entered.":
390.             log_content = f"Nodes connected to the network: {[a for a,b
in self.peers]]}"
391.             log(content=log_content, printing=True)
392.             client.close()
393.
394.     def run(self):
395.         log_content = f"***** Node program started just
right now! *****"
396.         log(content=log_content)
397.         print(f"***** START {self.ip_addr} *****")
398.         node.enter_network(f"You entered.")
399.
400.         # Create a thread to periodically informs the server to tell it
is still connected.
401.         timer_thread = Thread(target=node.inform_server_periodically,
args=(config.constants.NODE_TIME_INTERVAL,))
402.         timer_thread.setDaemon(True)
403.         timer_thread.start()
404.
405.         # Create a thread to listen request of looking for a file from
others peers.
406.         request_thread =
Thread(target=node.receive_request_search_from_nodes, args=())
407.         request_thread.setDaemon(True)
408.         request_thread.start()
409.
```

```
410.         # Create a thread to listen request of download a file from
         others peers.
411.         download_thread =
         Thread(target=node.receive_request_download_from_nodes, args=())
412.         download_thread.setDaemon(True)
413.         download_thread.start()
414.
415.         while True:
416.             print("ENTER YOUR COMMAND!")
417.             command = input()
418.             mode, filename = parse_command(command)
419.             ##### download mode #####
420.             if mode == 'download':
421.                 node.download(filename)
422.             ##### exit mode #####
423.             elif mode == 'exit':
424.                 node.exit_network()
425.                 exit(0)
426.             ##### show mode #####
427.             elif mode == 'show':
428.                 print(self.fetch_owned_files())
429.             ##### getip mode #####
430.             elif mode == 'getip':
431.                 print(self.ip_addr)
432.             ##### help mode #####
433.             elif mode == 'help':
434.                 print("COMMAND AVAILABLE:")
435.                 print("- download fileName: to download a fileName in
         the network.")
436.                 print("- exit: to exit from the network.")
437.                 print("- show: to show local file names available.")
438.                 print("- getip: to show own ip address.")
439.             else:
440.                 print("Error! Invalid command.")
441.
442.         if __name__ == '__main__':
443.             node = Node(request_port_TCP=generate_random_port(),
444.                         download_port_TCP=generate_random_port())
445.             node.run()
446.
```

5.3. utils.py

```
1. import socket
2. import random
3. import warnings
4. import os
5. from datetime import datetime
6. from configs import CFG, Config
```

```
7. config = Config.from_json(CFG)
8.
9. # global variables
10. used_ports = []
11.
12. def set_socket_TCP(port: int, addr = 'localhost') -> socket.socket:
13.     '''
14.     This function creates a new TCP socket
15.
16.     :param port: port number
17.     :return: A socket object with an unused port number
18.     '''
19.     sock = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
20.     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
21.     sock.bind((addr, port))
22.     used_ports.append(port)
23.
24.     return sock
25.
26. def set_socket_UDP(port: int, addr = 'localhost') -> socket.socket:
27.     '''
28.     This function creates a new UDP socket
29.
30.     :param port: port number
31.     :return: A socket object with an unused port number
32.     '''
33.     sock = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
34.     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
35.     sock.bind((addr, port))
36.     used_ports.append(port)
37.
38.     return sock
39.
40. def free_socket(sock: socket.socket):
41.     '''
42.     This function free a socket to be able to be used by others
43.
44.     :param sock: socket
45.     :return:
46.     '''
47.     used_ports.remove(sock.getsockname()[1])
48.     sock.close()
49.
50. def generate_random_port() -> int:
51.     '''
52.     This function generates a new(unused) random port number
53.
54.     :return: a random integer in range of [1024, 65535]
55.     '''
```

```
56.     available_ports = config.constants.AVAILABLE_PORTS_RANGE
57.     rand_port = random.randint(available_ports[0], available_ports[1])
58.     while rand_port in used_ports:
59.         rand_port = random.randint(available_ports[0], available_ports[1])
60.
61.     return rand_port
62.
63. def parse_command(command: str):
64.     '''
65.     This function parses the input command
66.
67.     :param command: A string which is the input command.
68.     :return: Command parts (mode, filename)
69.     '''
70.     parts = command.split(' ')
71.     if len(parts) <= 0 or len(parts) > 2:
72.         return "", ""
73.     mode = parts[0]
74.     filename = ""
75.     if len(parts) == 2:
76.         filename = parts[1]
77.     return mode, filename
78.
79.
80. def log(content: str, is_server=False, printing=False) -> None:
81.     '''
82.     This function is used for logging
83.
84.     :param content: content to be written
85.     :return:
86.     '''
87.     if not os.path.exists(config.directory.logs_dir):
88.         os.makedirs(config.directory.logs_dir)
89.
90.     # time
91.     now = datetime.now()
92.     current_time = now.strftime("%H:%M:%S")
93.
94.     content = f"[{current_time}] {content}\n"
95.     if printing:
96.         print(content)
97.
98.     if is_server:
99.         node_logs_filename = config.directory.logs_dir + 'server.log'
100.     else:
101.         node_logs_filename = config.directory.logs_dir + 'node' +
102.         '.log'
103.     if not os.path.exists(node_logs_filename):
104.         with open(node_logs_filename, 'w') as f:
```

```
104.         f.write(content)
105.         f.close()
106.     else:
107.         with open(node_logs_filename, 'a') as f:
108.             f.write(content)
109.             f.close()
```

5.4. configs.py

```
1.  """configs in json format"""
2.  import json
3.  import socket
4.
5.  CFG = {
6.      "directory": {
7.          "logs_dir": "logs/",
8.          "node_files_dir": "node_files/",
9.      },
10.     "constants": {
11.         "AVAILABLE_PORTS_RANGE": (1024, 65535), # range of available ports on
the local computer
12.         "SERVER_ADDR": ('172.20.0.2', 12345),
13.         "TCP_BUFFER_SIZE": 8192,
14.         "UDP_BUFFER_SIZE": 9216,
15.         "CHUNK_PIECES_SIZE": 9216 - 2000,
16.         "NODE_TIME_INTERVAL": 10,             # the interval time that each node
periodically informs the server (in seconds)
17.         "SERVER_TIME_INTERVAL": 10            #the interval time that the server
periodically checks which nodes are connected (in seconds)
18.     },
19.     "server_requests_mode": {
20.         "REGISTER": 0, # tells the server that it is in the network
21.         "EXIT": 1      # tells the server that it left the network
22.     },
23.     "node_requests_mode": {
24.         "SIZE": 0,      # tells request of file size
25.         "CHUNKS": 1,    # tells request of download file in chunks
26.         "DOWNLOAD": 2   # tells request of download total file
27.     }
28. }
29.
30. class Config:
31.     """Config class which contains directories, constants, etc."""
32.
33.     def __init__(self, directory, constants, node_requests_mode,
server_requests_mode):
34.         self.directory = directory
35.         self.constants = constants
36.         self.server_requests_mode = server_requests_mode
```

```
37.         self.node_requests_mode = node_requests_mode
38.
39.     @classmethod
40.     def from_json(cls, cfg):
41.         """Creates config from json"""
42.         params = json.loads(json.dumps(cfg), object_hook=HelperObject)
43.         return cls(params.directory, params.constants,
44.                    params.node_requests_mode, params.server_requests_mode)
45.
46. class HelperObject(object):
47.     """Helper class to convert json into Python object"""
48.     def __init__(self, dict_):
49.         self.__dict__.update(dict_)
```

5.5. message.py

```
1. from __future__ import annotations
2. import pickle
3.
4. class Message:
5.     def __init__(self):
6.         pass
7.
8.     def encode(self) -> bytes:
9.         return pickle.dumps(self.__dict__)
10.
11.     @staticmethod
12.     def decode(data: bytes) -> dict:
13.         return pickle.loads(data)
```

5.6. node2node.py

```
1. from messages.message import Message
2. from configs import CFG, Config
3. config = Config.from_json(CFG)
4.
5. class Node2Node(Message):
6.     def __init__(self, filename: str,
7.                  mode=config.node_requests_mode.DOWNLOAD, size=-1, range=None, portUDP=None):
8.         super().__init__()
9.         self.mode = mode
10.        self.filename = filename
11.        self.size = size
12.        self.range = range
13.        self.portUDP = portUDP
```


5.7. node2server.py

```
1. from messages.message import Message
2.
3. class Node2Server(Message):
4.     def __init__(self, mode: int, request_port_TCP: int, download_port_TCP:
       int):#, filename: str):
5.
6.         super().__init__()
7.         self.mode = mode
8.         self.request_port_TCP = request_port_TCP
9.         self.download_port_TCP = download_port_TCP
```

5.8. server2node.py

```
1. from messages.message import Message
2.
3. class Server2Node(Message):
4.     def __init__(self, search_result: list):#, filename: str):
5.
6.         super().__init__()
7.         self.search_result = search_result
```

5.9. chunk_sharing.py

```
1. from messages.message import Message
2.
3. class ChunkSharing(Message):
4.     def __init__(self, filename: str, range: tuple, idx: int ==-1, chunk:
       bytes = None):
5.
6.         super().__init__()
7.         self.filename = filename
8.         self.range = range
9.         self.idx = idx
10.        self.chunk = chunk
11.
```

5.10. docker-compose.yml

```
1. version: '2.4'
2.
3. services:
4.     server:
5.         image: server:latest
6.         tty: true           # docker run -t
7.         networks:
8.             docker-network-test:
```

```
9.         ipv4_address: 172.20.0.2
10.     volumes:
11.         - ./LOGS/server:/usr/app/src/logs
12.     node1:
13.         image: node:latest
14.         stdin_open: true # docker run -i
15.         tty: true        # docker run -t
16.         networks:
17.             - docker-network-test
18.         volumes:
19.             - ./VOLUMES/node1:/usr/app/src/node_files
20.             - ./LOGS/node1:/usr/app/src/logs
21.
22.     node2:
23.         image: node:latest
24.         stdin_open: true # docker run -i
25.         tty: true        # docker run -t
26.         networks:
27.             - docker-network-test
28.         volumes:
29.             - ./VOLUMES/node2:/usr/app/src/node_files
30.             - ./LOGS/node2:/usr/app/src/logs
31.
32.     node3:
33.         image: node:latest
34.         stdin_open: true # docker run -i
35.         tty: true        # docker run -t
36.         networks:
37.             - docker-network-test
38.         volumes:
39.             - ./VOLUMES/node3:/usr/app/src/node_files
40.             - ./LOGS/node3:/usr/app/src/logs
41.
42.     node4:
43.         image: node:latest
44.         stdin_open: true # docker run -i
45.         tty: true        # docker run -t
46.         networks:
47.             - docker-network-test
48.         volumes:
49.             - ./VOLUMES/node4:/usr/app/src/node_files
50.             - ./LOGS/node4:/usr/app/src/logs
51.
52.     node5:
53.         image: node:latest
54.         stdin_open: true # docker run -i
55.         tty: true        # docker run -t
56.         networks:
57.             - docker-network-test
```

```
58.   volumes:
59.     - ./VOLUMES/node5:/usr/app/src/node_files
60.     - ./LOGS/node5:/usr/app/src/logs
61.
62.   node6:
63.     image: node:latest
64.     stdin_open: true # docker run -i
65.     tty: true        # docker run -t
66.     networks:
67.       - docker-network-test
68.     volumes:
69.       - ./VOLUMES/node6:/usr/app/src/node_files
70.       - ./LOGS/node6:/usr/app/src/logs
71.
72.   node7:
73.     image: node:latest
74.     stdin_open: true # docker run -i
75.     tty: true        # docker run -t
76.     networks:
77.       - docker-network-test
78.     volumes:
79.       - ./VOLUMES/node7:/usr/app/src/node_files
80.       - ./LOGS/node7:/usr/app/src/logs
81.
82.   node8:
83.     image: node:latest
84.     stdin_open: true # docker run -i
85.     tty: true        # docker run -t
86.     networks:
87.       - docker-network-test
88.     volumes:
89.       - ./VOLUMES/node8:/usr/app/src/node_files
90.       - ./LOGS/node8:/usr/app/src/logs
91.
92.   node9:
93.     image: node:latest
94.     stdin_open: true # docker run -i
95.     tty: true        # docker run -t
96.     networks:
97.       - docker-network-test
98.     volumes:
99.       - ./VOLUMES/node9:/usr/app/src/node_files
100.      - ./LOGS/node9:/usr/app/src/logs
101.
102.     node10:
103.       image: node:latest
104.       stdin_open: true # docker run -i
105.       tty: true        # docker run -t
106.       networks:
```

```
107.         - docker-network-test
108.     volumes:
109.         - ./VOLUMES/node10:/usr/app/src/node_files
110.         - ./LOGS/node10:/usr/app/src/logs
111.
112.     networks:
113.         docker-network-test:
114.             driver: bridge
115.             ipam:
116.                 driver: default
117.                 config:
118.                     - subnet: 172.20.0.0/16
119.                       gateway: 172.20.0.1
120.
```

5.11. config.sh

```
1. #!/bin/bash
2. docker rmi $(docker images -a -q)
3. docker build --rm -t server -f SERVER/Dockerfile .
4. docker build --rm -t node -f NODE/Dockerfile .
```

5.12. reset.sh

```
1. #!/bin/bash
2. if [ $# -eq 1 ];then
3.     if [ "$1" = "-v" ];then
4.         rm -r VOLUMES
5.         cp -r .volumes_start VOLUMES
6.     fi
7. fi
8. rm -r LOGS
9. mkdir LOGS
10.rm .conf
11.docker kill $(docker ps -q)
12.docker rm $(docker ps -a -q)
```

5.13. run.sh

```
1. #!/bin/bash
2.
3. SERVER=0
4. NODES=0
5. NUMBER=0
6. NODE_NAME=""
7.
8. if [ -f .conf ];then
```

```
9.     NUMBER=$(wc -w < .conf)
10.fi
11.
12.echo "$NUMBER NODE/S are running."
13.
14.while test $# -gt 0; do
15.    case "$1" in
16.        -h|--help)
17.            echo "$package - attempt to capture frames"
18.            echo " "
19.            echo "$package [options] application [arguments]"
20.            echo " "
21.            echo "options:"
22.            echo "-h, --help          show brief help"
23.            echo "-s, --server        specify server run"
24.            echo "-n, --nodes=NUM     specify number of nodes to run
    (default=5)"
25.            echo "-nn, --nodes name=NAME specify name of node to run (e.g.
    'node4')"
26.            exit 0
27.            ;;
28.        -s)
29.            shift
30.            if [ $# -gt 0 ];then
31.                if [ $1 -eq 1 ];then
32.                    export SERVER=$1
33.                else
34.                    echo "ERROR: -s only require number 1!"
35.                    exit 1
36.                fi
37.            else
38.                echo "ERROR: -s only require number 1!"
39.                exit 1
40.            fi
41.            shift
42.            ;;
43.        -n)
44.            shift
45.            if test $# -gt 0; then
46.                if [ $1 -lt 1 ] || [ $1 -gt 10 ];then
47.                    echo "ERROR: -n number out of [0;10]!"
48.                    exit 1
49.                else
50.                    n=$((NUMBER+$1))
51.                    if [ $n -lt 11 ];then
52.                        export NODES=$1
53.                    else
54.                        echo "ERROR: -n number out of [0;10]!"
55.                        exit 1
```

```
56.         fi
57.     fi
58. else
59.     echo "ERROR: -n require a number!"
60.     exit 1
61. fi
62. shift
63. ;;
64. -nn)
65.     shift
66.     if test $# -eq 1; then
67.         export NODE_NAME=$1
68.     else
69.         echo "ERROR: -nn require a name node!"
70.         exit 1
71.     fi
72.     shift
73.     ;;
74. *)
75.     break
76.     ;;
77. esac
78. done
79.
80. if [ $SERVER -eq 1 ];then
81.     echo "RUN SERVER"
82.     konsole --geometry 500x350+0+0 -p tabtitle='SERVER' -e docker compose run
        --rm --name server server &
83.     time=$((5+$NODES))
84.     sleep $time
85. fi
86.
87. echo "RUN $NODES NODE/S"
88. count=0
89. for ((i=1; i<=10; i++)); do
90.     if [ $count -lt $NODES ];then
91.         if ! grep -q node$i ".conf";then
92.             x=$((($i % 4)*482))
93.             if [ $i -lt 4 ];then
94.                 y=0
95.             elif [ $i -lt 8 ];then
96.                 y=380
97.             else
98.                 y=760
99.             fi
100.             konsole --geometry 480x350+$x+$y -p tabtitle='node'$i -e
                docker compose run --rm --name node$i node$i &
101.             let count++
102.             echo node$i >> .conf
```

```
103.         fi
104.     else
105.         break
106.     fi
107. done
108.
109. if ! grep -q $NODE_NAME ".conf";then
110.     echo "RUN $NODE_NAME"
111.     konsole -p tabtitle=$NODE_NAME -e docker compose run --rm --name
        $NODE_NAME $NODE_NAME &
112.     echo $NODE_NAME >> .conf
113. else
114.     echo "ERROR! $NODE_NAME is already running"
115.     exit 1
116. fi
117.
```