PROGETTO DI FINE CORSO

PROGRAMMAZIONE DI RETI
ANNO 2021-2022

Traccia 2: Architettura client-server UDP per trasferimento file

Alni Riccardo
riccardo.alni@studio.unibo.it
0000971613

Analisi dei requisiti

Lo scopo del progetto è quello di progettare ed implementare in Python un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione con UDP come protocollo di strato di trasporto.

L'applicazione prevede l'interazione tra due entità: il client che manda messaggi di comando al server per fargli sapere che operazione vuole eseuire e il server che manda al client messaggi di risposta alle operazioni
Ci si pone come obiettivo la realizzazione di

- un comando "list" che restituisce al cliente la lista di file presenti nell'archivio del server e la dimensione di ciascuno
- un comando "get *_nomefile_*" che scarica nell'archivio del client il file selezionato presente nell'archivio server
- un comande "put *_nomefile_*" che carica il file selezionato dall'archivio del client all'archivio server

Il tutto viene gestito con due classi (server e client) e due folder (*server_storage* e *client_storage*)

Struttura generale delle due classi

Entrambe le classi usano i moduli socket, time (viene usata la funzione sleep) e os (per la gestione dei file)

```
import socket as sk
import time
import os
```

All'inizio entrambe la classi dichiarano quale tipo di socket utilizzano e memorizzano il percorso (valido per qualsiasi sistema operativo) in cui i due archivi sono presenti su disco (nel caso di *client_storage* la cartella se non è presente viene anche creata)

```
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
download_path = os.path.dirname(__file__) + os.path.sep + "client_storage" + os.path.sep
if not os.path.exists(download_path):
    os.mkdir(download_path)
```

```
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
storage_path = os.path.dirname(__file__) + os.path.sep + "server_storage" + os.path.sep
```

Le due classi procedono associando il socket alla porta (l'indirizzo del server è localhost)

Il codice seguente è contenuto in un try-catch per client così che segnali un'eventuale eccezione e con un finally viene chiusa la connessione lasciando libero il socket.

```
try:
    ...
except Exception as info:
    print (info)
finally:
    print("Ending connection")
    sock.close()
```

Nella classe server invece il codice è contenuto all'interno di un loop infinito (while True) in modo da rendere sempre possibile connettersi al server se un utente non è già connesso.

La connessione inizia con il client che manda un primo messaggio al server che risponde inviandogli un messaggio con la lista di operazioni possibili.

Server

```
Starting connection on  ('localhost', 10000)

Waiting for a client

Successfully connected to client  ('127.0.0.1', 54243)
```

Client

```
Trying to connect to server  ('localhost', 10000)

CONNECTION ESTABLISHED

Welcome, please select a command

1. list -> get all the files available in the server
2. get _filename_ -> download _filename_ from the server
3. put _filename_ -> upload a file to the server
```

Comando list

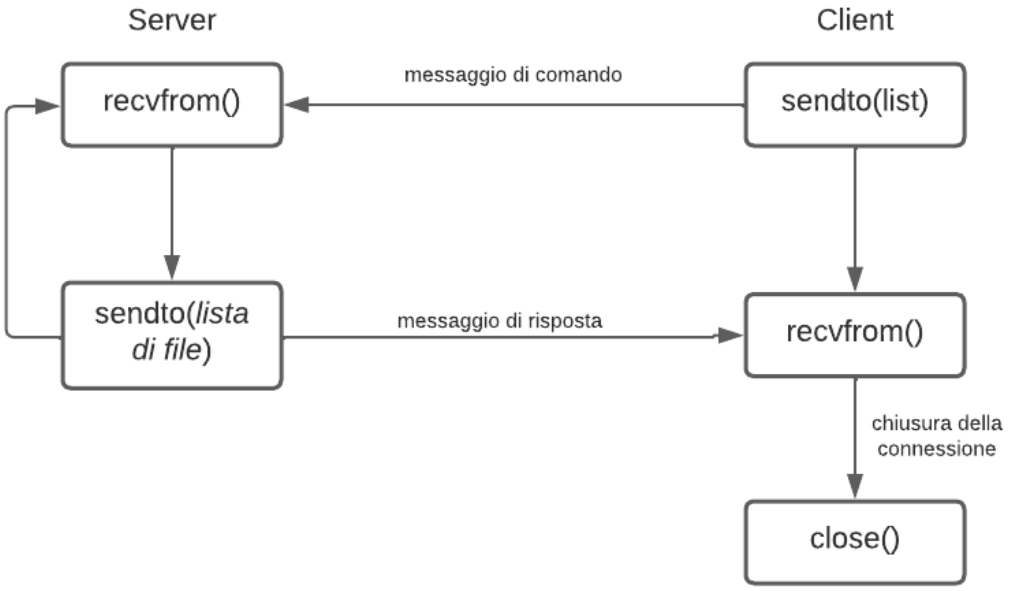
Il cliente può richiamare questo comando digitando "list" o "1" per ricevere dal server la lista di file presenti nell'archivio del server e la loro dimensione in KB

```
Starting connection on  ('localhost', 10000)

Waiting for a client

Successfully connected to client  ('127.0.0.1', 54243)
Client  ('127.0.0.1', 54243)  has chosen command 1-list

Waiting for a client
```

```
Trying to connect to server  ('localhost', 10000)

CONNECTION ESTABLISHED

Welcome, please select a command

1. list -> get all the files available in the server
2. get _filename_ -> download _filename_ from the server
3. put _filename_ -> upload a file to the server

list
bisez.m size: 0.6318359375 KB
messaggio.txt    size: 0.021484375 KB
nebulosa.jpg    size: 11.4033203125 KB

Ending connection
```

La lista viene gestita nel codice come una stringa e viene mandata in un unico messaggio, la stringa viene creata da un'apposita funzione getFiles() presente in server.py

```
def getFiles():
    files = ""
    for file in os.scandir(storage_path):
        files += file.name + "\tsize: "+ str(os.path.getsize(storage_path + file.name)/1024) + " KB\n"
    return files
```



Questa operazione esclude la presenza di errori quindi il messaggio di risposta può essere solo la lista di file, viene comunque controllata la sua integrità dal client.

Comando get

Il cliente può richiamare questo comando digitando “get *_filename_*” per scaricare nel suo archivio il file dal nome *_filename_* presente nell’archivio del server.

```
Successfully connected to client  ('127.0.0.1', 49163)
Client  ('127.0.0.1', 49163)  chose command 2-get messaggio.txt

File found...sending
File sent
```

```
get messaggio.txt
Starting download
File received
Download succesful

Ending connection
```

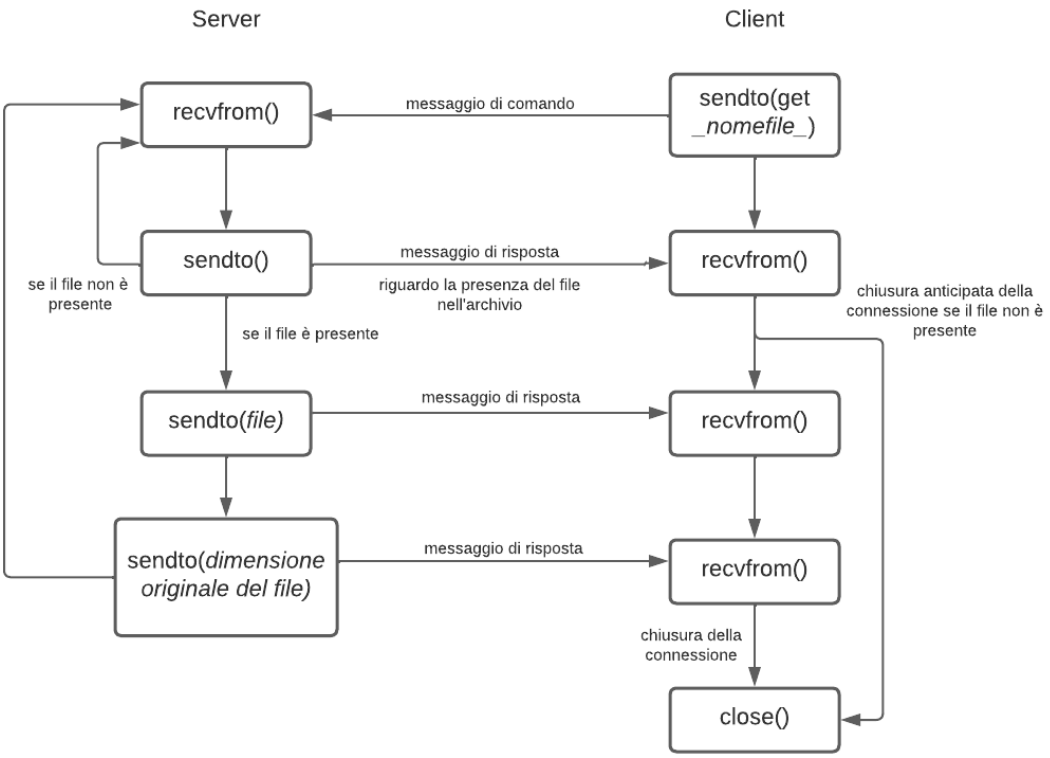
Dopo aver mandato il messaggio di comando, il server controlla che il file sia presente nel suo archivio e manda la risposta al cliente. In caso negativo la connessione viene interrotta.

```
Successfully connected to client  ('127.0.0.1', 49552)
Client  ('127.0.0.1', 49552)  chose command 2-get messaggio.jpg

File not found
```

```
get messaggio.jpg
The file does not exist
Ending connection
```

In caso contrario il server procede a spedire (attraverso diversi sendto() all’interno di un ciclo while) il file al cliente. Il ciclo finisce quando l’ultimo byte letto è nullo. Il server comunica la fine del download al cliente

mandandogli anche un messaggio contenente la dimensione originale del file, il cliente la userà per controllare che coincida con quella del file che ha ricevuto in modo da accertarsi che non ci siano state perdite.



Comando put

Il cliente può richiamare questo comando digitando “put *_filename_*” per caricare dal suo archivio il file dal nome *_filename_* all’interno dell’archivio del server. Questa operazione funziona quasi in maniera speculare rispetto all’operazione get.

```
Successfully connected to client  ('127.0.0.1', 61564)
Client  ('127.0.0.1', 61564)  has chosen command 3-put canzone.mp3

File received
Download succesful
```

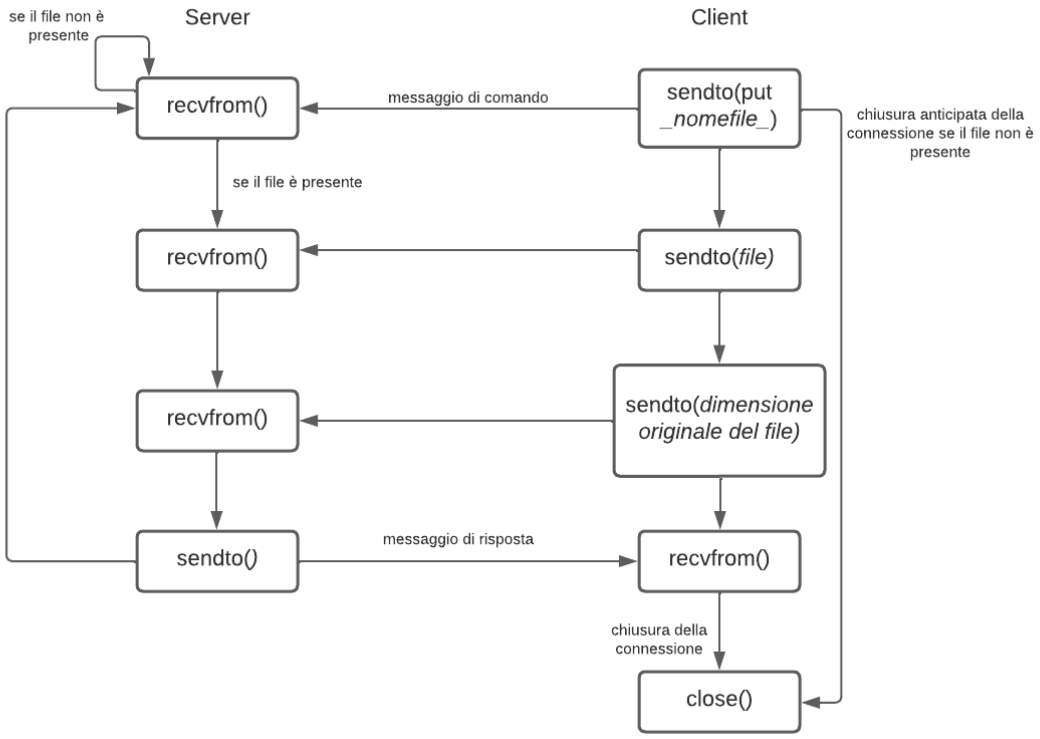
```
put canzone.mp3
Starting upload
File sent
File received without packet loss

Ending connection
```

In questo caso è il cliente a controllare che il file *_filename_* sia presente all’interno del suo archivio.

```
Successfully connected to client  ('127.0.0.1', 60955)
Client  ('127.0.0.1', 60955)  has chosen command 3-put fileerrato.mp4

The client ahve chosen an invalid file
```

```
put fileerrato.mp4
File not found

Ending connection
```

Se il file è presente si procederà all’upload del file, la strategia è la stessa del download ma a parti invertite.



Dopo aver ricevuto il file e la dimensione originale del file, il server manda il risultato dell'operazione al cliente.

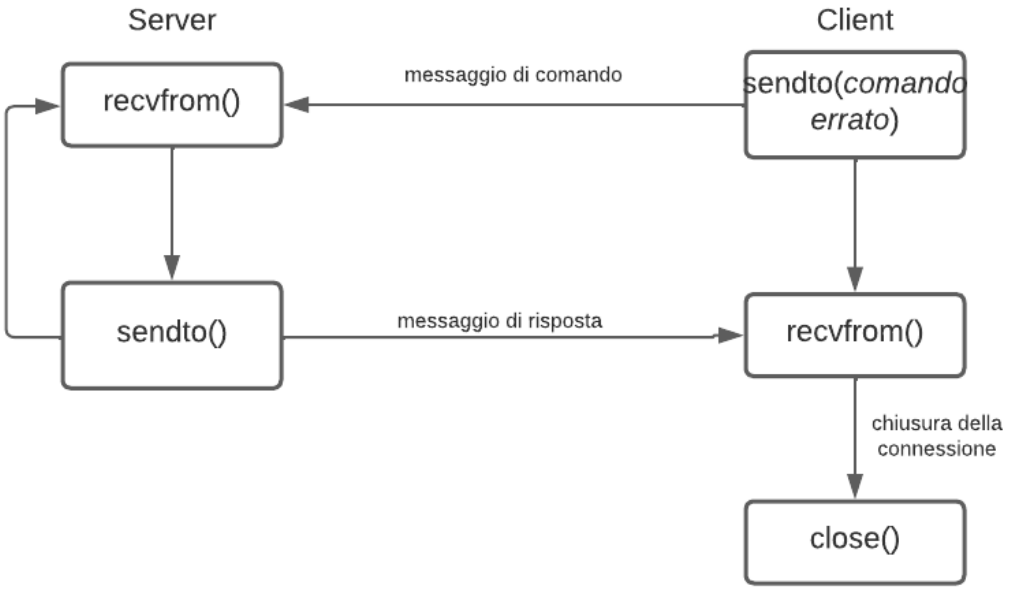
Sia in questo caso che nel comando get, l'upload o download di un file con un nome già presente nell'archivio di ricevimento procede con la sovrascrizione del vecchio file.

Comando errato

```
Waiting for a client

Successfully connected to client  ('127.0.0.1', 56897)
Client  ('127.0.0.1', 56897)  has chosen an invalid command


Waiting for a client
```

```
CONNECTION ESTABLISHED

Welcome, please select a command

1. list -> get all the files available in the server
2. get _filename_ -> download _filename_ from the server
3. put _filename_ -> upload a file to the server

4
You have chosen an invalid command
Ending connection
```

Se il cliente digita un comando non disponibile l'errore viene segnalato sia dal client che dal server.


Server
Client
recvfrom()
messaggio di comando
sendto(comando errato)
sendto()
messaggio di risposta
recvfrom()
chiusura della connessione
close()