Architettura Client-Server UDP per trasferimento file

Luca Bighini

Progetto Programmazione di Reti – 19 luglio 2022

Ingegneria e Scienze Informatiche, Cesena

Indice

OBIETTIVO

Lo scopo de progetto è quello di progettare e implementare in linguaggio Python un'applicazione Client-Server per il trasferimento di file che impieghi il servizio di rete senza connessione (UDP).

Il software dovrà permette:

- Connessione Client-Server senza autenticazione;
- La visualizzazione sul client dei file disponibili sul server
- Il download di un file dal server
- L'upload di un file sul server

Il Client dovrà permettere all'utente i comandi:

- List, per richiedere lista dei nomi dei file disponibili per la condivisione
- Get, per ottenere un file dal server
- Put, per caricare un file sul server
- Close, per chiudere il client socket
- Exit, per chiudere il client e server socket

Il Server dovrà permettere:

- L'invio del messaggio di risposta al comando list al client richiedente, con la lista dei file disponibili
- L'invio del messaggio di risposta al comando get contenente il file richiesto, se presente, od un opportuno messaggio di errore
- La ricezione di un messaggio put contenente il file da caricare sul server e l'invio di un messaggio di risposta con l'esito dell'operazione

PROGETTAZIONE

PROBLEMATICHE

Il software essendo basato su un'architettura Client-Server senza connessione (UDP), non controlla se il computer di destinazione è pronto per ricevere, ma invia direttamente i dati.
Il compito principale di un UDP è quello di prendere i dati dai protocolli di livello superiore e posizionarli nei messaggi UDP, che vengono poi spostati sull'IP per la trasmissione.
Questo permette ad una rete UDP di trasferire i dati ad una velocità maggiore, ma è meno affidabile nella trasmissione di dati di grandi dimensioni, come file audio e video, in quanto la probabilità di perdita di pacchetti è molto alta.
A differenza di TCP, UDP non garantisce la consegna dei dati, né ritrasmette i pacchetti persi, è solo un protocollo wrapper che facilita l'applicazione nell'accesso all'IP.

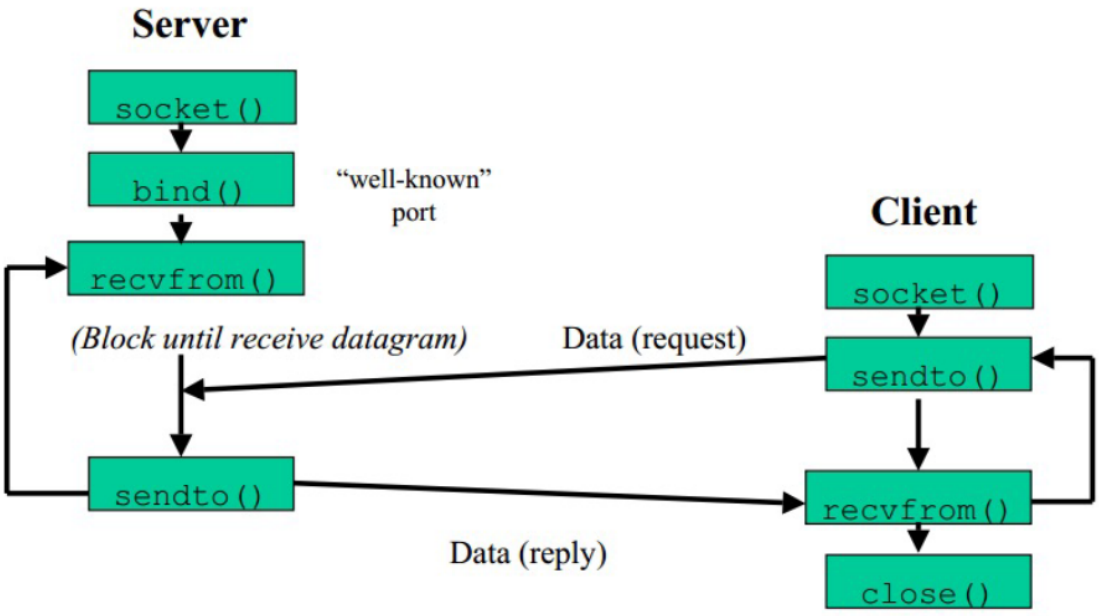

Figura 1: UDP Client-Server

Per risolvere il problema della perdita di pacchetti:

- impongo necessario l'invio di un messaggio di controllo (Ok) da parte del ricevente
- solamente una volta che il mandante del file ha ricevuto il messaggio di controllo procede all'invio del prossimo pacchetto, altrimenti ritorna ad inviare lo stesso

Questo comporta un calo della velocità di trasmissione in quanto è sempre necessario attendere il messaggio di controllo del ricevuto pacchetto.
Se tale messaggio informa di una mancata ricezione del pacchetto il mandante provvederà al rinvio dello stesso prima di procedere con i successivi.

PROTOCOLLO

Ad alto livello client e server comunicano tramite messaggi:

- `command`: stringa con comando da eseguire e se richiesto, il nome del file
- `lists`: contenente la lista di file
- `read`: contenente la parte di file appena letta
- `num_pack`: numero di pacchetti da ricevere
- `Ok`: stringa di corretta ricezione del pacchetto
- `Error` o `Resend`: stringa per la mancata ricezione del pacchetto

CODICE

CREAZIONE SOCKET

Sia il client, sia il Server come prima cosa creano il socket

```
#Create client socket UDP
try:
    sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
    print('Client socket initialized')
    sock.setblocking(0)
    sock.settimeout(10)
except sk.error:
    print('Client socket failed')
    sys.exit()
```

Figura 3: Client socket

```
#Create server socket UDP and bind it to the port
try:
    sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
    print('Server socket initialized: ')
    print ('starting up on %s port %s' % ADDR)
    sock.bind(ADDR)
except sk.error:
    print('Server socket failed')
    sys.exit()
```

Figura 2: Server socket

MENU DI SCELTA

Una volta che il Client riceve l'input entrambi chiamano le relative funzioni

```
if cm[0] == 'list':
    Server_List()

elif cm[0] == 'get':
    try:
       Server_Get(cm[1])
    except Exception as info:
        print(info)

elif cm[0] == 'put':
    try:
        Server_Put(cm[1])
    except Exception as info:
        print(info)

elif cm[0] == 'close':
    print('closing client socket')

elif cm[0] == 'exit':
    print('closing server socket')
    break

else:
    print('Error, command non found')
```

```
if cm[0].decode('utf8') == 'list':
    try:
        Client_List()
    except Exception as info:
        print(info)

elif cm[0].decode('utf8') == 'get':
    try:
        cm[1]
        Client_Get()
    except Exception as info:
        print(info)

elif cm[0].decode('utf8') == 'put':
    try:
        cm[1]
        Client_Put()
    except Exception as info:
        print(info)

elif cm[0].decode('utf8') == 'close':
    print('\nclosing client socket')
    break

elif cm[0].decode('utf8') == 'exit':
    print('\nclosing client and server socket')
    break

else:
    print('\nError, command non found')
```

Figura 4: Client-Server choose

COMANDI

- Comando `‘List’`

Il comando `list` prevede l’invio da parte del server della lista di tutti i file attualmente presenti al suo interno.

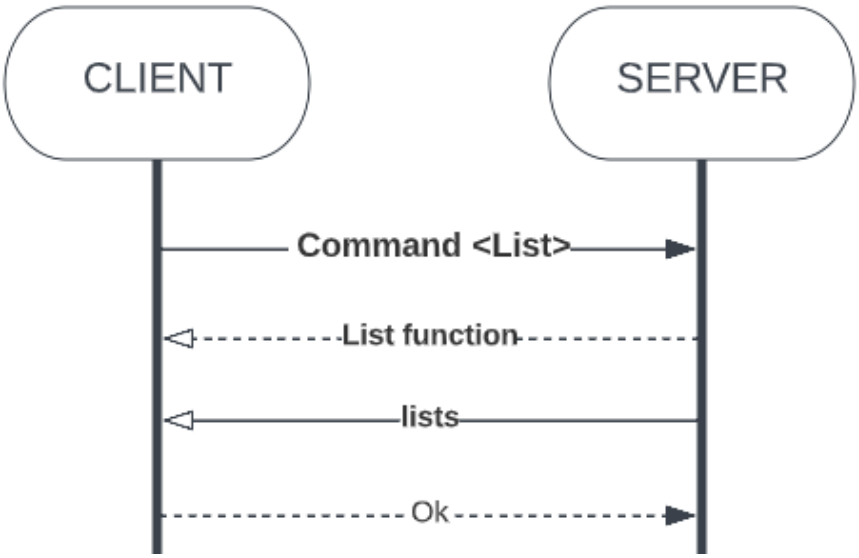

Figura 5: Comando List

- Comando `‘Get’`

Il comando `get` seguito dal `nome del file` prevede l’invio del file da parte del server al client richiedente.
Se non viene inserito il nome del file viene stampato il messaggio di errore: ‘list index out of range’
Se il nome di un file non è presente, il server invia al client la lunghezza del file a -1 e stampa: ‘Error, file not found’

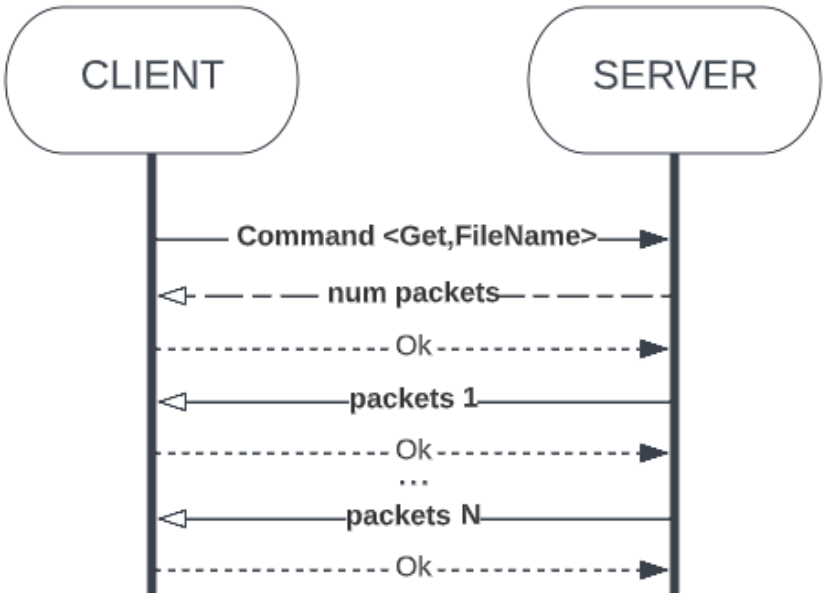

Figura 6: Comando get

- Comando **‘Put’**

Il comando **put** seguito dal **nome del file** prevede l’invio del file da parte del client al server.
Se non viene inserito il nome del file viene stampato il messaggio di errore: ‘list index out of range’
Se il nome di un file non è presente, il client invia al server la lunghezza del file a -1 e stampa: ‘Error, file not found’

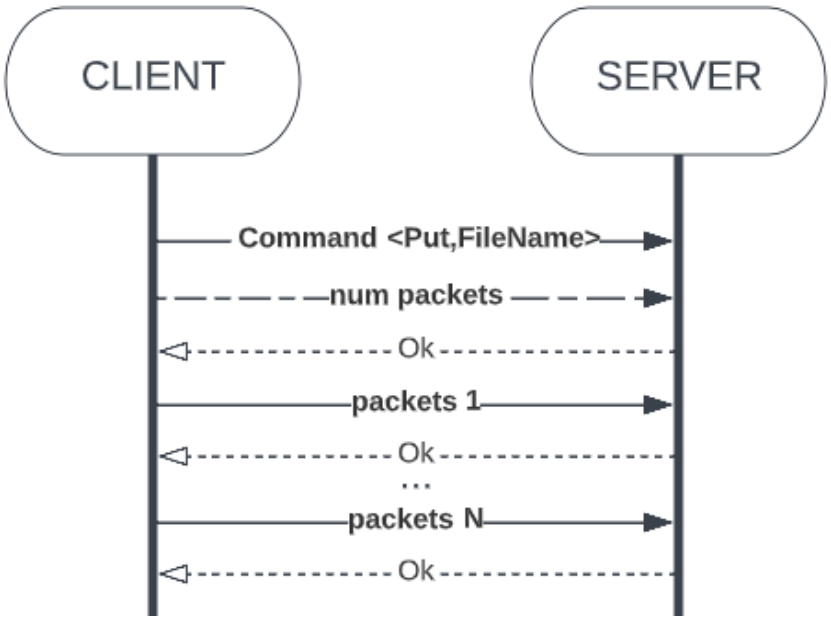

Figura 7: Comando put

- Comando **‘Close’**

Il comando **close** chiude il client socket

- Comando **‘Exit’**

Il comando **exit** chiude il server e il client socket

TIPI DI ERRORE

- **‘Server socket failed’** o **‘Client socket failed’**: se il socket non viene creato correttamente
- **‘Port not found’**: se il client tenta di inviare dati ad una porta inesistente
- **‘Error, command non found’**: se il comando inserito non è tra quelli consentiti
- **‘Error, not List’**: se il client non ha richiesto la lista
- **‘List NOT sent correctly’**: se la lista dei file non viene inviata correttamente
- **‘File NOT sent correctly’**: se il file non viene inviata correttamente
- **‘File NOT received correctly’**: se il file non viene ricevuto correttamente
- **‘Error, file not found’**: se il file cercato non è presente

GUIDA UTENTE

ENVIROMENT

Per eseguire i file all'interno del progetto è necessario usare una versione di Python 3.8 o superiore.
Non è necessario installare alcun pacchetto aggiuntivo in quanto vengono usati solo moduli presenti nella libreria standard di Python.

SERVER

Per eseguire il Server è sufficiente lanciare all'interno della cartella del file il comando:
`python3 UDP_Server.py`

CLIENT

Per eseguire il Client è sufficiente lanciare all'interno della cartella del file il comando:
`python3 UDP_Client.py`