

Relazione del progetto  
“Architettura client-server UDP per  
trasferimento file”

Angela Speranza

Settembre 2022

# Indice

<b>1</b>	<b>Architettura client-server UDP</b>	<b>2</b>
1.1	Requisiti del progetto . . . . .	2
<b>2</b>	<b>Struttura del progetto</b>	<b>3</b>
2.1	Funzionamento dettagliato . . . . .	3
2.1.1	Client . . . . .	4
2.1.2	Server . . . . .	7
2.1.3	Common Functionalities . . . . .	9
<b>A</b>	<b>Guida utente</b>	<b>11</b>

# Capitolo 1

## Architettura client-server UDP

### 1.1 Requisiti del progetto

Questo progetto si pone come scopo quello di progettare ed implementare una applicazione client-server - offline e senza autenticazione - che effettui il trasferimento di file di diversa natura.

La comunicazione tra la parte client e quella del server è stata gestita attraverso l'utilizzo del protocollo UDP, quindi un protocollo non orientato alla connessione e non confermato. Non vi è, dunque, uno scambio di dati per l'istaurazione e la chiusura delle sessione attive. Tuttavia, client e server comunicano tra loro in modo efficace tramite lo scambio di messaggi. Il client invia al server messaggi di "comando": in questo modo effettua delle richieste al server. Il server, in riferimento a questi messaggi, ne invia alcuni di "risposta" indicando al client l'esito della richiesta precedentemente effettuata.

Di seguito vengono riportati i requisiti necessari alla buona riuscita del compito assegnato. L'applicativo deve garantire le seguenti caratteristiche:

- Al client deve essere disponibile una lista - sempre aggiornata - dei file disponibili sul server e pronti ad essere scaricati.
- Deve essere possibile effettuare il download di questi file dal server.
- Deve essere possibile effettuare l'upload di un file sul server ad opera del client.

# Capitolo 2

## Struttura del progetto

### 2.1 Funzionamento dettagliato

Nella realizzazione di questo progetto, è stato propedeutico uno sguardo più approfondito sul funzionamento e sui principi cardine del protocollo UDP. Prima di costruire i singoli sorgenti del client e del server, infatti, ho definito le modalità di comunicazione che dovevano essere rispettate in questo tipo di relazione. Innanzitutto era necessario curare l'aspetto del cosiddetto "livello di trasporto", che, nel caso corrente, consiste in un numero di porta identificativo (non tra quelli delle well-known port), che determina univocamente il particolare processo applicativo in esecuzione sull' host.

Il passo successivo consiste nell'apertura dei socket. Il processo Server apre un socket di comunicazione dove si mette in ascolto sulla relativa porta, in attesa dell'arrivo di una richiesta. Lavoro simile effettua il Client che, sempre sulla medesima porta, invia, grazie al socket aperto, richieste all'altra parte.

Grazie a questa dinamica, UDP consente una comunicazione rapida e pressochè immediata: il protocollo è adatto a una trasmissione di dati che deve essere "costante".

## Socket programming with UDP

**UDP:** no “connection” between client and server:

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

**UDP:** transmitted data may be lost or received out-of-order

**Application viewpoint:**

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server processes

Application Layer: 2-101

Figura 2.1: Lo schema esplicativo dei principi del protocollo UDP.

A questo punto ho iniziato a creare i sorgenti del client e del server.

### 2.1.1 Client

Le funzionalità del client sono racchiuse in un unico file, denominato *client.py*. Dopo aver configurato il socket, in questo sorgente python ho implementato esclusivamente la dinamica di invio di richieste da parte del client. In sostanza, ho realizzato i comandi che l'utente può digitare.

```
Connecting to server...
Welcome. Please choose a command from the followings:

- List -> get the list of the file you can download.
- Get <filename> -> download the file from the archive
- Put <filename> -> upload the file to the archive.
- Exit -> close the program.
- Help -> show the available commands.

Write a command:
```

Figura 2.2: La lista dei comandi che l'utente può digitare.

### Exit

Il comando *exit* è stato molto semplice da implementare. Il server viene notificato della scelta dell'utente e, dopo ciò, il socket viene semplicemente chiuso. In questo modo, entrambi le parti vengono disconnesse l'una dall'altra, senza lasciare processi attivi.

### List

Il comando *list*, una volta digitato dall'utente, permette a quest'ultimo di visualizzare una lista di file. Questo elenco non è altro che una cartella sotto il controllo del Server, riempita con una serie di file disponibili per il download (questa azione in particolare è implementata nel comando *get*). L'utente può scegliere di scaricare uno tra i documenti nell'elenco, i quali possono essere anche di formati e dimensioni diverse. Fatto ciò, il terminale si rimette in ascolto, in attesa di un nuovo comando.

```
Write a command:
List

Showing files...
Files stored are: ['cat.png', 'pascoli.txt', 'reti.zip']

Write a command:
█
```

Figura 2.3: Funzionamento del comando *List*.

## Help

Anche il comando *help* è alquanto semplice. Il client, infatti, invia al server la richiesta di mostrare nuovamente il messaggio mostrato in Figura 2.2. In tal modo l'utente può riconsultare la lista di comandi disponibili.

## Get

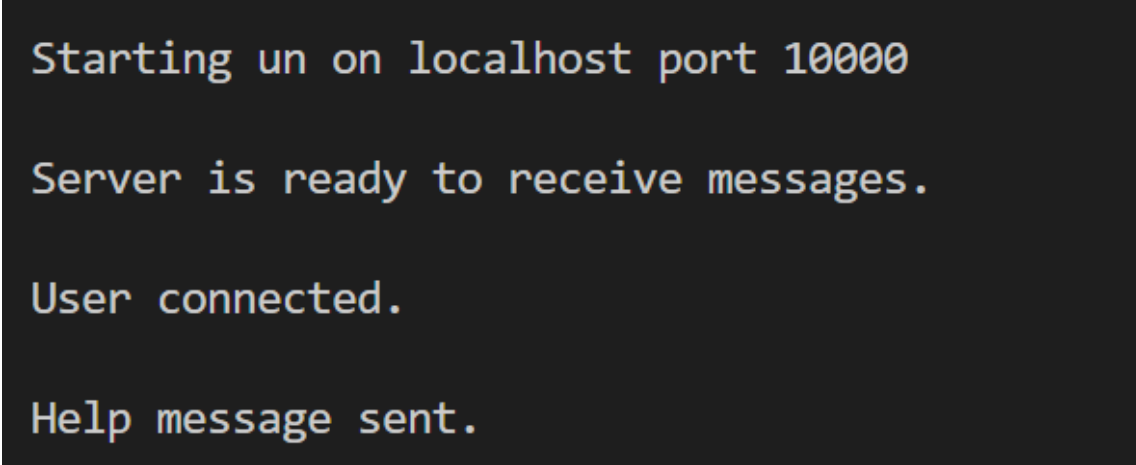
Il comando *get*, come anticipato poco sopra, consente all'utente di scaricare file tra quelli proposti dal comando *list*. Il suo funzionamento, come quello del comando *put* è più complesso da realizzare rispetto agli altri comandi. Come prima cosa, dopo aver notificato il server, controllo che il file richiesto sia effettivamente esistente. Se questo controllo va a buon fine, si può procedere con il download. Innanzi tutto si "calcola" la dimensione del messaggio, così da procedere con la "raccolta" dei pacchetti giunti a destinazione. La lista, in un primo momento vuota, viene riempita man mano nel corso del download, fino a raggiungere la dimensione desiderata. Per la struttura del protocollo, però, non è garantito che i pacchetti arrivino in ordine; per questo motivo, i questi vanno riordinati e comparati con l'hash originale della lista dei pacchetti arrivati dal server. Se questo confronto va a buon fine, il file non è corrotto e il file viene scaricato scrivendo in locale il contenuto di ogni pacchetto.

## Put

Il comando *put* permette all'utente di caricare un file inviandolo al server. Dopo aver controllato la validità del file, il client notifica il server e inizia il caricamento. Dapprima, ricevuto l'ACK, viene stimato e inviato il numero totale di pacchetti che il server deve ricevere. Questi pacchetti, conservati in una lista, vengono mandati uno alla volta, esattamente come per il download. Insieme a questi viene anche inviato l'hash del file.

### 2.1.2 Server

Il sorgente dedicato al lato server, *server.py*, è nato sulla falsariga del client. Così come nel caso precedente, anche in questa situazione ho qui gestito soltanto la ricezione dei messaggi inviati dal client e le relative azioni di risposta. Una volta configurata l'apertura del socket, il server si mette in ascolto sulla porta prestabilita. Soltanto una volta che un client si connette può iniziare lo scambio dei dati tra le due parti. La connessione da parte del client al server fa subito partire il messaggio *help* di cui ho parlato sopra. Di norma, subito dopo questo passaggio, l'utente digiterà uno dei comandi a disposizione.



```
Starting un on localhost port 10000  
  
Server is ready to receive messages.  
  
User connected.  
  
Help message sent.
```

Figura 2.4: Attivazione del server / Connessione di un client.

#### Exit

Il discorso per ciò che riguarda questo comando è del tutto analogo a quello presentato nel paragrafo precedente. Così, come accade per il client, in risposta a questo comando anche il socket del server viene chiuso e interrotta la comunicazione tra i due.

#### List

Il comando *list* genera una risposta molto semplice lato server. In tal caso è necessario soltanto consultare (attraverso il percorso tra le cartelle) la lista dei file conservati dal server e pronti al download, dopodiché trasmettere questa lista al client.



## Help

L'implementazione di questa risposta è alquanto banale. In questi casi il server manda in output sul client lo stesso messaggio iniziale, contenente la lista dei comandi.

## Get

Ricevuto il comando *get*, il server inizia ad attivarsi per inoltrare al client il file richiesto. Dapprima si effettua un check sul nome del file richiesto e sulla sua dimensione. Superato questo controllo, il server procede ad inviare l'ACK, dunque a trasferire l'intera lista dei pacchetti che compongono il file. Infine, viene anche inviato l'hash per il controllo finale sulla validità del file di cui ho parlato in precedenza.

## Put

Questa azione è simile (ma naturalmente "opposta") a quella che segue il comando precedente. Anche in questo caso, al server viene prima notificato il nome del file in arrivo e l'ACK. In questo caso, viene predisposto un array per conservare la lista dei pacchetti che stanno per essere caricati sul server. Questa lista viene riempita e poi riordinata seguendo il corretto ordine delle posizioni dei pacchetti. Controllato che non sia stato corrotto, il file ricevuto viene scritto definitivamente nella cartella locale dedicata al server.

## Client/server socket interaction: UDP

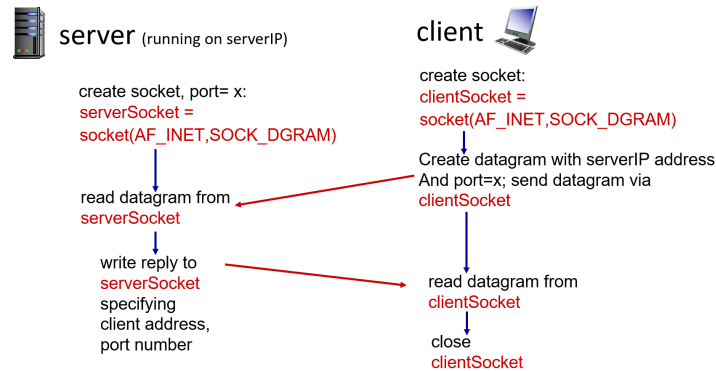


Figura 2.5: Schema di riferimento dell'interazione basilare tra client e server.

### 2.1.3 Common Functionalities

Ho creato, infine, un sorgente contenente alcune funzionalità utili sia al client che al server. Questo file nasce con l'intento di evitare ripetizioni e tenere quanto più possibile il codice "pulito".

```
7
8  # Some constants for the program
9  BUFFER=32768
10 PACKET=8192
11 SLEEP_TIME=0.001
12 SERVER_ADDR = ('localhost', 10000)
13
14 # Paths to the files
15 download_path = os.getcwd()+"/client/download/"
16 upload_path = os.getcwd()+"/client/upload/"
17 files_path = os.getcwd()+"/server/archive/"
```

Figura 2.6: Alcune costanti e i *path* dei file da gestire.

Nella parte alta del codice troviamo alcune costanti utili e i percorsi ai file mostrati in figura 2.6. Di seguito, sono state implementate alcune funzioni utili di cui andrò brevemente a spiegare il funzionamento.

- *hash list*: questa funzione genera l'hash list dei pacchetti per controllare l'integrità del file.
- *send data*: questa funzione permette lo scambio di dati attraverso i socket.
- *get files list*: questa funzione prende dall'archivio locale i file presenti e li racchiude in una lista.
- *get file length*; come suggerisce il nome, questa funzione restituisce la lunghezza del file indicato.
- *send help message*: consente di inviare il messaggio di aiuto quando richiesto.
- *files list*: questa funzione pulisce la lista dei file da mandare in output al client, mostrando soltanto i file "validi" al download.

# Appendice A

## Guida utente

L'applicativo è molto semplice da utilizzare. Il programma è pensato per essere utilizzato da terminale. Aperte due finestre distinte di una qualsiasi shell di comandi, posizionarsi nella cartella in cui risiede il progetto, con i sorgenti python e le cartelle dedicate ai file di client e server. Avviare questi ultimi due attraverso il comando **python3 server.py** nella prima, dunque nel secondo terminale digitare **python3 client.py**, in quest'ordine. Da qui in poi utilizzare il client per usufruire dei comandi disponibili: il server, in risposta, aggiornerà l'utente sullo stato delle sue azioni e lo notificherà di errori e successi.