

Progetto Programmazione di Reti: ”Architettura Client-Server UDP”

Ferri Samuele, Strada Nicola

Ferri Samuele
samuele.ferri3@studio.unibo.it
Matricola: 0000974660

Strada Nicola
nicola.strada@studio.unibo.it
Matricola: 0000971628

Indice

1	Analisi Progetto	3
1.1	Funzionalità Server	3
1.2	Funzionalità Client	3
2	Decisioni Progettuali	4
2.1	Struttura del Protocollo Utilizzato	4
2.2	Struttura Datagram	4
2.3	Gestione Tipologia Operazioni	6
2.4	Struttura Client	7
2.5	Struttura Server	7
2.6	Main Client e Main Server	8
3	Strutture Dati Impiegate	9
3.1	MakeDatagram	9
3.2	Operation	10
4	Threads in Client e Server	11
4.1	Threads in Client	11
4.2	Threads in Server	12
5	Guida Utente	13
5.1	Esecuzione Server	13
5.2	Esecuzione Client	13

1 Analisi Progetto

Si pone come obiettivo quello di progettare e realizzare un'applicazione, in linguaggio Python, Client-Server, per il trasferimento di ogni tipo di file. L'applicazione utilizzerà come protocollo di trasporto UDP, dovrà permettere una connessione Client-Server senza alcuna autenticazione, la visualizzazione dei file contenuti nel Server, e l'eventuale *Upload e Download* di nuovi.

1.1 Funzionalità Server

Le funzionalità di cui il Server sarà predisposto sono:

- L'invio di un messaggio di risposta al comando *'list'* da parte del Client. Che permetterà la visualizzazione completa al Client di ciascun nome dei file presenti sul Server.
- l'invio di un messaggio di risposta al comando *'get'* da parte del Client. Che permetterà il Download di un file, se presente sul Server, sul Client e l'eventuale generazione e gestione di errori durante l'operazione.
- l'invio di un messaggio di risposta al comando *'put'* da parte del Client. Che permetterà l'Upload di un file, se presente sul Client, nel Server e l'eventuale generazione e gestione di errori durante l'operazione.

1.2 Funzionalità Client

Le funzionalità di cui il Client sarà predisposto sono:

- L'invio di un comando *'list'* al Server, per richiedere la lista completa dei nomi dei file presenti sul Server.
- L'invio di un comando *'get'* al Server, per iniziare un'operazione di Download di uno dei file presenti sul Server, e nel caso di errori, la loro gestione.
- L'invio di un comando *'put'* al Server, per iniziare un'operazione di Upload, di uno dei file presenti sul Client, sul Server e nel caso di errori, la loro gestione.

2 Decisioni Progettuali

2.1 Struttura del Protocollo Utilizzato

Per la comunicazione tra Client e Server abbiamo definito un protocollo personalizzato, il quale utilizza un proprio **Datagram** (2.2), questo viene costruito tramite la classe *MakeDatagram.py*, e può avvenire in tre diverse modalità. Il primo, **datagram_operation₁**, verrà inviato dal Client e viene utilizzato per specificare quale sia l'operazione richiesta, precisata all'interno del pacchetto nell'apposito campo. Conseguentemente il Server, se aperto e in ascolto, riceverà l'operazione e controllerà se rientra tra quelle disponibili, in caso affermativo inizierà l'esecuzione dell'operazione. Una volta terminata verrà notificata la fine tramite un Datagram specifico, costruito con **datagram₂**, contenente la stringa *'FILE_END'* nel campo *'file_name'*. Diversamente, se nel corso dell'esecuzione si verificassero errori di qualunque genere, il Datagram impiegato, ovvero **datagram_error₃**, avrà nel campo *'file_name'* la stringa *'ERROR'*.

Ciascun Datagram inviato avrà una *size* definita anticipatamente, questa sarà anche discriminante per quanto riguarda lo stabilire della grandezza del buffer in ricezione, impostato con una grandezza doppia della size.

Durante le operazioni più onerose di **'get'** e **'put'**, la parte dei metadata, già in formato di byte, verrà codificata in Base64 permettendo così l'invio di qualsiasi tipo di file: che siano immagini, video, file audio o testo.

Infine, prima dell'invio del Datagram, anche tutto il pacchetto sarà codificato in byte.

2.2 Struttura Datagram

Ciascun Datagram impiegato utilizza uno scheletro predisposto dalla classe **MakeDatagram.py**, questa fornisce tre strutture diverse, ognuna delle quali si fonda su dizionari Python, in questo modo possiamo racchiudere in un'unica struttura più informazioni, avvicinandoci così alla struttura di un vero e proprio Datagram. Analizziamo di seguito le tre diverse strutture proposte:

- **datagram_operation:**

```
def datagram_operation(self, file_name, operation):
    datagram_operation = {
        'file_name' : file_name,
        'operation' : operation
    }
    return json.dumps(datagram_operation)
```

La prima che osserviamo viene utilizzata per comunicare quale operazione il Client vuole eseguire, specificata nel campo *'operation'* del dizionario. Nel caso sia un'operazione di *'get'* o *'put'*, ricorriamo al campo *'file_name'* per specificare il nome del file implicato nel procedimento, mentre nell'operazione di *'list'* il campo rimarrà vuoto.

- **datagram_error**

```
def datagram_error(self, file_name, command, error):
    datagram_error = {
        'file_name': file_name,
        'command' : command,
        'error' : base64.b64encode(error).decode('
                                                    ascii')
    }
    return json.dumps(datagram_error)
```

La seconda struttura è quella impiegata nello spiacevole caso di generazione di eventuali errori. In questo scenario, una volta catturato correttamente l'errore, l'operazione verrà annullata e sarà comunicato alla parte mittente la sua interruzione e la tipologia di errore verificatosi. Troviamo tre differenti campi: *'file_name'*, contenente la stringa *'ERROR'*, che servirà per identificare il Datagram come **datagram_error**, *'command'* con l'operazione annullata dal verificarsi dell'errore, e infine *'error'* dove troviamo la spiegazione dettagliata dell'errore.

- **datagram**

```
def datagram(self, file_name, size, metadata):
    datagram = {
        'file_name' : file_name,
        'size' : size,
        'metadata' : base64.b64encode(metadata).
                                                                decode('
                                                                    ascii')
    }
    return json.dumps(datagram)
```

L'ultimo scheletro fornito dalla classe è **datagram**, questa utilizzata per l'invio più generale di pacchetti dove troviamo una parte di informazioni, ovvero l'effettivo messaggio da recapitare. Il primo parametro che troviamo è *'file_name'*, contenente il nome del file impiegato nell'operazione, il secondo è *'size'* calcolata in byte su tutto il pacchetto inviato, e infine *'metadata'*, ovvero la parte di data trasmessi.

2.3 Gestione Tipologia Operazioni

Per quanto riguarda la distinzione delle diverse operazioni effettuabili, si è optato per la creazione di una classe, **Operation.py**, la quale semplicemente a seconda del metodo richiamato, restituisce in stringa il nome del comando corrispondente ad una operazione. I quattro comandi disponibili e utilizzabili sono:

- **__op_home = 'home'** : Corrisponde all'operazione di *'home'*, richiamata dal Client automaticamente sia ad inizio connessione, sia ogni qualvolta viene terminato uno degli altri comandi. L'esito corretto di questa operazione comporta la visualizzazione, da parte del Client, del menù, con annessa spiegazione delle funzionalità fornite dal Server.
- **__op_list = 'list'** : Prima operazione richiamabile interattivamente dal Client, questa permette all'utente di visualizzare una lista completa di tutti i nomi dei file presenti sul Server.
- **__op_get = 'get'** : Secondo comando richiamabile dal Client; questa operazione si compone però di due passaggi: il primo inevitabilmente è volto ad attivare il comando di *'get'*, mentre per il secondo viene richiesto l'inserimento del nome del file da voler scaricare dal Server. Una volta inserito il nome, necessariamente diverso da vuoto e corrispondente ad uno dei nomi già presenti sul Server, inizierà il Download. Una volta terminato correttamente si potrà trovare il file scaricato nella cartella predefinita del Client.
- **__op_put = 'put'** : Comando di *'put'*, costruito sulla falsa riga di *'get'*, composto dai medesimi due passaggi, con la differenza che il file deve essere questa volta presente nella cartella del Client. Questa operazione risulta però speculare alla precedente, dato che in questo caso il mittente è il Client e il Server è il destinatario. Una volta eseguito verrà quindi effettuato l'Upload del file sul Server, se questo è già presente verrà sovrascritto. Al termine dell'operazione si potrà trovare sul Server il file designato.
- **__op_close = 'close'** : Ultimo comando disponibile, una volta eseguito il Client chiude la connessione instaurata col Server, il programma lato utente così termina. Lo stato del Server non viene però influenzato da questa operazione.

2.4 Struttura Client

Per quanto riguarda la struttura del Client si è optato per una configurazione interna al codice, per questo motivo non servirà, al momento dell'inizializzazione, nessun parametro di impostazione. Di seguito osserviamo i parametri con i loro rispettivi valori.

```
def __init__(self):
    self.port_server = 10000
    self.address_server = 'localhost'
    self.buffer = 9216 #Buffer of 9216 byte
    self.socket = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
    self.sleep_time = 0.05
    self.socket_timeout = 10
```

I valori sono stati scelti anticipatamente, e naturalmente corrisponderanno a quelli utilizzati nel Server. La porta utilizzata dal Server, e quindi rispettivamente anche dal Client per comunicare, è la *10000*, mentre l'indirizzo IP servitosi è quello di *'localhost'*. Per quanto riguarda la *'buffer size'* si è definita una grandezza tale da permettere un invio piuttosto fluido di pacchetti, ottenendo un margine abbastanza grande anche per file piuttosto pesanti. Dobbiamo precisare, che sottoponendo all'applicazione file di grandezze nell'ordine di Gigabyte, i tempi di invio e ricezione si dilatano esponenzialmente, incrementando la possibilità del generare errori. Troviamo infine, oltre all'oggetto socket utilizzato per la comunicazione, il suo tempo di *timeout*; dopo il quale il socket terminerà la comunicazione, e il tempo di *sleep*, utilizzato tra l'invio di un pacchetto e l'altro.

2.5 Struttura Server

Per quanto riguarda il Server, la struttura è la medesima del Client, in quanto presenta i corrispettivi valori descritti nella sottosezione precedente (2.4).

```
def __init__(self):
    self.port = 10000
    self.address = 'localhost'
    self.socket = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
    self.buffer = 9216 #Buffer of 9216 byte
    self.sleep_time = 0.05
```

Come osserviamo, i parametri e i loro valori risultano gli stessi del Client, questo per permettere la comunicazione. Notiamo però che il Server, una volta eseguito, rimane in ascolto attendendo un messaggio da parte di un Client. Per poi terminarlo sarà necessario agire direttamente sul Server, chiudendo l'applicazione.

2.6 Main Client e Main Server

Per una struttura del codice più fluida ed ordinata, si è preferito separare le due classi principali: **Server** e **Client**, dai loro rispettivi **Main**. Così facendo per l'esecuzione dell'applicazione sarà necessario eseguire, in due console separate, **MainClient.py** e **MainServer.py**. Queste conterranno esclusivamente l'inizializzazione dei loro rispettivi oggetti, con l'aggiunta in **MainClient.py** della gestione degli input.

3 Strutture Dati Impiegate

3.1 MakeDatagram

La struttura di **MakeDatagram.py** ha come scopo quello di fornire, a seconda della configurazione selezionata, un facsimile di un Datagram. Troviamo tre diverse tipologie, ciascuna utilizza come base un oggetto dizionario di Python, riuscendo ad incapsulare più informazioni in un unico pacchetto. Infine prima di essere restituito e poi inviato, il *Datagram* viene convertito **JSON string**, utilizzando la funzione *json.dumps()*. Di seguito riportata la struttura della classe, con i tre diversi metodi: **datagram_operation**, **datagram_error** e **datagram**.

```
class MakeDatagram:

    def datagram_operation(self, file_name, operation):
        datagram_operation = {
            'file_name' : file_name,
            'operation' : operation
        }
        return json.dumps(datagram_operation)

    def datagram_error(self, file_name, command, error):
        datagram_error = {
            'file_name': file_name,
            'command' : command,
            'error' : base64.b64encode(error).decode('ascii')
        }
        return json.dumps(datagram_error)

    def datagram(self, file_name, size, metadata):
        datagram = {
            'file_name' : file_name,
            'size' : size,
            'metadata' : base64.b64encode(metadata).decode('ascii')
        }
        return json.dumps(datagram)
```

Precisiamo infine che, il contenuto di *'metadata'* in **datagram**, e *'error'* in **datagram_error**, viene codificato inoltre in Base64, permettendo l'invio di qualunque tipo di file: Immagini, video, file audio e testo.

3.2 Operation

Impieghiamo la classe **Operation**, come spiegato in precedenza, per definire uno "standard" sulle quattro operazioni eseguibili dall'applicazione. In questo modo ogni qualvolta verrà fatto riferimento, qualunque sia il motivo, ad uno dei comandi, verranno sfruttati i metodi di questa classe. Ciascuno di questi ritorna, in formato stringa, il nome della rispettiva operazione.

Di seguito troviamo come **Operation.py** è stata progettata.

```
class Operation:
    __op_home = 'home'
    __op_list = 'list'
    __op_get = 'get'
    __op_put = 'put'
    __op_close = 'close'

    def operation_home(self):
        return self.__op_home

    def operation_list(self):
        return self.__op_list

    def operation_get(self):
        return self.__op_get

    def operation_put(self):
        return self.__op_put

    def operation_close(self):
        return self.__op_close
```

4 Threads in Client e Server

4.1 Threads in Client

Troviamo di seguito, elencati, i *Threads* attivati dal Client:

- **get_home_server** → Operazione richiamata automaticamente dal Client: sia all'avvio, dopo lo stabilirsi della prima connessione, sia ogni volta terminato, in qualunque modo, un comando.
Questo processo semplicemente costruirà un **datagram_operation**, contenente l'operazione di *'home'*, ed una volta inoltrato al Server, rimarrà in attesa di un messaggio di risposta con l'informazione richiesta. Se il pacchetto ricevuto non è un messaggio di errore, il menù verrà decodificato e reso visualizzabile all'utente.
- **command_list** → Primo comando disponibile e selezionabile dall'utente. Inserendo la stringa *'list'* nella console del Client, verrà inviato al Server un **datagram_operation**, con operazione *'list'*, dopodiché sarà compito del Server ricavarci la lista dei file presenti nella sua cartella di default, e preparare il Datagram che verrà inviato come risposta. Il Client rimarrà perciò in ascolto attendendo il messaggio di risposta, il quale una volta decodificato sarà visualizzato dall'utente.
- **command_get** → Secondo comando selezionabile, questo si compone di due passaggi, il primo per selezionare il comando, mentre il secondo per specificare di quale file effettuare il Download dal Server. Una volta eseguito il comando e comunicato al Server l'eseguirsi di quale operazione, il Client rimarrà in attesa di uno o più pacchetti, ricostruendo il file in questione decodificando la parte di informazioni. Ricevuta poi la stringa *'FILE_END'*, il file, ricevuto correttamente, verrà chiuso e si troverà nella cartella di default del Client.
- **command_put** → Il terzo comando, *'put'*, si sviluppa similmente al *'get'*, ma a parti inverse. Il Client, una volta appreso il tipo di operazione, controllerà se il file è presente, e in caso affermativo comunicherà al Server con un **datagram_operation**, l'inizio dell'esecuzione del comando di Upload. Successivamente comincerà ad inviare al Server il file, suddividendolo in vari pacchetti a seconda della grandezza del buffer. Finita l'operazione lo segnalerà tramite un Datagram contenente la stringa *FILE_END*.
- **command_close** → Il quarto e ultimo comando disponibile per l'utente è il comando di *'close'*, se eseguito la connessione con il Server verrà chiusa, e di conseguenza terminerà la comunicazione insieme all'applicativo del Client.

4.2 Threads in Server

Analizziamo di seguito i Threads eseguiti dal Server, per lo più speculari al Client.

- **server_opening** → Richiamato subito dopo l'avvio dell'applicativo, in questo modo il Server viene predisposto per rimanere in continuo ascolto (*Busy Waiting*), aspettando, da parte di un Client, un Datagram di tipo **datagram_operation**. Dopodiché procederà con l'eseguire il Threads corrispondente all'operazione comunicatagli. Il Server tornerà in questa condizione, ogni qualvolta terminato, con qualsiasi esito, un comando.
- **send_home** → Impiegato come risposta all'operazione di *'home'*. Il Server inoltrerà, al Client che lo richiede, il menù contenente tutte le funzionalità eseguibili.
- **command_list** → Primo comando, attivato all'operazione di *'list'*, dove il Server provvederà a costruire una stringa contenente il nome di tutti i file presenti nella sua cartella di default. Infine verrà poi codificata e inviata al Client destinatario.
- **command_get** → Secondo comando attivatosi all'operazione di *'get'*, strutturato specularmente al *'put'* del Client, in quanto il Server procederà all'invio del file richiesto, dividendolo in più pacchetti a seconda del buffer. Il termine dell'esecuzione verrà poi notificata al Client con un Datagram contenente la stringa *FILE_NAME*. Nel caso si verifichino errori il comando verrà interrotto e si procederà con l'invio di un **datagram_error** al Client destinatario.
- **command_put** → Terzo comando di risposta fornito dal Server, costruito ancora una volta sulla falsa riga del *'get'* del Client. Il Server si predisporrà in ascolto, ricostruendo, pacchetto per pacchetto, il file ricevuto dall'Upload del Client. Nel caso di ricezione di un **datagram_error**, per il verificarsi di un qualche errore durante l'invio nel Client, il file non verrà salvato e l'operazione terminerà con esito negativo.

5 Guida Utente

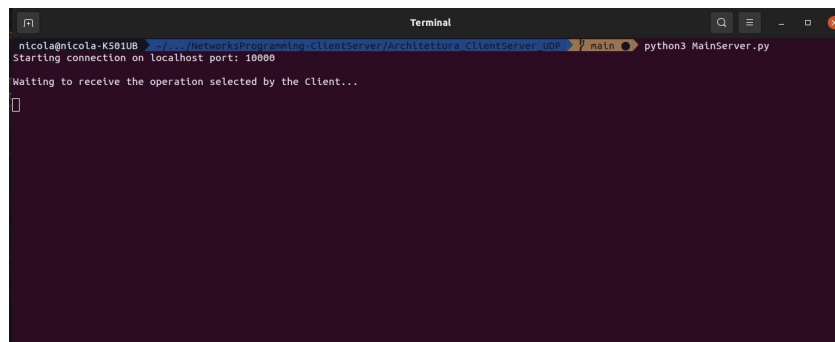
In quest'ultimo capitolo troviamo una breve *Guida Utente*, con tutte le indicazioni per l'utilizzo del software.

5.1 Esecuzione Server

Per un corretto utilizzo del software si proceda prima con l'applicativo del Server: **MainServer.py**, eseguendolo da terminale o qualsiasi IDE Python, con versione Py3 o superiore. Da console sarà sufficiente il comando:

```
python3 MainServer.py
```

Verrà così aperto il Server, già pronto all'uso, in quanto non sarà necessaria la configurazione di nessun parametro. Osserviamo come nella console in questione vengono visualizzate le informazioni di connessione: *Indirizzo e Porta*.



```
Terminal
nicola@nicola-K581UB: ~/NetworksProgramming-ClientServer/Architettura-ClientServer-UDP$ python3 MainServer.py
Starting connection on localhost port: 10000
Waiting to receive the operation selected by the client...
█
```

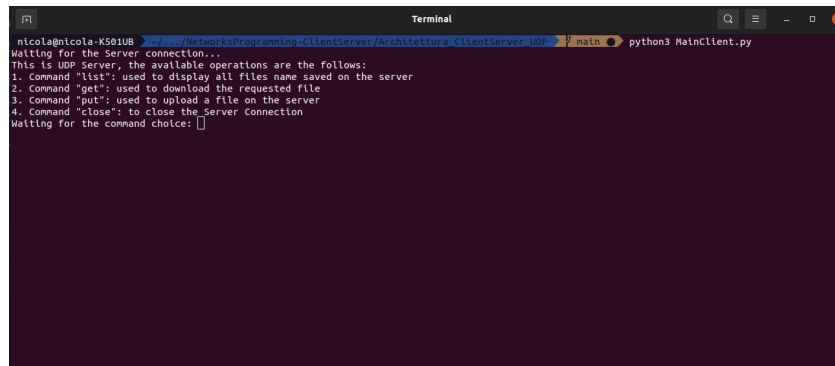
Figure 1: Esecuzione su console di **MainServer.py**

5.2 Esecuzione Client

Una volta eseguito il Server si proceda con l'applicativo del Client: **MainClient.py**. Questo dovrà essere eseguito su una console diversa, con il medesimo procedimento.

```
python3 MainClient.py
```

Verrà così eseguito il Client, stabilendo autonomamente una connessione con il Server lanciato precedentemente, per poi visualizzare sulla console la *home* inviategli.



```
nicola@nicola-K501UB: ~/NetworksProgramming/ClientServer/Architettura_ClientServer_UDP $ python3 MainClient.py
Waiting for the Server connection...
This is UDP Server, the available operations are the follows:
1. Command "list": used to display all files name saved on the server
2. Command "get": used to download the requested file
3. Command "put": used to upload a file on the server
4. Command "close": to close the Server Connection
Waiting for the command choice: 
```

Figure 2: Esecuzione su console di **MainClient.py**

Dopodiché si potrà procedere all'utilizzo, eseguendo una delle quattro operazioni proposte.

- **'list'** → Digitando su console la stringa *'list'*
- **'get'** → Digitando prima *'get'* per il comando, e poi inserendo il nome completo del file, compreso di estensione, per il Download dal Server.
- **'put'** → Digitando prima *'put'* per il comando, e poi inserendo il nome completo del file da caricare sul Server.
- **'close'** → Digitando la stringa *close*.