# Relazione Progetto di Programmazione di Reti: Traccia 2

Ezmiron Deniku
ezmiron.deniku@studio.unibo.it
matricola: 0000989206

# Indice

# Capitolo 1

# Analisi del problema

## 1.1 Descrizione

L'obiettivo del progetto è quello di implementare, tramite linguaggio Python, un'applicazione per il trasferimento di file che impieghi il servizio di rete senza connessione(UDP come protocollo di strato di trasporto).
Il software deve permettere:

- 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 server dovrà rimanere in ascolto rispondendo ai tre messaggi che può ricevere dal client, ovvero, la risposta al comando list mandando al richiedente la lista dei file scaricabili dal server; la risposta al comando get contente il file che l'utente vuole scaricare o eventuali messaggi di errore o buona riuscita di tale operazione. Inoltre il server deve poter ricevere file dal client in risposta al comando put.

Il client deve ricevere in input dall'utente quale delle tre operazione tra List, Get e Put egli voglia eseguire e comunicarla al server. Il server deve rispondere con messaggio ultimo l'esito delle operazioni.

# Capitolo 2

# Design

Il software è basato su un'architettura di tipo client-server.
A livello di trasporto si fa l'uso del protocollo UDP, quindi non vi è la garanzia della ricezione del messaggio e nemmeno dell'integrità di quest'ultimo.
Per ovviare a problematiche che possono nascere con la suddivisione di messaggi troppo grandi in vari pacchetti, come può essere la perdita di uno di questi, il mittente invia pacchetto finale con un header speciale e con contenuto il checksum cosicché il ricevente possa verificare la corretta ricezione dell'intero messaggio.

## 2.1 Protocollo

Ad alto livello, client e server si scambiano dei messaggi tra di loro e, in base alla dimensioni dei messaggi, questi sono convertiti in uno o più pacchetti.
La struttura dati del pacchetto è un dizionario composto di due chiavi: "HEADER" e "CONTENT".

Utilizzo, anche, un dizionario packet_header con chiavi i vari tipi di operazioni, a cui corrisponde una tupla con primo valore la conversione in numero del tipo di header e secondo valore un ulteriore informazione per la corretta riuscita della trasmissione.

```
1 packet_header = {'LIST': (1,0),
2                  'GET': (2,0),
3                  'PUT': (3,0),
4                  'SEND': (4,0),
5                  'END': (5,0)}
```

Tramite un'apposita funzione che prende in argomento un header e il suo contenuto si può dividere il messaggio in vari pacchetti:

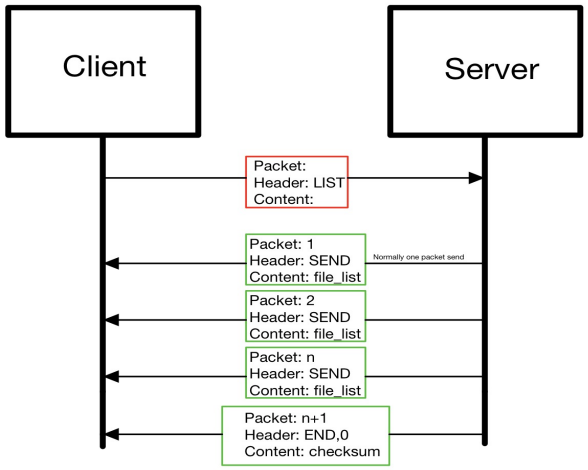
```
def packet_create(header, content = 0):
    packet = {"HEADER" : packet_header[header],
              "CONTENT" : content}
    return json.dumps(packet).encode()
```

Ogni messaggio viene inviato tramite un processo eseguito dalla funzione send, che invia un ulteriore pacchetto finale che è composto da un header di tipo 'END', con secondo valore di questi 0 in caso di buona riuscita o 1 in caso di errore nella trasmissione. Se l'header ha valore 0 il contenuto del pacchetto è il checksum del messaggio, altrimenti il contenuto è il tipo di errore avvenuto.

## 2.2 Comandi

### 2.2.1 List

Il client manda un messaggio, dove nell'header è presente il tipo di operazione List.

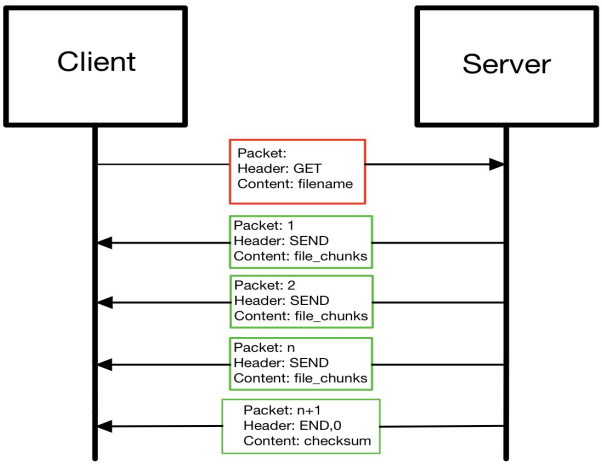



Come si può vedere in figura, il server prende riceve il pacchetto iniziale dove nell'header è specificato il tipo di operazione LIST, a questo punto il server risponderà mandando la lista dei file scaricabili, questo solitamente avviene con

la spedizione di un solo pacchetto, seguito dall'ultimo pacchetto di con header END,0 e il checksum come contenuto.

### 2.2.2 Get

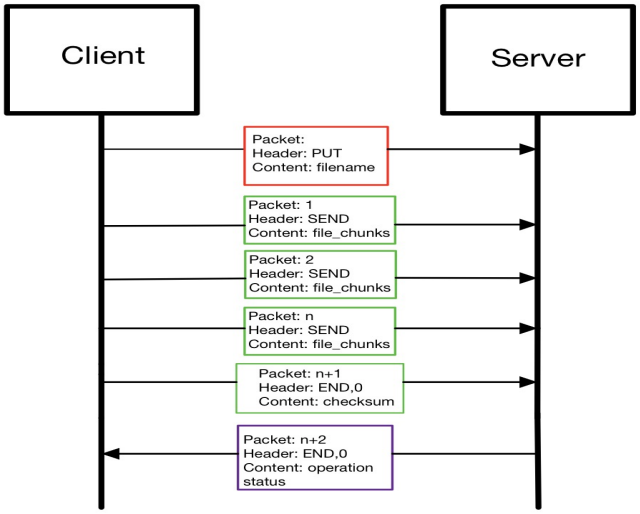
Il client manda un messaggio sotto forma di pacchetto, dove nell'header è presente il tipo di operazione Get e il contenuto è il nome del file che l'utente vuole scaricare




Il server riceve il messaggio con l'operazione Get, legge il file, che se dovesse essere più pesante del buffer del client viene diviso in vari pacchetti con header SEND e come contenuto il file che si vuole spedire. L'ultimo pacchetto spedito dal server è sempre con header END,0 in caso di corretta trasmissione e contenuto il checksum che verrà verificato dal client
Invece, se vi dovessero essere errori di ogni genere il server manda un pacchetto con header: END,1 per segnalare l'errore e con contenuto la descrizione dell'errore, come potrebbe essere la non presenza del file nel server.

### 2.2.3 Put

Il client manda un pacchetto con header di tipo PUT e contenuto il nome del file, poi inizia l'invio del file. Quando il server riceve il primo pacchetto capisce che sta per essere eseguito il comando PUT, quindi si prepara per ricevere il file. Vengono utilizzate le stesse funzioni che usa il comando GET, solo che mittente e ricevente sono diventati rispettivamente client e server.




Una volta ricevuto il file, il server verificherà il checksum e in base a ciò comunicherà al client l'esito dell'operazione.

## 2.3 Codifica Pacchetti

Per poter comunicare tra di loro server e client si scambiano uno o più pacchetti contenente il messaggio che vogliono inoltrare. Un pacchetto corrisponde alla struttura dati dizionario, con chiavi HEADER e CONTET, il tutto viene trasformato in un JSON che prima di essere inoltrato viene codificato in utf-8.

# Capitolo 3

# Sviluppo

Il progetto si compone di tre file:

- **server.py:** contiene il codice per gestire le richieste al server
- **client.py:** contiene il codice per gestire le richieste al client
- **utility_module.py:** contiene delle funzione e stutture dati utilizzate sia dal server e dal client

## 3.1 Sever

Per la creazione del server si fa uso della libreria socket. Il server è caratterizzato da un processo che lo mantiene costantemente in ascolto, infatti, esso può essere terminato solamente con la combinazione di tasti Ctrl+C.
Quando il server riceve un messaggio dal client, lo riceve sottoforma di un pacchetto, quindi ne analizza l'header a cui corrisponde un'operazione da eseguire, infine, risponde con il contenuto richiesto da questa operazione, per poi ritornare in ascolto.

La comunicazione avviene in fase di ascolto tramite la fuzione **receive** presente in **utility_module.py**, mentre in fase di trasmissione con la fuzione **send**, sempre contenuta in **utility_module.py**.

## 3.2 Client

Il client è responsabile della creazione del socket UDP per comunicare con il server, invia le richieste a quest'ultimo iniziando la comunicazione.

Il client stampa a video un piccolo menu di navigazione permette di scegliere all'utente le varie possibilità dell'applicazione inserendo le proprie scelte.

Il client prende questi input e inizia la comunicazione con il server sempre tramite l'uso delle funzioni **send** e **receive** presenti in **utility_module.py**.

## 3.3 utility_module

In questo file sono presenti la maggior parte delle funzioni utilizzate per la comunicazione client-server e per l'esecuzione delle varie operazione richieste dall'utente. Le funzioni principali sono:

- send()
- receive()

La funzione send() prende come argomenti il messaggio e l'indirizzo a cui spedire il primo. La funzione si occupa di dividere il messaggio in vari pacchetti per evitare una congestione del buffer e gestisce, anche, l'invio di eventuale errori che vi possono essere da parte del mittente.

La funzione receive() permette la ricezione dei vari pacchetti per poi riunirli nel messaggio o file che il mittente voleva trasmettere, si occupa inoltre di verificare l'integrità del messaggio utilizzando il controllo checksum.

Vi sono, anche, altre funzioni come read_file() e save_file() usate rispettivamente per leggere e salvare i file sia da client, che da server; la funzione packet_create() è stata già esplicata in precedenza.
Ci sono anche tre funzioni:

- op_one()
- op_two()
- op_three()

usate nel menu di navigazione del client.

# Capitolo 4

# Guida Utente

## 4.1 Envoironment

Per poter eseguire i file presenti all'interno di questo progetto è necessario utilizzare una versione di Python $\geqslant 3.8$.

Non è necessario installare alcun pacchetto aggiuntivo in quanti vengono usati solo moduli presenti nella libreria standard di Python.

## 4.2 Server

Per avviare il server è sufficiente lanciarlo da qualsiasi IDE o eseguire il seguente comando:
python3 server.py

## 4.3 Client

Per avviare il client è sufficiente lanciarlo da qualsiasi IDE o eseguire il seguente comando:
python3 client.py