Architettura Client-Server UDP per Trasferimento File

Sara Cappelletti
Rachele Margutti

10 giugno 2022

Indice

Capitolo 1

Analisi del Problema

1.1 Descrizione

Il progetto si pone come obiettivo quello di implementare in linguaggio Python un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (UDP).
Il software deve permettere:

- Connessione client-server senza autenticazione
- Visualizzazione sul client dei file disponibili sul server
- Download di un file dal server
- Upload di un file sul server

Il client dovrà permettere all'utente di usufruire del comando list per richiedere una lista dei nomi dei file disponibili e del comando get per ottenere un determinato file. Sarà inoltre necessario presentare un messaggio di feedback ogni volta che le operazioni vengono effettuate, sia in caso di fallimento che di successo.

Il server dovrà invece permettere l'invio del messaggio di risposta al comando list del client precedentemente citato con la lista dei file disponibili in quel momento, il messaggio di risposta al comando get contenente il file richiesto se presente, altrimenti un messaggio d'errore e infine la ricezione di un comando put contenente il file da caricare sul server con relativo messaggio d'errore in caso di fallimento. Il server dovrà inoltre permettere molteplici accessi concorrenti.

Capitolo 2

Design

Il software è basato su un'architettura client-server in cui il server è in grado di supportare più client connessi in parallelo.

Ai fini della nostra applicazione, il protocollo UDP presenta varie problematiche che è stato necessario risolvere:

1. Non garantisce la ricezione del messaggio
2. Non garantisce che l'ordine di invio sia lo stesso di quello di ricezione
3. Non presenta un concetto di connessione

Per ovviare ai problemi `1` e `2`, dopo la ricezione di un messaggio è necessario rispondere con un messaggio di `OK`. Questo permette di verificarne la ricezione e forza l'invio di un messaggio alla volta, risolvendo così eventuali problemi di ordine.
Il tradeoff è una minore velocità di trasmissione in quanto è sempre necessario attendere una risposta prima di poter inviare il prossimo messaggio e non è possibile inviare più messaggi in parallelo.

Il punto `3` è un problema in quanto il server e un client devono potersi inviare più di un messaggio (e.g., un singolo messaggio può dover essere diviso in più parti poiché troppo grande per essere contenuto in un unico datagram UDP). Nel caso del client questo non è veramente un problema in quanto comunica solo ed esclusivamente con il server. Nel caso del server però, è possibile comunicare con più client in parallelo ed è quindi necessario gestirli separatamente, tenendo conto di quale client ha inviato la richiesta. Questo problema viene risolto simulando il concetto di connessione:

- Alla ricezione di un nuovo messaggio di comando (i.e., `PUT`, `GET` o `LIST`) viene creato un nuovo Server dedicato solo alla gestione di quel comando per quel determinato client.
- I successivi messaggi (non di comando) provenienti da quel client (identificato da IP e porta) saranno reindirizzati solo al server precedentemente creato.

2.1 Protocollo

Ad alto livello, client e server comunicano tramite `Message`.
Un `Message` è così composto:

```python
class Message:
    content: bytes
    is_command: bool = False
    is_error: bool = False
    has_more: bool = False
```

Figura 2.1: Class Message

I parametri sono:

- `content`: contenuto del messaggio
- `is_command`: un booleano che indica se è un comando
- `is_error`: un booleano che indica se è un errore
- `has_more`: un booleano che indica se il messaggio è stato diviso in più messaggi e, in tal caso, se questo è l'ultimo messaggio o meno

2.1.1 Comandi

Comando PUT

Il comando `PUT` prevede l'invio da parte del client del nome del file, il contenuto del file e il suo checksum (SHA256).

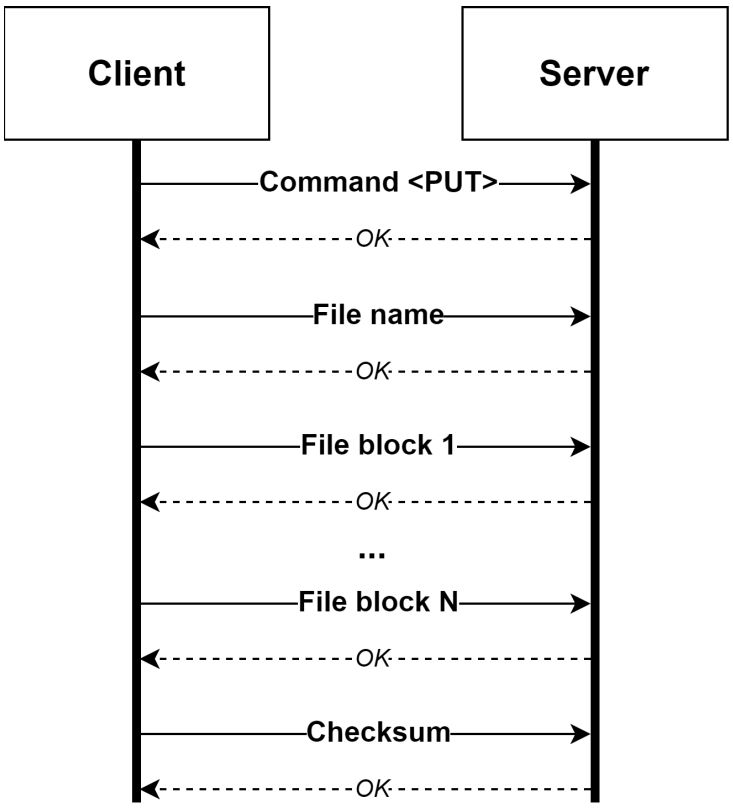

Figura 2.2: Put Command

Il server dovrà quindi creare un file con il nome specificato, scriverci i blocchi ricevuti e verificare che il checksum inviato dal client sia lo stesso del file generato.

Per evitare di scrivere un file parziale o corrotto, il file viene prima creato all'interno di una cartella temporanea e solo dopo aver verificato il checksum verrà spostato all'interno della cartella contenente i file del server.

In caso di errore (e.g., il checksum non è uguale) il server invierà al client un messaggio di errore.

```
send(Command.PUT)
send(filename)
send_file(input_path)
```

```
filename = receive()
check_filename_valid(filename)
receive_file(filename)
```

Figura 2.3: Pseudo-codice PUT client - server

Comando GET

Il comando `GET` prevede l'invio da parte del client del nome del file. Il server dovrà rispondere inviando il contenuto del file e il suo checksum.

Se un file con quel nome non esiste, il server invierà al client un messaggio di errore.

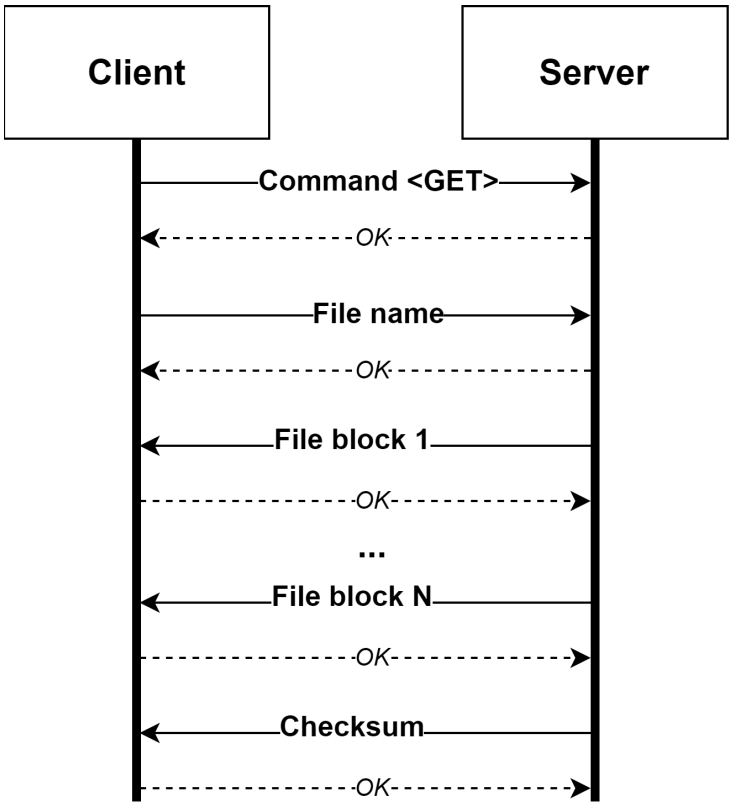

Figura 2.4: GET Command

Il client dovrà quindi creare un file con il nome specificato, scriverci i blocchi ricevuti e verificare che il checksum inviato dal server sia lo stesso del file generato.

Per evitare di scrivere un file parziale o corrotto, il file viene prima creato all'interno di una cartella temporanea e solo dopo aver verificato il checksum verrà spostato nella path indicata.

In caso di errore (e.g., il checksum non è uguale) il client dovrà mostrare un messaggio di errore.

```
send(Command.GET)
send(filename)
receive_file(output_path)
```

```
filename = receive()
check_filename_valid(filename)
check_file_exists(filename)
send_file(filename)
```

Figura 2.5: Pseudo-codice GET client - server

Comando LIST

Il comando `LIST` prevede l'invio da parte del server della lista di tutti i file attualmente presenti al suo interno. I nomi dei file sono inviati separati da un carattere di "a capo" (`\n`).

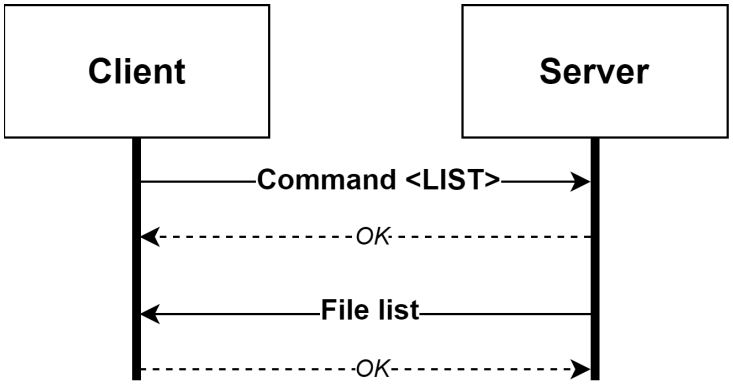

Figura 2.6: LIST Command

Il client dovrà semplicemente ricevere la lista di nomi e mostrarla all'utente.

```
send(Command.LIST)
filelist = receive()
print(filelist)
```

```
send('\n'.join(get_stored_files()))
```

Figura 2.7: Pseudo-codice LIST client - server

2.2 Gestione di Messaggi di Grandi Dimensioni

Può succedere di dover inviare messaggi di grandi dimensioni, per esempio nel caso in cui il server contiene un elevato numero di file e deve rispondere al comando LIST.

Questa problematica viene gestita in modo trasparente dal protocollo di comunicazione: durante l'invio il messaggio viene suddiviso in più messaggi di dimensioni minori; mentre durante la ricezione si attendono tutti i blocchi e li si ricompone in un unico messaggio.

Più nel dettaglio, i messaggi generati avranno la flag `has_more` settata a `True`, tranne l'ultimo che sarà settato a `False` per far sapere al destinatario quando ricomporli.

2.3 Gestione di File di Grandi Dimensioni

Sarebbe possibile gestire l'invio del contenuto dei file come un normale messaggio, come descritto nella sezione precedente.
Tuttavia, a differenza di normali messaggi, un file può avere dimensioni molto elevate, anche maggiori della RAM a disposizione e non sarebbe quindi possibile ricomporre tutti i blocchi in memoria.

Per questo si utilizza un approccio diverso per cui, dopo la ricezione di ogni blocco, questo viene scritto direttamente su disco, componendo messaggio per messaggio il file originale.

Alla fine della ricezione di tutti i blocchi, viene anche inviato un checksum (`SHA256`) per verificare che siano stati ricevuti correttamente.
Per evitare di ottenere un file corrotto, questo viene prima creato all'interno di una cartella temporanea e solo dopo aver verificato il checksum verrà spostato nella path desiderata.

2.4 Gestione errori

Durante la gestione di un comando, si possono verificare errori per due motivi:

- Errore di comunicazione: è scaduto il timeout di ricezione del messaggio

- Errore interno: qualcosa è andato storto lato server (e.g., il file richiesto non esiste)

2.4.1 Errori lato server

In caso di errore, il server annulla la gestione dell'intero comando e da per scontato che il client lo esegua nuovamente se necessario.
Inoltre, se l'errore è interno, il server manda al client un messaggio di errore (con `is_error` settato).

2.4.2 Errori lato client

Non appena il client riceve un messaggio di errore da parte del server, questo messaggio viene stampato e il client viene chiuso. Similmente, nel caso in cui il server non risponda oltre la durata del timeout, il client stampa un messaggio di errore ed esce.

2.5 Codifica `Message`

Per poter inviare oggetti `Message` tra client e server è necessario convertirli in un array di byte. Il contenuto del messaggio (`content`) è già un array di byte, restano quindi solo da convertire le flag `is_command`, `is_error` e `has_more`.
Il datagram inviato è composto da un header grande 1 byte e dal `content` del messaggio.

L'header è così composto:

- il primo bit meno significativo settato se `has_more`
- il secondo bit meno significativo settato se `is_command`
- il terzo bit meno significativo settato se `is_error`

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 inviare le richieste al server
- `common.py`: contiene classi e funzioni condivise tra client e server

3.1 Server

Per creare il server viene utilizzata la classe Python `ThreadingUDPServer` all'interno della libreria standard `socketserver`, che crea in automatico un nuovo thread per ogni datagram ricevuto.

`ThreadingUDPServer` richiede una classe da utilizzare per gestire il datagram ricevuto, attraverso l'apposito metodo `handle()`. All'interno di `server.py`, questa classe si chiama `RequestHandler`.

La vera classe che si occupa di gestire i comandi richiesti dal server è `Server`

`RequestHandler` si occupa di istanziare un'oggetto di tipo `Server` ogni volta che viene ricevuto un nuovo comando e di inoltrare tutti i successivi messaggi ricevuti da quel client (identificato da IP e porta) all'oggetto precedentemente creato.

Per fare questo viene usata una `SimpleQueue` del modulo `queue` di Python. Le `Queue` permettono di inviare e attendere nuovi messaggi in maniera efficiente tra due thread differenti.

Ogni `Server` creato possiede quindi una propria queue all'interno della quale il `RequestHandler` inserirà i nuovi messaggi indirizzati a quel `Server`.

In questo modo è possibile gestire la ricezione e l'invio di messaggi in maniera sincrona e bloccante, semplificando notevolmente il codice di gestione dei comandi. Infatti per ricevere il prossimo messaggio sarà sufficiente chiamare il metodo `receive()` (il quale attenderà il messaggio all'interno della coda).

3.2 Client

La classe `Client` è responsabile della creazione del socket UDP per comunicare con il server, dell'invio dei comandi e della relativa comunicazione.

La gestione del client è molto più semplice in quanto comunica solamente con il server. Quindi per la ricezione dei messaggi è sufficiente chiamare il metodo `recv` sull'oggetto `socket` creato.

3.3 Common

All'interno di `common.py` è presente:

- La classe `Message`
- La classe astratta `MessageHandler`, usata sia da `Client` che da `Server` per astrarre la ricezione e invio di messaggi
- Funzioni per la codifica, decodifica e splitting dei messaggi.
- Costanti di configurazione (e.g., secondi di timeout, massima dimensione dei `Message`)

Capitolo 4

Guida Utente

4.1 Environment

Per poter eseguire i file presenti all'interno di questo progetto è necessario usare una versione di Python ≥ 3.8.

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

4.2 Server

Per avviare il server è sufficiente il seguente comando:

```
python3 server.py
```

4.3 Client

Tutti i comandi disponibili e i relativi argomenti sono disponibili tramite il comando `--help`.

4.3.1 Comando LIST

```
python3 client.py list
```

4.3.2 Comando PUT

```
python3 client.py put nomefile /path/al/file
```

4.3.3 Comando GET

```
python3 client.py get nomefile /path/in/cui/salvarlo
```