Relazione elaborato di Programmazione di Reti

Componenti del gruppo

Cardone, Marco, 0000975894

Desiderio, Marco, 0000839614

Corso: Programmazione di Reti

Anno accademico: 2021-2022

Traccia scelta

N°2: Architettura client-server UDP per trasferimento file

Repository github

https://github.com/MarcoChardonnay/RetiProgettoUDP

Scopo del progetto

Lo scopo del progetto è realizzare una comunicazione UDP client-server per lo scambio di dati in Python.

Schema di progettazione dei programmi

Entrambi i programmi, client e server, sono stati realizzati seguendo la struttura seguente:

import	Inclusione delle librerie necessarie per l'utilizzo di specifiche funzionalità
costanti	Definizione delle variabili (*simil-costanti*) usate nel programma, inserite per semplificare la gestione dell'applicazione e le successive modifiche (ad esempio non esaustivo IP e porta del client/server)
procedure	Definizione delle funzioni di utilità per il programma
variabili di gestione	Definizione delle variabili utilizzate nel programma
controlli	Esecuzione dei controlli necessari ad avviare correttamente il programma oppure per l'utilizzo di specifiche funzionalità del programma
main	Programma vero e proprio

Analisi del Server

import

Sono stati importati **`socket`** per gestire la comunicazione e **`os`** per la gestione del filesystem. In più **`datetime.datetime`** per la stampa di data e ora nei messaggi di log.

costanti

- **`SERVER_HOST`**: IP su cui ci si vuole mettere in ascolto
- **`SERVER_PORT`**: Porta su cui ci si vuole mettere in ascolto
- **`UPLOAD_CHUNK`**: Grandezza massima in byte di una porzione di file durante lo scambio di dati con il client
- **`FILES_PATH`**: Percorso in cui si trovano i file scaricabili dal client

Chiaramente le prime tre è necessario che siano "sincronizzate" con il client

procedure

- **`cls()`**: Per eliminare tutti i messaggi nella schermata del terminale.
- **`log(message, address=False)`**: Per stampare un messaggio in console già formattato e preceduto da data e ora ed eventualmente l'IP del client che lo ha reso necessario.
- **`sendMessage(sock, address, message, encode=True)`**: Per inviare un messaggio ad un client. Permette, tramite il parametro encode, di scegliere se inviare il dato come stringa (True) o puramente in binario (False).

variabili di gestione

- **`sock`**: Per gestire la comunicazione UDP
- **`check_success:`** Per la gestione degli errori pre-avvio

controlli

Durante l'avvio del server, prima di tutto viene effettuato un controllo di esistenza della cartella in cui verranno caricati i file. Per scelta implementativa abbiamo deciso di crearla noi in automatico se non presente.
In caso la cartella non potesse essere creata per qualche motivo, ad esempio la mancanza dei permessi di scrittura, viene mostrato un errore a video e il programma non viene avviato.
Una volta che la cartella esiste, si dà per scontato che si abbiano i permessi di lettura e scrittura per poter gestire i file del sistema.
Il controllo di corretto avvio del socket è rimandato alla parte vera e propria di esecuzione del programma (main) poiché fatto utilizzando un try-except.

Analisi del Client

import

Sono stati importati **`socket`** per gestire la comunicazione e **`os`** per la gestione del filesystem.

costanti

- **`SERVER_HOST`**: IP su cui ci si vuole mettere in ascolto
- **`SERVER_PORT`**: Porta su cui ci si vuole mettere in ascolto
- **`UPLOAD_CHUNK`**: Grandezza massima in byte di una porzione di file durante lo scambio di dati con il client
- **`FILES_PATH`**: Percorso in cui si trovano i file scaricabili dal client

Chiaramente le prime tre è necessario che siano "sincronizzate" con il client

procedure

- **`cls()`**: Per eliminare tutti i messaggi nella schermata del terminale
- **`wait()`**: Per l'attesa di un input dall'utente
- **`sendMessage(sock, message, encode=True)`**: Per inviare un messaggio al server. Permette, tramite il parametro encode, di scegliere se inviare il dato come stringa (True) o puramente in binario (False).
- **`info(message)`**: Per stampare un messaggio di informazione in azzurro
- **`success(message)`**: Per stampa un messaggio di successo in verde
- **`error(message)`**: Per stampare un messaggio di errore in rosso
- **`header()`**: Per stampare l'intestazione con componenti del gruppo e indicazione che il pannello è di tipologia client

variabili di gestione

- **`sock`**: Per gestire la comunicazione UDP
- **`upload_available`**: Per gestire la possibilità di caricare file sul server
- **`downloading_file`**: Per mantenere il riferimento al file durante l'operazione di download
- **`option:`** per la gestione degli input dell'utente

controlli

Viene controllata l'esistenza della cartella per il caricamento dei file, in caso in cui non esista, l'utente può scegliere di farla creare al programma che sfrutta il comando `os.mkdir`; nel caso in cui la cartella non esista e l'utente decide di non farla creare viene disabilitata l'opzione di caricamento di file nel server.
Il controllo di corretta connessione al server è rimandato alla parte vera e propria di esecuzione del programma (main), poiché la trasmissione dati è in UDP e non c'è una vera e propria "connessione".

Scelte implementative del Server

Utilizzo della costante FILES_PATH

Abbiamo deciso di utilizzare una sottocartella che contenga i file destinati al cloud e non la cartella stessa del server, per evitare che il server stesso fosse scaricabile o sovrascrivibile:

```
FILES_PATH = "./files/"
```

Funzione di log

Per comprendere meglio cosa succede, viene utilizzata la funzione log per mostrare a schermo i messaggi di flusso con data, ora e indirizzo del client che ha generato il log:

```
def log(message, address=False):
    print("\x1b[0;36m["+datetime.now().strftime("%d/%m/%Y
%H:%M:%S")+"]"+(" ["+address[0]+":"+str(address[1])+"]" if address else
"")+"\x1b[0m "+message)
```

Comando list

Alla ricezione del comando di list, viene creato un vettore e, scorrendo i valori restituiti da `os.scandir(FILES_PATH)`, vengono inseriti soltanto i nomi soltanto dei file e non delle cartelle. Gli elementi del vettore vengono poi uniti separandoli dal carattere ":". La scelta di questo carattere è strategica poiché non è possibile che compaia all'interno del nome di un file.

Comando put

Il comando di upload richiede che, per iniziare un upload, venga inviato il comando "put start", seguito dal nome del file da caricare. Si dà per scontato che il nome del file sia sempre puro e non formato da un percorso.
Alla ricezione del comando di inizio, viene aggiunto un elemento al vettore **`uploading_file`** utilizzando come chiave la porta con cui si è connesso il client. Tale elemento sarà il puntatore al file da creare/sovrascrivere.
L'utilizzo del vettore è una scelta implementativa data dal fatto che abbiamo scelto di **permettere l'upload da parte di più client contemporaneamente**.
Tutti i messaggi successivi si intendono parte del file fino alla ricezione di un messaggio "put end". Ogni parte è formata da al massimo **`UPLOAD_CHUNK`** byte.
Al termine dell'upload viene chiuso il file e rimosso il suo riferimento nel vettore **`uploading_file`**.
Per nostra scelta, se viene caricato un file con un nome già esistente, il file viene sovrascritto con il contenuto del nuovo file.
Alla ricezione di ogni parte del file, viene spedito indietro al client un messaggio "put ok", per segnalargli che il server è pronto a ricevere il prossimo messaggio.

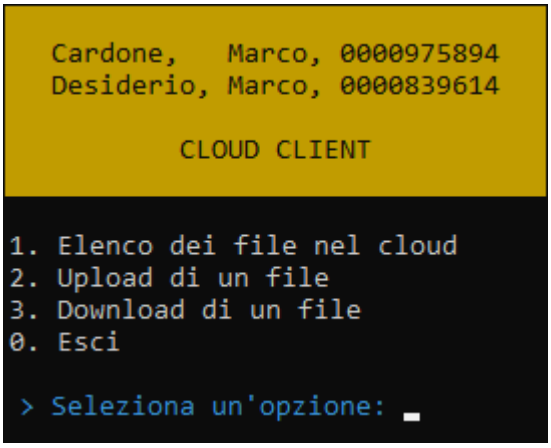
Comando get

Il comando di download richiede che venga inviato il comando “get”, seguito dal nome del file da scaricare. Si dà per scontato che il nome del file sia sempre puro e non formato da un percorso.
Alla ricezione del comando, viene controllato che il file esista. Se il file non esiste, viene inviato un messaggio testuale di errore. Se il file esiste invece viene inviato, per iniziare, il messaggio “get start”, successivamente tutte le parti di grandezza massima **`UPLOAD_CHUNK`** byte e infine il messaggio “get end”.

Scelte implementative del Client

Interfaccia

L'interfaccia è user-friendly e mostra un menu numerico con le tre funzionalità richieste: elenco (**list**), upload (**put**) e download (**get**), più la possibilità di terminare l'esecuzione del client:



Il sistema riconosce l'eventuale opzione non valida.

Scelta dell'opzione dal menu iniziale

Nonostante ci si aspetti un valore numerico, l'input è gestito come stringa per evitare di dover gestire con un try-except l'eventuale input non numerico.

Controllo: cartella di upload non presente

Se la cartella di upload non esiste, viene proposto all'utente di crearla. Se decide di non crearla, oppure la creazione non va a buon fine, ad esempio per mancanza di permessi, l'opzione di upload viene disattivata.

Comando list

All'invio del comando list, il client si mette in attesa di un messaggio di risposta. Alla sua ricezione, controlla prima di tutto se il messaggio è vuoto e, se lo è, mostra un messaggio informativo sul fatto che non ci sono file nel cloud; altrimenti mostra, uno sotto l'altro, i file nel cloud.
Il messaggio è formato dall'elenco dei file, separati dal carattere due punti. Il client separa questo elenco creando un vettore e stampando poi ogni elemento preceduto dal trattino per una visualizzazione ad elenco.

Comando put

All'invio del comando put, prima di tutto il client controlla che sia attivo il comando e, se non lo è, mostra un messaggio di errore.
Se invece il comando è attivo, viene prima di tutto richiesto il nome del file da caricare e verificato che esista nella cartella; in caso negativo, viene mostrato un messaggio di errore.
In caso il file esista, invece, viene inviato un comando "put start" seguito dal nome del file, una sequenza di pacchetti lunghi al massimo **`UPLOAD_CHUNK`** byte, ognuno dei quali è una parte del file e, infine, il comando "put end".
All'invio di ogni parte del file, il client rimane in attesa di un messaggio del server che indichi che il server è pronto a ricevere il prossimo messaggio.

Comando get

All'invio del comando get, viene prima di tutto richiesto il nome del file da scaricare e poi inviato il comando "get" seguito dal nome del file e poi il client si mette in attesa di un messaggio dal server. Se il messaggio appena successivo è "get start", allora tutto è andato a buon fine e il file sta per arrivare, altrimenti si tratta di un messaggio di errore che il client mostra a video.
Alla ricezione del comando di inizio, viene impostata la variabile di gestione **`downloading_file`** come il puntatore al file da scaricare.
Tutti i messaggi successivi si intendono parte del file fino alla ricezione di un messaggio "get end". Ogni parte è formata da al massimo **`UPLOAD_CHUNK`** byte.
Al termine del download viene chiuso il file e la variabile **`downloading_file`** viene reimpostata a **`False`**.
Per nostra scelta, se viene caricato un file con un nome già esistente, il file viene sovrascritto con il contenuto del nuovo file.
Alla ricezione di ogni parte del file, viene spedito indietro al client un messaggio "put ok", per segnalargli che il server è pronto a ricevere il prossimo messaggio.

Scelte implementative generiche

Gestione della lettura del nome di un file

Il nome di un file può contenere anche degli spazi. Lo spazio però è il carattere di separazione tra un'informazione e un'altra, nei comandi. Perciò, nel caso in cui ci si trovi un comando che invia il nome di un file, tutte le parti di comando separate da spazio, a partire dalla posizione in cui si dovrebbe trovare il nome del file in avanti, vengono riunite con uno spazio nel mezzo:

```
filename = " ".join(data[2:])
```

Invio di file più grandi della dimensione del buffer

Ovviamente il buffer di lettura non è infinito. Abbiamo scelto quindi di leggere i file da spedire "a pezzi" grandi quanto la grandezza del buffer (chunk) e poi spediti separatamente:

```
part = file.read(UPLOAD_CHUNK)
...
sendMessage(sock, part, False)
```