DOCUMENTO DI PROGETTO

TRACCIA 2

RELAZIONE

realizzato da **Filippo Velli** – mail: filippo.velli@studio.unibo.it – # di matricola: **0000977075**

INTRODUZIONE

La mia relazione del progetto sarà strutturata nel modo seguente:

- breve riepilogo sulla traccia scelta
- strutture dati da me utilizzate
- funzionamento generale dell'applicazione
- scelte di progetto che forniscono ulteriori dettagli sull'applicazione

L'applicazione è stata sviluppata e commentata in lingua inglese, mentre le indicazioni d'uso sono state redatte sia in inglese che in italiano. Ho realizzato il progetto da solo.

RIEPILOGO TRACCIA

La traccia da me scelta richiedeva la realizzazione di un'applicazione client/server in linguaggio Python che fosse in grado di trasferire file da server a client e viceversa utilizzando come protocollo di strato di trasporto UDP. Inoltre, la connessione tra il client e il server doveva restare senza autenticazione e l'applicazione doveva permettere, oltre all'upload di file al server e al download di file dal server, anche al client di poter vedere i file disponibili sul server.

DESCRIZIONE DELLE STRUTTURE DATI

Per realizzare l'applicazione richiesta è stato necessario l'utilizzo di due socket (oggetti software che consentono di trasmettere e ricevere dati tra host remoti o tra processi locali), uno che funga da server a cui viene associato un indirizzo e una porta fissi e un altro che funga da client. In questo caso i socket sono di tipo SOCK_DGRAM, ovvero fanno in modo che la comunicazione tra di loro non sia una connessione nel vero senso della parola, tipica di un socket TCP/SOCK_STREAM, ma sia realizzata mediante l'invio di datagrammi senza garantire che quest'ultimi siano inviati nell'ordine corretto o che non vengano corrotti durante l'invio.

Il client e il server si scambiano dei messaggi di diverso tipo mediante le funzioni sendto() e recvfrom() del modulo socket e nel caso in cui i dati scambiati tra i due non siano in byte si utilizzano le funzioni encode() dal socket mittente e decode() da quello ricevente per permetterne il trasferimento. Ciò è dovuto al fatto che Python 3 non lascia che i dati possano essere corrotti e rileva gli errori di codifica, sia dal socket mittente che da quello ricevente. Nell'applicazione sono usate anche delle liste che vengono in quasi ogni occasione rese in stringhe con la funzione join().

FUNZIONAMENTO DELL'APPLICAZIONE

Il client e il server non richiedono alcun argomento per la loro attivazione (l'unica accortezza è che il file del server venga attivato nella cartella in cui si trova se si intende usarlo da linea di comando) e dopo i controlli necessari al funzionamento del server, quest'ultimo attende una comunicazione preliminare dal client per iniziare la comunicazione tra i due socket. Prima il server invia al client la lista di comandi disponibili tra i quali, tramite la funzione input(), l'utente può selezionare quello desiderato (situazione di default). Scelto il comando, il server controlla che il comando richiesto dal client sia presente tra quelli disponibili e, se lo è, provvede a rispondere o a domandare al client il file da scaricare dal server o da caricare sul server, altrimenti provvede a comunicare l'esito.

Prima del trasferimento vengono svolti dei controlli ulteriori per far sì che il file coinvolto nell'operazione di upload (comando put) esista e che non abbia omonimie con file del server o nel caso di download (comando get) che il file sia presente sul server. Nel caso in cui qualcosa dovesse andare storto il socket che individua il problema comunica all'altro lo stato e si ritorna alla situazione di default. In fase di upload il server chiede al client il percorso assoluto del file, mentre per quella di download solo il nome e l'estensione del file richiesto. Se viene richiesta la lista dei file disponibili (comando list), il server la invia al client.

All'utente viene inoltre fornita la possibilità di interrompere la connessione tra il client e il server tramite il comando bye che, una volta inviato, fa chiudere il socket del client, mentre il server resta in attesa di un nuovo client.

SCELTE DI PROGETTO

PRESUPPOSTI. Dovendo realizzare un'architettura client/server UDP come prima cosa ho preso come punto di partenza un semplice client/server UDP su cui poi aggiungere le funzionalità richieste. Ho deciso in questa fase di far sì che il socket del server fosse associato al 'localhost' porta 10000 e che il client fosse a conoscenza dell'indirizzo del server. Da subito ho notato delle similitudini con un tipico client/server telnet, dato che vi è una comunicazione bidirezionale tra server e client in cui quest'ultimo manda richieste al primo, il quale risponde coerentemente. Bisogna però ricordare che per un'applicazione telnet viene usato un protocollo TELNET su TCP e non su UDP come richiesto dalla traccia.

MULTIPIATTAFORMA. Già nelle prime fasi ho voluto garantire, tramite funzioni disponibili in moduli di Python, che l'applicazione da me realizzata fosse multipiattaforma e non limitata all'esecuzione solo sulla piattaforma su cui ho lavorato, ovvero Windows. Per essere specifici sono state utilizzate funzioni dei moduli pathlib (PurePath().parts) e os (listdir(), getcwd() e linesep).

MULTITHREADING. Visto che tra le specifiche non viene richiesto di poter connettere contemporaneamente più client al server, ho scelto di non utilizzare forme di Multithreading per accettare più client, ma di limitare i collegamenti tra il server e solo un client. Il server usa due cicli annidati (while True): quello esterno rimane attivo nell'attesa del collegamento

di un client, mentre quello interno è attivo quando c'è un client e permette a quest'ultimo di inviare i comandi necessari all'esecuzione dell'applicazione.

CONTROLLI E MESSAGGI DI ERRORE. Nell'applicazione sono previsti tre tipi di errori:

- Errori dei presupposti iniziali: nel server viene controllata sia l'esistenza della cartella Server che la presenza di file in quest'ultima, condizioni necessarie al funzionamento dell'applicazione che altrimenti viene interrotta prima della creazione del socket; l'utente viene informato dell'errore e come rimediarvi. Questo tipo di controllo viene ripetuto subito dopo che un client invia la richiesta al server di eseguire uno dei comandi disponibili. Se in quel momento non dovesse esserci la cartella Server o file in essa, il server farà in modo di terminare l'esecuzione del client, del server stesso e la chiusura dei rispettivi socket. Questo controllo viene eseguito tramite la funzione preliminary_check().
- Errori nei comandi richiesti: nel caso in cui il client invii una richiesta che il server non può soddisfare, il server lo comunica all'altro socket e viene ripristinata la condizione di default dove il client potrà scegliere la prossima richiesta.
- Mancanza dei file richiesti e delle condizioni per il trasferimento dei file: prima che i file possano venir mandati da un socket all'altro vengono effettuati dei controlli diversi se si tratta del server o del client: viene controllato che i file richiesti dal client durante l'operazione di get siano presenti sul server e non esistano omonimie tra il file inviato dal client per l'upload e uno dei file già presenti. Ci si assicura inoltre che il file coinvolto nell'operazione di get non sia vuoto, avvertendo l'utente in tale evenienza. Ciò viene gestito dalla funzione check_file_status() presente sul server. Nel client invece viene controllato che il file esista prima dell'upload e che non sia vuoto.

Nei casi non vincolanti al funzionamento di base dell'applicazione viene comunicato l'errore all'altro socket che poi fa in modo di ritornare al caso base, ovvero alla selezione del comando da parte dell'utente tramite il client.

GESTIONE DELLA LISTA (LIST). Per generare la lista di file presenti sul server ho valutato due opzioni:

- la prima era di tenere traccia dei file disponibili su un file in formato txt che verrebbe modificato ogni qual volta un file venisse aggiunto (o rimosso manualmente dall'utente);
- la seconda invece prevedeva di creare una sottocartella in cui si sarebbero salvati tutti i file del server e la lista si sarebbe realizzata con la funzione listdir() del modulo os.

Per quanto la prima avrebbe potuto permettere una certa flessibilità nella locazione dei file, ho reputato più efficiente e meno probabile che generi errori il secondo approccio, anche perché i file non devono venir aggiunti manualmente a una lista e il controllo della loro esistenza risulta anche esso semplice, se non più semplice dell'altro metodo, grazie a funzioni fornite da moduli Python os e pathlib. Perciò vi è presente una cartella di nome Server con dei file d'esempio.

IMPLEMENTAZIONE DEL TRASFERIMENTO DEI FILE (GET E PUT). Per la realizzazione delle funzionalità di get (download di un file dal server) e di put (upload di un file sul server) la prima cosa che doveva essere gestita era il trasferimento dei file da un socket all'altro e per gestirlo ho scelto di inviare i file in byte facendo sì che il socket mittente apra il file richiesto in lettura binaria e invii i byte letti (usando un ciclo while che dura fino a che non si raggiunge il EOF) al socket ricevente, il quale creerà un file omonimo (o sovrascriverà il file se esiste già, solo nel caso in cui il socket ricevente sia il client), aperto in scrittura binaria, e con la funzione write() scriverà sul nuovo file i dati ricevuti (sempre usando un ciclo while da cui si esce con la funzione settimeout() del modulo socket).

Per fare queste operazioni di trasferimento ho creato le funzioni send() e receive() che in base al comando richiesto verranno attivate una nel client e l'altra nel server. Nei due socket le funzioni variano leggermente, principalmente nella cartella in cui verranno salvati i file ricevuti (per il server la sottocartella Server, mentre per il client la cartella in cui si trova il file Python). Visto che le differenze sono minime, piuttosto che avere funzioni pressoché identiche sia nel server che nel client ho deciso di salvarle in un modulo che si chiama sendrecieve.py e in base all'argomento user, le funzioni variano leggermente per soddisfare le necessità dell'uno o dell'altro socket.