RELAZIONE PER PROGETTO DI PROGRAMMAZIONE DI RETI

Traccia 2

CONTATTO STUDENTE

Studente: Giovanni Muccioli

Email: giovanni.muccioli@studio.unibo.it

Matricola: 0000977852

OBBIETTIVO

Realizzazione di un'architettura client-server UDP per trasferimento file (sviluppo della traccia numero 2).

TRACCIA e DETTAGLI IMPLEMENTATIVI

Lo scopo del progetto è quello di progettare e implementare in linguaggio Python un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (socket tipo SOCK_DGRAM, ovvero 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;

Inoltre, la comunicazione tra client e server deve avvenire tramite un opportuno protocollo. Il protocollo di comunicazione deve prevedere lo scambio di due tipi di messaggi:
• messaggi di comando: vengono inviati dal client al server per richiedere l'esecuzione delle diverse operazioni;
• messaggi di risposta: vengono inviati dal server al client in risposta ad un comando con l'esito dell'operazione.

INTRODUZIONE

L'architettura si basa sul protocollo di rete UDP (*User Datagram Protocol*), un protocollo di livello di trasporto di tipo *connectionless* (non orientato alla connessione), *stateless* (non tiene nota dello stato della connessione) e definibile come poco affidabile, in quanto non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi. Proprio per queste sue caratteristiche, è un protocollo in grado di supportare molti client attivi ed allo stesso tempo è molto rapido ed efficiente per le applicazioni "leggere", poiché esso garantisce soltanto i servizi basilari del livello di trasporto, ovvero:

• multiplazione delle connessioni, ottenuta attraverso il meccanismo di assegnazione delle porte.
• verifica dell'integrità dei dati mediante una checksum, inserita nell'header del pacchetto.

In questo contesto, che sta alla base dell'architettura, si sviluppa il sistema client-server: un computer (client) si connette ad un server per la fruizione di un servizio, come per esempio la condivisione di una certa risorsa hardware/software, appoggiandosi alla sottostante architettura protocollare precedentemente descritta.
Il server, ricevendo la richiesta dal client, risponderà attraverso la porta e l'indirizzo del client.

CLIENT

Un client è un qualunque componente software, presente tipicamente su una macchina *host*, che accede ai servizi o alle risorse di un'altra componente detta server, attraverso l'uso di determinati protocolli di comunicazione.

In particolare, il client richiesto deve fornire le seguenti funzionalità:
• L'invio del messaggio *list* per richiedere la lista dei nomi dei file disponibili;
• L'invio del messaggio *get* per ottenere un file
• La ricezione di un file richiesta tramite il messaggio di *get* o la gestione dell'eventuale errore
• L'invio del messaggio *put* per effettuare l'upload di un file sul server e la ricezione del messaggio di risposta con l'esito dell'operazione.

ESECUZIONE CODICE CLIENT

Il client è colui che richiede le risorse presenti sul server, quindi il processo client dovrà essere eseguito soltanto dopo che il processo server è già stato azionato per ottenere da esso la risposta desiderata.

- ***python UDP_Socket_Client.py***

(oppure "Run" UDP_Socket_Client.py su Spyder)

SERVER

Un server è un componente informatico di elaborazione e gestione del traffico di informazioni che fornisce un servizio ad altre componenti: i *clients* precedentemente descritti.
Il server sarà in ascolto su una precisa porta in attesa della ricezione dei messaggi. Ogni sua risorsa sarà poi raggiungibile dal client tramite la connessione al socket del server, un'astrazione software progettata per la apposita trasmissione e ricezione di dati attraverso una rete, identificata univocamente da indirizzo IP dell'host del server e dalla porta.

In particolare, il server richiesto deve fornire le seguenti funzionalità:
• L'invio del messaggio di risposta al comando *list* al client richiedente;
• il messaggio di risposta contiene la file list, ovvero la lista dei nomi dei file disponibili per la condivisione;
• L'invio del messaggio di risposta al comando *get* contenente il file richiesto, se presente, od un opportuno messaggio di errore;
• La ricezione di un messaggio *put* contenente il file da caricare sul server e l'invio di un messaggio di risposta con l'esito dell'operazione.

ESECUZIONE CODICE SERVER

Il server sarà azionato prima che il client possa fare la richiesta, in modo da mettersi in ascolto su porta ed indirizzo IP che permetteranno poi la trasmissione dei dati richiesti dal client.

- ***python multithread.py***

(oppure "Run" multithread.py su Spyder)

SCELTE PROGETTUALI E GUIDA PER L'UTENTE

Per la realizzazione del progetto viene utilizzato il multithreading con lo scopo di permettere di ricevere ma anche inviare messaggi, di gestire più richieste da più client in modo contemporaneo ma anche di gestire la possibilità di interruzione del server, conferendo a quest'ultimo tutte le caratteristiche necessarie per renderlo completo e funzionante.
Infatti, a livello di codice, *multithread.py* si occupa di creare un thread per ogni client che effettua la richiesta per poi eseguire il metodo *solve_request()* che permette la gestione e la risoluzione della richiesta richiamando le corrispondenti funzioni. Per questi motivi viene effettuata l'importazione del modulo *threading*.
Inoltre, il costrutto *try…except* è il responsabile della gestione dell'interruzione del server: se l'utente utilizza la combinazione di tasti '*Ctrl+C'*, il server verrà chiuso interrompendo l'esecuzione del ciclo *(while)* infinito (che permette al server di gestire più richieste) poiché viene richiamato il metodo *shutdown_server()*.

La completa gestione e realizzazione del server viene implementata nella classe *UDP_Socket_Server* del file *UDP_Socket_Server.py*
Dopo l'inizializzazione della porta e dell'indirizzo IP, queste vengono legate tramite il metodo *bind()* creando il socket UDP poiché la scelta della famiglia del socket è la SOCK_DGRAM.

Ne segue l'implementazione delle richieste attraverso le specifiche funzioni.
In particolare, la funzione *file_list()* e *get_file()* utilizzano il modulo *"os"* (che viene importato appositamente): questo modulo consente di lavorare con le funzionalità dipendenti dal sistema operativo in python, fornendo le funzioni per interagire con il sistema operativo.

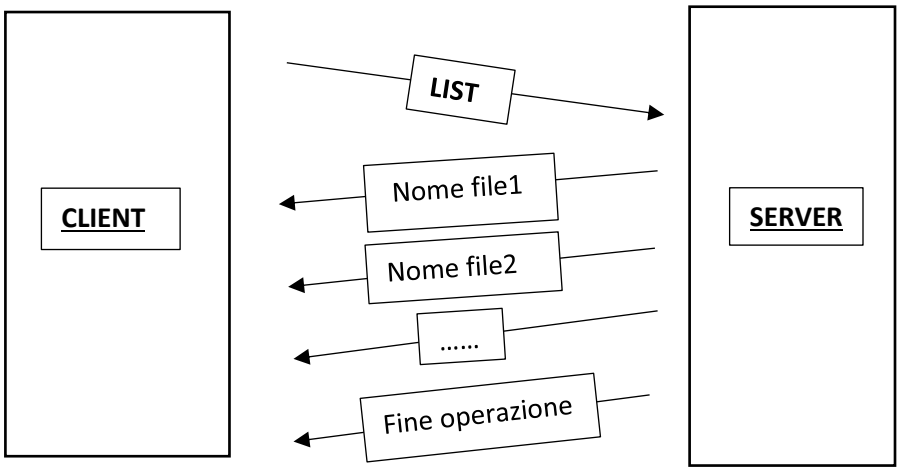
- La funzione *file_list()* invia al client la lista dei file presenti nella directory corrente.
- La funzione *get_file()* effettua il download *di un file presente nella directory corrente*, quindi controlla la sua esistenza e in caso di risposta positiva legge 4096 byte alla volta inviandoli al client che ha effettuato la specifica richiesta.
- La funzione *put_file()* si occupa di effettuare *l'upload di un file a patto che esso esista all'interno della directory corrente.* Quindi ne verifica la sua esistenza e utilizza il metodo *select.select()* per determinare quando il socket server è pronto a ricevere i dati di input o inviare i dati di output.
  Per questo motivo importiamo il modulo *"select".*

Infine la realizzazione del client viene implementata nella classe *UDP_Socket_Client* del file *UDP_Socket_Client.py.*
Dopo l'inizializzazione del socket UDP del client che interagirà col server, vengono implementate le possibili richieste che un client può effettuare attraverso l'implementazione delle corrispondenti funzioni.

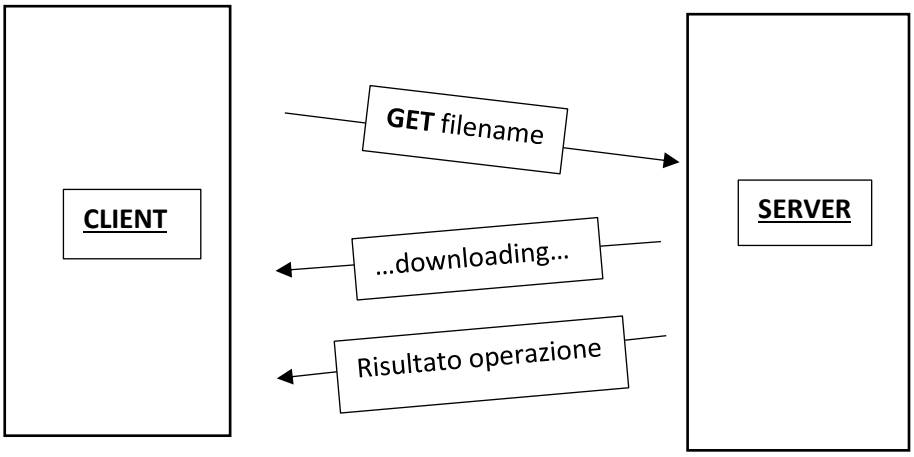
- La funzione *handle_request()* è il fulcro del client: mostra tutte le possibili richieste che può effettuare e lascia all'utente la possibilità di scelta di una di esse, per poi inviare tale messaggio di richiesta al server (inviando cioè un messaggio *list/get/put* al dipendere dalla scelta dell'utente) che la porterà a conclusione se non si verificano errori. Tale funzione si predispone anche a controllare la gestione della funzione *file_list()* e la gestione degli errori poiché in entrambi i casi il client riceverà un messaggio di risposta dal server.
  Infine si predispone a gestire le eccezioni e a chiudere forzatamente il socket client tramite il costrutto *try…except…finally* a seguito della combinazione di tasti Ctrl+C.
- La funzione *get_file()* permette al client di ricevere una risposta dal server che lo informa (al client) se il file di cui si vuole effettuare il download esiste o meno. Utilizza il metodo *select.select()* per determinare quando il socket client è pronto a ricevere i dati di input. Per questo motivo importiamo il modulo *"select".*
- La funzione *put_file()* permette al client di effettuare l'upload di un file esistente inviando un messaggio di richiesta al server. Nel caso di risposta positiva, ovvero nel caso in cui il file richiesto per l'upload esista, legge 4096 bytes alla volta inviandoli al server. Per verificare l'esistenza del file richiesto, viene importato il modulo *"os"* che permette di interagire con il sistema operativo.

SCHEMA DEI THREADS

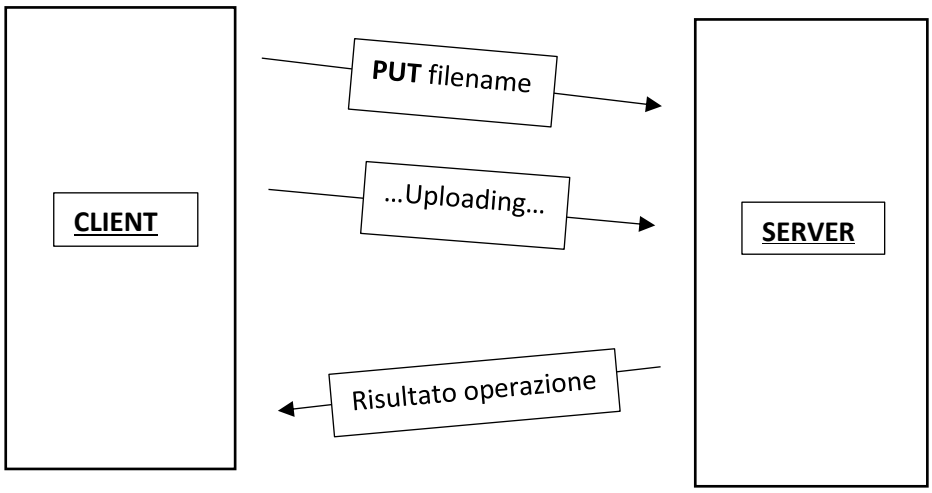
- **LIST**: visualizzazione sul client dei file disponibili sul server

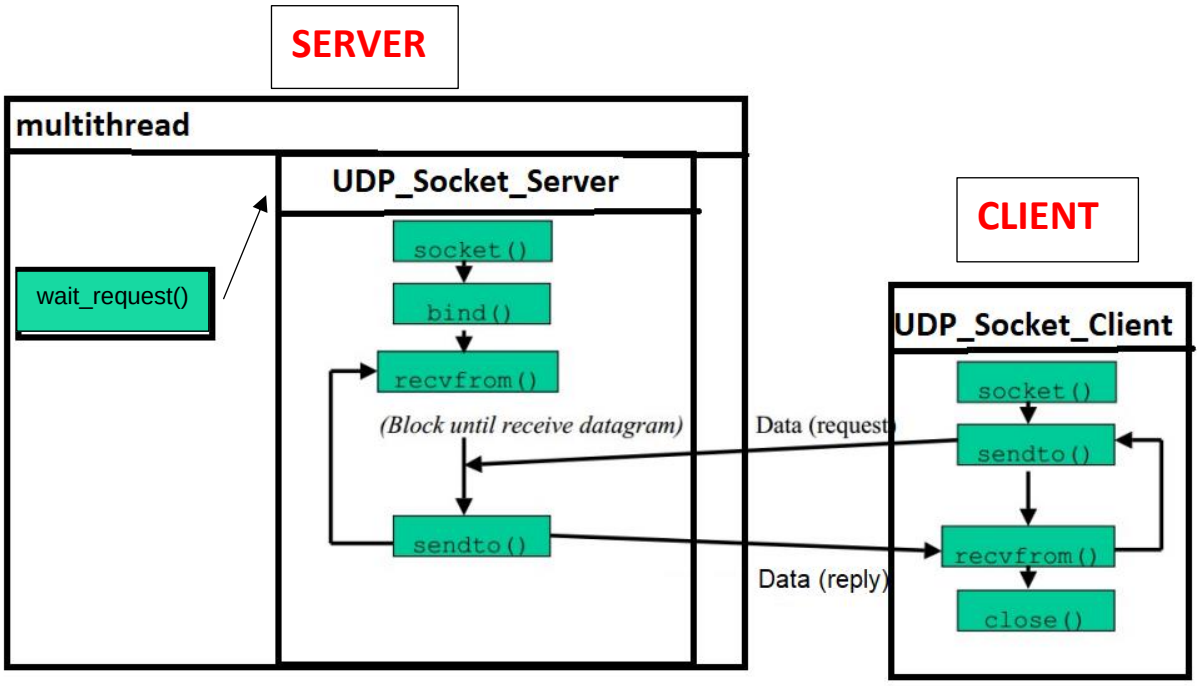


- **GET**: download di un file dal server



- **PUT**: upload di un file sul server





OSSERVAZIONI

La realizzazione del server, come anticipato, è basata sul multithreading. Di conseguenza, come si nota anche dallo schema dei thread, *multithread.py* importa *UDP_Socket_Sever*. In sostanza, in *multithread.py* si ha un'estensione della classe *UDP_Socket_Server* la quale, passando dalla funzione *wait_request(),* permette al server di rimanere aperto per le successive richieste dei client a meno che l'utente non ne richieda la chiusura.
Quindi per questi motivi e (come detto in precedenza) per la completa realizzazione del server, il server sarà composto da due moduli: *UDP_Socket_Server.py* e *multithread.py.*