RELAZIONE PROGETTO PROGRAMMAZIONE DI RETI

Traccia 2

Bedei Andrea andrea.bedei2@studio.unibo.it 0000978333

Bertuccioli Giacomo Leo giacomo.bertuccioli@studio.unibo.it 0000989426

Notaro Fabio fabio.notaro2@studio.unibo.it 0000978334

Sommario

INDICAZIONI ESECUZIONE CODICE

Di seguito si riporta un breve elenco puntato che descrive il corretto modo di eseguire il codice fornito:

1. Estrarre la cartella compressa allegata al documento
2. Aprire Spyder o qualunque IDE per Python
3. Da Spyder aprire i file server.py e client.py che si trovano nella cartella estratta
4. Mandare in esecuzione prima server.py e, solo dopo, eseguire client.py in un'altra console di Spyder

DESCRIZIONE DEL PROBLEMA

La traccia 2, denominata "Architettura client-server UDP per trasferimento file", prevede la realizzazione di un'applicazione con architettura client-server che, basandosi sul protocollo UDP, permetta un trasferimento di file tra client e server.

In particolare, le funzionalità offerte dal server sono:

- Listing → visualizzazione dei file disponibili sul server
- Getting → download di un file dal server al client
- Putting → upload di un file dal client al server

La comunicazione deve avvenire mediante un opportuno protocollo che preveda sia messaggi di comando (inviati dal client al server per richiedere i servizi citati sopra), che messaggi di risposta (inviati dal server al client per comunicargli l'esito di ciascuna operazione).

DESCRIZIONE GENERALE SULLE SCELTE DI PROGETTO

Durante la fase di analisi preliminare del progetto è emersa la prima scelta importante da intraprendere, riguardante la struttura del progetto stesso. Le due alternative possibili valutate sono state:

- Un numero molto ridotto di classi più corpose
- Molte piccole classi, ognuna con un compito specifico

Sebbene il secondo approccio offrisse modularità e maggiore chiarezza, si è scelto di adottare la strategia con poche classi, in modo da favorire la semplicità di sviluppo ed uniformarci alle strutture delle architetture che ci sono state presentate nel corso dei laboratori, le quali molto raramente si componevano di più di due classi (il client ed il server).

Il progetto si compone dunque di tre classi: il client, il server ed una classe di utilità.

La classe più semplice riguarda la classe di utilità response.py, la quale contiene la definizione di alcune costanti, come la dimensione dei buffer ed i possibili messaggi di risposta previsti dal protocollo adottato.

Infatti, come espressamente richiesto dalla consegna della traccia, la comunicazione tra client e server deve avvenire utilizzando un opportuno protocollo, in modo da rendere la comunicazione stessa più efficace.

La classe response.py, dunque, contiene la definizione di tutti i messaggi controllo di cui si compone il protocollo creato.

Per maggiori dettagli circa tali messaggi, si faccia riferimento alla tabella di seguito:

MESSAGGIO	MITTENTE	SCOPO
HELLO	Client	Instaurare la connessione con il server
OK	Client/Server	Comunicare alla controparte la correttezza del precedente messaggio ricevuto
FAIL	Server	Comunicare al client l'accadimento di un errore
DATA	Client/Server	Durante trasferimento file (get e put) serve a comunicare alla controparte che è in arrivo il contenuto effettivo del file
DONE	Client/Server	Durante trasferimento file (get e put) serve a comunicare alla controparte che il trasferimento del file è da ritenersi concluso

Per quanto riguarda le classi di client e server, esse rappresentano il cuore del progetto.

Di seguito si riporta un semplice diagramma UML che riporta per entrambe queste componenti le funzioni implementate:

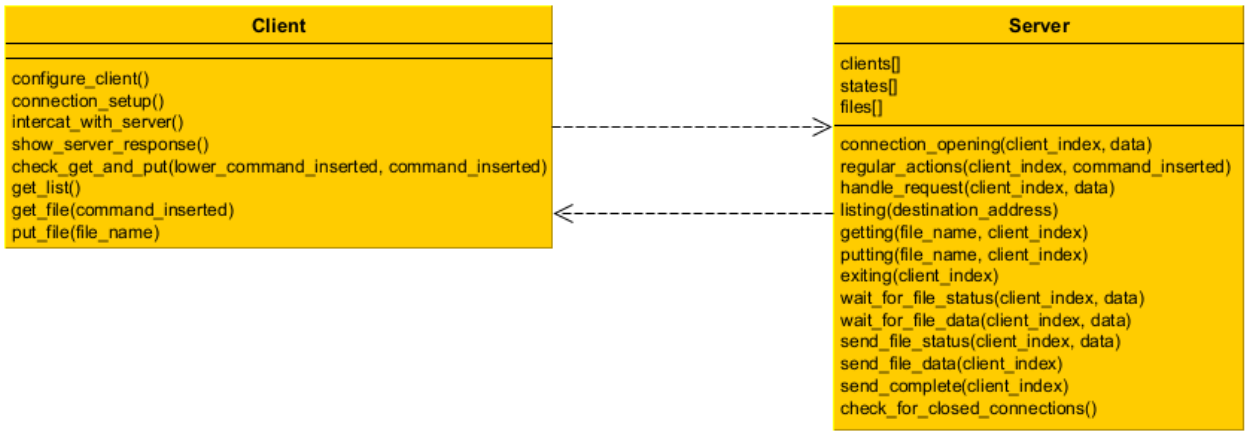
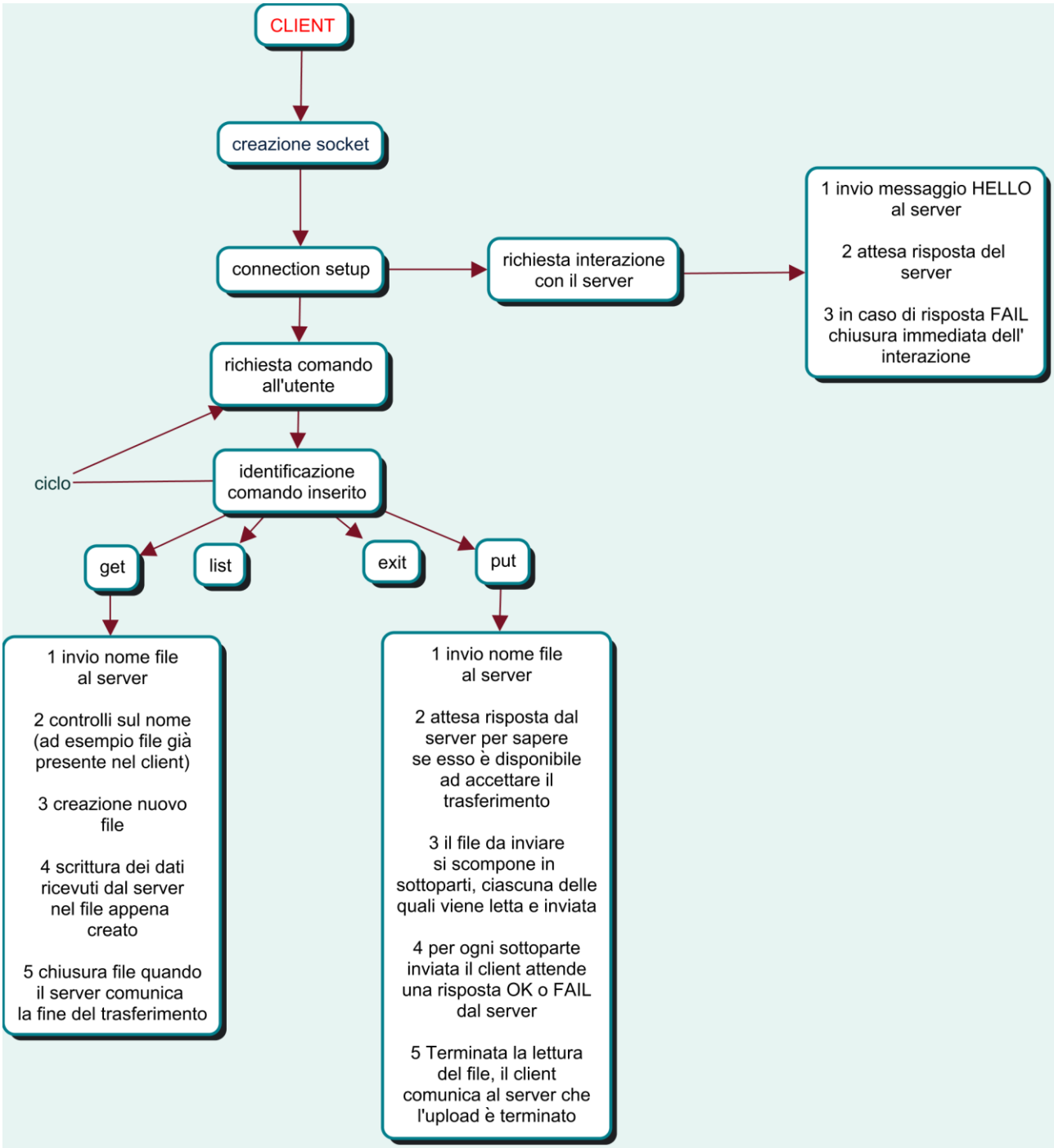


Figura 1 UML raffigurante le classi create e le loro funzioni: entrambe le componenti son descritte più dettagliatamente di seguito

Nello schema concettuale di seguito è riportato il funzionamento dettagliato del client:



Per quanto concerne la struttura del server, prima di descriverla è bene soffermarsi su un aspetto di primaria importanza: gli stati che esso può assumere nel corso del tempo e gli eventi che ne causano un cambiamento.

Il funzionamento del server, infatti, è dipendente dallo stato che il server stesso assume durante l'avanzamento della connessione con ciascun client.

Al fine di individuare gli stati possibili, è stato necessario svolgere un'approfondita attività di analisi.

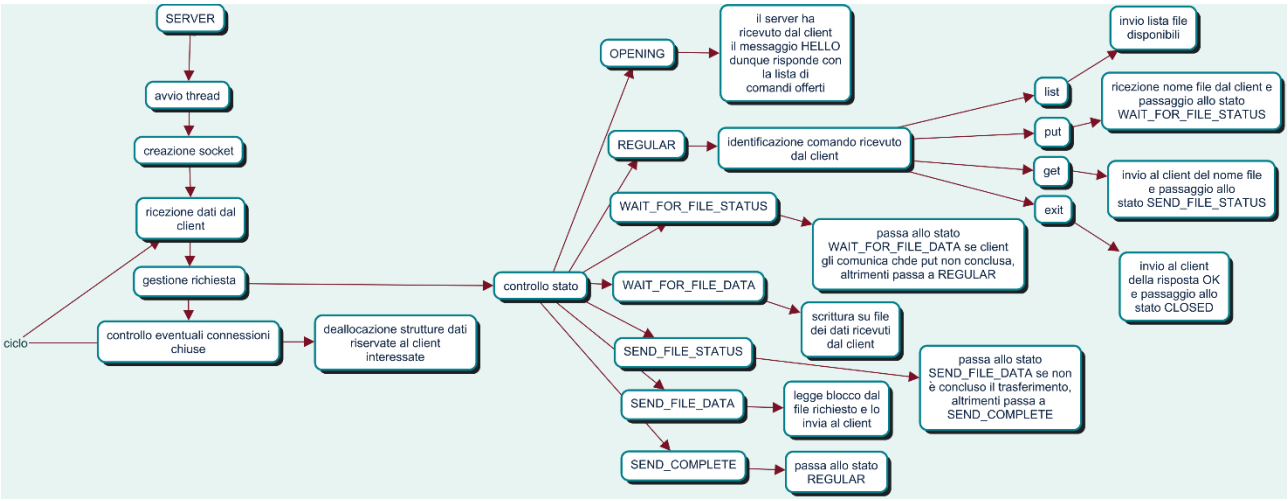
Di seguito sono riportati i possibili stati che il server può assumere e per ciascuno una breve descrizione:

- OPENING → stato iniziale assunto dal server al momento dell'apertura della connessione con un client → prevede di passare allo stato REGULAR
- REGULAR → il server è in attesa di ricevere il comando dal client
- WAIT_FOR_FILE_STATUS → è lo stato assunto dal server in seguito alla ricezione del comando put → se il client gli comunica che sono in arrivo altri dati per il file interessato dalla put passa nello stato WAIT_FOR_FILE_DATA, altrimenti passa allo stato REGULAR se il client gli comunica la terminazione del trasferimento
- WAIT_FOR_FILE_DATA → è lo stato assunto durante l'effettivo svolgimento della put, in quanto impone al server di interpretare i dati in arrivo dal client come contenuto da scrivere nel file
- SEND_FILE_STATUS → è lo stato assunto dal server in seguito alla ricezione del comando get → controlla se ha spedito tutto il contenuto del file richiesto al client → in caso negativo passa allo stato SEND_COMPLETE, altrimenti c'è ancora del contenuto da trasmettere, dunque passa allo stato SEND_FILE_DATA
- SEND_FILE_DATA → è lo stato assunto dal server durante l'effettivo svolgimento della get, in quanto impone al server di leggere il file richiesto dalla get stessa e trasmettere il suo contenuto al client
- SEND_COMPLETE → è lo stato assunto dal server quando, a seguito di una get, non c'è altro contenuto da trasmettere per il file richiesto, dunque si può passare allo stato REGULAR
- CLOSED → è lo stato assunto dal server in caso di comando exit o di accadimento errori

Altra peculiarità riguardante il server risiede nel fatto che questo, in ottica di gestione parallela di connessioni con client diversi, mantiene in opportune strutture dati

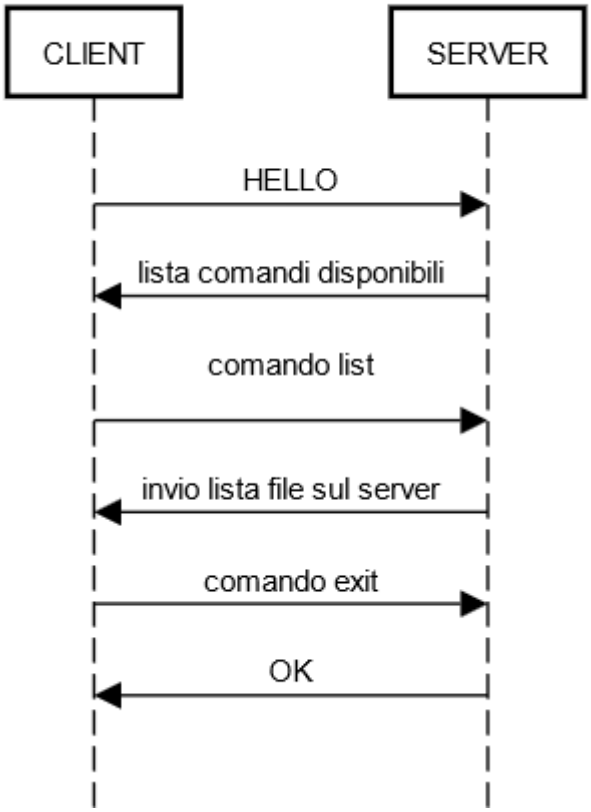
- L’indirizzo ip di ciascun client connesso
- Lo stato del server con ciascun client connesso
- Il file in elaborazione (sia per put che per get) richiesto da ciascun client connesso

Per meglio comprendere il funzionamento del server, si consideri la mappa concettuale di seguito:

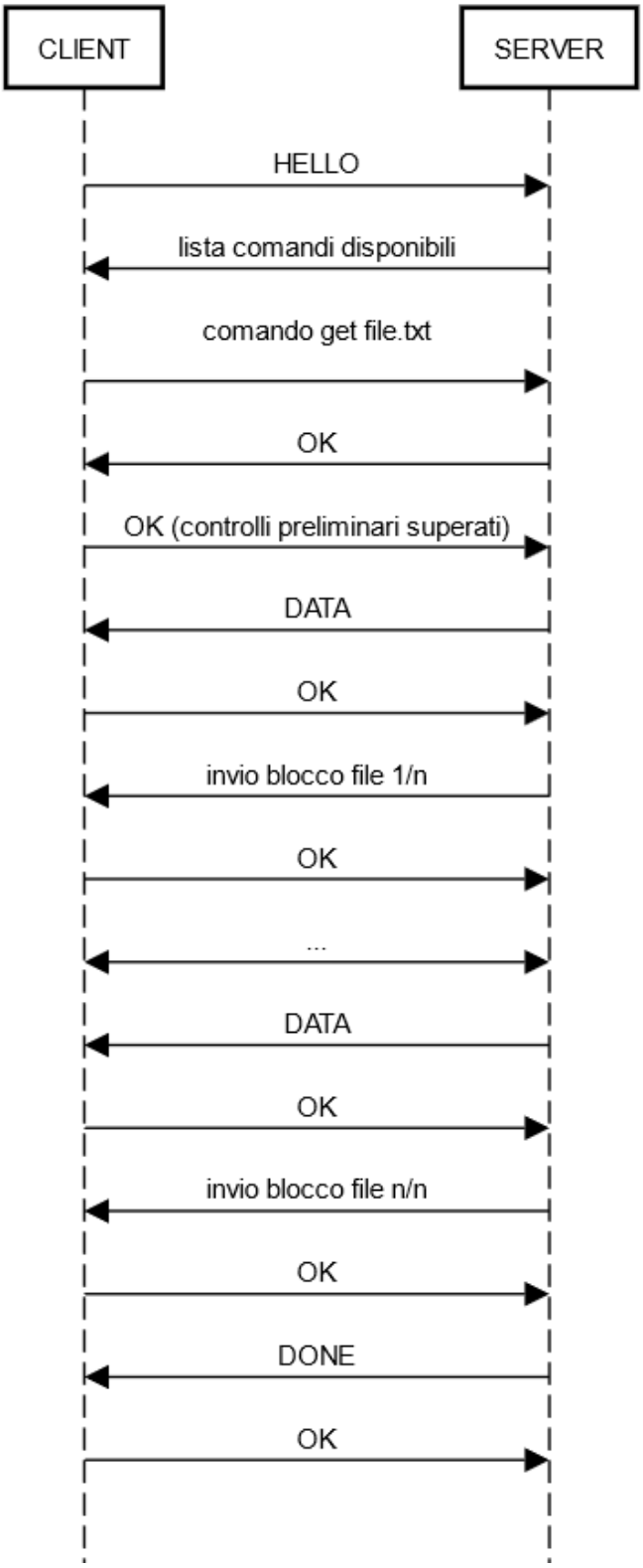


Per meglio comprendere la successione dei messaggi di controllo inviati tra client e server, si faccia riferimento ai diagrammi di sequenza riportati di seguito, che mostrano cosa accade con i comandi list, exit e get:

RICHIESTA LIST E EXIT



RICHIESTA GET



Per la scrittura del codice ci si è basati sui codici forniti a lezione (telnet, UDP client, UDP server…), arricchiti da materiale trovato online, comunque rielaborato e adattato al progetto.

DESCRIZIONE DELLE STRUTTURE DATI UTILIZZATE

Le principali strutture dati utilizzate sono state:

- Liste (principalmente per clients, stati delle connessioni e file aperti in download o upload)
- Tuple (principalmente per i socket)
- Modulo (creato da noi) con all'interno gli stati di risposta del protocollo

SCHEMA GENERALE DEI THREAD ATTIVI

Nonostante non fosse esplicitamente richiesto dal testo della traccia, si è deciso di attivare un thread all'avvio del server, il quale permette ai client di accedere in parallelo ai servizi offerti.

FUNZIONALITA' EXTRA

Oltre alle funzionalità richieste esplicitamente dalla consegna della traccia, si è pensato di aggiungere ulteriori funzionalità, ritenute di grande utilità. Le principali sono:

- Invio e ricezione di file di grandi dimensioni
- Funzione exit che termina la connessione tra un client ed il server (lasciando comunque attivo il server per altre connessioni con altri client)
- Attento controllo per quanto concerne errori di inserimento comandi (per ulteriori dettagli si faccia riferimento alla sezione di seguito)
- Utilizzo di un timer a fini statistici per misurare il tempo di trasferimento dei file
- Threading

TEST EFFETTUATI

Per garantire il corretto funzionamento dell'applicazione, si è deciso di investire molto tempo di sviluppo per un'attenta fase di testing.

La fase di inserimento dei comandi del client è stata individuata come quella più problematica, in cui possono accadere gli errori più gravi. Per questo motivo si è scelto di analizzare e rintracciare tutti i possibili errori di inserimento, in modo da prevedere una corretta gestione degli stessi.

Più dettagliatamente, sono stati gestiti i seguenti casi:

- comando non presente
- comando presente ma con troppi parametri ("list qualcosa", "exit ciao", "get file.txt prova")
- comando get con file non presente sul server
- comando get con file già presente sul client
- comando put con file già presente sul server
- comando put con file non presente nel client
- comando get senza file
- comando put senza file
- comando corretto ma digitato in maiuscolo
- comando get con file illegale (presente nel server ma in una cartella non permessa)
- comando put con file illegale (presente nel client ma in una cartella non permessa in quanto diversa da quella corrente)

Un'altra fase che ha richiesto un attento controllo degli errori riguarda il trasferimento di file di grandi dimensioni. Come spiegato nelle sezioni precedenti, la problematica dei file di grandi dimensioni è stata gestita suddividendo il file stesso in pacchetti più piccoli, ciascuno inviato separatamente.

Ulteriore test effettuato ha riguardato il controllo del corretto funzionamento dell'applicazione instaurando una reale connessione tra client e server su due host in rete.

Per effettuare tale verifica si è pensato di usare i computer di due componenti del gruppo, facendo svolgere ad uno il ruolo di server e all'altro quello di client.

Ovviamente è stato necessario cambiare i parametri del socket inserendovi, al posto di localhost, il reale indirizzo IP pubblico del computer server. E' stato inoltre dovuto abilitare il port forwarding sul router collegato al computer server.

Impostati questi parametri ed eseguendo i due codici, l'instaurazione della connessione è avvenuta con successo:

```
IPython 7.29.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/andre/Desktop/Reti/UDP_SERVER/Server/server2.py',
wdir='C:/Users/andre/Desktop/Reti/UDP_SERVER/Server')

 starting up on 192.168.178.126 port 10002
In ascolto
received 22 bytes from ('151.41.141.145', 64764)
151.41.141.145: Client connesso
In ascolto
```

Figura 2 Instaurazione della connessione lato server

```
In [1]: runfile('C:/Users/fabio/Desktop/client.py', wdir='C:/Users/fabio/Desktop')
Socket creato


Benvenuto sul Server come posso rendermi utile?

Opzioni disponibili:

    list ->         Restituisce la lista dei nomi dei file disponibili.
    get <NomeFile> -> Restituisce il file se disponibile.
    put <NomeFile> -> Carica il file se disponibile.
    exit ->         Esce


Inserire comando: list
```

Figura 3 Instaurazione della connessione lato client

Infine, si è deciso di utilizzare uno strumento importantissimo visto a lezione come Wireshark per controllare la correttezza del flusso di pacchetti tra client e server.

Come spiegato dal docente, tale applicativo ci ha permesso di studiare ed analizzare più approfonditamente ciò che avviene durante la comunicazione.

Si riportano di seguito alcuni screenshot ritenuti di particolare rilevanza:

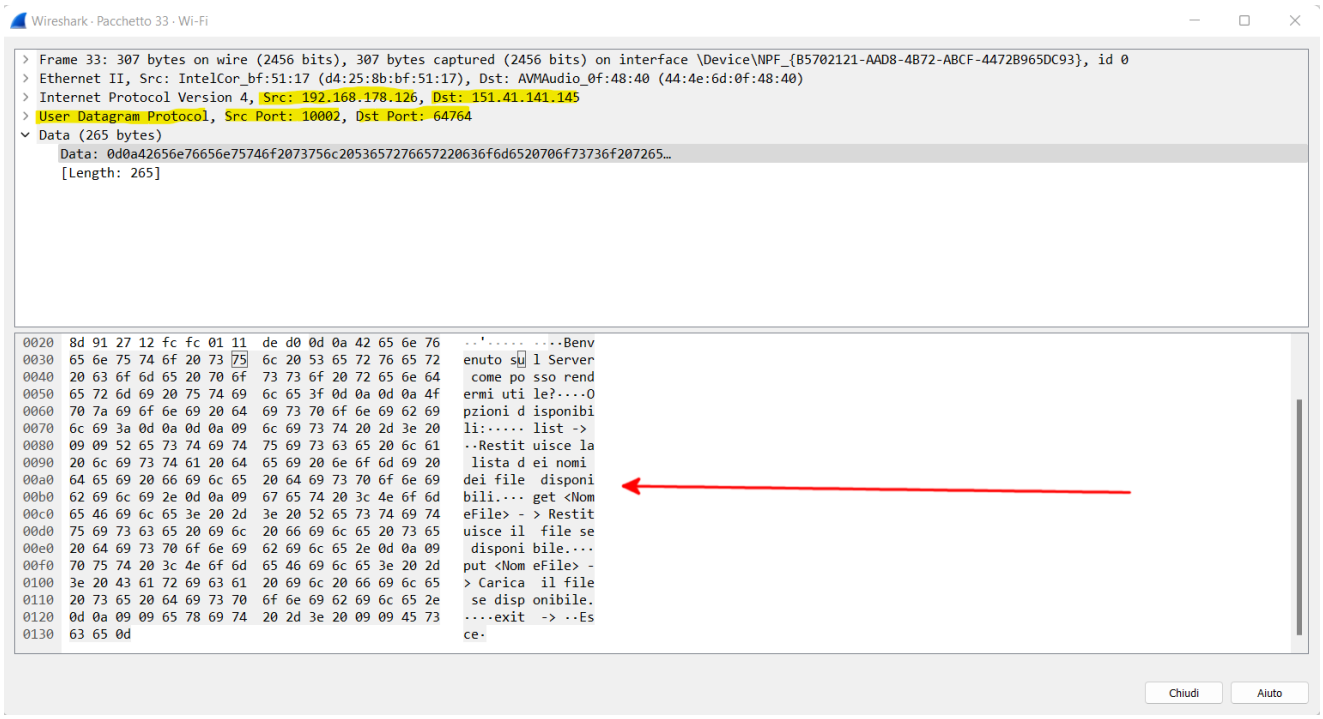


Figura 4 Pacchetto contenente il messaggio di benvenuto spedito dal server. Si notino i parametri evidenziati

Di seguito Wireshark riporta, in ordine cronologico inverso, gli eventi che si verificano durante il trasferimento di un file dal client al server (put):

375	92.961490	192.168.178.126	151.41.141.145	UDP	60 10002 → 64764 Len=18
374	92.959216	151.41.141.145	192.168.178.126	UDP	102 64764 → 10002 Len=60
372	92.923165	192.168.178.126	151.41.141.145	UDP	66 10002 → 64764 Len=24
371	92.921437	151.41.141.145	192.168.178.126	UDP	55 64764 → 10002 Len=13

- l'ultima riga rappresenta l'invio del pacchetto contenente il comando put dal client al server
- la riga immediatamente sopra è il pacchetto inviato dal server al client per comunicargli che è in attesa e pronto a ricevere il file (OK)
- la seconda riga invece rappresenta il pacchetto contenente il contenuto vero e proprio del file
- infine la prima rappresenta il pacchetto del server di corretto trasferimento (DONE)

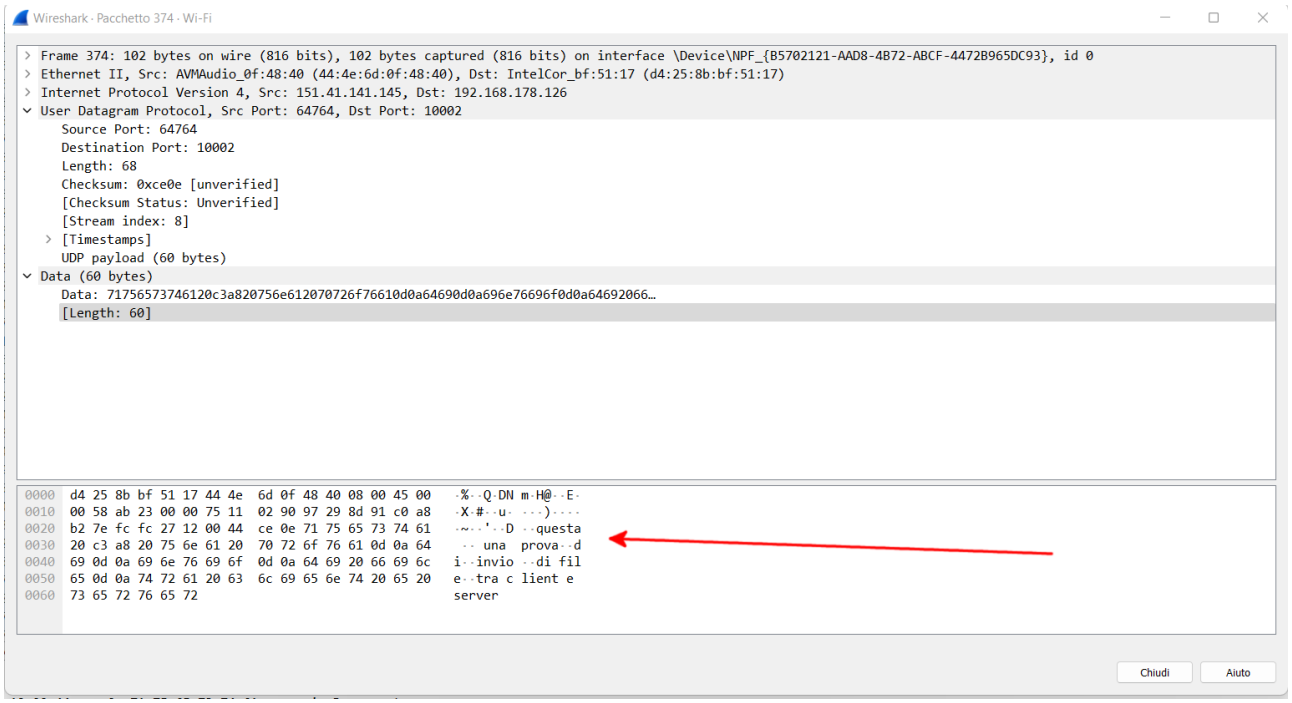


Figura 5 L'ultimo pacchetto inviato dal client al server è proprio il contenuto del file trasferito.

AUTOVALUTAZIONE E DIFFICOLTA' INCONTRATE

Le principali difficoltà incontrate sono state:

- corretta sincronizzazione tra client e server
- possibilità di inviare e ricevere file di grandi dimensioni
- permettere a più client di eseguire operazioni sul server contemporaneamente (connessioni in parallelo).

Nonostante queste problematiche, siamo molto soddisfatti progetto assegnatoci, in quanto ci ha permesso di approfondire importanti nozioni incontrate a lezione.

Siamo altresì soddisfatti del lavoro svolto in quanto, oltre alle specifiche base richieste dal testo della traccia, siamo stati in grado di inserire funzionalità non richieste e dunque approfondire aspetti anche molto diversi affrontati durante il corso (threading, tempo di esecuzione delle funzioni, manipolazione di stringhe, utilizzo dell'applicativo Wireshark…).

I vari membri del gruppo sono stati in grado di collaborare attivamente ed in maniera equilibrata, in modo anche da mettere in risalto le proprie capacità e verificare gli apprendimenti previsti dal corso affrontato.