Traccia 2

Architettura client-server UDP per trasferimento file

Luca Lucioli luca.lucioli@studio.unibo.it 0000988329

Scelte progettuali **2**

Buffer 2

Timeout 2

Comando PUT 2

Lock 3

Altre 3

Strutture dati **3**

Server 3

Client 3

Diagramma UML **4**

Threads **4**

Server 4

Considerazioni finali **5**

Scelte progettuali

Buffer

La dimensione del buffer del socket deve essere scelta sulla base della velocità con cui esso viene riempito e conseguentemente svuotato dal comando **recvfrom**. Inoltre è necessario assicurarsi che il buffer sia grande almeno quanto la dimensione dei pacchetti che si desidera inviare (nei socket DGRAM l'eccedenza di bytes viene silenziosamente scartata). In questa applicazione i bytes che vengono letti di volta in volta dai file coincidono proprio con la dimensione del buffer.

Timeout

Tutti i socket in uso nelle due componenti sono creati in blocking mode (di default), quindi l'esecuzione si blocca finchè non viene completata l'operazione sul socket (ad esempio ricezione di dati).

Inevitabilmente, quindi, al termine delle operazioni di **put** e **get**, quando il mittente ha letto tutto il file e lo ha inviato al ricevente, quest'ultimo rimarrebbe invano ad aspettare un'ulteriore datagram che però non arriverebbe, essendo la trasmissione effettivamente conclusa. La soluzione scelta in questo progetto è il settaggio di un timeout sul socket destinatario, trascorso il quale la trasmissione può considerarsi terminata e il file arrivato a destinazione.

Comando PUT

Come è possibile vedere nello schema dei threads, il server assegna ad ogni client un socket e un rispettivo thread che gestisce tutte le comunicazioni tra le due parti. Nel caso in cui un client esegua come primo comando il **put**, il server provvederà a creare quindi un nuovo socket, di cui però il client non conosce la porta. Dal momento che l'operazione di upload prevede che sia il server a ricevere dati dal client, quest ultimo ha necessità di sapere su quale porta deve spedire i datagrammi. In questa architettura si è scelto di introdurre un messaggio di "inizializzazione" che il client attende di ricevere dal server per poter iniziare la trasmissione. Grazie a questo messaggio il server si rivela al client con la porta che dovrà utilizzare per la trasmissione del file e per le successive comunicazioni.

Lock

Per assicurare la sincronizzazione tra i thread e un corretto accesso alle risorse condivise è stato utilizzato un Lock: un oggetto che ha due stati “locked” e “unlocked”. Il suo stato può essere modificato dai due metodi “acquire()” (il lock passa da unlocked a locked”) e “release()” (passaggio opposto). Nello specifico l’oggetto lock è stato impiegato per proteggere l’accesso alle operazioni di **get**, **put**, **list**, le quali, operando attivamente o indirettamente sui file, non devono essere eseguite simultaneamente (si pensi ad uno scenario in cui due client eseguono rispettivamente i comandi **put** e **get** sullo stesso file in contemporanea).

Altre

Nella fase di gestione dei nuovi client il server crea nuovi socket con porta individuata dalla somma tra il numero di porta del server e un numero casuale tra 1 e 100. Ciò implica che questa architettura può gestire potenzialmente 100 client contemporaneamente. Per permettere a più client di collegarsi al server si potrebbe optare per un range di porte più esteso o magari per un sistema di “riciclo” delle porte non più in uso: nella versione corrente le porte utilizzate non possono essere assegnate nuovamente (il dizionario porta-socket non viene svuotato alla chiusura della connessione di un client).

Strutture dati

Server

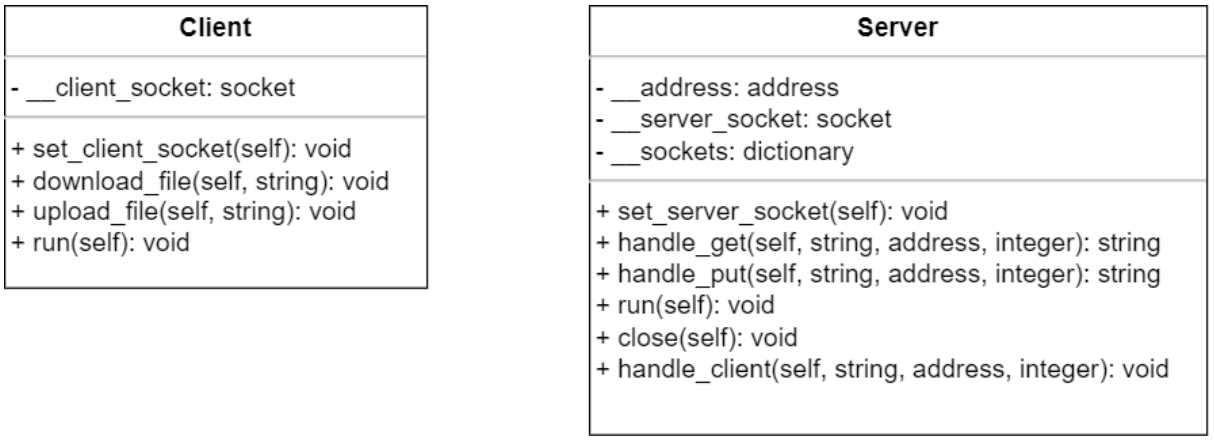
Il server è rappresentato da una classe che ha come attributi:

- **indirizzo** pair host, port
- **socket** principale
- **socket associati ai singoli client** dizionario le cui key sono le porte effimere e le value sono i proprio i socket

Client

Le funzionalità del client sono incapsulate in una classe che ha come unico attributo il proprio **socket**

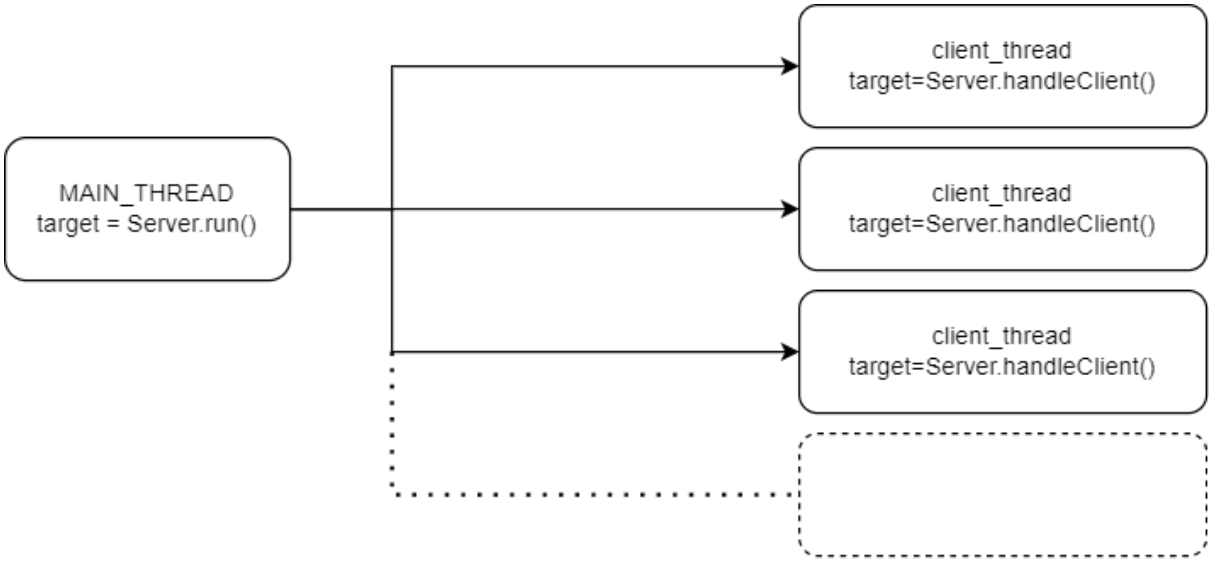
Diagramma UML



Threads

Server

Nel server abbiamo un thread che esegue il loop principale e tanti thread quanti sono i client che inviano un comando al server.



Considerazioni finali

Per lo scopo dell'applicazione (inviare e ricevere file), l'architettura progettata non risulta essere particolarmente adeguata, soprattutto per la scelta di UDP come protocollo di trasporto. La sua politica "best effort" implica infatti la possibilità di perdere pacchetti durante la trasmissione o di non riceverli in ordine, due scenari che poco si adattano ad impieghi non loss-tolerance come questo.
Una versione più realistica dell'applicazione sviluppata dovrebbe perciò utilizzare TCP come protocollo di trasporto, il quale garantisce la consegna dei datagram (in ordine).