Progetto del Corso di Programmazione di Reti

Traccia 2

Architettura client-server UDP per trasferimento file

Giogio Fantilli

giorgio.fantilli@studio.unibo.it

0000988142

Indice

Scelte progettuali 2
Buffer 2
Gestione di piú client 2
Lock delle risorse condivise 3
Interrupt 3
Connessioni 4
Socket timeout 4
Strutture dati 5
Server 5
UML 5
Schema dei thread 6

Scelte progettuali

Buffer

È necessario assicurarsi che i buffer di ricezione del server siano grandi almeno quanto quelli di invio del client, al fine di evitare che nella comunicazione tra i due, essendo di tipo UDP, i byte inviati in eccedenza vengano silenziosamente scartati.

Ho perciò deciso di omologare le dimensioni dei buffer di invio e ricezione di client e server; inoltre, i byte letti/scritti dai due in fase di lettura/scrittura di un file su disco, corrispondo esattamente alla dimensione dei buffer di trasferimento e ricezione degli stessi, così da consentirne lo scambio in maniera rapida ed efficiente, senza perdita di dati.

Gestione di piú client

L'applicazione, o per meglio dire il server, consente la connessione simultanea di piú client. Questo è possibile grazie all'utilizzo di piú thread con altrettante connessioni: il server esegue un main thread, avente un socket UDP sempre aperto sulla medesima coppia ip:porta, che si occupa di accettare i vari client che richiedono una connessione.

Alla connessione di un nuovo client il main thread crea un nuovo socket, di medesimo ip ma di porta differente dall'originale, calcolata randomicamente, ed avvia un nuovo thread. A quest'ultimo verrà assegnato il nuovo socket e l'indirizzo del client appena collegato; sarà poi tale thread a fornire i vari servizi richiesti e ad occuparsi della relativa connessione. Il client viene a conoscenza della nuova porta grazie all'invio, da parte del thread appena creato, dell'iniziale messaggio di ben venuto.

Socket e porte utilizzate vengono memorizzate all'interno di uno specifico dizionario condiviso dai thread, evitando così, al set up di un nuovo socket, di riutilizzare erroneamente porte già in uso. Ogni thread, alla sua terminazione, elimina dal dizionario l'entry relativa alla sua connessione, consentendo il riutilizzo della porta associata.

Come scelta implementativa il server può gestire un massimo di 100 connessioni simultanee con altrettanti client.

Lock delle risorse condivise

Poichè il server è di fatto un'applicazione multithread che utilizza risorse condivise (il dizionario dei socket in uso) ed esegue operazioni di lettura del file system e di lettura/scrittura su file, è necessario che tali operazioni non vengano eseguite simultaneamente dai vari thread, altrimenti si avrebbe un serio rischio di inconsistenza dei dati o di corruzione.

La sincronizzazione dei thread è possibile attraverso un oggetto Lock, il quale mette a disposizione i metodi *acquire()* e *release()*, da chiamare rispettivamente subito prima e subito dopo un blocco di codice che esegue istruzioni critiche, offrendo la possibilità di accedere alle risorse condivise in mutua esclusione (quando un thread esegue la porzione di codice critita, gli altri non possono eseguire le proprie porzioni, venendo bloccati dalla *acquire()* in attesa del loro turno; al termine della sezione interessata la *release()* sblocca uno degli altri thread precedentemente bloccati, e così via).

Ho deciso di utilizzare due variabili lock differenti, una per proteggere le operazioni di accesso al dizionario delle porte in uso, e un'altra per proteggere le operazioni di scrittura o lettura su file. Questo poichè le due sono risorse completamente differenti e non avrebbe alcuna utilità bloccare tutti i thread indipendentemente da quale stiano accedendo (ma anzi sarebbe un motivo di lentezza del codice).

Interrupt

Il main thread del server rimane sempre in esecuzione, eseguendo in un ciclo infinito il codice di accettazione di un nuovo client. La sua terminazione è possibile premendo da tastiera la combinazione di tasti *Ctrl+C*: viene eseguita una funzione di terminazione che chiude il socket di accettazione di nuovi client e termina il processo.

I client thread precedentemente avviati non vengono killati e diventano così orfani, ma ancora operativi: chiuderanno il relativo socket e termineranno quando il client associato invierà il predisposto comando di terminazione.

Anche per i client è possibile terminare il processo premendo da tastiera la combinazione di tasti *Ctrl+C*: viene eseguita una funzione di terminazione che chiude il socket e termina il processo.

Connessioni

Come detto in precedenza il server utilizza due connessioni per comunicare con il client: la prima, che rimane sempre aperta per tutta la vita del main thread, con la quale vengono accettati i nuovi client che si connettono, la seconda invece è quella che utilizza il client thread, sulla quale avviene poi l'effettivo scambio di pacchetti.

Quest'ultima quindi è una comunicazione bidirezionale tra le due entità in gioco. Vi transitano i comandi da client a server per la richiesta di un servizio, il messaggio di risposta del server che notifica se si è verificato o meno un errore nell'offrire il servizio scelto (ad esempio il client ha indicato di scaricare dal server un file che non esiste), ed i dati del file letto, o in un verso o in un altro (nei casi in cui il client richieda di scaricare o caricare un file sul o dal server). Tale comunicazione viene chiusa dal thread al momento della disconnessione del client.

Il client quindi è praticamente all'oscuro di questo doppio socket del server e del trasferimento della connessione da uno all'altro, utilizzando durante tutta la sua esecuzione uno e un unico socket.

Socket timeout

In fase di trasferimento di un file, l'entità della comunicazione che riceve dal socket i relativi byte, per poi andarli a scrivere in un file locale, riesce ad accorgersi della terminazione della trasmissione, cioè dell'esaurimento dei dati del file , attraverso un timeout sul socket.

La ricezione di dati attraverso il metodo *recvfrom()* sul socket è infatti bloccante, quindi, nel caso precedentemente descritto in cui il mittente dei byte ha terminato la trasmissione di tutto il file, il ricevente non avrebbe nessun modo di accorgersene, e rimarrebbe bloccato in attesa di nuovi dati, che però non arriverebbero mai (non è possibile inviare un comando di terminazione poichè non c'è modo di distinguerlo dal vero e proprio contenuto del file da riceve).

Per risolvere il problema ho quindi optato per il set di un timeout sul socket, che viene poi eliminato una volta che l'intera operazioe termina: se il ricevente non riceve dal socket dei dati entro 2 secondi dalla chiamata della *recvfrom()*, quest'ultima lancia un'eccezione (se in due secondi non si riceve nulla significa che il mittente ha terminato l'invio di tutto il file). Quest'ultima viene poi gestita con l'uscita dal ciclo di ricezione e la successiva chiusura del file scritto su disco.

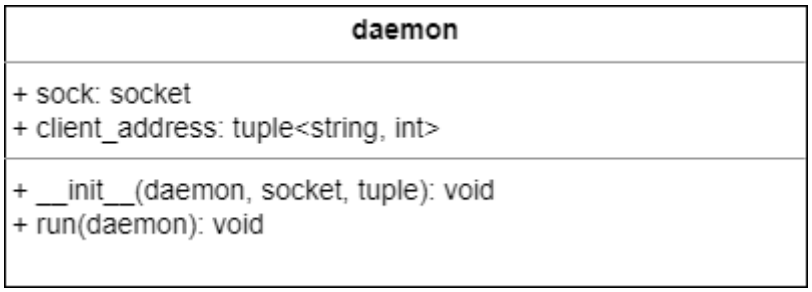
Strutture dati

Server

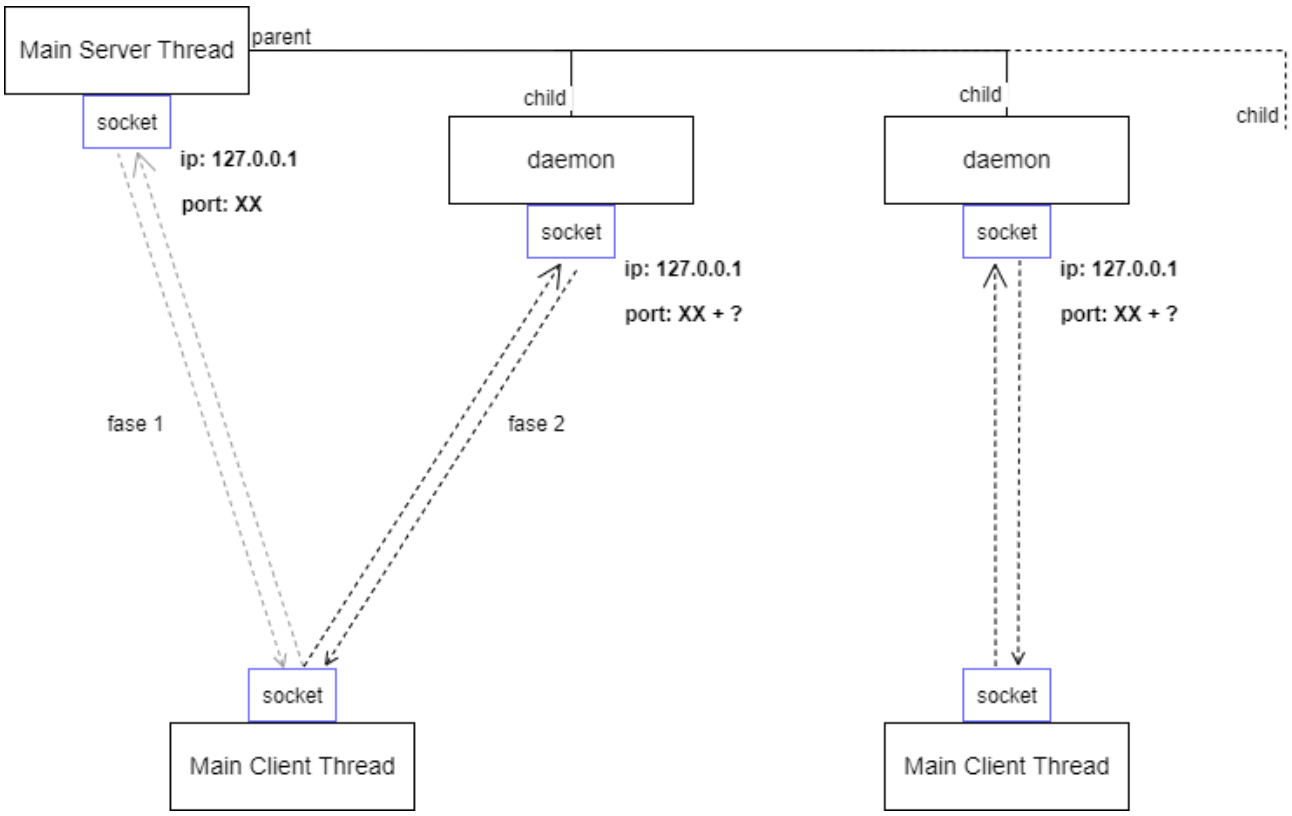
- Il thread che viene creato dal processo principale, con lo scopo di gestire la connessione con uno specifico client, è definito da una specifica classe chiamata *daemon*.
  Essa estende la classe *Thred* (libreria *threading*), ed implementa il relativo metodo *run()*, il quale avvia un processo figlio che esegue le routin di gestione dei vari comandi inviati dal client attraverso il socket.
  Tale socket, insieme al l'indirizzo del client (ip:porta), corrispondono agli attributi della classe *daemon*, inizializzatti nel costruttore.

- L'applicazione mantiene le porte attualmente in uso dai vari client thread all'interno di un dizionario. I vari numeri di porta vengono indicizzati attraverso il relativo socket.
  I thread avviati condividono tra loro, e con il main thread, tale dizionario: quest'ultimo viene dichiarato nel processo padre mentre i figli vi accedono come variabile globale.
  Tale accesso condiviso, come detto prima, viene protetto da un relativo lock.
  Lo scopo di tale dizionario è duplice:
  - Il main thread, alla creazione del socket per un nuovo processo figlio, controlla se il relativo numero di porta, calcolato randomicamente, sia già in uso o meno.
    In caso affermativo procede al ricacolo della porta finchè non ne trova una libera; se il dizionario è pieno rigetta la nuova connessione.
  - Il thread che serve i client, al momento di chiusura della connesione, elimina dal dizionario globale l'entry che ha per chiave il suo socket, permettendo così il riutilizzo della porta che stava precedentemente usando.

UML



Schema dei thread



Note

L'applicazione riesce a fuzonare correttamente poichè client e server sono rappresentati da due processi che vengono eseguiti sulla stessa macchina, perciò i datagrammi che vengono scambiati tra i due non viaggiano realmente nella rete.

In condizioni reali infatti, per via della scelta di utilizzare UDP come protocollo di trasporto, sarebbe probabile incappare in corruzione o perdita di parte dei dati inviati tramite il canale, e quindi degli stessi file caricati o scaricati dal server, rendendoli sostanzialmente inutili. UDP infatti è un protocollo connection-less: non garantisce il recapito di tutti i datagrammi inviati e non garantisce nemmeno il loro arrivo in ordine.

Queste garanzie vengono invece offere dal protocollo TCP, che sarebbe risultato notevolmente più adatto ad a questo tipo di applicazione.