Programmazione Distribuita I

Test di programmazione socket, 21 luglio 2017 – Tempo: 2 ore 10 minuti

Il materiale per l'esame si trova nella directory "`exam_dp_jul2017`" all'interno della propria home directory, che verrà chiamata `$ROOT` nel seguito.
Per comodità la directory contiene già uno scheletro dei file C che si devono scrivere (nella directory "`$ROOT/source`"). E' **FORTEMENTE RACCOMANDATO** che il codice venga scritto inserendolo all'interno di questi file **senza spostarli di cartella** e che questo stesso venga letto attentamente e completamente prima di iniziare il lavoro!
L'esame consiste nello scrivere un server e un client che scambiano files rappresentati tramite un formato simile a quello sviluppato nell'esercizio di laboratorio 2.3 (trasferimento file), ma più semplice. In maggior dettaglio, quando una connessione viene stabilita, il client **invia** un file al server usando il seguente formato:

B1	B2	B3	B4	Contenuto file……

dove i bytes B1 B2 B3 B4 nella figura sono un numero intero senza segno su 32 bit in network byte order che rappresenta il numero di bytes del file. Come si può vedere dalla figura precedente, i bytes del file seguono questi 4 bytes. Mentre riceve il file, il server effettua una trasformazione sui bytes del file e spedisce il file trasformato indietro al client, usando la stessa connessione e lo stesso formato dei dati. Nel frattempo, il client riceve il file trasformato. Quando il file trasformato è stato trasferito completamente, la connessione è chiusa in modo ordinato. Al fine di massimizzare le prestazioni e consentire la gestione di files grandi, il client inizia a ricevere il file trasformato immediatamente, invece di iniziare la ricezione dopo aver finito di inviare l'interno file. Allo stesso modo, al server è richiesto di bufferizzare non più di 10000 bytes (i.e., non può leggere più di 10000 bytes senza spedire qualcosa indietro al client).

Parte 1 (obbligatoria per passare l'esame, max 6 punti)
Creare un server che implementa il protocollo descritto sopra. Naturalmente, è possibile sfruttare la propria soluzione del laboratorio 2.3 a questo scopo. Il server deve ascoltare sulla porta specificata (come numero decimale) come primo parametro sulla linea di comando, su tutte le interfacce disponibili. Il secondo argomento della linea di comando, invece, è una stringa ASCII che specifica la trasformazione da applicare al file ricevuto: ogni carattere del file ricevuto che corrisponde ad uno dei caratteri di questa stringa deve essere rimpiazzato dal carattere ASCII asterisco (*), mentre tutti gli altri bytes devono essere lasciati inalterati.
Il/i files C del programma server devono essere tutti scritti dentro la directory `$ROOT/source/server1`, ad eccezione dei file comuni (per esempio quelli dello Stevens) che si vogliono riusare negli altri programmi da sviluppare per questo esame, e che possono essere messi nella directory `$ROOT/source` (si ricorda che per includere questi files nei propri sorgenti è necessario specificare il percorso "`..`").
Per testare il proprio client si può lanciare il comando

```
./test.sh
```

dalla directory `$ROOT`. Si noti che il programma lancia anche altri tests (per le parti successive). Il programma indicherà se almeno i test obbligatori sono passati.

Parte 2 (max 6 punti)
Creare un client che implementa il protocollo descritto in precedenza. Naturalmente, è possibile sfruttare la propria soluzione del laboratorio 2.3 a questo scopo.
Il client riceve l'indirizzo IPv4 (in notazione decimale puntata) e il numero della porta (in notazione decimale) del server a cui collegarsi rispettivamente come primo e secondo argomento della linea di comando, il nome del file da scaricare come terzo argomento, e il nome da dare al file trasformato ricevuto come quarto argomento. Il file da inviare deve essere presente nella directory locale del client, altrimenti il client deve terminare prima di collegarsi al server. Il file trasformato ricevuto dal client deve essere scritto nella stessa directory. Se un file con lo stesso nome è già presente, questo deve essere sovrascritto.
Suggerimento: è più facile scrivere un client concorrente che usa due processi: il padre invia il file mentre il figlio riceve le risposte e scrive il file trasformato. Il padre attende la terminazione del figlio prima di terminare.
Il/i files C del programma client devono essere scritti tutti dentro la directory `$ROOT/source/client1`.
Il comando di test indicato per la Parte 1 tenterà di testare anche la Parte 2.

Parte 3 (max 4 punti)
Scrivere una nuova versione del server sviluppato nella Parte 1 (qui chiamato `server2`) che si comporti come il `server1` ma che

- sia un server concorrente che può servire almeno 3 clients contemporaneamente (per `server1` non c'era alcun requisito riguardo la concorrenza).
- chiude la connessione con il client se il client non ha terminato di inviare il numero di bytes annunciato all'inizio e non riceve alcun byte dal client per 3 secondi (al `server1` non era richiesto di farlo).

Il/i files C del programma server2 devono essere scritti dentro una directory `$ROOT/source/server2`.
Il comando di test indicato per la Parte 1 tenterà di testare anche la Parte 3.

Ulteriori istruzioni (comuni a tutte le parti)
Per passare l'esame è necessario almeno implementare un `server1` che risponda correttamente ad un upload di file secondo il nuovo protocollo, **oppure** è necessario implementare un `client1` ed un `server1` che possano interagire tra loro come ci si aspetta (cioè, che siano almeno in grado di scambiarsi, con upload e download, un correttamente un piccolo file).
La soluzione sarà considerata valida **se e solo se** può essere compilata tramite i seguenti comandi lanciati dalla cartella `source` (gli scheletri forniti compilano già tramite questi comandi, non spostarli, riempirli solo):

```
gcc -o socket_server1 server1/*.c *.c -Iserver1 -lpthread -lm

gcc -o socket_client1 client1/*.c *.c -Iclient1 -lpthread -lm

gcc -o socket_server2 server2/*.c *.c -Iserver2 -lpthread -lm
```

Per comodità, il programma di test controlla anche che la soluzione possa essere compilata con i comandi precedenti.
Si noti che tutti i files che sono necessari alla compilazione del client e del server devono essere inclusi nella directory `source` (per esempio è possibile usare i files dal libro dello Stevens, ma questi files devono essere inclusi da chi sviluppa la soluzione).
Tutti i files sorgenti prodotti (`client1, server1, server2,` e i files comuni) devono essere inclusi in un singolo archivio zip creato tramite il seguente comando bash (lanciato dalla cartella `$ROOT`):

```
./makezip.sh
```

Il file zip con la soluzione deve essere lasciato nella directory in cui è stato creato dal comando precedente.

Nota: controllare che il file zip sia stato creato correttamente estraendone il contenuto in una directory vuota, controllando il contenuto, e controllando che i comandi di compilazioni funzionino con successo (o che il programma di test funzioni).

Attenzione: gli ultimi 10 minuti dell'esame DEVONO essere usati per preparare l'archivio zip e per controllarlo (e aggiustare eventuali problemi). Se non sarà possibile produrre un file zip valido negli ultimi 10 minuti l'esame verrà considerato non superato.

La valutazione del lavoro sarà basata sui tests forniti ma anche altri aspetti del programma consegnato (per esempio la robustezza) saranno valutati. Di conseguenza, passare tutti i tests non significa che si otterrà necessariamente il punteggio massimo. Durante lo sviluppo del codice si faccia attenzione a scrivere un buon programma, non solo un programma che passa i tests forniti.

Per comodità, il testo dell’esercizio di laboratorio 2.3 è riportato nel seguito.

Esercizio 2.3 (server TCP iterativo)

Sviluppare un server TCP (in ascolto sulla porta specificata come primo parametro sulla riga di comando) che accetti richieste di trasferimento file da client ed invii il file richiesto.

Sviluppare un client che possa collegarsi ad un server TCP (all'indirizzo e porta specificati come primo e secondo parametro sulla riga di comando) per richiedere dei file e memorizzarli localmente. I nomi dei file da richiedere vengono forniti su standard input, uno per riga. Ogni file richiesto deve essere salvato localmente e deve essere stampato su standard output un messaggio circa l'avvenuto trasferimento, con nome, dimensione del file e timestamp di ultima modifica.

Il protocollo per il trasferimento del file funziona come segue: per richiedere un file il client invia al server i tre caratteri ASCII "GET" seguito dal carattere ASCII dello spazio e dai caratteri ASCII del nome del file, terminati da CR LF (*carriage return* e *line feed*, cioè due bytes corrispondenti ai valori 0x0d 0x0a in esadecimale, ossia `'\r'` e `'\n'` in notazione C, sempre senza spazi):

G	E	T		…filename…	CR	LF

(Nota: il comando include un totale di 6 caratteri più quelli del nome del file)

Il server risponde inviando:

+	O	K	CR	LF	B1	B2	B3	B4	T1	T2	T3	T4	File content……

Notare che il messaggio è composto da 5 caratteri, seguiti dal numero di byte del file richiesto (un intero senza segno su 32 bit in network byte order - bytes B1 B2 B3 B4 nella figura), e quindi dal timestamp dell'ultima modifica (Unix time, cioè numero di secondi dall'inizio dell' "epoca"), rappresentato come un intero senza segno su 32 bit in network byte order (bytes T1 T2 T3 T4 nella figura), e infine dai byte del file in oggetto.

Per ottenere il timestamp dell'ultima modifica al file, si faccia riferimento alle chiamate di sistema *stat* o *fstat*.

Il client può richiedere più file inviando più comandi GET. Quando intende terminare la comunicazione invia:

Q	U	I	T	CR	LF

(6 caratteri) e chiude il canale.

In caso di errore (es. comando illegale, file inesistente) il server risponde sempre con

-	E	R	R	CR	LF

(6 caratteri) e quindi chiude il canale col client.

Si faccia attenzione a porre l'eseguibile del client in una directory DIVERSA da quella dove gira il server. Si chiami questa cartella con lo stesso nome del programma client (questo per evitare che, con prove sullo stesso PC, salvando il file si sovrascriva lo stesso file che viene letto dal server).

Provare a collegare il proprio client con il server incluso nel materiale fornito per il laboratorio, ed il client incluso nel materiale fornito con il proprio server (notare che i files eseguibili sono forniti per

architetture sia a 32 bit sia a 64 bit. I files con suffisso _32 sono compilato per girare su sistemi Linux a 32 bit, quelli senza suffisso per sistemi a 64 bit. I computer al LABINF sono sistemi a 64 bit). Se sono necessarie delle modifiche al proprio client o al server, controllare attentamente che il client ed il server modificati comunichino correttamente sia tra loro sia con i client e server forniti. Al termine dell'esercizio, si deve avere un client e un server che possono comunicare tra loro e possono operare correttamente con il client ed il server forniti nel materiale del laboratorio.

Provare a trasferire un file binario di dimensioni notevoli (circa 100 MB). Verificare che il file sia identico tramite il comando cmp o diff e che l'implementazione sviluppata sia efficiente nel trasferire il file in termini di tempo di scaricamento.

Mentre è in corso un collegamento provare ad attivare un secondo client verso il medesimo server.

Provare ad attivare sul medesimo nodo una seconda istanza del server sulla medesima porta.

Provare a collegare il client ad un indirizzo esistente ma ad una porta su cui il server non è in ascolto.

Provare a disattivare il server (battendo CTRL+C nella sua finestra) mentre un client è collegato.