Programmazione Distribuita I

Test di programmazione socket, 29giugno 2015 – Tempo: 2 ore 10 minuti

Il materiale per l'esame si trova nella directory "`exam_dp_jun2015`" all'interno della propria home directory. Per comodità la directory contiene già uno scheletro dei file C che si devono scrivere (nella sotto-directory "`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 una variante del client e del server dell'esercizio 2.3 dei laboratori del corso (trasferimento di file). Per passare l'esame è necessario almeno scrivere il server che si comporti secondo le specifiche della parte 1.

Parte 1 (obbligatoria per passare l'esame, max 8 punti)
Scrivere un server (server1) che riceve due argomenti sulla linea di comando. Il primo argomento è il numero intero decimale della porta TCP sulla quale il server sta in ascolto, mentre il secondo è un altro numero intero decimale che specifica ogni quanti secondi il server deve verificare il contenuto dei files ed eventualmente aggiornarli utilizzando le connessioni già attive. Nel momento in cui viene instaurata una connessione il server si comporta secondo il protocollo specificato nell'esercizio di laboratorio 2.3, con le seguenti differenze:

1) Nella risposta positiva ad una richiesta GET, il server spedisce anche un numero intero senza segno su 32 bit in network byte order che rappresenta il tempo di ultima modifica del file che sarà trasferito, espresso come tempo Unix (cioè il numero di secondi da una certa data, ossia 1 gennaio 1970 ore 00.00.00 UTC). Questo numero, che può essere ottenuto dal server come specificato più avanti, è trasmesso immediatamente prima dell'intero che rappresenta il numero di bytes da spedire. Quindi, riassumendo, la risposta positiva alla richiesta GET è fatta di 5 caratteri ASCII "**`+OK`** `CR LF`", seguiti dal tempo di ultima modifica del file (un numero intero senza segno su 32 bit in network byte order), seguito dal numero di bytes del file (un numero intero senza segno su 32 bit in network byte order), seguito dai bytes del contenuto del file.
2) Dopo aver spedito il contenuto del file, mentre il client mantiene la connessione aperta, ogni N secondi, dove N è il valore del secondo argomento della linea di comando, il server verifica il tempo di ultima modifica dell'ultimo file richiesto dal client sul filesystem locale e, se questo valore è più grande del tempo di ultima modifica inviato in precedenza al client, il server spedisce un update del contenuto del file al client. L'update inizia con i 5 caratteri ASCII "**`UPD`** `CR LF`", seguiti dal nuovo tempo di ultima modifica, seguiti dal numero di bytes del file, seguito dai bytes del contenuto del file stesso. Se il client richiede un nuovo file tramite un comando GET, l'update del file richiesto in precedenza è interrotto. In quel caso, il server invierà il nuovo file richiesto e poi inizierà la procedura di update qui descritta per il nuovo file. In caso di errori nell'accesso al file durante un update, il server deve chiudere la connessione.

Il tempo di ultima modifica di un file può essere letto chiamando la funzione stat. Un esempio di chiamata è riportato di seguito (per maggiori informazioni si consulti la pagina di manuale di stat):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

char * fname      /* the file name */
struct stat buf;  /* a buffer where file information is read */
…
if (stat(fname, &buf)!=0) {
    // error
} else {
    unsigned intlastmodtime = buf.st_mtime; // read last modification time
}
```

Il/i files C del programma server devono essere scritti dentro una directory `$ROOT/source/server1`, dove `$ROOT` è la directory dell'esame (ovvero la cartella "exam_dp_jun2015"). Se si devono includere file di libreria (per esempio quello dello Stevens) che si vogliono riusare per le parti seguenti, questi files possono essere messi nella directory `$ROOT/source` (si ricorda che per includere questi files nei propri sorgenti è necessario specificare il percorso "`../source`").
Per testare il proprio server 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.
Si noti che per passare l'esame è sufficiente implementare la variazione specificata al punto 1. Per questa ragione, si suggerisce di implementare per prima cosa il punto 1, poi verificare che i tests passino, nel qual caso si salvi la soluzione usando il comando zip dato più avanti, e quindi implementare il punto 2.

Parte 2 (max 4 punti)
Scrivere un client (chiamato `client`) che riceve esattamente 3 argomenti sulla linea di comando: il primo argomento è l'indirizzo IP del server in notazione decimale puntata, il secondo argomento è il numero intero decimale della porta TCP, il terzo è un nome di file.
All'avvio il client deve collegarsi al server specificato dai primi due parametri della linea di comando e poi deve richiedere il file il cui nome è specificato come terno parametro della linea di comando. Il client deve poi ricevere il file, salvarne una copia nella directory corrente (quella da cui il client è stato lanciato) e settare, per la copia corrente, lo stesso valore di tempo di ultima modifica del file specificato dal server (si veda oltre per come settare il tempo di modifica del file). Dopo aver completato la ricezione del file, il client deve continuare ad attendere updates dal server. Quando il server spedisce un update, il client deve ricevere il nuovo contenuto del file e aggiornare la copia locale del file, insieme con il tempo di ultima modifica. Nel caso un cui si rilevino errori nei dati che arrivano dal server, o nel caso in cui il server chiuda la connessione, il client deve stampare un messaggio di errore sulla console e terminare.

Il tempo di ultima modifica del file può essere impostato chiamando la funzione utime(). Un esempio di chiamata è descritto di seguito (si veda la pagina di manuale di utime per maggiori informazioni):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <utime.h>

char * fname             /* the file name */
structutimbufbuf; /* a buffer where file information is read */
…
buf.actime = … ;  // set access time
buf.modtime = … ; // set last modification time

if (utime(fname, &buf)!=0) {
    // error
} else {
// success
}
```

Nota IMPORTANTE: Chiamare utime() dopo fclose(), altrimenti la fclose() sovrascrive il tempo di ultima modifica.

Il/i files C del programma client devono essere scritti dentro una directory `$ROOT/source/client`, dove `$ROOT` è la directory dell'esame (ovvero la cartella "exam_dp_jun2015").

Parte 3 (max 4 punti)

Scrivere una nuova versione del server sviluppato nella prima parte (server2) che sia in grado di servire almeno 3 clientsin parallelo (per il server1 non c'era alcun requisito riguardo la concorrenza). Se il server1 aveva già questa funzionalità, si può semplicemente copiare il codice di server1 su server2.

Il/i files C del programma server devono essere scritti dentro una directory`$ROOT/source/server2`, dove `$ROOT` è la directory dell'esame (ovvero la cartella "exam_dp_jun2015"). Il comando di test indicato per la Parte 1 tenterà di testare anche il server2.

Ulteriori istruzioni (comuni a tutte le parti)

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_client client/*.c *.c -Iclient -lpthread -lm

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

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).

Per comodità, il programma di test controlla anche che la soluzione possa essere compilata con i comandi precedenti.
Tutti i files sorgenti prodotti (`server1, server2, client` e i files comuni) devono essere inclusi in un singolo archivio zip creato tramite il seguente comando bash (lanciato dalla cartella `exam_dp_jun2015`):

```
zip socket.zip source/client/*.[ch] source/server[12]/*.[ch] source/*.[ch]
```

Il file zip con la soluzione deve essere lasciato nella directoryin 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 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.