SISTEMI DI ELABORAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA
SPECIFICHE DI PROGETTO A.A. 2013/2014

Il progetto consiste nello sviluppo di un'applicazione client/server. Client e server devono comunicare tramite **socket TCP**. Il server deve essere concorrente e la concorrenza deve essere implementata con i thread POSIX. Il thread main deve rimanere perennemente in attesa di nuove connessioni e le deve smistare ad un **pool di thread preallocati** che hanno il compito di gestire le richieste.
Si richiede di sviluppare un'applicazione di file sharing seguendo il paradigma client-server. L'applicazione deve permettere lo scambio di file fra utenti. In sintesi, un utente deve poter condividere dei file sul proprio computer con gli altri utenti del sistema. Deve inoltre poter scaricare sul proprio computer i file che altri utenti del sistema hanno condiviso.
Tutti i file condivisi sono registrati presso il server: il server dunque sa quali sono i file messi a disposizione e sa quali client li hanno condivisi. Fisicamente, tuttavia, i file risiedono solo sui client che li hanno condivisi. I client devono comunicare ogni operazione al server ed è il server che gestisce lo scambio dei messaggi e dei file fra i client.
Per sviluppare l'applicazione devono essere realizzati tre programmi: **sharing_server** per il lato server, **file_sender** e **file_receiver** per il lato client, rispettivamente per gestire le operazioni di condivisione (upload) e di ricezione (download) dei file.

1. FILE SENDER

Il File Sender deve essere avviato con la seguente sintassi:

```
./file_sender  <host remoto>  <porta>
```

dove:

- `<host remoto>` è l'indirizzo dell'host su cui è in esecuzione il server;
- `<porta>` è la porta su cui il server è in ascolto.

I comandi disponibili per l'utente devono essere:

- `!help`
- `!register`
- `!share`
- `!quit`

Il File Sender deve stampare tutti gli eventuali errori che si possono verificare durante

l'esecuzione.

All'avvio della connessione il File Sender **deve** inserire il suo username. Il server **deve** registrare l'avvenuta connessione specificando quale thread è stato assegnato a quel File Sender e ne gestirà le richieste.

Un esempio di esecuzione è il seguente:

```
$ ./file_sender 127.0.0.1 1234

Connessione al server 127.0.0.1 (porta 1234) effettuata con successo

Sono disponibili i seguenti comandi:
* !help --> Mostra l'elenco dei comandi disponibili
* !register --> Notifica al server quali file condividere
* !share --> Condividi i file (attendi richieste da servire)
* !quit --> Disconnettiti dal server e chiudi il programma

Inserisci il tuo nome: FS1
>
```

Un possibile messaggio in caso di errore di connessione è il seguente:

```
$ ./file_sender 127.0.0.1 1071

Impossibile connettersi a 127.0.0.1:1071
```

Implementazione dei comandi

a) `!help`: mostra l'elenco dei comandi disponibili.

Esempio di esecuzione:

```
Sono disponibili i seguenti comandi:
* !help --> Mostra l'elenco dei comandi disponibili
* !register --> Notifica al server quali file condividere
* !share --> Condividi i file (attendi richieste da servire)
* !quit --> Disconnettiti dal server e chiudi il programma
```

b) `!register`: notifica al server quali file condividere.

Il File Sender deve comunicare al server quanti e quali file vuole condividere con gli altri utenti del sistema. In particolare, il File Sender comunica al server il nome dei file che vuole condividere. Per semplicità, si assume che il nome del file sia sufficiente per identificare univocamente un file.
File Sender distinti possono condividere lo stesso file.
Il server mantiene un'apposita struttura dati che contiene le informazioni circa i File Sender connessi in ogni momento. Per ciascun File Sender, si memorizza lo username con cui il File Sender si è registrato al momento della connessione, lo stato del File Sender (disponibile o non), l'indice del thread che ne gestisce le richieste, il numero e il nome dei file che il File Sender

condivide col sistema.

Esempio di esecuzione:

```
>  !register
Quanti file vuoi condividere?
3
Inserisci il nome dei file da condividere:
(1/3) huey.png
(2/3) dewey.txt
(3/3) louie.doc
```

Bisogna gestire il caso di errore in cui i nomi dei file inseriti non si riferiscano a file esistenti.
Possibile segnalazione di errore:

```
>  !register
Quanti file vuoi condividere?
3
Inserisci il nome dei file da condividere:
(1/3) huey.png
(2/3) dewey.txt
(3/3) lcouie.doc
Errore. Lcouie.doc non esiste. Riprova
(3/3) louie.doc
```

c) `!share`: condividi i file (attendi richieste da servire).

Il comando permette di condividere con i File Receiver del sistema i file in precedenza registrati. Essenzialmente, il File Sender comunica al server di essere disponibile a ricevere richieste di trasferimento file e si mette in attesa, aspettando che il server gli comunichi che un File Receiver ha richiesto un file tra quelli messi in condivisione. All'arrivo di una richiesta, il File Sender verifica se il file richiesto è tra quelli registrati. In caso affermativo, stampa lo username del File Receiver richiedente e inizia la trasmissione del file verso il server (il quale inoltrerà i dati ricevuti al File Receiver richiedente). Un File Sender può effettuare un solo trasferimento alla volta. Il comando ritorna una volta che la trasmissione è terminata.

Il comando `share` determina lo stato del File Sender sul server. Per l'esattezza, prima di effettuare il comando `share` il File Sender è marcato come 'non disponibile' (i file registrati non sono infatti ancora accessibili). Appena effettuato il comando, lo stato del File Sender diventa 'libero'. Durante il trasferimento il server marca come 'non disponibile' il File Sender, così che non gli vengano inoltrate ulteriori richieste.

Esempio di esecuzione:

```
>  !share
In attesa di richieste...
Il File Receiver FR1 richiede il file louie.doc
Trasferimento del file louie.doc in corso...
Trasferimento del file louie.doc terminato!
```

È necessario gestire il caso di errore in cui il file richiesto per il trasferimento non esista.

Esempio di esecuzione:

```
>  !share
In attesa di richieste...
Il File Receiver FR1 richiede il file louie.doc
Errore! Il file louie.doc non esiste!
```

d) `!quit`: disconnettiti dal server e chiudi il programma.

il File Sender chiude il socket con il server ed esce. Il server stampa un messaggio che documenta la disconnessione del client. Il server, inoltre, dovrà gestire in maniera appropriata la disconnessione del client: dovrà pertanto rimuovere dalle proprie strutture dati i riferimenti al File Sender che si è disconnesso e ai file che condivideva.

Esempio di esecuzione:

```
>  !quit

Client  disconnesso  correttamente
```

2. FILE RECEIVER

Il File Receiver deve essere avviato con la seguente sintassi:

```
./file_receiver  <host remoto>  <porta>
```

dove:

- `<host remoto>` è l'indirizzo dell'host su cui è in esecuzione il server;
- `<porta>` è la porta su cui il server è in ascolto.

I comandi disponibili per l'utente devono essere:

- `!help`
- `!list`
- `!get`
- `!quit`

Il File Receiver deve stampare tutti gli eventuali errori che si possono verificare durante l'esecuzione.

All'avvio della connessione il File Receiver **deve** inserire il suo username. Il server **deve** registrare

l'avvenuta connessione specificando quale thread è stato assegnato a quel File Receiver e ne gestirà le richieste.

Un esempio di esecuzione è il seguente:

```
$ ./file_receiver 127.0.0.1 1234

Connessione al server 127.0.0.1 (porta 1234) effettuata con successo

Sono disponibili i seguenti comandi:
* !help --> Mostra l'elenco dei comandi disponibili
* !list --> Visualizza la lista dei file scaricabili
* !get --> Richiedi un file
* !quit --> Disconnettiti dal server e chiudi il programma

Inserisci il tuo nome: FR1
>
```

Un possibile messaggio in caso di errore di connessione è il seguente:

```
$ ./file_receiver 127.0.0.1 1071

Impossibile connettersi a 127.0.0.1:1071
```

Implementazione dei comandi

a) `!help`: mostra l'elenco dei comandi disponibili.

Esempio di esecuzione:

```
Sono disponibili i seguenti comandi:
* !help --> Mostra l'elenco dei comandi disponibili
* !list --> Visualizza la lista dei file scaricabili
* !get --> Richiedi un file
* !quit --> Disconnettiti dal server e chiudi il programma
```

b) `!list`: visualizza la lista dei file scaricabili.

Il File Receiver interroga il server, per sapere quali file sono stati condivisi dai vari File Sender e quindi quali file possono essere scaricati. Poiché più File Sender possono condividere il medesimo file, bisogna evitare che la lista dei file restituita dal server contenga dei duplicati.
[OPZIONALE] La lista dei file restituita può essere in ordine alfabetico.

Esempio di esecuzione:

```
> !list
I file disponibili sono:
canzone.mp3
huey.png
dewey.txt
file.txt
```

```
louie.doc
movie.mkv
pippo.png
pluto.jpeg
```

c) `!get`: richiedi un file.

Il File Receiver richiede di scaricare un file, indicandone il nome. La richiesta viene inviata al server, il quale individua un File Sender disponibile fra quelli che hanno condiviso il file desiderato. Il server inoltrerà la richiesta al File Sender e, per conto di tale File Sender, inizia a trasmettere il file al File Receiver. Se l'operazione va a buon fine, il File Receiver stampa il nome del File Sender da cui sta ricevendo il file e il trasferimento inizia automaticamente.

Esempio di esecuzione:

```
>  !get
Nome del file richiesto: louie.doc
File Sender FS1 disponibile al trasferimento
Trasferimento del file louie.doc in corso...
Trasferimento del file louie.doc terminato!
```

I possibili errori da gestire sono:

- Il file richiesto non è presente nel sistema,
- Nessuno tra i File Sender che condividono il file è disponibile (non hanno effettuato il comando `!share` o risultano impegnati in altri trasferimenti),
- errori a livello protocollare (e.g. errori durante il trasferimento del file).

Possibili segnalazioni di errore:

```
>  !get
Nome del file richiesto: lcouie.doc
Impossibile scaricare il file lcouie.doc: file non presente nel sistema.
```

```
>  !get
Nome del file richiesto: louie.doc
Impossibile scaricare il file louie.doc: nessun File Sender disponibile.
```

d) `!quit`: disconnettiti dal server e chiudi il programma.

il File Receiver chiude il socket con il server ed esce. Il server stampa un messaggio che documenta la disconnessione del client. Il server dovrà gestire in maniera appropriata la disconnessione del client.

Esempio di esecuzione:

```
>  !quit

Client  disconnesso  correttamente
```

3. SHARING SERVER

Il programma **sharing_server** si occupa di accettare nuove connessioni TCP, registrare nuovi utenti, gestire le richieste dei vari client per registrare e scaricare i file e gestire lo scambio dei file tra i client. Il server **sharing _server** è concorrente ed utilizza i thread per gestire le richieste. Il thread main, prima di mettersi in attesa di connessioni, prealloca un pool di thread gestori. Il thread main assegna ad un thread gestore (libero) del pool ogni nuova connessione che riceve. Il numero di thread del pool è specificato a tempo di compilazione (è una costante e non varia durante l'esecuzione del programma).

La sintassi del comando è la seguente:

```
./sharing_server <host> <porta>
```

dove:

- `<host>` è l'indirizzo su cui il server viene eseguito;
- `<porta>` è la porta su cui il server è in ascolto.

Possiamo schematizzare lo schema di funzionamento del server nel seguente modo:

1. il thread main crea `NUM_THREAD` thread gestori;
2. il thread main si mette in attesa delle connessioni in ingresso;
3. quando riceve una connessione in ingresso, il thread main:
    a. controlla il numero di thread occupati (ovvero quelli che stanno attualmente eseguendo una richiesta);
    b. se tutti i thread del pool sono occupati si blocca in attesa che uno diventi libero;
    c. se c'è almeno un thread libero ne sceglie uno (senza nessuna politica particolare) e lo attiva;
4. il thread gestore (all'infinito):
    a. si blocca finché non gli viene assegnata una richiesta;
    b. riceve il comando da un client;
    c. esegue la richiesta del client;
    d. se il thread ha ricevuto il comando `!quit`, il thread torna libero e a disposizione per servire un altro client. Altrimenti, il thread si blocca in attesa di un nuovo comando dallo stesso client, oppure si blocca in attesa che il trasferimento avanzi.

Se ne ricava che ogni thread gestore è associato ad uno e ad un unico client per tutto il tempo che quest'ultimo è connesso al server.

Una volta eseguito, **sharing_server** deve stampare a video delle informazioni descrittive dello stato del server (creazione del socket di ascolto, creazione dei thread, connessioni accettate, operazioni richieste dai client, ecc.).

Un esempio di esecuzione del server è il seguente:

```
$ ./sharing_server 127.0.0.1 1235
Indirizzo: 127.0.0.1 (Porta: 1235)
THREAD 0: pronto
THREAD 1: pronto
THREAD 2: pronto
THREAD 3: pronto
MAIN: connessione stabilita con il client 127.0.0.1:1235
MAIN: E' stato selezionato il thread 0
THREAD 0: pippo si è connesso
MAIN: connessione stabilita con il client 127.0.0.1:1235
MAIN: E' stato selezionato il thread 1
THREAD 1: pluto si è connesso
THREAD 0: ricezione comando !register
THREAD 0: pippo ha registrato i file huey.png, dewey.txt, louie.doc
THREAD 0: ricezione commando !share
THREAD 0: pippo è disponibile per il trasferimento
THREAD 1: ricezione comando !list
THREAD 1: ricezione comando !get
THREAD 1: pluto richiede louie.doc
THREAD 1: pluto inizia trasferimento di louie.doc da pippo
...
```

4. AVVERTENZE E SUGGERIMENTI

- **Test**
  Si testino le seguenti configurazioni:
  - un client viene avviato mentre alcuni thread gestori sono già occupati (ma ce n'è almeno uno libero);
  - un client viene avviato mentre non ci sono più thread gestori liberi.

- **Modalità di trasferimento dati tra client e server (e viceversa)**
  Client e server si scambiano dei dati tramite socket TCP. Prima che inizi ogni scambio è necessario che il ricevente sappia quanti byte deve leggere dal socket. Non è ammesso che vengano inviati su socket numeri arbitrari di byte.

- **Ogni risorsa condivisa deve essere protetta da opportuni meccanismi semaforici**

- **Non sono accettati meccanismi di attesa attiva**
  Quando un client è in attesa che l'utente inserisca un comando, il thread corrispondente nel server si blocca (l'operazione `recv`() è bloccante). Ugualmente, quando un client è in attesa di una richiesta di un file da parte di un altro client o semplicemente attende l'avanzamento del trasferimento, il thread corrispondente nel server si blocca (e dovrà

essere risvegliato al momento opportuno).

5. VALUTAZIONE DEL PROGETTO

Il progetto viene valutato durante lo svolgimento dell'esame. Il codice sarà compilato ed eseguito su sistema operativo Debian. Si consiglia di testare il sorgente su Debian prima dell'esame. La valutazione prevede le seguenti fasi.

1. **Compilazione dei sorgenti**. I client e il server vengono compilati attivando l'opzione -*Wall* che abilita la segnalazione di tutti i *warning*. Si consiglia di usare tale opzione anche durante lo sviluppo del progetto, *interpretando i messaggi forniti dal compilatore*.

2. **Esecuzione dell'applicazione**. I client e il server vengono eseguiti simulando una tipica sessione di utilizzo. In questa fase si verifica il corretto funzionamento dell'applicazione e il rispetto delle specifiche fornite.

3. **Esame del codice sorgente**. Il codice sorgente di client e server viene esaminato per controllarne l'implementazione.