SISTEMI DI ELABORAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA
SPECIFICHE DI PROGETTO A.A. 2011/2012

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 (insieme) di thread che hanno il compito di gestire le richieste.
L'applicazione da sviluppare è una CHAT semplificata. Una chat è uno strumento che permette la comunicazione tra due client. A tale scopo devono essere realizzati due programmi, **`chat_server`** per il lato server e **`chat_client`** per il lato client.

Lato client

Il client deve essere avviato con la seguente sintassi:

```
./chat_client <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`
- `who`
- `send_msg`
- `recv_msg`
- `bye`.

Il client deve stampare tutti gli eventuali errori che si possono verificare durante l'esecuzione.
All'avvio della connessione il client **deve** inserire il suo username. Il server **deve** registrare l'avvenuta connessione specificando quale thread è stato assegnato a quel client e ne gestirá le richieste.
Un esempio di esecuzione è il seguente:

```
$ ./chat_client 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
 * who --> mostra l'elenco dei client connessi al server
 * send_msg --> invia un messaggio ad un client specifico
 * recv_msg --> pone il client in attesa di un messaggio
 * bye --> disconnette il client dal server

Inserisci il tuo nome: client1
>
```

Un possibile messaggio in caso di errore in connessione può essere:

```
$ ./ chat_client 127.0.0.1 1071
Impossibile connettersi a 127.0.0.1:1071
```

Implementazione dei comandi

`help:` mostra l'elenco dei comandi disponibili.
Esempio di esecuzione:

```
> help
Sono disponibili i seguenti comandi:
 * help --> mostra l'elenco dei comandi disponibili
 * who --> mostra l'elenco dei client connessi al server
 * send_msg --> invia un messaggio ad un client specifico
 * recv_msg --> pone il client in attesa di un messaggio
 * bye --> disconnette il client dal server

>
```

`who`: mostra l'elenco dei client connessi. Il server mantiene una lista dei client connessi in ogni momento. La lista contiene gli username con cui i client si sono registrati al momento della connessione.
Esempio di esecuzione:

```
> who
Client connessi al server: client1 client2 client3 client4
>
```

`send_msg`: invia un messaggio ad un client specifico.
Un client vuole mandare un messaggio ad un altro client. A tale scopo il client ricevente deve essere messo preventivamente in ascolto (vedi comando `recv_msg`). L'utente deve poter inserire il messaggio da inviare e lo username del client ricevente.
Gli errori da gestire sono:

- client ricevente inesistente,
- client ricevente non in ascolto,
- errori a livello protocollare.

Esempio di esecuzione:

```
> send_msg
Inserire messaggio: ciao
Inserire destinatario: client2
Messaggio inviato correttamente
```

Possibile segnalazione di errore:

```
> send_msg
Inserire messaggio: ciao
Inserire destinatario: client_2
Impossibile inviare il messaggio perche' client inesistente
```

```
> send_msg
Inserire messaggio: ciao
Inserire destinatario: client2
Impossibile inviare il messaggio perche' client non in ricezione
```

`recv_msg`: pone il client in attesa di un messaggio. Un client per poter ricevere un messaggio da un altro client deve preventivamente mettersi in ascolto con questo comando.

```
> recv_msg
Rimango in attesa di un messaggio
```

`bye`: il client 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 di un cliente.

```
> bye
Client disconnesso correttamente
```

Lato server

Il programma **`chat_server`** si occupa di gestire le richieste provenienti dai client. Il server **`chat_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:

```
./chat_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:
    a. si sblocca finché non gli viene assegnata una richiesta;
    b. riceve il comando da un client;
    c. esegue la richiesta del client;
    d. si blocca in attesa di un nuovo comando dallo stesso client a meno che il comando non sia stata la bye e quindi, in tal caso, il thread ritorna libero e a disposizione per servire un altro client.

Una volta eseguito, **`chat_server`** deve stampare a video delle informazioni descrittive sullo 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:

```
$ ./server 127.0.0.1 1235
MAIN: 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 e' connesso
MAIN: Connessione stabilita con il client 127.0.0.1:1235
MAIN: E' stato selezionato il thread 1
THREAD 1: pluto si e' connesso
THREAD 1: Ricezione comando recv_msg
THREAD 0: Ricezione comando send_msg
THREAD 0: ricezione messaggio ck dal client pippo
THREAD 0: ricezione destinatario pluto dal client pippo
THREAD 0: Invio messaggio ciao al client pluto dal client pippo
THREAD 0: Ricezione comando recv_msg
THREAD 1: Ricezione comando send_msg
THREAD 1: ricezione messaggio fd dal client pluto
THREAD 1: ricezione destinatario pippo dal client pluto
THREAD 1: Invio messaggio ciao al client pippo dal client pluto
```

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. 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.

- **Come realizzare il server**.
  Conviene realizzare il server in modo incrementale, cosicché si possa testare singolarmente il funzionamento di diverse porzioni del codice.
  - Per testare il corretto funzionamento della parte relativa ai socket si può partire da un server che è in grado di servire una sola richiesta alla volta (server mono-processo).
  - Si può estendere il server così ottenuto introducendo la creazione dinamica dei thread (come descritto nei lucidi relativi alla programmazione distribuita).
  - Infine si può modificare il server multi-threaded utilizzando thread preallocati piuttosto che thread creati dinamicamente.

  Ovviamente lo scopo del progetto è creare un server concorrente che utilizzi un pool di thread preallocati. *Non saranno accettate soluzioni che non si attengano alle specifiche descritte in precedenza* (cfr. sezione seguente).

Valutazione del progetto

Il progetto viene valutato durante lo svolgimento dell'esame. La valutazione prevede le seguenti fasi.

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

Un esempio di esecuzione:

SERVER	CLIENT 1	CLIENT 2
$ ./server 127.0.0.1 1234	**$** ./client 127.0.0.1 1234	**$** ./client 127.0.0.1 1234
MAIN: Indirizzo: 127.0.0.1 (Porta: 1234)	Connessione al server 127.0.0.1 (porta 1234) effettuata con successo	Connessione al server 127.0.0.1 (porta 1234) effettuata con successo
THREAD 0: pronto		
THREAD 1: pronto	Sono disponibili i seguenti comandi:	Sono disponibili i seguenti comandi:
THREAD 2: pronto	* help --> mostra l'elenco dei comandi disponibili	* help --> mostra l'elenco dei comandi disponibili
THREAD 3: pronto	* who --> mostra l'elenco dei client connessi al server	* who --> mostra l'elenco dei client connessi al server
MAIN: Connessione stabilita con il client	* send_msg --> invia un messaggio ad un client specifico	* send_msg --> invia un messaggio ad un client specifico
MAIN: E' stato selezionato il thread 0	* recv_msg --> pone il client in attesa di un messaggio	* recv_msg --> pone il client in attesa di un messaggio
THREAD 0: client1 si e' connesso	* bye --> disconnette il client dal server	* bye --> disconnette il client dal server
MAIN: Connessione stabilita con il client		
MAIN: E' stato selezionato il thread 1	Inserisci il tuo nome: client1	Inserisci il tuo nome: client2
THREAD 1: client2 si e' connesso	> send_msg	> who
THREAD 1: Ricezione comando who	Inserire messaggio: ciao	Client connessi al server: client1 client2
THREAD 1: invio messaggio client1 client2 al client1	Inserire destinatario: client2	> recv_msg
THREAD 1: Ricezione comando recv_msg	Messaggio inviato correttamente	Rimango in attesa di un messaggio
THREAD 0: Ricezione comando send_msg	> recv_msg	Messaggio ricevuto dal client client1: ciao
THREAD 0: Ricezione messaggio ciao dal client client1	Rimango in attesa di un messaggio	> send_msg
THREAD 0: ricezione destinatario client2 dal client client1	Messaggio ricevuto dal client client2: ciao	Inserire messaggio: ciao
THREAD 0: Invio messaggio ciao al client client2 dal client client1	> bye	Inserire destinatario: client1
THREAD 0: Ricezione comando recv_msg	Client disconnesso correttamente	Messaggio inviato correttamente
THREAD 1: Ricezione comando send_msg		> bye
THREAD 1: ricezione messaggio ciao dal client client2		Client disconnesso correttamente
THREAD 1: ricezione destinatario client1 dal client client2		

THREAD 1: Invio messaggio ciao al client client1 dal client client2 THREAD 0: Ricezione comando bye THREAD 0: il client client1 ha chiuso la propria connessione THREAD 0: pronto THREAD 1: Ricezione comando bye THREAD 1: il client client2 ha chiuso la propria connessione THREAD 1: pronto		