SISTEMI DI ELABORAZIONE

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA ELETTRONICA

SPECIFICHE DI PROGETTO A.A. 2012/2013

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 preallocati che hanno il compito di gestire le richieste.
L'applicazione da sviluppare è il gioco del Mastermind seguendo il paradigma client-server.
Mastermind è un gioco testa a testa, in cui ciascuno dei due giocatori deve indovinare un codice segreto composto dal suo avversario. Nella versione originale di Mastermind[1], il codice segreto è di quattro cifre e per comporlo il giocatore ha a disposizione le dieci cifre del sistema decimale (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).
I due giocatori devono entrambi comporre un proprio codice, dopodiché devono cercare di indovinare la combinazione dell'avversario. A turno, ciascun giocatore fa un tentativo indicando una combinazione. Dopo ogni tentativo, l'avversario gli fornisce degli aiuti comunicando il numero di cifre giuste al posto giusto, cioè le cifre del tentativo che sono effettivamente presenti nel codice al posto tentato, e il numero di cifre giuste al posto sbagliato, cioè le cifre del tentativo che sono effettivamente presenti nel codice, ma non al posto tentato. Non bisogna comunicare quali cifre sono giuste o sbagliate ma solo quante. Vince il giocatore che riesce ad indovinare la combinazione dell'avversario nel minor numero di turni.
Per sviluppare l'applicazione devono essere realizzati due programmi, **mastermind_server** per il lato server e **mastermind_client** per il lato client. I client devono comunicare ogni operazione e mossa al server: è il server che gestisce lo scambio dei messaggi fra i client.

1 Lato client

Il client deve essere avviato con la seguente sintassi:

```
./mastermind_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:

[1] Ben più nota è la variante basata sui colori, ma per semplicità d'implementazione il progetto si riferisce alla versione originale.

- `!aiuto`
- `!giocatori`
- `!crea`
- `!partecipa`
- `!abbandona`
- `!esci`

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 gestirà le richieste.
Un esempio di esecuzione è il seguente:

```
$ ./mastermind_client 127.0.0.1 1234

Connessione al server 127.0.0.1 (porta 1234) effettuata con successo

Sono disponibili i seguenti comandi:
 * !aiuto --> mostra l'elenco dei comandi disponibili
 * !giocatori --> mostra l'elenco dei client connessi al server
 * !crea --> crea una nuova partita e attendi un avversario
 * !partecipa --> unisciti ad una partita e inizia a giocare
 * !abbandona --> abbandona l'attuale partita
 * !esce --> disconnettiti dal server e chiudi il programma

Inserisci il tuo nome: client1
>
```

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

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

Implementazione dei comandi

`!aiuto:` mostra l'elenco dei comandi disponibili.

Esempio di esecuzione:

```
Sono disponibili i seguenti comandi:
 * !aiuto --> mostra l'elenco dei comandi disponibili
 * !giocatori --> mostra l'elenco dei client connessi al server
 * !crea --> crea una nuova partita e attendi un avversario
 * !partecipa --> unisciti ad una partita e inizia a giocare
```

```
 * !abbandona --> abbandona l'attuale partita
 * !esce  -->  disconnettiti  dal  server e chiudi il programma
```

`!giocatori`: mostra l'elenco dei client connessi.
Il server mantiene un'apposita struttura dati che contiene i client connessi in ogni momento. Per ciascun client, si memorizza lo username con cui il client si è registrato al momento della connessione e l'indice del thread che ne gestisce le richieste.

Esempio di esecuzione:

```
>  !giocatori
Client  connessi  al  server:  client1  client3  client4
>
```

`!crea`: il client comunica al server l'intenzione di dare inizio ad una nuova partita. Dopodiché il client resta in attesa che un avversario si unisca alla partita. Quando l'avversario è stato trovato, la partita inizia.

Esempio di esecuzione:

```
>  !crea
Nuova partita creata.
In attesa di un avversario...
client2 si è unito alla partita.
La partita è iniziata.
Digita la tua combinazione: 7394

Sta a te. Inserisci il tuo tentativo: 9273
client2 dice: 1 Cifra giusta al posto giusto, 0 cifre giuste al posto sbagliato

…
```

`!partecipa`: il client comunica al server l'intenzione di sfidare un utente (che avrà preventivamente provveduto a creare una nuova partita tramite il comando `!crea`).
L'utente deve poter inserire lo username dell'avversario.

I possibili errori da gestire sono:

- lo username inserito è inesistente,
- l'avversario non è in ascolto (non ha eseguito il comando `!crea`),
- l'avversario è già occupato in una partita,
- errori a livello protocollare.

Se l'operazione va a buon fine, la partita inizia automaticamente.

Esempio di esecuzione:

```
>  !partecipa
Inserire lo username dell'utente da sfidare: client1
La partita è iniziata.
Digita la tua combinazione segreta: 5296

In attesa che client1 faccia la sua mossa...
client1 dice 3726. Il suo tentativo è sbagliato

Sta a te. Inserisci il tuo tentativo: 1234
client1 dice: 1 Cifra giusta al posto giusto, 2 cifre giuste al posto sbagliato

…
```

Possibile segnalazione di errore:

```
>  !partecipa
Inserire lo username dell'utente da sfidare: client1
Impossibile  connettersi a client1:  utente  inesistente.
```

```
>  !partecipa
Inserire lo username dell'utente da sfidare: client1
Impossibile  connettersi a client1:  l'utente è impegnato in altra partita.
```

`!abbandona`: disconnette il client dall'attuale partita. Il comando è eseguibile soltanto durante una partita ed è l'unico comando eseguibile durante una partita.

```
>  !abbandona
Disconnessione  avvenuta  con  successo:  TI  SEI  ARRESO
```

Quando il client comunica al server la disconnessione, l'utente è di nuovo libero e può iniziare una nuova partita. Il server provvede a comunicare al client avversario che ha vinto la partita per resa dello sfidante.

`!esci`: 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 del client.

```
>  !esci
Client  disconnesso  correttamente
```

Fasi del gioco

`Scelta della combinazione`: appena la partita inizia, i due giocatori scelgono la rispettiva combinazione segreta (di 4 cifre). Ciascun client memorizza la propria combinazione, senza comunicarla al server.

`Effettuare un tentativo`: dopo aver scelto la combinazione, i giocatori iniziano la sfida: a turno, cercano di indovinare la combinazione dell'avversario. Il client del giocatore che effettua il tentativo trasmette la combinazione proposta al server, il quale la inoltra al client dell'avversario. Il client che ha creato la partita (tramite il comando `!crea`) inizia per primo.

`Attendere la mossa dell'avversario`: quando è il turno dell'avversario, il client si pone in attesa della combinazione del giocatore sfidante. Quando il server gli comunica il tentativo dell'avversario, il client verifica la correttezza di quest'ultimo e invia i suggerimenti (numero di cifre esatte al posto esatto e numero di cifre esatte al posto sbagliato) al server, che li inoltra all'avversario. La verifica del tentativo è eseguita in automatico via software senza l'intervento dell'utente.
Dopo aver inviato i suggerimenti, l'utente acquisisce il turno di gioco. L'avversario si metterà in attesa della sua mossa.
Il client che si è aggiunto come sfidante alla partita (tramite il comando `!partecipa`) inizia per secondo. Durante il primo turno dunque aspetta il tentativo dello sfidante.

`Fine della partita`: la partita termina quando un giocatore indovina la combinazione segreta dell'avversario oppure abbandona la partita oppure il suo avversario abbandona la partita.

2 Lato server

Il programma **mastermind_server** si occupa di accettare nuove connessioni TCP, registrare nuovi utenti, gestire le richieste dei vari client per aprire nuove partite e gestire lo scambio di messaggi tra i client durante la partita. Il server **mastermind_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:

```
./mastermind_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 `esci`, il thread ritorna 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 della mossa da parte dell'avversario del client servito.

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, **mastermind_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:

```
$ ./mastermind_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 !crea
THREAD 1: ricezione comando !partecipa
THREAD 1: ricezione richiesta partita contro pippo
...
```

3 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 della mossa da parte dell'avversario, il thread corrispondente nel server si blocca (sarà il thread associato all'avversario a doverlo risvegliare quando l'avversario avrà fatto la sua mossa).

4 Valutazione del progetto

Il progetto viene valutato durante lo svolgimento dell'esame. Il tutto verrà fatto girare su una macchina con sistema operativo Debian. Si consiglia caldamente di testare il sorgente su una macchina Debian prima di venire all'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.