RETI INFORMATICHE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
SPECIFICHE DI PROGETTO A.A. 2010/2011

Il progetto consiste nello sviluppo di un'applicazione client/server. Sia il server che il client dovranno essere mono-processo e sfrutteranno l'I/O multiplexing (tramite l'api `select`) per gestire più input ed output simultaneamente.

L'applicazione da sviluppare è una CHAT peer2peer 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.

La scambio di informazioni tra client e server avverrà tramite socket TCP. Queste informazioni saranno solo informazioni di controllo che serviranno per implementare la comunicazione peer2peer. Lo scambio di messaggi di testo tra i client avverrà tramite socket UDP.

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`
- `!quit`
- `!connect nome_utente`
- `!disconnect`

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 e porta di ascolto per i messaggi di testo che verranno veicolati tramite socket udp.

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
 * !connect nome_client --> avvia una chat con l'utente nome_client
 * !disconnect -->disconnette il client dall'attuale chat intrapresa con un altro peer
 * !quit --> 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:

```
Sono disponibili i seguenti comandi:
 * !help --> mostra l'elenco dei comandi disponibili
 * !who --> mostra l'elenco dei client connessi al server
 * !connect nome_client --> avvia una chat con l'utente nome_client
 * !disconnect -->disconnette il client dall'attuale chat intrapresa con un altro peer
 * !quit --> 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 l'inidirizzo ip e la porta di ascolto UDP con cui i client si sono registrati al momento della connessione.

Esempio di esecuzione:

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

`!connect nome_client`: il client avvia una chat con l'utente nome_client.

Un client vuole avviare una chat con un altro client di nome nome_client.

Gli errori da gestire sono:

- nome_client inesistente,

- errori a livello protocollare.
- nome_client gia occupato in una chat

Più in dettaglio il client farà richiesta al server (sempre tramite tcp) se esiste l'utente nome_client.

Se esiste e non è occupato il server manderà una richiesta al client nome_client per sapere se è intenzionato ad accettare la chat con il client. Se la risposta è affermativa allora il server comunicherà al client l'indirizzo ip e porta di ascolto UDP dell client nome_client. Se negativa il server risponderà con uno specifico messaggio di errore.

Una volta avviata la chat ogni messaggio deve essere preceduto dal nome del client.

La concorrenza tra standard input e socket dovrà essere gestita sempre tramite select.

Esempio di esecuzione:

```
> !connect nome_client
[Opzionale] nome_client ha accettato la sessione
Chat avviata con nome_client
#ciao
#nome_client: ciao
#
```

[Opzionale] gestire chat con più di due utenti.

[Opzionale] implementare la chat (messaggi di testo su UDP) in modo di avere l'affidabilità della consegna dei messaggi o almeno avere un messaggio di errore se il messaggio mandato non è arrivato oppure è arrivato solo in parte

Possibile segnalazione di errore:

```
> !connect nome_client
Impossibile connettersi a nome_client: utente inesistente.
```

```
> !connect nome_client
[Opzionale]Impossibile connettersi a nome_client: l'utente ha rifiutato la chat.
```

`Chat Avviata:` Quando la chat è avviata il sistema deve accettare qualsiasi stringa come input (messaggi da mandare) ma dovrà anche accettare i seguenti comandi:

1. !disconnect
2. !quit
3. [opzionale] !who
4. [opzionale] !help

Se una chat è avviata si deve capire dal primo carattere della shell:

1. > shell comandi (sono accettati solo i comandi di base, se immesso altro viene restituito un errore)
2. # shell chat (si accetta qualsiasi stringa compreso i comandi precedentemente elencati)

`!disconnect`: disconnette il client dall'attuale chat.

```
# !disconnect
Disconnessione avvenuta con successo
```

Quando un client esegue una disconnessione comunicherà

1. al server (tramite tcp) che l'utente è di nuovo libero
2. all'altro client (tramite udp) che è stata effettuata una disconnessione

Il client che riceve il messaggio di disconnessione dovrà comunicare al server che è di nuovo libero.

`!quit`: il client chiude il socket con il server, il socket udp 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.

```
> !quit
Client disconnesso correttamente
```

Lato server

Il programma **`chat_server`** si occupa di gestire le richieste provenienti dai client. Il server **`chat_server`** tramite l'uso della select, accetterà nuove connessioni tcp, registrerà nuovi utenti e gestirà le richieste dei vari client per aprire nuove chat.

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.

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

Un esempio di esecuzione del server è il seguente:

```
$ ./server 127.0.0.1 1235
Indirizzo: 127.0.0.1 (Porta: 1235)
Connessione stabilita con il client 127.0.0.1:1235
```

```
pippo si e' connesso
pippo è libero
Connessione stabilita con il client 127.0.0.1:1235
pluto si e' connesso
pluto è libero
pippo si è connesso a pluto
pluto si è disconnesso da pippo
pippo è libero
pluto è libero
```

Avvertenze e suggerimenti

- **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.
- **Il client si disconnette in automatico da una eventuale chat dopo 1 minuto di inattività:**
  - Non viene scritto niente nello standard input per un minuto
  - Non si riceve niente sul socket udp per un minuto

Valutazione del progetto

Il progetto viene valutato durante lo svolgimento dell'esame. Il tutto verrà fatto girare su una macchina con sistema operativo FreeBSD v6.2. Si consiglia caldamente di testare il sorgente su una macchina FreeBsd 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.