1.1RETI INFORMATICHE

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

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 il gioco della BATTAGLIA NAVALE seguendo il paradigma peer2peer. Il server avrà il compito di memorizzare gli utenti connessi e le porte su cui rimarranno in ascolto.

La mappa del gioco dovrà essere di 10x10 dove le coordinate sono rappresentate da lettere e numeri (esempio: C4). Il numero di navi che dovranno essere presenti nella mappa sono:

Numero NaviDimensione Nave16243342

Lo 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.

1.1.1Lato client

Il client deve essere avviato con la seguente sintassi:

```
./battaglia_navale_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`
- `!show_enemy_map`
- `!show_my_map`

- `!hit coordinates`

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, porta di ascolto UDP per i comandi relativi al gioco, posizione di ogni singola nave e relativo orientamento (verticale/orizzontale). Inoltre il client deve effettuare un check per ogni posizionamento in maniera tale da controllare se la nave appena impostata esce dalla mappa oppure se va a posizionarsi sopra un'altra nave. Se si verifica un errore il client lo comunica e si aspetta di nuovo di ricevere le coordinate. Le informazioni relativi alla posizione delle navi deve risiedere nel client.

Un esempio di esecuzione è il seguente:

```
$ ./battaglia_navale_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 partita con l'utente nome_client
 * !disconnect -->disconnette il client dall'attuale partita intrapresa con
un altro peer
 * !quit --> disconnette il client dal server
 * !show_enemy_map --> mostra la mappa dell'avversario
 * !show_my_map --> mostra la proprio mappa
 * !hit coordinates --> colpisce le coordinate coordinates (valido solo
quando è il proprio turno)


Inserisci il tuo nome: client1
Inserisci la porta UDP di ascolto: 1025
Inserisci le coordinate della nave 1 di dimensione 6:
Inserisci l'orientamento della nave 1 di dimensione 6:
…
…
```

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 partita con l'utente nome_client
 * !disconnect -->disconnette il client dall'attuale partita intrapresa con
un altro peer
 * !quit --> disconnette il client dal server
```

```
 * !show_enemy_map --> mostra la mappa dell'avversario
 * !show_my_map --> mostra la proprio mappa
 * !hit coordinates --> colpisce le coordinate coordinates (valido solo
quando è il proprio turno)
```

`who`: mostra l'elenco dei client connessi. Il server mantiene una lista dei client connessi in ogni momento. La lista contiene gli username l'indirizzo 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 partita con l'utente nome_client.

Un client vuole avviare una partita 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 partita

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 partita con il client. Se la risposta è affermativa allora il server comunicherà al client l'indirizzo ip e porta di ascolto UDP del client nome_client. Se negativa il server risponderà con uno specifico messaggio di errore.

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

Esempio di esecuzione:

```
> !connect nome_client
nome_client ha accettato la partita
partita avviata con nome_client
E' il tuo turno:
#!hit c4
nome_client: COLPITO
E' il turno di nome_client
nome_client colpito la zona E7: ACQUA
E' il tuo turno:
#!hit c5
nome_client: ACQUA
```

Possibile segnalazione di errore:

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

```
> !connect nome_client
Impossibile connettersi a nome_client: l'utente ha rifiutato la partita.
```

`Partita Avviata:` Quando la partita è avviata il sistema dovrà accettare i seguenti comandi:

1!disconnect

2!quit

3[opzionale] !who

4[opzionale] !help

5!hit coordinates

6!show_my_map

7!show_enemy_map

Se una partita 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 partita (si accettano i comandi relativi al gioco)

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

```
# !disconnect
Disconnessione avvenuta con successo: TI SEI ARRESO
```

Quando un client esegue una disconnessione comunicherà

1al server (tramite tcp) che l'utente è di nuovo libero

2all'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 e stampare a video un messaggio di vittoria.

`!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
```

`!show-my-map`: il client mostra lo stato attuale della propria mappa, visualizzando la posizione delle proprie navi e degli attacchi pervenuti fino a quel momento da parte dell'avversario.

`!show-enemy-map`: il client mostra lo stato attuale della mappa dell'avversario, visualizzando gli attacchi fatti fino a quel momento differenziando se sono state colpite navi oppure no.

1.1.1Lato server

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

La sintassi del comando è la seguente:

```
./battaglia_navale_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, **battaglia_navale_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:

```
$ ./ battaglia_navale_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
```

1.1.1Avvertenze e suggerimenti

- **Modalità di trasferimento dati tra client e server (e viceversa)**.

1Client 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 partita dopo 1 minuto di inattività:**

1Non viene scritto niente nello standard input per un minuto

2Non si riceve niente sul socket udp per un minuto

1.1.1

1.1.2Valutazione 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.