Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Classe n. L-31

Progetto di sistemi operativi

Traccia A

Professore:
Finzi Alberto

Candidati:
Turco Mario
Matr. N8600/2503
Longobardi Francesco
Matr. N8600/2468

Anno Accademico
2019/2020

Indice

1 Istruzioni preliminari

1.1 Modalità di compilazione

Il progetto è provvisto di un file makefile il quale è in grado di compilare autonomamente l'intero progetto. Per utilizzare il makefile aprire la cartella del progetto tramite la console di sistema e digitare "make".
In alternativa è possibile compilare manualemente il client ed il server con i seguenti comandi:

```
gcc -o server server.c boardUtility.c parser.c list.c -lpthread
gcc -o client client.c boardUtility.c parser.c list.c -lpthread
```

2 Guida all'uso

2.1 Server

Una volta compilato il progetto è possibile avviare il server digitando da console il seguente comando

```
./server users
```

L'identificativo *users* ri riferisce al nome del file sul quale sarà salvata la lista degli utenti e delle loro credenziali.
È possibile scegliere un nome a piacimento per il file purchè esso sia diverso da *log*.

2.2 Client

Una volta compilato il progetto è possibile avviare il cient digitando da console il seguente comando:

```
./client ip porta
```

Dove *ip* andrà sostituito con l'ip o l'indirizzo URL del server e *porta* andrà sostituito con la porta del server.
Una volta avviato il client comprarià il menu con le scelte 3 possibili: accedi, registrati ed esci.
Una volta effettuata la registrazione dell'utente è possibile effettuare l'accesso al programma al seguito del quale verranno mostrate sia la mappa dei gioco si le istruzioni di gioco.

3 Comunicazione tra client e server

Di seguito verranno illustrate le modalità di comunicazione tra client e server.

3.1 Configurazione del server

Il socket del server viene configurato con famiglia di protoccoli PF_NET, con tipo di trasmissione dati SOCK_STREAM e con protocollo TCP. Mostriamo di seguito il codice sorgente:

Listato 1: Configurazione socket del server

```
1  void configuraSocket(struct sockaddr_in mio_indirizzo) {
2    if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
3      perror("Impossibile creare socket");
4      exit(-1);
5    }
6    if (setsockopt(socketDesc, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) <
         0)
7      perror("Impossibile impostare il riutilizzo dell'indirizzo ip e della "
8             "porta\n");
9    if ((bind(socketDesc, (struct sockaddr *)&mio_indirizzo,
10             sizeof(mio_indirizzo))) < 0) {
11     perror("Impossibile effettuare bind");
12     exit(-1);
13   }
14 }
```

È importante notare anche come il server riesca a gestire in modo concorrente più client tramite l'uso di un thread dedicato ad ogni client. Una volta aver configurato il socket, infatti, il server si mette in ascolto per nuove connessioni in entrata ed ogni volta che viene stabilita una nuova connessione viene avviato un thread per gestire tale connessione. Di seguito il relativo codice:

Listato 2: Procedura di ascolto del server

```
1  void startListening()
2  {
3      pthread_t tid;
4      int clientDesc;
5      int *puntClientDesc;
6      while (1)
7      {
8        if (listen(socketDesc, 10) < 0)
9          perror("Impossibile mettersi in ascolto"), exit(-1);
10       printf("In ascolto..\n");
11       if ((clientDesc = accept(socketDesc, NULL, NULL)) < 0)
12       {
13         perror("Impossibile effettuare connessione\n");
14         exit(-1);
15       }
16       printf("Nuovo client connesso\n");
17       puntClientDesc = (int *)malloc(sizeof(int));
18       *puntClientDesc = clientDesc;
19       pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
20     }
21     close(clientDesc);
22     quitServer();
23 }
```

In particolare al rigo 19 notiamo la creazione di un nuovo thread per gestire la connessione in entrata a cui passiamo il descrittore del client di cui si deve occupare.

3.2 Configurazione del client

Il cliente invece viene configurato e si connette al server tramite la seguente funzione:

Listato 3: Configurazione e connessione del client

```
1  int connettiAlServer(char **argv) {
2      char *indirizzoServer;
3      uint16_t porta = strtoul(argv[2], NULL, 10);
4      indirizzoServer = ipResolver(argv);
5      struct sockaddr_in mio_indirizzo;
6      mio_indirizzo.sin_family = AF_INET;
7      mio_indirizzo.sin_port = htons(porta);
8      inet_aton(indirizzoServer, &mio_indirizzo.sin_addr);
9      if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
10       perror("Impossibile creare socket"), exit(-1);
11     else
12       printf("Socket creato\n");
13     if (connect(socketDesc, (struct sockaddr *)&mio_indirizzo,
14                 sizeof(mio_indirizzo)) < 0)
15       perror("Impossibile connettersi"), exit(-1);
16     else
17       printf("Connesso a %s\n", indirizzoServer);
18     return socketDesc;
19   }
```

Si noti come al rigo 9 viene configurato il socket ed al rigo 13 viene invece effettuato il tentativo di connessione al server.
Al rigo 3 invece viene convertita la porta inserita in input (argv[2]) dal tipo stringa al tipo della porta (uint16_t ovvero unsigned long integer).
Al rigo 4 notiamo invece la risoluzione dell'url da parte della funzione ipResolver che è riportata di seguito:

Listato 4: Risoluzione url del client

```
1  char *ipResolver(char **argv) {
2      char *ipAddress;
3      struct hostent *hp;
4      hp = gethostbyname(argv[1]);
5      if (!hp) {
6              perror("Impossibile risolvere l'indirizzo ip\n");
7              sleep(1);
8              exit(-1);
9          }
10     printf("Address:\t%s\n", inet_ntoa(*(struct in_addr *)hp->h_addr_list[0]));
11     return inet_ntoa(*(struct in_addr *)hp->h_addr_list[0]);
12 }
```

Al rigo 4, tramite l'url o l'indirizzo ip viene riempita la struttura hostent da cui poi possiamo estrarre l'indirizzo ip presente nel campo h_addr_list che, in effetti, è un array che contiene i vari indirizzi ip associati a quell'host.
Infine, al rigo 11 decidiamo di ritoranre soltanto il primo indirizzo convertito in ascii.

3.3 Comunicazione tra client e server

La comunicazione tra client e server avviene tramite write e read sul socket.
Il comportamento del server e del client è determinato da particolari messaggi inviati e/o ricevuti che codificano, tramite interi o caratteri, la richiesta da parte del client di usufruire di un determinato servizio e la relativa risposta del server.

3.3.1 Esempio: la prima comunicazione

In particolare, una volta effettuata la connessione, il server attenderà un messaggio dal client per poter avviare una delle tre possibili procedure, ovvero login, registrazione ed uscita (rispettivamente codici: 1,2,3).
Di seguito sono riportate le realtive funzioni di gestione che entrano in esecuzione subito dopo aver stabilito la connessione tra client e server.

Listato 5: Prima comunicazione del server

```
1  void *gestisci(void *descriptor) {
2    int bufferReceive[2] = {1};
3    int client_sd = *(int *)descriptor;
4    int continua = 1;
5    char name[MAX_BUF];
6    while (continua) {
7      read(client_sd, bufferReceive, sizeof(bufferReceive));
8      if (bufferReceive[0] == 2)
9        registraClient(client_sd);
10     else if (bufferReceive[0] == 1)
11       if (tryLogin(client_sd, name)) {
12         play(client_sd, name);
13         continua = 0;
14       } else if (bufferReceive[0] == 3)
15         disconnettiClient(client_sd);
16       else {
17         printf("Input invalido, uscita...\n");
18         disconnettiClient(client_sd);
19       }
20   }
21   pthread_exit(0);
22 }
```

Si noti come il server riceva, al rigo 7, il messaggio codificato da parte del client e metta in esecuzione la funzione corrispondente.

Listato 6: Prima comunicazione del client

```
1  int gestisci() {
2    char choice;
3    while (1) {
4      printMenu();
5      scanf("%c", &choice);
6      fflush(stdin);
7      system("clear");
8      if (choice == '3') {
9        esciDalServer();
10       return (0);
11     } else if (choice == '2') {
12       registrati();
13     } else if (choice == '1') {
14       if (tryLogin())
15         play();
16     } else
17       printf("Input errato, inserire 1,2 o 3\n");
18   }
```

4 Comunicazione durante la partita

Una volta effettuato il login, il client potrà iniziare a giocare tramite la funzione play (Vedi List. 7 e List. 8) che rappresentano il cuore della comunicazione tra client e server.
La funzione play del server consiste di un ciclo infinito nel quale il server invia al client tre informazioni importanti:

- La griglia di gioco (Rigo 26)
- Il player con le relative informazioni (Rigo 28 a 31)
- Il timer (Rigo 32)

Dopodichè il thread del server rimane in attesa di ricevere l'input del client per spostare il giocaore sulla mappa tramite la relativa funzione. (Rigo 34)
Oltre questo, la funzione play del server si occupa anche di generare la nuova posizione del player (Rigo 52 a 66)quando viene effettuato il cambio di mappa (allo scadere del tempo o alla raggiunta del massimo punteggio).

Listato 7: Funzione play del server

```
void play(int clientDesc, char name[]) {
  int true = 1;
  int turnoFinito = 0;
  int turnoGiocatore = turno;
  int posizione[2];
  int destinazione[2] = {-1, -1};
  PlayerStats giocatore = initStats(destinazione, 0, posizione, 0);
  Obstacles listaOstacoli = NULL;
  char inputFromClient;
  if (timer != 0) {
    inserisciPlayerNellaGrigliaInPosizioneCasuale(
        grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
        giocatore->position);
    playerGenerati++;
  }
  while (true) {
    if (clientDisconnesso(clientDesc)) {
      freeObstacles(listaOstacoli);
      disconnettiClient(clientDesc);
      return;
    }
    char grigliaTmp[ROWS][COLUMNS];
    clonaGriglia(grigliaTmp, grigliaDiGiocoConPacchiSenzaOstacoli);
    mergeGridAndList(grigliaTmp, listaOstacoli);
    // invia la griglia
    write(clientDesc, grigliaTmp, sizeof(grigliaTmp));
    // invia la struttura del player
    write(clientDesc, giocatore->deploy, sizeof(giocatore->deploy));
    write(clientDesc, giocatore->position, sizeof(giocatore->position));
    write(clientDesc, &giocatore->score, sizeof(giocatore->score));
    write(clientDesc, &giocatore->hasApack, sizeof(giocatore->hasApack));
    sendTimerValue(clientDesc);
    // legge l'input
    if (read(clientDesc, &inputFromClient, sizeof(char)) > 0)
      numMosse++;
    if (inputFromClient == 'e' || inputFromClient == 'E') {
      freeObstacles(listaOstacoli);
      listaOstacoli = NULL;
      disconnettiClient(clientDesc);
    } else if (inputFromClient == 't' || inputFromClient == 'T') {
      write(clientDesc, &turnoFinito, sizeof(int));
      sendTimerValue(clientDesc);
    } else if (inputFromClient == 'l' || inputFromClient == 'L') {
      write(clientDesc, &turnoFinito, sizeof(int));
      sendPlayerList(clientDesc);
    } else if (turnoGiocatore == turno) {
      write(clientDesc, &turnoFinito, sizeof(int));
      giocatore =
          gestisciInput(grigliaDiGiocoConPacchiSenzaOstacoli,
                        grigliaOstacoliSenzaPacchi, inputFromClient, giocatore,
                        &listaOstacoli, deployCoords, packsCoords, name);
    } else {
      turnoFinito = 1;
      write(clientDesc, &turnoFinito, sizeof(int));
      freeObstacles(listaOstacoli);
      listaOstacoli = NULL;
      inserisciPlayerNellaGrigliaInPosizioneCasuale(
          grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
          giocatore->position);
      giocatore->score = 0;
      giocatore->hasApack = 0;
      giocatore->deploy[0] = -1;
      giocatore->deploy[1] = -1;
      turnoGiocatore = turno;
      turnoFinito = 0;
      playerGenerati++;
    }
  }
}
```

Listato 8: Funzione play del client

```
1  void play() {
2    PlayerStats giocatore = NULL;
3    int score, deploy[2], position[2], timer;
4    int turnoFinito = 0;
5    int exitFlag = 0, hasApack = 0;
6    while (!exitFlag) {
7      if (serverCaduto())
8        serverCrashHandler();
9      if (read(socketDesc, grigliaDiGioco, sizeof(grigliaDiGioco)) < 1)
10       printf("Impossibile comunicare con il server\n"), exit(-1);
11     if (read(socketDesc, deploy, sizeof(deploy)) < 1)
12       printf("Impossibile comunicare con il server\n"), exit(-1);
13     if (read(socketDesc, position, sizeof(position)) < 1)
14       printf("Impossibile comunicare con il server\n"), exit(-1);
15     if (read(socketDesc, &score, sizeof(score)) < 1)
16       printf("Impossibile comunicare con il server\n"), exit(-1);
17     if (read(socketDesc, &hasApack, sizeof(hasApack)) < 1)
18       printf("Impossibile comunicare con il server\n"), exit(-1);
19     timer = getTimer();
20     giocatore = initStats(deploy, score, position, hasApack);
21     printGrid(grigliaDiGioco, giocatore);
22     char send = getUserInput();
23     write(socketDesc, &send, sizeof(char));
24     read(socketDesc, &turnoFinito, sizeof(turnoFinito));
25     if (turnoFinito) {
26       system("clear");
27       printf("Turno finito\n");
28       sleep(1);
29     } else {
30       if (send == 'e' || send == 'E')
31         printf("Disconnessione in corso...\n"), exit(0);
32       if (send == 't' || send == 'T')
33         printTimer();
34       else if (send == 'l' || send == 'L')
35         printPlayerList();
36     }
37   }
38 }
```

Listati