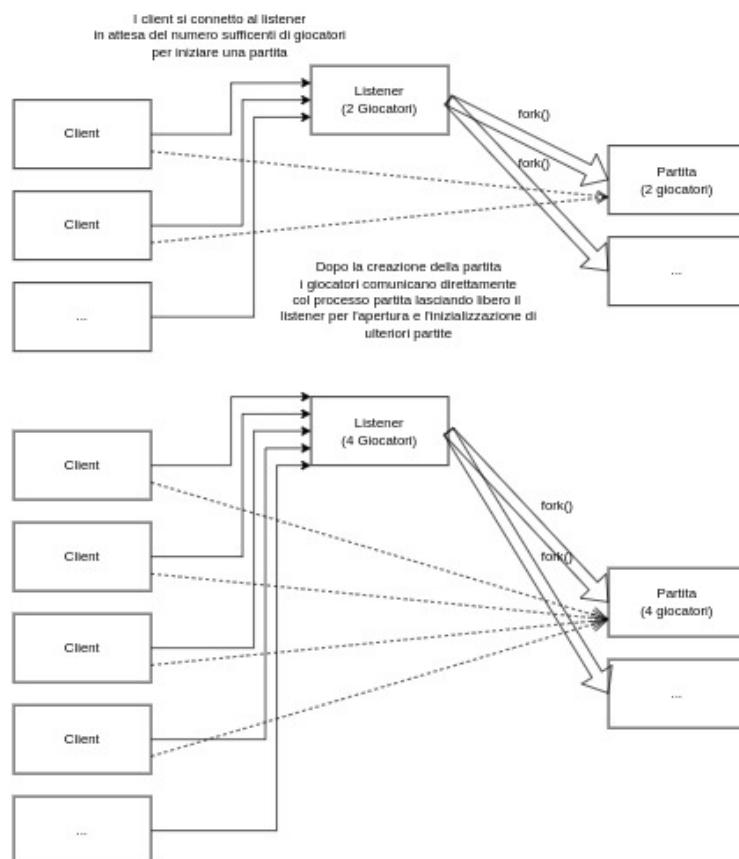




Battaglia Navale Multigiocatore

Di Christian Cecili e Cetorelli Mirko



Descrizione Generale

Nello svilupparsi dell'esecuzione è possibile individuare le tre principali entità in gioco:

- **Client**: Il device del giocatore che desidera partecipare ad un Game di battaglia navale;
- **Listener**: Colui che ha il ruolo di attendere l'arrivo del numero di giocatori necessario a poter avviare partite. Il listener è in grado di essere avviato in modo tale da poter gestire partite da 2 o 4 giocatori;
- **Game**: La specifica istanza di gioco che gestisce l'evolversi del gioco per i 2 o 4 giocatori ad essa associata.

Le dinamiche di interazione tra i 3 soggetti sono le seguenti:

Client → Listener: il client si connette al listener corrispondente in base a se desidera partecipare ad un Game da 2 o 4 giocatori

Listener → Game: quando al listener si saranno connessi un numero sufficiente di client, esso avvierà una nuova Game alla quale parteciperanno i giocatori che si sono connessi.

Successivamente il listener torna in ascolto di connessioni client in attesa di poter far partire un altro Game.

Client → Game: una volta che il Game è iniziato, i client non comunicano più in alcun modo con il listener. Le future interazioni (ovvero quelle strettamente legate la gioco) avvengono direttamente tra il Client e la specifica istanza di Game alla quale stanno partecipando.

Al completamento della partita, i Client e la specifica istanza di Game vengono opportunamente terminati.

Struttura

Entrando in dettaglio, i servizi precedentemente descritti sono stati implementati con una struttura a 2 strati, in modo da permettere una concorrenza nello sviluppo del sistema ed una modularità che garantisce una semplicità nella modifica e/o aggiunta di funzionalità future. Di seguito vi sarà una spiegazione dettagliata dei layers coinvolti.

Hosting Layer

L'Hosting Layer è lo strato del sistema che si occupa di ricevere connessioni di un numero arbitrario di giocatori che desiderano partecipare ad un Game, a

prescindere dalle modalità di svolgimento dello specifico gioco.

Questo strato è composto da tanti processi listener, ciascuno in grado di gestire una specifica configurazione delle partite.

Allo stato corrente, il listener può essere istanziato due volte per gestire due diverse modalità di gioco: Game da 2 giocatori e Game da 4 giocatori.

Considerata la struttura del software, è importante evidenziare quanto sia semplice aggiungere ulteriori configurazioni di questo tipo. La suddivisione in strati, infatti, delega la gestione delle partite a un livello specifico (il *Game Layer*), che utilizza come parametri le configurazioni passate dal Listener alla particolare istanza di gioco. Questo evidenzia come la struttura sia progettata per favorire la modularità nelle operazioni.

E' possibile decomporre il funzionamento di questo strato nelle seguenti macro operazioni:

▼ Configurazione del socket in ascolto

```
int setup_listening_socket(int no_port, int queue_len){
    int sockfd = -1;
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0){
        printf("inizializzazione listening socket fallita\\
exit(EXIT_FAILURE);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(no_port);

    if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof
        printf("inizializzazione listening socket fallita\\
exit(EXIT_FAILURE);
    }

    listen(sockfd, queue_len);
```

```
    return sockfd;  
}
```

All'esecuzione del processo listener la prima operazione effettuata sarà la configurazione di un socket di ascolto con le seguenti caratteristiche:

- Socket basato sul protocollo IPv4 (`AF_INET`)
- Comunicazione basata sull streaming di byte (`SOCK_STREAM`)
- In ascolto su tutte le interfacce di rete (`INADDR_ANY`)
- La porta di ascolto è proprio il parametro discriminante con il quale distinguiamo le diverse configurazioni di partite e di conseguenza listener.

Se il listener è avviato al fine di gestire una configurazione di partite da 2 giocatori la sua porta sarà `2222`, altrimenti (configurazione di partite da 4 giocatori) la sua porta sarà `4444`.

```
int main(int argc, char *argv[]) {  
    if(argc < 2) {  
        printf("utilizzo: listener <DIMENSIONE PARTITE: 2  
        exit(EXIT_FAILURE);  
    }  
    int num_player = *argv[1] - '0';  
    int no_port;  
    switch(num_player){  
        case 2:  
            no_port = 2222;  
            break;  
        case 4:  
            no_port = 4444;  
            break;  
        default:  
            printf("dimensione partite invalida, dimensio  
            exit(EXIT_FAILURE);  
    }  
  
    int sockfd = setup_listening_socket(no_port, num_player);  
}
```

```

while(1){
    int players_fd[num_player];
    wait_game_players(num_player, players_fd, sockfd);
    store_as_env(num_player, players_fd);
    init_game_creation_thread(num_player);
}
}

```

▼ Attendo la connessione di N giocatori (in base al configurazione del Listener)

Successivamente al completamento della configurazione del socket di ascolto, il listener si pone effettivamente in ascolto di connessioni in arrivo. Alla ricezione di una nuova connessione, il listener verifica se la connessione è stata instaurata correttamente, comunica al client un identificativo numerico a lui associato e modifica lo stato interno del Client in modo da porlo in attesa degli altri giocatori.

```

void wait_game_players(int num_player, int *players_fd, int
    socklen_t clilen;
    int num_conn = 0;
    while (num_conn < num_player) {
        players_fd[num_conn] = accept(sockfd, NULL, NULL);
        if (players_fd[num_conn] < 0){
            printf("errore durante la connessione di un giocatore");
            exit(EXIT_FAILURE);
        }

        write(players_fd[num_conn], &num_conn, sizeof(int));
        num_conn++;
        if (num_conn < (num_player-1)) {
            for(int i = 0; i < num_conn; i++){
                int n = write(players_fd[i], MSG_WAIT_FOR_PLAYER);
                if (n < 0){
                    printf("errore durante la connessione di un giocatore");
                    exit(EXIT_FAILURE);
                }
            }
        }
    }
}

```

```
    }  
}
```

▼ Persistenza dei file descriptor delle connessioni client tra le variabili di ambiente

Ogni volta che un client si connette, si verifica la creazione di un socket di connessione che ha il ruolo di canale tra il processo in grado di far uso di tale socket e lo specifico client connesso, perciò al completamento del processo di connessione avremo N file descriptor, ciascuno che descrive un canale con uno specifico client.

La strategia adottata per passare questi file descriptor al processo Game successivamente istanziato si basa sulla creazione di specifiche variabili di ambiente, nella quali mappiamo all'identificativo assegnato al client il file descriptor del canale socket che ci porta a tale client.

Questo approccio ci garantisce una semplicità implementativa superiore nonché una scalabilità superiore al fronte dell'introduzione di configurazioni di processi Game ulteriori

```
void store_as_env(int num_player, int *players_fd){  
    for(int i = 0; i < num_player; i++){  
        char name[20];  
        char fd[20] = {0};  
        sprintf(name, "%d", i);  
        sprintf(fd, "%d", players_fd[i]);  
        setenv(name, fd, 1);  
    }  
}
```

▼ Creazione del processo Game

una volta che tutti i giocatori richiesti per la creazione di una Game si sono connessi, il listener predisponde una procedura in 2 passaggi per l'inizializzazione della Game:

1. Creazione di un thread al quale delegare il processo di creazione

La prima cosa che fa è creare un thread secondario a cui delegare la responsabilità di creazione della partite, in modo tale che il thread

principale concentrati la sua computazione strettamente sull'ascolto di connessioni in arrivo, massimizzandone le prestazioni.

```
void init_game_creation_thread(int thread_arg){  
    pthread_t thread;  
    int *arg = malloc(sizeof(int));  
    *arg = thread_arg;  
    int result = pthread_create(&thread, NULL, init_game,  
        if(result) {  
            printf("errore durante l'inizializzazione del thread");  
            exit(EXIT_FAILURE);  
        }  
        pthread_join(thread, NULL);  
        free(arg);  
}
```

2. Istanziamento del processo Game

Quello che farà il thread secondario è eseguire una fork, la quale crea una copia dell'address space del listener, il quale subito dopo viene sovrascritto dall'address space del processo Game, portando perciò alla creazione di due processi separati (legati esclusivamente dalla relazione padre figlio che sussiste tra i due e dalle informazioni che quest'ultimo ha ereditato dal padre).

il thread secondario ancora in esecuzione termina portando alla distruzione del thread stesso, lasciando in esecuzione solo il thread principale (il thread di ascolto)

```
void *init_game(void *num_player){  
    if(fork() == 0){  
        char argv[2];  
        sprintf(argv, "%d", *(int *)num_player);  
        execl("./target/", "./game", &argv, NULL);  
        exit(EXIT_FAILURE);  
    }  
    pthread_exit(NULL);  
}
```

Game Layer

Il Game Layer è lo strato del sistema che si occupa dello svilupparsi della singola partita.

Questo strato è composto da tanti processi Game, dove ciascuno di essi comunica con i client che stanno partecipando a quella specifica istanza di Game.

E' possibile decomporre il funzionamento di questo strato nelle seguenti macro operazioni:

▼ Recupero dei file descriptor predisposti dal listener e inizializzazione dell'ambiente di gioco

La Prima cosa che il processo Game fa all'esecuzione è recuperare i descrittori dei canali socket dei giocatori della Game dalle variabili di ambiente, successivamente inizializza a 0 le strutture usate per la gestione del gioco, tra le quali spiccano:

- **Placement grid:** questo è un array di matrici, nelle quali il processo Game memorizzerà i posizionamenti delle navi scelti da ciascun giocatore, essa sarà la griglia alla quale si farà riferimento per verificare se si ha colpito un giocatore o meno.
- **Game grid:** anch'essa è un array di matrici, il cui ruolo però è di tenere traccia delle mosse subite da ciascun giocatore dagli altri, al fine di poter prevenire l'attacco di celle già attaccate, tenere traccia delle navi parzialmente affondate e offrire ai giocatori la possibilità di consultare lo stato della partita.
- **Dead:** è una wildcard utilizzata per assicurare che vengano inclusi esclusivamente i giocatori ancora in vita nel contesto di gioco. Questo meccanismo consente di filtrare automaticamente i partecipanti, escludendo quelli che non sono più attivi, e garantendo che tutte le operazioni si concentrino solo sugli utenti attivi, senza necessità di ulteriori controlli.

▼ Inserimento delle navi nella propria griglia per ciascun giocatore

All'inizio della partita il sistema richiederà il popolamento da parte di ciascun giocatore della propria placement grid, in modo da poter successivamente iniziare a giocare.

L'inserimento avviene in modo sequenziale per ciascun giocatore, durante questa fase verrà mostrato lo stato corrente della placement grid del

giocatore, indicata la nave che si deve inserire e si attende che il giocatore immetta la sequenza di celle che la nave dovrà occupare (per ogni inserimento il server prevede opportuni controlli qualora la formattazione inserita non sia valida, le celle inserite o parte di esse siano già occupate, non si fornisca una numero sufficiente di celle oppure se ne fornisca un numero eccessivo).

```
//SERVER PROCESS
void *run_game(int *cli_sockfd) {
    char msg[4];
    printf("i giocatori stanno riempiendo le griglie...\n");
    write_clients_msg(cli_sockfd, MSG_GAME_START);
    for(int cur = 0; cur < PLAYERS_NUM; cur++){
        notify_active_player(cli_sockfd, cur, MSG_FILL_GRID, MSG_DRAW_GRID);
        recv_grid(players_placement_grid[cur], cli_sockfd[cur]);
        draw_grid(players_placement_grid[cur], 0, cur);
        fillBoatArray(cur);
        printf("la griglia del giocatore %d e' stata riempita\n");
    }
    .....
}

//CLIENT PROCESS
void fill_data(SOCKET sockfd) {
    int player_grid[GRID_SIZE][GRID_SIZE]; void *run_game(int *cli_sockfd);
    char msg[4];
    printf("i giocatori stanno riempiendo le griglie...\n");
    write_clients_msg(cli_sockfd, MSG_GAME_START);
    for(int cur = 0; cur < PLAYERS_NUM; cur++){
        notify_active_player(cli_sockfd, cur, MSG_FILL_GRID, MSG_DRAW_GRID);
        recv_grid(players_placement_grid[cur], cli_sockfd[cur]);
        draw_grid(players_placement_grid[cur], 0, cur);
        fillBoatArray(cur);
        printf("la griglia del giocatore %d e' stata riempita\n");
    }
    initialize_grid(player_grid);

    const char *torpedo = "Torpediniera";
    const char *submarine = "Sottomarino";
```

```

const char *aircraft_carrier = "Portaerei";

draw_grid_placement(player_grid);
printf("\nPer iniziare, posiziona il tuo %s (%d caselle).
printf("Scegli le caselle che vuoi riempire (es. 0D-0E
place_ship(player_grid, torpedo, 2, 1);
for (int i = 0; i < 2; i++) {
    draw_grid_placement(player_grid);
    printf("\nBene, ora posiziona il %s %d. (es. 2D-2E
    place_ship(player_grid, submarine, 3, i + 2);
}
draw_grid_placement(player_grid);
printf("\nInfine, posiziona il tuo %s (es. 8A-8B-8C-8D
place_ship(player_grid, aircraft_carrier, 4, 4);

printf("Perfetto! La tua griglia è stata riempita correttamente.
write_server_data(sockfd, player_grid);
draw_legends();
draw_grid_placement(player_grid);
}

```

▼ Gestione turno giocatore

Dopo l'inserimento delle navi, il gioco vero e proprio inizia assegnando il turno iniziale al primo giocatore.

Un turno di un giocatore è l'insieme di molteplici fasi che culminano nell'attacco di uno degli altri giocatori.

Le fasi del turno sono le seguenti:

- 1. Elenco dei giocatori ancora “vivi” (non eliminati)**

La prima cosa che il sistema propone è un elenco dei giocatori correntemente in vita, e di conseguenza attaccabili

- 2. Immissione dell'utente**

Subito dopo l'elenco viene chiesto al giocatore come desidera procedere, difatti sono previste 3 operazione, discriminate in base al tipo e numero di informazioni fornite in input:

a. <ID GIOCATORE>

Se si fornisce esclusivamente l'identificativo di uno dei giocatori vivi, o il proprio identificativo, il sistema restituirà lo stato corrente della game grid del giocatore inserito, in modo da poter analizzare la propria situazione e quella degli altri.

b. <ID GIOCATORE> <COORDINATA DA ATTACCARE>

Se si fornisce l'identificativo di un giocatore vivo (diverso da te stesso) seguito da una mossa, il sistema interpreterà l'istruzione come l'intenzione di attaccare il giocatore con tale identificativo nella cella specificata

```
void take_turn(SOCKET sockfd, int my_id) {
    const char letters[10] = {'A', 'B', 'C', 'D', 'E',
    char buffer[100];
    int square[3];

    display_alive_players(sockfd);
    printf("Scrivi il numero del giocatore per vedere le mosse disponibili\n");
    printf("Per attaccare scrivere il numero del giocatore e la coordinata\n");

    while (1) {
        if (fgets(buffer, sizeof(buffer), stdin) == NULL)
            error("Errore durante input");
    }
    buffer[strcspn(buffer, "\n")] = 0;

    char *player_token = strtok(buffer, " ");
    int player_id = parse_player_id(player_token);
    if (player_id < 0) {
        continue;
    }

    char *coordinate_token = strtok(NULL, " ");
    if(coordinate_token != NULL && player_id == my_id) {
        printf("Non puoi attaccare te stesso.\n");
        continue;
    }
}
```

```
        handle_user_choice(sockfd, player_id, coordinates);
        recv_and_print_player_game_grid(sockfd);
        break;
    }
}
```

3. Notifica esito mossa

Successivamente all'immissione di un attacco verso un altro giocatore, il sistema verificherà se l'attacco ha colpito una nave o meno (confrontando la posizione con la placement grid dell'attaccato), e notificherà a tutti i giocatori l'esito dell'attacco.

Se l'attacco è stato un successo, al giocatore attaccante sarà offerta la possibilità di attaccare nuovamente lo stesso o un altro giocatore, in caso contrario il turno passa al giocatore successivo in modo ciclico.

▼ Condizioni di completamento della Game

Quando tutte le navi di un giocatore sono eliminate, il giocatore è considerato morto (eliminato), ma ciò non porta alla conclusione della partita, bensì previene semplicemente il giocatore dall'interagire ulteriormente con la partita in corso, disconnettendolo.

La partita termina quando rimane vivo 1 solo giocatore, il quale sarà incoronato vincitore.