Laboratorio di Sistemi Operativi - Prof. Finzi

Traccia C

Libutti Simone N86002928

Trinchillo Giovanni N86003133

Luglio 2021

Contents

1 Guida d'uso

1.1 Guida d'uso per il server

Il server è avviabile tramite l'apposito bash script `compila.sh`, che prende come parametro il numero di giocatori per partita `N`, da cui dipendono le dimensioni della griglia e il numero di territori che ogni giocatore dovrà conquistare. Lo script compila il codice del server con il comando `gcc -pthread -o server server.c llist.c funzioni.c` e lo avvia con parametro `N` tramite il comando `./server N`

Il server risiede su di una macchina virtuale Azure modello Standard B1 (1 CPU, 1GiB RAM) con sistema operativo Linux (Ubuntu 20.04).

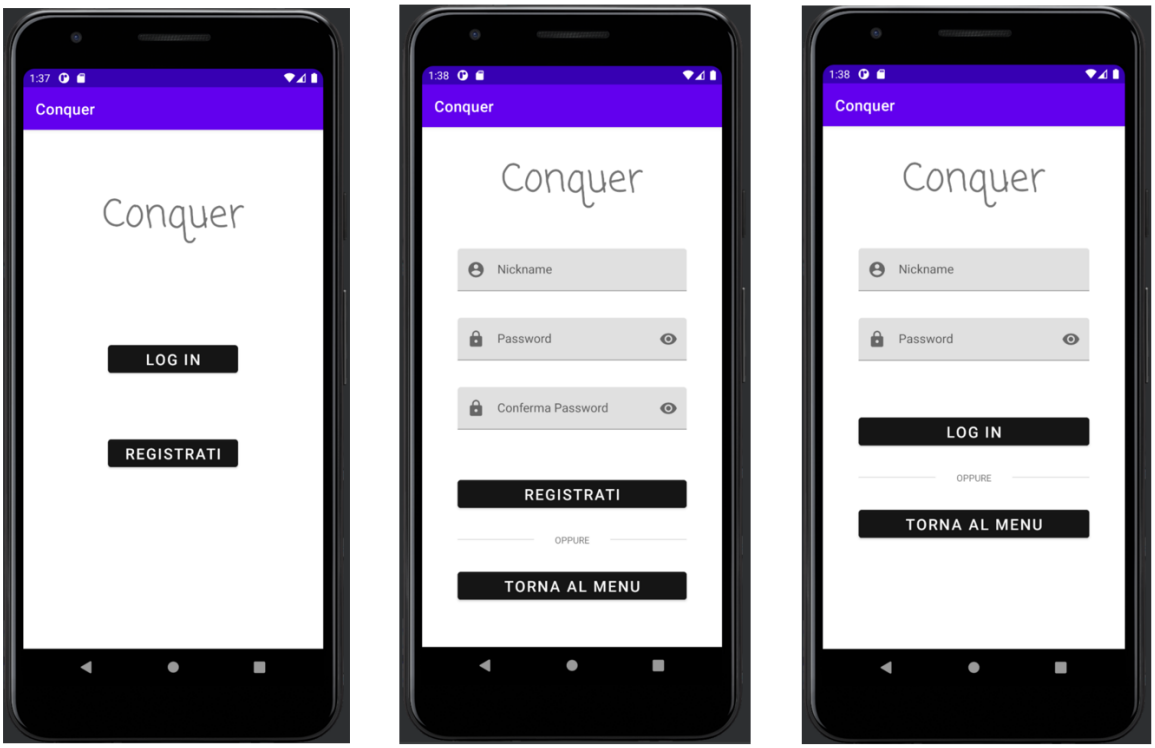
All'avvio del server verranno aperti due file: nel primo, **utenti.txt**, saranno conservati i dati degli utenti nella forma di righe "nickname password", mentre nel secondo, **eventi.txt**, saranno loggati i vari eventi riguardanti le interazioni degli utenti con il server (connessioni, disconnessioni, registrazioni e accessi). Per ogni partita verrà inoltre aperto un file ad-hoc per poter riportare la data e l'ora delle conquiste e l'esito.

1.2 Guida d'uso per il client

Il primo passo per effettuare l'installazione del client è quello di abilitare l'installazione di applicazioni da sorgenti sconosciute sul proprio dispositivo Android (effettuabile dalle **impostazioni** del dispositivo, solitamente nella sezione dedicata alle app, o in quella dedicata alla sicurezza).
Fatto ciò sarà possibile installare il client tramite il file .apk fornito; il client possiede già le informazioni necessarie per connettersi al server.

Una volta installato il client si verrà connessi automaticamente al server. Qualora dovesse verificarsi un errore di connessione sarà necessario premere l'apposito pulsante per tentare la riconnessione:

Una volta connessi sarà possibile effettuare la registrazione, e successivamente il log-in:



Dopo aver effettuato il log-in saranno disponibili due opzioni: avviare la ricerca di una partita o visualizzare gli utenti attivi:

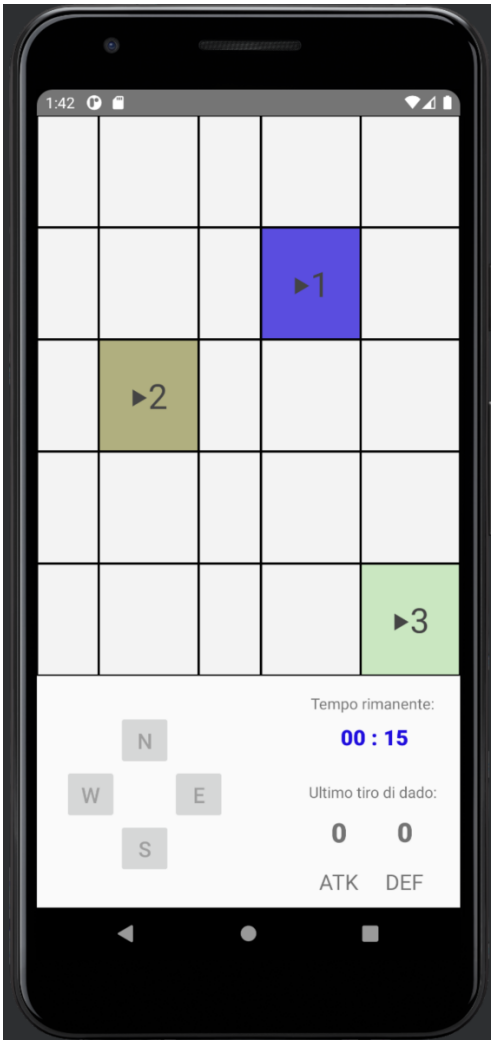
La prima opzione, se selezionata, metterà il client in attesa che si raggiunga il numero di giocatori necessario a far iniziare una partita, numero indicato come parametro al momento dell'avvio del server, come specificato nella sezione 2.1. Una volta raggiunto tale numero, la partita comincerà automaticamente e verrà assegnata a ogni giocatore una posizione casuale sulla griglia di gioco.

I giocatori vengono rappresentati sulla griglia con un numero e una freccia per indicare la loro posizione, e ciascuno di essi avrà un colore. Le caselle del colore di un giocatore sono di proprietà del suddetto giocatore.

Ogni giocatore ha 30 secondi per il proprio turno, in cui potrà scegliere di muoversi su un'altra casella (tramite pulsanti N, S, E, W, che rappresentano le 4 direzioni) e tentare di conquistarla. Se la casella è libera viene conquistata, altrimenti si effettuerà un tiro di dadi contro il giocatore che la possiede per determinare se verrà o meno conquistata. Il primo giocatore a conquistare un numero prestabilito di territori vince la partita.

Il client riporta anche un dettagliato log di ogni evento di gioco, tra cui le azioni effettuate dai giocatori durante la partita e il loro effetto, le scadenze dei timer e i nomi dei giocatori che hanno abbandonato la partita.

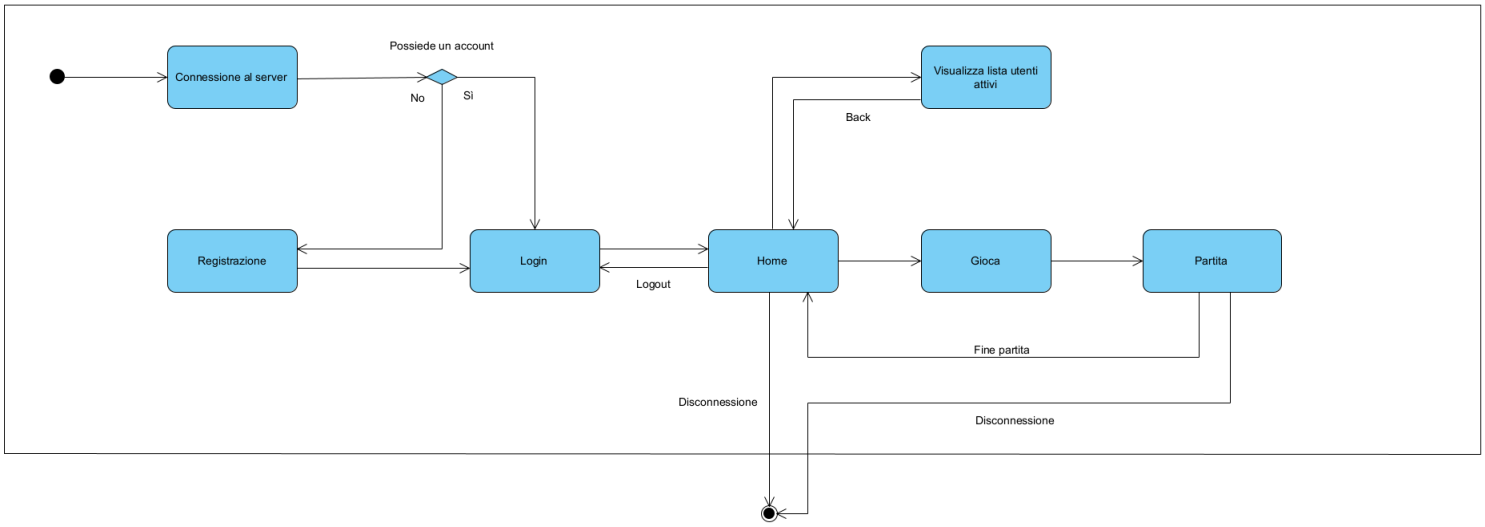
Di seguito un esempio di schermata di una partita:



Al termine della partita, ovvero quando un giocatore ha conquistato il numero di territori prestabilito, viene mostrato un messaggio di vittoria con il nome del giocatore vincente, e si viene riportati alla schermata principale, da cui è possibile cercare una nuova partita.

La seconda opzione disponibile dal menu principale è quella di visualizzare una lista di giocatori attualmente connessi, come mostrato di seguito:

Per rappresentare in maniera concisa la struttura dell'interfaccia utente è stato prodotto il seguente state chart:



2 Protocollo di comunicazione Client-Server

2.1 Lato server

All'avvio, il server si occupa di aprire una socket di ascolto su porta predefinita (50000) tramite la chiamata `socket()`; il dominio del canale di comunicazione è *AF_INET* (famiglia TCP/IP con IPv4) e il tipo è *SOCK_STREAM*, che garantisce sequenze ordinate bidirezionali di byte. La socket così creata si occuperà di accettare le connessioni che le arrivano da qualunque indirizzo IP (tramite l'impiego di *INADDR_ANY*).

```c
// Creazione della socket
if ((listener_socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("Creazione della socket fallita");
    exit(EXIT_FAILURE);
}
printf("Socket di ascolto creata...\n");

// Impostazione delle opzioni di riutilizzo dell'indirizzo e della porta (prevenzione)
if (setsockopt(listener_socket_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &options, sizeof(options))) {
    perror("Errore durante l'impostazione delle opzioni della socket");
    exit(EXIT_FAILURE);
}
printf("Opzioni socket impostate...\n");

address.sin_family      = AF_INET;                // Impostazione della famiglia di indirizzi a IPv4
address.sin_addr.s_addr = htonl(INADDR_ANY);      // Impostazione degli indirizzi ammissibili (tutti)
address.sin_port        = htons(PORT);            // Impostazione della porta
```

Per ogni `accept()` effettuata correttamente verrà creato un nuovo client e un thread ad esso associato, e verrà usata la nuova socket generata per la comunicazione con quel client tramite il suo file descriptor; tutti i client saranno quindi serviti in maniera concorrente:

```c
while(1) {

    // Accetta un nuovo client
    new_socket = accept(listener_socket_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
    if (new_socket < 0) {
        perror("Errore di accept");
        exit(EXIT_FAILURE);
    }

    printf("Client %d trovato.\n\n", new_socket);

    client = createClient(clientnum, new_socket);

    log_ClientConnection(new_socket, address.sin_addr);

    pthread_mutex_lock(&clients_lock);
    clients = append(clients, client);
    pthread_mutex_unlock(&clients_lock);

    pthread_create(&client->thread, NULL, clientThread, client);

    clientnum++;

}
```

La comunicazione dal server al client avviene tramite le system call `read`, `write`, `recv`, e `send`.
Maggiori dettagli in merito a queste ed altre system call adoperate nella creazione del server saranno forniti nelle sottosezioni dedicate.

2.2 Lato client

Lato client, una apposita classe di servizio `ConnectionHandler` si occupa di gestire l'avvio della connessione al server e lo scambio di messaggi.
In dettaglio, si utilizza la classe `Socket` del pacchetto **java.net** per istanziare una socket adibita alla connessione. Di ciò si occupa il metodo `startConnection()`, che invoca `connect(InetAddress address, int timeout)` sulla socket appena creata. Se la connessione ha successo, si inizializzano gli attributi `in` e `out` della classe con, rispettivamente, l'`InputStream` e l'`OutputStream` della socket. Per essere precisi, l'`OutputStream` è avvolto da un `PrintWriter` per la formattazione del flusso di byte e l'`InputStream` da un `BufferedReader` per garantire

bufferizzazione dei byte letti.

Di seguito vengono mostrati gli attributi della classe `ConnectionHandler` e il codice della funzione di avvio della connessione appena descritta.

```java
public class ConnectionHandler {

    private static ConnectionActivity connectionActivity;

    private static final String SERVER_IP = "20.203.137.149";         //IP di Azure
    //private static final String SERVER_IP = "192.168.1.75";             //IP di test
    private static final int SERVER_PORT = 50000;

    public static final String CONNECTION_ERROR_MESSAGE = "Errore di connessione";

    private static Socket clientSocket;
    private static PrintWriter out;
    private static BufferedReader in;


    public static boolean startConnection () {

        clientSocket = new Socket();
        try {
            clientSocket.connect(new InetSocketAddress(SERVER_IP, SERVER_PORT),  timeout: 5000);
            out = new PrintWriter(clientSocket.getOutputStream(),  autoFlush: true);
            in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            return true;
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }

    }
```

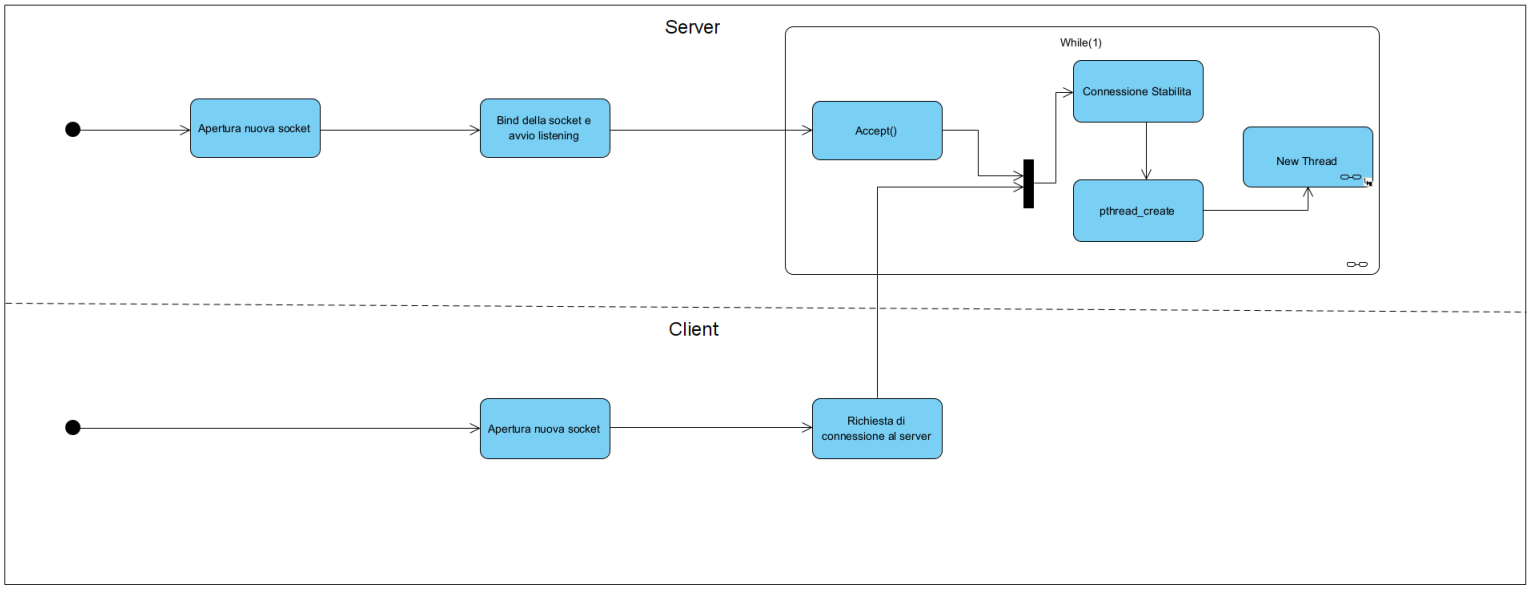
Gli attributi `in` e `out` della classe vengono utilizzati rispettivamente per ricevere ed inviare messaggi. Sono privati e vengono utilizzati all'interno delle funzioni ad-hoc, `read()` e `write(String line)` mostrate di seguito, le quali vengono invocate all'interno dell'applicazione dai controllori nel momento in cui è necessario comunicare col server:

```java
public static String read () throws IOException {
    if (in != null) {
        return in.readLine();
    }
    else {
        return null;
    }
}

public static void write (String line) {
    if (out != null) {
        out.print(line);
        out.flush();
    }
}
```

È importante fare caso al fatto che, come si potrà notare leggendo il codice sorgente, tutte le operazioni di I/O sono svolte in thread separati, in modo da prevenire l'insorgere di `NetworkOnMainThreadException`.

Per maggiore chiarezza, nel seguente activity diagram è stato sintetizzato il meccanismo di connessione tra un client e il server fase per fase, dall'avvio alla creazione del client:


Server
While(1)
Apertura nuova socket
Bind della socket e avvio listening
Accept()
Connessione Stabilita
New Thread
pthread_create
Client
Apertura nuova socket
Richiesta di connessione al server

3 Dettagli implementativi

In questa sezione verranno esplicitati i dettagli della struttura del software lato server e lato client, con particolare attenzione riposta nei riguardi delle chiamate di sistema Linux. Per sopperire ad eventuali dettagli mancanti si rimanda al codice sorgente integrale.

3.1 Gestione dei client

I client sono rappresentati nel server tramite delle apposite `struct` **Client** contenenti le principali informazioni ad essi relative, che fungeranno da argomento della funzione di thread:

```
typedef struct Client {
    int id;
    char nickname[32];
    int socket;
    int pipe[2];
    int activeGame;
    int positionInArrayOfPlayers;
    pthread_t thread;
} Client;
```

Come si è osservato in sezione 3.1, a ciascuna `accept()` segue la creazione di un client e il suo inserimento all'interno di una lista globale `clients` che tiene traccia dei client attualmente collegati. La funzione di creazione del client, oltre ad inizializzare alcuni valori e impostare a 630 secondi il timeout della socket, contiene un'invocazione della system

call `pipe()` sul campo `int pipe[2]` della struttura:

```
Client* createClient (int id, int socket) {

    Client* client;
    int p[2];
    struct timeval tv;

    client = (Client*)malloc(sizeof(Client));
    client->id = id;
    client->socket = socket;
    client->activeGame = -1;
    client->positionInArrayOfPlayers = -1;

    memcpy(client->pipe, p, sizeof(client->pipe));
    if (pipe(client->pipe) < 0) {
        perror("Errore di pipe");
        exit(EXIT_FAILURE);
    };

    tv.tv_sec = 630;
    tv.tv_usec = 0;

    if (setsockopt(socket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof tv)) {
        perror("Errore nell'impostazione del timeout della socket");
        exit(EXIT_FAILURE);
    };

    return client;

}
```

Questa pipe viene utilizzata dal thread del client per comunicare con il thread della partita in cui esso si trova, come verrà rimarcato nella sottosezione dedicata. Dopo l'inserimento viene avviata la funzione di thread di quel client con esso stesso come argomento di thread.

La funzione di thread ha la seguente struttura:

```
while (TRUE) {

    if (recv(this->socket, clientMessage, sizeof(clientMessage), 0) <= 0) {…}

    action = clientMessage[0] - '0';   // Converte char a cifra singola nel rispettivo int

    switch (action) {
        case SIGNUP:…
        break;

        case LOGIN:…
        break;

        case VIEW_ACTIVE_USERS:…
        break;

        case LOOK_FOR_MATCH:…
        break;

        case STOP_LOOKING_FOR_MATCH:…
        break;

        case GAME_ACTION:…
        break;

        case LEAVE_MATCH:…
        break;

        case LOGOUT:…
        break;

        default:…
        break;
    }

    memset(clientMessage, 0, sizeof(clientMessage));

}
```

Il ciclo infinito fa in modo tale che il server sia perennemente in lettura su ciascun client che si connette, lettura che avviene adoperando la system call `recv()`, specifica per le socket. Il primo carattere di ciascun messaggio ricevuto è

numerico, ed indica il tipo di azione da eseguire. Esso verrà gestito tramite un apposito costrutto `switch()`. I casi su cui soffermarsi sono *LOOK_FOR_MATCH* e *GAME_ACTION*.

```
case LOOK_FOR_MATCH:
    pthread_mutex_lock(&lookingForMatch_lock);
    clientsLookingForMatch = append(clientsLookingForMatch, this);
    pthread_mutex_unlock(&lookingForMatch_lock);

    if (length(clientsLookingForMatch) == NUMBER_OF_PLAYERS) {
        start_match();
    }
break;
```

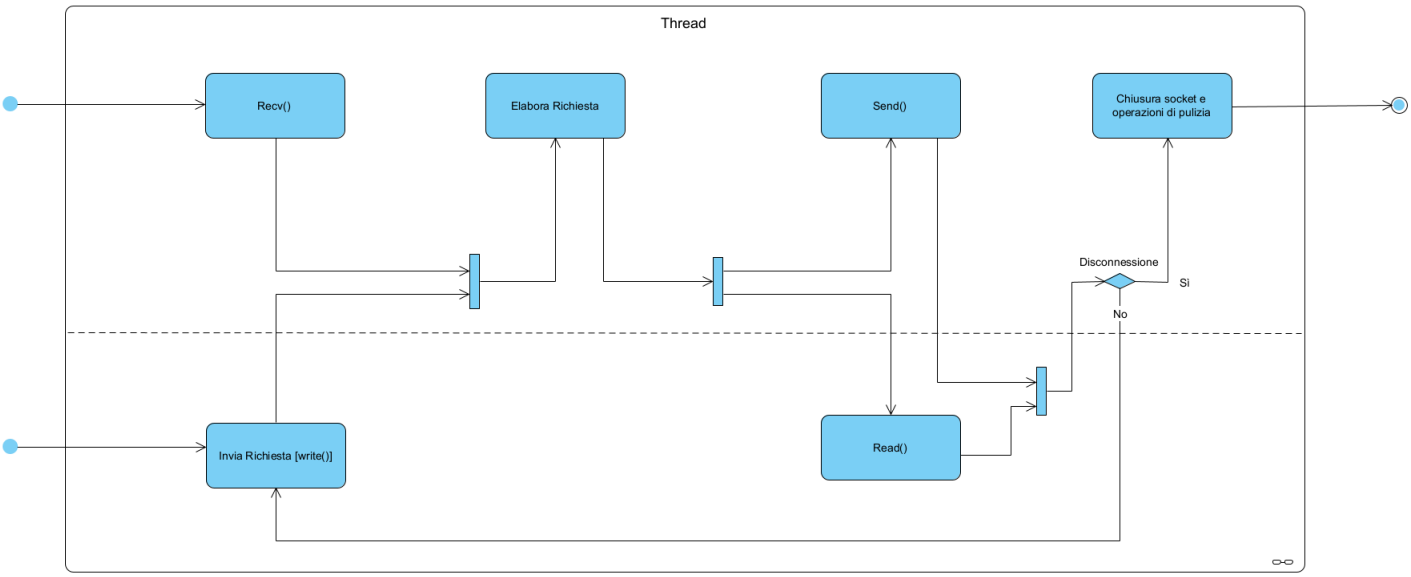
Nel caso di *LOOK_FOR_MATCH*, il server accoderà il client a una lista globale `usersLookingForMatch`. Dopo averlo fatto, controllerà se la lista contiene il numero di giocatori necessario a far iniziare una nuova partita. In caso affermativo, la funzione `start_match()` si occuperà della creazione e inizializzazione della partita, a cui seguirà l'avvio del thread di partita.

```
case GAME_ACTION:
    snprintf(clientMessage, sizeof(clientMessage), "%c", clientMessage[1]);
    if (write(this->pipe[1], clientMessage, strlen(clientMessage)) < 0) {
        perror("Errore di scrittura sulla pipe");
        exit(EXIT_FAILURE);
    }
break;
```

Per quanto riguarda *GAME_ACTION*, rappresenta il caso in cui l'utente invia un messaggio che costituisce un'azione di gioco, ovvero una mossa. Il secondo carattere del messaggio viene convertito in stringa e inviato, tramite la system

call `write()` sulla pipe del client, al thread della partita. Quest'ultimo lo elaborerà ed effettuerà le dovute operazioni.

Anche in questo caso è stato creato un activity diagram per specificare in che modo avviene il ciclo di comunicazione tra server e client fino all'eventuale disconnessione di quest'ultimo:



3.2 Registrazione

Quando un utente decide di effettuare una registrazione su di un client dopo essersi connesso, avverrà l'inserimento dei dati nell'apposita Activity, e la classe `AuthController` sarà incaricata di prelevarli ed invocare il metodo `addUser` di `UserDataHandler`, il quale li invierà al server tramite `ConnectionHandler.write()` :

```java
public static int addUser (String nick, String pswd) {

    String outcome;
    int outcome_int;

    ConnectionHandler.write( line: AuthController.SIGNUP + nick + SEPARATOR + pswd);
    try {

        outcome = ConnectionHandler.read();
        if (outcome == null) throw new IOException();
        outcome_int = Integer.parseInt(outcome);

        if (outcome_int == AuthController.SIGNUP_SUCCESS) {
            return AuthController.SIGNUP_SUCCESS;
        }
        else if (outcome_int == AuthController.USER_ALREADY_EXISTS) {
            return AuthController.USER_ALREADY_EXISTS;
        }

    } catch (IOException e) {
        ConnectionHandler.stopConnection();
        return AuthController.GENERIC_SIGNUP_FAILURE;
    }

    return AuthController.GENERIC_SIGNUP_FAILURE;

}
```

Il server, letto questo messaggio, avvierà la procedura `saveCredentialsToFile()` mediante la quale verranno estratte le credenziali da quest'ultimo, e salvate sul file degli utenti tramite la system call `write()` (a meno di un tentativo di registrazione di un utente già registrato) :

```c
int saveCredentialsToFile (char* credentialsBuffer, int socket) {

    strcat(credentialsBuffer, "\n");

    char credentials[64] = {0};
    char nickname[32] = {0};

    int i, nick_flag = TRUE, ret = TRUE;

    pthread_mutex_lock(&users_file_lock);

    for (i = 1; credentialsBuffer[i] != '\0'; i++) {}
    credentials[i] = '\0';

    if (userExists(nickname)) {}

    if (ret) {
        if (write(users_fd, credentials, strlen(credentials)) == -1) {
            perror("Errore di scrittura sul file degli utenti");
            exit(EXIT_FAILURE);
        }

        log_SignUp(nickname, socket);
    }

    pthread_mutex_unlock(&users_file_lock);

    return ret;

}
```

Dopo l'operazione, il server ne invierà l'esito al client, che tramite `ConnectionHandler.read()` lo riceverà e gestirà.

3.3 Log In

Per effettuare il log in, dopo la connessione, l'utente inserirà le proprie credenziali nella schermata di Log In. Anche qui `AuthController` sarà incaricata di gestire l'operazione tramite l'invocazione del metodo `logIn()` di `UserDataHandler` con le suddette credenziali come parametro. Nuovamente, esse verranno inviate al server utilizzando `ConnectionHandler.write()`

```
public static int logIn (String nick, String pswd) {

    String outcome;
    int outcome_int;

    ConnectionHandler.write( line: AuthController.LOGIN + nick + SEPARATOR + pswd);
    try {

        outcome = ConnectionHandler.read();
        if (outcome == null) throw new IOException();
        outcome_int = Integer.parseInt(outcome);

        if (outcome_int == AuthController.LOGIN_SUCCESS) {
            return AuthController.LOGIN_SUCCESS;
        }
        else if (outcome_int == AuthController.USER_DOES_NOT_EXIST) {
            return AuthController.USER_DOES_NOT_EXIST;
        }
        else if (outcome_int == AuthController.WRONG_PASSWORD) {
            return AuthController.WRONG_PASSWORD;
        }
        else if (outcome_int == AuthController.USER_ALREADY_CONNECTED) {
            return AuthController.USER_ALREADY_CONNECTED;
        }

    } catch (IOException e) {
        ConnectionHandler.stopConnection();
        return AuthController.GENERIC_LOGIN_FAILURE;
    }

    return AuthController.GENERIC_LOGIN_FAILURE;

}
```

L'operazione di log in lato server è effettuata dalla procedura `signIn()`, e consiste in primo luogo nell'estrarre le credenziali dal messaggio del client. Successivamente, viene impiegata la system call `lseek()` per impostare l'offset del file degli utenti in posizione iniziale, e in seguito `read()` con un buffer di dimensione 1 per leggere il file carattere per carattere, in modo da poter confrontare i dati forniti con quelli presenti sul file stesso. La struttura della procedura è la seguente:

```
int signIn (char* credentialsBuffer, Client* client) {

    strcat(credentialsBuffer, "\n");

    char nickname[32];
    char password[32];

    char nick_to_compare[32];
    char password_to_compare[32];

    int i, j = 0;
    int nick_flag = TRUE;

    for (i = 1; credentialsBuffer[i] != '\n'; i++) {…}
    password[j] = '\0';

    i = 0;
    j = 0;

    pthread_mutex_lock(&users_file_lock);

    if (lseek(users_fd, 0, SEEK_SET) < 0) {…}

    char read_char[1];
    int nickname_found = FALSE;
    int password_reached = FALSE;
    int ret = 1;

    while (read(users_fd, read_char, 1) > 0) {…}

    pthread_mutex_unlock(&users_file_lock);

    return ret;

}
```

L'esito del log in viene inviato al client dopo l'esecuzione della funzione. Se esso ha avuto successo, il client procederà a impostare l'utente come utente corrente e ad aggiornare l'interfaccia grafica, conducendolo alla schermata principale.

3.4 Utenti attivi

Lato server, si tiene traccia degli utenti attivi tramite una lista globale di stringhe, `activeUsersList`. Ciascun nodo della lista contiene al suo interno il nickname di un utente, e viene aggiornata ad ogni connessione e disconnessione.

Il client può fare richiesta di visualizzare questa lista mediante l'apposito pulsante, situato sulla schermata principale. Essa viene recuperata tramite il metodo `fetchActiveUsers()` di `UserDataHandler`, invocato da `ActiveUsersController`.

```
public static boolean fetchActiveUsers (List<String> activeUsers) {

    activeUsers.clear();

    ConnectionHandler.write( line: "3");
    String readVal;

    try {

        while (true) {
            readVal = ConnectionHandler.read();

            if (readVal == null) throw new IOException();
            if ("|".equals(readVal)) break;

            activeUsers.add(readVal);
        }

    }
    catch (IOException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

Come si può notare, questo metodo invia al server un messaggio per notificarlo del fatto che il client ha fatto richiesta di visualizzare gli utenti attivi, e si mette in attesa di riceverli tramite `ConnectionHandler.read()`. `ActiveUsersController` avrà poi il compito una volta ricevuti, di mostrarli in un'apposita `RecyclerView`

L'invio della lista da parte del server avviene con l'apposita procedura `sendUsersList()`, che itera su di essa inviando elemento per elemento al client.

```
void sendUsersList (Client* client) {

    pthread_mutex_lock(&activeUsers_lock);

    List* p = activeUsersList;
    char* nickname;
    int nick_size;

    while (p) {

        nick_size = strlen((char*)(p->data)) + 1;
        nickname = (char*)malloc(nick_size * sizeof(char));

        strcpy(nickname, (char*)(p->data));
        strcat(nickname, "\n");

        if (send(client->socket, nickname, nick_size, MSG_NOSIGNAL) < 0) {⇨}

        free(nickname);
        p = p->next;

    }
    if (send(client->socket, "|\n", 2, MSG_NOSIGNAL) < 0) {
        onClientDisconnection(client);
    }

    pthread_mutex_unlock(&activeUsers_lock);

}
```

3.5 Partite

Ciascuna partita è rappresentata, nel server, con una `struct` **Game**.

```c
typedef struct Game {
    int id;
    pthread_t thread;
    pthread_mutex_t nullPlayerLock;
    pthread_mutex_t gameFileLock;
    Symbol** grid;
    Player** players;
    Player* activePlayer;
    int file;
} Game;
```

Come specificato nella sottosezione 4.1, la funzione `start_match()` ha il compito di avviare la partita una volta raggiunto il giusto numero di giocatori, allocando memoria per un **Game** e avviando un thread di partita con quest'ultimo come argomento. Tutte le partite attive vengono conservate in una lista globale `games`. La funzione del thread di partita è strutturata in questo modo:

```
while (!winIsReached) {

    setNewActivePlayer(this, &activePlayerIndex);

    sendActivePlayerAndTimerEnd(this, activePlayerIndex);

    pthread_create(&timer, NULL, timerThread, this);

    // La lettura dell'azione di gioco avviene tramite pipe
    if (read(this->activePlayer->client->pipe[0], clientMessage, sizeof(clientMessage)) < 0) {…}

    pthread_cancel(timer);

    gameAction = clientMessage[0];
    switch (gameAction) {…}
    move = gameAction;

    if (squareIsOwnedByEnemy(this, new_x, new_y)) {
        handleSquareIsOwnedByEnemy(this, new_x, new_y, move);
    }
    else {
        if (squareIsOwnedBySelf(this, new_x, new_y)) {
            handleSquareIsOwnedBySelf(this, new_x, new_y, move);
        }
        else {
            handleSquareIsFree(this, new_x, new_y, move);
        }
    }

    if (this->activePlayer->territories == WIN) {
        printf("%s vince la partita\n\n", this->activePlayer->client->nickname);
        winIsReached = TRUE;
    }

    memset(clientMessage, 0, sizeof(clientMessage));

}
```

A ogni iterazione viene impostato il giocatore attivo, ovvero il successivo giocatore non nullo nell'array, e questa informazione viene inviata a tutti i client assieme all'istante di fine del timer di quel giocatore. Viene dunque avviato il timer, e quando il giocatore attivo deciderà di effettuare una mossa, essa verrà letta tramite la system call `read()` sulla pipe del client di quel giocatore e opportunamente gestita. Se la mossa ha determinato la vittoria del giocatore, la partita termina.

Per quanto riguarda il lato client, quando un giocatore preme sull'apposito pulsante di ricerca partita viene messo in lettura in attesa di raggiungere il numero di giocatori sufficenti per iniziare. Raggiunto tale numero, il server notificherà tutti i client dell'inizio, i quali riceveranno i dati dei giocatori, ed avvieranno un ciclo quasi identico a quello riscontrato nel thread di partita del server:

```
while (!winIsReached) {

    try {

        serverMessage = ConnectionHandler.read();

        if (serverMessage == null) throw new IOException();

    }
    catch (IOException e) {...}

    serverMessage_array = serverMessage.toCharArray();

    event = serverMessage_array[0];

    switch (event) {

        case RECEIVE_ACTIVE_PLAYER_AND_TIME:
            cancelTimer(timer);
            oldActivePlayer_index = activePlayer_index;
            activePlayer_index = Integer.parseInt(serverMessage.substring(1, 3));
            timerEnd = Long.parseLong(serverMessage.substring(3));
            clearGridOfDeserters(oldActivePlayer_index, activePlayer_index);
            setActivePlayerAndButtonsAndTime(activePlayer_index, timerEnd);
        break;

        case SUCCESSFUL_ATTACK:
            cancelTimer(timer);
            handleSuccessfulAttack(serverMessage);
        break;

        case FAILED_ATTACK:
            cancelTimer(timer);
            handleFailedAttack(serverMessage);
        break;

        case MOVE_ON_OWN_SQUARE:
            cancelTimer(timer);
            handleMoveToFreeOrOwnSquare(serverMessage,  free: false);
        break;
        case MOVE_ON_FREE_SQUARE:
            cancelTimer(timer);
            handleMoveToFreeOrOwnSquare(serverMessage,  free: true);
        break;

        case TIME_ENDED:
            if (activePlayer.getNickname().equals(AuthController.getCurrUser())) {
                activity.runOnUiThread(() -> Toast.makeText(activity,  text: "Tempo scaduto!", Toast.LENGTH_SHORT).show());
            }
            activity.log(activePlayer.getColor(), EVENT_LOGMSG_SIZE,  time: true,  text: " Tempo scaduto per " + activePlayer.getNickname() + ".",  vSpace: 2);
        break;

        case MATCH_LEFT:
            cancelTimer(timer);
            clear();
            goToMainActivity();
        return;

        default:
        return;

    }
}
```

Come si può notare, il ciclo legge un messaggio da parte del server, che consiste nell'esito della mossa effettuata, nella scadenza di un timer o nell'abbandono della partita, ed effettua le dovute operazioni.

3.6 Disconnessioni

Quando un client perde la connessione con il server a causa di un crash o di una chiusura volontaria dell'applicazione, una apposita funzione del server `onClientDisconnection()` è incaricata di gestire le operazioni di pulizia:

```
void onClientDisconnection (Client* client) {

    // Operazioni di disconnessione del client (va invocata solo nel thread del Client)

    int gameID = client->activeGame;

    printf("Client %d disconnesso\n", client->socket);

    pthread_mutex_lock(&activeUsers_lock);
    activeUsersList = delete(activeUsersList, client->nickname, areEqual_str, NULL);
    pthread_mutex_unlock(&activeUsers_lock);

    pthread_mutex_lock(&lookingForMatch_lock);
    clientsLookingForMatch = delete(clientsLookingForMatch, client, areEqual_cli, NULL);
    pthread_mutex_unlock(&lookingForMatch_lock);

    if (gameID >= 0) {

        pthread_mutex_lock(&games_lock);
        pthread_mutex_lock(&getGameByID(games, gameID)->nullPlayerLock);
        if (getGameByID(games, gameID)->activePlayer->client == client) {
            if (write(client->pipe[1], MATCH_LEFT_PIPE_MESSAGE_WRITE, strlen(MATCH_LEFT_PIPE_MESSAGE_WRITE)) < 0) {
                perror("Errore di scrittura sulla pipe");
                exit(EXIT_FAILURE);
            };
        }
        else {
            getGameByID(games, gameID)->players[client->positionInArrayOfPlayers] = NULL;
        }
        pthread_mutex_unlock(&getGameByID(games, gameID)->nullPlayerLock);
        pthread_mutex_unlock(&games_lock);

    }

    log_ClientDisconnection(client->socket);

    memset(client->nickname, 0, sizeof(client->nickname));

    close(client->socket);

    pthread_mutex_lock(&clients_lock);
    clients = delete(clients, client, areEqual_cli, NULL);
    pthread_mutex_unlock(&clients_lock);

    free(client);

    pthread_exit(NULL);

}
```

Se un utente aveva effettutato il log in su quel client, il suo nickname viene rimosso dalla lista di utenti attivi, e il client viene rimosso dalla lista di client in cerca di una partita, qualora vi fosse.

Se il client era in una partita al momento della disconnessione, ed era il giocatore attivo, si invia sulla sua pipe il messaggio di abbandono, in modo tale che al momento della lettura esso venga ricevuto, rimuovendo il giocatore dalla partita. Se il client non era il giocatore attivo, la sua posizione nell'array di giocatori della partita viene semplicemente settata a `NULL`.
In conclusione, la disconnessione viene registrata sul file di log degli eventi (vedi sottosezione successiva) e la socket viene chiusa. Il client viene eliminato dalla lista di client attivi `clients` e la memoria per esso allocata viene liberata. L'ultima operazione è l'uscita dal relativo thread.

Lato client, una disconnessione improvvisa causa un'eccezione di input/output. L'eccezione viene catturata e l'utente viene rimandato alla schermata di connessione per effettuare una nuova connessione quando sarà possibile.

3.7 Logging degli eventi

Sul server, gli eventi vengono registrati in degli appositi file di testo.

Connessioni, disconnessioni, log in e registrazioni vengono riportati nel file **eventi.txt**, aperto all'avvio del server. Le connessioni vengono riportate con il relativo indirizzo IP, e ogni evento viene loggato con relativa data di avvenimento.

Per quanto riguarda le partite, viene utilizzato un file separato per registrare gli eventi di ciascuna di esse. L'identificativo di tale file è conservato all'interno della struttura **Game** che la rappresenta. Il file viene aperto all'inizio di una partita e chiuso alla fine, e riporta l'inizio della partita, i giocatori presenti, le conquiste avvenute, la fine della partita e l'eventuale vincitore (una partita in cui tutti i giocatori hanno abbandonato non ha vincitore e viene registrata come parita conclusa per abbandono).

Di seguito un esempio su come viene registrata la connessione di un client sul file degli eventi tramite la funzione `log_ClientConnection()`:

```c
void log_ClientConnection (int client, struct in_addr ip) {

    char log[128] = "";
    char timestamp[32];
    char ip_str[INET_ADDRSTRLEN];

    makeTimestamp(timestamp);
    inet_ntop(AF_INET, &ip, ip_str, INET_ADDRSTRLEN);

    snprintf(log, sizeof(log), "[%s] CONNESSIONE CLIENT: si connette il client %d con indirizzo %s\n\n", timestamp, client, ip_str);

    pthread_mutex_lock(&user_events_file_lock);
    if (write(user_events_fd, log, strlen(log)) == -1) {
        perror("Errore di scrittura sul file degli eventi degli utenti");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_unlock(&user_events_file_lock);

}
```

3.8 Gestione della concorrenza tra thread

Per gestire le race condition generate dalla condivisione tra i vari thread delle variabili globali sono stati ritenuti sufficienti i seguenti mutex:

```c
pthread_mutex_t users_file_lock       = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t user_events_file_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t clients_lock          = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t activeUsers_lock      = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lookingForMatch_lock  = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t games_lock            = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t gamenum_lock          = PTHREAD_MUTEX_INITIALIZER;
```

utilizzati rispettivamente per

- file dei dati degli utenti (**utenti.txt**)
- file degli eventi (**eventi.txt**)
- lista globale di client (`clients`)
- lista globale di utenti attivi (`activeUsersList`)
- lista globale di client in cerca di una partita (`clientsLookingForMatch`)

- lista globale di partite in corso (`games`)
- numero di partita in corso (`gamenum`)

Oltre ai due presenti nella struttura **Game**, utilizzati per quando va settato un giocatore a `NULL` durante la partita e quando va modificato il file di quella specifica partita.

4 Link Google Drive al codice sorgente

https://drive.google.com/file/d/1lIR-zkzfeOGLD4_8Gmt_DTWP8Q4Q0lOf/view?usp=sharing