

# RELAZIONE PROGETTO WORDLE

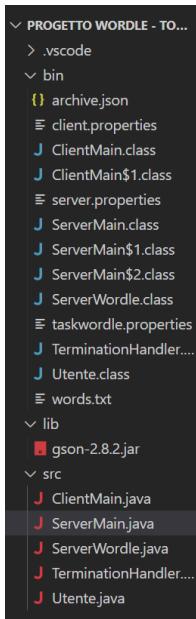
## Introduzione

Wordle è un progetto basato su un programma Client-Server che fornisce la possibilità a diversi utenti di giocare al giochino di parole omonimo divenuto celebre qualche anno fa. Gli utenti potranno, dopo essersi connessi al server di gioco, giocare tentando di indovinare la parola fornita ciclicamente dal server stesso, l'obiettivo degli utenti sarà cercare di indovinarla anche grazie agli aiuti forniti dal gioco, per scalare le classifiche e condividere i propri risultati con altri giocatori.

## Struttura del progetto

L'architettura generale del sistema è basata sul paradigma Client-Server: gli utenti del servizio possono accedere al sistema mediante un Client, che invia le richieste ad un Server (in rete locale) che le elabora e restituisce le informazioni richieste dal Client. Il progetto ricalca ampiamente l'idea dietro al gioco Dungeon Adventures, dove il client, mandando richieste tramite linea di comando, riceveva relative risposte che permettevano il proseguimento del gioco in varie direzioni, stessa cosa in questo progetto: il Client può eseguire una serie di comandi che lo interfacceranno ad una sequenza di botta e risposta col server stesso, dando così 'vita' al gioco in sé.

Il progetto è stato suddiviso in 5 classi principali che si trovano all'interno di `src`:



- `ServerMain.java` → classe alla base del server, reperisce i parametri di input dal file di configurazione `server.properties`, ripristina la mappa da `archive.json` che contiene e mantiene i vari giocatori e gestisce un pool di thread per l'interazione con i client.
- `ClientMain.java` → classe che si occupa del client, permette l'interazione uomo-client grazie ai comandi passati da linea di comando e interfaccia questa comunicazione col server per la ricezione di risposte che verranno stampate per far capire all'utente come sta proseguendo il gioco e il risultato delle azione che sta compiendo.
- `ServerWordle.java` → classe task lanciata ogni volta la pool di thread esegue, è la vera e propria "interfaccia" tra server e client in quanto riceve il comando da quest'ultimo, esegue una determinata azione di conseguenza e riferisce la risposta al client stesso.
- `TerminationHandler.java` → classe gestore della terminazione corretta lato server, interrompendo il server con CTRL+C entrerà in esecuzione permettendo di salvare la struttura dati che contiene gli utenti, rendendo quindi di fatto questa struttura persistente e riutilizzabile in un avviamento del server successivo.
- `Utente.java` → classe che tiene traccia di tutte le statistiche e informazione dell'utente in gioco, è utilizzata come base di appoggio per la gestione degli utenti dal server.

Librerie in `lib` :

Utilizzo la libreria `gson-2.8.2.jar` per la serializzazione e deserializzazione della mappa che contiene gli utenti, all'interno del file `archive.json`, garantendo così persistenza delle informazioni.

Cartella `bin` :

E' la cartella contenente le classi compilate e i file fondamentali per la buona riuscita del programma:

- `words.txt` → file contenente tutte le parole possibili scrivibili nel gioco, sono 30824 parole lunghe 10 caratteri, una di esse verrà ciclicamente estratta dal server.
- `server.properties` → file di configurazione del server, contiene la porta per la socket lato server, il numero di thread gestiti dalla threadpool, il tempo massimo prima della shutdown forzata della threadpool nel TerminationHandler, il `keepAliveTime` della threadpool e il tempo tra una parola da indovinare e l'altra, il server legge tutto ciò dal file grazie alla `readConfig()`.
- `client.properties` → file di configurazione del server, contiene nome dell'host, porta uguale a quella del server, porta ed host per la multicastSocket per unirsi al gruppo sociale. Il client leggerà tutte queste info invocando una `readConfig()`.

```
1  #
2  # Reti e Laboratorio III - A.A. 2022/2023
3  # Wordle
4  #
5  # File di configurazione del server
6  #
7
8  # Porta di ascolto del server.
9  port=1609
10 # Numero di thread disponibili.
11 nThread=12
12 # Tempo di attesa prima della chiusura del pool di thread.
13 maxDelay=15000
14 # KeepAliveTime della pool, aspetta 1 minuto per vedere se ci sono altri task da fare.
15 terminationDelay=60000
16 # Tempo che intercorre tra due parole (5min).
17 timeBetweenWords=300000
```

```
1  #
2  # Reti e Laboratorio III - A.A. 2022/2023
3  # Wordle
4  #
5  # File di configurazione del client
6  #
7
8  # Nome dell'host.
9  hostname=localhost
10 # Porta come quella server.
11 port=1609
12 # Porta relativa al multicast.
13 multicastPort=5555
14 # Host del gruppo multicast [224.0.0.0-239.255.255.255]
15 multicastHost=226.226.226.226
```

- `taskwordle.properties` → file di configurazione del task del server, che contiene solo le informazioni per l'implementazione multicast lato server in modo da inviare messaggi ai partecipanti al gruppo sociale.
- `archive.json` → file json contenente la stringa json che rappresenta la mappa degli utenti che hanno giocato o sono solo registrati al gioco Wordle.

```
# Reti e Laboratorio III - A.A. 2022/2023
# Wordle
#
# File di configurazione del task wordle
#
# Porta relativa al multicast.
multicastPort=5555
# Host del gruppo multicast [224.0.0.0-239.255.255.255]
multicastHost=226.226.226.226
```

```
{
  "tommy": {
    "username": "tommy",
    "password": "vanz",
    "haPartecipato": false,
    "puoGiocare": false,
    "logged": false,
    "partitaTerminata": false,
    "haVinto": false,
    "haPerso": false,
    "tentativi": 0,
    "partiteGiocate": 2,
    "percentualeVittorie": 0,
    "lengthLastWinStreak": 1,
    "lengthMaxWinStreak": 1,
    "guessDistribution": 0,
    "vittorie": 1,
    "myWinStreak": 0,
    "arrayTentativi": [
      1,
      2,
      3
    ]
  }
}
```

# Scelte effettuate nel progetto

Il mio progetto ha una struttura Client-Server: per prima cosa va avviato il server che si metterà in ascolto sulla porta passata con i parametri di configurazione, successivamente posso avviare uno o più client che saranno l'interfaccia tra il server e l'utente giocante.

L'idea è che il client stia in ascolto delle richieste passate da linea di comando dall'utente e il ServerWordle task mandi le risposte in base a che richiesta ha ricevuto.

Fondamentalmente tutto il lavoro di botta-risposta tra client e server è gestito dal task ServerWordle e ClientMain.

Di seguito le classi "core" del progetto spiegate:

## ServerMain

Va avviato per primo e prende i parametri di configurazione dal `server.properties` grazie alla funzione `readConfig()`; una volta fatto ciò il suo compito principale è stare in ascolto sulla porta passatagli per ricevere richieste di connessione.

Con una try with resources apre una listening socket sulla porta, un file ad acceso random sul vocabolario di parole e una multicast socket per poi poter mandare notifiche ai client che joineranno il gruppo sociale.

Un'altra azione importante, cruciale per l'immagazzinamento e la buona riuscita di più partite consecutive è il ripristino della mappa contenente i vari utenti, essa viene ripristinata dal file `archive.json` letto come stringa json. Se tale stringa è vuota, vuol dire che non ci sono utenti ancora registrati al servizio di gioco e quindi inizializzo una `ConcurrentHashMap` vuota, altrimenti la ripristino utilizzando le funzionalità di `gson`.

Prima del ciclo di ascolto di richieste affidato alla threadpool (inizializzata con un `ThreadPoolExecutor`), viene avviato un `TimerTask`, che ogni 5 minuti mi scansiona il file di parole da indovinare scegliendone una in particolare, aggiornandola rispetto alla precedente e resettando alcune variabili-Utente per poter permettergli di rigiocare con la nuova parola (la chiave da indovinare è una Atomic Reference, quindi anche se un client era già in esecuzione, la parola da indovinare viene aggiornata istantaneamente)

Una volta che il server ha le socket, la mappa di utenti e la parola da indovinare, il server entra in un loop dove accetta le richieste di connessione provenienti dal client, utilizzando un pool di thread, passandogli il task `ServerWordle` che si occuperà della interazione effettiva.

```
// Ciclo gestione dei client con un pool di thread
while (true) {
    Socket socket = null;
    // Accetto le richieste provenienti dai client
    // Quando il TerminationHandler chiude la ServerSocket, si solleva una SocketException
    try {
        socket = serverSocket.accept();
    } catch (SocketException e) {
        break;
    }
    // Avvio un ServerWordle per interagire con il client.
    pool.execute(new ServerWordle(socket, multicastSocket, map, keyword));
} // Gli passo anche la mcSocket almeno combacia con tutti i client, la mappa com
```

```
String file = "archive.json";
String jsonString = readfileAsString(file);
if(jsonString.equals("null") || jsonString.equals("anObject: null")){
    System.out.println("[SERVER] Inizializzo la mappa -> " + map);
    map = new ConcurrentHashMap<String, Utente>();
} else {
    System.out.println("[SERVER] Mappa ripristinata dal MainServer -> " + map);
    map = new Gson().fromJson(jsonString, new TypeToken<ConcurrentHashMap<String, Utente>>() {
    });
}
```

Invece il salvataggio della struttura dati degli utenti è affidato al `TerminationHandler`, esso è un thread che viene inizializzato subito dopo aver ripristinato la mappa nel `ServerMain` MA va in esecuzione solamente allo spegnimento del server e/o alla sua brusca interruzione con `CTRL+C`, il `TerminationHandler`, è praticamente lo stesso visto a lezione, in aggiunta ha solo il meccanismo di salvataggio della mappa, che viene serializzata e messa nel file json di archivio degli utenti.

```
// Trasformo la mappa in un json e lo metto nel file archive.json
System.out.println("[SERVER TERMINATION] Mappa salvata dal TerminationHandler -> " + map);
Gson gson = new GsonBuilder().setPrettyPrinting().create();
try (FileWriter fileWriter = new FileWriter(fileName: "archive.json")) {
    gson.toJson(map, fileWriter);
} try {
    fileWriter.close();
} catch (IOException e1) {
    System.err.printf(format: "[SERVER ERROR] Errore chiusura FileWriter.");
}
} catch (JsonIOException | IOException e2) {
    System.err.printf(format: "[SERVER ERROR] Errore trasformazione mappa in json.");
}
```

```
// Timer che ogni tot secondi reimposta la possibilità di giocare e aggiorna la parola da indovinare
// tempo tra una parola e l'altra reperito da server.properties
Timer timer = new Timer();
TimerTask task = new TimerTask() {
    public void run(){
        // Scelta stringa da usare come parola da indovinare.
        byte[] bufferString = new byte[10];
        int randomNum = ThreadLocalRandom.current().nextInt(origin: 0, 30824 + 1);
        try{
            // Il file parte dal byte 0, la seconda parola si trova all'11° bit e così via (parole di 10 byte)
            f.seek(randomNum * 11);
            f.read(bufferString);
        } catch (IOException e) {
            System.out.println(x: "[SERVER ERROR] Errore lettura parola chiave.");
        }
        // Stampo e setto la nuova stringa da cercare
        String newString = new String(bufferString);
        keyword.set(newString);
        System.out.println("[SERVER] Chiave da indovinare ---> " + keyword.toString());
    }
}
```

## ClientMain

Il client è il mezzo tra utente umano e server. Anch'esso riceve i parametri di configurazione da un file di proprietà nella cartella *bin* leggendoli tramite la `readConfig()`; anche il client con una try with resources (utile per chiudere autonomamente le risorse) si inizializza la socket verso il server, la multicastSocket tramite la quale si legherà al gruppo sociale e due importantissimi Scanner, uno per prender l'input dall'umano tramite linea di comando, l'altro come input dal ServerWordle tramite uno stream di input per ricevere le risposte del server.

Il client può inserire un insieme ben definito di comandi da mandare al server (da ora in poi mi riferirò al server non come ServerMain bensì come struttura che riceve richieste e manda risposte grazie al ServerWordle task) , inserire comandi diversi comporta al ricevere un messaggio di risposta di default che indica di cambiare comando.

Lista di comandi :

- `register`
- `login`
- `logout`
- `play wordle` (abbreviazione `play`)
- `send word` (abbreviazione `send`)
- `send me statistics` (abbreviazione `send stats`)
- `share`
- `show me sharing` (abbreviazione `show shares`)
- `remove user`
- `exit`

Tutto ciò avviene sempre all'interno del ciclo while di ascolto di risposte. Il client manda UNA STRINGA come comando e riceve UNA STRINGA come risposta, ho deciso di riutilizzare il paradigma usato nel Dungeon Adventure poiché non era necessaria la ricezione o invio di più di una stringa per volta.

In base a determinati tipi di risposte, il client effettuerà ulteriori azioni:

- Se il client richiede ed effettua un `login` ed essa avviene con successo, allora esso si legherà al gruppo sociale sul quale sono presenti il server ed eventuali altri client, joinandolo. Non solo, ma avvierà anche un thread (`threadListenerUDPs`) per stare in ascolto di eventuali notifiche UDP mandate da altri client che hanno effettuato una `share`.

```
// Thread che inizializzo non appena il client fa la join
Thread threadListenerUDPs = new Thread () {
    public void run () {
        System.out.println("[" + CLIENT + "] Il client si mette in attesa di share dagli altri utenti.");
        while(!exits){
            // Se ricevo condivisioni tramite UDP le stampo nel client che ha mandato show me sharing
            DatagramPacket response = new DatagramPacket(new byte[1024], length: 1024);
            try {
                multicastSocket.receive(response);
            } catch (IOException e) {
                System.out.println("[" + CLIENT + "] Il client non riceve notifiche al momento.");
            }
            String responseUDPfromServer = new String(response.getData()), offset: 0, response.getLength());
            arrayNotifiche.add(responseUDPfromServer);
            // Decomenta se vuoi vedere ogni volta il client che notifiche riceve
            // System.out.println("[CLIENT] Ricevuta condivisione risultati dal server: " + responseUDPfromServer);
        }
    }
};
```

```
// Se ho loggato un utente con successo lato server allora riceverò una unica risposta, quindi
// unisco il client al gruppo di multicast di cui fa parte il server stesso
if(inputString.equals(anObject: "[LOGIN] Utente loggato con successo.")){
    arrayNotifiche.clear();
    exitsms = false; // Risetto eventualmente a false per riavviare l'ascolto sul gruppo
    InetAddress group = InetAddress.getByAddress(multicastHost); // Stesso gruppo del server
    InetSocketAddress address = new InetSocketAddress(group, multicastPort);
    NetworkInterface networkInterface = NetworkInterface.getByName(group);
    try { // Una volta loggato con successo, se il client non ha ancora joinato, joina il gruppo
        multicastSocket.joinGroup(address, networkInterface);
        System.out.println("[" + CLIENT + "] Il client joina il gruppo sociale " + multicastHost);
        // E si mette in ascolto di info sharate dagli altri da mettere nell'array di notifiche
        threadListenerUDPs.start();
    } catch (SocketException e) {
        // Se il client aveva già precedentemente joinato allora "joinGroup"
        // darà errore, che verrà catturato con SocketException
        System.out.println("[" + CLIENT + "] Il client ha joinato il gruppo sociale precedentemente.");
        continue;
    }
}
```

Il thread immagazzina le eventuali notifiche ricevute in un array di notifiche e solamente se il client stesso effettuasse una `show me sharing` per voler vedere le notifiche degli altri utenti, l'array di notifiche verrebbe restituito.

```
if(inputString.equals(anObject: "[SHOW ME SHARING] Richiesta condivisione risultati avvenuta con succ")){
    // Attivo un thread che sta in attesa di ricevere messaggi UDP dal server e che li stampa sul client
    System.out.println("[" + SHOW_ME_SHARING + "] Notifiche ricevute -> " + arrayNotifiche.toString());
}
```

- Se un utente, al contrario, effettuasse una `logout`, allora il client chiuderebbe la comunicazione col gruppo sociale ed uscirebbe dal thread che era in attesa di notifiche UDP.

```
// Se ho fatto la logout sull'utente devo chiudere la multicast socket per questo client,
// interrompo il thread che era in ascolto delle notifiche e svuoto l'array di notifiche
if(inputString.equals(anObject: "[LOGOUT] Logout effettuato sull'utente.")){
    multicastSocket.close();
    exits = true;
    System.out.println("[" + CLIENT + "] Client uscito dal gruppo sociale.");
    continue;
}
```

## ServerWordle

Il grosso del lavoro del gioco è svolto in questa classe, ne viene creata un'istanza ogni qual volta un client si connette al server, tutto ciò grazie al pool di thread e alla socket. Il task riceve dal server la socket di connessione col client (fondamentale per ricevere i comandi in input e mandare le risposte in output), la multicast socket per istanziare il gruppo sociale in quanto farà le veci di server “inviatore di notifiche di gioco”, la mappa aggiornata con i vari utenti e la parola da indovinare che si aggiornerà atomicamente essendo una `AtomicReference`.

Tutto ciò che fa è contenuto in un grosso metodo `run()`: essendo un task, per prima cosa come il client e il server, si legge dal file di configurazione la porta e l'host multicast, quest'ultimo lo utilizza subito per istanziare il gruppo sociale corretto (vi manderà solo messaggi UDP, non lo joina);

Poi in una try with resources inizializza l'input per ricevere i SINGOLI comandi dal client e l'output per mandare SINGOLE stringhe di risposta al client e inizia a svolgere il compito principale: **un while dove reperisce continuamente i comandi che gli arrivano dal client e tramite uno switch, in base a che comando è stato inviato dal client, effettuerà una data azione.**

```
try {
    Scanner in = new Scanner(socket.getInputStream()); // Recepisce le linee dal
    PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true)
}
{
    while (in.hasNextLine()) {
        end:
        switch (in.nextLine()) {
            case "register": {
                String[] args = in.nextLine().split(" ");
                if (args.length < 2) {
                    out.println("Error: missing arguments");
                    continue;
                }
                String username = args[0];
                String password = args[1];
                if (username.isEmpty() || password.isEmpty()) {
                    out.println("Error: invalid arguments");
                    continue;
                }
                if (users.containsKey(username)) {
                    out.println("Error: user already exists");
                    continue;
                }
                users.put(username, password);
                out.println("User registered successfully");
            }
            case "login": {
                String[] args = in.nextLine().split(" ");
                if (args.length < 2) {
                    out.println("Error: missing arguments");
                    continue;
                }
                String username = args[0];
                String password = args[1];
                if (username.isEmpty() || password.isEmpty()) {
                    out.println("Error: invalid arguments");
                    continue;
                }
                if (!users.containsKey(username)) {
                    out.println("Error: user does not exist");
                    continue;
                }
                if (!users.get(username).equals(password)) {
                    out.println("Error: wrong password");
                    continue;
                }
                User user = new User(username);
                user.setLoggedIn(true);
                users.put(username, user);
                out.println("User logged in successfully");
            }
            case "logout": {
                String[] args = in.nextLine().split(" ");
                if (args.length < 1) {
                    out.println("Error: missing arguments");
                    continue;
                }
                String username = args[0];
                if (username.isEmpty()) {
                    out.println("Error: invalid arguments");
                    continue;
                }
                if (!users.containsKey(username)) {
                    out.println("Error: user does not exist");
                    continue;
                }
                User user = users.get(username);
                user.setLoggedIn(false);
                users.put(username, user);
                out.println("User logged out successfully");
            }
            case "play wordle": {
                String[] args = in.nextLine().split(" ");
                if (args.length < 1) {
                    out.println("Error: missing arguments");
                    continue;
                }
                String word = args[0];
                if (word.isEmpty()) {
                    out.println("Error: invalid arguments");
                    continue;
                }
                if (word.length() > 5) {
                    out.println("Error: word too long");
                    continue;
                }
                if (!word.matches("[a-zA-Z]+")) {
                    out.println("Error: word contains non-alphabetic characters");
                    continue;
                }
                String[] words = wordList.split(" ");
                if (!words.contains(word)) {
                    out.println("Error: word not found in dictionary");
                    continue;
                }
                User user = users.get(username);
                user.setWordleWord(word);
                user.setWordleAttempts(0);
                users.put(username, user);
                out.println("Game started! Good luck!");
            }
            case "send": {
                String[] args = in.nextLine().split(" ");
                if (args.length < 2) {
                    out.println("Error: missing arguments");
                    continue;
                }
                String word = args[0];
                String attempts = args[1];
                if (word.isEmpty() || attempts.isEmpty()) {
                    out.println("Error: invalid arguments");
                    continue;
                }
                if (word.length() > 5) {
                    out.println("Error: word too long");
                    continue;
                }
                if (!word.matches("[a-zA-Z]+")) {
                    out.println("Error: word contains non-alphabetic characters");
                    continue;
                }
                if (!attempts.matches("\\d+")) {
                    out.println("Error: attempts must be a number");
                    continue;
                }
                User user = users.get(username);
                user.setWordleWord(word);
                user.setWordleAttempts(Integer.parseInt(attempts));
                users.put(username, user);
                out.println("Word sent successfully");
            }
            case "send me statistics": {
                User user = users.get(username);
                out.println("User statistics:");
                out.println("Wins: " + user.getWins());
                out.println("Losses: " + user.getLosses());
                out.println("Ties: " + user.getTies());
                out.println("Accuracy: " + (user.getWins() + user.getTies() == 0 ? 0 : ((user.getWins() + user.getTies()) / user.getLosses()) * 100));
            }
            case "share": {
                User user = users.get(username);
                if (user.getShares() > 0) {
                    out.println("Shares: " + user.getShares());
                } else {
                    out.println("No shares available");
                }
            }
            case "show me sharing": {
                User user = users.get(username);
                if (user.getShares() > 0) {
                    out.println("Shares: " + user.getShares());
                } else {
                    out.println("No shares available");
                }
            }
            case "exit": {
                out.println("Exiting game");
                break end;
            }
            default: {
                out.println("Unknown command");
            }
        }
    }
}
```

Lato server naturalmente i comandi sono gli stessi del lato client, in aggiunta sono nascosti 3 comandi “speciale” che può ricevere lo stesso che mi hanno aiutato durante la scrittura del programma: `CHEAT` , `UNLOG USERS` e `reset my stats` (questi non sono ‘visibili’ client-side).

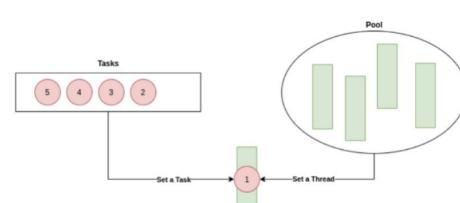
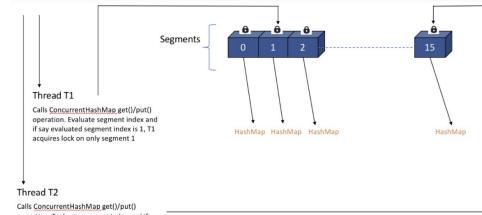
I restanti comandi visualizzati anche all'accensione del client, mi permettono l'implementazione del gioco:

- `register` : mi permette di registrare un nuovo utente alla ConcurrentHashMap chiedendo prima username e poi password uno dietro l'altro, controllando che sia un utente “nuovo” e non nullo, se va tutto a buon fine inserisce il nuovo utente nella mappa
- `login` : consente all'umano di loggarsi con un dato utente, esso deve essere presente nella mappa e la password deve essere corretta, una volta loggato, l'utente lato cliente joinerà il gruppo sociale, l'utente appena loggato diventerà l'utente a cui farà riferimento il ServerWordle su cui è stata fatta la login.
- `logout` : scollega l'utente attualmente loggato dal gioco, lato client chiuderà la socket relativa al multicast ed esce dal thread di ricezione delle notifiche UDP.
- `play wordle` : permette all'utente loggato di iniziare la sessione di gioco, non deve aver già giocato oppure la parola non deve essersi ancora aggiornata per effettuare correttamente questo comando.
- `send` : permette all'utente di mandare una parola che pensa sia quella corretta, una volta mandato il comando, se tutto va a buon fine in base ai vincoli imposti dal gioco, l'utente può inviare una parola esistente nel vocabolario, se la parola fosse troppo corta o inesistente, avverte l'utente senza consumare tentativi, in caso contrario la parola sarà semplicemente inesatta e viene mandato un indizio tramite array di caratteri con formato indicato nella consegna del progetto (+,?,X). Se l'utente indovina, ha vinto e termina di giocare fino a che non esce la prossima parola allo scattare dei 5 minuti.
- `send me statistics` : manda le statistiche all'utente relative all'utente attualmente loggato, le statistiche che gli manda sono tutte presenti all'interno della classe Utente, le cinque statistiche principali descritte dal punto 1 del progetto vengono, sia passate al costruttore per inizializzare un Utente, che salvate dal server e ripristinate al riavvio in modo da poter effettuare diverse partite con i salvataggi effettuati sulle giocate precedenti. La guess distribution l'ho rappresentata sia come array dei tentativi effettuati nelle varie partite, che come media degli stessi, per dare un'idea di quanto un utente ci abbia messo a indovinare una data parola.
- `share` : solo dopo aver effettuato una partita, con esito positivo o negativo, l'utente potrà condividere sul gruppo sociale l'esito della partita stessa, la notifica arriverà solo agli altri client già con utente loggato e potranno vederla solamente dopo aver effettuato una `show me sharing` , tale notifica verrà immagazzinata nei vari array di notifiche dei client in ascolto grazie al threadListener attivato dopo la join.
- `show me sharing` : lato server non succede praticamente nulla mandando questo comando dal client, infatti semplicemente il server risponde con una stringa di successo o fallimento, il grosso avviene tutto lato client, perché se il client stesso riceve un responso positivo alla richiesta di mostrare le condivisioni, allora mostrerà l'array di notifiche salvate man mano che gli altri utenti effettuavano le `share` .
- `exit` : lato server non fa nulla, manda solo una stampa sul terminale del server, invece lato client chiude tutte le socket ed esce dal while di ricezione comandi.

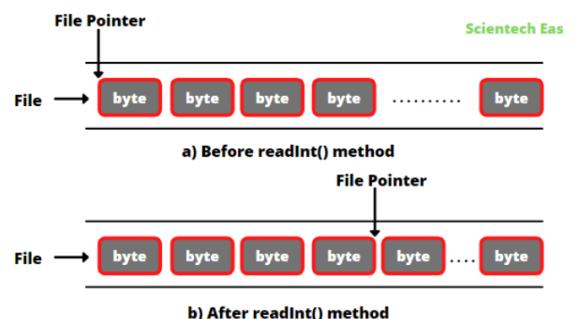
# Strutture utilizzate

## Lato Server :

- **ConcurrentHashMap<String, Utente>** : ho utilizzato una concurrenthashmap per immagazzinare i vari utenti, mi era tornata utile nell'assignment sulle occorrenze, anche in quello si era utilizzata una concurrenthashmap in combo con una threadpool, infatti ho pensato di riutilizzarla associando questa volta username-oggetto Utente come chiave-valore, in modo che poi potessi riutilizzare questa struttura nelle operazioni effettuate all'interno del task server. Avere una stringa rappresentante l'username in associazione con l'utente dello stesso username mi è parso utile da subito.
- **ThreadPoolExecutor** : ideale per la gestione di più richieste e per interagire con più client, in quanto si ha una gestione automatica dei task dall'ExecutorService. Inoltre con la LinkedBlockingQueue ho una coda di lunghezza "infinita".

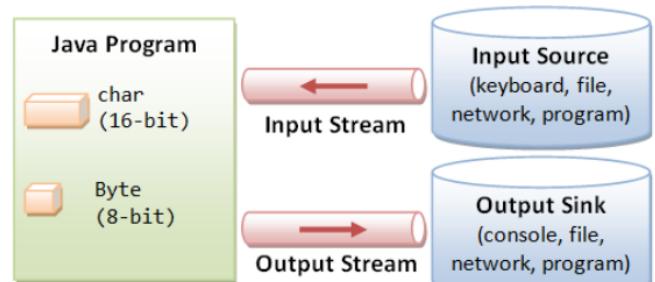


- **RandomAccessFile** : utilizzo un RandomAccessFile per la lettura della parola segreta, all'interno del thread TimerTask che 'pesca' la parola randomicamente in quando mi torna utile il meccanismo di puntatore all'interno del file ad accesso randomico, infatti settando il file pointer con una `seek` che prende un offset random ma con parametri corretti ( $0 \leq \text{offset} \leq 30824 * 11\text{byte}$ ) , riesco a leggere sempre una parola di 10 lettere randomica nel file `words.txt`



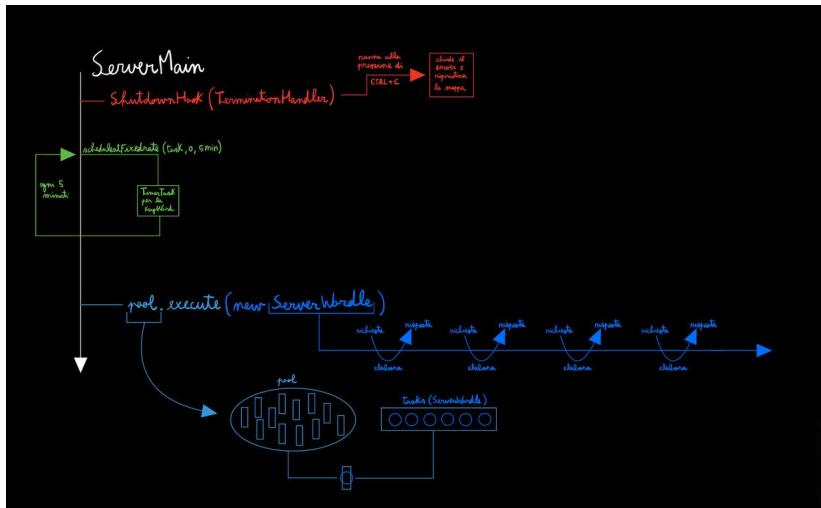
## Lato Client :

- **ArrayList<String>** : mi serve un array di stringhe dove poter immagazzinare le notifiche che ricevo lato client (tramite una .add) una volta joinato il gruppo multicast.
- **Stream** : InputStream e OutputStream rispettivamente per ricevere la risposta dal server task e per mandarla al medesimo.

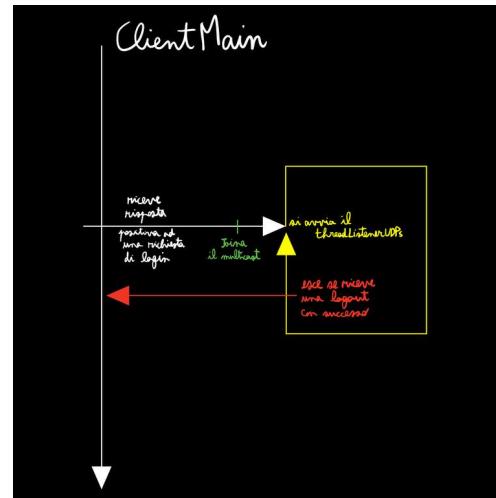


# Schema Thread attivati

Lato Server :



Lato Client :



## Istruzioni

### Compilazione progetto

La compilazione corretta del progetto prevede:

- Aprire il terminale powershell di VSCode o Windows PowerShell;
- Utilizzare il comando `Set-Location` oppure iterativamente “`cd` + Tab” per andare alla cartella di destinazione contenente i files .java (nel mio caso essi sono contenuti all'interno di `src` in una cartella Progetto Wordle);
- Una volta all'interno della cartella `src` bisogna compilare i file .java tramite `javac -cp ..\lib\gson-2.8.2.jar" *.java -d ..\bin`, ciò compilerà il codice con l'ausilio della libreria gson che si trova in lib e metterà i .class all'interno di bin, dove si devono trovare anche i file di supporto al programma (json e properties);
- Successivamente, una volta che i file di supporto e gli eseguibili sono insieme, per eseguire i main sia lato client che lato server e dare così inizio al gioco, bisogna andare nella cartella `bin` (se rimanessi nella `src` non permetterebbe l'esecuzione) tramite i comandi `cd ..` e `cd bin` e inserire il comando di esecuzione `java -cp ..\lib\gson-2.8.2.jar" ServerMain` (oppure `ClientMain` se volessi mettere in esecuzione un client).

Passaggi su VSCode e su Windows PowerShell:

Server (prima su VSC e poi su WP):

```
PS C:\Users\tomas> cd D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> cd src
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> javac -cp ..\lib\gson-2.8.2.jar" *.java -d ..\bin
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> cd bin
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> java -cp ..\lib\gson-2.8.2.jar" ServerMain
[SERVER] In ascolto sulla porta: 1609
[SERVER] Mappe ripristinate dal MainServer -> [tomy = {tomy,vanz,2,0,1,1,0} , sonia = {sonia,delgiu,0,0,0,0,0} ]
[SERVER] Chiave da indovinare --> jeezreelite
```

Client:

```
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> cd bin
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> java -cp ..\lib\gson-2.8.2.jar" ClientMain
[CLIENT] Connessione all' host localhost sulla porta: 1609
[CLIENT] L'utente puo' eseguire le seguenti azioni:
--> register
--> login
--> logout
--> play word | play
--> send word | send
--> send me statistics | send stats
--> share
--> show me sharing | show shares
--> remove user
--> exit
--> help
[LOGIN] Username:
tomy
[LOGIN] Password:
vanz
[LOGIN] Utente loggato con successo.
[CLIENT JOIN] Il client joina il gruppo sociale 226.226.226.226
[CLIENT] Il client si mette in attesa di share dagli altri utenti.
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Install la versione più recente di PowerShell per nuove funzionalità e miglioramenti. https://aka.ms/PSWindows

PS C:\Users\tomas> Set-Location D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> cd src
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> javac -cp ..\lib\gson-2.8.2.jar" *.java -d ..\bin
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> cd bin
PS D:\Progetti e Codici\LABORATORIO DI RETI\Progetto Wordle - Tommaso Vanz> java -cp ..\lib\gson-2.8.2.jar" ServerMain
[SERVER] Server online.
[SERVER] In ascolto sulla porta: 1609
[SERVER] Inizializzazione gruppo sociale sull'host 226.226.226.226
[SERVER] Mappe ripristinate dal MainServer -> [tomy = {tomy,vanz,2,0,1,1,0} , sonia = {sonia,delgiu,0,0,0,0,0} ]
[SERVER] Chiave da indovinare --> undisposed
```

## Giocare a Wordle

```
[CLIENT] L'utente puo' eseguire le seguenti azioni:  
|-> register  
|-> login  
|-> logout  
|-> play wordle | play  
|-> send word | send  
|-> send me statistics | send stats  
|-> share  
|-> show me sharing | show shares  
|-> remove user  
|-> exit  
login ●  
[LOGIN] Username: ●  
tommaso ●  
[LOGIN ERROR] Utente non esistente, devi effettuare la register. ●  
login ●  
[LOGIN] Username: ●  
tommy ●  
[LOGIN] Password: ●  
vanz ●  
[LOGIN] Utente loggato con successo. ●  
[CLIENT JOIN] Il client joina il gruppo sociale 226.226.226.226  
[CLIENT] Il client si mette in attesa di share dagli altri utenti.  
play ●  
[PLAY] L'utente tommy inizia a giocare. ●  
play ●  
[PLAY ERROR] L'utente tommy ha partecipato per la parola corrente. ●  
send ●  
[SEND] Inserisci la parola che pensi sia corretta: ●  
absolutely ●  
[SEND HINT] 'absolutely' parola ERRATA : (Tentativi effettuati: 1). [INDIZIO] --> [?, X, ?, X, ?, ?, ?, ?, ?, X] ●  
send ●  
[SEND] Inserisci la parola che pensi sia corretta: ●  
surtularia ●  
[USER WIN] HAI VINTO! La parola da indovinare era 'surtularia'. ●
```

lista comandi da inserire

● = input utente  
● = risposta server

Per giocare è necessario inserire i comandi mostrati all'avvio del Client, bisogna inserirne uno dopo l'altro SOLO dopo aver ricevuto la risposta del Server avendolo precedentemente avviato, inserendo il comando corrispondente in base alle azioni che si vuole conseguire, nell'esempio sopra ho iniziato a giocare con un utente già registrato, partendo con una mappa già costruita e che ho fornito già in archive.json , man mano che inserisco comandi il server mi risponde prontamente di conseguenza, attuando le azioni opportune nel ServerWordle, ad esempio per fare dei tentativi per indovinare la parola, devo (dopo aver fatto `play`), inserire `send`, INVIARE CON ENTER e successivamente scrivere la parola che penso sia corretta e inviarla.

Una volta inserito un comando, finché esso sarà diverso dall' `exit` , il client riceverà messaggi di risposta che iniziano con una stringa tra [ ] (ad esempio [LOGIN]) che fa capire il tipo di risposta mandata dal server in base a che comando è stato mandato dal client, ogni comando porta con sé delle conseguenze e vincoli e permette il proseguimento del gioco.

Lista di comandi e che cosa effettuano:

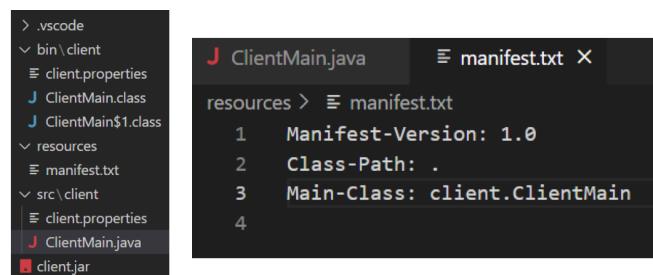
Questi comandi appena mostrati vanno eseguiti **così come sono scritti, senza spazio in fondo**, in caso contrario il server non riconoscerà il comando stesso avvertendo il giocatore. Seguendo l'approccio botta e risposta pensato come fondamenta del progetto, ad ogni tentativo di indovinare il giocatore sarà costretto sempre a inserire il comando `send` , questo per limitare a quanto scritti il numero di comandi possibili sacrificando la velocità di gioco.

## JAR ed esecuzione

Precedentemente ho mostrato come compilare e come eseguire il progetto direttamente dal terminale, ora passiamo alla creazione dei file jar suddividendo il progetto in due pacchetti in quanto ho due main da gestire:

- Lato client ho creato un package "client" all'interno del quale ho inserito il ClientMain, considerando che il client fa quel che deve tutto da solo, inoltre ho creato una cartella resources dove ho inserito il file manifest.txt per la gestione del jar

Una volta fatto ciò son passato alla creazione del jar stesso: prima si compila come visto prima, successivamente il comando da inserire è `jar cvfm client.jar resources\manifest.txt -C bin . src` per la creazione del jar, una volta ottenuto il file `client.jar` per eseguirlo bisogna inserire `java -jar client.jar`



```
> .vscode  
└ bin\client  
  └ client.properties  
    J ClientMain.class  
    J ClientMain$1.class  
  resources  
    manifest.txt  
  src\client  
    └ client.properties  
      J ClientMain.java  
    client.jar
```

ClientMain.java	manifest.txt
resources >	manifest.txt
1 Manifest-Version: 1.0	
2 Class-Path: .	
3 Main-Class: client.ClientMain	
4	

- Lato server invece sono presenti i restanti file, ma la procedura fondamentalmente è la stessa cambiando i nomi utilizzati ed inserendo i file nelle cartelle opportune si utilizza `jar cvfm server.jar resources\manifest.txt -C bin . src lib` per la creazione e `java -jar server.jar` per l'esecuzione, l'unica **piccola differenza è che nel manifest.txt devo aggiungere il ClassPath per allacciarmi alla libreria gson in quanto nel server la utilizzo.**

The screenshot shows a Java project structure named "SERVER" in a code editor. The "resources" folder contains a "manifest.txt" file with the following content:

```
Manifest-Version: 1.0
Class-Path: lib/gson-2.8.2.jar
Main-Class: server.ServerMain
```

The terminal window shows the command `jar cvfm server.jar resources\manifest.txt -C bin . src lib` being run, followed by the output of the jar creation process. Then, the command `java -jar server.jar` is run, and the server starts listening on port 1609. The log shows the server's initialization and a message about a password reset from the MainServer.

Come si può vedere, dall'esempio sotto sono presenti sia server che client, con i relativi jar, che vengono avviati utilizzando il jar stesso ed inizia così la comunicazione tra loro.

This screenshot shows a Java project structure with both "Client" and "Server" components. Both have their own "resources" and "bin" folders containing "manifest.txt" files. The "Client" side's manifest.txt includes the "lib" directory. The "Server" side's manifest.txt includes the "bin" and "src" directories. The terminal shows the creation of both jars and their execution. The Client logs a password reset message, while the Server logs its startup and a message about a password reset from the Client.

Come si può vedere, dall'esempio sotto sono presenti sia server che client, con i relativi jar, che vengono avviati utilizzando il jar stesso ed inizia così la comunicazione tra loro.

La struttura finale del progetto aggiungendovi anche i jar è la seguente:

All'interno della cartella "Client" e "Server" sono presenti le rispettive applicazioni singole, con i rispettivi file mostrati sopra e **CONTENGONO IL JAR** per l'esecuzione del programma, infine la cartella "Progetto Wordle" contiene tutto il codice sorgente sia lato client che lato server visto all'inizio e che è possibile eseguire anch'esso tramite compilazione ed esecuzione da terminale.

Client (applicazione singola del client, co...)	14/12/2022 18:22	Cartella di file
Progetto Wordle - Tommaso Vanz	14/12/2022 17:14	Cartella di file
Server (applicazione singola del server, c...)	14/12/2022 18:19	Cartella di file

*Tommaso Vanz*