# Wordle 3.0

Laboratorio di Reti

2022/2023

Francesco Simonetti 618867

**SCHEMA GENERALE**:

Nello stesso progetto ci sono due cartelle separate: Client e Server.

**Client**:

Client è composto da due classi ClientMain e ClientCondivisioni.

ClientMain main si preoccupa instaurare una connessione con il server tramite socket utilizzando una connessione TCP, una volta stabilita si preoccupa di gestire le diverse fasi che consistono nel capire le richieste dell'utente, e basandosi su queste instaurare uno scambio di messaggi con il server perché tutto funzioni. ClientMain inoltre (nella fase di login ) inizializza l'oggetto ClientCondivisioni che si preoccupa della ricezione e della stampa (quando richiesto) delle partite degli altri utenti.

**Server**:

Server è composto da ServerMain, CentralServer, Worker, Partita, Utente, TerminationHandler.

ServerMain si basa principalmente nel lanciare i thread Worker che gestiranno autonomamente le partite degli utenti rimanendo sempre connessi con Client. Prima di lanciare i Worker, ServerMain crea un oggetto di tipo CentralServer, che verrà passato per riferimento a tutti i singoli thread Worker; questo perché i Worker dovranno (durante le singole fasi) memorizzare e ricevere dati che dovranno essere "raccolti" tutti insieme e questo verrà fatto in CentralServer (dopo vederemo in modo più dettagliato).

L'oggetto Utente viene creato dentro Worker ogni volta che un utente effettua una registrazione con successo, dopo di che Worker chiamerà un metodo di CentralServer passandogli come prametro l'utente in modo che l'utente possa essere memorizzato in un HashMap (in CentralServer).

L'oggetto Partita viene creato dentro Worker durante la fase  di gioco passandogli utente e la parola da indovinare; grazie ai metodi di partita, Worker potrà gestire la fase di gioco ottenendo i suggerimenti necessari (quando validi) ed inviarli al Client. L'oggetto partita durante la fase di gioco, oltre a dare le giuste informazioni a worker, si creare via via una stringa che rappresenta la partita corrente, questa (se richiesto) verrà richiamata da Worker che la utilizzerà per condividerla ne gruppo multicast con gli utenti.

TerminationHandler viene inizializzato il ServerMain e viene lanciato quando viene rilevato un segnale di terminazione. TermiationHandler si preoccupa del salvataggio dati.

## CLASSI, METODI, VARIABILI E STRUTTURE DATI

Vediamo nello specifico le classi e i metodi principali; prima lato client e poi lato server

**Oss1.** Non sono presenti tutti i metodi ma solo quelli dove sono state prese scelte personali e che hanno quindi bisogno di spiegazione

**Oss2.** Per non essere ridondante molte cosa che verranno spiegate successivamente nella parte di "FLUSSO DI ESECUZIONE" non sono presenti in questo elenco di classi e metodi.

---------- Client ----------

### ClientMain

**Metodi:**

`public static void registrazione`
Si preoccupa della fase di registrazione ( spiegato nel dettaglio successivamente )

`public static void login`
Si preoccupa della fase di login ( spiegato nel dettaglio successivamente )

`public static void wordle`
Si preoccupa della fase in cui siamo dentro a wordle ( spiegato nel dettaglio successivamente )

`public static void public static void statistiche`
Si preoccupa di stampare le statistiche inviate in formato stringa dal server

### ClientMulticast

**Variabili e strutture dati:**
- porta per gruppo multicast (passata per parametro da ClienMain)
- indirizzo per gruppo multicast in formato stringa (passata per parametro da ClienMain)
- listaPartire = questo è array sincronizzato dove verranno inserite le partite ricevute dal server

`public void run`
All'interno di run utilizzando le variabili passate utente si unisce al gruppo. Dopo essersi unito parte un ciclo while(true) sempre attivo in che aspetta e pacchetti dal server (partite), le converte in stringhe e le inserisce dentro la lista listaPartite.

**Metodi:**

`public void showMeSharing()`
Questo metodo molto semplicemente scorre listaPartite e le stampa una a una. In questo metodo (scelta personale) verranno mostrate anche le partite dell'utente stesso, essendo un gruppo ho pensato che tutti vedranno le partite di tutti, le loro comprese. Altra scelta personale, nessuno saprà con che parola hanno giocato gli altri utenti altrimenti potrebbero ricavare informazioni.

## ---------- Server ----------

`ServerMain`

Ottiene dal file properties indirizzi e porte necessarie per le connessioni sia TCP che UDP.
nella parte di “FLUSSO DI ESECUZIONE” sarà spiegato il funzionamento di questa classe.

`Worker`

**Variabili:**
All’oggetto Worker vengono passati, da ServerMain, questi parametri: socket, centralserver, indirizzo multicast e porta multicast.

**Metodi:**

```
public void registrazione(DataInputStream in, DataOutputStream out)
```

```
public void login(DataInputStream in, DataOutputStream out)
```

```
public void gioca(DataInputStream in, DataOutputStream out)
```

Questi tre metodi verranno spiegati più nel dettaglio successivamente nella parte di “FLUSSO DI ESECUZIONE”.

```
public void inviopartita(Partita partita)
```

Il metodo ottiene la partita sotto forma di stringa chiamando il metodo getPartita della classe Partita (formato stringa), fatto questo  crea il datagramma e lo spedisce in multicast.

`CentralServer()`

Questa classe verrà utilizzata per creare l’oggetto centralserver condiviso da tutti i worker. Qui saranno tenute tutte le informazioni utili per il gioco. Non gli viene passato nessun valore.
**variabili e strutture dati:**
**- HashMap<String, Utente> HMUtenti =** questa è la struttura dati che ho deciso di creare per tenermi tutte le informazioni riguardo agli utenti. Come chiave ho utilizzato il nome utente e come valore associato l’oggetto utente che contiene tutte le informazioni relative all’utente (password, statistiche, etc.). Questa scelta potrebbe non essere ottima in spazio ma è molto veloce in tempo per molte operazioni che periodicamente vengono svolte come la ricerca di un utente o della sua password. La struttura dati deve poter gestire la concorrenza perché essendo un oggetto unico condiviso tra tutti i worker allora si possono creare situazioni in cui più thread cerchino contemporaneamente farci delle modifiche (es. aggiungere un nuovo utente dopo una registrazione). Per questo ho deciso di sincronizzare tutti i metodi che lavorano sull’HashMap; questo potrebbe rallentare le prestazioni ma permette di non aver problemi con accessi in contemporanea.
**- word =** è la parola da indovinare, questa cambia ogni 24h
**- words =** path che indica il file che contiene il vocabolario
**- startTimeWord =** indica il momento in cui è stata aggiornata l’ultima volta word
**- fixedTime =** indica 24h il tempo che deve passare prima di aggiornare word

**Costruttore:**

```
CentralServer()
```

Questo è il costruttore, come prima cosa controlla se esiste un file json di utenti che il server crea quando riceve un segnale di interruzione; se il file esiste allora chiamo il metodo recuperaUtenti passandogli il path del file json, così da ricostruirmi l’HashMap contenente gli utenti.

La stessa cosa la faccio con un altro file json, questo contiene word e startTimeWord, l'unico motivo per cui ho separato le cose è per semplicità di scrittura e lettura sul file e inoltre sono cose concettualmente diverse, da una parte informazioni sugli utenti dall'altra informazioni sulla parola con cui giocare.
Quindi ogni volta che il server riparte il costruttore con questi controlli si accorge se c'è stato un salvataggio di dati precedentemente e in caso affermativo andrà a chiamare i due metodi recuperaUtenti e recuperaWord recuperando tutte le informazioni necessarie.
Fatto questo a prescindere dai controlli chiamo il metodo generateNewWord che si preoccupa di generare una nuova parola e di cambiarla ogni 24h. Questo metodo contiene altri controlli spiegati successivamente.

**Metodi:**

`public synchronized boolean ExistsUserName(String nomeUt)`
Metodo che controlla se nell'HashMap esiste un utente con quel nome, questo si fa controllando se alla chiave nomeUT è associato qualche utente.

`public synchronized boolean verificaPassword(String nomeUt, String passwordUt)`
Metodo a cui viene passato un nome utente (ricordo che questo è univoco), cerco utente associato a quel nome e controllo se la sua password corrisponde a quella che mi è stata inviata

`public void generateNewWord()`
Metodo in cui inizializzo scheduler come `ScheduledExecutorService`, questo servizio mi permette di pianificare l'esecuzione di un certo task. Adesso definisco un oggetto runnable **chooseWordTask** che manderò in esecuzione periodicamente; questo oggetto all'interno del suo run esegue una sola cosa, cioè la chiamata del metodo `chooseWord()` che attraverso la generazione di un numero random x identifica una riga del file words (vocabolario) e aggiorna la parola word con la parola identificata alla riga x.
Prima di lanciare il thread faccio un controllo con l'obiettivo di far durare ogni word da indovinare 24h a prescindere che ci siano cadute del server o meno, dopo 24 ore dalla sua generazione word deve cambiare.
Inizializzo tempoRimasto a 0 e calcolo:
StartTimeWord + 24 ore
- caso 1
se questo valore è maggiore del momento attuale (nel codice now) allora significa che quella parola è stata generata meno di 24 ore fa e quindi bisogna continuare ad utilizzare quella. Per capire per quanto tempo ancora deve essere in utilizzo aggiorno tempoRimato (0 di default) in questo modo:
tempoRimasto = tempo in cui è stata generata word + 24h – now
- caso 2
la somma sopra risulta essere minore di now, questo singifica che sono passate più di 24h e devo cambiarla; non faccio niente lascio tempoRimasto = 0

Adesso verrà lanciato il thread con il valore temporimasto in modo corretto, ovviamente verrà passato **chooseWordTask** che aggiornerà word.
Ogni volta che word viene aggiornata viene aggiornata anche StartTimeWord.

`public void recuperaUtenti(File jsonFileUtenti)`
Per deseriallizzare l'hashMap (contenente gli utenti) ho utilizzato un metodo diretto offerto dalla classe Gson: `fromJson.` Al metodo va passato file da cui leggere e specificato (grazie a **TypeToken)** che tipo di oggetto andrà a deserializzare; specificato che l'oggetto (nel mio caso) è un HashMap<String, Utente> il metodo riuscirà a ricostruirla correttamente e verrà assegnata alla varabile HMUtenti.

`public void recuperaWord(File jsonFile)`
In questo metodo deserializzo manualmente essendo un caso molto semplice, solo due stringhe.

`public void saveServer()`
Prima di tutto metto tutti gli utenti offline, questo metodo viene chiamato da terminationHandler quando

server sta per chiudersi e quindi quando poi andrà ad accendersi di nuovo dovrà avere tutti gli utenti offline e per questo prima di memorizzarli li setto offline. Se non facessi questo settaggio all'accensione successiva del server nel recuperare i dati ci sarebbero utenti che risulterebbero online (cosa non vera) e questi non potrebbero giocare risultando già connessi con il gioco.

Fatto questo serializzo HashMap su un nuovo file `utenti.json` (se dovesse già esistere lo svorscrivo e questo va bene dovendomi salvare solo i nuovi dati). Per questa serializzazione utilizzo GsonBuilder che attraverso il suo metodo `toJson` mi permette di creare una Stringa in formato json in modo diretto, basta passargli l'oggetto (in questo caso HMUtenti)

Come ultima cosa serializzo word e startTimeWord manualmente nel file `word.json`

## `Partita`

Partita contiene utente che sta giocando e word che sta cercando di indovinare (passati per parametro da Worker). Attraverso i metodi di Partita, Worker capisce se utente ha già giocato ( in partita viene confrontata word con l'ultima parola con cui ha giocato utente), se un tentativo è valido e ottiene i suggerimenti da inviare al Client. Inoltre, Partita ha un altro ruolo fondamentale, quello di tenersi traccia della partita via via e di memorizzarsi le informazioni in una stringa che verrà poi inviata nel gruppo se richiesto (l'invio verrà fatto da worker)

## `Utente`

Utente contiene tutte le informazioni relative all'utente e i suoi metodi permettono aggiornare o recuperare le sue informazioni (es. settare utente online o controllare se è online etc.).
**variabili e strutture dati:**
- Per la disposizione dei risultati ho deciso di tenere un array grande 13 settato inizialmente tutto a 0 e ogni volta che mi arriva una vittoria incremento di uno il valore alla posizione dell'array uguale al tentativo con cui ho vinto arr[tentativi] += 1

**Metodi:**

`public void aggiornaValori(String word, Integer tentativi, String esito)`
Metodo che permette di aggiornare le statistiche dell'utente e la parola con cui ha giocato; in base ai valori che gli vengono passati aggiorna tutte le variabili di utente riguardanti le sue statistiche (es. partite vinte, perse, etc.)

`public String getStatistics()`
Questo metodo attraverso le variabili interne di Utente, aggiornate durante le varie partite, crea una stringa che rappresenta le statistiche generali: percentuale di vittoria, disposizione delle partite, etc. Il metodo ritorna la stringa generata.
Scelta personale:
- ultima sequenza di vittorie = nel mio progetto rappresenta l'ultima sequenza conclusa di vittore, questa sequenza parte quindi da o dopo l'ultima perdita. Una sequenza di vittorie non conclusa, quindi che può continuare a crescere, non viene considerata last streak.
- miglior sequenza di vittorie = questa è semplicemente la più lunga anche se non attualmente finita.

`TerminationHandler`
A questa classe (che viene lanciata da ServerMain) viene passato l'oggetto centralserver così che potrà richiamare i suoi metodi all'interno di run per il salvataggio dei dati.

**Metodi:**

`public void run()`
All'interno run chiamo il metodo centralserver.saveServer (visto sopra) che andrà a scrivere su file tutte le informazioni utili da non perdere.
Dopo di che fa terminare il pool di thread dandogli un po' di tempo prima di chiudersi definitivamente

## FLUSSO DI ESECUZIONE

Si attiva ServerMain, si attiva ClientMain e tra questi si andrà ad istaurare una connessisone TCP tramire socket. In entrambi verranno inizializzati DataInputStream e DataOutputStream che ci permeteranno l'invio e la ricezione di messaggi tra il client e il server.
Vediamo il flusso di esecuzione prima da parte del client e poi da parte del server.

**Oss** il funzionamento di alcuni metodi sono stati specificati sopra e non verranno ripetuti, altri invece verranno spiegati più nel dettaglio in questa parte. Altri ancora essendo molto semplici non hanno bisogno di spiegazione.

---------- Client ----------

`ClientMain`

Una volta stabilita la connessione tramite socket, ClientMain parte con un ciclo while in cui richiede all'utente cosa vuole fare ("**Menù iniziale"**) catturando le diverse possibilità attraverso 3 if e chiamando quindi gli adeguati metodi:
**1**. Registrazione => `public static void registrazione`
**2**. Login => `public static void login`
**0**. Exit => `public static void chiudi` (visto precedentemente)

Adesso vediamo il metodo registrazione che comprende poi anche la chiamata al metodo login; infatti, possiamo accedere in due modi al metodo login, o direttamente dal "Menù iniziale" o subito dopo aver eseguito la registrazione a wordle con successo.

**Oss** (per non ripetere) in ogni momento (es. registrazione o dentro menù di wordle) ho sempre la possibilità di premere 0 e tornare al Menù iniziale. Questa cosa viene gestita semplicemente inviando al server "back" e facendo return. Server alla lettura di "back" in modo speculare farà return. L'unico momento in cui questo non è possibile è durante una partita.

**Utente ha digitato 1 nel Menù iniziale**
`public static void registrazione`
Entro all'interno di un while, chiedo di inserire nome per registrarsi, faccio qualche controllo sul nome digitato (almeno 5 caratteri senza spazi vuoti), lo invio al server e aspetto una risposta; se server mi dice che è corretto ("ok") allora setto la guardia del while (unicName) a true che mi farà uscire dal ciclo e mi farà

passare alla seconda parte della registrazione.
Uscito dal while (trovato nome utente valido) allora procedo con la richiesta della password e dopo gli adeguati controlli (almeno 5 caratteri senza spazi vuoti) la invio al server e aspetto la conferma dal server che, in caso affermativo, mi restituirà la stringa "registrazione avvenuta con successo".

Dopo la stampa di avvenuta registrazione chiamo direttamente il metodo login; dopo di questo return così che, quando login terminerà tornerò al "Menù inziale".

**Utente si è registrato o utente ha digitato 2 dal "Menù iniziale**

```
public static void login(DataInputStream in, DataOutputStream out)
```

Entro dentro il while chiedo di inserire username, aspetto risposta "ok" da server, in caso affermativo chiedo password e nuovamente aspetto "ok" da server. Uscito dal while (login avvenuto con successo) aspetto un messaggio di conferma da parte del server dell'avvenuto login e lo stampo ("Benvenuto in Wordle!"). In questa fase (scelta personale) server mi risponderà "ok" all'invio del nome utente se questo risulta essere presente tra i registrati e risulta offline (non c'è quindi modo di distinguere i due casi).

Dopo di che l'utente viene aggiunto al gruppo multicast:
- viene creato oggetto ClientMuticast **gruppomulticast** che farà unire utente al gruppo
- viene avviato un thread che prende questo oggetto come argomento

Adesso che utente si è unito al gruppo viene chiamato il metodo wordle;

Alla fine del metodo c'è un return che farà tornare a Menù iniziale grazie al return del metodo chiamante (registrazione -> Menù iniziale)

**Utente ha effettuato il login con successo**

```
public static void wordle(DataInputStream in, DataOutputStream out, ClientMulticast gruppomulticast)
```

Entra in un ciclo while e chiede a utente cosa vuole fare:
**1**. Giocare => `public static void gioca`
**2**. Ottenere statistiche => `public static void statistiche` (visto precedentemente)
**3**. Mostrargli le partite degli altri utenti => `gruppomulticast.showMeSharing();` (visto precedentemente)
**0**. Uscire e tornare al menù iniziale ("**Menù di Wordle**") => faccio solo return tornando al Menù inziale.

L'utente digita una scelta che verrà inviata al server e in base a questa verrà chiamato il metodo adeguato quando necessario.

**Oss** alla fine di ogni ramo intrapreso (tranne se digitiamo 4) verrà richiesto ogni volta di inserire una scelta (**Menù di Wordle**); solo la scelta 4 mi permetterà di uscire dal while.

Quando utente deciderà di uscire (digita 4 dal **Menù di Wordle**) tornerà al "**Menù iniziale"** grazie ai vari return dei metodi chiamanti (login -> registrazione -> Menù iniziale)

**Utente ha digitato 1 nel "Menù di Wordle"**

```
public static void gioca(DataInputStream in, DataOutputStream out)
```

Prima di partire il metodo aspetta un messaggio da server che gli dice se ha già giocato o no con quella parola (se riceve "ok" prosegue con il gioco altrimenti stampa e return)

Se utente non ha già giocato allora entra in un ciclo while da 12 iterazioni (il numero dei tentativi). Ad ogni iterazione utente inserisce il proprio tentativo e questo verrà inviato al server. Adesso si aspetta la risposta da parte del server che può essere di 3 tipi:
1) "**ok" =>** allora tentativo valido, aspetto nuovo messaggio (che sarà la stringa rappresentante i corretti suggerimenti) e la stampo. Nel caso di vittoria la stringa "suggerimenti" che arriva è questa: " **\n!!! HAI VINTO !!! \n COMPLIMENTI** " con un if controllo e se sono in questo caso esco dal while altrimenti continuo a ciclare.

2) "**no**" (nel codice solo else) allora **=>** faccio un continue, così da non incrementare guardia del while (non conta come tentativo)

Uscito dal while (nei due casi di vittoria o perdita) viene chiesto a utente se vuole condividere la partita, in caso affermativo client invia "cond" al server, "NoNcond" altrimenti.

**OSS.** In questa fase non c'è la possibilità di uscire o tornare indietro, essendo un gioco molto breve basterà provare qualche tentativo in modo casuale e in pochi secondi saremo fuori.

**Utente ha digitato 2 nel "Menù di Wordle"**

```
public static void public static void statistiche(DataInputStream in, DataOutputStream out)
```

Riceve statistiche dal server

**Utente ha digitato 3 nel "Menù di Wordle"**

```
gruppomulticast.showMeSharing();
```

Visto in precedenza.

**Adesso vediamo cosa succede dalla parte del server**

```
------------------------------------ Server ----------------------------------------
```

`ServerMain`

Attivato ServerMain e stabilitata la connessione la classe crea l'oggetto centralServer che conterrà tutte le informazioni utili come specificato sopra.
Creato questo oggetto lancio un pool di thread (worker) e ad ognuno viene passato:
- la socket per comunicare con client
- l'oggetto centralServer
- indirizzo muliticast (come stringa)
- porta multicast

Oltre al pool di thread viene avviato l'handler di terminazione a cui passa:
- il pool di thread
- il tempo che dovrà attendere prima di chiudere i thread Worker
- l'oggetto centralserver (da cui chiamerà il metodo saveServer)
- la socket che verrà chiusa

Vediamo cosa succede dentro Worker lanciato da ServerMain

`Worker`

`Worker(Socket Socket, CentralServer centralserver, String indirizzoMulticast ,int portaMulticast )`
Questo è il costruttore, vedremo poi come vengono utilizzati questi parametri.

`public void run()`
Inizializzo DataInput e DataOutput per comunicare con client e dopo di che entra dentro un while in cui aspetta un messaggio da parte del client, a seconda di ciò che client invia viene chiamato il metodo adeguato. Il funzionamento di queste fasi è speculare a quello che succede in client.
**Menù iniziale** (lato server):
**1**. Registrazione => `public static void registrazione`
**2**. Login => `public static void login`
**0.** Exit => break e esco dal while, uscito da qui entro in finally che rimuove utente come online e chiude socket

Adesso vediamo il metodo registrazione che comprende poi anche la chiamata al metodo login; infatti, possiamo accedere in due modi al metodo login, o direttamente se client desidera fare login o subito dopo la registrazione

**Oss** (per non ripetere) in ogni momento (es. registrazione o dentro menù di wordle) client ha sempre la possibilità di premere 0 e tornare al Menù inziale. Questa cosa viene gestita semplicemente ricevendo da client "back" e facendo return in modo speculare a client. L'unico momento in cui questo non è possibile è durante una partita.

**Caso in cui client ha inviato 1**
`public void registrazione(DataInputStream in, DataOutputStream out)`
Il metodo entra dentro un while aspetta un possibile nome utente, una volta ricevuto attraverso il metodo existUserName di centralserver controlla se questo nome utente esiste già, in caso negativo invia "ok" ( non esiste quindi va bene per nuova registrazione). Se inviato "ok" esco dal while.
Controllato il nome utente adesso si mette in attesa di una password che verrà accettata immediatamente quindi nessun while (un piccolo controllo verrà fatto lato client).

Adesso server può quindi registrare un nuovo utente chiamando centralserver.aggiungiUtente(username, password) e inviare un messaggio di conferma che la registrazione è andata a buon fine.

Viene poi chiamato il metodo login; dopo di questo c'è return così che, quando login terminerà tornerò al "Menù inziale" (lato server).

**Caso in cui client ha inviato 2 o ha appena effettuato registrazione**
`public void login(DataInputStream in, DataOutputStream out)`

Il metodo entra in un while da cui uscirà solo quando client invierà un nome utente valido.
Si mette in attesa di un nome utente, appena lo riceve chiede a centralserver.ExistName se questo nome utente è esistente o meno, se lo è invio "no" altrimenti "ok" e esco dal while. (caso affermativo significa che effettivamente è un utente registrato)
Adesso viene fatta la stessa cosa con la password con la differenza che per la correttezza della password viene chiamato centralserver.verificaPassword(username, password)

Quando tutti i controlli saranno passati allora il metodo invia un messaggio di conferma, e setta utente online in centralserver (utente.putOnline).
Adesso utente è online e siamo nel "**Menù di Wordle**":

Entro in un ciclco while e aspetto un messaggio da client:
**1**. Giocare => `public static void gioca`
**2**. Ottenere statistiche => `public static void statistiche`
**0**. Uscire e tornare al menù iniziale ("**Menù di Wordle**") => esce dal while, mette utente offline e retuna;

Client invia la scelta fatta dall'utente e in base a questa verrà chiamato il metodo adeguato quando necessario.

**Caso in cui client ha inviato 1**
```
public void gioca(DataInputStream in, DataOutputStream out)
```

Prima cosa il metodo prende la parola da centralserver, e crea un oggetto partita (worker gestirà la partita chiamando i metodi di partita) new Partita(utente, word)

Controlla che utente non abbia già giocato (chiedendo a partita) in caso affermativo inizia la fase di gioco altrimenti invia "no" al client.

Caso affermativo metodo entra dentro while (lungo 12 come i tentativi), ad ogni ciclo aspetta un messaggio da parte del client (tentativo), controllo se è corretto chiedendo a partita (risponde con valore booleano) in caso affermativo lo comunica al client che potrà mettersi in attesa dei suggerimenti.
Se il tentativo è valido il metodo chiede a partita i suggerimenti (partita.suggerimenti(tenatativo) e partita può rispondere in due modi:
1) Stringa con "x..?..+" che indicano i vari colori in base al tentativo =>allora viene inviato (da Worker) a client la stringa suggerimenti

2) Stringa "vinto " => viene inviato a client (da Worker) la stringa "speciale" che indica la vittoria:" **\n!!! HAI VINTO !!! \n COMPLIMENTI "**.

Se il tentativo non era valido allora viene inviato "no" e viene fatto continue.

Finito il while come prima cosa viene fatta una chiamata a utente per aggiornare le sue statistiche passandogli: utente.(aggiornaValori, end, "perso"/"vinto") a seconda se utente ha vinto o meno (per fare questo controllo basta vedere il numero dei tentativi fatti)
**end** rappresenta i tentativi fatti (iterazioni nel while)

Adesso il metodo aspetta che client gli comunichi se vuole condividere o meno la partita in caso affermativo ("client invia 1") allora viene chiamato il metodo inviopartita(partita) (visto sopra).

Partita condivisa o no il metodo a questo punto returna.

**Caso in cui client ha inviato 2**
```
public void statistiche(DataInputStream in, DataOutputStream out)
```

Molto banalmente attraverso il metodo utentegetPartita() ottiene la stringa rappresentate le statistiche e le invia a client.

## COMPILARE ED ESEGUIRE

### Compilazione

1. Aprire il terminale
2. Posizionarsi nella cartella “ProgettoWordle”
3. (client) eseguire il comando: javac .\client*.java
4. (server) eseguire il comando: javac -cp .\server\resources\gson-2.10.jar .\server*.java

### Esecuzione

Server:

1. Aprire il terminale
2. Posizionarsi nella cartella “ProgettoWordle”
3. Eseguire il comando: java -cp "server\resources\gson-2.10.jar;. " server.ServerMain

Client:

1. Aprire il terminale
2. Posizionarsi nella cartella “ProgettoWordle”
3. Eseguire il comando: java client.ClientMain

### Esecuzione utilizzando i JAR

Server:

1. Aprire il terminale
2. Posizionarsi nella cartella “ProgettoWordle”
3. Eseguire il comando: java -jar Server.jar

Client:

1. Aprire il terminale
2. Posizionarsi nella cartella “ProgettoWordle”
3. Eseguire il comando: Java -jar Client.jar