Laboratorio di Reti
WORDLE: un gioco di parole 3.0
Progetto di Fine Corso A.A. 2022/23

Federico Fornaciari - 619643

Federico Fornaciari - 619643

Scelte implementative

Struttura del progetto

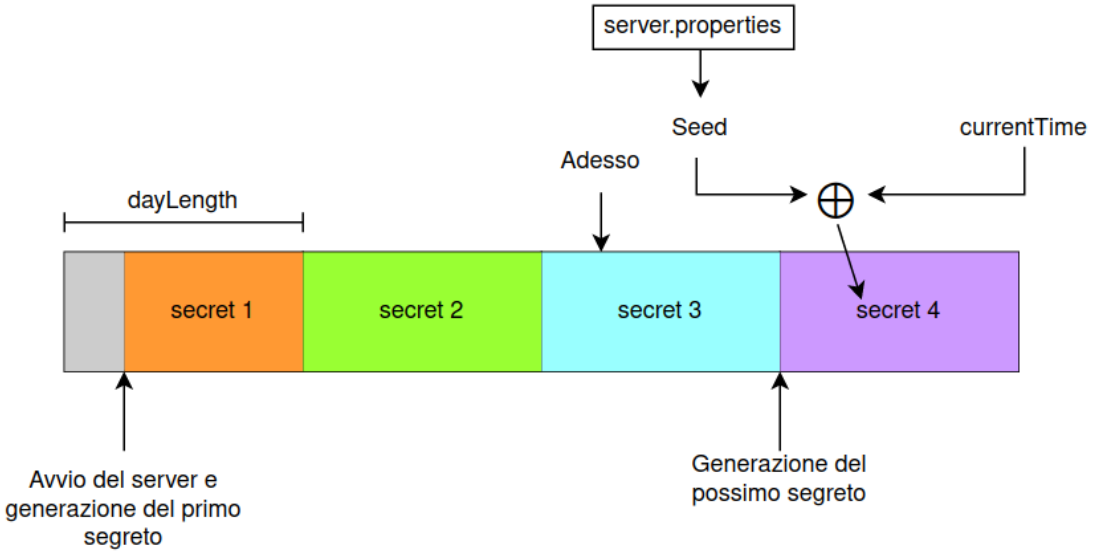
WordleClient:

- *ClientMain.java* contiene tutto il codice tranne che per la gestione delle notifiche
  - *main()* carica il file *client.properties*, crea il socket, *BufferedInputStream* e *BufferedOutputStream* sul socket e poi inizia a chiedere cosa fare all'utente chiamando le funzioni relative, descritte nella sezione della comunicazione client-server, con uno switch
- *NotificationsThread.java* estende Thread, si occupa di ricevere notifiche dal gruppo multicast, memorizzarle in una lista e stamparle sulla CLI quando l'utente lo richiede
  - il costruttore:
    - crea un *MulticastSocket*
    - chiama il metodo *joinGroup* per entrare nel gruppo multicast dedicato alle notifiche
    - inizializza la lista delle notifiche
    - si salva l'username per poter escludere le notifiche mandate dall'utente che ha fatto il login
  - *run()*: ha un while che attende nuove notifiche sul *MulticastSocket* e salva nella lista i *DatagramPacket* ricevuti
  - *safeStop()*: usato per terminare l'esecuzione del thread, dato che il metodo *Thread.stop()* è deprecato:
    - termina il metodo *run()* impostando la variabile volatile *executing = false*
    - esce dal gruppo multicast
    - chiude il socket
  - *printNotifications()*: per ogni notifica nella lista:
    - prende i dati del *DatagramPacket* con il metodo *getData()*
    - prende l'username del mittente nella seconda parte del buffer
    - controlla se è lo stesso dell'utente che ha eseguito l'accesso, in caso scarta la notifica
    - altrimenti stampa sulla CLI il nome dell'utente che ha inviato il pacchetto e per ogni tentativo stampa il match con la secret word

WordleServer:

- *ServerOps.java* contiene un elenco di metodi statici utili alle altre classi del server
- *ConnectedUser.java* si occupa di gestire la comunicazione con un singolo client
  - legge 1 byte contenente l'istruzione da eseguire
  - chiama la funzione relativa
  - chiama *os.flush()* per mandare tutto lo stream generato dalla funzione chiamata al client
- *SecretGenerator.java*
  - Thread che si occupa di generare le secret word ogni giorno, con lunghezza di un giorno definita in *server.properties*

- prende i millisecondi passati dal 1 gennaio 1970 (anche se Java implementa le date con delle classi, mi è più comodo tenere tutto in variabili *long* dato che devo farci molte operazioni e non mi serve stampare date formattate)
- li arrotonda per eccesso alla lunghezza di 1 giorno (momento in cui generare il prossimo segreto)
- calcola quanto manca a quel momento e fa una sleep
- genera la secret word chiamando *ServerOps.newSecret()*.
  questo metodo userà il giorno in XOR con il seed in *server.properties* per scegliere la parola, in questo modo anche riavviando il server la parola generata sarà la stessa.
  Il seed non deve essere diffuso agli utenti, altrimenti questi potrebbero usarlo per sapere quali saranno le parole future



- *ShutdownListener.java*
  - Chiamato dallo *shutdownHook* quando il server termina, salva *users.json* e chiude i socket
- *User.java*: contiene lo stretto necessario da memorizzare di un utente in *users.json*, ha anche dei metodi che permettono alle altre classi di operare con le statistiche e i dati dell'utente
  quindi non mi memorizzo il resoconto di ogni partita vinta dall'utente ma tengo solo l'ultima nella memoria della sessione, l'utente potrà condividerla solo nella sessione stessa in cui ha giocato la partita
- *ServerMain.java* contiene solo la funzione *main* e si occupa di inizializzare il server e attendere nuove connessioni dai client:
  - inizializza il *CachedThreadPool* con i task associati ai client
  - carica il file *server.properties*
  - carica il file con gli utenti
  - crea il *ServerSocket* che accetta le connessioni con i client
  - crea il *DatagramSocket* che si occupa di mandare le notifiche in multicast ai client

- Imposta lo *shutdownHook* per la *SIGINT* in modo che esegua le istruzioni definite nel metodo *run()* di *ShutdownListener* quando il server termina (quando si preme CTRL+C)
- Avvia il thread che si occupa di generare le secret word, il costruttore genererà il primo segreto in modo sequenziale (per evitare che i task *ConnectedUser* abbiano bisogno della secret word prima che venga generata)
- Si mette in attesa di nuove connessioni sul socket e le passa al costruttore di *ConnectedUser* che viene passato al threadpool

NIO multiplexing o IOStream?

Multiplexing NIO:

- È pensato per operare su un singolo thread → più efficiente dato che non c'è *context-switching* ma non sfrutta tutti i core del server
- È più efficiente essendo a livello più basso, però ha bisogno, per ogni operazione di ogni client connesso, di poterla interrompere e riprenderla (o ricominciarla) in caso non abbia ancora ricevuto tutti i byte necessari per quell'operazione, dato che ogni *SocketChannel* deve essere non bloccante.
  Questo è complesso da implementare e rende il codice poco pulito
  Le operazioni *register*, *login*, *receiveWord* che devono ricevere più di 1 byte (quindi per le quali la notifica del *Selector* non indica necessariamente che il server ha ricevuto tutti i dati dal client) avranno bisogno di:
  - mettere nel *ByteBuffer position = limit* e *limit = capacity* (una specie di *flip* al contrario) dato che deve essere predisposto a continuare a leggere dal *SocketChannel* nel punto in cui mancano informazioni
  - annullare tutte le operazioni fatte fino a quel momento (o non farne nessuna fino a quando non si sono ricevuti tutti i dati)

  Il client avrebbe meno problemi dato che potrà avere un *SocketChannel* bloccante, però dovrebbe comunque controllare ogni volta se ha ricevuto tutti i dati.
  Tutto questo può rendere il codice più complesso, quindi meno leggibile e quindi più prono a contenere bug.

Con IOStream, gestendo ogni client con un task nel server e operazioni bloccanti, posso occuparmi solo delle transazioni tra singolo client e server senza preoccuparmi della concorrenza tra i *Socket* associati ai vari client, dato il context switching sarebbe gestito dal threadpool in modo trasparente, questo mi permette di avere un codice più pulito e self-documenting
Dato che i miglioramenti in efficienza di NIO (se presenti, dato che non sfrutta tutti i core) sono trascurabili, scelgo di gestire la connessione con IOStream.

Federico Fornaciari - 619643

Strutture dati

- *client.properties*: contiene tutti i parametri del client, cioè IP e porta del server e del gruppo multicast a cui unirsi per ricevere le notifiche
- *server.properties*: contiene tutti i parametri del server, in particolare:
  - la lunghezza di un giorno in secondi
  - il seed per la generazione casuale delle secret word
  - IP e porta del server e del gruppo multicast sul quale mandare le notifiche ai client
- *users.json*: file contenente una hashmap serializzata di oggetti di tipo *User*
- *words.txt*: fornito con la consegna, per migliorare l'occupazione di memoria non viene caricato ma ne vengono lettere singole parole mediante RandomAccessFile dai metodi di ServerOps:
  - newSecret: per la generazione della secret word, accedendo in una posizione pseudocasuale, O(1) in spazio e in tempo
  - WordValid: quando un utente inserisce una parola, per controllare se esiste, viene fatta una ricerca binaria nel file, con complessità O(log n) in tempo e O(1) in spazio, caricando l'intero file in memoria invece si avrebbe una complessità O(1) in tempo (usando una hashmap) e O(n) in spazio

Federico Fornaciari - 619643

Descrizione dei thread e delle tecniche di sincronizzazione usate

WordleClient:

- *main* esegue tutte le operazioni tranne che per la ricezione di notifiche
- un thread di tipo *NotificationsThread* gestisce la ricezione di notifiche, attende su un *MulticastSocket* e salva in una *BlockingQueue* i *DatagramPacket*, è necessario che sia thread-safe perché mentre questo thread aggiunge notifiche alla lista, il thread *main* le può rimuovere chiamando il metodo *printNotifications()*
  Questo thread può essere terminato chiamando *safeStop()* che utilizza una variabile volatile, necessaria per garantire la visibilità delle modifiche in memoria da parte di altri thread, per terminare l'esecuzione del ciclo while nel metodo *run()*

WordleServer:

- *main*, dopo aver inizializzato le strutture necessarie al funzionamento del server in modo sequenziale e aver creato un thread di tipo SecretGenerator, si occupa solo di stare in ascolto sul welcome socket di nuove connessioni dai client, le passa a un nuovo oggetto di tipo *ConnectedUser* che viene passato come task a una *CachedThreadPool*
- *secretGenerator*, nel costruttore, (eseguito in modo sequenziale dal *main*), viene generato il primo segreto, in questo modo garantisco che quando il server inizierà a accettare connessioni avrò una secret word pronta
  Successivamente ogni giorno crea un nuovo segreto, la generazione deve essere atomica rispetto ai *ConnectedUser* dato che questi potrebbero leggere il segreto nuovo ma la data del segreto vecchio, quindi le operazioni sul secret le faccio con metodi *synchronized*
- I task *ConnectedUser* associati ai client sono gestiti da una *CachedThreadPool*, questi attendono comunicazioni dal client, leggono il primo byte e in base a quello scelgono l'istruzione da eseguire
  - Per memorizzare gli utenti utilizzo una *ConcurrentHashMap* "*users*" che mi permette di operarci atomicamente
  - il metodo *register*, dopo aver verificato che la password non è vuota, chiama il metodo *ServerOps.checkIfUserExistsAndCreate()* che chiama il metodo atomico *putIfAbsent()* su *users,* restituendo l'esito dell'operazione
  - in *login* non sono sufficienti le operazioni atomiche su *users* per verificare se un utente ha già eseguito l'accesso, per questo uso un *synchronized hash set* che tiene conto di tutti gli utenti loggati.
    Inoltre il controllo dell'accesso eseguito viene fatto in un blocco *synchronized* sulla lock implicita dell'utente, in questo modo l'esecuzione non viene serializzata (a parte la modifica alla *ConcurrentHashSet*) a meno che non avvengano più accessi contemporanei con lo stesso account
    Dopo aver fatto questo è garantito che tutte le operazioni su un singolo utente vengano eseguite da un thread solo, quindi non sono necessari ulteriori sistemi di sincronizzazione sul singolo utente

Federico Fornaciari - 619643

Comunicazione Client - Server

Nel server, dopo che il welcome socket ha ricevuto una richiesta di connessione da un client, viene creato un task di tipo *ConnectedUser* che si occupa di gestire la comunicazione con quel client
La parte principale del codice per la comunicazione tra un client e il server è quindi locata in *ClientMain.java* per il client e *ConnectedUser.java* per il server

Tutte le comunicazioni sono strutturate in modo che il client manda al server tutti i dati necessari per gestire la richiesta e questo risponde con tutti i dati necessari al client, non ci sono quindi comunicazioni con più step
In particolare, i messaggi sono strutturati come:
Client → Server: 1 byte che indica l'operazione da svolgere + altri dati (se necessari)
Server → Client: 1 byte (se necessario) esito dell'operazione + altri dati (se necessari)

Le stringhe di lunghezza variabile sono inviate come: 1 byte contenente la lunghezza + byte contenenti la stringa
Gli interi sono separati in 4 byte con operazioni di shift bitwise

Di seguito elencate le operazioni che il client può richiedere al server, per ognuna prima il client manda un byte contenente il numero relativo all'operazione e poi eventualmente altri dati:
register [1]:

- Client: legge username e password da CLI e li manda al server
- Server: controlla se l'utente è già registrato o se la password è vuota e risponde al client

login [2]:

- Client: legge username e password da CLI e manda al server 2 + stringa username + stringa password
- Server:
  - controlla se l'utente è registrato, se la password corrisponde e se l'utente ha già eseguito l'accesso e risponde al client
  - se l'accesso è avvenuto con successo si memorizza l'utente che ha eseguito l'accesso e segna l'utente come loggato in modo che non possa rifare il login con un'altra sessione contemporaneamente
- Client: se l'accesso è avvenuto con successo:
  - mette isLoggedIn a true
  - crea e avvia il thread per la ricezione delle notifiche multicast (non era possibile crearlo prima perché il costruttore richiede il nome dell'utente per escludere le notifiche relative alle sue partite)

logout [3]:

- Client: dato che il metodo stop() è deprecato, termina il notificationsThread usando il metodo safeStop()
- Server:
  - chiamo stopPlaying() per terminare la partita in corso dato che ogni tentativo d'indovinare una parola si intende concluso se l'utente esegue il logout
  - rimuove l'utente dalla lista di utenti loggati

playWordle [4]:

- Server: controlla se l'utente è loggato e se non ha già giocato questo giorno e risponde al client, si memorizza che il giocatore ha giocato la partita di oggi e inizializza le variabili necessarie alla partita

sendWord [5]: (receiveWord() per il server)

- Client: legge la parola che l'utente vuole inserire, controlla che sia di 10 caratteri e la manda al server
- Server:
  - riceve la guess dal client
  - controlla che sia nel dizionario, in caso positivo:
    - confronta la guess con la secret word chiamando matchSecret
    - se user ha vinto chiama *addWonMatch(nTrial)* per aggiungere una partita vinta a user al tentativo *nTrial*+1 (per l'array della guess distribution)
    - se user ha perso chiama *addLostMatch()* per aggiungere una partita persa all'utente e resettare la streak
    - invia al client un byte che indica se ha vinto, perso o può continuare a giocare
  - in caso negativo lo comunica al client

stopPlaying [6]:

- Il server controlla se l'utente sta giocando e in caso positivo chiama *addLostMatch()*

share [7]:

- Se l'utente è ancora in partita o non ha una partita salvata in sessione da condividere non fa niente, altrimenti chiama *ServerOps.share(user, matches)* che crea un nuovo *DatagramPacket* con 120 caratteri (12 tentativi * parole da 10 lettere) + l'username e lo manda sul socket UDP

statistics [8]:

- Server: manda 4 + 12 interi contenenti le statistiche all'utente
- Client: le stampa su *System.out*

Come descritto in share (7) la funzione del server che si occupa di mandare le notifiche al gruppo multicast è *ServerOps.share(user, matches)*, questa funzione:

- prende dall'oggetto user il nome dell'utente
- crea un nuovo DatagramPacket
- inserisce nel DatagramPacket 120 byte della partita dell'utente
- inserisce dopo quei 120 byte il nome dell'utente
- invia il pacchetto sul socket multicast

Federico Fornaciari - 619643

Compilazione e esecuzione

Per avviare il programma as is è possibile eseguire da terminale:
`java -jar WordleServer.jar` una sola volta, per avviare il server, è importante avviare il server prima di avviare i client
`java -jar WordleClient.jar` da eseguire per ogni client che si vuole avviare
assicurandosi che:
client.properties sia nella cartella dalla quale si avvia WordleClient.jar:
gson.jar, words.txt e server.properties siano nella stessa cartella dalla quale si avvia WordleServer.jar

Verrà stampata sulla CLI una guida sulle operazioni che è possibile eseguire in quel momento, è sufficiente scrivere il numero dell'operazione che si vuole eseguire e premere invio

Per compilare:
posizionarsi nella cartella del progetto, dove è la libreria gson.jar e eseguire i comandi:
`mkdir build` crea la cartella dove andranno i pacchetti compilati
`javac -d build -cp gson.jar wordleserver/*.java` compila nella cartella build il pacchetto wordleserver usando la libreria esterna gson

- `javac`: eseguibile del compilatore del jdk (java development kit)
- `-d build`: directory di destinazione del pacchetto
- `-cp gson.jar`: classpath per librerie esterne, in questo caso gson
- `wordleserver/*.java`: cosa compilare, quindi tutti i file .java nella cartella wordleserver

`javac -d build wordleclient/*.java` compila nella cartella build il pacchetto wordleclient

È poi possibile eseguirli con i comandi:
Server: `java -cp gson.jar:build wordleserver/ServerMain`

- `java`: strumento di esecuzione di file java compilati
- `-cp gson.jar:build`: classpath, quindi la libreria gson nella cartella corrente e la cartella con il pacchetto contenente tutti i file .class generati (su Windows le classpath dovranno essere separate da `,` invece che da `:`)
- `wordleserver/ServerMain`: percorso per la classe contenente il main, non è necessario indicare il .class

Client: `java -cp build wordleclient/ClientMain`
assicurandosi sempre che i file client.properties, server.properties, words.txt e gson.jar siano presenti