“LABORATORIO DI SISTEMI OPERATIVI A.A. 2021-22”

NOME DEL GRUPPO: NovaTec Paper
EMAIL DEL REFERENTE DEL GRUPPO: carol.franceschiello@studio.unibo.it
COMPONENTI DEL GRUPPO:
Carolina Franceschiello - 901222
Matilde Bertazzoni – 900629
Marianna Gimigliano – 915343
Marco Amodeo - 942815

1. Descrizione del progetto:

(a) Architettura generale

Il progetto Novatec Paper, implementato in Java, ha l’obiettivo di creare un file server che permette a più utenti di connettersi contemporaneamente per poter condividere, visualizzare ed eventualmente modificare i file di testo. Gli utenti accedono a questi file dalla propria macchina attraverso un’applicazione client-server, in questo modo gli utenti possono:

- creare nuovi file;
- leggere i file esistenti;
- modificare i file esistenti;
- rinominare i file esistenti;
- eliminare i file esistenti;

Nel momento in cui un utente richiede di leggere un file, il server apre una sessione di lettura che riguarda quel file, viceversa apre una sessione di scrittura quando un utente richiede di editare un file. Non c’è un limite sul numero di utenti che possono esserci all’interno di una sessione di lettura: se un utente decide di leggere un file e poco dopo ci sarà un altro utente interessato a leggere lo stesso file, questo è possibile. Non è invece possibile la presenza di due utenti all’interno della sessione di lettura: se un utente decide di editare un file, non può esserci un altro utente che vuole leggere lo stesso file. Questo sarà possibile grazie ai metodi *syncronized* implementati nella classe Files.java. Inoltre, un utente non può decidere di editare un file, se un altro utente sta leggendo lo stesso file. Viceversa, se un utente sta leggendo un file, un altro utente non può editare lo stesso file.

Nessun file può essere rinominato o eliminato mentre qualcuno lo sta leggendo o editando; infatti, un utente che decidere di eliminare o rinominare un file mentre un altro utente sta leggendo o editando entrerà in uno stato di ‘wait’, fino a quando l’utente non uscirà dalla

modalità di scrittura o lettura. Ovviamente, non deve esserci più nessun utente in modalità lettura o scrittura per far sì che l'utente esca dallo stato di 'wait'.

(b) Descrizione dettagliata delle singole componenti

Client.java

Il lato client del software include le classi Client e ServerHandler.

Il client viene avviato da linea di comando e accetta come argomenti l'indirizzo IP e la porta del file server a cui connettersi.

Viene creato un nuovo socket solo quando il server ha accettato la connessione; in caso contrario, otterremo un'eccezione di connessione rifiutata ('Server non raggiungibile!').

Una volta creato correttamente, possiamo quindi ottenere flussi di input e output da esso per comunicare con il server: utilizziamo lo Scanner per ottenere l'input dall'utente e l'oggetto PrintWriter per inviare i dati al server.

Il flusso di input del client è connesso al flusso di output del server, proprio come il flusso di input del server è connesso al flusso di output del client.

Successivamente si delega la gestione della nuova connessione a un thread ServerHandler dedicato.

Per fare in modo che il metodo ***run()*** dell'istanza Runnable venga chiamato dal nuovo thread creato, *serverHandlerThread*, bisogna chiamare il metodo ***start()***.

```
Thread serverHandlerThread = new Thread(new ServerHandler(s));
serverHandlerThread.start();
```

Il ciclo do-while viene utilizzato per eseguire un blocco di istruzioni in modo continuo, fino a quando l'input da parte dell'utente non sarà uguale a ***'quit'***.

Infine, il metodo ***close()*** chiude il socket specificato e lo scanner.

Server.java

Il lato server del software include le classi Server e ClientHandler.

Il server viene avviato da linea di comando e accetta come argomenti il percorso della cartella della macchina host, in cui verranno custoditi i file, e la porta su cui il server resta in ascolto di richieste di connessione da parte dei client.
Se tutto procede a buon fine, il server accetta la connessione. Dopo l'accettazione, il server ottiene un nuovo socket, *clientSocket*, associato alla stessa porta locale.
A questo punto, il nuovo oggetto socket mette il server in connessione diretta con il client. Possiamo quindi accedere ai flussi di output e input per scrivere e ricevere messaggi rispettivamente da e verso il client.
Per ogni nuovo client, il server necessita di un nuovo socket, restituito dalla chiamata di accettazione.
Usiamo il *serverSocket* per continuare ad ascoltare le richieste di connessione, mentre ci prendiamo cura delle esigenze dei client connessi. Per consentire la continuità nella comunicazione e permettere al server non solo di rimanere in attesa di nuove connessioni, ma anche di poter inviare delle richieste abbiamo implementato un ciclo do-while.

```
} while (!request.equals(anObject: "quit"));
```

Fin quando il server non invia una richiesta di terminazione, attraverso il comando ***'quit'*** non usciremo dal ciclo, inoltre in base alle richieste inviate dal server verranno eseguite determinate azioni. Se il server lancia la richiesta ***'Ascolta'*** vuol dire che è in ascolto e in attesa di connessioni da parte di nuovi client, se lancia la richiesta ***'info'*** verranno mostrate a schermo delle informazioni, tra cui: il numero di file gestiti dal server, il numero di client attualmente connessi in lettura e infine il numero di client attualmente connessi in scrittura. Nel momento in cui la richiesta è ***'quit'***, usciremo dal ciclo, tutte le connessioni dei client verranno terminate e verrà terminata anche la connessione del server.

ClientHanlder.java

La classe ClientHandler permette al server di leggere le richieste effettuate dal client.

Il costruttore di questa classe accetta tre parametri, che possono identificare in modo univoco qualsiasi richiesta in arrivo, ovvero un Socket, il file Files, e la cartella in cui sono presenti i files.

```
public ClientHandler(Socket s, Files input, String p) throws IOException {
  this.s = s;
  this.input = input;
  this.path = p;
  this.toClient = new PrintWriter(s.getOutputStream(), autoFlush: true);
  this.fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
}
```

All'interno del metodo ***run()*** di questa classe si eseguono diverse operazioni: leggere in input la richiesta dell'utente grazie al metodo *readLine()* della classe BufferedReader, stampare la richiesta al server e di conseguenza scrivere l'output a lato Client attraverso il metodo *println()* della classe PrintWriter.

In seguito, sono presenti una serie di molteplici istruzioni condizionali if-else la cui esecuzione è condizionata dall'input digitato dall'utente:

- L'utente digiterà 'create file.txt' per creare il file, il quale, se non già presente all'interno della cartella, verrà creato attraverso il metodo *createNewFile()*.
- L'utente scriverà 'read file.txt' per entrare in modalità lettura, invocando il metodo ***startRead(),*** il quale incrementerà il numero di lettori presenti all'interno di quel file.

  Lo schema di loop illustrato qui sotto rappresenta un'iterazione sulle righe del file. Infatti, il metodo *readLine()* restituisce il valore null se non ci sono dati da leggere dal file (ad esempio, quando siamo alla fine del file). Tale condizione viene utilizzata per verificare la terminazione del ciclo.

```
String line = br.readLine();
while (line != null) {
  toClient.println(line);
  line = br.readLine();
}
```

  Il codice all'interno della clausola finally verrà eseguito nel momento in cui l'utente digiterà ':close' per chiudere la modalità scrittura, grazie al metodo ***endRead()***, il quale decrementerà il numero di lettori.

- Se l'utente vorrà rinominare un determinato file, dovrà digitare 'rename oldFile.txt newFile.txt'. Il metodo *split()* divide una stringa in più stringhe dato il delimitatore che le separa. L'oggetto restituito è un array che contiene le stringhe divise, in questo caso il nome del file da rinominare e il nuovo nome da assegnargli.

```
String[] split = nameFiles.split(regex: " ");
File oldFile = new File(path + split[0]);
File newFile = new File(path + split[1]);
```

Se il file potrà essere rinominato, verrà richiamato il metodo *renameTo()*, per poi essere stampato un messaggio di successo per informare l'utente.

- Il comando 'list' utilizza la classe SimpleDateFormat per la formattazione e il parsing delle date in una modalità locale-sensitive, cioè che si adegua al formato temporale in uso nell'area geografica del sistema.

```
SimpleDateFormat sdf = new SimpleDateFormat(pattern: "dd/MM/yyyy HH:mm:ss");
```

Il metodo *lastModifiedTime()* della classe File è utilizzato per la visualizzazione della data dell'ultima modifica di un file.

Per quanto riguarda il numero di lettori e di scrittori presenti in ogni file, abbiamo richiamato i metodi ***getReadMap()*** e ***getWriteMap()***.

- Se l'utente digiterà 'edit file.txt' entrerà in modalità scrittura grazie alla chiamata del metodo ***startEdit().***

Il ciclo while rimarrà in funzione fin quando la variabile requestType non risulterà negativa.

Il comando ***':backspace'*** permette all'utente di rimuovere l'ultima riga del file.

Qui sotto mostriamo come avviene la memorizzazione di ogni singola riga del file all'interno del vettore v.

```
Vector<String> v = new Vector<String>(); // contiene tutte le righe del file
String linea = ""; // Leggo tutto il file e lo memorizzo nel Vector
while ((linea = br.readLine()) != null) {
  v.add(linea);
}
```

Successivamente, se nel file sarà presente almeno una riga di testo, riscriverò il file tralasciando l'ultima riga.

```
if (v.size() == 0) { // Ora riscrivo tutto, tranne l'ultima riga
  toClient.println(x: "Il file è vuoto!");
} else {
  PrintStream ps = new PrintStream(new FileOutputStream(path + requestArg));
  for (int i = 0; i < v.size() - 1; i++) { // Il -1 indica di tralasciare l'ultima
    ps.println((String) v.elementAt(i));
  }
```

Il comando **':close'** chiuderà la sessione di scrittura.

Ogni input diverso da ***':backspace'*** e ***':close'*** verrà considerato come una riga di testo da aggiungere al file, grazie al metodo *write()* della classe BufferedWriter.

ServerHandler.java

La classe ServerHandler permette al client di leggere le richieste effettuate dal server.

Il costruttore di questa classe accetta un unico parametro, rappresentato dalla Socket.

L'unica richiesta che il server invia al client è la richiesta **'quit':** nel momento in cui il server invia questo comando disconnette eventuali client ancora connessi e chiude il server. Le altre richieste del server non sono state implementante in questa classe, dal momento che non è necessario che il client le riceva.

```
public ServerHandler(Socket s) throws IOException {
  this.s = s;
  this.toServer = new PrintWriter(s.getOutputStream(), autoFlush: true);
}
```

Files.java

La classe Files funge da classe ausiliaria all'applicazione software. Vi sono implementati tutti i metodi che gestiscono la concorrenza tra thread nel momento in cui più client tentano di eseguire azioni sui file contenuti nel server.

La struttura dati utilizzata per memorizzare tutte le variabili necessarie alla gestione della concorrenza è la *HashTable*. Abbiamo scelto questa struttura in quanto i suoi metodi sono già essi stessi sincronizzati. Lo stesso ragionamento è stato usato nella scelta di *Vector* per mantenere la lista dei file contenuti nel Server.

Per il conteggio del numero di lettori, scrittori e client in attesa sono state create 3 table con lo stesso funzionamento, rispettivamente *readTable, writeTable, waitingTable*. Ogni table contiene una serie di coppie chiave-valore formate da "nome del file" - "numero di

lettori/scrittori/in attesa", per tenere traccia del numero di client attualmente su un dato file.

```
private Hashtable<String, Integer> readTable = new Hashtable<>(); // hashtable file - num lettori
private Hashtable<String, Integer> writeTable = new Hashtable<>(); // hashtable file - num scrittori
private Hashtable<String, Integer> waitingTable = new Hashtable<>(); // hashtable file - num client in attesa
```

Allo stesso modo funzionano le table *dbReadingTable, dbWritingTable, dbWaitingTable.* In queste ultime sono contenute coppie "nome del file" - "booleano true o false". Questo booleano funge da variabile generale per controllare se su un dato file ci sono lettori, scrittori, client in attesa, sulla base dei contatori contenuti nelle table precedenti.

```
private Hashtable<String, Boolean> dbReadingTable = new Hashtable<>(); // hashtable file - booleano di lettura
private Hashtable<String, Boolean> dbWritingTable = new Hashtable<>(); // hashtable file - booleano di scrittura
private Hashtable<String, Boolean> dbWaitingTable = new Hashtable<>(); // hashtable file - booleano di client in attesa
```

In più ci sono variabili generiche che tengono traccia del numero di lettori, scrittori, client in attesa su tutti i file del Server, per implementare il comando "info" del server.

Sono quindi implementati una serie di metodi applicabili su ogni file per gestire la concorrenza. I metodi *'startRead()', 'endRead()', 'startEdit()', 'endEdit()', 'startDeleteRename()', 'endDeleteRename()'* sono tutti metodi synchronized.

Funzionamento di *'startEdit()'* a titolo di esempio anche per tutti gli altri:

```
public synchronized int startRead(String fileName)
  throws IOException, InterruptedException {
  while (dbWritingTable.get(fileName) == true) {
    wait(); // dbReading non gestito perchè più lettori sono ammessi
  }
  ++readerCount;

  int count = readTable.get(fileName);
  count++;
  readTable.replace(fileName, count); // aggiorno la table dei lettori

  if (count == 1) {
    dbReadingTable.replace(fileName, value: true); // aggiorno la table
  }

  System.out.println(
    "Numero di lettori nel file '" + fileName + "': " + readTable.get(fileName)
  );
  return readerCount;
}
```

Viene controllato che i valori di *dbReadingTable e dbWritingTable* siano false, cioè non ci siano lettori o scrittori attualmente sul file in questione, altrimenti il client che vuole editare il file viene messo in attesa. In caso contrario viene incrementato il valore che conta gli scrittori nel file nella *writeTable* e settato a true il valore della *dbWritingTable.*

Sono presenti, infine, i metodi ***'inizialize()', 'insert()', 'update()', 'delete()'*** che sono metodi ausiliari per la gestione dei file nelle table. Al momento della creazione di un nuovo file, l'eliminazione, il rename e al momento dell'avvio del ClientHandler questi metodi aggiornano le table con i file ed i rispettivi valori corretti.

(c) Suddivisione del lavoro tra i membri del gruppo

1. **Modello Client-Server:** Carolina, Matilde
2. **Comandi Client:** tutti i membri del gruppo
3. **Comandi Server:** tutti i membri del gruppo
4. **Metodi Syncronized:** Carolina, Matilde, Marianna
5. **Documentazione:** tutti i membri del gruppo

2. Descrizione e discussione del processo di implementazione:

(a) Descrizione dei problemi e degli ostacoli incontrati

- Problemi legati alla concorrenza:

Il principale problema che abbiamo riscontrato nella strutturazione dei metodi sincronizzati è stato quello del metodo *startDeleteRename().* La problematica è sorta nel momento in cui due client cercavano di rinominare o eliminare lo stesso file contemporaneamente.

Generica situazione in cui abbiamo capito che la concorrenza non eseguiva correttamente: il client c1 legge/scrive su un file, il client c2 decide di eliminare quel file, il client c3 decide di rinominare quel file. In questa situazione il client c2, una volta che il client c1 chiude la sua modalità di lettura/scrittura, riesce a portare a termine la sua operazione, quindi il file viene eliminato. Succede però che il client c3, che era in uno stato di wait, dopo che il client c2 esce dal wait ed elimina il file, continua a rimanere in attesa senza poter compiere più nessuna azione.

Inizialmente abbiamo tentato di creare un metodo che gestisse la concorrenza, senza però mantenere una lista dei client che cercavano di rinominare o cancellare il file. In questo modo, a seconda del client che veniva sbloccato per primo, potevano sorgere errori (es. rinominare un file non esistente, cancellare due volte lo stesso file, ecc).

Abbiamo infine fatto in modo che un client, per poter eliminare o rinominare un file, debba comportarsi allo stesso modo di un utente che vuole modificare un file. Esso si comporta come un vero e proprio scrittore. In più rispetto al metodo *startEdit()* viene mantenuta una table dei client in attesa, così da avere una vera e propria coda per gestire la concorrenza e non avere più i problemi elencati precedentemente.

- Problemi legati al modello Client-Server:

  Uno dei problemi principali si è presentato a livello del Server.

  Il Server deve sia ricevere le richieste da ogni singolo client, sia inviare comandi ai client.

  Inizialmente avevamo pensato di implementare la connessione Client-Server come una chat, in cui ogni comando inviato dal client veniva ricevuto dal server e viceversa. Questa soluzione non era ottimale poichè non tutti i comandi inviati dal server dovevano poi arrivare al client; dunque, abbiamo creato una classe ServerHanlder, il cui funzionamento è uguale alla classe ClientHandler, gestendo qui l'unica richiesta che doveva essere inviata dal server ed arrivare a tutti i client: la richiesta *'quit'*.

  All'interno della classe server il tutto è stato gestito da un ciclo do-while, che permette al

  server di restare in ascolto ed eventualmente inviare comandi.

(b) Descrizione degli strumenti utilizzati per l'organizzazione. In particolare, applicazioni, piattaforme, servizi utilizzati per:

- Per sviluppare il progetto è stato utilizzato Visual Studio Code. Di comune accordo è stato deciso che fosse l'IDE più adatto allo sviluppo del progetto.
- Per comunicare tra i membri del gruppo è stata utilizzata una chat WhatsApp, nello specifico per accordarsi sulle riunioni e per comunicazioni quotidiane varie. Per riunioni e chiamate invece è stato utilizzato Teams, di modo da poter condividere lo schermo del proprio computer e mostrare/analizzare il codice prodotto con tutti i membri del gruppo.
- Per la condivisione del codice prodotto è stato utilizzato GitLab. È sempre stato caricato tutto sul main branch del progetto in quanto la revisione del codice prodotto, prima di essere caricato, avveniva sempre tramite una videochiamata su Teams alla quale partecipavano tutti i membri del gruppo.
- Per tenere traccia del lavoro svolto, del lavoro rimasto da svolgere, delle decisioni ad alto livello prese dal gruppo, etc. è stato utilizzato Google Docs, nello specifico un foglio di Word

nel quale venivano annotate funzionalità sviluppate, da sviluppare e problematiche da risolvere.

3. Requisiti e istruzioni passo-passo per compilare e usare le applicazioni consegnate

Compilazione -> javac Server.java Client.java

Esecuzione Server -> java Server <path/><port> (es. java Server Files/ 25565)

Esecuzione Client -> java Server <host><port> (es. java Client localhost 25565)

COMANDI DEL CLIENT

Comando 'create' -> create file.txt

Comando 'read' -> read file.txt

- **Comando 'close' ->** :close

Comando 'list' -> list

Comando 'edit' -> edit file.txt

- **Comando 'close' ->** :close
- **Comando 'backspace' ->** :backspace
- **Comando di scrittura ->** … (es. riga di testo)

Comando 'rename' -> rename oldFile.txt newFile.txt

Comando ‘delete’ -> delete file.txt

Comando ‘quit’ -> quit

COMANDI DEL SERVER

Comando ‘Ascolta’ -> Ascolta

Comando ‘info’ -> info

Comando ‘quit’ -> quit