LABORATORIO DI SISTEMI OPERATIVI A.A 2021-2022

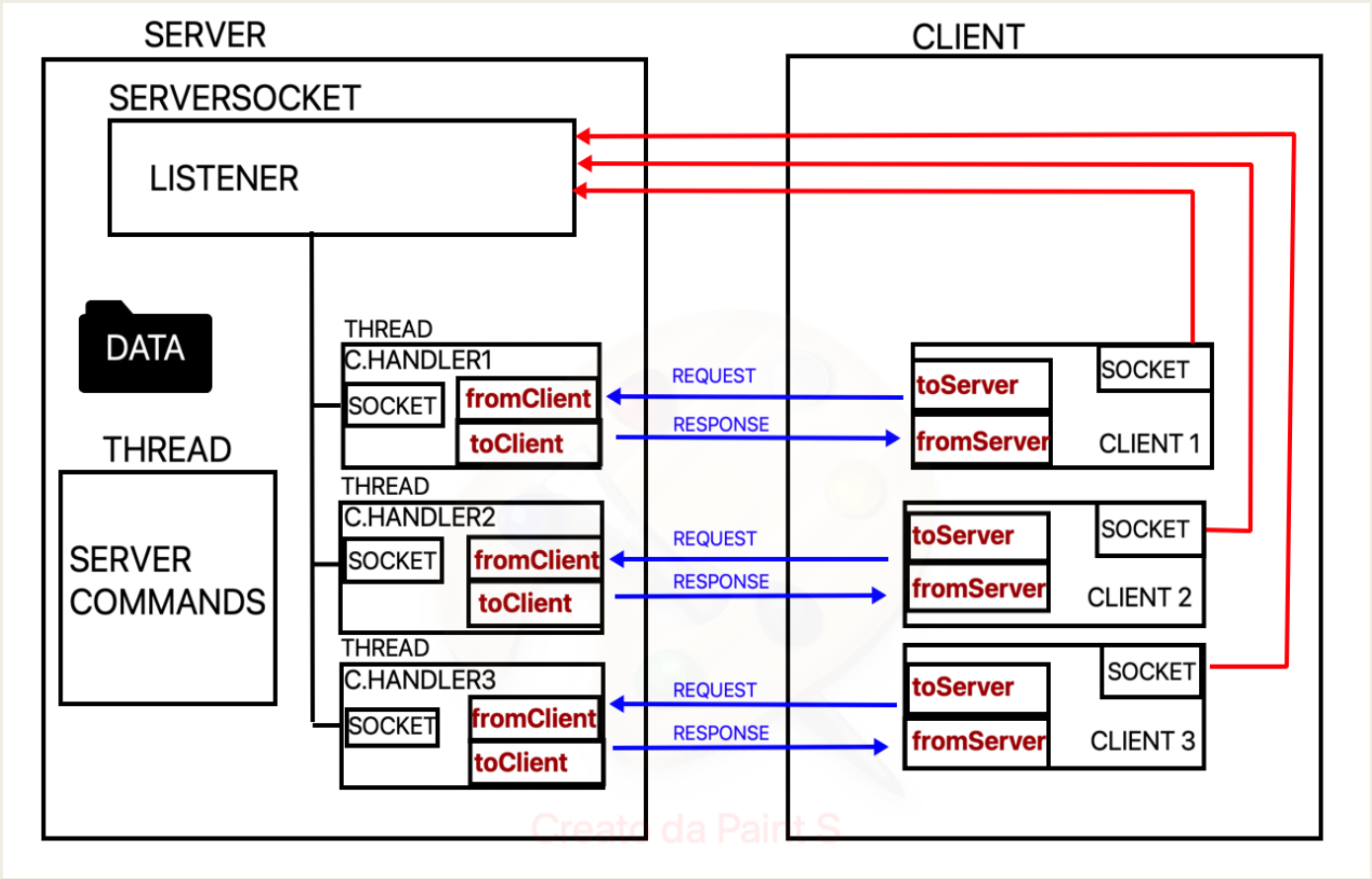
DAWID NIKODEM PIESLA 970296
(dawidnikodem.piesla@studio.unibo.it)
EUGENIO MARIA DE ROSA 978596
MARCO TENACE 988908
ALESSANDRO LAERA 901889

GRUPPO:

I BILLONI

IL PROGETTO

architettura generale



Spiegazione dello schema:

Il server una volta avviato caricherà con il metodo loadFiles() i file di testo presenti nella cartella “data” e avvierà il Thread “serverCommands” che permette di ricevere informazioni sul Server e di chiuderlo.

Successivamente si creerà una variabile ServerSocket chiamata “listener” che si occuperà di accettare le connessioni dei Client, le quali verranno delegate ad un ClientHandler. Il ClientHandler creerà un Thread del Client apposito e i due comunicheranno dall’inputStream e outputStream del Socket. Le richieste del Client verranno inviate tramite il PrintWriter “toServer” allo Scanner “fromClient” che elaborerà le richieste e invierà un messaggio di risposta sullo stato dell’operazione tramite il PrintWriter “toClient” allo Scanner "fromServer" che stamperà sul terminale le risposte del ClientHandler.

DESCRIZIONE DELLE CLASSI

Le classi che abbiamo utilizzato per la realizzazione del progetto sono sei e sono: Client, Server, ClientHandler, ServerCommands, Filebox e ReadersWritersHandler.

La classe Client è la classe che ci permette di aprire la connessione con il Server e ci permette di prendere a carico diversi comandi che l'utente può inserire da tastiera.

I diversi comandi che la classe Client riesce a gestire sono:

- Il comando «list» che ci restituisce un array di file che erano già presenti nella cartella presa in considerazione o file che sono stati creati dall'utente stesso attraverso il comando «create».
- Il comando «create» che serve per creare un file e inserirlo nella cartella. Questo comando deve essere seguito dal nome del file che si vuole assegnare.
- Il comando «read» ci permette, come suggerisce il nome stesso, di leggere le varie righe del file e questo comando deve essere seguito dal nome del file che si vuole leggere.
- Il comando «edit» invece ci permette di scegliere tra tre opzioni disponibili; «write» , «backspace» e «close» che ci permettono rispettivamente di scrivere nel file, eliminare l'ultima riga del file e il comando close viene utilizzato per chiudere la fase di modifica del file.
- Il comando «rename» ci permette di rinominare un file e deve essere seguito dal nome del file che l'utente vuole modificare e dal nuovo nome che il file dovrà assumere e che l'utente vuole assegnare.
- Il comando «delete», invece, deve essere seguito anch'esso dal nome di un file esistente e serve per eliminare il file selezionato dalla cartella.
- Il comando « quit» , infine, serve per chiudere la connessione e di conseguenza arrestare il Client

Le classi Client e ClientHandler comunicano tra di loro attraverso l'invio e la ricezione di stringhe utili per la realizzazione dei metodi della classe ClientHandler e dei comandi richiesti nella classe Client.

Nella classe ClientHandler vengono gestiti i comandi della classe Client attraverso dei metodi e uno scambio continuo di stringhe, utilizzati come messaggi di start, do e stop.

Tra i principali metodi implementati nella classe ClientHandler troviamo:

fileExist() che ci indica se un file esiste ed è presente nella cartella presa in considerazione e getFilePos() che ci indica la posizione di un determinato file all'interno di un array di file.

La classe FileBox invece è una classe di supporto al ClientHandler in quanto vengono implementati i metodi utili per la realizzazione dei comandi della classe client che a sua volta vengono presi in carico dalla classe ClientHnadler.

Molteplici sono i metodi situati nella classe FileBox, tra i principali troviamo:

getLastModified() ci permette di visualizzare quando è avvenuta l'ultima modifica del file, appendLine() aggiunge una riga alla fine di un file, removeLastLine() invece rimuove l'ultima riga ad un file e getRemovedLine() ci indica qual è l'ultima riga che è stata rimossa da un documento.

La classe Server ci permette di accettare e far collegare simultaneamente diversi Client che ci chiedono di connetterci ad esso. Anche nella classe Server sono implementati alcuni metodi e riesce a rispondere a diversi comandi che l'utente inserisce da tastiera.

I comandi che la classe Server riesce a gestire sono:

- Il comando «info» restituisce il numero di file gestiti dal server, il numero di client connessi in lettura e in scrittura.
- Il comando quit chiude il server e disconnette eventuali client ancora connessi al server.

La classe ServerCommands è una classe di supporto al Server in quanto avviene uno scambio di messaggi tra la classe Server e la classe ServerCommands utili per la realizzazione dei comandi richiesti dall'utente.

La classe ReadersWritersHandler gestisce la concorrenza per l'accesso alle risorse sul server. Il flusso di richieste per ogni file viene regolamentato da semafori che fanno accedere sequenzialmente gli utenti in base all' attività che desiderano svolgere.

DESCRIZIONE DELLE CLASSI NEL DETTAGLIO

Di seguito illustreremo nel dettaglio le principali classi per il corretto funzionamento del progetto.

CLASSE CLIENT → Inizialmente facciamo un controllo per verificare che gli argomenti che vengono passati in input da riga di comando dall'utente siano minori di tre in quanto «IP» e «port» sono gli argomenti che ci aspettiamo. Memorizziamo «IP» e «port» dentro due variabili e le passiamo come argomento dentro il costruttore della classe Socket. Classe Socket che ci sarà utile per l'invio e la ricezione di dati tra host.

A questo punto inizializziamo uno Scanner per ricevere informazioni dal Server (nel caso specifico dal ClientHandler), un PrintWriter per inviare informazioni al Server(nel caso specifico al ClientHandler) e un altro Scanner per leggere quello che l'utente inserisce da terminale.

Entriamo così nel ciclo di vita del Client ed entriamo subito in un ciclo while che ci permette di comunicare con il ClientHandler e permette ad ogni Client che vuole collegarsi al Server di inserire un proprio username. Quindi dopo aver fatto un veloce controllo per verificare che il Client avesse o meno già un username si procede ad inserirlo, altrimenti si passa nella sezione seguente dove viene chiesto all'utente di inserire la propria richiesta che poi verrà inoltrata al Server.

Successivamente la richiesta viene splittata in più parti e vengono memorizzati dentro più variabili il tipo di richiesta, ovvero il comando che l'utente desidera eseguire e i vari argomenti indispensabili per l'esecuzione del programma.

Grazie alla suddivisione in più parti della richiesta, siamo in grado di selezionare il comando richiesto e dunque inoltrare la richiesta nella sezione corretta.

A questo punto avremo una serie di controlli if che ci permettono di indirizzare la richiesta dentro la sezione corretta. Per fare un esempio se l'utente inserisce il comando «rename» seguito dal nome del file che vuole modificare e il nuovo nome che vuole attribuire a quel determinato file, allora verranno fatti dei controlli sulla stringa «rename» attraverso molteplici «if» e quando viene trovato l'»if» corretto, ovvero «if (requestType.equals(«rename»))» allora entreremo in quella sezione e gestiremo il comando «rename».

Prima di poter arrestare il client, chiudiamo i vari Scanner aperti e la connessione attraverso al metodo close().

CLASSE CLIENTHANDLER → La prima parte della classe ClientHandler si può paragonare alla prima parte della classe Client in quanto subito dopo aver inizializzato uno scanner che ci permette di leggere dati dal Client e un PrintWriter che ci permette di inviare dati al Client, entriamo nel ciclo di vita del ClientHandler, entrando subito dentro un ciclo che ci permette di inserire il nome del Client, come precedentemente spiegato, e non appena aver passato questa parte leggiamo e splittiamo la richiesta in più parti, anche questo passaggio è stato precedentemente spiegato nella classe Client.

A questo punto, come avveniva anche nella classe Client, vi è un susseguirsi di controlli attraverso il comando «if» che ci permette di indirizzare la richiesta, ricevuta dal Client, nella sezione corretta.

In questa parte avviene un vero e proprio «dialogo» tra le classi Client e ClientHandler in quanto per la realizzazione del comando richiesto vengono scambiate delle stringhe tra le due classi in modo tale da controllare e organizzare il passo seguente da sviluppare.

Facciamo qualche esempio pratico per capire meglio.

- Se la richiesta che riceve il Client è uguale a «rename oldDoc newDoc», il Client inoltra la richiesta al ClientHandler che la prende a carico entrando nella sezione apposita per questo comando. Il ClientHandler verifica che il file da rinominare esista e che il nuovo nome che deve assumere il file non sia un valore null. Il ClientHandler se non trova problemi procede con il metodo per rinominare il file e una volta terminato manda una stringa al Client comunicando «ENDRENAME», ovvero comunicando che ha terminato la modifica del nome ed è andato tutto per il verso giusto. Il Client che era in attesa di quella stringa, riceve la stringa «ENDRENAME» ed esce dal ciclo relativo al comando «rename».

- Se la richiesta che riceve il Client è uguale a «list», il Client inoltra la richiesta al ClientHandler che la prende a carico entrando nella sezione apposita per questo comando. Il comando viene gestito da un ciclo for che cicla sulla grandezza dell'ArrayList di file presenti in cartella, andando ad incrementare un contatore ad ogni iterazione e stampando per ogni posizione(da 0 a files.length() - 1) la posizione di ogni file, il nome, e la data dell'ultima modifica che ogni file ha subìto. Non appena ha terminato il ciclo e quindi tutte le informazione su ogni file sono state stampate il ClientHandler manda una stringa al Client con su scritto «ENDLIST». Il client che era in attesa di questa stringa, riceve la stringa ed esce dal ciclo relativo al comando «list».

Prima di poter arrestare il client, chiudiamo i vari Scanner aperti e la connessione attraverso al metodo close().

CLASSE FILEBOX → La classe FileBox si può considerare una classe di supporto dei file di testo, in quanto vengono implementati metodi utili per la realizzazione dei comandi richiesti dall'utente.

Andremo a vedere nel dettaglio alcuni dei metodi principali che si trovano in questa classe. Tra i principali metodi troviamo, getLastModified(), appendLine(), removeLastLine() e getRemovedLine().

- Il metodo getLastModified() è il metodo che ci permette di poter eseguire correttamente il comando «list» in quanto andiamo a completare la stampa delle stringhe con i vari nomi presenti in cartella con la data dell'ultima modifica e altri dati del file stesso.

Per la corretta realizzazione di questo metodo inizialmente dichiaro la variabile «fileName1» che prende il path dove sono stati salvati i file e ci aggiunge il nome del file corrente seguito da «.txt» e memorizza questa stringa dentro una variabile di tipo Path. Dichiariamo adesso una variabile di tipo BasicFileAttributes di nome «attr» e chiamiamo il metodo readAttributes() passando come argomento il path che ho creato precedentemente e il parametro «BasicFileAttributes.class». A questo punto su questa variabile che ci siamo creati posso chiamare i metodi creationTime(), lastAccessTime() e lastModifiedTime() che servono rispettivamente per capire quando un file è stato creato, l'ultimo accesso al file e l'ultima modifica.

Adesso modifichiamo il formato di data e ora infine stampiano il risultato finale dei vari metodi precedentemente chiamati e modificati.

- Il metodo appendLine() è il metodo che ci permette di scrivere delle stringhe in coda ad un file. Questo comando viene utilizzato per la realizzazione del comando «edit».

- Il metodo removeLastLine() è il metodo che ci permette di eliminare l'ultima riga di un file, se il file non è vuoto. Questo metodo viene utilizzato per la realizzazione del comando «edit» e in particolare del comando «:backspace» dove viene, difatti, chiesto di eliminare l 'ultima riga di un file.

In questo metodo inizialmente creiamo un RandomAccessFile(), settiamo una length corrispondente alla lunghezza del file – 1. Diminuisco la lunghezza di un'unità ad ogni iterazione e con il metodo seek() individuiamo fino a dove arriva all'interno del file quella determinata lunghezza e leggiamo i byte. Continuo ad iterare all'interno del ciclo fin quando non trovo una nuova riga.

- Il metodo getRemovedLine() è il metodo che ci ritorna l'ultima riga che è stata rimossa da un file.

CLASSE SERVER → La classe Server è una classe fondamentale in quanto ci permette di accettare delle nuove connessioni attraverso l'utilizzo del socket. Inizialmente facciamo un controllo per verificare che gli argomenti che vengono passati in input da riga di comando dall'utente siano minori di due in quanto «port» è l'argomento che ci aspettiamo.

Memorizziamo «port» dentro una variabile e la passiamo come argomento dentro il costruttore della classe ServerSocket. Classe ServerSocket che ci sarà utile per creare un socket dal Server sulla porta specificata.

Adesso creiamo un nuovo thread di «ServerCommands» avviando così il metodo «run()» della classe che rimane in attesa di un comando che deve inserire il proprietario del Server.

Entriamo nel ciclo di vita del Server e attraverso il metodo «accept()» accettiamo le richieste in arrivo.

Creiamo un thread di «ClientHandler» che, come detto precedentemente, avvia il metodo «run()» della classe ClientHandler. E ripeto il ciclo.

In questa classe vengono implementati anche alcuni metodi come «loadFiles()», «showInfo()», «countBw()», «countBr()», «countAw()», «countAr()», e «closeServer()».

CLASSE SERVERCOMMANDS → Inizialmente dichiariamo uno scanner che ci permette di leggere da console.

Se la richiesta è uguale a «info» , grazie al metodo «showInfo()» della classe Server, stampiamo informazioni relative al numero di file gestiti dal Server, numero di Client connessi in lettura e scrittura.

Se la richiesta è uguale a «quit» allora vengono disconnessi eventuali Client ancora connessi e chiude il Server.

CLASSE READERSWRITERSHANDLER →all'interno di questa classe avviene la gestione della concorrenza tramite l'utilizzo di appositi semafori. All'inizio della classe abbiamo il costruttore di Semaphore con cui possiamo interagire con i metodi p() e v().

Vengono poi dichiarati 4 semafori, due per i lettori/scrittori, uno per rename e l'ultimo per delete. Attraverso i metodi begin/end Write, begin/end Read, delete e rename è possibile gestire la concorrenza tra gli utenti per l'accesso alle risorse sul server.

Per iniziare una sessione di lettura è indispensabile che sul file non siano presenti scrittori attivi o bloccati, altrimenti l'utente viene messo temporaneamente nella coda dei lettori bloccati

Per utilizzare la sessione di scrittura non ci possono essere scrittori e lettori attivi, sennò l'utente viene messo temporaneamente nella coda degli scrittori bloccati

Per richiamare beginRename o beginDelete sulla risorsa non ci possono essere presenti richieste di alcun tipo, in caso contrario l'utente sarà pregato di riprovare a eliminare/rinominare la risorsa più tardi, quando nessuno la starà più utilizzando

La politica adottata per gestire le file degli lettori/scrittori bloccati viene spiegata nel dettaglio più avanti

GESTIONE DELLA CONCORRENZA

Per gestire la concorrenza tra i vari client abbiamo deciso di implementare i semafori:
In ogni istanza di FileBox e' presente la variabile handler di tipo ReadersWritersHandler che ha il ruolo di lock, quest ultimo viene associato ad ogni file presente sul server.
All'interno della classe ReadersWritersHandler possiamo trovare una serie di semafori dedicatati a gestire le operazioni di scrittura, lettura, ecc...Per risolvere il problema degli accessi simultanei dei lettori e scrittori allo stesso file sono stati usati quatto metodi, rispettivamente: beginWrite, endWrite, beginRead, endRead. Queste funzioni attraverso l'uso dei metodi p() e v() modificano lo stato del rispettivo semaforo (rs per i lettori e ws per gli scrittori) gestendo così il flusso di accessi al file.
La logica che abbiamo adottato è molto simile a quella vista a lezione durante il corso di Sistemi Operativi: più utenti possono accedere allo stesso file in modalità lettura, ma per ogni file può essere presente solo uno scrittore. I controlli presenti all' interno dei metodi citati sopra servono proprio a questo, verificare il numero di scrittori/lettori su ogni file e stabilire se consentirne l'accesso oppure mettergli temporaneamente in coda con gli altri utenti che attendono il proprio turno prima di poter accedere alla risorsa.
Sulle code degli scrittori/lettori bloccati viene adottata una gestione fair, che evita la starvation degli scrittori. I lettori possono accedere al file finche' uno scrittore non richiede l'accesso; a questo punto i lettori attivi possono terminare la propria sessione, mentre lo scrittore attende l'uscita di tutti i lettori attivi per iniziare la propria sessione di scrittura. Una volta terminata la fase scrittura, se presenti i lettori nella coda BR(blocked readers queue) ottengo diretto accesso al file in ordine rispetto al momento della richiesta. A questo punto si attende una nuova richiesta di scrittura per impedire l'acceso al file a nuovi lettori e il ciclo si ripete.

Successivamente era sorto il problema di come gestire la concorrenza sui comandi rename e delete. L'idea di fondo è di sfruttare le stesse variabili per non appesantire il codice; come fare a sfruttare le stesse variabili su questi comandi inseriti dall'utente?

La soluzione è stata vedere un client che rinomina o cancella un file come un client che modifica un file, quindi abbiamo sfruttato la variabile booleana aw che attiva quando un client modifica un file.

Inoltre abbiamo sfruttato dei semafori, perché appunto i client posso accedervi uno alla volta per eseguire questi comandi, e non in contemporanea come succede per il comando rename.

Quindi se più client effettuerebbero la chiamata di rename o delete nello stesso momento tutti tranne 1 sarebbero subito bloccati dato che il semaforo all'inizio viene settato a 1 e appena scatta il beginRename o il beginDelete il semaforo chiama subito p() che la setta subito a 0 facendo si che nessun altro ci potesse entrare. Successivamente viene riposto a 1 soltanto quando viene eseguita l'operazione endRename o endDelete dove il semaforo chiama v() , cioè viene aumentato il semaforo di 1.

DIVISIONE MANSIONI DEL GRUPPO:

Per dividerci il lavoro abbiamo deciso di dividerci in 2 parti, una per la gestione e implementazione della concorrenza (Dawid e Alessandro) e l'altra per la costruzione della struttura di Filebox, Client, Server e ClientHandler (Eugenio e Marco)

- Eugenio: l'implementazione dell'username, getLastModified(), appendLine(), removeLine(), getRemovedLine(), loadFiles(), closeServer(), broadcastQuit(), showInfo(), controlli di persistenza dei dati, FileBox e gestione comandi lato Client-ClientHandler.
- Marco: implementazione metodo getLastModified(), metodo fileExist(), metodo showInfo(), closeServer() + struttura classe Client, classe Server, classe Filebox, classe ClientHandler
- Dawid: gestione concorrenza e semafori, ReaderWritersHandler. Read, Write e Rename, showInfo(), list
- Alessandro: implementazione metodo beginRename(), beginDelete(), gestione concorrenza di tali metodi su ClientHandler, creazione semaforo su ReadersWritersHandler

Descrizione e discussione del processo di implementazione:

PROBLEMI SERVER-CLIENT:

Uno dei problemi più ostici da risolvere del progetto è stato il comando "quit" dal lato Server. Quando viene eseguita la chiusura del Server e, quindi, la disconnessione dei vari Client viene chiuso l'input del Socket e viene chiuso il listener che si occupava di accettare le connessioni.

Il problema vero e proprio sta nei metodi chiamati successivamente per chiudere il Client, il quale sarà bloccato nella lettura da terminale in quanto nextLine() è un metodo bloccante. Per ovviare al problema abbiamo creato un metodo chiamato broadcastQuit() che invierà una stringa di chiusura al Client, che una volta uscito dal metodo bloccante verrà chiuso con successo catturando una NoSuchElementException, poiché cercherà di leggere una stringa di risposta dal Server che non esiste visto che il Server è stato chiuso. Viene inoltre stampato a schermo un messaggio di errore indicante il tipo di eccezione e consiglia di controllare che il Server sia aperto.

Questa eccezione viene lanciata anche nella classe ClientHandler quando un Client disconnette improvvisamente il suo input chiudendo il terminale, comportando la lettura di una richiesta dal Client che ha valore che non esiste. Verrà quindi stampato a schermo sul Server quale Client ha disconnesso il suo input.

Infine l'ultimo problema che abbiamo incontrato è stato come mettere a conoscenza il Client della terminazione delle operazioni del ClientHandler per poter ricominciare il ciclo while dell'invio delle richieste e rendere pronto il Client per l'input di nuovi comandi. Per risolvere questo problema abbiamo utilizzato un sistema di invio di stringhe di chiusura della specifica sessione di modifica/lettura dei dati. Ad esempio dopo aver richiesto la modalità di modifica sarà il ClientHandler a notificare il Client con il messaggio "ENDEDIT" per uscire dal ciclo di modifica e rendersi disponibile per una nuova richiesta. Questo approccio viene usato in maniera simile anche nell'impostazione del nome del Client, nella lettura della lista di file presenti, nella lettura di un file e nella rinomina dei file.

PROBLEMA DELL'IMPLAMENTAZIONE DELLA CONCORRENZA SU OGNI FILE

Per gestire la concorrenza degli accessi alle risorse presenti sul server è stato necessario ideare un modo per associare ad ogni file disponibile, un lock che tenesse conto del numero e del tipo di operazioni che vengono svolte sul file stesso.

La soluzione è stata aggiungere alla classe FileBox un campo di tipo ReadersWritersHandler, in questo modo ogni volta che viene creato un file attraverso uno dei costruttori di FileBox, al suo interno avrà il proprio Handler che sarà utilizzato per effettuare controlli sul tipo di operazioni che vengono svolte e per impedire o consentire l'accesso a nuovi utenti.

E' fondamentale che durante il ciclo di vita del server, i file con i relativi Handler annessi, vengano creati una volta sola e aggiornati tempestivamente tutte le volte che arriva una nuova richiesta da parte dei client. In questo modo è possibile garantire l'accesso sequenziale alle risorse con una corretta gestione della concorrenza.

PROBLEMA DEL METODO RENAME

L'implementazione di questo metodo è stata particolarmente difficoltosa, perché a differenza dei metodi read e write, rename esegue un controllo sul file d'interesse per verificare se sono presenti dei lettori/scrittori attivi o in coda.

In caso di esisto positivo, l'operazione di rename viene annullata e viene stampato un messaggio di errore. Il client potrà continuare ad usare le funzionalità del server senza essere bloccato in coda e successivamente potrà decidere di riprovare a rinominare il file.

In caso contrario l'utente che intende rinominare il file riceverà accesso alla risorsa e verrà visto come uno scrittore, che blocca temporaneamente il file agli altri lettori e scrittori durante l'esecuzione del metodo rename stesso.

Durante la stesura del codice ci siamo accorti dell'importanza di chiudere tutti gli scanner usati sui file, infatti, uno scanner aperto impedisce l'accesso alla risorsa ad altri processi lanciando l'eccezione java.nio.file.FileSystemException.

ORGANIZZAZIONE DEL GRUPPO:

Per organizzare il lavoro da svolgere e assegnarci i relativi compiti abbiamo usufruito di diverse piattaforme tra cui principalmente Telegram e Discord per collegarci in chiamata e svolgere il lavoro congiuntamente. Purtroppo non siamo riusciti mai ad essere tutti presenti in chiamata o di presenza, difatti abbiamo avuto qualche difficoltà organizzativa che ha rallentato i nostri tempi di elaborazione e sviluppo del progetto ma che, tuttavia, abbiamo risolto man mano.

Ognuno di noi, dopo aver scritto del codice, mandava il progetto aggiornato su Telegram dove tutti i componenti del gruppo potevano visualizzare e commentare il lavoro svolto.

Non abbiamo utilizzato molto la piattaforma GitLab, come invece era stato specificatamente richiesto, non per una nostra mancanza ma semplicemente perché abbiamo avuto delle difficoltà nel caricare o editare il codice già riportato sulla piattaforma dal referente del gruppo, quindi per motivi di comodità abbiamo preferito utilizzare principalmente altre piattaforme, senza mai trascurare GitLab.

Prima di iniziare a sviluppare il progetto abbiamo deciso di dividerci le mansioni anche in base agli impegni lavorativi, di studio o personali che ognuno di noi aveva ed ha avuto durante la realizzazione del progetto.

Istruzione passo-passo per avviare l'applicazione

Per eseguire l'applicazione basterà aprire una nuova finestra del terminale, raggiungere la cartella contenente le classi, chiamata "CodiceV2", e da qui eseguire la compilazione. Per compilare usiamo il comando ***javac *.java*** che compilerà tutte le classi.

```
CodiceV2 % javac *.java
```

Una volta compilate le classi verranno generati i file con estensione ".class" che ci permetteranno di effettuare con successo il comando di esecuzione della classe Server.

Per eseguire il Server torniamo alla directory precedente contenente la cartella "CodiceV2" e il comando ***java CodiceV2...Server*** seguito ***dall'indirizzo di porta***.

```
% cd ..
```

```
% java CodiceV2.Server 25565
```

Il Server sarà ora in esecuzione e nuovi Client potranno collegarsi. Apriamo dunque tante nuove finestre del terminale quanti i Client che vogliamo connettere. Sempre nella stessa cartella precedente eseguiamo il comando per avviare un Client ***java CodiceV2.Client*** seguito da indirizzo IP (in questo caso "**localhost**") e porta del Server.

```
% java CodiceV2.Client localhost 25565
```

Il Client sarà ora connesso al Server e potrà incominciare a effettuare le operazioni dopo aver inserito il suo UserName.

```
Connected
Enter username
```

Lato Server ci verrà notificata la connessione del nuovo Client e tornerà in ascolto per nuove connessioni.

```
Listening...
Connected
Listening...
```

Informazioni sui comandi dell'applicazione

Comandi Client:

• create "nomeFile" : crea il file e restituisce un messaggio in base all’esito dell'operazione.

• read "nomeFile" : legge il file richiesto e termina con la stringa ":close" la sessione di lettura.

• rename "nomeFile" "nomeFileNuovo" : rinomina il nome del file e restituisce un messaggio sull'esito dell'operazione.

• delete "nomeFile" : elimina il file, se esiste, e restituisce un messaggio in base all’esito dell'operazione.

• edit "nomeFile" : permette di modificare il file selezionato, se esistente, con le seguenti operazioni:

1. :backspace : elimina l’ultima riga del file, se esistente.

2. :close : chiude il file ed esce dalla sessione di scrittura.

3. Ogni comando che non inizia con ‘ : ’ : scrive sul file quello che viene digitato

• quit : termina la connessione e chiude il Client.

Nota: per lavorare con le operazioni sui file basterà scrivere il nome del file senza preoccuparsi della sua estensione.

Comandi Server :

• info : stampa le informazioni sul server riguardo le sessioni di lettura/scrittura e il numero di file.

• quit : chiude il server e disconnette i client.