Documentazione Progetto Sistemi Operativi 2022

Nome Gruppo: **WishTXT**

Link progetto: https://gitlab.com/Giosc4/labso_wishtxt

Data: 04.07.2022

Componenti:

Giovanni Maria Savoca 970094

Juri Hartmann 977457

Niccolò Bellucci 998755

Descrizione degli strumenti utilizzati per l'organizzazione.

Dalla consegna del progetto, abbiamo concordato tra di noi come suddividere il carico di lavoro:

Niccolò Bellucci si è occupato della parte di server inerente alla connessione tra server e client e gestione tra essi e i file.

Juri Hartmann si è occupato della parte di server inerente alla concorrenza, i problemi legati ad essa e la sincronizzazione.

Giovanni Maria Savoca si è occupato della parte di client principalemente l'interfaccia comandi e il loro utilizzo e la documentazione.

Anche se comunque tutto il gruppo ha dato una mano in tutti gli ambiti seppur minore in alcuni di essi.

Durante la comprensione e la scrittura del progetto abbiamo lavorato sia online sia di persona; quindi, spesso il lavoro è stato caricato (su GitLab) da uno ma sono tutti che hanno lavorato.

Come da specifiche abbiamo creato un repository su GitLab dove abbiamo caricato gradualmente il codice sorgente, lavorando insieme abbiamo condiviso il codice attraverso un'estensione di Visual Studio Code chiamata Live Share.

Che cosa fa il progetto?

Questo programma è composto da due file main(eseguibili), uno è il server e l'altro è il client.

Il server deve essere in esecuzione quando i clients tentano di accedere per farli comunicare con il database, le operazioni concesse sono di lettura o modifica di file.

I clients possono fare richieste di lettura e scrittura. I lettori hanno la possibilità di leggere il file, invece gli scrittori possono modificare un file esistente oppure crearne di nuovi. Ogni nuovo client si crea un nuovo thread client che comunica con il server.

Abbiamo bisogno di controllare ogni accesso ad ognuno dei file, attraverso i semafori e la classe Syncro di Java. Questo semaforo decide se il lock è occupato dal lettore o dallo scrittore.

Quando il lettore sta leggendo un file, altri lettori hanno accesso al file, ma non gli scrittori.

Invece, per quanto riguarda lo scrittore, se un client sta modificando un file nessun altro lettore o scrittore può accedere al file.

L'utilizzo dei semafori è importante per i problemi di concorrenza tra file, per essere sicuri che se qualcuno vuole modificare un file lo faccia con sicurezza di non perdere le informazioni per colpa di una collisione.

Il server/Main.java contiene un'HashMap con il nome file e il suo semaforo. Questo semaforo serve a bloccare il file.

DESCRIZIONE E DISCUSSIONE DEL PROGETTO DI IMPLEMENTAZIONE

Troviamo 3 directory diverse: in “client” troviamo i file inerenti al Client. Per poter compilare questo programma dobbiamo prima di tutto aver già compilato correttamente il main nella cartella “server”.

n.b. verrà spiegato successivamente in dettaglio

Da terminale dobbiamo entrare nella directory del server per farlo partire.

Il server, quando viene lanciato, ha bisogno della porta del collegamento, per semplicità possiamo utilizzare ‘1234’. Dopo aver fatto partire il Main del server, possiamo far partire più client. Per i clienti il procedimento è simile. Quando da terminale vogliamo far partire un nuovo client dobbiamo dargli l’indirizzo host del server e la stessa porta assegnata. Se stiamo eseguendo sia il server che il client dallo stesso computer possiamo inserire “localhost”.

Se il collegamento è andato a buon termine sulla console sia del server che del client leggiamo “Connected”.

Metodi file fileHandler

Quando i client sono connessi con il server si possono inviare i comandi per leggere, modificare, creare o eliminare file. Questi comandi sono all’interno della classe FileHandler.java:

La classe FileHandler è la classe richiamata dal ClientHandler nel metodo:

```
private boolean getResponse(ObjectOutputStream toClient, ObjectInputStream fromClient) throws IOException, ClassNotFoundException {
```

All’interno della classe abbiamo i metodi che consentono al Database di leggere, modificare, creare o eliminare file o righe dai file esistenti.

Nella classe troviamo questi metodi:

```
public String getFilesName() {
```

Che attraverso la path caricata dal costruttore ci consente di avere una lista con tutti i file del database. Tutti i file possono essere creati soltanto all’interno della path specifica.

```
public String renameFile(String oldName, String newName){
```

Questo metodo richiede in input il nome del file da rinominare e il nuovo nome. Il metodo ritorna un errore nel caso newFile sia già esistente oppure un "Rename successful" se il cambio nome ha avuto successo.

```
public String deleteFile(String fileName){
```

Il metodo deleteFile elimina dal database il file dato in input come String

```
public String readFile(String fileName){
```

Questo metodo vuole in input il nome del file da leggere e ritorna una stringa formattata con il contenuto del file. Il metodo ritorna un errore se il fileName non esiste nel database.

```
public String newFile(String fileName) throws IOException{
```

Questo metodo crea un file nel database con il nome dato in input dal client. Ritorna una stringa di commento se l'operazione è andata a buon fine.

```
public void writeLine(String fileName, String input) throws IOException {
```

Il metodo vuole in input il nome file e la String con il contenuto da aggiungere al file selezionato.

```
public void backSpace(String fileName) throws IOException {
```

Questo metodo elimina l'ultima riga del file dato in input.

La Classe ClientHandler è una classe che crea i Threads, infatti troviamo il metodo Override run(), metodo richiamato quando facciamo partire il thread ClientHandler

```
//Delega la gestione della nuova connessione a un thread ClientHandler dedicato
Thread clientHandlerThread = new Thread(new ClientHandler(s, critSecHndl));
clientHandlerThread.start();
```

Quando parte il thread, vediamo partire il metodo run(), questo metodo crea un thread per ogni client connesso al server. Controlla anche se il messaggio che riceve ha con sé un comando che è segnato dopo dei ":". Se il messaggio inviato è comprensibile per il programma, richiama il metodo getResponse() che richiama il metodo pertinente alla richiesta avvenuta dal client. Qui, infatti, è possibile controllare il comando da eseguire e richiamare il metodo proprio.

```
private boolean getResponse(ObjectOutputStream toClient, ObjectInputStream fromClient) throws IOException, ClassNotFoundException {
```

Nel caso in cui viene digitato ":edit … "troveremo un altro metodo che ci aiuta con i semafori per la scrittura del file.

Il nostro progetto vuole che se un client sta modificando un file, nessun'altro può leggere o modificare lo stesso file.

```
private void editFile(ReaderWriterSem semaphore, FileHandler fileHandler, ObjectInputStream fromClient, ObjectOutputStream toClient)
```

Similmente il metodo readFile serve per far partire un semaforo di sola lettura. Questo vuol dire che quando il primo client vuole solo leggere il file, questo viene bloccato dal semaforo per chiunque volesse modificarlo, ma a differenza del editFile (che consente a solo 1 client di leggere e modificare) consente i client di leggere lo stesso file.

DESCRIZIONE DEI PROBLEMI RISCONTRATI

Problemi sulle risorse condivise e concorrenza

Le risorse condivise del progetto sono l'HashMap con i nomi dei file e i semafori annessi e i file della cartella dove i clients hanno accesso. La situazione di mutua esclusione avviene quando entriamo in modalità edit, in cui il server concede la risorsa file ad un solo client impedendo agli altri di accedervi.

Problemi legati al modello client-server

Un problema con il modello client server è legato alla comunicazione tra essi. Il server si aspetta un comando da ogni client ed entrerà in situazione di deadlock nel caso in cui lo stesso client invia due richieste (ping-pong).

Per questo motivo quando vogliamo eseguire il comando edit, il server prima di far entrare il client nella modalità edit stampa il contenuto del file per mantenere una connessione ping-pong.

Compilazione e utilizzo

Per poter eseguire il programma in java abbiamo bisogno innanzitutto di scaricare la JVM, in seguito per compilare e utilizzare il software bisogna eseguire alcuni comandi da terminale.

Bisogna inserirsi nella cartella del programma stesso, per poi eseguire i comandi di compilazione sia per il server sia per il client, i comandi sono i seguenti:

comando per compilare il server

```
javac server/Main.java
```

comando per compilare il client

```
javac client/Main.java
```

Una volta compilate le due sezioni di codice bisognerà eseguirle attraverso i seguenti comandi:

n.b. bisognera eseguire obbligatoriamente prima il server per permettere poi ai client di connettersi

comando per eseguire il server

```
java server.Main [porta]
```

comando per eseguire il client

```
java client.Main [indirizzo] [porta]
```

Per utilizzare il software bisognerà digitare determianti comadi dal terminale del client:

:new [nomeFile.txt]

permette di creare un nuovo file

:rename [nomeVecchio.txt];[nomeNuovo.txt]

permette di rinominare un file già esistente

:delete [nomeFile.txt]

permette di eliminare un file

:dir

permette di visualizzare tutti i file contenuti nella directory

:read [nomeFile.txt]

permette di visualizzare il contenuto di un file

:edit [nomeFile.txt]

permette di modificare il contenuto di un file, aggiungendo o eliminando una riga del file:

backspace:

permette di eliminare l'ultima riga del file

exit:

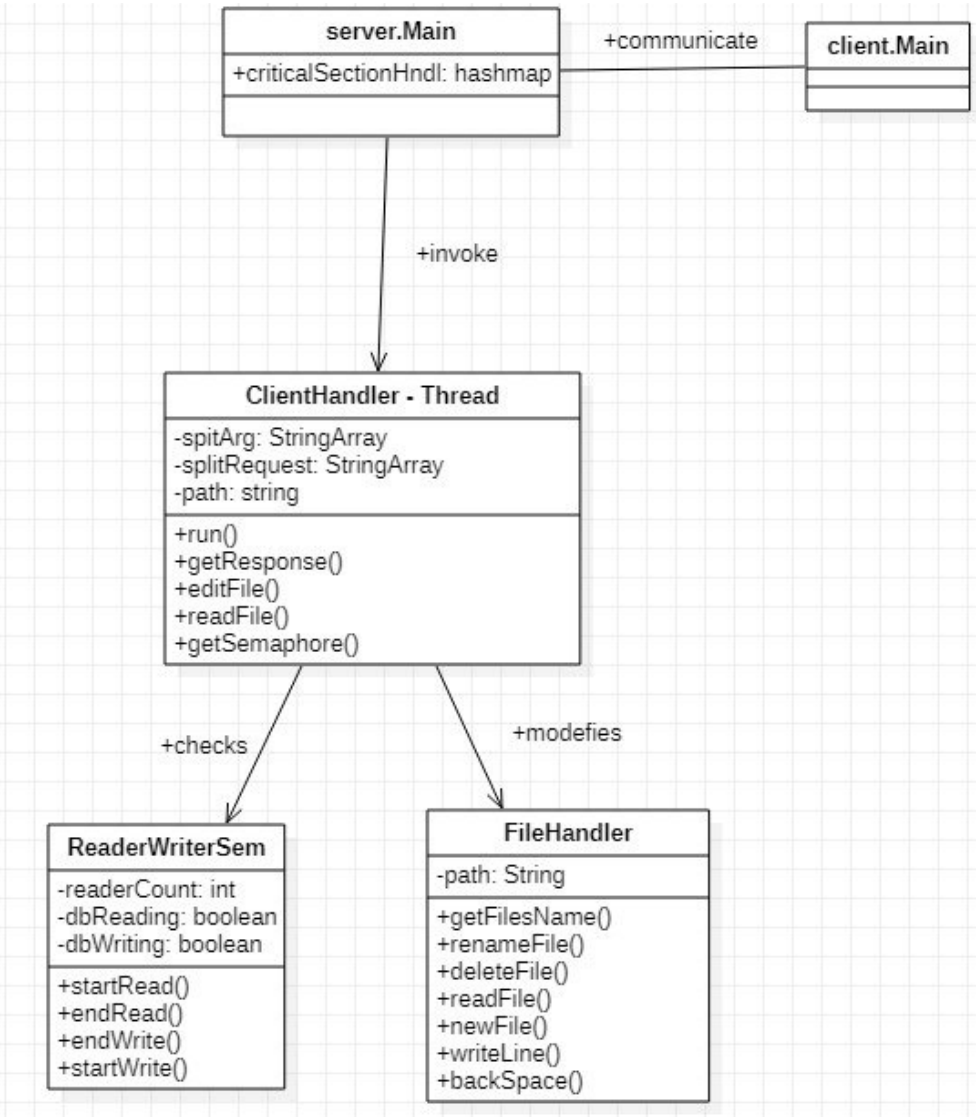
permette di uscire dalla modalità edit

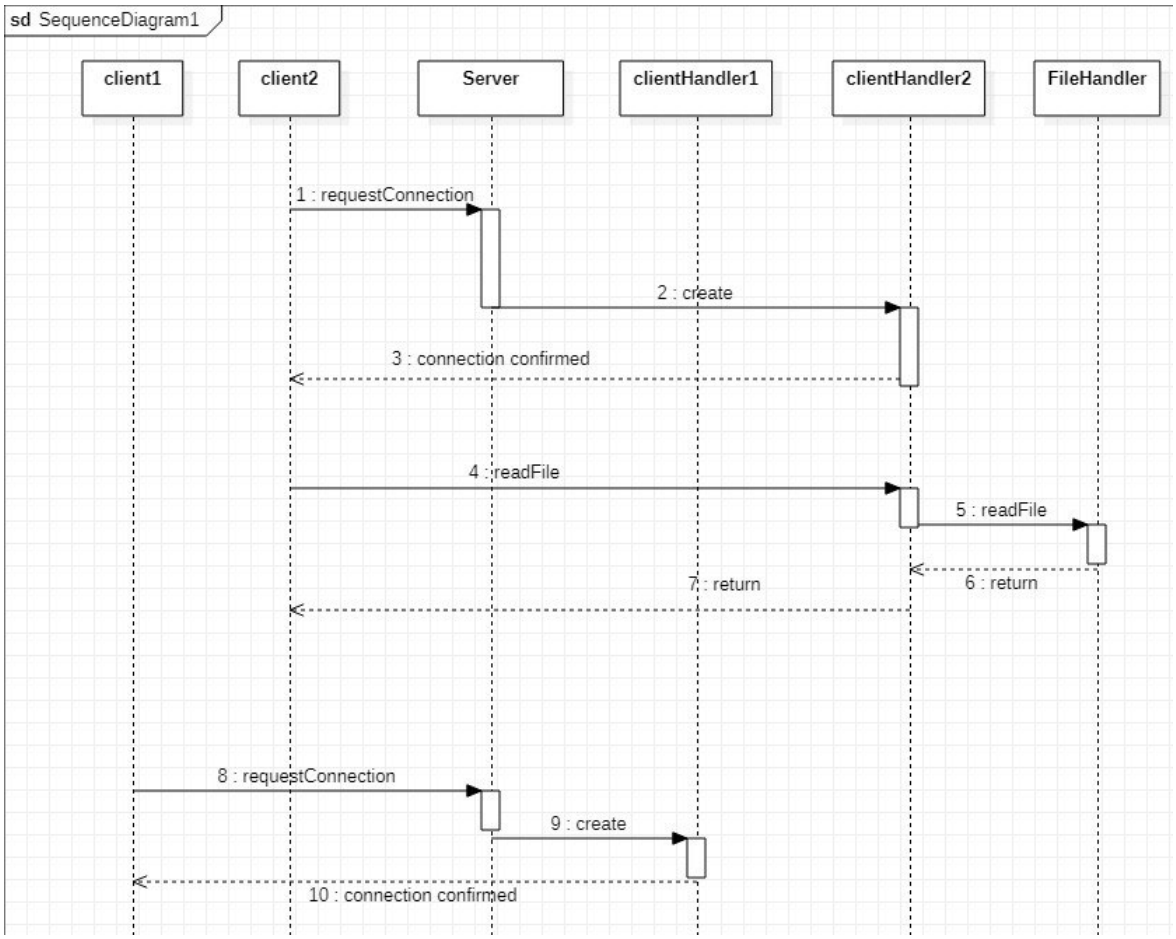
:quit

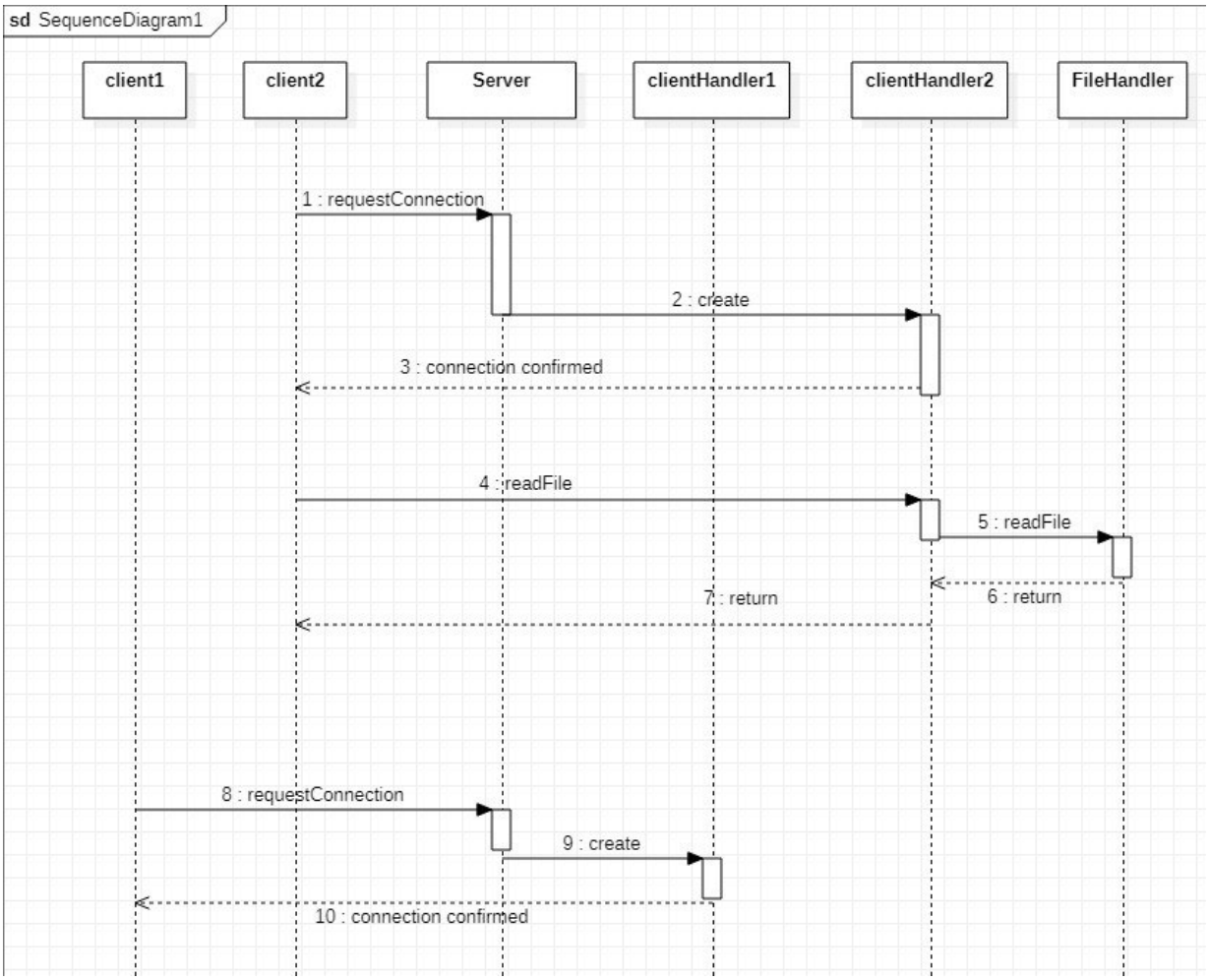
permette di uscire dal programma, chiudendo il client

A fine descrizione ci sono delle mappe concettuali ULM per la comprensione delle classi, dei metodi e su come funziona parte dei processi.

Questa prima mappa riguarda le classi create con i loro metodi. Il comando inizia dal client.Main, che comunica con il server.Main e questo, comprende il comando e richiama la classe FileHandler con il metodo richiesto dal client.


server.Main
+criticalSectionHndl: hashmap
+communicate
client.Main
+invoke
ClientHandler - Thread
-spitArg: StringArray
-splitRequest: StringArray
-path: string
+run()
+getResponse()
+editFile()
+readFile()
+getSemaphore()
+checks
+modefies
ReaderWriterSem
-readerCount: int
-dbReading: boolean
-dbWriting: boolean
+startRead()
+endRead()
+endWrite()
+startWrite()
FileHandler
-path: String
+getFilesName()
+renameFile()
+deleteFile()
+readFile()
+newFile()
+writeLine()
+backSpace()


sd SequenceDiagram1
client1
client2
Server
clientHandler1
clientHandler2
FileHandler
1 : requestConnection
2 : create
3 : connection confirmed
4 : readFile
5 : readFile
6 : return
7 : return
8 : requestConnection
9 : create
10 : connection confirmed


sd SequenceDiagram1
client1
client2
Server
clientHandler1
clientHandler2
FileHandler
1 : requestConnection
2 : create
3 : connection confirmed
4 : readFile
5 : readFile
6 : return
7 : return
8 : requestConnection
9 : create
10 : connection confirmed