Laboratorio di Sistemi Operativi A. A. 2021-22

Nome progetto: **WishTXT**

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

Indirizzo email: giovannimaria.savoca@studio.unibo.it

Data di consegna: 28.11.2022

Componenti:

Giovanni Maria Savoca	970094
Juri Horstmann	977457
Niccolò Bellucci	998755

INDICE

Descrizione del progetto:

Architettura generale:

Il progetto è suddiviso in tre cartelle: *server*, *client* e *utils*. Nelle prime due cartelle troviamo il codice sorgente delle corrispettive *Main*. La cartella *utils* contiene i file *.java* necessari al server per poter gestire le connessioni, i comandi e la concorrenza. In seguito, verranno spiegati alcuni file java della cartella utils.

La visione di alto livello è la seguente:

Il server deve rimanere attivo e connesso ad un LAN oppure ad un indirizzo statico nel web.

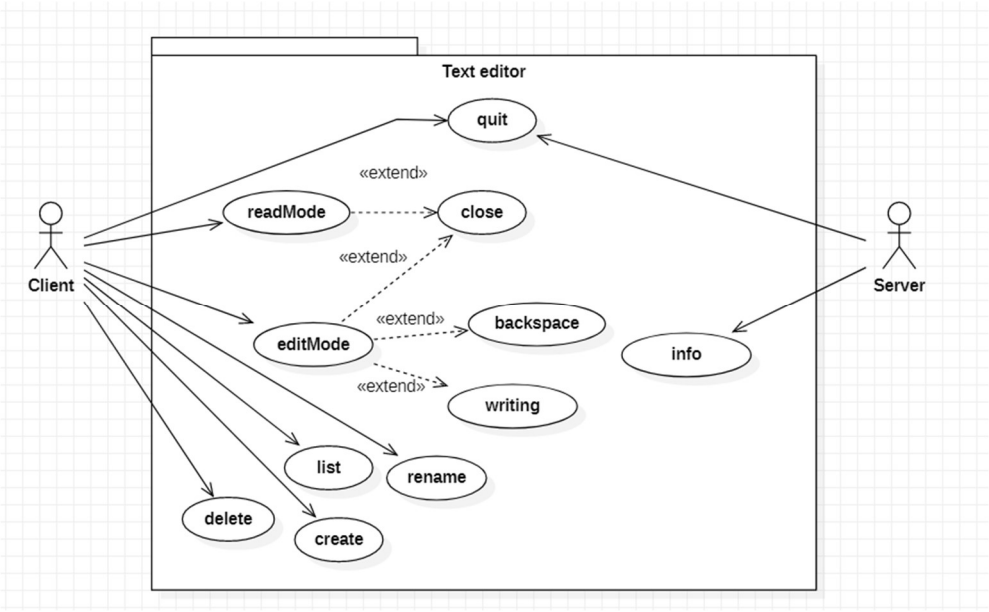
I client si possono connettere (attraverso LAN oppure il web) al server con una connessione di tipo TCP/IP, ovvero il protocollo più affidabile.

Nell'istante della richiesta di connessione da parte del client, il server inizializza una nuova istanza (o thread), per poter gestire il singolo client. Se non avvengono problemi di connessione, il client potrà usare i comandi descritti nelle specifiche del progetto, ovvero: *list*, *create*, *read*, *edit*, *rename*, *delete* e *quit*.

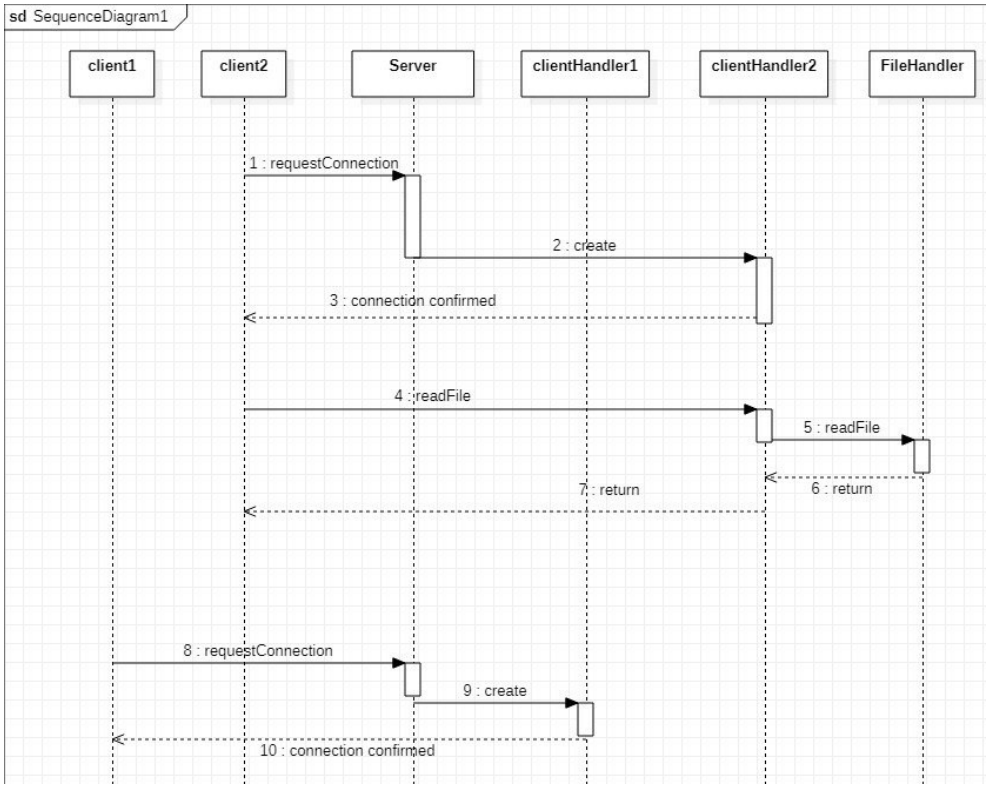
Tutte le modifiche vengono apportate all'interno di una specifica cartella scelta dal server (che chiameremo *root*), dalla quale per ovvi motivi di sicurezza non è possibile uscire. Per la stessa ragione abbiamo deciso di non offrire la possibilità ai client di accedere ad altre cartelle del server. Dare la possibilità ai client di navigare le cartelle sottostanti, apre la porta alla ricerca di un exploit da parte di client malintenzionati che potrebbe permettere ai client di uscire dalla cartella *root*, dando totale accesso, modifica e cancellazione di tutti i dati del server. Una possibile soluzione per permettere ai client di navigare le cartelle sarebbe di isolare una parte della memoria dal resto del sistema, con, ad esempio, una virtual memory.

Quando un client si disconnette dal server in modo inaspettato oppure col comando *quit*, il thread dedicato a quel client viene ucciso assieme a tutte le risorse dedicate alla gestione della concorrenza collegate con quel client.

Di seguito riportiamo alcuni diagrammi UML.

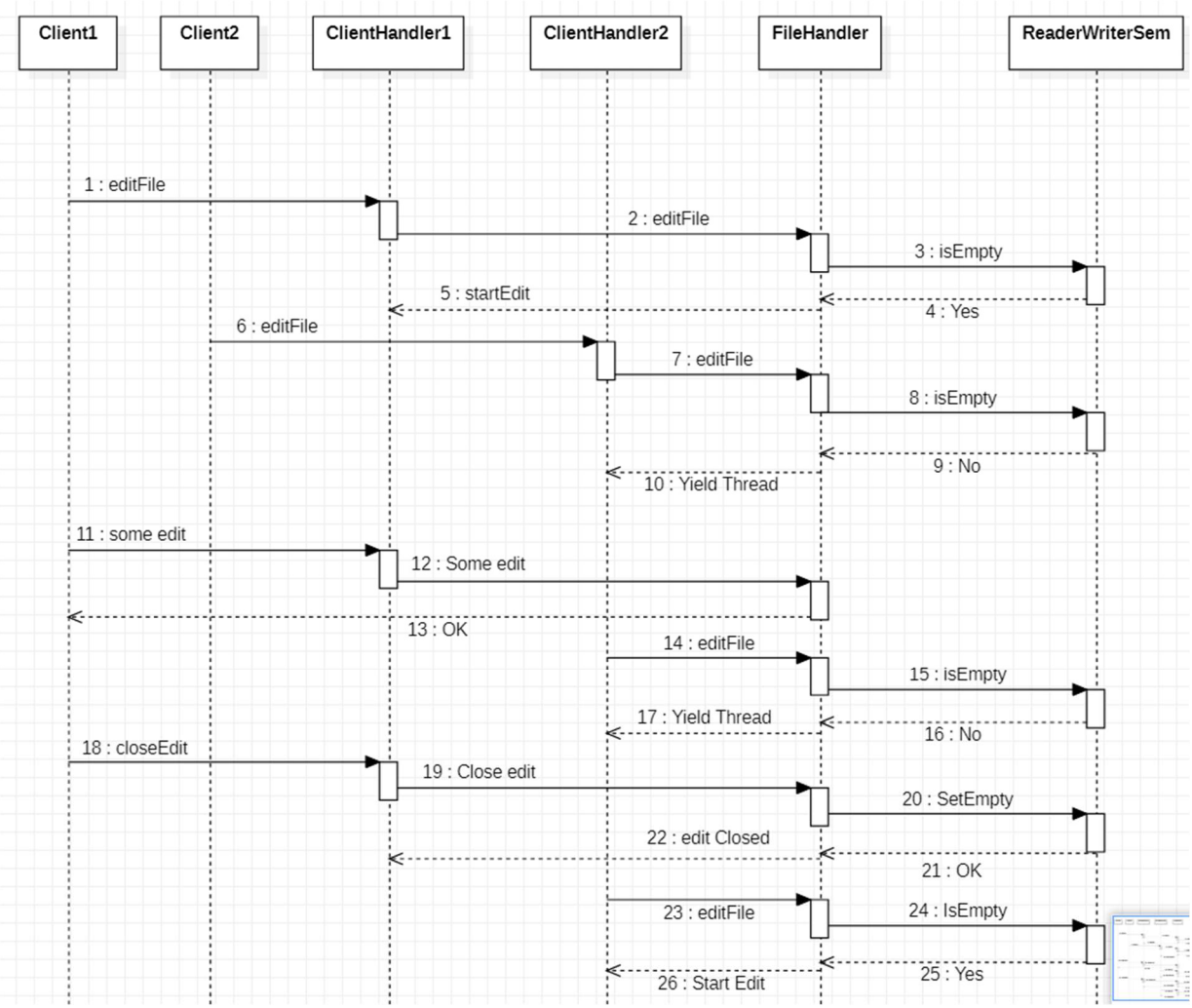


Un “use case” diagramma che spiega tutti i comandi esistenti nel nostro text editor.



Il diagramma di sequenza descrive una prima connessione ad un server e una classica situazione di come viene gestito un comando (senza considerare la concorrenza).

Nel prossimo esempio vediamo invece come viene gestita la concorrenza, con due client che cercano di modificare lo stesso file in “contemporanea”:
In questo diagramma di sequenza non siamo riusciti a creare il grafico sulla classe *ReaderWriterSem,* che gestisce internamente lo “yield thread” che troviamo nei punti 10 e 17.



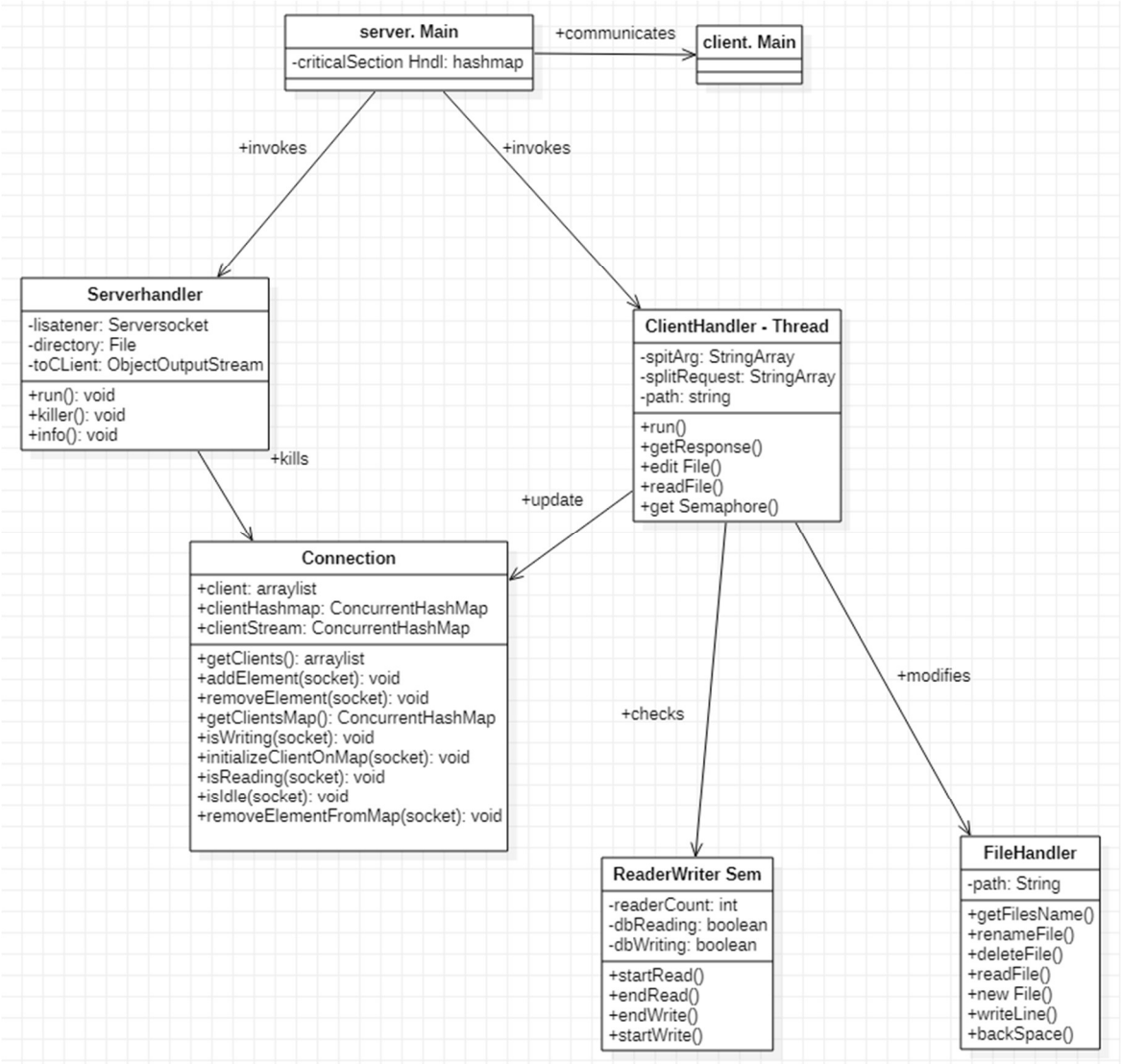
Descrizione dettagliata delle singole componenti:

Il client: all'esecuzione del programma il client inserisce l'indirizzo IP del server e la porta sulla quale si deve connettere.
Quando un client è connesso ad un server, si avvia una comunicazione ping-pong dove inizialmente il client scrive un comando sul terminale, e il server invia una risposta.
Se il client inserisce il comando *quit* il processo viene internamente terminato.

Il server:
La relativa divisione dei compiti al livello di thread è:

- 1 thread *Main* per gestire le nuove connessioni
- 1 thread *ServerHandler* per gestire i comandi dell'utente server (*info* e *quit*)
- "n" thread *ClientHandler* per gestire ed eseguire i comandi dei singoli client.
  "n" corrisponde al numero di client connessi in qualsiasi momento.

Per poter visualizzare meglio la divisione dei compiti al livello di classi ci possiamo aiutare con il seguente diagramma di classe:



Nel diagramma UML, manca una classe che abbiamo aggiunto per alleggerire il *ClientHandler*: *ModesHandler*, la quale gestisce le modalità dei comandi read/write.

La classe *Connection* tiene aggiornato lo stato di ogni singolo thread C*lientHandler*, queste informazioni ci servono per il comando *info*, ma anche per chiudere la connessione tra client e server senza sprecare risorse.

La classe *ReaderWriterSem* risolve il problema del reader-writer utilizzando e gestendo i semafori.

La classe *FileHandler* attua le operazioni di creazione e modifica dei singoli file.

La classe *ClientHandler* è senza dubbio la classe più complicata per la quale vale la pena soffermarsi.

Il suo lavoro è triplice:

Ricevere e mandare i messaggi al client;

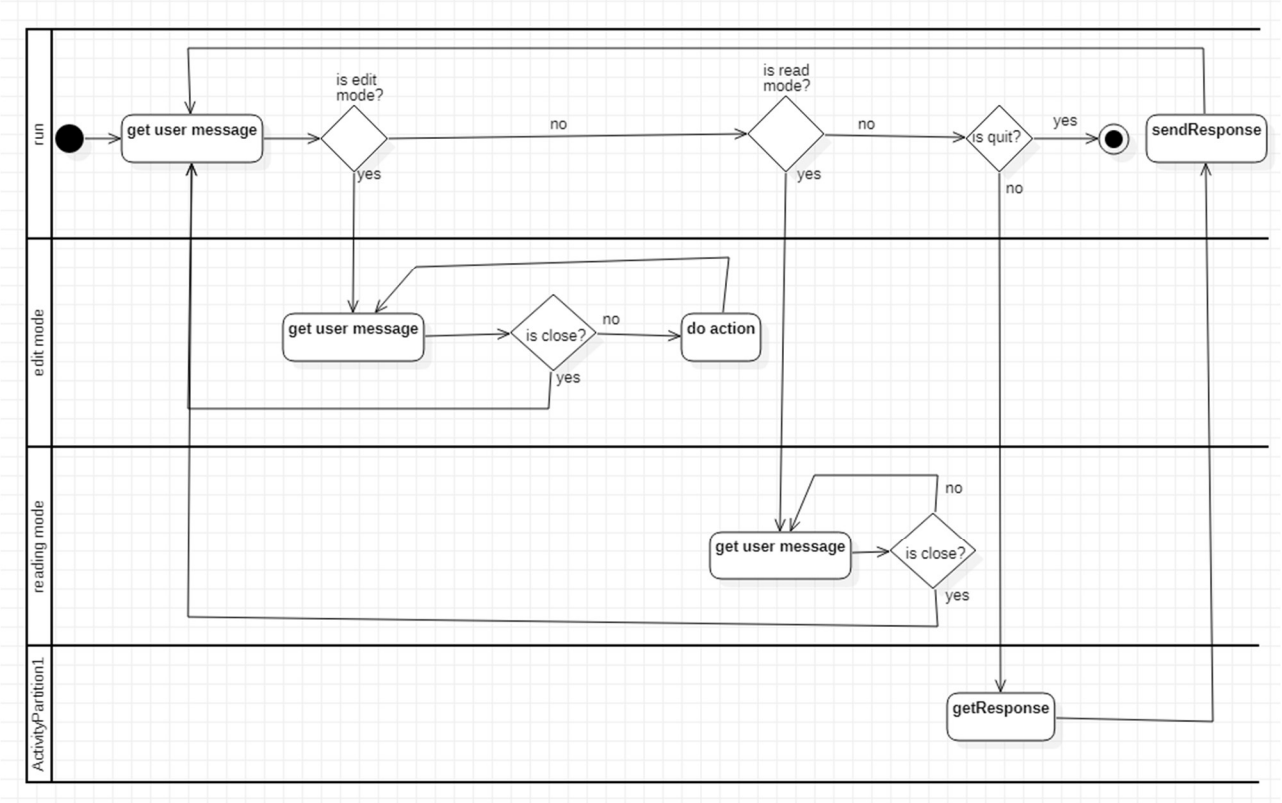
Interpretare i comandi del client;

Gestire l'entrata e l'uscita dalla sezione critica.

Come verrà approfondito nella sezione "descrizione dei problemi" il programma sfrutta la comunicazione ping-pong, il cui pro è la semplicità di comunicazione, ma il cui svantaggio è che non si possono mandare due messaggi di seguito, altrimenti il client e il server entrano in deadlock, dove entrambi aspettano di ricevere un messaggio dall'altro. Di conseguenza a volte abbiamo dovuto mandare un messaggio anche se in realtà non era strettamente necessario.

Il metodo *run()* in *ClientHandler*, che gestisce la comunicazione, è sostanzialmente un loop infinito nel quale il processo aspetta un messaggio dall'utente, riceve una risposta da un altro metodo e infine manda la risposta al client. L'unico modo per interrompere il loop è con il comando *quit*, destinato all'interruzione della connessione. Lo stesso sistema di loop lo troviamo anche nei metodi che si occupano delle modalità di scrittura e lettura.

In seguito, troviamo un diagramma delle attività che spiega il funzionamento dei loop e le modalità scrittura e lettura.



Infine, poniamo l'attenzione al metodo *backspace(String fileName)* nella classe *FileHandler*. In questo metodo, dove dobbiamo cancellare l'ultima riga del file, lavoriamo al contrario. Con la variabile *length* come pointer andiamo a controllare il valore ASCII di ogni carattere all'interno del file scelto. Questa operazione viene ripetuta finché il valore ASCII non equivale a 10, ovvero una nuova riga *[\n].* A quel punto il file viene troncato fino all'inizio dell'ultima riga.

Suddivisione interna dei lavori.

- Giovanni Maria Savoca: si è occupato della parte di client principalmente l'interfaccia comandi e il loro utilizzo;

- Niccolò Bellucci: si è occupato della parte di server inerente alla connessione client-server e la gestione delle eccezioni.

- Juri Horstmann: si è occupato dell'implementazione, gestione della concorrenza dei file e creazione della documentazione.

Annotiamo che la suddivisione del lavoro non era così rigida come descritto qua sopra. Spesso ci siamo aiutati a vicenda per poter superare i problemi con cui ci siamo scontrati.

Descrizione e discussione del processo di implementazione:

Descrizione dei problemi.

Problemi legati alla concorrenza:

Decisamente il più grande problema coi cui ci siamo scontrati nell'intero progetto è stata un'implementazione sbagliata della mutua esclusione. Abbiamo consegnato il progetto usando la soluzione al problema reader-writer, ma implementandola soltanto nella modalità di lettura e scrittura.

Fino alla presentazione del progetto eravamo convinti di aver implementato correttamente la concorrenza. Col senno di poi, possiamo dire che bisognava ovviamente gestire la concorrenza anche nel caso dei comandi *rename* e *delete*, poiché è possibile che prima di

completare la cancellazione di un file, le risorse a quel processo vengano tolte e date ad un altro processo che potrebbe utilizzare lo stesso file. Più precisamente, coi comandi *rename* e *delete* bisogna usare i semafori di tipo *writer* per bloccare l'accesso di qualsiasi thread intenzionato a leggere o modificare il file che sta per essere rinominato o cancellato.
Risolvere questo problema non è stato laborioso, dal momento che dovevamo semplicemente aggiungere le sezioni critiche tra i comandi *rename* e *delete*.

Un altro grande problema era la gestione dei dati inerenti ai semafori. La soluzione da noi proposta è una *HashMap* al cui interno troviamo come *key* il nome del file e come *value* l'oggetto *semaphore* che gestisce i semafori per quel file. Questa *HashMap* è una risorsa condivisa tra tutti i thread, ad eccetto del thread che gestisce i comandi da terminale del server.

<u>Problemi legati al modello client-server:</u>
Il problema più grande della connessione tra i clients e il server è stato il rischio di deadlock. Una difficoltà è stata la gestione del client e del server, la quale evita di portare il sistema in una situazione di deadlock, nel momento in cui aspettano un messaggio l'uno dall'altro.
Per evitare lo stallo del programma, il server in determinati casi comunica e risveglia il client. Abbiamo scelto di implementare una comunicazione ping-pong per evitare di entrare in deadlock. Esempio emblematico di questa soluzione è ciò che succede quando lo user vuole accedere alla modalità *edit*. Prima di eseguire il metodo, l'intero testo del file viene stampato sul terminale del client. Non l'abbiamo fatto solo per migliorare l'user experience ma anche per svegliare il client con un messaggio, il che permette allo user di interagire nuovamente con il terminale.
Un'alternativa sarebbe programmare un sistema di comunicazione più complesso, dove non solo esiste il ping-pong, ma alcuni comandi alterano la modalità di comunicazione tra server e client. Un esempio: immaginiamo di ampliare la comunicazione server-client per il comando *edit,* vogliamo che il client possa inviare due messaggi di fila
es.

1) edit [nomeFile]
2) [testo da aggiungere]

In questo caso dovremmo gestire il codice dalla classe *Main* del client e nella classe *ClientHandler*. Nella *Main* dovremmo implementare che il comando *if(msg == "edit")* comporta l'invio di un altro messaggio, mentre analogamente se il *ClientHandler* riceve il

comando *edit* allora aspetterà di ricevere un secondo messaggio.

L'altro problema che abbiamo riscontrato, per cui abbiamo dovuto creare la classe a parte *Connection*, è il comando *info* dal lato server. La classe si occupa di salvare tutti i dati dei client connessi, permettendoci di tenere traccia di tutte le connessioni in entrata al server e i relativi stream, così facendo sapremo il numero di client connessi in quel momento e la modalità in cui si trovano (read, write, idle). Inoltre, nel caso in cui il server dovesse essere spento attraverso il comando *quit*, si occuperà di contattare i client in broadcast effettuando la chiusura delle connessioni ancora attive.

Descrizione degli strumenti utilizzati:

Sviluppo del progetto:

Per lo sviluppo del progetto abbiamo comunicato utilizzando Discord, dove discutevamo: dei problemi, dei bug e ci aggiornavamo con tutte le modifiche che ognuno dei partecipanti ha fatto. Queste discussioni sono state molto importanti per la parte di debug del codice.
Per lo sviluppo tecnico, la scrittura e revisione del codice abbiamo utilizzato sia IntelliJ IDEA sia Visual Studio Code.
Per la condivisione del codice, abbiamo utilizzato la piattaforma GitLab dove caricavamo le modifiche del codice nel corso del tempo, grazie all'utilizzo di questa piattaforma abbiamo imparato ad utilizzare Git per caricare il codice aggiornato.

Comunicazione tra i membri:

Le comunicazioni brevi e per iscritto avvenivano attraverso WhatsApp, mentre su Discord abbiamo avuto degli scambi al telefono più lunghi. Queste piattaforme ci hanno aiutato per le discussioni dei vari problemi riscontrati e per aiutarci a vicenda. Su Discord è stato molto utile spiegare il proprio codice agli altri membri del gruppo per poter al meglio comprendere il sistema e integrarlo con la propria parte di codice.

Condividere il codice sorgente

Come da specifiche abbiamo usato il sistema Git per condividere il progetto. Entrando nel dettaglio abbiamo condiviso il codice su GitLab, ovvero un servizio che offre gratuitamente uno spazio cloud, sulla piattaforma abbiamo caricato i vari aggiornamenti del progetto.
Siamo riusciti ad integrare il sistema Git nei text editor e nell' IDE, potendo così utilizzare Git

più comodamente, oltre ad accedere al codice del progetto attraverso comandi *push* e *pull* eseguiti su Git Bash.

Tenere traccia del lavoro svolto.

Inizialmente abbiamo tenuto traccia del lavoro svolto utilizzando una copia delle specifiche del progetto, tenendo conto delle parti completate e quelle da completare. Abbiamo deciso di evitare trello o altri sistemi scrum non ritenendo il progetto così complesso. Col senno di poi, possiamo affermare che questo fu un errore e di sicuro ci ha causato alcuni problemi di organizzazione del lavoro. Dopo la prima consegna, in cui siamo stati respinti, il professore ci ha fatto notare gli errori del nostro progetto. Ci siamo segnati tutti gli errori e con un file condiviso abbiamo poi corretto tutte le criticità che sono emerse.

Istruzioni passo a passo per usare l'applicazione:

1. Per poter eseguire i due software bisognerà innanzitutto compilare i sorgenti attraverso il terminale, ciò genererà i file **.class* che poi andremo ad eseguire attraverso altri comandi.
2. Ci posizioniamo all'interno della cartella *src/* ed eseguiremo il seguente comando, se ricevuto un feedback positivo dal terminale vuol dire che tutto è andato a buon fine permettendoci di proseguire.

Compilazione ed esecuzione del progetto:

Una volta compilati i codici sorgenti, il primo file da eseguire sarà *server.Main*, in quanto avvierà il nostro server a cui poi i nostri client si collegheranno, l'esecuzione del server richiederà alcuni argomenti che gli dovremo dare obbligatoriamente:

- La path in cui verranno salvati i file a cui i client potranno accedere. Il percorso è da intendere come relativo e non assoluto in quanto Java si occuperà di creare la cartella di destinazione all'interno della cartella del progetto.

- La porta al quale il servizio si metterà in ascolto di eventuali connessioni. Crediamo sia utile accennare che non si può semplicemente usare una porta a scelta, ma bisogna scegliere una porta non utilizzata da altri servizi. Il range di porte non assegnate è 48620-

49150, ma esistono anche altre porte libere o non ufficiali. Per verificare quali porte sono libere possiamo consultare il sito della IANA (www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml)

Successivamente eseguendo il file *client.Main*, indicando alcuni argomenti:

- L'indirizzo IP del server. Questo indirizzo può essere il *localhost* se si esegue il server e il client sulla stessa macchina oppure l'indirizzo pubblico del server se si esegue sulla rete.
- La porta a cui il client dovrà richiedere la connessione.

Sintassi:

Compilazione:

javac server/Main.java && javac client/Main.java

Esecuzione:

java server.Main [PATH_RELATIVA] [PORT]

java client.Main [INDIRIZZO IP] [PORT]

In base ai sistemi operativi potrebbe capitare che se chiuso male il server la porta possa rimanere occupata dal processo rimasto appeso, per liberare la porta basterà chiudere e riaprire una nuova finestra del terminale.

Il server riceve una notifica nel momento in cui si connette un nuovo client, il client riceve anch'esso una notifica quando si connette o disconnette dal server.

Il client ha a sua disposizione alcuni comandi con cui può creare, modificare e cancellare file di testo. In seguito, troviamo una breve lista dei comandi possibili per il client e la loro sintassi:

create [fileName]

Con questo comando si crea un file txt all'interno della directory del server. Se quel nome esiste già viene mandato un messaggio al client.

rename [old filename].txt [new fileName]

Con questo comando possiamo rinominare i file .

Il thread entra in *wait()* se qualcuno sta leggendo o modificando il file da rinominare.

delete [fileName]

Con questo comando il file viene eliminato dalla cartella.

quit

Con questo comando si chiude la connessione al server.

L'unico comando dove è obbligatorio l'utilizzo dell'estensione **.txt* è il *rename* in cui il nome del file da rinominare dovrà essere composto obbligatoriamente anche dall'estensione. In tutti gli altri comandi che richiedono il nome del file non è obbligatorio specificare l'estensione in quanto il software si occuperà automaticamente di gestire questo caso. Questa feature non è obbligatoria, l'utente potrà comunque aggiungerla a suo piacimento.

Inoltre, abbiamo due comandi che attivano due specifiche sessioni: *read* e *edit*.
Con il primo comando entriamo nella modalità di sola lettura del file, dalla quale possiamo uscire con il comando *:close*.
Per eseguire i comandi all'interno delle modalità *read* e *edit* il commando dovrà essere preceduto dal carattere doppio punto *[:]*.

1) *read [fileName]*
2*) :close*

Il comando *edit* è leggermente più complicato: oltre alla possibilità di chiudere la sessione modifica, abbiamo anche il comando *:backspace*, il quale cancella l'ultima riga di testo presente nel file.
Per aggiungere una riga al file basta scrivere ciò che vogliamo dopo aver avviato la modalità e premere enter:

1) *edit [fileName]*
2) *[testo da aggiungere]*
3) *:backspace*
4) *:close*

Lo user dalla parte del server ha a disposizione altri due comandi, entrambi senza argomenti da aggiungere: *quit* e *info*.

Il comando *quit* chiude la propria connessione e di conseguenza tutte le connessioni con i vari clients connessi.
Il comando *info* stampa su terminale il numero di clients connessi nelle loro modalità e il numero dei file presenti nella cartella.

Conclusione:

Portare a termine questo progetto è stato un lavoro intenso e laborioso, ma soddisfacente. Ci siamo scontrati con alcuni dei problemi discussi durante le lezioni di teoria, specialmente ai problemi legati alla concorrenza e alla mutua esclusione.
Troviamo che sia stato molte utile avere una sezione pratica nel corso dove potevamo ragionare su quali fossero i concetti base imparati durante le lezioni di teoria.
Col progetto abbiamo avuto la possibilità di capire può profondamente come la concorrenza funziona e come al meglio sfruttarla in futuro.
Un altro aspetto piacevole è stata la possibilità di lavorare in gruppo, ci siamo spesso e volentieri confrontati su come completare le richieste del progetto e nel frattempo abbiamo avuto anche la fortuna di conoscerci meglio.

Giovanni Maria Savoca
Juri Horstmann
Niccolò Bellucci