Laboratorio di Sistemi Operativi A.A. 2021-22

Andrea Serrano – 0000988271
andrea.serrano2@studio.unibo.it

Enjun Hu – 0000944041
enjun.hu@studio.unibo.it

Filippo Berveglieri – 0000974754
filippo.berveglieri@studio.unibo.it

Gabriele Centonze – 0000971019
gabriele.centonze@studio.unibo.it

02/07/2022

Nome del gruppo	Team Gli Alpaca
Email del referente	andrea.serrano2@studio.unibo.it

1 Descrizione del progetto

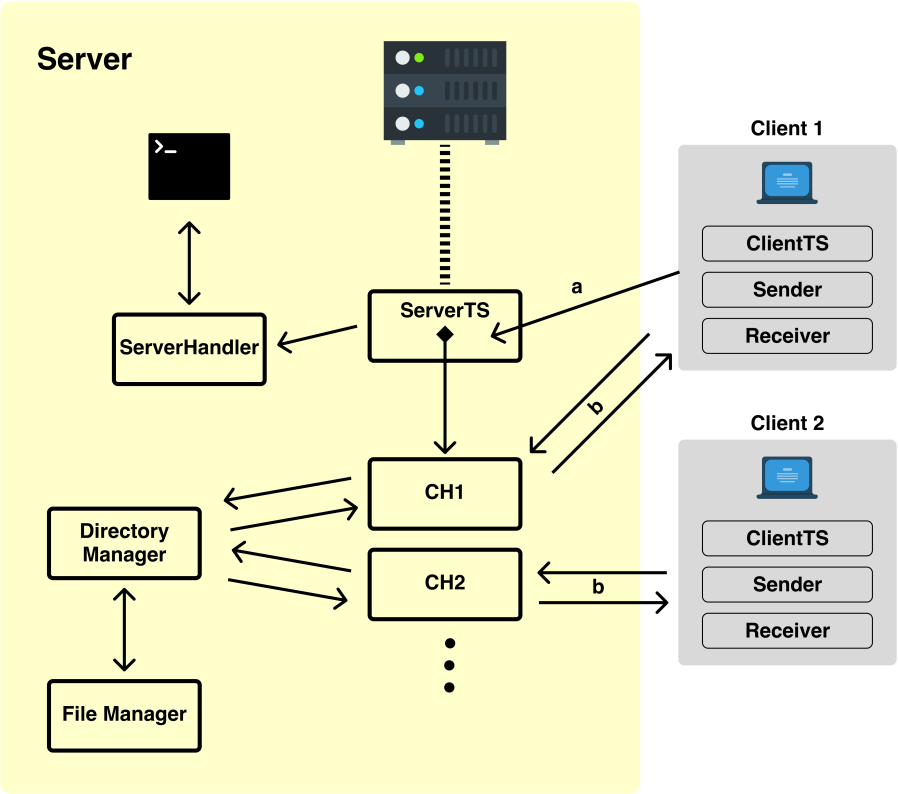
1.1 Architettura generale

L'applicazione è strutturata in due package principali: Client e Server.
Il Server ha lo schema del producer-consumer, mentre l'invio e ricezione di informazioni dal e verso il Client rispecchia lo schema di una chat, ovvero tramite lo schema sender-receiver.
Il Client si connette al Server tramite un thread principale (ClientTS), a questo punto la gestione del Client è lasciata a due thread, receiver e sender, che gestistcono rispettivamente la ricezione e l'invio di informazioni da e verso il Server.
Il Server accetta richieste di connessioni grazie ad un thread principale (ServerTS), a questo punto la gestione della connessione con il Client appena connesso è lasciata ad un thread creato appositamente (ClientHandlerTS).
In parallelo viene eseguito un ulteriore thread che si occupa esclusivamente di gestire gli input ricevuti da linea di comando.
Il Server gestisce la directory e i file in essa contenuti tramite apposite classi (DirectoryManager e FileManager). Saranno queste ultime ad operare sulla directory e sui file.
Possiamo suddividere l'architettur3 in 3 macro punti:

a. ClientTS instaura una connessione con ServerTS attraverso i socket

b. la gestione di tutta la connessione a lato Client viene lasciata a Sender e Receiver, mentre ClientHandlerTS gestirà la connessione a lato server richiamando i metodi di DirectoryManager e FileManager all'occorrenza.

c. gli input dal terminale del server vengono affidati a ServerHandlerTS



1.2 Descrizione delle componenti

Il package Client contiene le tre classi utili per l'esecuzione del Client:

- ClientTS: è la classe principale da eseguire a lato Client. Dato l'indirizzo e la porta del Server, si occupa della connessione con quest'ultimo creando un socket che utilizza proprio indirizzo e porta.
  Una volta creato, il socket viene utilizzato per istanziare due thread, uno per il Sender e uno per il Receiver (a cui viene passata anche la reference del thread del Sender). Questi due thread gestiranno i flussi di informazioni che passano attraverso il canale creato dalla connessione al Server, e quindi verranno utilizzati per gestire l'invio e la ricezione delle informazioni.
  La chiusura di ClientTS è dipendente dalla chiusura dei due thread sopracitati, se Sender e Receiver vengono terminati allora il socket verrà chiuso con conseguente chiusura della connessione.

- Sender: è una classe che estende *Runnable*, si occupa esclusivamente di reindirizzare al Server gli input dell'utente.
  Legge l'input dell'utente da console tramite *Scanner* e, visto che ha accesso al socket, viene istanziato un *PrintWriter* sull'outputStream di quest'ultimo, che verrà usato per inviare al Server i messaggi dell'utente.
  Se il messaggio inviato al Server è per l'apertura di una sessione, viene messo il thread in stato di attesa (*waiting*), così da non permettere al thread di continuare la sua esecuzione.

- Receiver: è anch'essa una classe che estende *Runnable*, si occupa di ricevere informazioni dal Server e di notificare l'utente.
  Tipicamente, legge il messaggio dal Server e lo stampa sulla console. Se però riceve un messaggio che inizia con l'identificatore del Client (messaggio di servizio) seguito da codici specifici, il suo comportamento cambia:
  - Se il messaggio finisce con il codice 101, allora il thread del Receiver proverà a risvegliare il thread del Sender, che era stato messo in uno stato di *waiting*, in attesa che si completasse l'apertura della sessione di scrittura o lettura;
  - Se il messaggio finisce con il codice 503, allora vuol dire che è stata avviata la chiusura della connessione tra il Client e il Server;

Il package Server contiene cinque classi principali, utili per l'esecuzione del Server e per la gestione dei file:

- ServerTS: è la classe principale da eseguire lato Server, data la directory su cui sono presenti i file che si vogliono trattare, e la porta su cui mettersi in ascolto, si occupa esclusivamente di accettare nuove connessioni.
  Tramite la creazione di un *ServerSocket*, si mette in ascolto di possibili Client che chiedono di connettersi; crea un'istanza di DirectoryManager; un ArrayList su cui saranno salvati tutti i socket lato Server che verranno creati, inoltre crea e lancia un istanza della classe serverHandlerTS.
  All'arrivo di un Client, il *ServerSocket* crea un apposito socket, il quale verrà usato per istanziare un oggetto della classe ClientHandlerTS; infine il socket appena creato verrà aggiunto alla lista dei socket.
  Verrà poi lasciato al thread del clientHandlerTS il controllo totale della

connessione; mentre la terminazione di ServerTS, cioè l'interruzione del *ServerSocket*, è completamente gestita dal thread del serverHandlerTS.

- ServerHandlerTS: è una classe eseguibile che implementa *Runnable* e si occupa di gestire le richieste in input dal terminale del Server.
  Ha accesso al *ServerSocket*, dell'ArrayList contenente i socket attivi e dell'istanza di DirectoryManager.
  Per tutto il suo ciclo di vita non fa altro che rimanere in ascolto di input da console e, in base ad esso, agire di conseguenza. E' al suo interno che sono implementati i comandi "info" e "quit":
  - info: estrapola la lista dei file che sono nella directory, quindi determina quanti file sono gestiti in quel momento e, basandosi sulla loro presenza o meno nel *ConcurrentHashMap* del DirectoryManager, quanti di questi sono in lettura o scrittura.
  - quit: chiude il ServerSocket, a questo punto verrà terminato ServerTS in quanto sarà sollevata una IOException che viene gestita proprio terminando il thread di ServerTS. A questo punto l'*ArrayList* contenente i socket attivi viene usata per chiudere ogni socket presente; in questo modo la connessione con ogni Client connesso verrà chiusa.

- ClientHandlerTS: è una classe eseguibile che implementa *Runnable*, si occupa di gestire la connessione e le operazioni che vorrà fare il Client. Ogni Client avrà un suo ClientHandlerTS che legge le richieste grazie ad uno *Scanner* collegato all'inputStream del socket; invoca DirectoryManager per gestire le operazioni richieste dall'utente e invia la risposta grazie ad un *PrintWriter* collegato all'outputStream del socket.
  Il suo ciclo di vita è così strutturato:
  in fase di lettura della richiesta separa il tipo di comando dall'argomento del comando. In base al tipo di comando, dunque, esegue l'apposito codice che, in tutti i casi eccetto per il comando di quit, richiamerà appositi metodi del DirectoryManager. In caso di quit il socket che del ClientHandlerTS sta controllando verrà rimosso dalla lista dei socket e verrà terminato.

- DirectoryManager: è una classe i cui metodi gestiscono la directory su cui il Server sta lavorando.
  Grazie ai metodi di DirectoryManager possiamo: creare un nuovo file nella directory (*createNewFile*), eliminare un file (*delete*), rinominare

un file (*rename*), ritornare il FileManager di uno specifico file (*getFileManager*).
Le operazioni di delete, rename e getFileManager sono synchronized così da evitare problemi di race condition.
Nel costruttore del DirectoryManager viene implementato un *ConcurrentHashMap* che è un *HashMap* "Thread-safe". Il *ConcurrentHashMap* associa a ogni file il suo FileManager creato appositamente nel caso in cui quel file venga utilizzato, sia per la lettura che per la scrittura

- FileManager: è una classe i cui metodi gestiscono la lettura/scrittura di uno specifico file e le sue informazioni riguardo l'essere utilizzato in lettura o in scrittura in un dato momento. Ogni file ha un suo FileManager.
  Per le fasi di apertura/chiusura di una sessione di lettura/scrittura vengono usati dei lock così da gestire correttamente il problema dei lettori-scrittori.

1.3 Suddivisione del lavoro

Il lavoro è stato suddiviso tra i membri del Team nel seguente modo:

- Andrea Serrano: struttura Client-Server; gestione della chiusura delle connessioni a lato Client e anche a lato Server. Implementazione della classe ServerHandlerTS; scrittura della documentazione; revisione generale del progetto e della documentazione.

- Enjun Hu: gestione dei problemi di concorrenza; creazione e gestione della classe FileManager; implementazione del Sender e Receiver; scrittura della documentazione; revisione generale del progetto e della documentazione.

- Filippo Berveglieri: implementazione del comando "create", "rename" e "delete" su ClientHandler; e i rispettivi metodi su DirectoryManager.

- Gabriele Centonze: implementazione del comando "edit" sul ClientHandler e i metodi relativi (*write* e *deleteLastRow*) sul FileManager; creazione dei grafici nella documentazione.

2 Processo di implementazione

2.1 Scelte implementative

Per l'implementazione del progetto si è usato l'approccio "outside to inside", partendo da uno schema generale, con la struttura dei vari componenti, per poi implementare di volta in volta l'interno di ogni singolo componente con le relative funzionalità richieste.
Nello sviluppo di questo progetto si è cercato di applicare il concetto generale per cui, solo le classi dei thread principali o quelli dedicati alla comunicazione dovrebbero stampare sulla console.
Inoltre come buona norma, vengono usati classi già resi disponibili da Java per risolvere i problemi più comuni; invece di reinventarsi delle classi da zero se esiste già uno che fa al caso nostro.
Viene utilizzato la documentazione ufficiale di Java (JavaDocs) per avere una comprensione migliore delle classi Java utilizzate e i tutorial ufficiali di Oracle, in particolare quelle sulla concorrenza (The Java Tutorial).
Si è deciso di non implementare la classe FileManager come un estensione della classe File, nonostante sia sensato dal punto di vista della programmazione, ma sarebbe concettualmente contorto visto che apriamo dentro di esso dei FileWriter e FileReader.
Si è deciso di creare un'eccezione personalizzata (*FileOccupiedException*) così da gestire al meglio le varie situazioni che si incontravano; diverse dalle situazioni in cui venivano lanciate le eccezioni dei metodi di java e i casi in cui non bastava ritornare 2 tipi di risultati (*true* o *false*).

2.2 Problemi principali e relative Soluzioni

Nel corso della progettazione e implementazione, si son dovuti affrontare problemi legati alla Concorrenza (Principalmente consistenza delle informazioni e gestione delle risorse condivise) ed alcuni problemi del Modello Client-Server.

2.2.1 Problemi del Modello Client-Server

Il modello Client-Server prevede che il Client sia composto da 3 thread: uno che si occupa di instaurare la connessione, uno che si occupa di inviare le richieste al Server, uno che si occupa di ricevere le riposte del Server. Mentre

il Server è composto da 2 thread fissi e tanti thread quanti sono i Client connessi: uno che si occupa di accettare nuove connessioni, uno che si occupa di gestire i comandi del Server, e ulteriori thread che si occupano di gestire i Client connessi.
I problemi riscontrati in questa fase sono stati principalmente quattro:

1. Chiusura del Server e disconnessione dei Client ancora connessi:
   il comando che implementa questa funzionalità è il comando "quit", che viene impartito dal terminale del Server, dunque il thread che riceve questo input è il thread del serverHandler. L'idea iniziale era di notificare gli altri thread coinvolti nell'operazione così che potessero terminare in autonomia. Sfruttando, però, il fatto che questi thread eseguono tutti sulla stessa macchina e possono operare sulla stessa istanza di un oggetto, si è optato per una soluzione che prevedeva la chiusura di tutto il Server direttamente dal thread di serverHandler. Per questo motivo ServerHandlerTS prende come parametri il ServerSocket e l'ArrayList dei socket connessi; in questo modo in caso di comando "quit" il ServerSocket verrà chiuso e il Server non potrà più accettare nuove connessioni, ma le connessioni già instaurate sono ancora attive e funzionanti in quanto tutti i clientHandlerThread sono ancora in esecuzione e il relativo socket è ancora attivo.
   Per risolvere questo inconveniente ed implementare, dunque, la disconnessione dei Client, si è optato per una lista che viene popolata con tutti i socket creati dal thread ServerTS in fase di instaurazione della connessione, e che viene usata in fase di chiusura del Server scorrendola e chiudendo ogni socket. In questo modo la connessione di ogni Client viene interrotta lato Server.
   Si genera, però, un inconveniente: la connessione è effettivamente terminata ma i thread ServerTS e ClientHandlerTS sono ancora in esecuzione inutilmente. Per risolvere il problema, si utilizzano le eccezioni sollevate dalla chiusura del ServerSocket e socket:

   - per quanto riguarda il ServerSocket, quindi la terminazione del thread ServerTS: il metodo accept(), che si occupa di instaurare le connessioni ritornando un socket, e' ancora in esecuzione ma, dato che il ServerSocket viene chiuso inaspettatamente dal thread del ServerHandler, solleva una IOException, che gestiamo semplicemente terminando il thread.
     Anche la creazione del ServerSocket in fase di avvio del Server

può sollevare la stessa eccezione, anche in questo caso terminiamo il thread in quanto senza ServerSocket non possiamo fare nulla, dunque è inutile mantenere in esecuzione il thread.

- per quanto riguarda i socket, quindi la terminazione dei thread ClientHandlerTS: il Wrapper dell'input stream del socket, dato che è in costante lettura di eventuali messaggi del Client, solleverà una NoSuchElementException quando il socket viene chiuso dato che non riuscirà più a leggere nulla. Sapendo che tale eccezione occorre solo in caso di chiusura del socket, questa verrà sfruttata per terminare il thread. In altre parole, visto che il socket e' chiuso, il ClientHandler puo' terminare.
  Infine, avendo aggiunto una lista di socket, in fase di chiusura della connessione, prima di chiudere il socket, questo verrà rimosso dalla lista dei socket.

2. Stampa consecutiva del Client:
   inizialmente il Client era strutturato in un solo thread, ClientTS. In questo caso il problema consisteva nel fatto che un messaggio del Server, se contenente più righe, tali righe non sarebbero state stampate una dopo l'altra ma per stampare la successiva, il thread necessitava di un qualsiasi input da console.
   Il problema è stato risolto ristrutturando il Client, aggiungendo due ulteriori thread, uno che si occupa di ricevere messaggi dal Server (Receiver) e uno che si occupa di inviare messaggi al Server (Sender).
   In questo modo la fase di ricezione di messaggi e la fase di invio di messaggi sono state separate garantendo una stampa continua di un eventuale messaggio contenente più righe.

3. Blocco del Terminale durante l'apertura di una Sessione:
   durante l'attesa dell'apertura di una sessione, in scrittura o in lettura, l'utente non dovrebbe essere in grado di inviare ulteriori comandi.
   Per implementare questa funzionalità, si è deciso di mettere in waiting il thread Sender, finchè non sarà completata l'apertura della sessione. Per notificare l'avvenuta apertura della sessione e quindi risvegliare il thread Sender tramite il metodo notifyAll(), serve che il Server comunichi in qualche maniera l'avvenuta apertura, così che il Receiver possa procedere con lo sblocco del terminale. Per implementare questa funzionalità, si è deciso di generare un Identificatore del Client a lato

Server, che verrà passato al Client alla prima comunicazione.
Quest'ultimo verrà utilizzato per identificare i messaggi di servizio, che non devono essere stampati sul terminale, ma i quali verranno usati per identificare il tipo di operazioni da eseguire. La struttura dei messaggi di servizio sarà del tipo "IdentificatoreXXX", quindi l'identificatore seguito da un numero (XXX) che sta a indicare il tipo operazione.

4. Gestione della chiusura della connessione:
   Si è deciso di implementare la procedura di chiusura della connessione tra il Client e il Server, facendola passare sempre tramite il Server. Ovvero è sempre il Server a "ufficializzare" la chiusura, inviando un messaggio di servizio [*Identificatore 503*], che verrà ricevuto dal thread Receiver del Client, il quale procederà con la terminazione degli altri thread del Client.
   Implementandolo in questa maniera, si permette anche al Server di disconnettere un Client in caso di chiusura del Server.
   Inoltre, se fosse stato implementato direttamente tramite un controllo sulla classe Sender del Client, in caso di comando *quit* durante una sessione di lettura/scrittura, questo sarebbe stato eseguito e il Client si sarebbe disconnesso.

2.2.2 Problemi di Concorrenza

Visto che il Server gestisce varie componenti e risorse condivise da più thread, bisognava garantire l'integrità e la consistenza dei dati, tramite la mutua esclusione e evitando al contempo situazioni che portano a possibili deadlock. Si è cercato di evitare il più possibile la sequenzializzazione, ovvero dover eseguire le azioni in modo sequenziale e non in maniera parallela.

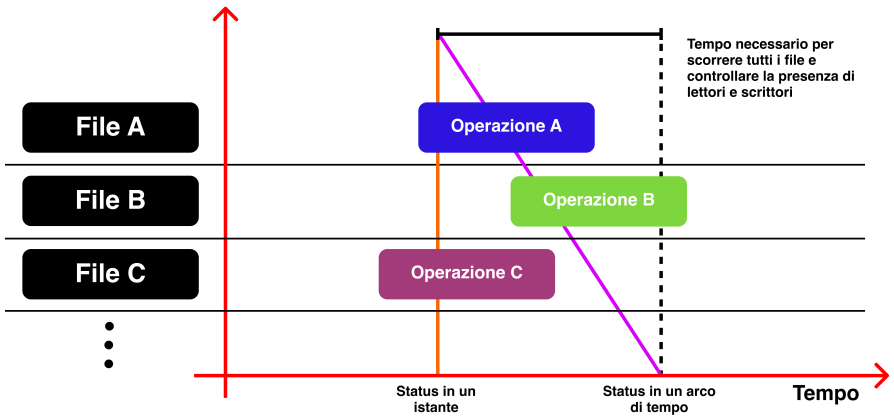
1. Readers and Writers Problem:
   Visto che stiamo lavorando con la lettura e scrittura di file, sorge il *Readers and Writers Problem*, dove bisogna gestire la lettura e scrittura concorrente su file.
   Per risolvere questo problema si è deciso di usare una classe già esistente di Java, il *ReentrantReadWriteLock*, che permette di gestire le sezioni critiche relative alla scrittura e la lettura condivisa da più thread. Questa classe fornisce due Lock, uno in lettura (readLock) e uno in scrittura (writeLock), gestendo poi internamente il problema

dei lettori e scrittori, garantendo che più thread possono acquisire il readLock, mentre solo un thread alla volta può acquisire il writeLock; ovviamente gestisce anche la mutua esclusione dei due lock, se è stato già preso uno dei due lock, l'altro tipo di lock non potrà essere preso.
Pertanto, basta richiedere il relativo Lock quando si vuole entrare nella relativa sessione (Scrittura o Lettura) e se il Lock è disponibile allora si potrà accedere alla sezione critica, mentre in caso contrario, si rimarrà in attesa fino a quando il thread non riuscirà ad acquisire il Lock.
Si è implementato nel progetto, un *ReentrantReadWriteLock* non *Fairness*, dove c'è l'eventualità di Starvation da parte di un thread che vorrebbe entrare in sessione di scrittura, ma non riesce ad acquisire il lock dovuto ad un flusso continuativo di thread che entrano in lettura.
Per risolvere questo problema basterà attivare la *Fair mode* del *ReentrantReadWriteLock* nel suo costruttore.
Per maggiori dettagli sul *ReentrantReadWriteLock* si fa riferimento alla Documentazione Ufficiale di Java.

2. Gestione dei FileManager:
   Ogni file è associato ad un rispettivo FileManager, il quale permette di aprire e chiudere le sessioni su quello specifico file. Un file deve avere solo un unico FileManager associato, quindi era necessaria una struttura dati condivisa, dove i thread potessero cercare ed eventualmente inserire i FileManager.
   Si è optato per una classe Java già esistente, il *ConcurrentHashMap* le quali operazioni son considerate "Thread Safe".
   Il vantaggio di usare il *ConcurrentHashMap* al posto di usare un normale *HashMap* ed eventualmente gestire le operazioni dentro ad una sezione critica tramite un blocco synchronized che prende come monitor l'HashMap, è il fatto che più thread possono accedere contemporaneamente a più celle del *ConcurrentHashMap*, mentre nell'altro caso, un thread che voleva accedere ad una cella, bloccava l'intero HashMap per tutti gli altri thread.
   Questo risolve solamente il problema dell'accesso concorrenziale alla struttura dati, così da non diminuire il parallelismo, però bisogna ancora garantire la consistenza delle informazioni al suo interno, ovvero la sezione critica che partiva dal controllo se il *ConcurrentHashMap* conteneva il FileManager di uno specifico file, alla creazione e inserimento di quest'ultimo nel *ConcurrentHashMap*.

3. Controllo e Inserimento nel CHM:
   Come visto nel punto precedente, il *ConcurrentHashMap* permette di garantire il parallelismo delle operazioni su di esso, ma la gestione della consistenza delle informazioni nel CHM bisognava gestirla esternamente.
   Serviva una soluzione che permettesse di rendere "atomico" o almeno "Thread Safe" tutto il blocco di controllo ed eventualmente l'inserimento nel CHM del FileManager, senza rendere seriali le operazioni eseguite.
   Si è deciso pertanto di sacrificare un pizzico di parallelismo, rendendo 3 metodi, rename, delete e getFileManager, che contenevano questa sezione critica *synchronized*, rendendo così la sezione in mutua esclusione.
   La criticità di questi 3 metodi, ed è anche il motivo per cui è stato messo *synchronized* solo a questi 3, è dovuto al fatto che il controllo della presenza del FileManager nel CHM, modifica il comportamento finale di questi 3 metodi (Eliminazione o Inserimento nel CHM).
   Facendo così ci siamo assicurati la consistenza dei dati nel *ConcurrentHashMap* per tutti i thread.

4. Accesso del CHM fuori dal DirectoryManager:
   Per i comandi *list* del Client e *info* del Server, era necessario accedere al *ConcurrentHashMap* per controllare eventuali thread in lettura o scrittura. Invece di richiamare sempre il metodo *getFileManager()* di DirectoryManager, si è deciso di accedere al *ConcurrentHashMap* in maniera diretta, questo perchè in entrambi i casi avremmo ottenuto un "immagine" dello status dei file della cartella in un arco temporale e non in un istante preciso.



Questo è dovuto al fatto che per avere l'immagine completa dello status di ogni singolo file nella cartella, dato un istante preciso, bisognerebbe bloccare l'accesso al *ConcurrentHashMap* e le operazioni di lettura e scrittura sui file per il tempo necessario a scorrere tutti i file presenti, fare i controlli e stampare il risultato.
Visto che si tratta di operazioni "Passive" e non "Attive", quindi operazioni che non vanno ad intaccare la consistenza dei dati, si è ritenuta più che accettabile questa soluzione.

5. Mutua esclusione tra i metodi di FileManager e DirectoryManager:
   Per come è strutturato il progetto, per l'apertura delle sessioni in lettura o scrittura, si usano direttamente i metodi di FileManager, senza passare dal DirectoryManager, che gestisce la rinomina ed eliminazione dei file. Questo porta ad un eventuale problema di consistenza delle informazioni, generando una sezione critica in alcuni punti di tutti questi metodi. Visto che i metodi del FileManager e quelli di DirectoryManager fanno parti di classi distinte, il fatto che i metodi di quest'ultimo siano *synchronized* non risolvono il problema creato dal BufferedWriter nel seguente esempio:

   - Un Thread A che ha già acquisito il FileManager dal metodo *getFileManager()* di DirectoryManager vuole aprire una sessione in scrittura, nel mentre un Thread B invoca il metodo *rename() o delete()* di DirectoryManager sullo stesso file.
     Entrambe le operazioni eseguono contemporaneamente, visto che non c'è una mutua esclusione; il Thread B riesce ad eseguire e

passare dopo i controlli di eventuali lettori o scrittori presenti sul FileManager dato che il Thread A ha iniziato l'apertura della sessione, ma non ha ancora acquisito il Lock in scrittura.
Se Il Thread B continua con l'esecuzione di eliminazione o rinomina del file; il thread A quando istanzierà il BufferedWriter creerà un nuovo file, violando così l'obbiettivo del metodo (Per creare un file bisogna usare il metodo *create()* di DirectoryManager).

Per affrontare questo problema, bisogna mettere in mutua esclusione il metodo delete() e rename() di DirectoryManager con l'apertura della sessione di scrittura del FileManager. Visto che i metodi fanno parte di classi diverse, non è possibile risolvere il problema usando *synchronized*. Per questo si è deciso di creare un nuovo Lock nel FileManager, che viene utilizzato per gestire la mutua esclusione tra le due classi. Così quando si arriva alla critical section, i metodi di DirectoryManager proveranno ad acquisire il Lock, se non si riesce ad acquisirlo significa che durante l'esecuzione del metodo, un thread ha iniziato la sessione in lettura o scrittura.

6. Critical Section gestite automaticamente da Java:
Ci sono alcune Critical Section che non gestiamo direttamente, possiamo trovarle nei seguenti casi:

   - Il metodo *createNewFile()* di DirectoryManager non è *synchronized*, visto che Java gestisce l'eventuale tentativo di creare un file già esistente. Si è consapevoli che le operazioni di *delete()* e *renameTo()* dei metodi della classe File, rispettivamente per eliminare e rinominare un file, non sono operazioni atomiche, ma non si ritiene al contempo che comportino un problema di integrità o consistenza, in quanto il metodo create() gestisce internamente questo problema. Se si volesse avere la certezza della corretta gestione, si potrebbe mettere il metodo *create()* di DirectoryManager come *synchronized*, al costo della riduzione di parallelismo.
   - La mutua esclusione tra il metodo *OpenWriteSession()* di FileManager e i metodi *rename()* e *delete()* di DirectoryManager, non viene implementato sul metodo *OpenReadSession()* di FileManager perchè non soffre della stessa criticità.
   Java non permette l'eliminazione o la rinomina di un file in cui c'è

un InputStream aperto, mentre non permette di aprire un InputStream (FileWriter) se il file non esiste (viene lanciato un *FileNotFoundException*).

2.3 Strumenti utilizzati per l'organizzazione

Il progetto è stato sviluppato con l'utilizzo di Eclipse e Visual Studio Code, in particolare viene utilizzato lo strumento di Debugger in VSCode per l'analisi e il testing di eventuali problemi di concorrenza (Consistenza delle informazioni tra un thread e un altro).
Lo strumento di comunicazione principale è stato Telegram, utile per la sua funzione di poter fissare e modificare messaggi. Discord e Teams per riunioni a cadenza settimanale utili per la coordinazione del lavoro e brainstorming per risoluzione condivisa delle problematiche riscontrate.
Per la condivisione del codice prodotto si è utilizzato GitLab collegato agli IDE, così da semplificare le operazioni di commit, push e pull.
Per tenere traccia del lavoro svolto è stata utilizzata la funzione di Telegram di fissare e modificare i messaggi, così da tener traccia dei maggiori problemi riscontrati e della loro risoluzione.
La documentazione viene scritta in LaTex attraverso l'utilizzo del sito OverLeaf.

3 Istruzioni per l'esecuzione

3.1 Avvio Server

3.1.1 Compilazione Server

1. Aprire il terminale;
2. Posizionarsi sulla directory TextShare/src/server;
3. Scrivere: javac *.java

3.1.2 Esecuzione Server

1. Aprire il terminale;
2. posizionarsi sulla directory TextShare/src
3. Scrivere: java server.ServerTS [path] [port]

NB: path e port sono rispettivamente l'absolute path della directory dei file su cui l'applicazione lavora e la porta su cui mettersi in ascolto.

3.2 Avvio Client

3.2.1 Compilazione Client

1. Aprire il terminale;
2. Posizionarsi sulla directory TextShare/src/client;
3. Scrivere: javac *.java

3.2.2 Esecuzione Client

1. Aprire il terminale;
2. posizionarsi sulla directory TextShare/src
3. Scrivere: java client.ClientTS [host] [port]

NB: host e port sono l'indirizzoIP e la porta del Server.