Laboratorio di Sistemi Operativi A.A. 2023-24

Nome gruppo: I Tre Romagnoli

Mail referente: maurizio.amadori4@studio.unibo.it

Componenti gruppo:

- Maurizio Amadori 0001078717
- Magrini Lorenzo 0001070628
- Alessandro Nanni 0001027757

Architettura generale

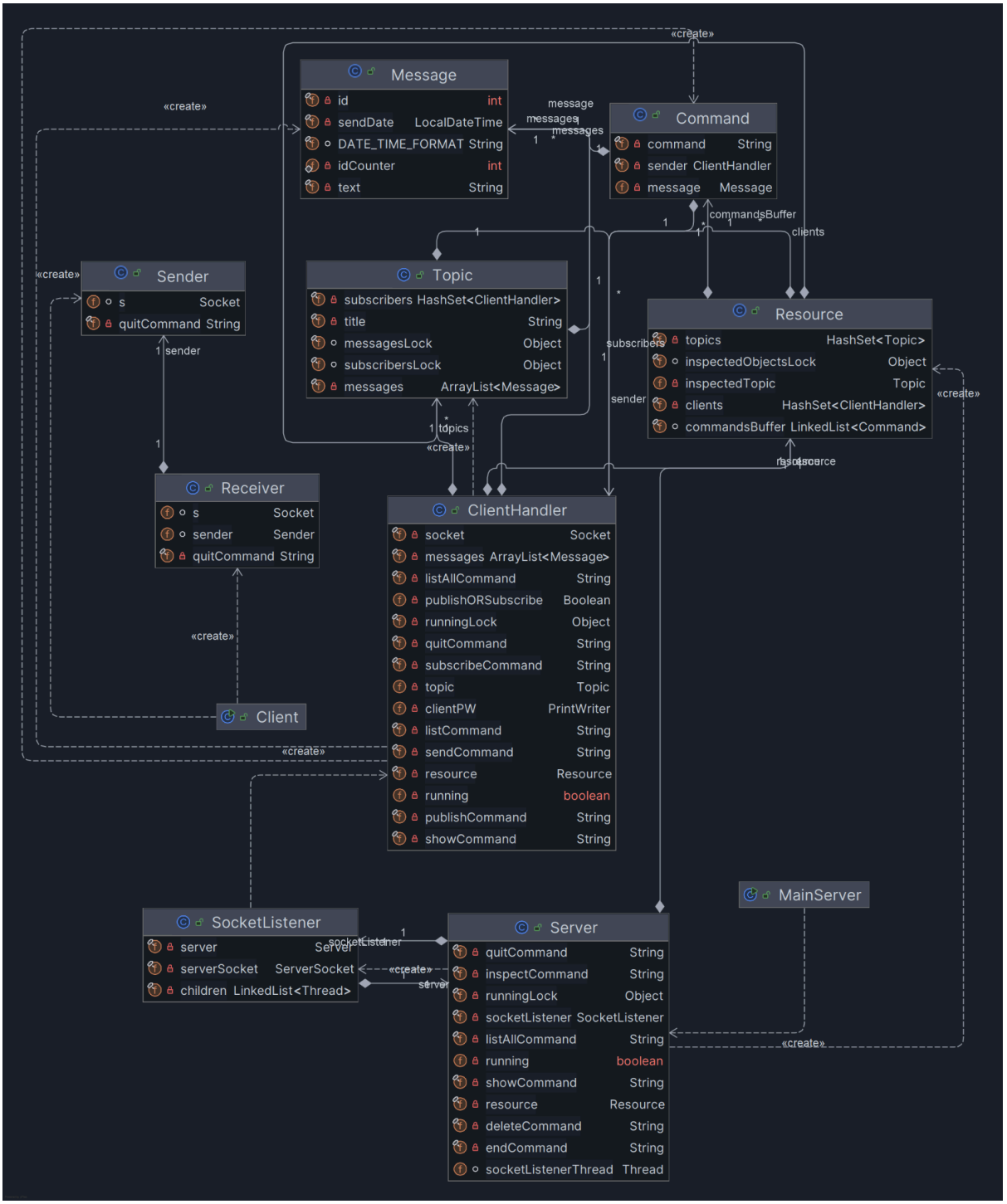
Il paradigma di alto livello implementato è di tipo client - server. Il client si preoccupa di ottenere da console i comandi e i messaggi dell'utente e di spedirli al server, andando poi a mostrare le risposte che quest'ultimo gli fornisce. Il server invece si preoccupa di ricevere messaggi e comandi, verificarne la validità, e dare una risposta consona al client.
Il server stesso offre un'interfaccia a linea di comando che permette a un utente di svolgere operazioni su quest'ultimo in maniera sicura. Senza intaccare quindi il corretto svolgimento di operazione da parte del server su richiesta del client.

Idea alla base

Abbiamo inizialmente diviso client e server ed elencato le possibili classi utili per la strutturazione del programma come: Topic, Messaggio, Comando. Riflettendo sulle differenze tra Subscriber e Publisher pensavamo inizialmente di fare due classi separate, queste avrebbero ereditato alcuni campi e metodi da una superclasse Client. Abbiamo poi deciso di fare solo una classe generale Client per non complicare la fase di stabilimento della comunicazione. Quindi abbiamo proceduto a strutturare meglio client e server, che abbiamo suddiviso nelle seguenti classi:

- Client, Sender, Receiver
- MainServer, Server, ClientHandler

In seguito all’espansione della classe Server abbiamo deciso di ristrutturare l’architettura lato server in modo da renderla più semplice, utilizzabile e manutenibile. Quindi abbiamo deciso di spostare alcuni pezzi di codice nelle classi SocketListener e Resource.
Per implementare il tutto siamo partiti dal codice fornitoci dal Tutor in quanto ci sembrava già valido e strutturato, per poi espanderlo dove necessario e integrarlo con quanto necessario per il completamento del progetto.



- Client: si occupa di istanziare una socket, connettersi al server e di gestire i thread relativi al Sender e al Receiver.
- Sender: prende in input da console messaggi e comandi e li invia al server.
- Receiver: sta in ascolto sulla socket e attende che arrivino messaggi per poi inviarli alla console.
- MainServer: prende in input le informazioni necessarie a istanziare una nuova socket e avvia un nuovo Server sulla socket specificata mettendosi poi in attesa che termini.

- Server: si occupa di fornire un interfaccia console per dialogare con il server. Inoltre mette a disposizione le proprie risorse in maniera sicura e istanzia un thread SocketListener.
- SocketListener: resta in ascolto su una socket per eventuali nuovi collegamenti da parte dei client. Una volta stabilita la connessione questa viene lasciata in gestione alla classe ClientHandler
- ClientHandler: gestisce la comunicazione con uno specifico client garantendo in questo modo una corretta e sicura interazione con le risorse della classe Server.

Funzionamento Componenti Principali

Client

La classe Client delega Sender e Receiver a gestire la comunicazione con il server. Il Sender si occupa di ottenere i messaggi dall'utente e di inviarli al server. Il Receiver riceve i messaggi dal server e li invia alla console dell’utente.

Server

La classe Server fornisce un'interfaccia, tramite i suoi metodi, per accedere alla classe Resource (contenente i dati dati scambiati da publisher e subscriber) e prendere in input i comandi tramite la console. Il Server inoltre delega una SocketListener che si metterà in ascolto per nuove connessioni dei client; stabilita la comunicazione con un client, la sua gestione passa a un ClientHandler. Il ClientHandler si occuperà di comunicare con la socket per inviare e ricevere comandi e messaggi da un client specifico, eseguire le sue richieste ed interfacciarsi con la classe Resource.

Descrizione dettagliata delle singole componenti

MainServer

La classe `MainServer` è responsabile dell'avvio di un server TCP su una specifica porta, fornendo il punto di ingresso per l'esecuzione del sistema.

Funzionalità Principali

- Inizializza e avvia un'istanza di un server (`Server`) su una porta TCP specificata.
- Permette di configurare la porta tramite argomenti della riga di comando.

Flusso di Esecuzione

1. Impostazione della Porta
    - Porta di default: `9000`.
    - Se un argomento viene passato al programma viene convertito in un numero intero. Eventuali errori di formattazione (`NumberFormatException`) vengono gestiti.

2. Avvio del Server
    - Crea un'istanza della classe `Server` con la porta specificata.
    - Lancia un thread dedicato per eseguire il server (`Thread`).
    - Imposta il nome del thread come `"server"` per facilitare l'identificazione.

3. Attesa della Terminazione
    - Utilizza `join()` per mettere il main thread in attesa finché il thread del server non termina.

Server

La classe `Server` gestisce il lato server dell’applicazione e offre un'interfaccia console per interagire con il sistema, gestendo connessioni da parte di client tramite l'uso di thread. Essa fornisce comandi per manipolare risorse e interagire con i topic. La classe inoltre gestisce la sicurezza delle risorse durante l'esecuzione e la comunicazione con i client tramite il `SocketListener`.

Funzionalità Principali

1. Gestione della Connessione:
    - Istanzia un `SocketListener` in un thread separato per accettare nuove connessioni dai client (`Server.java`, righe 28-30).
2. Comandi del Server:
    - Implementa i comandi per interagire con il server da console, come "quit", "show", "inspect", "delete", ecc.
3. Interazione con le Risorse:
    - Gestisce risorse condivise in modo sicuro (topic e messaggi) attraverso un oggetto `Resource`.

Variabili d'Istanza

Variabile	Tipo	Descrizione
`running`	`boolean`	Indica se il server è in esecuzione o è stato fermato.
`runningLock`	`Object`	Lock per la sincronizzazione degli accessi alla variabile running
`socketListener`	`SocketListener`	Gestisce le connessioni in entrata dai client.
`resource`	`Resource`	Contiene le risorse (topic, messaggi) e i metodi per manipolarle.
`socketListenerThread`	`Thread`	Thread che esegue il `SocketListener`.
`outMesCommandWithOutParameters`	`String`	Messaggio di errore per comandi che non accettano parametri.

Metodi Principali

Metodo	Descrizione
`run()`	Metodo eseguito dal thread principale del server. Gestisce i comandi da console e le risorse.

Metodo	Descrizione
`quit()`	Interrompe il server, chiudendo il SocketListener e le connessioni client attive.
`inspect()`	Imposta un topic in modalità di ispezione o segnala un errore se il topic non esiste.
`end()`	Termina la fase di ispezione di un topic.
`delete()`	Elimina un messaggio dal topic ispezionato, chiamato solo durante la fase di ispezione.
`show()`	Mostra tutti i topic disponibili sul server.
`listAll()`	Elenca tutti i messaggi nel topic ispezionato, chiamato solo durante la fase di ispezione.

Dettagli del Codice

1. Gestione Comandi via Console:
    - Il server riceve comandi da console attraverso il metodo `run()`, che esegue comandi in base alla modalità (ispezionando o meno un topic) (`Server.java`, righe 50-57).
2. Sicurezza delle Risorse:
    - L'accesso alle risorse (topic e messaggi) è sincronizzato tramite il lock `resource`, per evitare condizioni di competizione tra i thread.
3. Chiusura Sicura:
    - Il comando `quit()` gestisce una chiusura sicura del server, interrompendo il `SocketListener` e liberando le risorse (`Server.java`, righe 135-138).

SocketListener

La classe `SocketListener` è in ascolto delle connessioni in entrata sulla porta del server e crea nuovi thread per gestire la comunicazione con ogni client connesso.

Funzionalità Principali

1. Creazione e ascolto su una ServerSocket

La classe viene inizializzata con un oggetto `ServerSocket` su una porta specifica (passata al costruttore). Utilizza `serverSocket.accept()` per mettersi in ascolto di nuove connessioni in ingresso dai client. Quando un client tenta di connettersi, l'esecuzione del metodo è bloccata fino a che una connessione non viene stabilita (`SocketListener.java`, riga 36).

2. Creazione di nuovi thread per la gestione dei client

Ogni volta che una nuova connessione viene accettata, la classe crea un nuovo oggetto `ClientHandler`, responsabile della gestione della comunicazione con il client. Un nuovo thread viene creato per eseguire il `ClientHandler` e avviato subito dopo la sua creazione, permettendo la gestione contemporanea di più client (`SocketListener.java`, righe 41-45).

3. Gestione delle interruzioni e chiusura delle connessioni

Se il thread `SocketListener` viene interrotto (ad esempio, se il server viene arrestato), il ciclo di ascolto si interrompe, e nuove connessioni non saranno più accettate.
Una volta che il server viene fermato o se viene lanciata una `SocketException`, la connessione corrente viene chiusa tramite `clientSocket.close()`.

4. Gestione delle eccezioni

Durante la gestione delle connessioni, se si verifica una `IOException` (ad esempio, un errore di rete), viene lanciata una `RuntimeException` per gestire l'errore.
Se il server viene arrestato mentre è in ascolto, una `SocketException` può essere generata durante la gestione della connessione. In questo caso, se il server non è più in esecuzione (`server.isRunning()` è false), il ciclo di ascolto si interrompe, e la connessione viene chiusa.

5. Sincronizzazione e gestione concorrente

La classe utilizza la sincronizzazione attraverso il metodo `synchronized` per proteggere la creazione di nuovi thread e l'accesso a risorse condivise, come la `ServerSocket`, specialmente durante l'interruzione del thread di ascolto. La sincronizzazione viene utilizzata per assicurarsi che il thread di ascolto non accetti nuove connessioni mentre il server sta fermando o interrompendo il ciclo.

Command

La classe `Command` rappresenta un pattern per memorizzare ed eseguire i comandi che il server può ricevere da un client, ma non può eseguire perché in ispezione.

Funzionalità Principali

1. A cosa serve?

Consente di incapsulare i comandi inviati dai client, associandoli a:

- Il nome del comando (`command`).
- Il client che lo ha inviato (`sender`).
- Un eventuale contenuto aggiuntivo, come un messaggio (`message`).

Quando il server è in fase di ispezione e un client prova a inviare un comando a esso, viene creato un nuovo oggetto Command che contiene il comando. Eventuali parametri e il ClientHandler che lo ha inviato. Il nuovo comando viene poi aggiunto a una lista di comandi in attesa nelle risorse del server.

2. Costruttori

- `Command(String command, Message message, ClientHandler sender)`: Utilizzato per comandi che richiedono un messaggio (`send`).
- `Command(String command, ClientHandler sender)`: Usato per comandi che non richiedono un messaggio (es. `list`, `listall`).

3. Esecuzione del Comando

Il metodo `execute()` gestisce l'esecuzione del comando incapsulato (tra `list`, `listall` e `send`) (`Message.java`, righe 29-35)

Resource

La classe `Resource` gestisce e memorizza i dati relativi alle comunicazioni sul server, garantendo un accesso sincronizzato. Include funzionalità per la gestione di topic, client connessi e comandi durante la fase di ispezione. I ClientHandler dispongono di un riferimento alla variabile resource del server.

Funzionalità Principali

1. Gestione dei Topic:
    - Aggiunta e recupero dei topic dal server.
    - Elenco e visualizzazione di tutti i topic presenti.
2. Ispezione dei Topic:
    - Impostazione del topic in ispezione.
    - Lettura e rimozione dei messaggi relativi al topic ispezionato.
3. Gestione Client:

    - Aggiunta e rimozione dei client connessi al server.
    - Sconnessione di tutti i client.
4. Buffer dei Comandi:
    - Coda che tiene traccia dei comandi ricevuti durante l'ispezione, per poi eseguirli nell'ordine in cui sono stati inviati (`Resource.java`, righe 86-88) quando il server termina l'ispezione.

ClientHandler

La classe `ClientHandler` gestisce la comunicazione tra il server e un client specifico. Permette di eseguire comandi relativi a topic, pubblicazione e iscrizione, gestisce invio, visualizzazione e rimozione dei messaggi. Più in generale gestisce la ricezione di tutti i comandi previsti per i client, facendo una distinzione al suo interno tra publisher e subscriber.

Proprietà principali

- `socket`: Connessione di rete con il client.
- `resource`: Risorse del server.
- `publishORSubscribe`: Indica se il client è un publisher (false) o un subscriber (true).
- `topic`: Topic a cui il client è iscritto o sta pubblicando.
- `messages`: Elenco dei messaggi inviati dal client.
- `clientPW`: PrintWriter per inviare risposte al client.
- `running`: Flag che indica se la connessione è attiva.

Funzioni Principali

Esecuzione dei comandi

- `publish(String parameter)`: Registra il client come publisher su un topic.
- `subscribe(String parameter)`: Registra il client come subscriber a un topic.
- `show()`: Mostra tutti i topic disponibili.
- `send(String text)`: Invia un messaggio su un topic (solo per publisher).
- `list()`: Elenca i messaggi inviati dal publisher su un topic.
- `listAll()`: Elenca tutti i messaggi scambiati su un topic.
- `quit()`: Disconnette il client dal server.

Metodi di supporto

- `manageCommands(String command, String parameter)`: Gestisce l'elaborazione dei comandi ricevuti dal client.

- `isPublisherOrSubscribeCommand(String command)`: Verifica se un comando può essere eseguito dal publisher.
- `sendExecute(Message message)`: Invia un messaggio a tutti i subscriber del topic.
- `forward(String text)`: Invia una stringa al client.
- `delMessage(Topic t, int id)`: Rimuove un messaggio dalla copia locale del topic se il parametro `t` è uguale al topic del ClientHandler (`ClientHandler.java`, righe 410-412).
- `releaseResources()`: Rilascia tutte le risorse associate al client quando la connessione viene interrotta.

Gestione della connessione

- `run()`: Loop principale che legge i comandi dal client e li elabora.
- `closeSocket()`: Chiude la connessione con il client.
- `quit()`: Termina la connessione del client con il server.
- `isRunning()`: Verifica se il client è ancora connesso.

Sincronizzazione

La classe utilizza sincronizzazione per operazioni su variabili condivise come `messages`, `running`, e `topic` per evitare problemi di accesso condiviso delle risorse.

Client

La classe `Client` implementa un client TCP per connettersi a un server, gestendo la comunicazione bidirezionale (ricezione e invio) tramite thread separati.

Funzionalità principali

1. Connessione al server:
    - Usa `Socket` per connettersi a un server all'indirizzo (default: `127.0.0.1`) e porta (default: `9000`), modificabili tramite argomenti della riga di comando.
2. Gestione della comunicazione:
    - Vengono avviati due thread, `Sender` che invia messaggi al server, e `Receiver`, che riceve messaggi dal server e gestisce la chiusura.
3. Gestione degli errori:
    - `IOException`: Errore di connessione.
    - `InterruptedException`: Interruzione dei thread.

Sender

La classe `Sender` è responsabile per la lettura dei messaggi da console e l'invio di comandi o messaggi al server tramite una connessione di socket. Gestisce l'invio di dati dal client al server ed è implementata come un thread, permettendo di funzionare in modo asincrono.

Funzionalità Principali

1. Lettura dei Messaggi da Console:
    - La classe attende l'input dell'utente da console.
2. Invio al Server:
    - I messaggi o comandi letti vengono inviati al server tramite la socket associata.
3. Interruzione della Connessione:
    - Se viene inviato il comando "quit", la connessione viene terminata.

Receiver

La classe `Receiver` ascolta i messaggi inviati dal server attraverso una socket e li presenta sulla console del client. Questa classe estende `Thread`, consentendo di gestire l'ascolto in un thread separato.

Funzionalità Principali

1. Ascolto Continuo: Il `Receiver` rimane in ascolto dei messaggi inviati dal server tramite la socket specificata. Quando viene ricevuto il comando `quit`, termina il ciclo di ascolto e notifica la disconnessione.
2. Gestione delle Disconnessioni: In caso di disconnessione del server (volontaria o improvvisa), gestisce gli eventi e interrompe il thread associato al `Sender`.
3. metodo run: viene creato un BufferedReader, che legge i dati provenienti dal server tramite la socket. Un `InputStreamReader` converte l'input byte-stream del socket in una stringa. La connessione viene chiusa quando il `readLine()` del BufferedReader è `null`. Oppure se viene ricevuto il comando quit dal server, che interrompe il ciclo con un `break` (`Receiver.java`, righe 27-35).
   Alla fine del ciclo, nel blocco `finally`, viene interrotto anche il thread del Sender.

Topic

La classe `Topic` rappresenta un argomento di discussione col quale i client possono inviare e ricevere messaggi. Gestisce la lista di messaggi e degli iscritti (subscribers) e fornisce i metodi per pubblicare, rimuovere messaggi e gestire gli iscritti.

Proprietà principali

- `title`: Titolo del topic.
- `messages`: Lista di messaggi scambiati sul topic.
- `subscribers`: Set di client iscritti al topic.

Metodi principali

Gestione dei messaggi

- `getStringMessages()`: Restituisce l'elenco dei messaggi scambiati.
- `addMessage(Message message)`: Aggiunge un messaggio al topic.
- `removeMessage(int id)`: Rimuove un messaggio tramite il suo ID.

Gestione degli iscritti (Subscribers)

- `addSubscriber(ClientHandler subscriber)`: Aggiunge un client al topic.
- `forwardToAll(Message message)`: Invia un messaggio a tutti gli iscritti al topic.
- `removeSubscriber(ClientHandler subscriber)`: Rimuove un subscriber dal topic.

Uguaglianza tra topic

- `equals(Object obj)`: Sovrascrive il metodo per vedere se due topic sono uguali in base al loro titolo (`Topic.java`, righe 114-119).

Sincronizzazione

Utilizza oggetti di sincronizzazione (`messagesLock`, `subscribersLock`) per garantire l'atomicità delle operazioni che coinvolgono messaggi e iscritti.

Message

La classe `Message` rappresenta un messaggio testuale utilizzato nella comunicazione tra un publisher e un subscriber. La classe consente di memorizzare e gestire informazioni relative al messaggio.

Funzionalità Principali

1. Creazione del Messaggio

- Ogni messaggio riceve un identificatore univoco incrementale (`id`) generato automaticamente in maniera sincronizzata.
- Include il contenuto del messaggio (`text`).

- Memorizza la data e l'ora di invio (`sendDate`) utilizzando la libreria `LocalDateTime`.

2. Formato Data e Ora
   - La data e l'ora sono formattate secondo il pattern `dd/MM/yyyy - kk:mm:ss`

Suddivisione del lavoro

Magrini

- Classi: Message e Client
- Implementazione della struttura dei messaggi di output
- Implementazione di una notifica mandata client quando viene eseguito un loro comando messo in "pausa" per via dell'ispezione sul topic
- Logica iniziale di ispezione del Topic
- Problema di concorrenza nella creazione di oggetti di tipo Message sull'ID univoco
- Relazione

Nanni

- Classi: Server, ClientHandler e SocketListener
- Logiche di comunicazione del paradigma client - server
- Relazione
- Grafico classi

Amadori

- Classi: Client, Sender, Receiver e Topic
- Logiche di comunicazione del paradigma client - server
- Logiche di gestione dei dati e delle risorse
- Gestione dei thread e della chiusura della comunicazione
- Sincronizzazione
- Relazione

Problemi e ostacoli

Chiusura della comunicazione

Inizialmente in alcune classi veniva utilizzato uno `Scanner` per leggere i messaggi da tastiera; questo era un problema dato che lo scanner è bloccante. Il problema si manifestava col non riuscire a chiudere un thread senza l'interazione con l'utente che lo stava utilizzando e con la sovrapposizione di comandi diversi in console. Questo problema inizialmente l'abbiamo risolto utilizzando un BufferedReader e

verificando se c'erano dati pronti per essere letti nello stream sottostante in questo modo la lettura dell'input non era bloccante. Successivamente testando il programma da console ci siamo resi conto che facendo in questo modo andavamo a bloccare la visualizzazione dell'input a console che restava invisibile all'utente finchè non veniva premuto invio. Per questo abbiamo deciso di tornare sui nostri passi ed affrontare la cosa in maniera differente, accettando che il client sarebbe rimasto aperto finchè non si preme invio a console

Strutture dati da utilizzare e relativa gestione degli accessi e della sincronizzazione

Ci siamo posti il quesito: "Quali strutture dati è più corretto utilizzare? E come?". Abbiamo valutato se utilizzare strutture dati sincrone (ad esempio `ConcurrentHashMap`), ma abbiamo deciso di non utilizzarle e spendere più tempo nella sincronizzazione del progetto. Riflettendo su come memorizzare i dati abbiamo deciso di mantenere più strutture (e non tenere tutto assieme) rendendo più veloce la lettura dei dati a discapito dell'eliminazione. In sostanza la classe Resource mantiene tutti i dati:

- `HashSet<Topic>`: lista di tutti i topic, ogni Topic ha una lista di messaggi un titolo e una lista di client iscritti a quel topic
- `HashSet<Client>`: lista di tutti i client collegati, ogni ClientHandler ha una sua lista di messaggi che il client ha inviato. Il topic su cui i messaggi sono stati inviati è un flag per identificare se quel client è un publisher o un subscriber. Avendo per ogni ClientHandler una struttura dati che memorizza i messaggi inviati dal publisher associato non è necessario richiedere tali messaggi all'arraylist presente sul server.
- `Queue<Command>`: lista di tutti i comandi in sospeso

Modalità di gestione della sincronizzazione

La sincronizzazione è stata gestita in prevalenza con metodi `synchronized` o blocchi `synchronized`, abbiamo valutato se interpellare metodi `wait`, `notify`, semafori o altro ma ci è sembrato inutile oltre che più complesso.

- **Il meccanismo dei comandi sospesi** viene implementato acquisendo il lock sull'istanza della classe Resources e switchando la variabile booleana inspectedLock. In questo modo è sempre possibile sapere se si è in una fase di ispezione o no (Se inspectedLock è null non si è in ispezione altrimenti si). Non potrà quindi mai accadere che il thread associato alla classe ClientHandler svolga comandi indesiderati durante la fase di ispezione o mentre questa sta iniziando/terminando. Perchè se questa ha il lock per eseguire tali operazioni non lo avrà il thread associato al server e viceversa.
- **La classe Resources** gestisce l'accesso alle strutture dati e a variabili condivise da più thread. Questa sincronizzazione avviene sempre tramite il costrutto `synchronized`, e in modo tale da ottenere sempre un lock il più

specifico possibile e il più breve possibile. In questo modo si cerca di limitare l'acquisizione prolungata di una risorsa da parte di un thread. Per questo si utilizzano variabili specifiche per l'acquisizione del lock e non si usa il lock implicito della classe Resources. La classe Resources esiste con lo scopo principale di separare la gestione dei dati dalla gestione delle logiche del programma e di garantire l'accesso ai dati in maniera sincronizzata e sicura, oltre che per rendere il codice più leggibile e manutenibile.

- **La classe Topic** gestisce l'accesso alle proprie risorse partendo dalle stesse idee della classe Resources. Sono presenti quindi diversi oggetti che fungono da lock, questi servono per gestire gli accessi alle relative risorse (messages e subscribers) e garantire la massima efficienza tramite l'indipendenza.
- **La classe ClientHandler** oltre a quanto già detto prima sul meccanismo dei comandi sospesi, dispone del metodo `forward(String text)` sincronizzato in modo tale che non ci siano race condition quando thread diversi provano a inviare messaggi allo stesso client. Per esempio questo può accadere se più publisher scrivono sullo stesso topic nello stesso momento. Il metodo `quit()` è `synchronized` per evitare che Server e ClientHandler, e quindi client, quittino nello stesso momento. Infine è presente un metodo `isRunning()` per verificare in maniera sincronizzata se il thread sta venendo chiuso e alcuni blocchi `synchronized` per gestire il cambio della variabile `running`.
- **La classe Server**, oltre a quanto già detto prima sul meccanismo dei comandi sospesi, ha un metodo `isRunning()` per verificare in maniera sincrona se il thread sta venendo chiuso. Questo metodo unito ad alcuni blocchi `synchronized` per gestire il cambio della variabile running permettono un accesso sicuro alla variabile.
- **La classe Socket Listener** contiene un metodo sincrono per la chiusura della socket e un blocco `synchronized`. Tramite questi è possibile garantire che quando il thread del Server richiama il metodo `close()` non vengano accettate nuove comunicazioni o se succede vengano poi immediatamente chiuse.

Comandi che non richiedono parametri vengono eseguiti anche se vengono forniti dei parametri

Per come abbiamo implementato il parsing dei comandi sia lato client che server, abbiamo notato che i comandi che non richiedono parametri venivano eseguiti anche se questi erano forniti. Per risolvere il problema abbiamo controllato che se il parametro è `null` (caso server) o una stringa vuota (caso client), la funzione associata al comando viene eseguita, altrimenti viene inviato un messaggio d'errore.

Strumenti utilizzati per l'organizzazione

- Editor: IntelliJ Community/Ultimate e Visual Studio Code
- Repository codice: GitHub
- Comunicazione: Discord e Whatsapp

Compilare e utilizzare l'applicazione

1. Estrarre la cartella zip, creare (almeno) due terminali: uno per il server e gli altri per i client.
2. Nel primo terminale, compilare ed eseguire la classe MainServer con
   `javac MainServer.java`
   `java MainServer <porta>`

   ```
   PS C:\Users\Ale\Documents\GitHub\LABSO_iTreRomagnoli\src> javac MainServer.java
   PS C:\Users\Ale\Documents\GitHub\LABSO_iTreRomagnoli\src> java MainServer 101b
   Errore: inserisci un numero di porta per la socket.
   PS C:\Users\Ale\Documents\GitHub\LABSO_iTreRomagnoli\src> java MainServer 9000
   Server avviato

   > Inserisci comando
   ```

   Se non viene specificata la porta, il server verrà aperto sulla porta `9000`.
3. Negli altri terminali, compilare ed eseguire la classe Client con
   `javac Client.java`
   `java Client <host> <porta>`

   ```
   PS C:\Users\Ale\Documents\GitHub\LABSO_iTreRomagnoli\src> javac Client.java
   PS C:\Users\Ale\Documents\GitHub\LABSO_iTreRomagnoli\src> java Client 127.0.01 9000
   Connesso al server
   ```

   Omettendo la porta o sia porta che host, il client verrà avviato con host `127.0.01` e porta `9000`.
   Esempio di esecuzione client quando il server non è raggiungibile (porta sbagliata o server non avviato).

   ```
   PS C:\Users\Ale\Documents\GitHub\LABSO_iTreRomagnoli\src> java Client 127.0.01 8000
   Errore: Il server è irraggiungibile, riprovare cambiando host e numero di porta.
   ```
4. Fase di chiusura della comunicazione
   - se un client viene interrotto tramite quit o perchè il thread viene brutalmente chiuso il server gestisce la cosa rilasciando le risorse associate a quel client
   - se il server viene interrotto tramite quit o perchè il thread viene brutalmente chiuso il client gestirà la cosa chiudendosi

Esempi di esecuzione e output

`@` indica chi sta eseguendo il comando, `>` indica il testo inviato e `<` indica il testo ricevuto. Per motivi di chiarezza sono stati esclusi i prompt del server `> Inserisci`

`comando` e `> Inserisci comando (ispezionando topic "<topic>")`, che appaiono all'avvio del server e in seguito all'invio di qualsiasi comando.

Registrazione di client come publisher

```
@ Client
> publish cibo
< Registrato come publisher al topic cibo
```

Client invia comando invalidato per via della presenza di un parametro, invio comando non conosciuto

```
@ Client(publisher)
> show cibo
< Questo comando non accetta parametri
> get cibo
< Comando non riconosciuto: get
```

Esecuzione del comando subscribe per un publisher

```
@ Client(publisher)
> subscribe cibo
< Non puoi più eseguire questo comando
```

Invio di messaggio

```
@ Client(publisher)
> send
< Non puoi inviare un messaggio vuoto
> send carbonara
< Inviato messaggio "carbonara"
```

Server invia comando invalidato per via della presenza di un parametro e ispezione di topic non esistente

```
@ Server
> show cibo
< Questo comando non accetta parametri
> inspect sport
< Il topic sport non esiste
```

Ispezione di topic esistente

```
@ Server
> inspect cibo
< Ispezionando il topic: cibo
```

Elenco di messaggi inviati sul topic

```
@ Server
> listall
< Sono stati inviati 1 messaggi in questo topic.
  Messaggi Inviati:
     - ID: 1
       Testo: carbonara
       Data: 26/10/2024 - 11:56:45
```

Client invia un comando quando il server è in fase di ispezione

```
@ Client(publisher)
> send carbonara
< Messaggio "carbonara" in attesa. Il server è in fase di
ispezione.
```

Comando non riconosciuto in fase di ispezione

```
@ Server
> quit
< Comando non riconosciuto: quit
```

Fine ispezione topic, sender viene notificato dell'invio del suo comando

```
@ Server
> end
< Fine ispezione del topic cibo.
@ Sender(publisher)
< Il tuo messaggio "carbonara" è stato inviato
```

Ricezione messaggi con contenuto uguale e ricezione messaggi terminata la precedente fase di ispezione

```
@ Server
> inspect cibo
< Ispezionando il topic: cibo
> listall
< Sono stati inviati 2 messaggi in questo topic.
  Messaggi Inviati:
     - ID: 1
       Testo: carbonara
       Data: 26/10/2024 - 11:56:45
```

```
      - ID: 2
        Testo: carbonara
        Data: 26/10/2024 - 12:07:06
```

Cancellazione messaggio con id valido e id invalido

```
@ Server
> delete 2
< Messaggio eliminato
> delete 2
< Messaggio con id 2 non esiste
> end
< Fine ispezione del topic cibo.
```

Esecuzione non consentita di comandi esclusivi ai publisher o ai subscriber

```
@ Client
> send sport
< Devi essere registrato come publisher per inviare questo
comando
> listall
< Devi registrarti come publisher o subscriber prima di poter
eseguire questo comando
> list
< Devi essere registrato come publisher per inviare questo
comando
```

Iscrizione a topic non esistente, esistente e richiesta elenco topic presenti

```
@ Client
> subscribe cinema
< Il topic inserito non esiste
> show
< Topic presenti:
        - cibo
> subscribe cibo
< Registrato come subscriber al topic cibo
```

Elenco dei messaggi sul topic a cui il client è iscritto

```
@ Client(publisher)
> listall
< Messaggi:
```

```
    - ID: 1
      Testo: carbonara
      Data: 26/10/2024 - 12:26:21
```

Invio messaggio su un topic al quale il subscriber è iscritto

```
@ Client(publisher)
> send matriciana
< Inviato messaggio "matriciana"
@ Client(subscriber)
> Nuovo messaggio pubblicato
  ID: 1
  Testo: matriciana
  Data: 26/10/2024 - 12:55:39
```

Arresto client e scollegamento dal server

```
@ Client(publisher)
> quit
< Connessione al server terminata
  Client Chiuso
```

Arresto Server e scollegamento dei client da esso

```
@ Server
> quit
< Interruzione dei client connessi:
  Interruzione client ClientHandler@72e856ca
  Server arrestato.
@ Client
< Connessione al server terminata
> [premi invio]
< Client chiuso
```

Disconnessione client dovuta ad arresto forzato del Server (chiusura terminale)

```
@ Client
< Il server si è impropriamente disconnesso
```

Arresto forzato del Client (chiusura terminale)

```
@ Server
< Il client ClientHandler@60b3cd07 si è impropriamente
disconnesso
```

```
Rilascio tutte le risorse associate al client...
```