

PUBLISH / SUBSCRIBE

LABORATORIO DI SISTEMI OPERATIVI
A.A. 2023-24

30/09/2024

Nome del Gruppo :

ACD2024

JACOB ANGELES
XXXXXXX

ANDREA CROCI
XXXXXXX

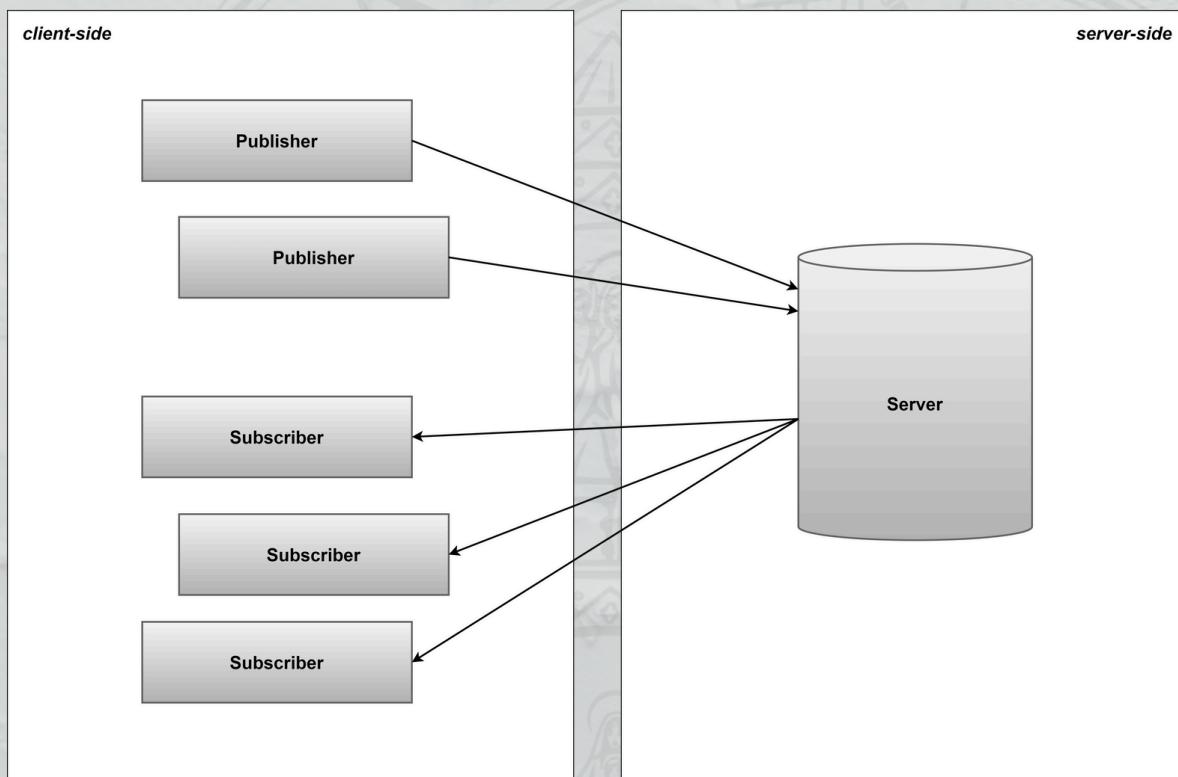
FRANCESCA D'ORIANO
XXXXXXX

Email del referente:
jacob.angeles@studio.unibo.it

IL PROGETTO

Il progetto è basato su un'architettura **client-server** in cui i client comunicano con il server tramite **socket**. Il server, in generale, si occupa di gestire le connessioni, ricevere ed elaborare le richieste, e coordinare la gestione dei topic. Un aspetto chiave di questa architettura è la **concorrenza**: i client e il server sono **multithreaded** e ogni client connesso è gestito da un thread dedicato.

Questa architettura permette ai client di eseguire operazioni come la pubblicazione e la sottoscrizione a topic in modo indipendente e simultaneo. Il server centralizza la gestione dei topic utilizzando una **risorsa condivisa** che coordina la pubblicazione dei messaggi e le sottoscrizioni dei client.



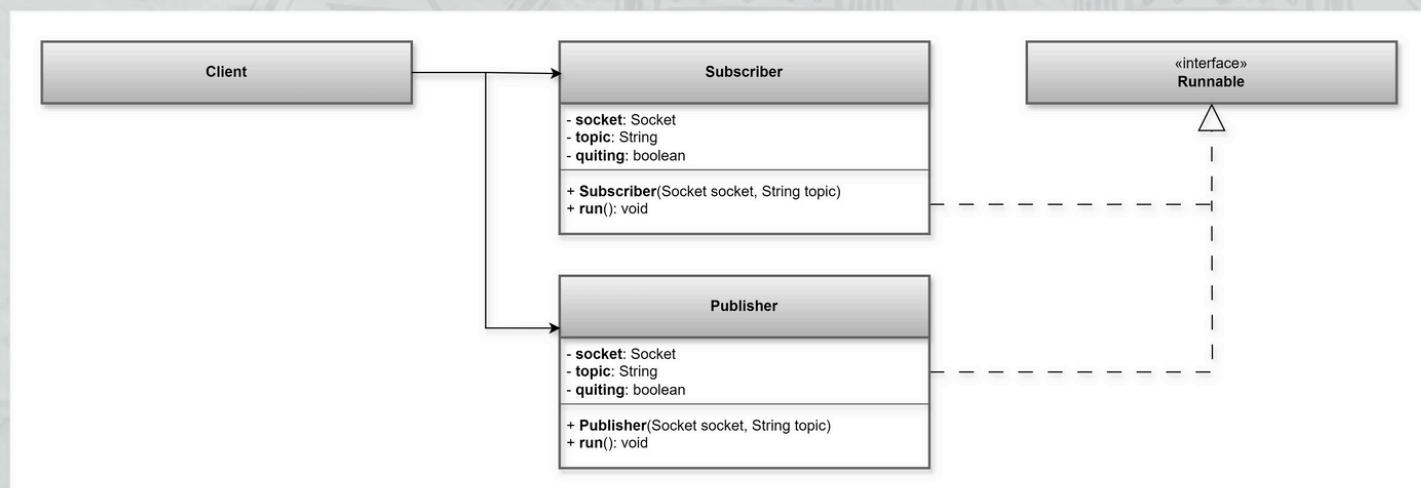
COMPONENTI

Client-side

Gli oggetti client-side gestiscono le operazioni dei client, come inviare richieste al server, ricevere messaggi e gestire la pubblicazione e sottoscrizione ai topic.

Le classi principali sono:

- **Client.java**: Il punto di avvio per un client. Si connette al server tramite il socket e delega le operazioni ad un thread separato per gestire i comandi come publish, subscribe, e show.
- **Publisher.java**: Gestisce l'invio di messaggi su un topic specifico. Viene eseguito su un thread separato e consente richiedere la pubblicazione dei messaggi o elencare quelli già inviati.
- **Subscriber.java**: Consente al client di iscriversi a un topic per ricevere messaggi pubblicati e consente richiedere l'elencare quelli già inviati.



COMPONENTI

Server-side

Gli oggetti server-side gestiscono tutte le operazioni legate alla ricezione, elaborazione e gestione delle richieste provenienti dai client. Il server è multi-threaded e ogni client connesso ha un thread dedicato per la gestione delle comunicazioni.

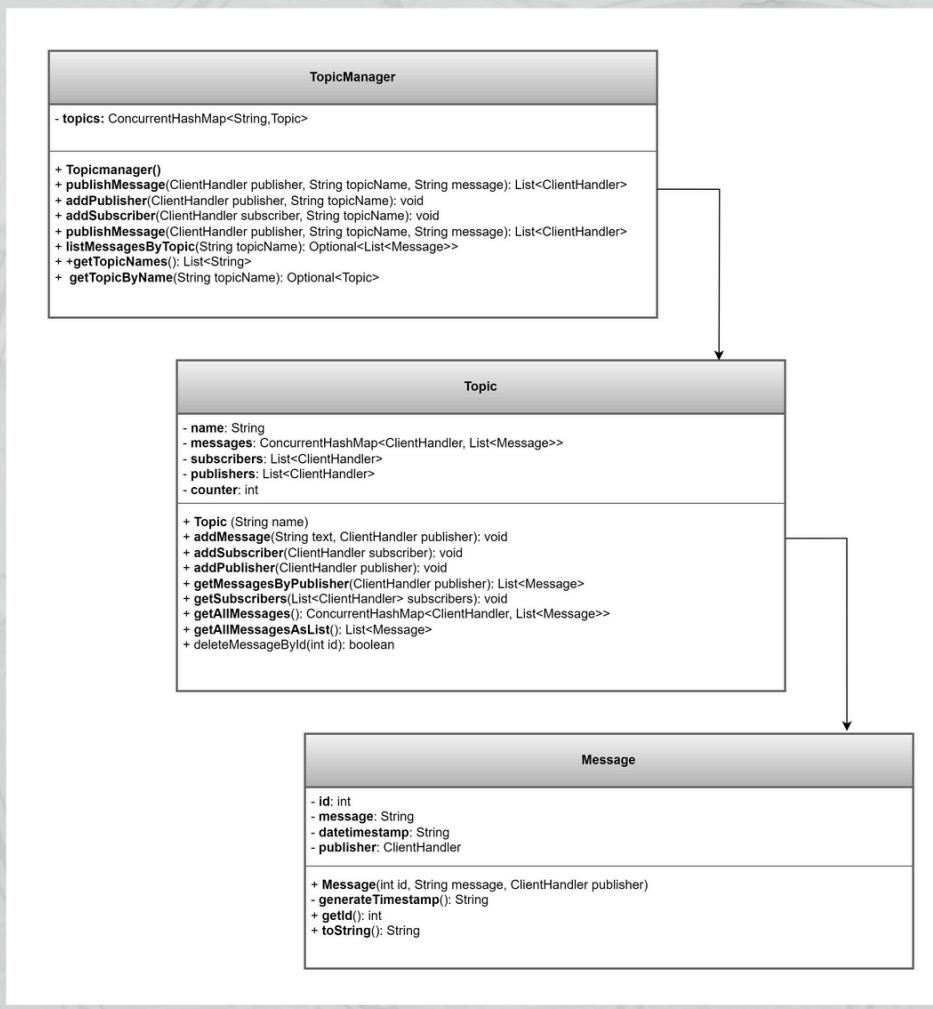
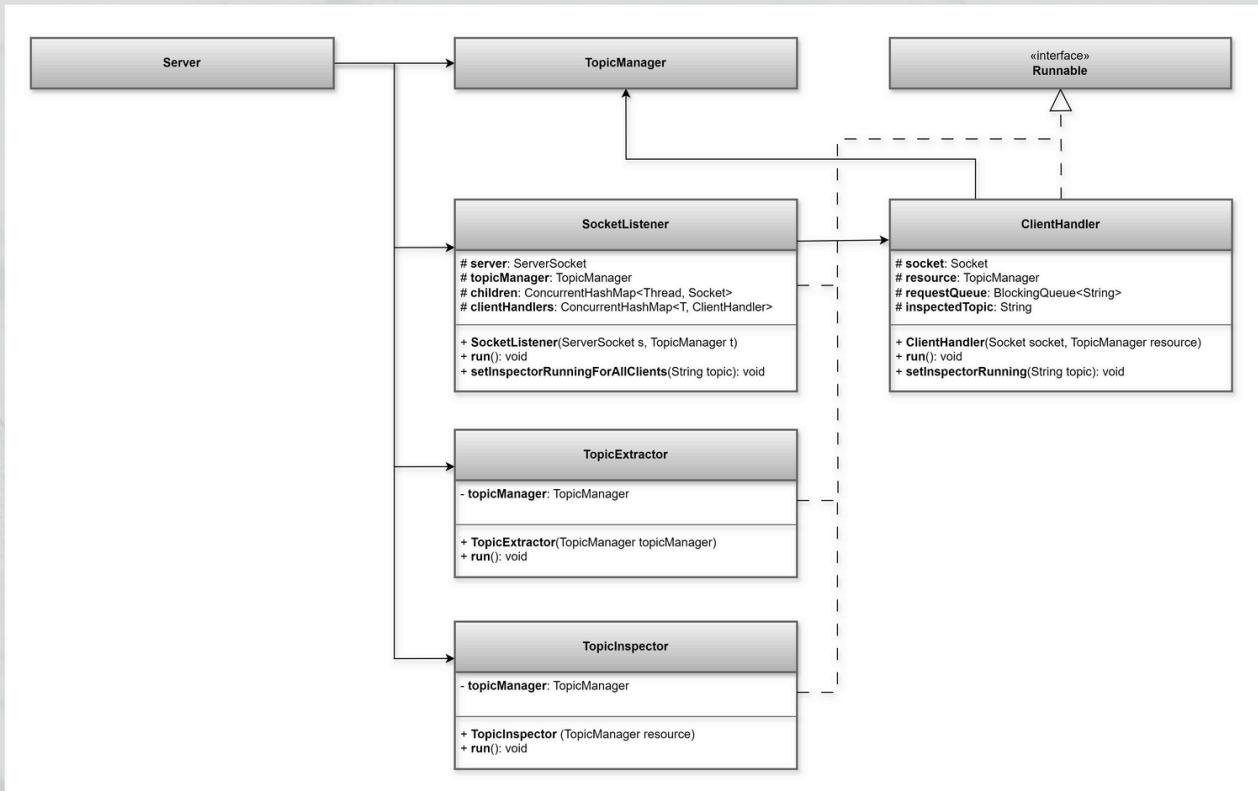
Le classi principali sono:

- **Server.java**: Il punto di avvio per il server. Crea un thread separato e delega l'ascolto delle connessioni in arrivo al socketlistener. Gestisce i comandi dell'utente per ispezionare e visualizzare i topic attraverso thread dedicati.
- **SocketListener.java**: Ascolta le connessioni in arrivo dai client. Ogni volta che un client si connette, viene creato un nuovo thread per gestirlo, delegando la gestione a un ClientHandler.
- **ClientHandler.java**: Gestisce la comunicazione con un client specifico. Esegue comandi come publish, subscribe, list, e quit. Inoltre, gestisce la sincronizzazione quando viene eseguita l'ispezione dei topic.
- **TopicInspector.java**: Esegue l'ispezione di un topic in modalità interattiva. Consente all'utente di vedere i messaggi pubblicati e di eliminare quelli selezionati.
- **TopicExtractor.java**: Estrae e mostra l'elenco di tutti i topic gestiti dal server.

Le classi che rappresentano la risorsa sono:

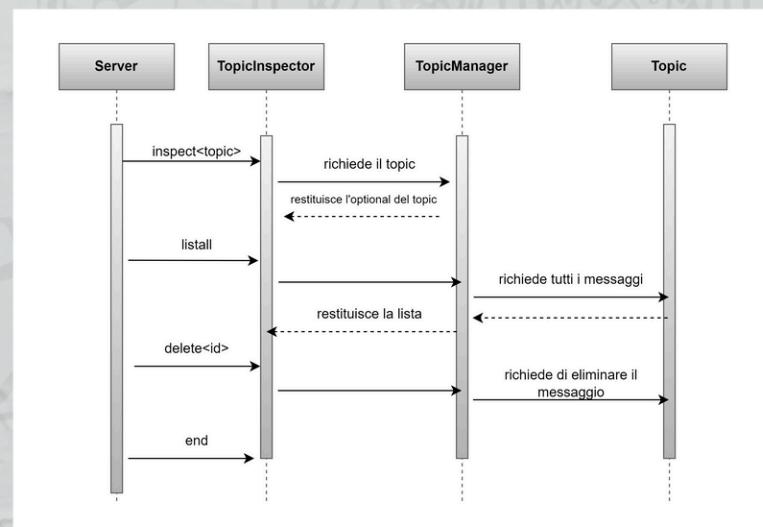
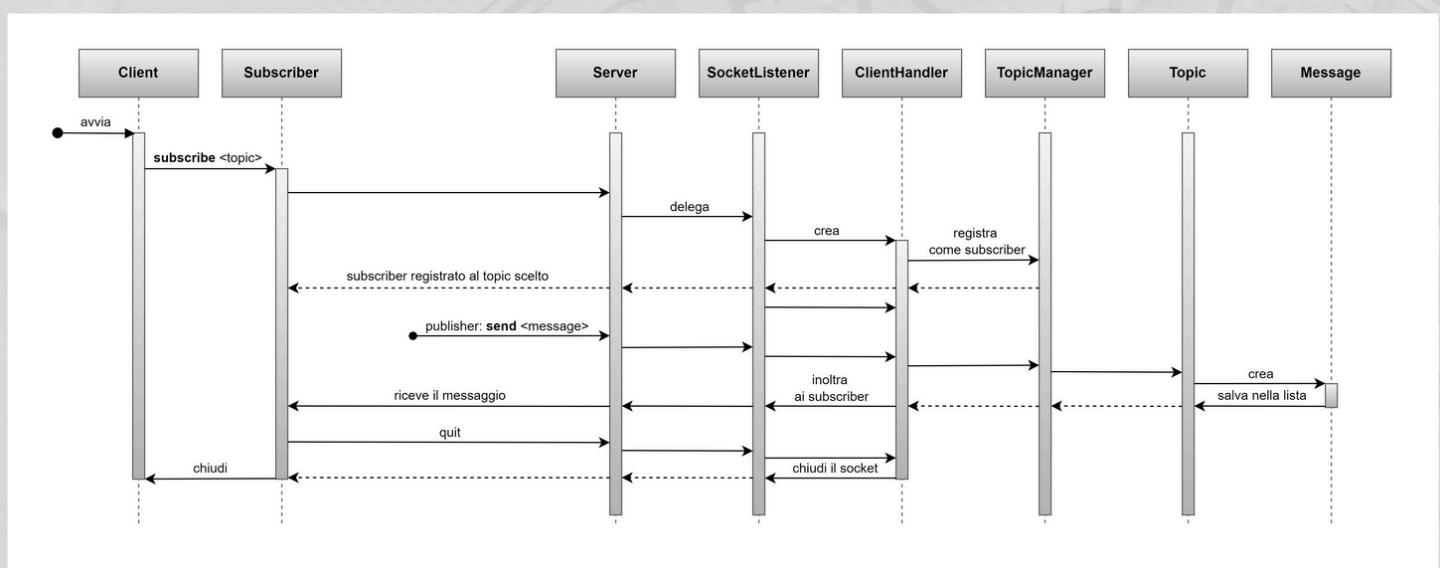
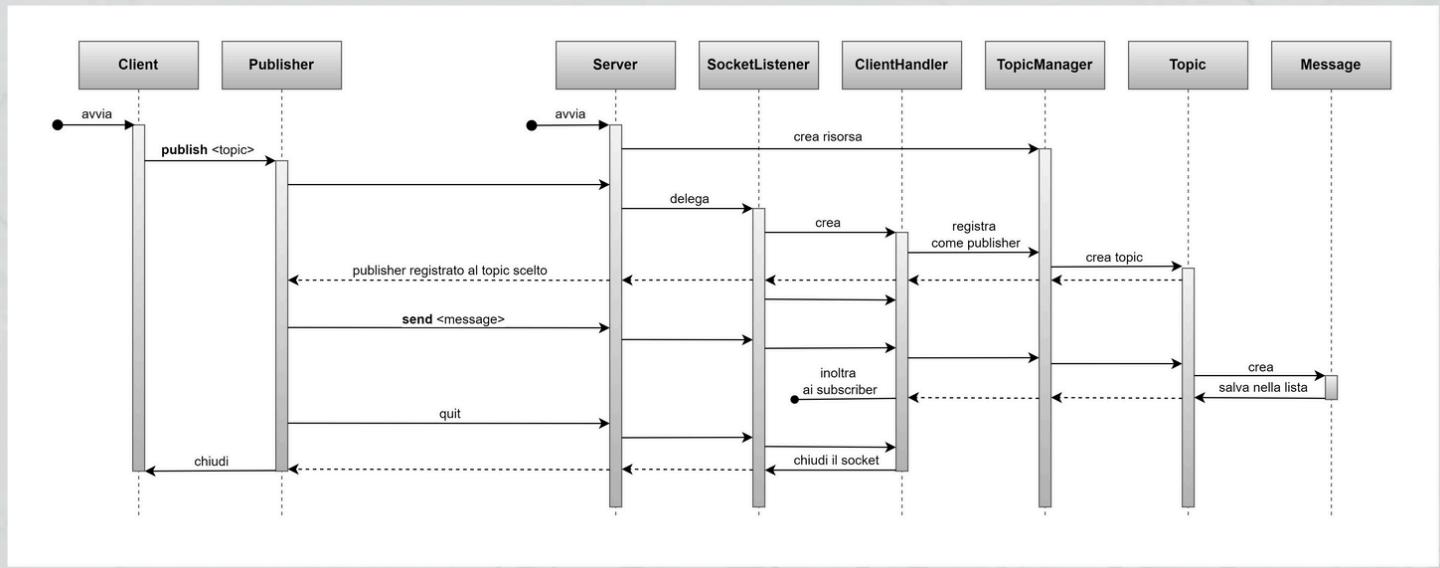
- **TopicManager.java**: Gestisce tutti i topic presenti nel sistema. Coordina i publisher e i subscriber, permettendo di aggiungere nuovi topic, iscrivere utenti, pubblicare messaggi e richiedere informazioni sui topic esistenti.
- **Topic.java**: Gestisce un singolo topic, includendo publisher, subscriber e i messaggi pubblicati. Fornisce metodi per aggiungere publisher e subscriber, pubblicare messaggi e visualizzare o eliminare quelli esistenti.
- **Message.java**: Rappresenta un messaggio con ID univoco, contenuto, timestamp e publisher associato. I messaggi sono gestiti all'interno dei topic, consentendo visualizzazione e cancellazione.

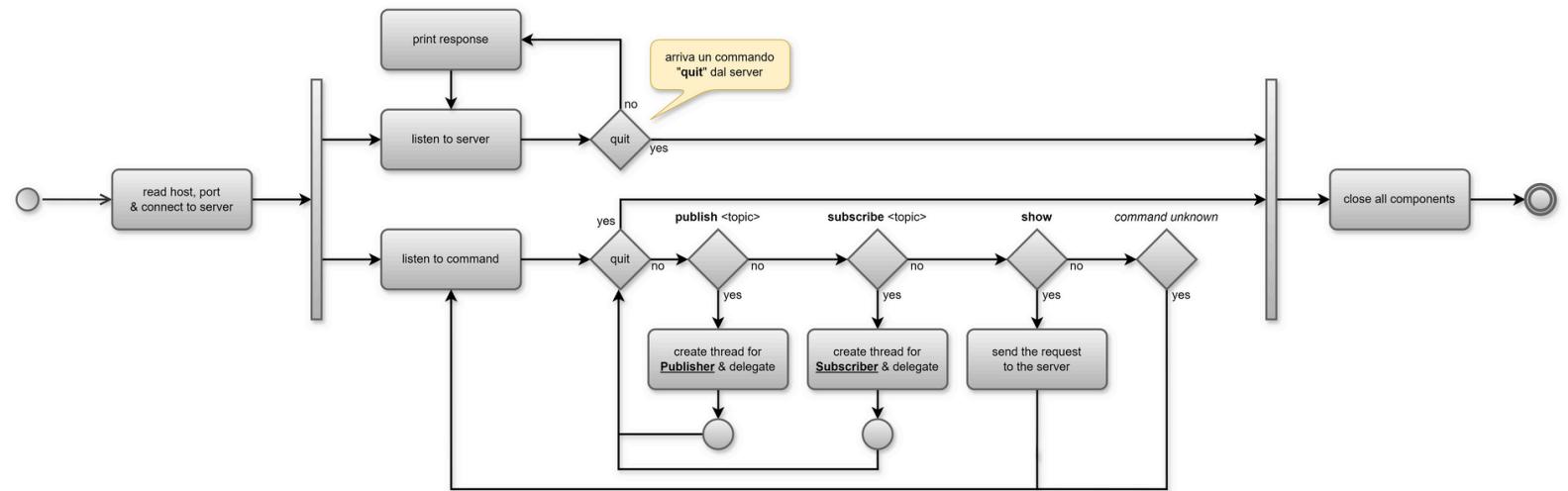
COMPONENTI



INTERAZIONE TRA COMPONENTI

Di seguito alcuni diagrammi di sequenza che illustrano chiaramente le interazioni tra gli oggetti e come avvengono le comunicazioni all'interno del sistema.





activity diagram: classe **client**

CLIENT

La classe **Client** stabilisce una connessione **socket** con un server, consentendo la comunicazione bidirezionale tramite comandi. Il main verifica che i parametri *host* e *port* siano forniti; in caso contrario, termina con un messaggio d'errore. Se la connessione viene stabilita con successo, il client stampa un messaggio di conferma e lancia un thread **serverListener** dedicato all'ascolto continuo dei messaggi provenienti dal server.

L'array *quit* viene utilizzato per tenere traccia dello stato del client, facilitando il controllo dei cicli di esecuzione e dei thread. Il thread **serverListener** legge i messaggi dal server tramite **Socket.getInputStream()**, interrompendosi se riceve un comando di disconnessione (*quit*) o se il socket viene chiuso. Eventuali thread attivi, come **Publisher** o **Subscriber**, vengono interrotti, gestendo in modo sicuro le risorse e le eccezioni legate alla chiusura della connessione.

Il client gestisce i comandi dell'utente tramite un ciclo while, che consente l'inserimento di comandi come *publish*, *subscribe*, *show* e *quit*, attivando per ciascuno di essi un'azione specifica. I comandi *publish* e *subscribe* avviano rispettivamente un thread **Publisher** o **Subscriber**, associato a un topic, e ogni thread gestisce il proprio socket e viene seguito da un join per garantire la sincronia. Il comando *show* invoca il comando *show* al server tramite **PrintWriter(toServer)** per richiedere i topic disponibili. Il comando *quit* chiude la connessione inviando un messaggio al server e interrompendo eventuali thread attivi, consentendo al client di chiudere in modo sicuro.

La gestione delle risorse è ottimizzata: al termine, *socket* e *userInput* vengono chiusi. Prima della chiusura finale, **serverListener** viene interrotto e raggiunto tramite *join*, assicurando una chiusura completa e sicura del client, che stampa un messaggio finale per segnalare la terminazione. Questo design garantisce un'esecuzione sicura e reattiva, evitando blocchi o errori di concorrenza.

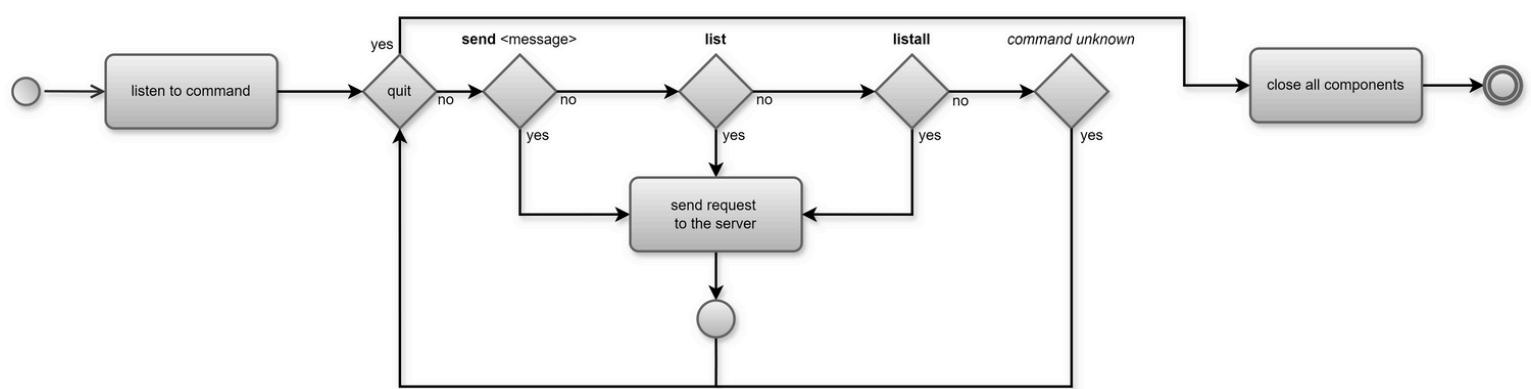
PUBLISHER

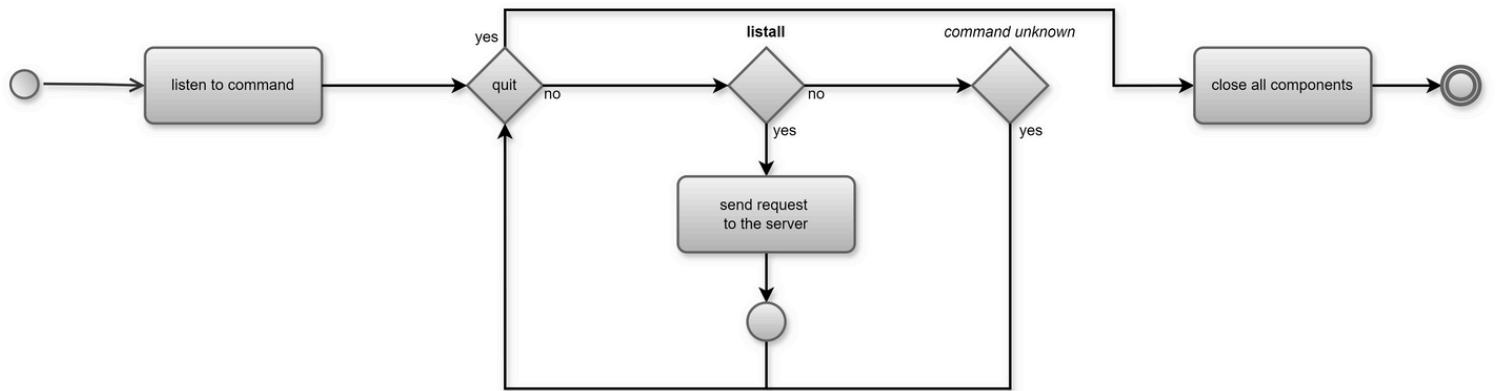
La classe **Publisher** implementa l'interfaccia **Runnable** e gestisce la pubblicazione di messaggi su un topic specifico tramite un socket connesso al server. Al momento della creazione, il costruttore accetta un **Socket** per la comunicazione e un String per il nome del topic, assegnando entrambi ai campi di istanza. Il metodo run definisce il comportamento del thread Publisher, configurando un **PrintWriter** per inviare comandi al server e uno scanner per leggere i comandi dell'utente da tastiera. Una volta avviato, il thread invia il comando **publish** per dichiarare l'intenzione di pubblicare sul topic specificato e avvia un ciclo per consentire all'utente di interagire attraverso comandi di pubblicazione e gestione.

Il ciclo del Publisher verifica la variabile **quiting**, una **volatile** che assicura la visibilità tra i thread, e include una pausa di due secondi tra le iterazioni per ridurre il carico. All'interno del ciclo, vengono gestiti diversi comandi: **send** per inviare un messaggio al topic, **list** per richiedere i messaggi pubblicati solo da questo publisher, **listall** per visualizzare tutti i messaggi del topic, e **quit** per terminare la connessione. Se l'utente inserisce il comando **send**, il programma estrae il messaggio e lo invia al server con **toServer.println**, mentre i comandi **list** e **listall** richiedono informazioni sul topic. In caso di comando **quit**, il programma invia al server una notifica di disconnessione, interrompe il ciclo impostando **quiting** a true e chiude il thread e le risorse.

Il design della classe è robusto nella gestione delle risorse: ogni sessione si conclude con la chiusura del socket e dello scanner per evitare fughe di risorse. Inoltre, la classe gestisce eventuali eccezioni **IOException** durante l'interazione con il server, notificando l'utente in caso di problemi di connessione. Questo approccio garantisce una pubblicazione affidabile e una disconnessione sicura, permettendo al publisher di interagire con il server in modo flessibile e sicuro, evitando blocchi e ottimizzando l'uso delle risorse.

activity diagram: classe **publisher**





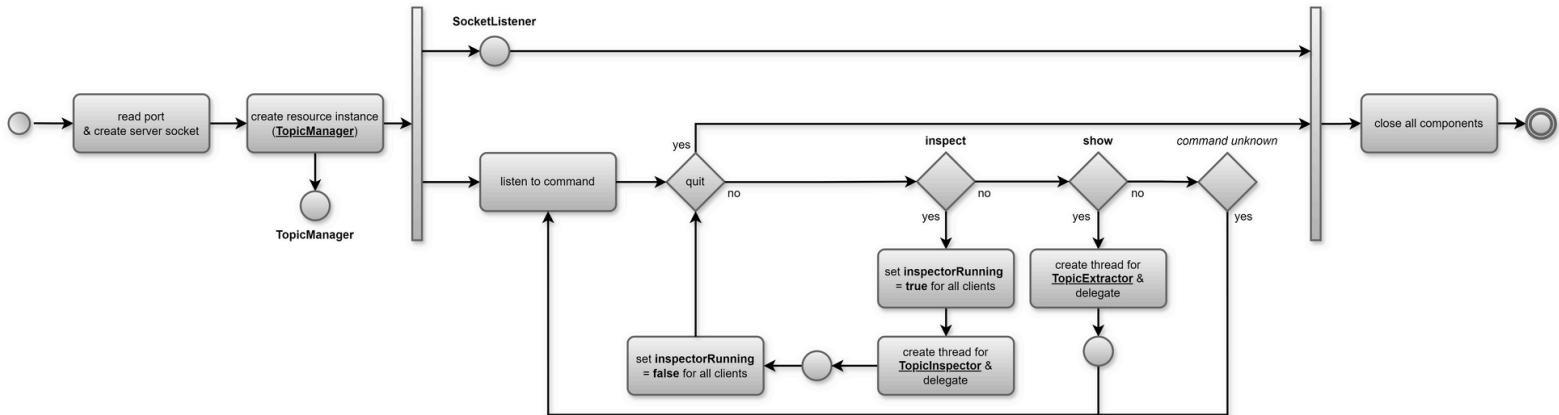
activity diagram: classe **subscriber**

SUBSCRIBER

La classe **Subscriber** implementa l'interfaccia **Runnable** e gestisce la sottoscrizione a un topic su un server, consentendo a un client di ricevere aggiornamenti su tale topic. Il costruttore accetta un oggetto **Socket** per la comunicazione e un String per il nome del topic a cui il Subscriber si iscrive. Questi parametri vengono salvati nei campi di istanza, e il metodo run implementa il comportamento del thread, configurando un **PrintWriter** per inviare comandi al server e uno scanner per ricevere l'input dell'utente. All'avvio, il thread invia il comando subscribe al server per confermare l'iscrizione al topic, e quindi entra in un ciclo che permette all'utente di interagire attraverso comandi come listall e quit.

All'interno del ciclo principale, il thread verifica lo stato della variabile `quiting`, che è dichiarata volatile per garantire la visibilità tra i thread e che viene utilizzata per determinare quando interrompere il ciclo. Ogni iterazione include una pausa di due secondi, gestita da un blocco **try-catch**, per rallentare l'esecuzione. Il comando **listall**, se inserito dall'utente, invia una richiesta al server per ottenere la lista di tutti i messaggi associati al topic. Se invece l'utente inserisce **quit**, il client invia una notifica al server per terminare la sottoscrizione, imposta `quiting` a true e interrompe il ciclo.

Al termine, la classe gestisce attentamente le risorse chiudendo il **PrintWriter** e lo scanner per evitare problemi di memoria. Eventuali **IOException** o altre eccezioni vengono gestite, informando l'utente in caso di problemi con la connessione al server. Il design della classe garantisce un'esperienza utente continua e affidabile, assicurando la chiusura delle risorse e la gestione delle eccezioni per evitare interruzioni non controllate durante l'esecuzione del thread.



activity diagram: classe **server**

SERVER

La classe **Server** avvia e gestisce le connessioni client su una porta specifica, permettendo di eseguire operazioni sui topic tramite comandi. Il metodo `main` verifica che la porta sia specificata tra gli argomenti, avvia un **ServerSocket** sulla porta indicata e crea un'istanza di **TopicManager**, condivisa tra i thread per la gestione centralizzata dei topic. Dopo l'avvio del server, viene creato un thread **SocketListener** che si occupa di accettare le connessioni dei client e di delegarle ai rispettivi **ClientHandler** per la gestione di ogni client connesso.

Il server utilizza un ciclo principale per gestire i comandi inseriti dall'utente, tra cui **show**, **inspect** e **quit**. Quando l'utente inserisce **show**, il server avvia un thread **TopicExtractor**, che estrae e visualizza tutti i topic attualmente disponibili tramite **TopicManager**. L'utente può inserire il comando **inspect**, che consente di ispezionare un topic specifico: viene chiesto il nome del topic, e se il topic è valido, **SocketListener** notifica a tutti i **ClientHandler** che è in corso un'ispezione. Un thread **TopicInspector** viene quindi avviato per esaminare il topic, e al termine dell'ispezione, il server ripristina l'operatività dei client.

Se l'utente digita **quit**, il server esce dal ciclo principale e avvia la procedura di chiusura. Prima di terminare, interrompe il thread **SocketListener** e attende che completi la sua esecuzione tramite **join**, per garantire che tutti i client siano disconnessi in modo sicuro. La gestione delle eccezioni, come le **IOException**, permette al server di identificare e gestire errori di connessione e altri problemi in modo efficiente, con un feedback all'utente. Al termine, il server chiude le risorse e notifica all'utente la conclusione dell'esecuzione.

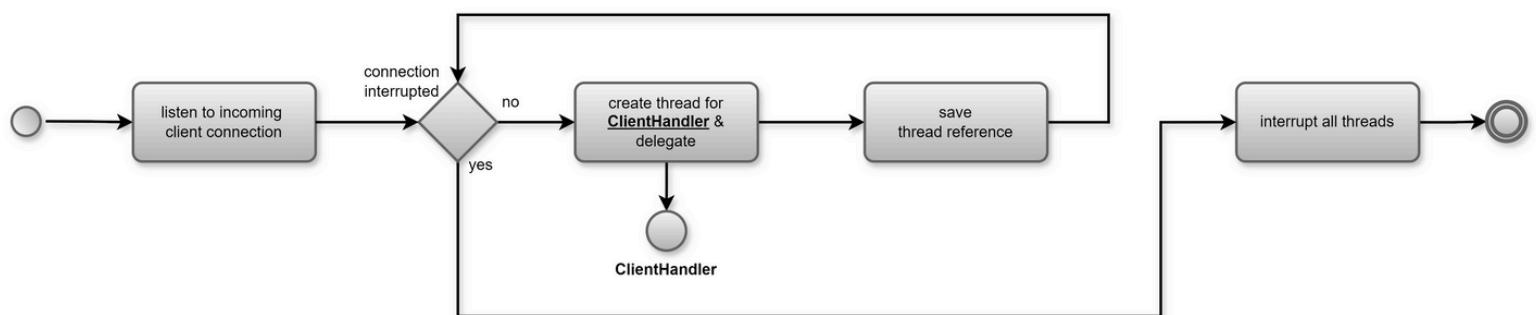
SOCKETLISTENER

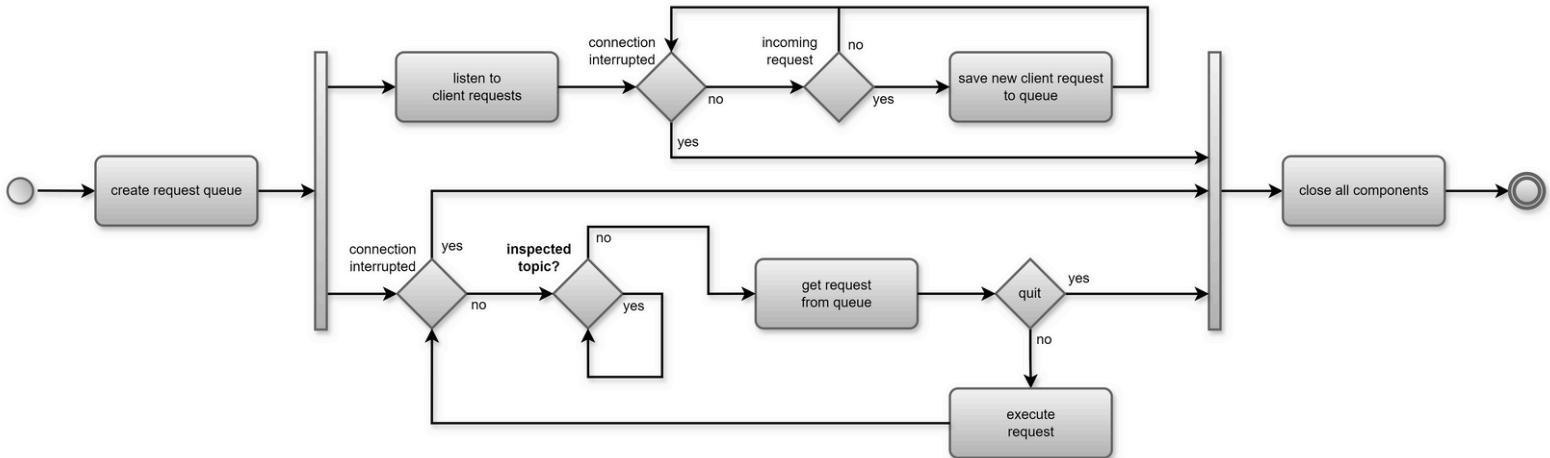
La classe **SocketListener** implementa l'interfaccia **Runnable** e gestisce l'accettazione delle connessioni dai client, distribuendole a **ClientHandler** dedicati per ogni client connesso. Al momento della creazione, riceve un oggetto **ServerSocket** per ascoltare le connessioni in ingresso e un **TopicManager** condiviso per gestire i topic. Le variabili **children** e **clientHandlers** sono **ConcurrentHashMap** usate per mappare i thread client con i rispettivi **Socket** e **ClientHandler**, permettendo una gestione concorrente sicura. Il campo **inspectedTopic**, **volatile**, indica il topic corrente sotto ispezione per evitare conflitti durante l'accesso ai topic da parte dei client.

Nel metodo **run**, il **SocketListener** entra in un ciclo per accettare le connessioni, utilizzando un timeout per evitare blocchi indefiniti. Quando un client si connette, viene creato un thread **ClientHandler** dedicato che gestisce la comunicazione e le operazioni per quel client, e il thread viene mappato nelle tabelle **children** e **clientHandlers** per il controllo e la gestione centralizzata. In caso di interruzione del thread, o se il server chiude il socket, la connessione viene terminata e il ciclo di accettazione si conclude.

Il metodo **setInspectorRunningForAllClients** aggiorna **inspectedTopic** e notifica ogni **ClientHandler** per sospendere le operazioni su quel topic finché l'ispezione non termina. Infine, alla chiusura, **SocketListener** esegue una procedura di terminazione per ogni connessione: interrompe ogni thread **ClientHandler** e invia il comando **quit** ai client, assicurando una disconnessione sicura. Il design del **SocketListener** garantisce la gestione concorrente dei client, mantenendo un controllo centralizzato e sicuro delle connessioni e dei topic, con gestione ottimizzata delle risorse e delle eccezioni.

activity diagram: classe **socketlistener**





activity diagram: classe **clienthandler**

CLIENTHANDLER

La classe **ClientHandler** implementa **Runnable** e gestisce la comunicazione tra il server e un client connesso tramite un socket. Durante la creazione, il costruttore riceve il **Socket** per il collegamento al client, una risorsa condivisa **TopicManager** per la gestione centralizzata dei topic, e una variabile **inspectedTopic**, che indica se un'ispezione è in corso su un topic specifico. Il thread gestito da ClientHandler utilizza una coda **requestQueue** per memorizzare le richieste ricevute dal client e processarle in modo sicuro.

Nel metodo **run**, ClientHandler avvia un thread **requestReader**, che legge i comandi inviati dal client e li inserisce nella **requestQueue**. Il ciclo principale preleva ogni richiesta dalla coda e verifica se il topic associato è soggetto a ispezione; in tal caso, il thread attende finché l'ispezione non termina. Questo controllo utilizza il metodo **setInspectorRunning**, che notifica tutti i thread quando l'ispezione è completata, sincronizzando l'accesso ai topic.

Durante l'elaborazione dei comandi, il thread gestisce diversi tipi di richieste, tra cui **publish**, **subscribe**, **list**, **listall**, **send**, **show** e **quit**. Ogni comando richiama una funzione specifica nel TopicManager per interagire con i topic, come l'aggiunta di un publisher o la lista dei messaggi per un topic. In caso di comando **send**, ClientHandler invia un messaggio a tutti i subscriber del topic, sincronizzando l'accesso al socket di ciascun subscriber per prevenire race conditions. Se l'utente invia **quit**, il thread chiude le risorse e termina.

Alla chiusura, ClientHandler interrompe i flussi di input/output e chiude il socket, assicurando che tutte le risorse siano gestite in modo sicuro. Eventuali eccezioni di input/output vengono catturate per garantire la stabilità del server, con un feedback appropriato all'utente in caso di errore. Questo design assicura un'interazione client-server sicura, stabile e sincronizzata.

TOPICEXTRACTOR

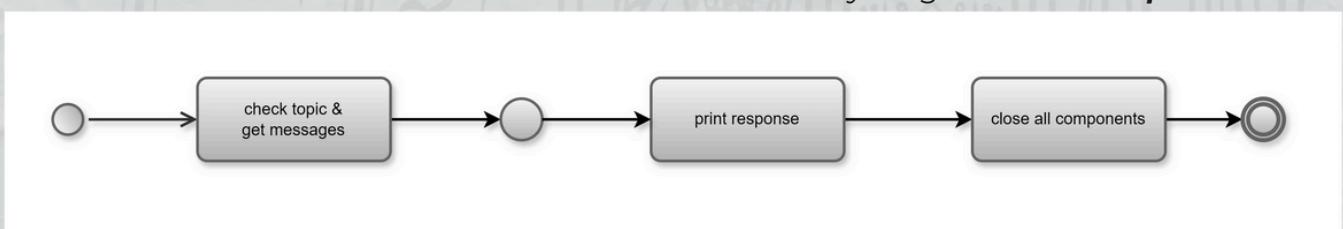
La classe **TopicExtractor** implementa l'interfaccia **Runnable** ed è utilizzata per estrarre e stampare i nomi dei topic gestiti dal **TopicManager**. Questa classe viene eseguita su un thread separato per evitare di bloccare il flusso principale del programma durante l'estrazione delle informazioni relative ai topic. Il costruttore della classe riceve come parametro un'istanza di TopicManager, che rappresenta la risorsa condivisa tra tutti i thread del server per gestire i topic di comunicazione tra i client.

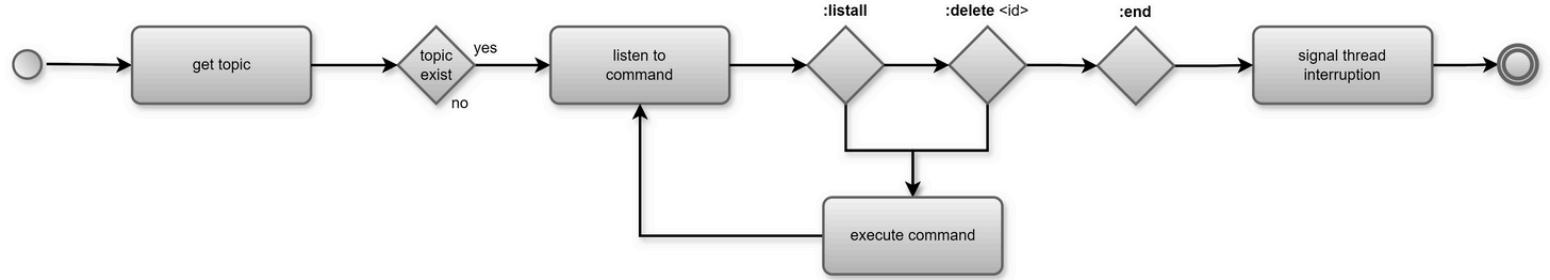
Quando il metodo **run** viene chiamato, il thread esegue la logica di estrazione dei nomi dei topic dal **TopicManager**. Il metodo **getTopicNames** viene invocato sul TopicManager, il quale restituisce una lista contenente i nomi di tutti i topic attivi nel sistema. Se la lista è vuota, significa che non ci sono topic attualmente disponibili, e quindi viene stampato un messaggio che avvisa l'utente di questa condizione. In caso contrario, i nomi dei topic vengono iterati e stampati a video, ciascuno preceduto da un trattino, per migliorarne la leggibilità.

La classe è progettata per essere semplice ed efficiente, delegando completamente la gestione dei topic al TopicManager, che funge da punto centrale per tutte le operazioni legate ai topic. Non contiene meccanismi complessi di sincronizzazione o gestione avanzata delle risorse, poiché il suo scopo è limitato alla lettura e visualizzazione delle informazioni.

L'uso di un thread separato permette al server di continuare a gestire altre operazioni, come le connessioni con nuovi client, senza doversi fermare per aspettare la stampa dei topic. Questo approccio migliora la reattività generale del sistema, specialmente in ambienti con molti client e topic attivi.

activity diagram: classe **topicextractor**





activity diagram: classe **topicinspector**

TOPICINSPECTOR

La classe **TopicInspector** implementa l'interfaccia **Runnable** e fornisce una sessione interattiva per l'ispezione e la gestione di un topic specifico all'interno di un **TopicManager** condiviso. Il costruttore riceve un'istanza di TopicManager, che contiene tutti i topic, e il nome del topic da ispezionare. Il metodo **run** gestisce l'esecuzione del thread e avvia la sessione interattiva, permettendo all'utente di visualizzare o modificare i messaggi all'interno del topic selezionato.

All'inizio del metodo **run**, TopicInspector cerca il topic specificato utilizzando il TopicManager e, una volta ottenuto, entra in un ciclo interattivo, in cui accetta comandi come **:listall**, **:delete <id>** e **:end**. Il comando **:listall** visualizza tutti i messaggi presenti nel topic, elencandoli in base alla cronologia. Se non ci sono messaggi, informa l'utente che non sono stati inviati messaggi per il topic corrente. Il comando **:delete <id>** consente di eliminare un messaggio specifico tramite il suo ID. Se l'ID fornito è valido e corrisponde a un messaggio esistente, il messaggio viene eliminato e l'utente riceve un feedback di conferma; altrimenti, viene informato che il messaggio non è stato trovato o che l'ID non è valido. Il comando **:end** termina la sessione di ispezione, consentendo al thread di uscire dal ciclo interattivo e concludere l'esecuzione.

La classe TopicInspector garantisce una gestione sicura e controllata delle operazioni sui messaggi, offrendo all'utente un'interfaccia interattiva per ispezionare e modificare i contenuti del topic. La gestione delle eccezioni, come **NumberFormatException**, assicura un'esperienza utente fluida, impedendo interruzioni impreviste e garantendo che i comandi non validi vengano gestiti con messaggi appropriati.

TopicManager
<p>- topics: ConcurrentHashMap<String,Topic></p>
<pre>+ Topicmanager() + publishMessage(ClientHandler publisher, String topicName, String message): List<ClientHandler> + addPublisher(ClientHandler publisher, String topicName): void + addSubscriber(ClientHandler subscriber, String topicName): void + publishMessage(ClientHandler publisher, String topicName, String message): List<ClientHandler> + listMessagesByTopic(String topicName): Optional<List<Message>> + getTopicNames(): List<String> + getTopicByName(String topicName): Optional<Topic></pre>

class diagram: **topicmanager**

TOPICMANAGER

La classe **TopicManager** gestisce la creazione, la modifica e l'interazione con i topic. È implementata come un oggetto **thread-safe** utilizzando una mappa concorrente, **ConcurrentHashMap**, per garantire che le operazioni sui topic siano eseguite in modo sicuro quando ci sono accessi simultanei da parte di più client.

Il costruttore inizializza la mappa topics, che associa i nomi dei topic agli oggetti **Topic**. Quando un publisher o subscriber tenta di interagire con un topic, vengono utilizzati i metodi **addPublisher** e **addSubscriber**. Se il topic non esiste, ne viene creato uno nuovo; altrimenti, l'utente viene semplicemente aggiunto all'elenco dei partecipanti (*publisher* o *subscriber*). Questi metodi sono sincronizzati per prevenire condizioni di corsa, garantendo che solo un thread possa eseguire l'operazione alla volta.

Il metodo **publishMessage** consente di pubblicare un messaggio su un topic specifico. Esso verifica se il topic esiste e, in tal caso, aggiunge il messaggio associandolo al publisher che lo ha inviato. Inoltre, ritorna la lista dei subscriber associati al topic, che verranno notificati del nuovo messaggio.

Per quanto riguarda la lettura dei messaggi, **listMessages** e **listMessagesByTopic** permettono di ottenere rispettivamente i messaggi di un publisher specifico o tutti i messaggi relativi a un topic. L'uso degli **Optional** gestisce in modo sicuro il caso in cui un topic non esista, evitando eccezioni null pointer.

Infine, i metodi **getTopicNames** e **getTopicByName** forniscono rispettivamente una lista di tutti i nomi dei topic e un **Optional** per verificare l'esistenza di un topic. Queste operazioni sono utili per mostrare i topic attivi e facilitare la navigazione da parte degli utenti.

La classe **Topic** gestisce tutte le informazioni e operazioni relative ai singoli topic, tra cui la gestione di publisher, subscriber e messaggi. Ogni oggetto Topic contiene il nome del topic, una mappa concorrente per associare i publisher ai loro messaggi, e due liste per gestire i publisher e i subscriber. La sincronizzazione viene applicata ai metodi che modificano lo stato interno del topic per garantire che operazioni concorrenti da parte di più thread siano eseguite correttamente.

Il costruttore inizializza la mappa messages per tenere traccia dei messaggi associati a ciascun publisher, le liste dei subscriber e dei publisher, e un contatore per assegnare ID univoci ai messaggi. Quando viene aggiunto un nuovo messaggio, il metodo **addMessage** utilizza il contatore per assegnare un ID e aggiorna la lista dei messaggi per quel publisher. La mappa messages garantisce che ogni publisher abbia la propria lista separata di messaggi.

I metodi **addSubscriber** e **addPublisher** gestiscono rispettivamente l'aggiunta di nuovi subscriber e publisher al topic. Questi metodi sono sincronizzati per evitare problemi di concorrenza e assicurarsi che l'aggiunta avvenga correttamente anche se ci sono più thread attivi.

Per ottenere i messaggi di un publisher, il metodo **getMessagesByPublisher** restituisce la lista dei messaggi associati a quel publisher. Esiste anche la possibilità di ottenere tutti i messaggi del topic tramite **getAllMessages**, che restituisce la mappa completa di tutti i publisher e dei loro messaggi, o tramite **getAllMessagesAsList**, che combina tutti i messaggi in una singola lista.

Infine, il metodo **deleteMessageById** permette di eliminare un messaggio dato il suo ID. Il metodo itera attraverso tutte le liste dei messaggi e, se trova un messaggio con l'ID specificato, lo rimuove. Questo metodo restituisce un valore booleano che indica se l'operazione di eliminazione è avvenuta con successo.

TOPIC

Topic

```
- name: String  
- messages: ConcurrentHashMap<ClientHandler, List<Message>>  
- subscribers: List<ClientHandler>  
- publishers: List<ClientHandler>  
- counter: int  
  
+ Topic (String name)  
+ addMessage(String text, ClientHandler publisher): void  
+ addSubscriber(ClientHandler subscriber): void  
+ addPublisher(ClientHandler publisher): void  
+ getMessagesByPublisher(ClientHandler publisher): List<Message>  
+ getSubscribers(List<ClientHandler> subscribers): void  
+ getAllMessages(): ConcurrentHashMap<ClientHandler, List<Message>>  
+ getAllMessagesAsList(): List<Message>  
+ deleteMessageById(int id): boolean
```

class diagram: **topic**

MESSAGE

La classe **Message** è responsabile della gestione e della rappresentazione di singoli messaggi all'interno di un sistema di pubblicazione e sottoscrizione. Ogni istanza della classe rappresenta un messaggio univoco, identificato da un ID, che viene associato a un determinato publisher, e contiene un testo e un timestamp di creazione.

Il costruttore della classe accetta tre parametri: l'ID del messaggio, il contenuto del messaggio e il riferimento al publisher che ha inviato il messaggio. Una volta creato il messaggio, viene generato automaticamente un timestamp utilizzando il metodo **generateTimestamp()**. Questo metodo utilizza la classe **SimpleDateFormat** per formattare la data corrente nel formato "dd/MM/yyyy - HH:mm", permettendo di tenere traccia del momento esatto in cui il messaggio è stato creato.

L'attributo **id** fornisce un identificatore univoco per ciascun messaggio, consentendo una gestione specifica e operazioni come l'eliminazione o la modifica di un messaggio. Il campo **message** contiene il testo effettivo del messaggio, mentre l'attributo **publisher** tiene traccia dell'oggetto **ClientHandler** che ha originato il messaggio. Questo permette di mantenere una relazione tra i messaggi e i loro autori, facilitando la gestione del flusso di comunicazione.

Il metodo **getId()** restituisce l'ID del messaggio, permettendo di accedere facilmente all'identificatore univoco di ciascun messaggio per eseguire operazioni specifiche, come la ricerca o la cancellazione. Il metodo **toString()** fornisce una rappresentazione leggibile del messaggio, includendo il suo ID, il contenuto, il timestamp di creazione e il publisher associato. Questa rappresentazione è utile per la visualizzazione dei messaggi e per operazioni di log o debugging.

Message

```
- id: int  
- message: String  
- datetimestamp: String  
- publisher: ClientHandler  
  
+ Message(int id, String message, ClientHandler publisher)  
- generateTimestamp(): String  
+ getId(): int  
+ toString(): String
```

class diagram: **message**

SUDDIVISIONE DEI TASK

Il nostro gruppo ha adottato una strategia di lavoro collaborativa che ha permesso a ciascun membro di essere coinvolto sia nello sviluppo del codice che nella documentazione e nella creazione dei diagrammi UML. La strategia seguita è stata di dividere i compiti in modo complementare: se un membro sviluppava una parte del codice, l'altro membro si occupava di documentarla e di produrre i diagrammi correlati. Questo approccio ha garantito che ogni membro fosse allineato sul lavoro svolto dall'altro, permettendo una stretta collaborazione e una revisione reciproca continua.

Per organizzare i task, abbiamo utilizzato **GitHub Projects**, uno strumento che ci ha permesso di tracciare facilmente i progressi, assegnare i task e gestire le scadenze. I task erano suddivisi in modo chiaro e assegnati ai membri in base alle loro competenze e alla parte del progetto in cui erano coinvolti.

Publish-Subscribe Project SO2024			
Backlog	Tasks	+ New view	
Filter by keyword or by field			
Title	Assignees	Status	...
andreacrox, francescodoriano, and ngljcb 2
8 ② Analisi del Progetto + Creazione UML #2	andreacrox, francescodoriano, and ngljcb	Done	...
9 ② Creazione classe 'ClientHandler' #38	andreacrox, francescodoriano, and ngljcb	Done	...
andreacrox 7
1 ② Creazione classe 'Publisher' #4	andreacrox	Done	...
2 ② Documentare: classe 'Publisher' #11	andreacrox	Done	...
3 ② Creazione classe 'TopicManager' #3	andreacrox	Done	...
4 ② Creazione classe 'Topic' #25	andreacrox	Done	...
5 ② Documentare: classe 'Message' #28	andreacrox	Done	...
6 ② Creazione classe "TopicInspector" #40	andreacrox	Done	...
7 ② Documentare: classe "TopicExtractor" #41	andreacrox	Done	...
francescodoriano 8
10 ② Creazione classe 'ShowTopics' #9	francescodoriano	Done	...
11 ② Documentare: classe ShowTopics #23	francescodoriano	Done	...
12 ② Documentare: classe 'Topic' #27	francescodoriano	Done	...
13 ② Documentare: classe 'Subscriber' #12	francescodoriano	Done	...
14 ② Creazione classe 'Subscriber' #5	francescodoriano	Done	...
15 ② Creazione classe 'TopicExtractor' #39	francescodoriano	Done	...
16 ② Creazione classe 'Message' #26	francescodoriano	Done	...
17 ② Documentare: classe "TopicInspector" #42	francescodoriano	Done	...
ngljcb 6
18 ② Documentare: classe 'Client' #10	ngljcb	Done	...
19 ② Documentare: classe Server #21	ngljcb	Done	...
20 ② Creazione classe 'SocketListener' #20	ngljcb	Done	...
21 ② Creazione classe 'Server' #13	ngljcb	Done	...
22 ② Creazione classe 'Client' #6	ngljcb	Done	...
23 ② Documentare: classe SocketListener #24	ngljcb	Done	...

PROBLEMI RISCONTRATI E SOLUZIONI

1. System.in è bloccante:

- Problema: Quando l'input dell'utente viene gestito tramite System.in nel ciclo principale del client, il client non è in grado di visualizzare le risposte del server in tempo reale. Questo perché l'ascolto del server e la gestione dell'input avvenivano nello stesso thread, causando un blocco nell'attesa dell'input da tastiera, che impedisce la ricezione e la visualizzazione immediata dei messaggi dal server.
- Soluzione: L'ascolto dei messaggi dal server è stato separato in un thread dedicato. Questo approccio permette al client di continuare a ricevere e visualizzare i messaggi dal server senza attendere l'input dell'utente, migliorando la reattività del client e garantendo che i messaggi del server vengano mostrati immediatamente.

2. Utilizzo di Socket e componenti chiusi:

- Problema: In alcune situazioni, i socket o altre risorse del client continuavano a essere utilizzati anche dopo la loro chiusura, portando a eccezioni come tentativi di lettura o scrittura su socket già chiusi.
- Soluzione: Sono stati aggiunti controlli, come `socket.isClosed()`, per verificare lo stato del socket prima di eseguire operazioni su di esso. Questo garantisce che il client non tenti di utilizzare risorse già chiuse, evitando eccezioni di input/output e migliorando la gestione delle risorse.

3. Sospensione delle Richieste durante l'Esecuzione del TopicInspector:

- Problema: Durante l'esecuzione del TopicInspector, le richieste dei client dovevano essere temporaneamente sospese per evitare conflitti con l'ispezione in corso. Le richieste dovevano essere salvate in un buffer e riprese solo al termine dell'ispezione.
- Soluzione: È stato introdotto un buffer (BlockingQueue) per salvare le richieste dei client. Nella classe SocketListener è stato aggiunto il metodo `setInspectorRunningForAllClients`, che segnala a tutti i ClientHandler che l'ispezione è in corso. Prima di avviare il TopicInspector, viene chiamato `setInspectorRunningForAllClients`, sospendendo l'elaborazione delle richieste dei client. Al termine dell'ispezione, viene chiamato `setInspectorRunningForAllClients`, consentendo ai client di riprendere l'esecuzione delle richieste in sospeso.
- Il blocco synchronized insieme ai metodi `wait()` e `notifyAll()` assicura che i thread client siano messi in attesa durante l'ispezione e riprendano l'elaborazione delle richieste in modo sicuro, una volta completata l'ispezione.

STRUMENTI UTILIZZATI

Sviluppo del progetto:

Abbiamo scelto **Visual Studio Code (VSCode)** come ambiente di sviluppo per la sua flessibilità e gli strumenti integrati di gestione del codice e debugging. Grazie alle sue estensioni, abbiamo potuto gestire facilmente il controllo delle versioni e collaborare in modo efficiente, mantenendo un flusso di lavoro rapido e ben organizzato.

Comunicazione:

La comunicazione tra i membri del gruppo è stata gestita principalmente tramite una chat di gruppo su **WhatsApp**, utilizzato per rapide discussioni e aggiornamenti. Per i meeting più strutturati e le discussioni tecniche, abbiamo fatto uso di **Microsoft Teams**, dove ci siamo coordinati con videochiamate e condivisione schermo per affrontare le problematiche in modo collaborativo.

Condivisione del codice:

Per la gestione del repository e il controllo delle versioni, abbiamo utilizzato **GitHub**. Ogni membro del team lavorava su branch separati per sviluppare nuove funzionalità o risolvere bug. Dopo il completamento, il codice veniva sottoposto a una revisione e successivamente mergeato nel branch principale tramite *pull request*. Questo processo ha assicurato la qualità e l'integrità del codice, evitando conflitti durante lo sviluppo.

Gestione del progetto:

La gestione delle attività e la pianificazione del lavoro sono state gestite tramite **GitHub Projects**, dove abbiamo suddiviso le task in colonne come "To Do", "In Progress", e "Done". Questo ci ha permesso di avere una visione chiara del progresso del progetto e di tracciare le attività rimanenti, assicurando che tutte le scadenze fossero rispettate e che il lavoro fosse ben distribuito tra i membri del team.

ISTRUZIONI D'USO

Il progetto è organizzato in due cartelle principali: *client* e *server*. Segui queste istruzioni per compilare e avviare le applicazioni client e server.

1. Compilazione del progetto

Vai nella cartella client o server usando il terminale. In ogni cartella (client o server), esegui il comando seguente per compilare tutti i file .java presenti:

```
javac *.java
```

2. Avvio del Server

Una volta compilato, puoi avviare il server navigando nella cartella server e usando il seguente comando:

```
java Server <port>
```

Sostituisci <port> con il numero di porta che desideri utilizzare.

Ad esempio:

```
java Server 8080
```

Il server sarà ora in ascolto sulla porta 8080 per ricevere le connessioni dei client.

3. Avvio del Client

Per avviare il client, naviga nella cartella client e usa il seguente comando:

```
java Client <host> <port>
```

Sostituisci <host> con l'indirizzo IP o il nome del server a cui vuoi connetterti, e <port> con la porta su cui il server sta ascoltando.

Ad esempio:

```
java Client localhost 8080
```

ISTRUZIONI D'USO

Una volta connesso al server, come client hai 3 opzioni:

1. Diventare un **Publisher** su un topic:

```
publish <topic>
```

Ad esempio:

```
publish sports
```

2. Diventare un **Subscriber** ad un topic per ricevere i messaggi pubblicati.

```
subscribe <topic>
```

Ad esempio:

```
subscribe sports
```

3. Mostrare tutti i topic disponibili sul server.

```
show
```

Esempio di output:

Topics:

- sports
- technology

ISTRUZIONI D'USO

Come **Publisher**, hai 4 comandi, compreso la *quit*:

Inviare un messaggio su un topic:

```
send <message>
```

Ad esempio:

```
send calcio serie A
```

Elenca i messaggi pubblicati dal client corrente.

```
list
```

Esempio di output:

Messaggi:

- ID: 1
Message: Game tonight at 8 PM
Date: 21/09/2023 - 18:00:00
Publisher: ClientHandler@1f32e575

Elenca tutti i messaggi, indipendentemente dal publisher.

```
listall
```

Esempio di output:

Sono stati inviati 2 messaggi per il topic sports.

- ID: 1
Message: Game tonight at 8 PM
Date: 21/09/2023 - 18:00:00
Publisher: ClientHandler@1f32e575

- ID: 2
Message: Who's playing?
Date: 21/09/2023 - 18:05:00
Publisher: ClientHandler@4a9b837a

ISTRUZIONI D'USO

Come **Subscriber**, hai 2 comandi:

Analogo al Publisher, **quit** per terminare il client:

```
quit
```

Elenca tutti i messaggi, indipendentemente dal publisher.

```
listall
```

Esempio di output:

```
Sono stati inviati 2 messaggi per il topic sports.
```

```
- ID: 1
```

```
    Message: Game tonight at 8 PM
```

```
    Date: 21/09/2023 - 18:00:00
```

```
    Publisher: ClientHandler@1f32e575
```

```
- ID: 2
```

```
    Message: Who's playing?
```

```
    Date: 21/09/2023 - 18:05:00
```

```
    Publisher: ClientHandler@4a9b837a
```

ISTRUZIONI D'USO

Lato Server, hai 3 comandi, compreso la *quit*:

Mostra tutti i topic disponibili:

```
show
```

Avviare un'ispezione interattiva sui messaggi di un topic.

```
inspect
```

Durante l'ispezione interattiva, inserire il topic da ispezionare:

Ad esempio:

```
sports
```

Da questo momento, hai 3 comandi, **:listall**, **:delete <id>** e **:end**

```
:listall
```

Esempio di output:

Sono stati inviati 2 messaggi per il topic sports.

- ID: 1

Message: Game tonight at 8 PM

Date: 21/09/2023 - 18:00:00

Publisher: ClientHandler@1f32e575

- ID: 2

Message: Who's playing?

Date: 21/09/2023 - 18:05:00

Publisher: ClientHandler@4a9b837a

Per eliminare un messaggio

```
:delete <id>
```

Per terminare l'ispezione

```
:end
```