

PUBLISH / SUBSCRIBE

LABORATORIO DI SISTEMI OPERATIVI
A.A. 2023-24

30/09/2024

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

Nome del Gruppo :

ACD2024

JACOB ANGELES
1114812

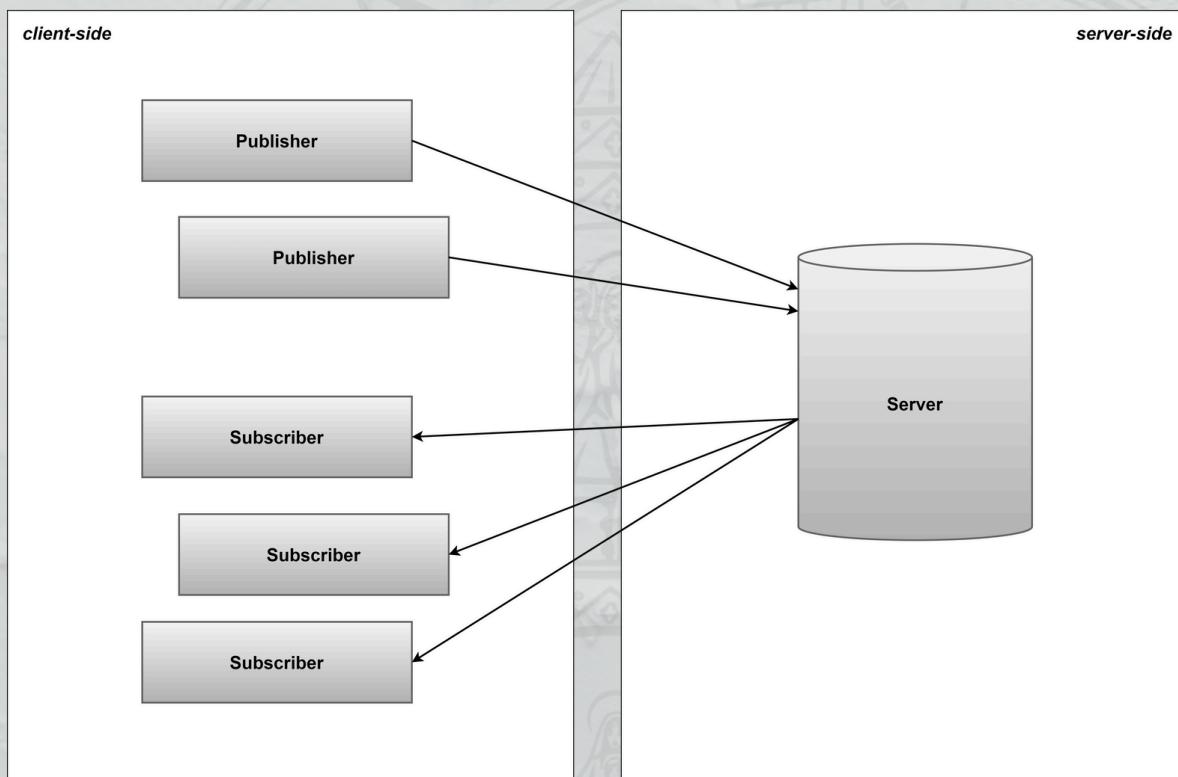
ANDREA CROCI
1054710

FRANCESCA D'ORIANO
1002020

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.



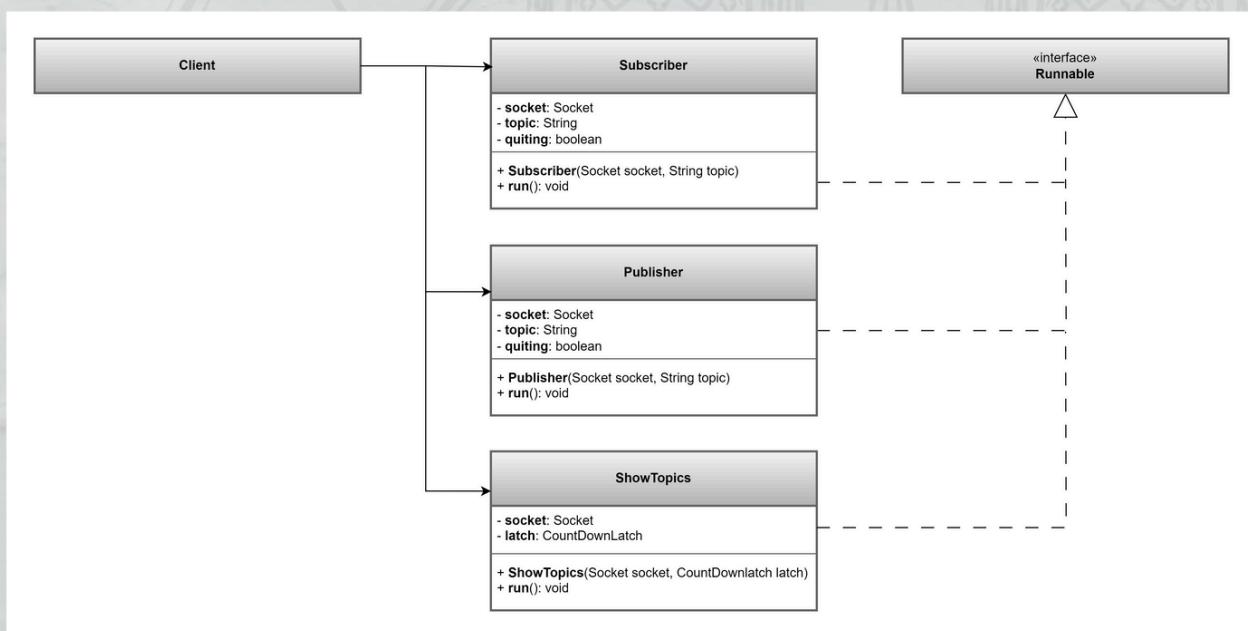
COMPONENTI

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 di pubblicare i messaggi o elencare quelli già inviati.
- **Subscriber.java**: Consente al client di iscriversi a un topic per ricevere messaggi pubblicati. Resta in ascolto dei messaggi fino a quando il client non termina la connessione.
- **ShowTopics.java**: Richiede al server la lista dei topic disponibili e la mostra al client. Utilizza un meccanismo di sincronizzazione per segnalare il completamento dell'operazione.



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.

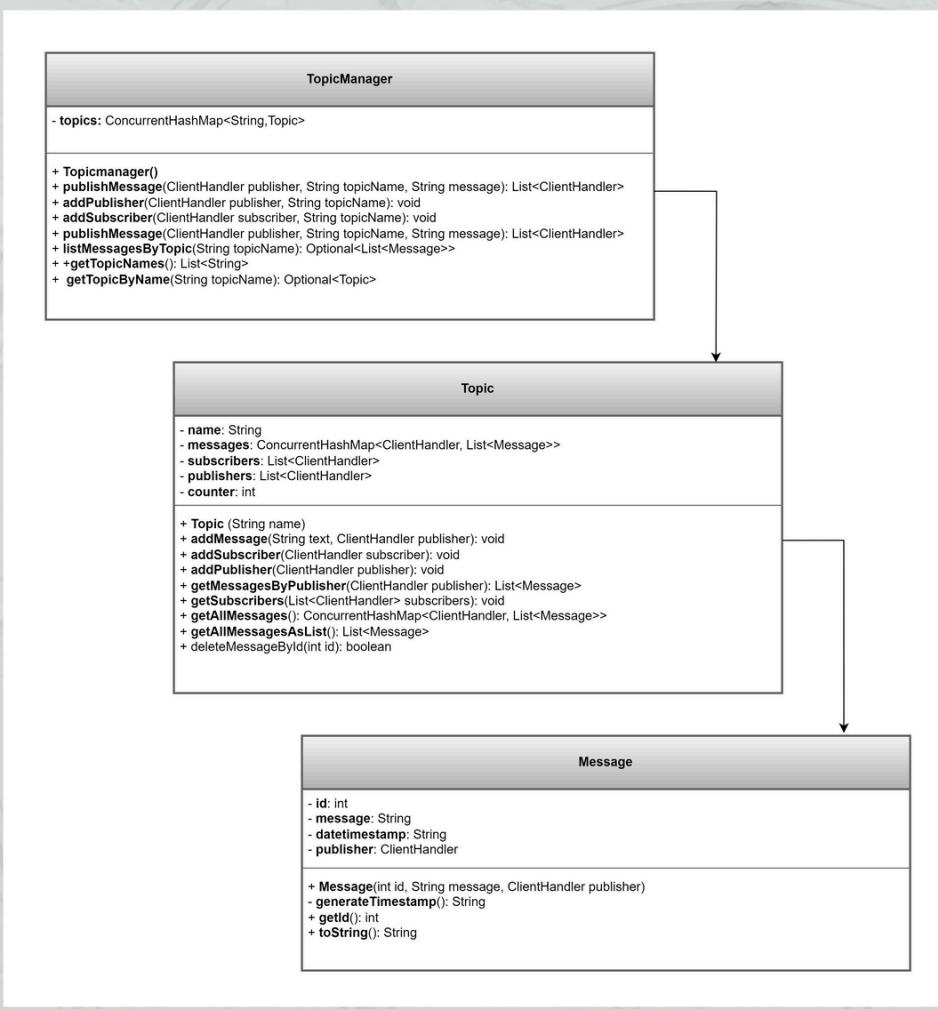
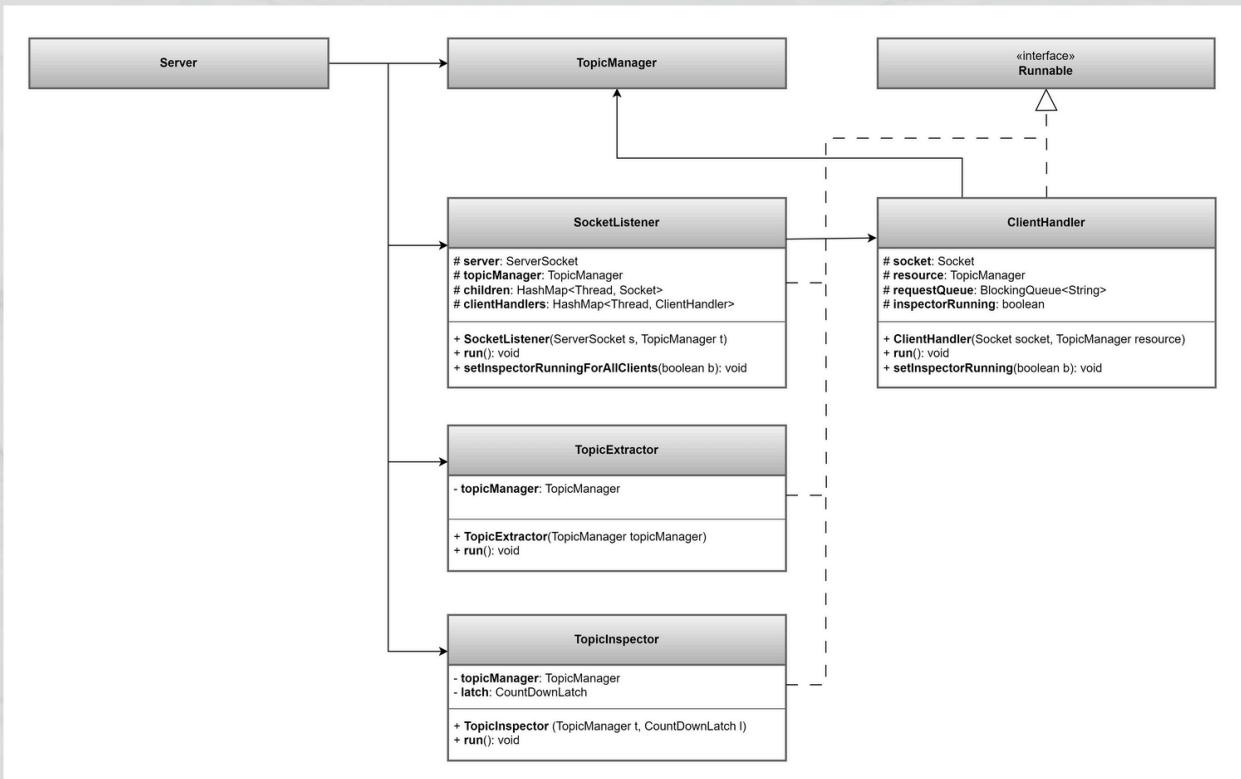
Le classi principali sono:

- **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:

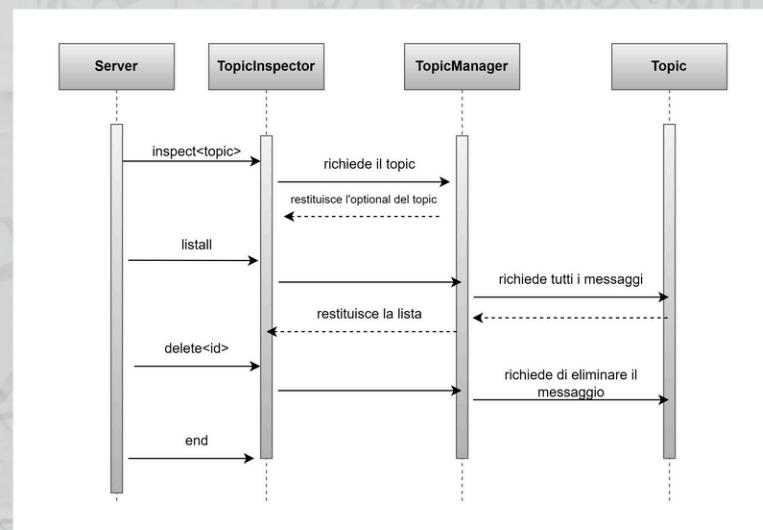
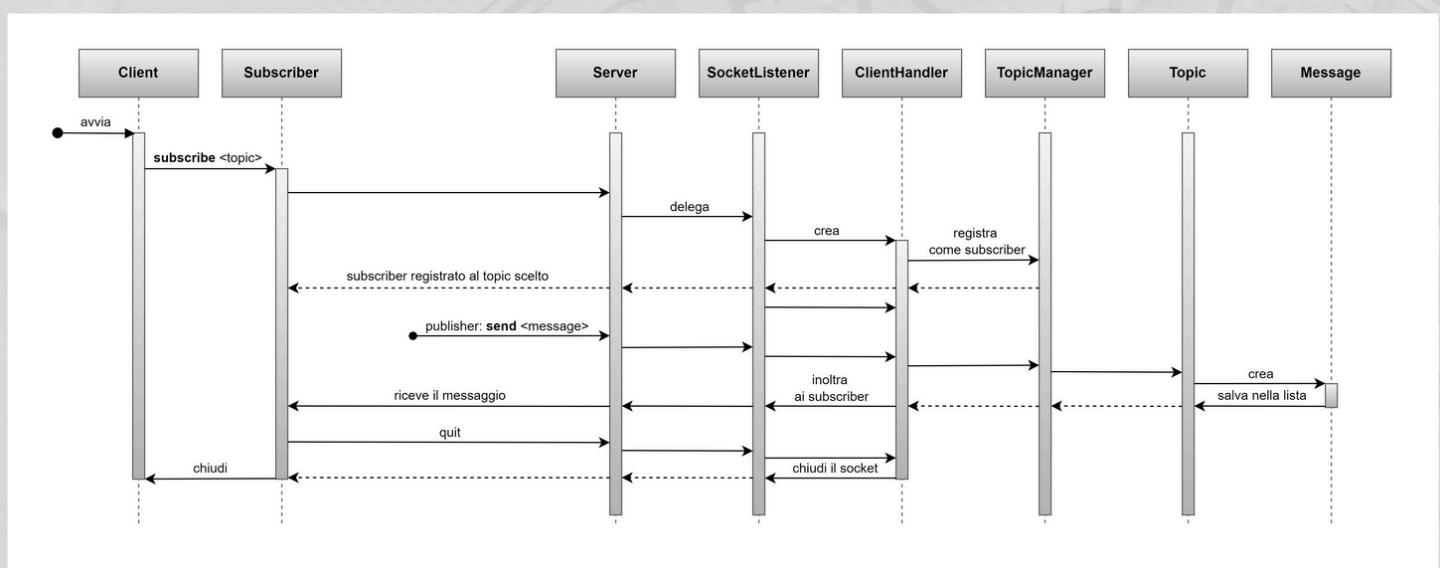
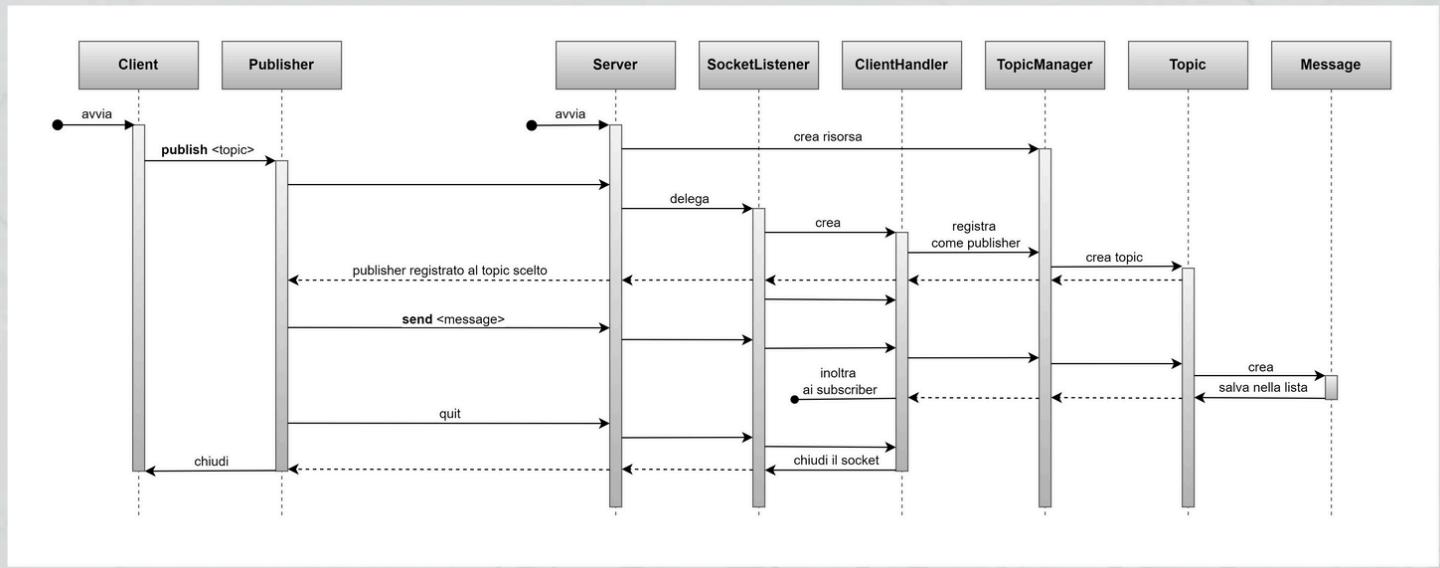
- **TopicManager.java**: Gestisce tutti i topic presenti nel sistema. Coordina i publisher e i subscriber, permettendo di aggiungere nuovi topic, iscrivere utenti, pubblicare messaggi e richiedere informazioni sui topic esistenti.
- **Topic.java**: Gestisce un singolo topic, includendo publisher, subscriber 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.

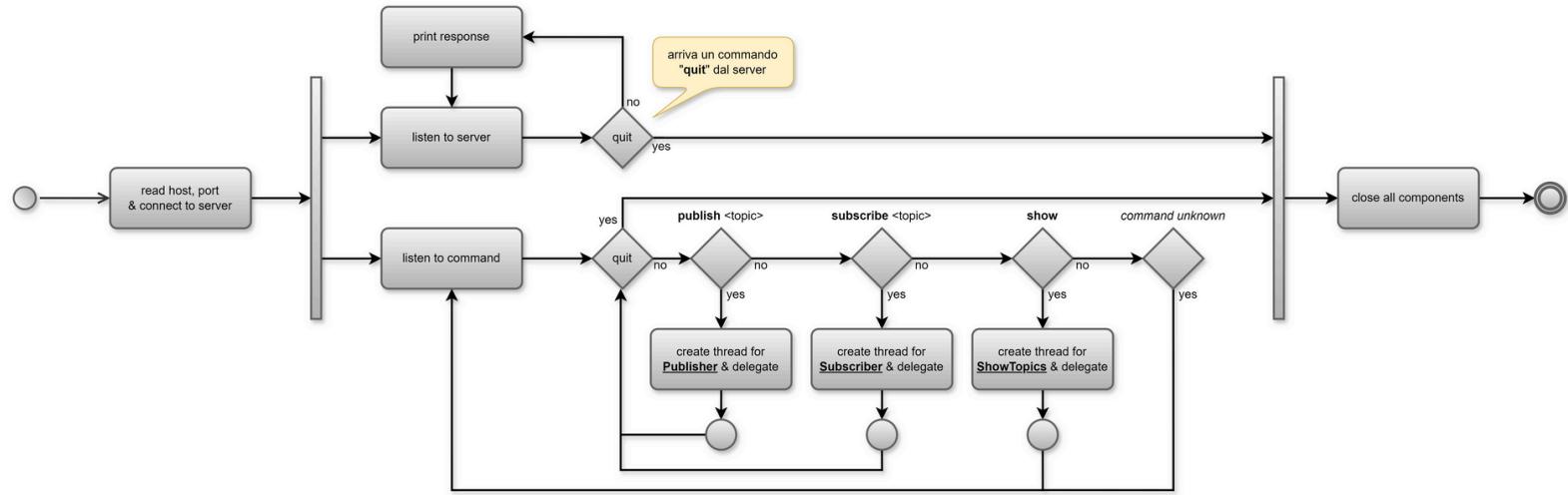
COMPONENTI



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 tra un client e un server, utilizzando un socket per la comunicazione bidirezionale. All'avvio, il programma verifica che i parametri host e porta siano forniti correttamente. Una volta stabilita la connessione con il server tramite **Socket socket = new Socket(host, port)**, il client stampa un messaggio di conferma. L'interazione principale avviene in due thread distinti: uno per l'ascolto dei messaggi dal server e l'altro per la gestione dell'input utente.

Il thread principale rimane responsabile dell'interazione con l'utente tramite **Scanner**, ricevendo comandi come *publish*, *subscribe*, *show* e *quit*. Separando l'ascolto del server in un thread dedicato (**serverListener**), la classe evita il blocco dell'esecuzione dovuto all'uso di **System.in**, che è bloccante. Questo design permette al client di ricevere messaggi dal server anche mentre l'utente sta inserendo comandi. Il thread di ascolto riceve i messaggi dal server, li stampa, e gestisce eventuali disconnessioni, come quando il server invia un comando *quit* o il socket viene chiuso. In tal caso, il thread viene interrotto e il client termina l'esecuzione.

Per ogni comando inserito dall'utente, il client gestisce il comando corrispondente in un nuovo thread separato. Per *publish* e *subscribe*, ad esempio, vengono creati nuovi socket dedicati e thread separati (es. *Publisher*, *Subscriber*), che eseguono rispettivamente la pubblicazione di messaggi o l'iscrizione ai topic.

Il comando *show* utilizza un **CountDownLatch** per sincronizzare l'esecuzione del thread che visualizza i topic disponibili, garantendo che il flusso principale del programma non prosegua finché il comando non è completato. Il client gestisce attentamente la chiusura delle risorse: dopo il comando *quit*, chiude il socket principale e tutti i thread attivi, evitando l'uso di risorse già chiuse con controlli come **socket.isClosed()**.

PUBLISHER

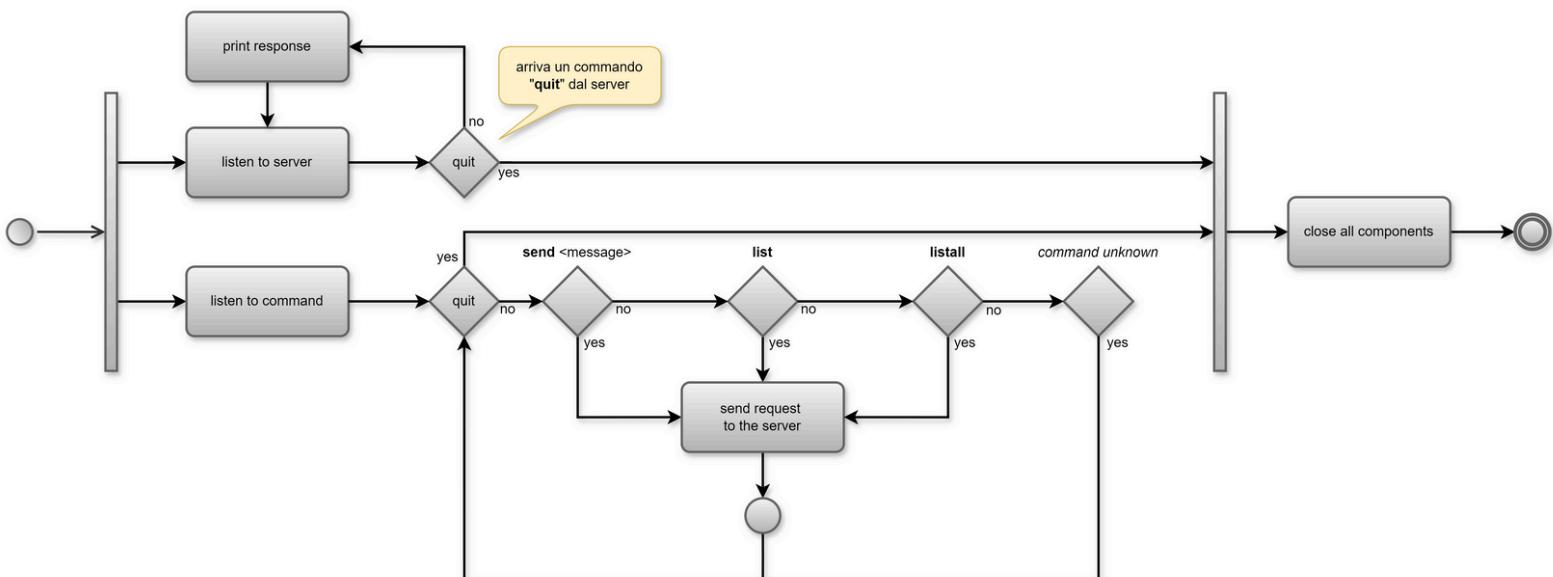
La classe **Publisher** gestisce l'invio di messaggi da un client verso un topic specifico all'interno di un sistema di comunicazione client-server. Il suo funzionamento è orchestrato tramite l'uso di socket per la connessione con il server e thread per la gestione concorrente. All'avvio, il costruttore inizializza il socket e il topic su cui il Publisher opererà. Il comando **publish <topic>** viene immediatamente inviato al server per registrare il client come publisher per quel topic.

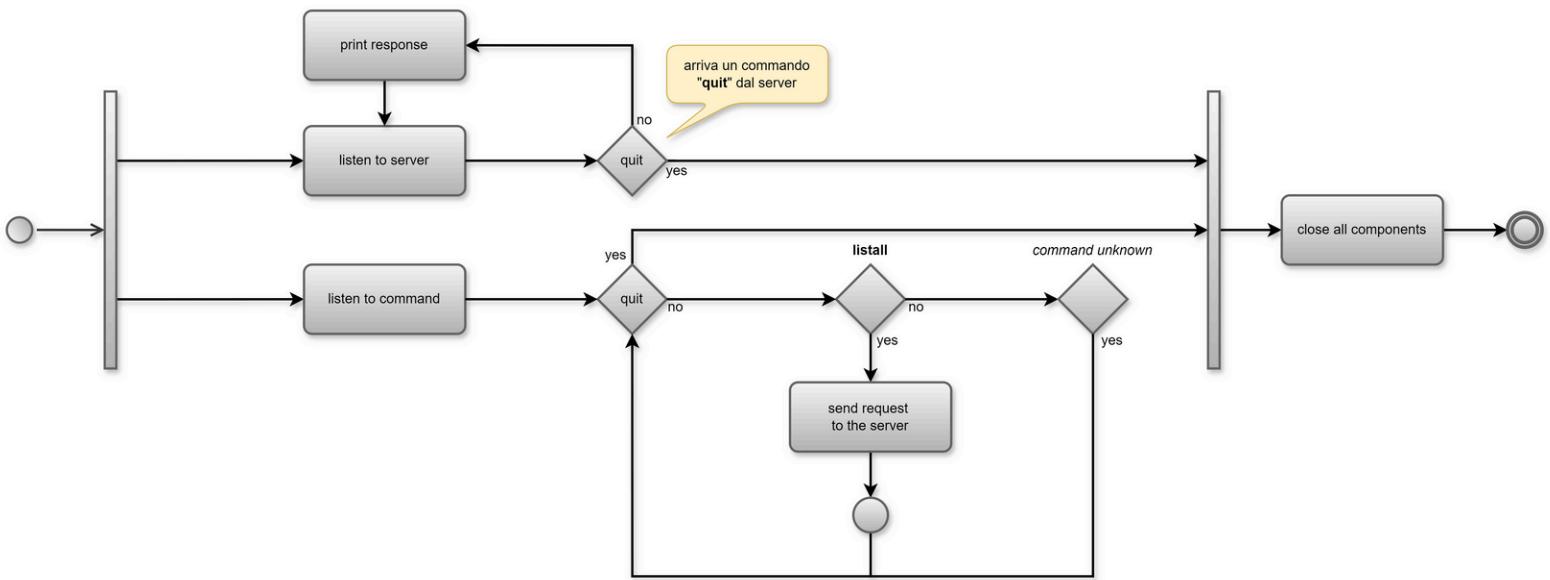
La classe crea due thread: uno per gestire l'ascolto dei messaggi provenienti dal server e uno per l'interazione con l'utente. Il thread di ascolto (**serverListener**) è progettato per ascoltare i messaggi dal server in modo continuo, evitando che l'esecuzione del programma rimanga bloccata durante l'attesa dell'input utente. Durante il ciclo di ascolto, se il server invia un comando **quit** o se il socket viene chiuso, il Publisher interrompe la connessione. Questo thread gestisce anche eventuali errori di comunicazione, come la perdita della connessione, e avvisa l'utente.

Il ciclo principale della classe consente all'utente di inviare comandi: **send <message>** per pubblicare un messaggio sul topic specificato, **list** per elencare i messaggi del client, e **listall** per ottenere tutti i messaggi del topic da vari publisher. Durante l'esecuzione, viene controllato lo stato del socket per assicurarsi che non sia chiuso prima di inviare i comandi, evitando eccezioni di input/output.

Il comando **quit** permette di terminare la pubblicazione, interrompendo i thread attivi e chiudendo le risorse in uso. Il Publisher utilizza **volatile** per la variabile **quiting**, garantendo che i thread condividano correttamente lo stato di terminazione. Una volta ricevuto il comando di disconnessione, il thread di ascolto si interrompe e il Publisher chiude il socket, chiudendo tutte le risorse associate.

activity diagram: classe **publisher**





activity diagram: classe **subscriber**

SUBSCRIBER

La classe **Subscriber** rappresenta un client che si scrive a un topic su un server per ricevere i messaggi pubblicati da altri client. Essa implementa l'interfaccia **Runnable** e gestisce la sua esecuzione su un thread separato. Al momento della creazione, un oggetto **Subscriber** prende un socket e un nome di topic come parametri, utilizzati per stabilire la connessione al server e per iscriversi al topic specifico. La variabile **quiting** viene usata per controllare quando il thread del subscriber deve terminare.

Quando viene eseguito il metodo **run**, il subscriber invia al server il comando **subscribe** per iscriversi al topic desiderato tramite il socket. Successivamente, viene creato un thread separato chiamato **serverListener**, il cui compito è quello di ascoltare i messaggi in arrivo dal server. Questo thread utilizza un Scanner per leggere i dati dal socket e continua ad ascoltare finché il subscriber non termina l'esecuzione. Se il server invia un messaggio **quit** o se il socket viene chiuso, il thread interrompe l'ascolto e la variabile **quiting** viene impostata a **true**.

Nel ciclo principale del thread, il subscriber attende l'input dell'utente, permettendo l'invio di comandi come **listall**, per ottenere una lista di tutti i messaggi nel topic, o **quit**, per disconnettersi dal server. Se l'utente invia un comando non riconosciuto, viene stampato un messaggio di errore.

Alla fine, il thread attende la terminazione del **serverListener** tramite il metodo **join**, assicurandosi che tutti i messaggi dal server siano stati ricevuti correttamente prima della chiusura. Infine, tutte le risorse, come il **PrintWriter** e il **Scanner**, vengono chiuse.

SHOWTOPICS

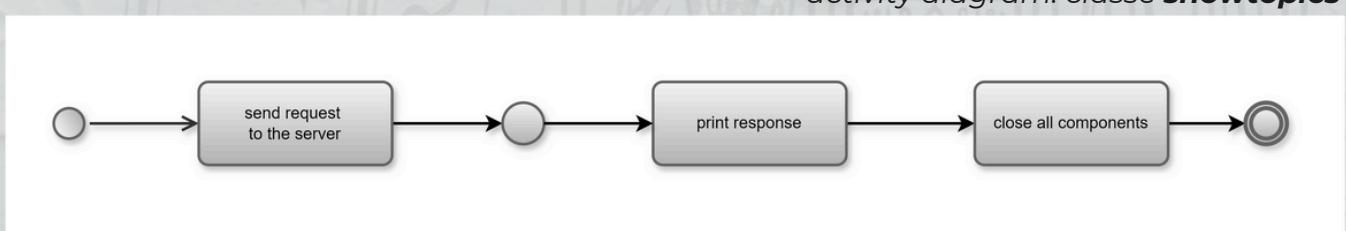
La classe **ShowTopics** implementa l'interfaccia **Runnable** ed è progettata per essere eseguita su un thread separato. Il suo compito è quello di richiedere al server la lista dei topic disponibili e mostrarla al client. Al momento dell'istanziazione, la classe riceve un **Socket** per la comunicazione con il server e un oggetto **CountDownLatch**, utilizzato per sincronizzare il thread con il flusso principale del client. Il CountDownLatch permette di segnalare quando l'operazione di richiesta e visualizzazione dei topic è completata, in modo che il thread principale possa riprendere l'esecuzione.

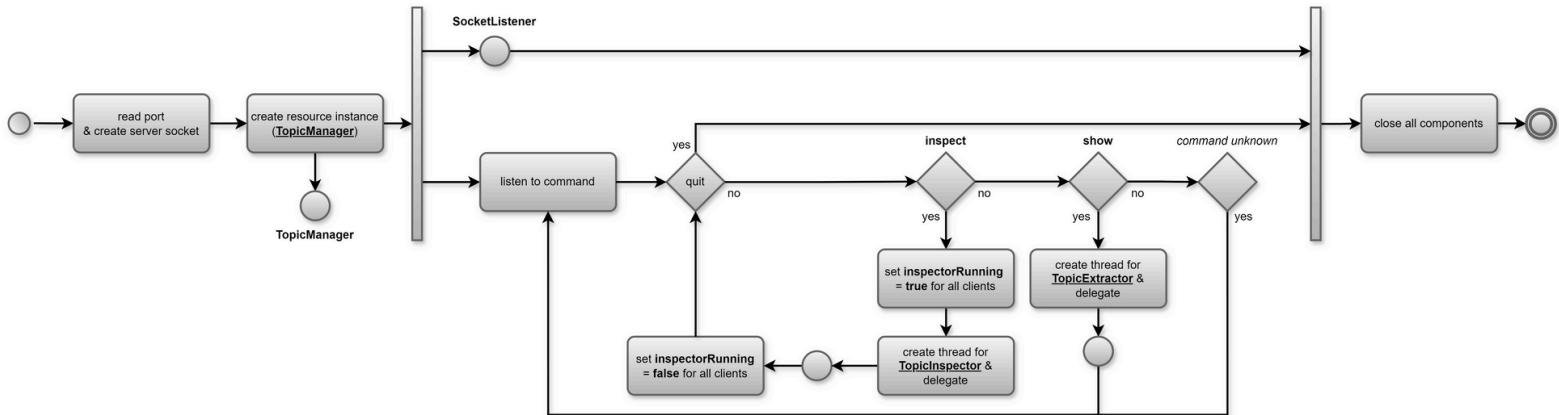
Nel metodo **run**, che viene eseguito quando il thread viene avviato, la classe utilizza un **PrintWriter** per inviare la richiesta "show" al server tramite il socket, richiedendo al server la lista dei topic. Successivamente, un **Scanner** viene utilizzato per leggere le risposte dal server, riga per riga, attraverso lo stesso socket. Ogni riga rappresenta un topic disponibile, e questi vengono stampati direttamente nella console del client.

Il ciclo di lettura continua fino a quando il server non invia un segnale specifico, "SHOWTOPICbreak", che indica la fine della lista di topic. Una volta ricevuto questo segnale, il ciclo viene interrotto e le risorse associate alla comunicazione con il server, come il Scanner e il PrintWriter, vengono chiuse correttamente per evitare perdite di risorse.

Infine, viene utilizzato il **CountDownLatch** per segnalare al thread principale che l'operazione è stata completata. Il metodo **countDown()** decrementa il contatore del **latch**, consentendo al thread principale di riprendere il controllo e procedere con altre operazioni. La gestione delle eccezioni viene eseguita tramite un blocco **try-catch**, che intercetta eventuali problemi di input/output durante la comunicazione con il server, come la perdita di connessione o problemi nel flusso di dati.

activity diagram: classe **showtopics**





activity diagram: classe **server**

SERVER

La classe **Server** rappresenta il punto di ingresso per avviare il server e gestire le connessioni dei client. Il metodo **main** accetta la porta del server come argomento dalla linea di comando, assicurandosi che l'argomento sia fornito correttamente. Se la porta è corretta, viene creato un oggetto **ServerSocket** che si mette in ascolto delle connessioni in arrivo. Un componente centrale della classe è il **TopicManager**, che funge da risorsa condivisa tra tutti i thread e gestisce la creazione, la pubblicazione e la sottoscrizione dei topic.

Per gestire le connessioni dei client in maniera separata, viene creato e avviato un thread **SocketListener**, il cui compito è quello di accettare le connessioni dei client e creare thread individuali per ogni client connesso. Il **SocketListener** gestisce questi thread client in modo efficiente, permettendo una gestione concorrente delle richieste di pubblicazione e sottoscrizione ai topic.

Il server, nel ciclo principale, accetta comandi dall'utente attraverso **System.in**. L'utente può inviare i comandi **show** per visualizzare i topic disponibili, **inspect** per ispezionare i messaggi di un topic specifico, o **quit** per fermare il server. Quando il comando *show* viene inserito, viene avviato un thread **TopicExtractor** che estrae e visualizza i topic correnti tramite il **TopicManager**. Se il comando *inspect* viene inserito, viene avviato un thread **TopicInspector** che ispeziona un topic. Durante l'ispezione, il metodo **setInspectorRunningForAllClients(true)** sospende temporaneamente le attività dei client, utilizzando un **latch** per sincronizzare il termine dell'ispezione e il metodo **setInspectorRunningForAllClients(false)** per riattivare le attività dei client.

Quando l'utente inserisce il comando **quit**, il server chiude correttamente il ciclo principale, interrompendo il thread **SocketListener** e attendendo la sua terminazione. Le risorse, come il **ServerSocket**, vengono chiuse correttamente e le connessioni terminate in modo sicuro.

SOCKETLISTENER

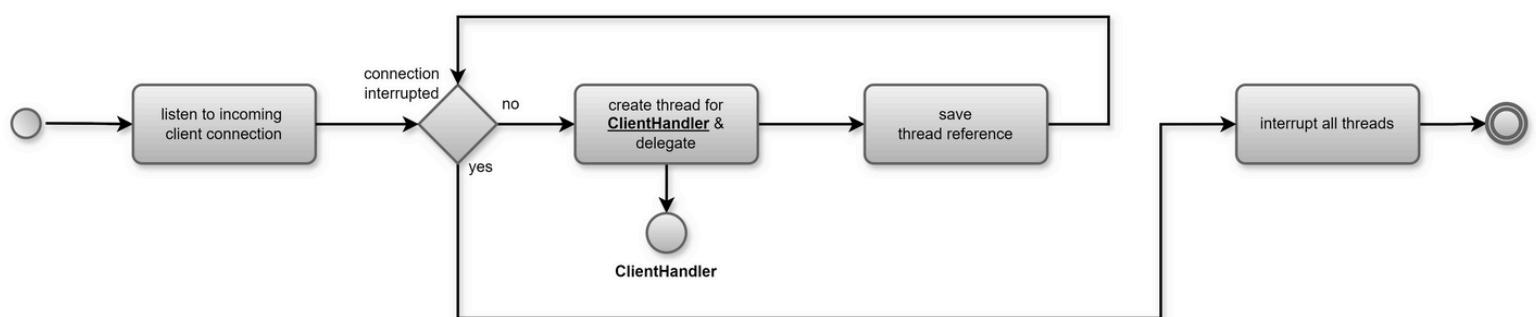
La classe **SocketListener** implementa l'interfaccia **Runnable** ed è responsabile di gestire le connessioni dei client che si collegano al server tramite un **ServerSocket**. Il suo compito principale è accettare nuove connessioni in entrata e creare un thread separato per ciascun client utilizzando un **ClientHandler**, che gestisce le richieste inviate dai client. Il costruttore della classe accetta un'istanza di **ServerSocket** e un **TopicManager**, che rappresenta la risorsa condivisa per gestire i topic a cui i client possono iscriversi o su cui possono pubblicare.

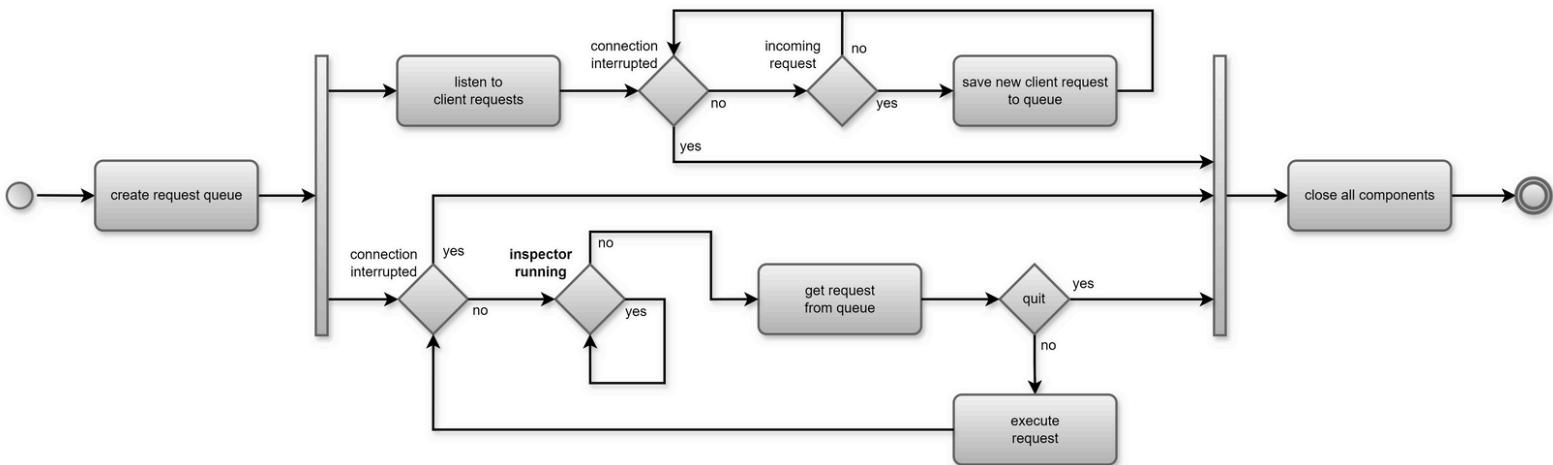
Nel metodo run, il server si mette in ascolto delle connessioni in entrata all'interno di un ciclo, utilizzando **accept()** per gestire ogni client. Ogni volta che viene stabilita una nuova connessione, il SocketListener crea un nuovo ClientHandler, encapsulato in un thread separato, che viene poi avviato. Ogni thread e socket associato a un client viene memorizzato in due mappe: **children** per associare thread e socket, e **clientHandlers** per mappare thread a ClientHandler. Questo garantisce una gestione centralizzata delle connessioni, permettendo di monitorare e gestire ogni client attivo.

Il metodo **run** include un timeout del socket per gestire eventuali blocchi e prevenire attese infinite. Se viene rilevata una **SocketTimeoutException**, il ciclo continua a tentare nuove connessioni. Alla fine del ciclo, quando il server viene interrotto, il ServerSocket viene chiuso e tutti i thread dei client attivi vengono terminati. Questo avviene inviando un messaggio di "quit" tramite il socket associato a ogni client e interrompendo i rispettivi thread.

Un aspetto importante è la gestione dell'ispezione dei topic, eseguita tramite il metodo **setInspectorRunningForAllClients**, che imposta lo stato di esecuzione dell'Inspector su tutti i client connessi. Questo metodo consente di sospendere temporaneamente l'esecuzione dei ClientHandler durante l'ispezione di un topic.

activity diagram: classe **socketlistener**





activity diagram: classe **clienthandler**

CLIENTHANDLER

La classe **ClientHandler** implementa l'interfaccia **Runnable** ed è responsabile della gestione delle comunicazioni tra il server e un singolo client. Ogni connessione client è gestita da un thread separato, il cui compito è ricevere richieste dal client tramite un **Socket** e interagire con un oggetto **TopicManager**, che coordina le operazioni legate alla gestione dei topic di pubblicazione e sottoscrizione. Il costruttore della classe accetta come parametri il socket del client e una risorsa condivisa TopicManager, che consente la gestione centralizzata delle operazioni sui topic.

Uno degli aspetti fondamentali della classe è la gestione concorrente delle richieste client. La classe utilizza una coda **BlockingQueue** per accumulare le richieste in arrivo, che vengono lette da un thread separato (**requestReader**) e inserite nella coda. Il metodo **run** esegue un ciclo principale che prende le richieste dalla coda e le processa. Ogni richiesta viene elaborata in modo sincrono, con un blocco **synchronized** per garantire che il thread non esegua operazioni mentre un eventuale **TopicInspector** è attivo, utilizzando il flag **inspectorRunning** per sospendere temporaneamente l'esecuzione e invocando il metodo **wait** fino al termine dell'ispezione.

Le richieste dei client possono includere operazioni come *publish*, *subscribe*, *list*, *listall*, *send* e *show*. Ogni richiesta è processata in base alla sua tipologia, e l'interazione con il client avviene tramite un **PrintWriter**, che consente di inviare le risposte al client. Ad esempio, in caso di una richiesta *publish*, il ClientHandler aggiunge un publisher a un topic specificato, mentre in caso di una richiesta *subscribe*, il client viene iscritto a un topic per ricevere i messaggi pubblicati da altri publisher.

Il meccanismo di chiusura è gestito tramite il flag **closed**, che permette di terminare il ciclo principale e chiudere le risorse del socket, come il flusso di input/output e il thread attivo.

TOPICEXTRACTOR

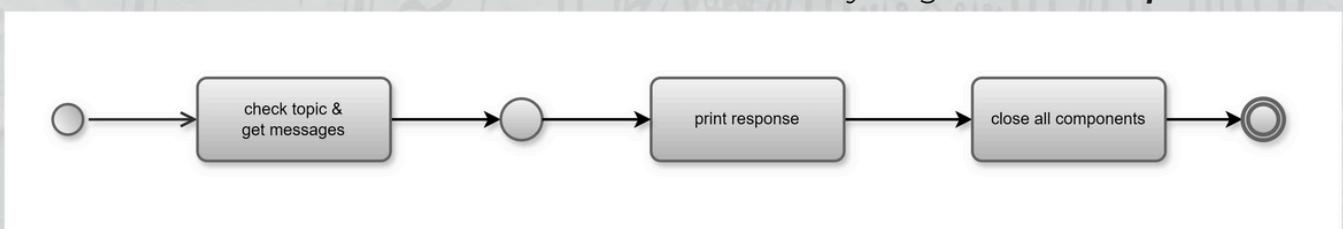
La classe **TopicExtractor** implementa l'interfaccia **Runnable** ed è utilizzata per estrarre e stampare i nomi dei topic gestiti dal **TopicManager**. Questa classe viene eseguita su un thread separato per evitare di bloccare il flusso principale del programma durante l'estrazione delle informazioni 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.

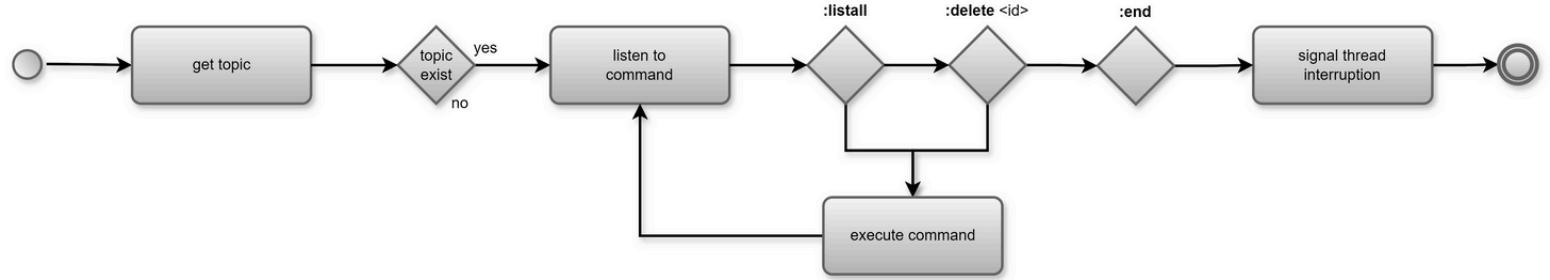
Quando il metodo **run** viene chiamato, il thread esegue la logica di estrazione dei nomi dei topic dal **TopicManager**. Il metodo **getTopicNames** viene invocato sul TopicManager, il quale restituisce una lista contenente i nomi di tutti i topic attivi nel sistema. Se la lista è vuota, significa 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 viene utilizzata per avviare una sessione interattiva di ispezione e modifica dei messaggi di un topic. Il suo costruttore riceve un oggetto **TopicManager**, che è la risorsa condivisa utilizzata per gestire i topic, e un **CountDownLatch**, che viene impiegato per sincronizzare il termine della sessione con il thread principale.

Il metodo **run** gestisce l'intero ciclo di interazione dell'utente per ispezionare un topic specifico. Dopo aver richiesto il nome del topic da ispezionare tramite input da tastiera, il programma tenta di recuperare il topic corrispondente dal TopicManager. Se il topic non viene trovato, il **latch** viene decrementato e il thread termina immediatamente. Se il topic esiste, l'utente entra in una sessione interattiva dove può eseguire operazioni come elencare tutti i messaggi associati al topic, eliminare messaggi specifici identificati da un ID, o terminare la sessione con il comando **:end**.

Durante la sessione interattiva, l'utente può utilizzare il comando **:listall** per visualizzare tutti i messaggi inviati al topic. Se non ci sono messaggi, viene visualizzato un messaggio che notifica l'assenza di contenuti. Se l'utente inserisce il comando **:delete** seguito da un ID messaggio, il sistema tenterà di eliminare il messaggio corrispondente. In caso di ID non valido, viene mostrato un messaggio di errore. La sessione termina quando l'utente inserisce il comando **:end**, interrompendo il ciclo di interazione.

Il **latch** è fondamentale per segnalare al thread principale che il lavoro del TopicInspector è terminato. Questo meccanismo di sincronizzazione consente al server di continuare ad eseguire altre operazioni senza interferire con l'ispezione interattiva del topic.

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 impediva 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. Ritorno del flusso alla classe Client dopo il comando 'show':

- Problema: Dopo l'esecuzione del comando show, che elenca i topic disponibili, il client non tornava immediatamente al flusso principale per accettare altri comandi. Il thread che eseguiva il comando show bloccava il flusso principale fino al completamento, causando un'interruzione del ciclo principale.
- Soluzione: È stato utilizzato un meccanismo di **CountDownLatch** per sincronizzare il thread che esegue il comando show con il thread principale del client. Una volta che il comando show termina la sua esecuzione e visualizza tutti i topic, il **latch** viene decrementato, segnalando al thread principale che può riprendere l'esecuzione. Questo assicura che il flusso torni correttamente alla classe Client senza blocchi.

3. 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.

PROBLEMI RISCONTRATI E SOLUZIONI

4. 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
```