

# PUBLISH / SUBSCRIBE

LABORATORIO DI SISTEMI OPERATIVI  
A.A. 2023-24

30/09/2024

**Email del referente:**  
[jacob.angeles@studio.unibo.it](mailto:jacob.angeles@studio.unibo.it)

**Nome del Gruppo :**

ACD2024

JACOB ANGELES  
1114812

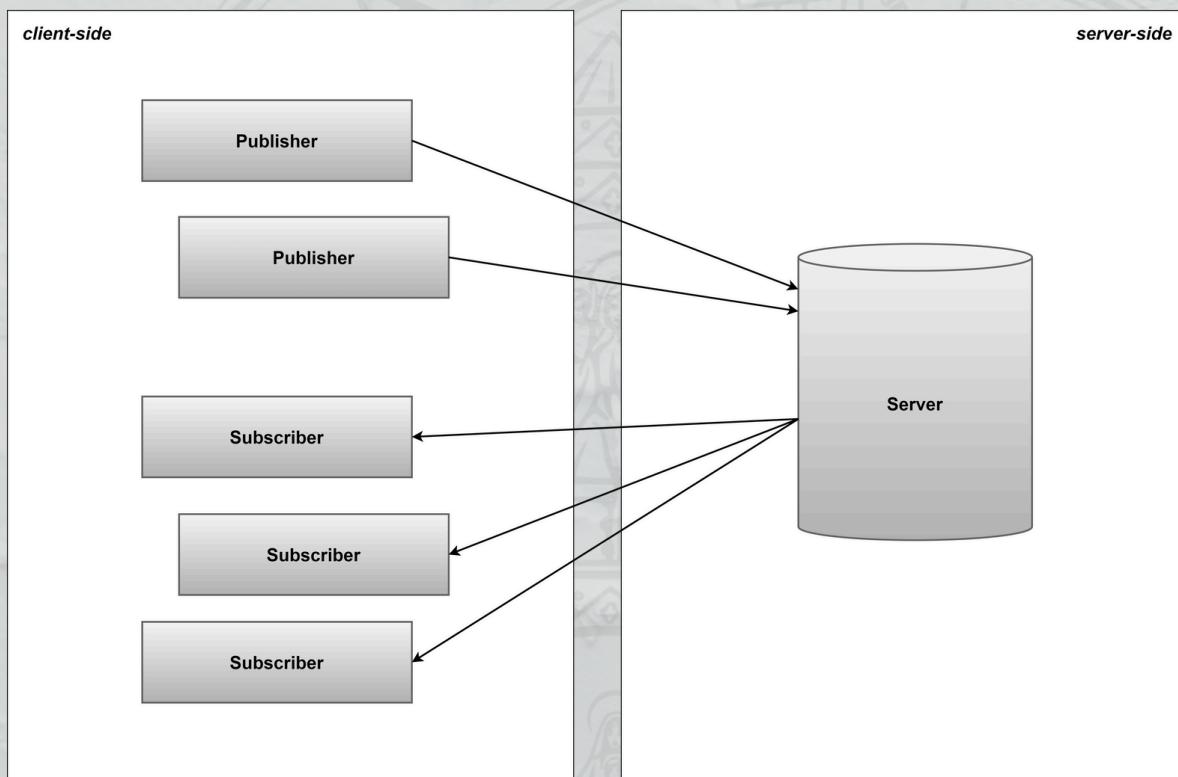
ANDREA CROCI  
1054710

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.



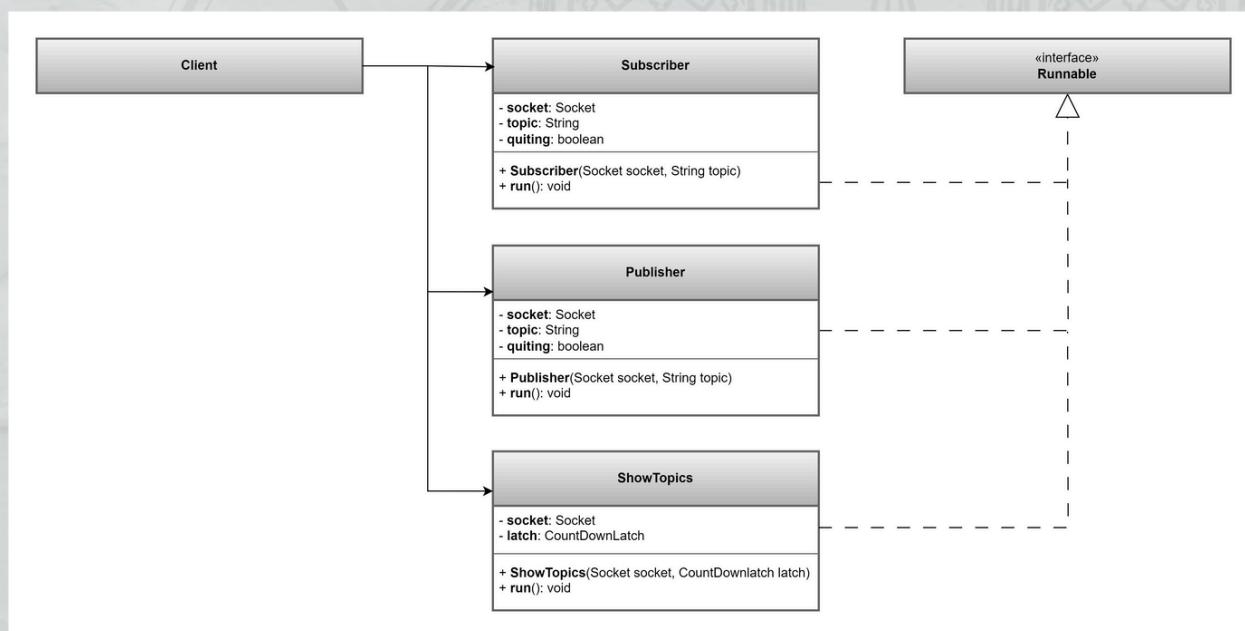
# COMPONENTI

## 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 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.

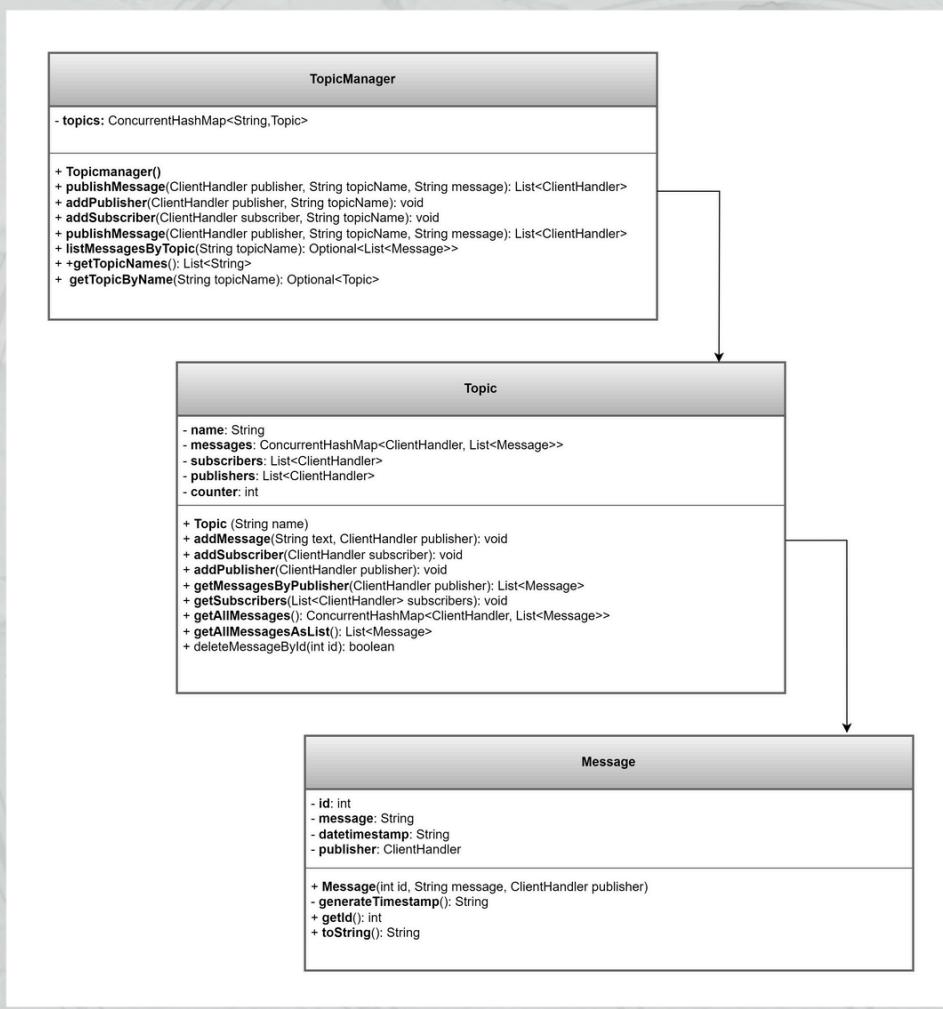
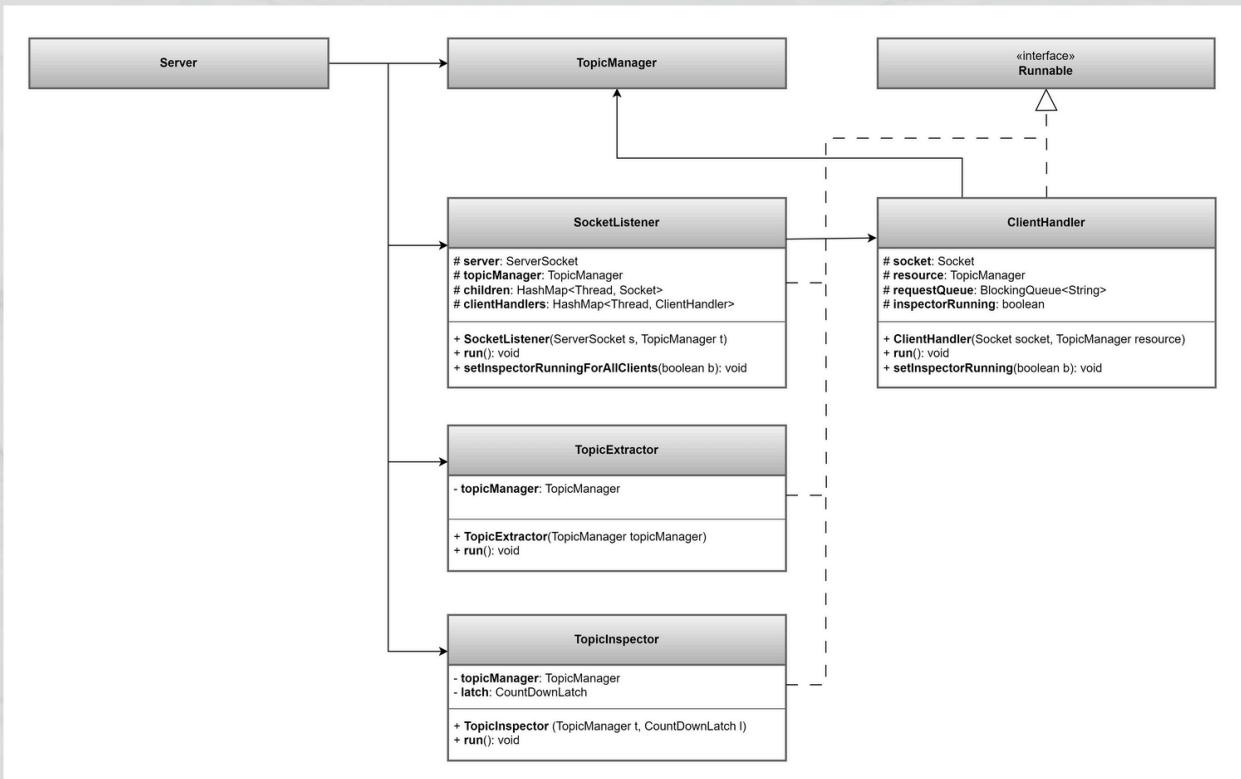
Le classi principali sono:

- **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:

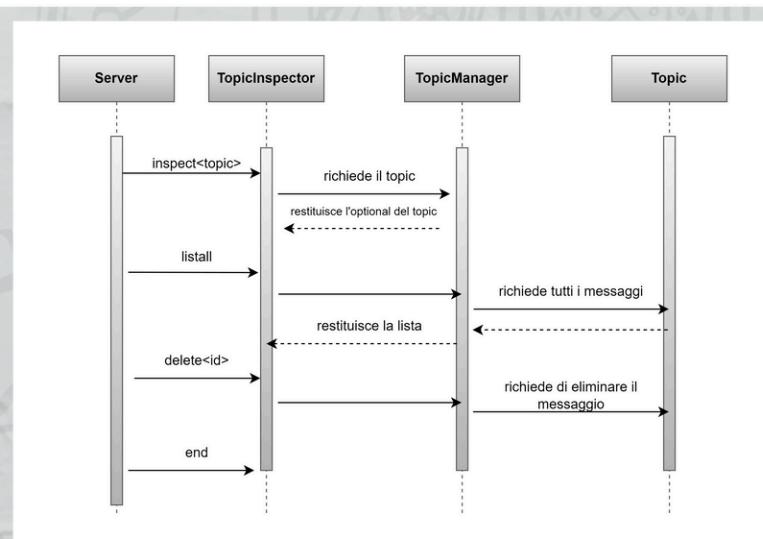
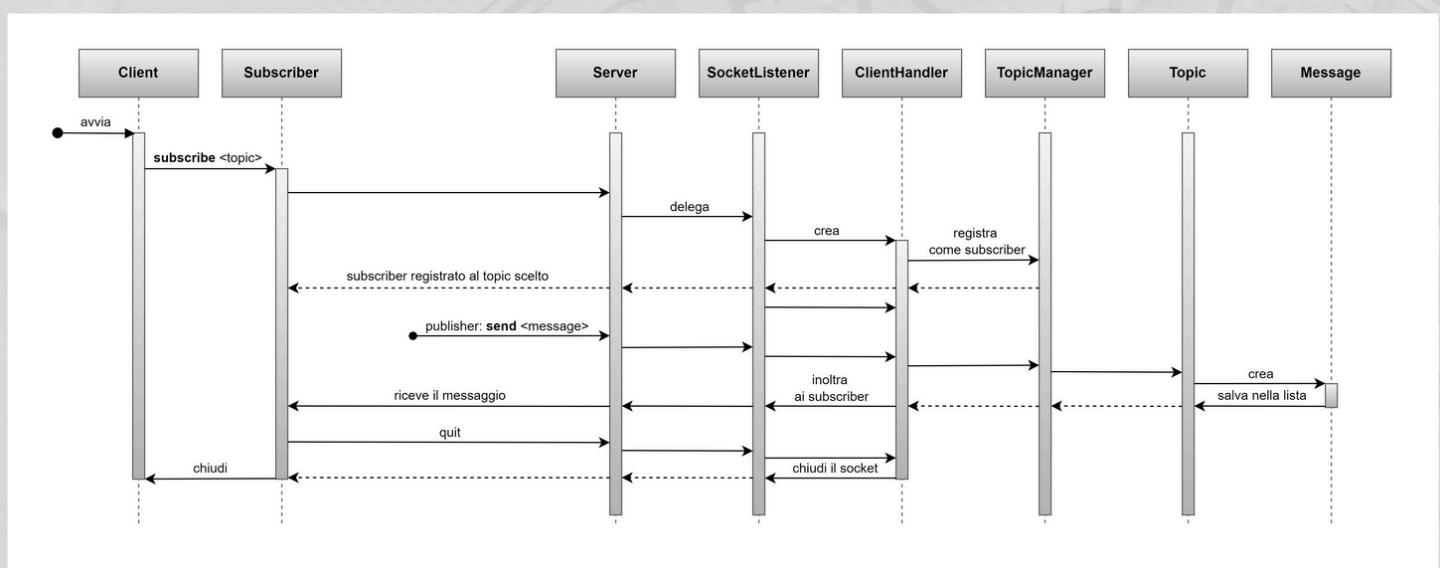
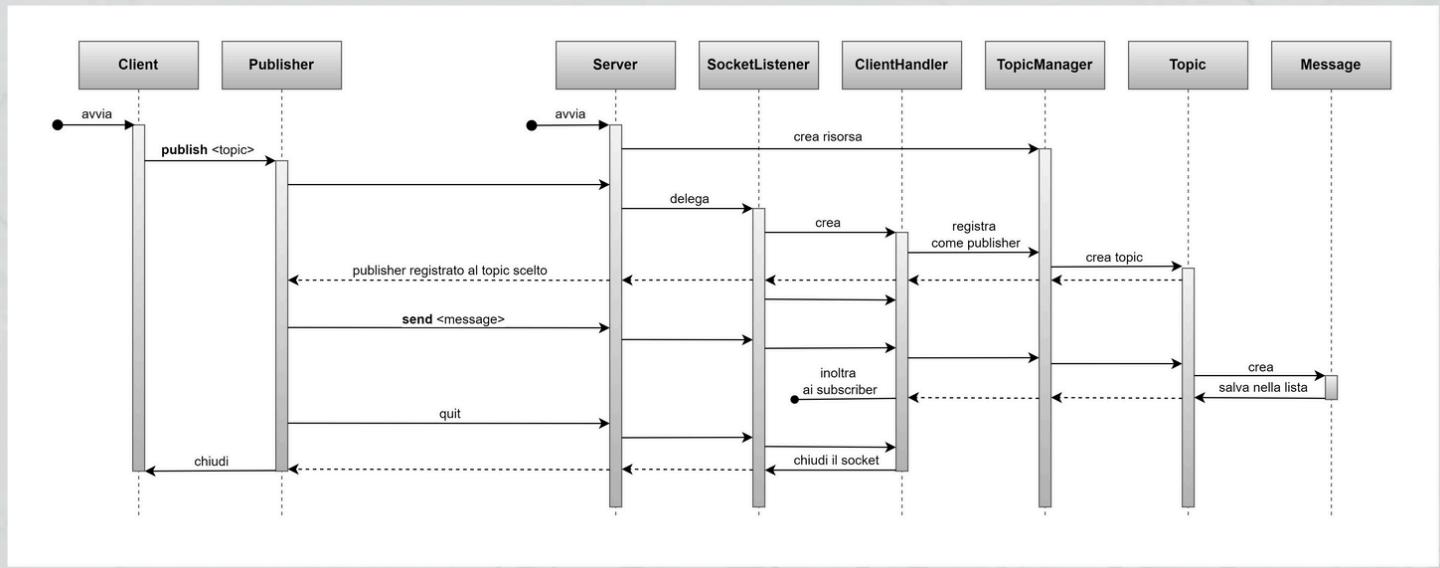
- **TopicManager.java**: Gestisce tutti i topic presenti nel sistema. Coordina i publisher e i subscriber, permettendo di aggiungere nuovi topic, iscrivere utenti, pubblicare messaggi e richiedere informazioni sui topic esistenti.
- **Topic.java**: Gestisce un singolo topic, includendo publisher, subscriber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



# 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 un groupchat 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
```