Scheda 1

**Laboratorio di Sistemi Operativi A.A. 2023-24**

# Nome del gruppo: Byte Crew.

Email referente del gruppo: diego.gallini3@studio.unibo.it

Il gruppo di lavoro è composto dalle seguenti persone.

| **Nome** | **Cognome** | **Matricola** |
| --- | --- | --- |
| Francesco | Forlani | 0001069583 |
| Andrea | Caselli | 0001077887 |
| Nicolas | Cola | 0001080556 |
| Diego | Gallini | 0001082285 |

# **DESCRIZIONE DEL PROGETTO:**

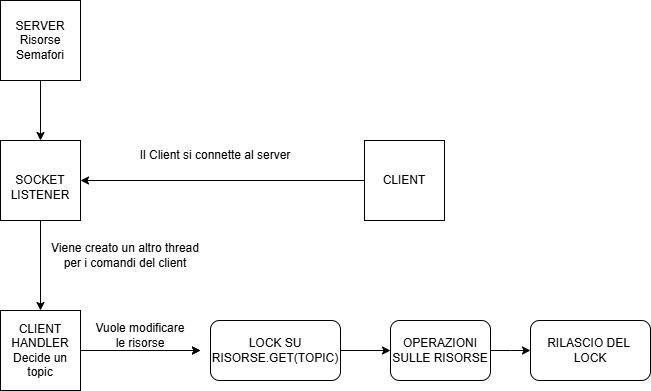
**Architettura generale:**

L’applicazione è composta da una struttura Client - Server, nella quale vengono gestiti, su thread separati, i comandi impartiti da questi. La funzione principale dell’applicazione è quella di offrire ai Client connessi la possibilità di mandare messaggi o riceverli su un Topic di loro interesse. Questi vengono memorizzati su una Hashmap, componente di una classe più ampia, chiamata Resource, nella quale alla chiave Topic corrisponde la lista di messaggi corrispondenti.

La struttura dati in questione viene inizializzata come statica nel momento in cui viene avviato il Server e le operazioni del Client su questa avvengono in una classe dedicata, alla quale la struttura dati viene passata come argomento.

Con un meccanismo simile viene gestita la mutua esclusione dei thread sulle risorse, viene infatti utilizzata una HashMap avente come chiavi i nomi dei Topic esistenti e per ognuno di essi un Lock che viene acquisito e rilasciato per operare sulla struttura dati condivisa.

Di seguito uno schema che semplifica il funzionamento dell’applicazione.

****

**Descrizione dettagliata:**

**Client, Server, relativa suddivisione dei compiti e sottocomponenti:**

Il sistema client-server è basato su una comunicazione asincrona, gestita tramite socket e thread. L'architettura prevede un server centrale e vari client che insieme garantiscono l'interazione tra utenti e risorse condivise.

**Server**

La classe Server gestisce l'avvio e il funzionamento di un server che fornisce accesso ai topic e messaggi contenuti nella classe Resource. Utilizza connessioni socket e thread per supportare la concorrenza e comunicare col client, con un'architettura che combina una console per i comandi del server e un listener separato per le connessioni client.

Sottocomponenti di Server:

Le sottocomponenti principali sono:

1. ***Semaphores***: mappa che associa a ciascun topic un lock ReentrantReadWriteLock per controllare l'accesso concorrente. Permette di gestire accessi in lettura multipli, ma riserva il lock in scrittura per operazioni esclusive come la rimozione di messaggi.
2. ***Topics***: oggetto Resource che rappresenta l'insieme di topic e messaggi e tiene traccia degli iscritti, i “subscribers”.
3. ***inspectLocks***: mappa che tiene traccia dello stato di ispezione dei topic. Se un topic è in fase di ispezione, il relativo valore booleano è true; altrimenti, è false. I valori vengono utilizzati come Flag per notificare il client nel caso in cui tenti di compiere azioni sul topic di interesse mentre una sessione di ispezione è attiva.
4. ***userInput***: oggetto Scanner che legge i comandi inseriti dall'amministratore del server tramite la console.
5. ***ServerSocket***: rappresenta il socket server che ascolta le connessioni in entrata sulla porta specificata.
6. ***SocketListener***: classe che quando avviata entra in un ciclo di ascolto che attende nuove connessioni tramite serverSocket.accept(). Questo ciclo continua finché il server è attivo.
7. ***ServerThread***: quando Server viene avviato, delega ad un altro thread, ServerThread, la gestione di tutte le connessioni del client. A questo viene passato come parametro nel costruttore un oggetto SocketListener e perciò segue il comportamento descritto in questa classe.

#### Metodi principali di Server:

#### **1. *main(String[] args)***

Il metodo principale avvia il server seguendo questi passaggi:

* Verifica che l’argomento args contenga il numero di porta, necessario per aprire il ServerSocket;
* inizializza un listener su una porta specifica e avvia un thread (serverThread) per gestire tutte le connessioni client tramite la classe SocketListener;
* resta in attesa di comandi nella console del server tramite processServerCommands();
* al termine dei comandi, interrompe e attende la terminazione di serverThread, chiudendo il socket del server.

#### **2. *processServerCommands()***

Questo metodo gestisce i comandi inseriti dall'operatore del server:

* ***quit***: chiude il server;
* ***show***: mostra tutti i topic presenti in Resource;
* ***inspect <topic>***: permette l'ispezione di un topic specifico, con accesso in modalità scrittura. Prima di chiamare manageInspect, acquisisce il lock in scrittura sul topic, assicurandosi che nessun altro possa modificarlo durante l'ispezione.

#### **3. *manageInspect(String topic)***

Questo metodo consente di eseguire comandi specifici all'interno di una "sessione di ispezione" su un topic:

* ***end***: termina la sessione di ispezione e sblocca il topic rilasciando il lock e segnala la fine della sessione di ispezione impostando inspectLocks.put(topic, false);
* ***listall***: recupera e stampa tutti i messaggi associati al topic;
* ***delete <id>***: elimina il messaggio con l’ID specificato dal topic. Se l’ID è valido, viene rimosso e viene stampato un messaggio di conferma; altrimenti, mostra un errore.

Durante l'esecuzione di manageInspect, la sessione rimane attiva finché non si riceve il comando “end”. Questo loop consente all'amministratore di eseguire più operazioni sul topic in ispezione, finché non termina volontariamente la sessione.

In sintesi, la classe Server offre un’interfaccia console per l’amministratore e un listener per le connessioni client, utilizzando lock e thread per mantenere una gestione sicura e organizzata dell'accesso ai topic e messaggi.

**Client**

La classe Client consente agli utenti di connettersi al server e interagire con i topic e i messaggi. Utilizza socket per la comunicazione e thread separati per la gestione dell'invio e della ricezione dei messaggi, permettendo una comunicazione bidirezionale con il server.

#### Sottocomponenti di Client:

Le sottocomponenti principali sono:

* ***Socket***: Rappresenta la connessione tra il client e il server. Il client si collega al server tramite l’indirizzo IP e la porta specificati, consentendo una connessione bidirezionale per inviare e ricevere messaggi.
* ***Sender***: È un thread dedicato esclusivamente all’invio delle richieste effettuate dal client al server. Il thread Sender legge l’input dell’utente e lo trasmette al server attraverso il socket.
* ***Receiver***: È un thread dedicato alla ricezione delle risposte alle richieste inviate dal server. Receiver resta in ascolto delle risposte del server e le mostra nella console del client, aggiornando l'utente sui messaggi e sulle informazioni relative ai topic.
* ***Thread di gestione (Sender e Receiver)***: I thread Sender e Receiver vengono avviati e gestiti dal thread principale della classe Client. L’uso di questi due thread consente al client di inviare e ricevere messaggi simultaneamente, mantenendo la sessione attiva per tutta la durata della connessione. La connessione si chiude automaticamente quando uno dei due thread termina.

#### 

#### 

#### Metodi principali di Client:

1. ***main(String[] args)***Il metodo principale avvia il client e gestisce la connessione con il server tramite i seguenti passaggi:
   * Verifica la presenza dei parametri host e port richiesti per stabilire la connessione al server. Se i parametri sono assenti, visualizza un messaggio di errore e termina l’esecuzione.
   * Inizializza un Socket con l'indirizzo, “host”, e la porta, “port”, per connettersi al server. In caso di successo, il client conferma la connessione stabilita.
   * Avvia i thread di gestione della comunicazione:
     + ***Sender***: avvia un thread dedicato per leggere i messaggi dell'utente e inviarli al server tramite il socket.
     + ***Receiver***: avvia un thread per leggere i messaggi dal server e visualizzarli per l'utente.
   * Attende che entrambi i thread terminino la loro esecuzione. Una volta che entrambi i thread sono terminati, chiude il socket e termina l'esecuzione del client.

In sintesi, la classe Client fornisce un'interfaccia di comunicazione per l'utente, stabilendo una connessione con il server e permettendo un'interazione continua con i topic e i messaggi. Utilizzando socket e thread per gestire l'invio e la ricezione di messaggi, la classe Client supporta una comunicazione efficiente e in tempo reale con il server.

**Classi fondamentali:**

Per alcune classi implementate nell’applicazione è necessaria una descrizione approfondita

**SocketListener:**

La classe SocketListener è progettata per gestire le connessioni dei client a un server multithread. Questa classe si occupa di accettare nuove connessioni in arrivo e di creare un thread dedicato per ciascun client, mantenendo la gestione di più client in parallelo. Rappresenta, quindi, un componente chiave per l'architettura del server, garantendo che ogni client connesso possa interagire con le risorse condivise in modo sicuro e concorrente. Nel costruttore vengono passate le strutture dati necessarie ad operare sulle risorse condivise e un ServerSocket che funge da punto di accesso principale per i client che desiderano connettersi al server.

Questa classe, che implementa runnable, una volta avviata entra in un ciclo infinito che perdura fintanto che Server è attivo e chiama serverSocket.accept() per aspettare nuove connessioni.

Quando una nuova connessione viene stabilita, viene creato un nuovo thread, al quale viene passato nel costruttore un nuovo oggetto ClientHandler. Questo nuovo thread si occuperà di gestire i comandi impartiti dal client e viene aggiunto ad una lista in modo da tenerne traccia.

**Resource:**

La classe Resource gestisce l’insieme dei topic e dei relativi messaggi, inoltre mantiene la lista dei client registrati come subscriber per notificare eventuali aggiornamenti. Questa classe rappresenta il nucleo delle risorse condivise tra il server e i client e fornisce i metodi necessari per aggiungere, visualizzare e gestire topic e messaggi.

Gli attributi principali sono:

* ***topics***: HashMap<String, ArrayList<Message>> – associa ogni topic a una lista di messaggi (Message) ad esso relativi. Questa struttura consente di organizzare e recuperare i messaggi in base al topic;
* ***subscribers***: List<ClientHandler> – lista dei client registrati come subscriber, gestita per notificare nuovi messaggi sui topic.
* ***topicCounters***: HashMap<String, AtomicInteger> – mantiene un contatore per ogni topic per tenere traccia del numero di messaggi e assegnare ID univoci ai nuovi messaggi.

In generale, i metodi della classe si occupano semplicemente di restituire delle risorse o modificarle, come aggiungere o eliminare messaggi.

Per quanto riguarda il metodo di notifica ai subscribers di un messaggio aggiunto sul topic di loro interesse: come scritto precedentemente questi vengono aggiunti ad una lista per tenerne traccia; quando un messaggio viene aggiunto su un topic, viene iterato sulla lista in questione e stampato il nuovo messaggio sulla console dei subscriber se il topic da loro selezionato è lo stesso su cui è stato aggiunto il nuovo messaggio.

**ClientHandler:**

La classe ClientHandler gestisce la connessione con un singolo client all'interno del server. Ogni client connesso al server viene servito da un'istanza di ClientHandler, che permette l’interazione con i topic e i messaggi tramite comandi specifici. ClientHandler implementa Runnable e viene eseguito in un thread separato per gestire ciascun client in modo concorrente.

#### Gli attributi principali sono:

* ***socket***: Socket – rappresenta la connessione al client, attraverso cui avviene la comunicazione tramite i metodi di input e output.
* ***topics***: Resource – rappresenta la risorsa condivisa che gestisce i topic e i messaggi. Viene utilizzata per registrare i nuovi messaggi e per recuperare i messaggi relativi a un determinato topic.
* ***chosenTopic***: String – memorizza il topic scelto dal client corrente. Viene aggiornato ogni volta che il client seleziona o modifica il topic su cui vuole operare.
* ***semaphores*:** HashMap <String, Reentrant Read Write Lock> – mappa che associa un ReentrantReadWriteLock a ciascun topic, consentendo l’accesso concorrente in lettura ma limitando le operazioni di scrittura. Questo assicura che le operazioni di lettura/scrittura sui topic siano gestite in modo sicuro.
* ***inspectLocks*:** HashMap <String, Boolean>– mappa che traccia lo stato di "ispezione" per ogni topic. Quando un topic è in fase di ispezione, il valore è true; questo previene la modifica da parte dei client finché l'ispezione non termina.
* ***from e to*: Scanner e PrintWriter** – gestiscono rispettivamente l’input e l’output con il client. Scanner legge i comandi dal client, mentre PrintWriter invia le risposte al client.
* **closed**: boolean – flag che indica lo stato della connessione con il client. Se impostato a true, indica che la sessione è terminata e ClientHandler chiude la connessione.

#### Metodi principali:

1. ***run()***: metodo principale eseguito dal thread di ClientHandler. Attende e gestisce i comandi dal client in un ciclo continuo. Quando il client invia un comando, run() lo passa a processClient(). Alla fine, chiude le risorse e termina la connessione.
2. ***processClient(String request)***: metodo che interpreta i comandi ricevuti dal client e ne gestisce la richiesta. In base al comando:
   * ***publish***: registra il client come publisher per un topic e chiama managePublisher().
   * ***subscribe***: registra il client come subscriber a un topic e chiama manageSubscriber().
   * ***quit***: imposta closed a true e termina la connessione.
   * ***show***: restituisce una lista di tutti i topic disponibili.
3. ***processPublisher(String[] parts)***: configura il client come publisher per un topic. Se il topic non esiste, lo aggiunge a Resource, crea un lock per il topic e imposta il flag di ispezione a false. Dopo l'inizializzazione, managePublisher() gestisce i comandi specifici del publisher.
4. ***processSubscriber(String[] parts)***: configura il client come subscriber per un topic esistente. Se il topic esiste, registra il client nella lista dei subscriber di Resource e passa il controllo a manageSubscriber(), altrimenti restituisce un errore.
5. ***managePublisher() e manageSubscriber()***: gestiscono rispettivamente i comandi specifici dei publisher e dei subscriber:
   * Publisher:
     + ***send***: invia un nuovo messaggio al topic selezionato.
     + ***list***: mostra i messaggi del topic inviati dal publisher corrente.
     + ***listall***: mostra tutti i messaggi del topic, indipendentemente dal publisher.
     + ***quit***: viene interrotta la sessione del client come publisher
   * Subscriber:
     + ***listall***: mostra tutti i messaggi relativi al topic selezionato.
     + ***quit***: viene interrotta la sessione del client come publisher
6. ***executeCommand(Runnable command, boolean isWrite)***: metodo asincrono che esegue i comandi di lettura/scrittura dei client in modo sicuro utilizzando i lock di lettura e scrittura. Prima di eseguire un comando, controlla se il topic è in ispezione e, in tal caso, attende la fine della sessione di ispezione per procedere.
7. ***sendMessage(String[] parts)***: aggiunge un nuovo messaggio al topic selezionato dal publisher. Utilizza topics per creare un nuovo Message e lo invia ai subscriber registrati.
8. ***listCurrentClientMessages() e listAllMessages()***: restituiscono rispettivamente i messaggi inviati dal publisher corrente e tutti i messaggi del topic selezionato.
9. ***getMessageAdded(String topic, Message msg)***: invia il nuovo messaggio ricevuto su un topic a tutti i subscriber registrati su quel topic, se il topic selezionato dal subscriber è lo stesso del messaggio inviato.

In sintesi, la classe ClientHandler gestisce le connessioni client-server, processa le richieste dei client (come publisher o subscriber), e si assicura che l’accesso ai topic sia sicuro grazie a lock in lettura/scrittura. Utilizzando un'architettura concorrente, consente a più client di interagire con i topic simultaneamente mantenendo l'integrità dei dati.

**Receiver**:

La classe Receiver è un thread dedicato alla ricezione di messaggi inviati dal server al client. Si occupa di mantenere una connessione aperta con il server tramite un socket e di gestire in modo continuo la ricezione di dati fino alla chiusura del collegamento.

Componenti principali:

* ***socket***: Socket – rappresenta il socket di connessione tra client e server, utilizzato per ricevere i messaggi dal server.
* ***sender***: Thread – riferimento al thread Sender che gestisce l'invio dei messaggi al server. Questo riferimento viene utilizzato per interrompere Sender quando Receiver termina.

Metodi principali:

1. ***run()***
   * Questo metodo implementa il comportamento principale di Receiver come thread.
   * Viene creato uno Scanner per leggere l'input dal socket (socket.getInputStream()), consentendo di ricevere continuamente messaggi dal server.
   * Il thread rimane in esecuzione finché non riceve il comando "quit" dal server:
     + Ogni messaggio ricevuto viene stampato sulla console del client.
     + Se il messaggio ricevuto è "quit", Scanner chiude lo stream di input e il ciclo di lettura si interrompe.
   * In caso di eccezioni come IOException o NoSuchElementException, il thread gestisce gli errori loggando un messaggio di errore.
   * Infine, nel blocco finally, Receiver stampa un messaggio di chiusura e interrompe il thread Sender, segnalando la necessità di chiudere la connessione.

In sintesi, Receiver garantisce una comunicazione asincrona tra il client e il server, ricevendo continuamente i messaggi dal server finché non viene richiesto di chiudere la connessione.

**Sender:**

La classe Sender è un thread che gestisce l'invio di messaggi dal client al server. Sender utilizza un socket per trasmettere comandi o messaggi in tempo reale, leggendo input dall'utente e inoltrandolo al server fino a quando non viene richiesto di terminare la connessione.

Componenti principali:

* ***socket***: Socket – il socket che collega il client al server e attraverso cui vengono inviati i messaggi.
* ***userInput***: Scanner – uno scanner per leggere l'input dalla tastiera, consentendo all'utente di inviare comandi e messaggi al server.

Metodi principali di Sender

1. ***run()***
   * Il metodo principale eseguito dal thread Sender.
   * Crea un PrintWriter per inviare i messaggi attraverso il socket (socket.getOutputStream()).
   * Entra in un ciclo continuo in cui legge l'input dalla console (userInput.nextLine()) e lo invia al server.
   * Se il thread è stato interrotto, invia il messaggio "quit" al server per segnalare la chiusura e interrompe il ciclo.
   * Il ciclo continua finché l'input dell'utente non è "quit", il quale invia il comando "quit" anche al server, segnando la fine della comunicazione.
   * Gestisce eventuali eccezioni (IOException) loggando un messaggio di errore per problemi di comunicazione con il server.
   * Nel blocco finally, chiude userInput e segnala la chiusura del thread con un messaggio in console.

In sintesi, Sender consente l'invio continuo di messaggi al server fino alla richiesta di chiusura, supportando la comunicazione asincrona client-server in parallelo al thread Receiver.

**Suddivisione del lavoro tra i membri del gruppo:**

Nel gruppo, il lavoro è stato svolto con la partecipazione attiva di tutti i membri. Durante ogni incontro, le attività sono state assegnate in modo dinamico, suddividendo le mansioni sul momento in base alle necessità e alle competenze richieste per ciascuna fase del progetto. La distribuzione dei compiti, quindi, è avvenuta in modo flessibile e condiviso, senza una suddivisione predefinita, ma sempre nel rispetto delle capacità e delle preferenze di ciascun membro. Tuttavia, in generale, il lavoro è stato suddiviso a coppie di membri:

* Interazione Client-Server, utente-console, output e input console: Andrea Caselli e Diego Gallini
* Gestione della concorrenza, gestione delle strutture dati e risorse condivise, debug: Francesco Forlani e Nicolas Cola

**DESCRIZIONE E DISCUSSIONE DEL PROCESSO DI IMPLEMENTAZIONE:**

**Descrizione dei problemi**

Abbiamo riscontrato difficoltà nella gestione della concorrenza, in particolare per quanto riguarda l’implementazione della mutua esclusione nel metodo inspect. Questo metodo deve mantenere il lock attivo finché l’amministratore del server non termina la sessione di ispezione e deve inoltre applicarsi a un singolo topic alla volta.

Per risolvere il problema, abbiamo implementato una HashMap che associa a ciascun topic (chiave della mappa) un lock specifico. Il tipo di lock scelto è ReentrantReadWriteLock, che consente l’accesso concorrente in modalità lettura o scrittura. In questo modo, più thread possono accedere contemporaneamente alla risorsa in modalità lettura, migliorando la scalabilità. Tuttavia, se un thread necessita di bloccare completamente l’accesso al topic, come nel caso di inspect, il lock viene acquisito in modalità scrittura, impedendo l’accesso ad altri thread fino al termine dell’operazione.

Grazie a questo approccio, abbiamo potuto gestire correttamente la concorrenza sulle risorse condivise (classe Resource), poiché a ciascun metodo che accede a queste risorse, sia lato client sia lato server, è associata una specifica modalità di lock. La mutua esclusione era necessaria in quanto le modifiche concorrenti sulle risorse condivise potevano compromettere la loro integrità.

Un’alternativa possibile per gestire la mutua esclusione sarebbe stata quella di utilizzare il meccanismo wait e notify. In questo caso, ogni topic avrebbe un proprio oggetto di sincronizzazione nella HashMap. Quando un thread esegue il metodo inspect su un topic, acquisisce il monitor dell’oggetto di sincronizzazione corrispondente, bloccando l’accesso degli altri thread. I thread che tentano di accedere allo stesso topic durante inspect eseguono wait() sull’oggetto di sincronizzazione, rimanendo in attesa fino alla fine dell’ispezione. Quando l’amministratore termina inspect, viene chiamato notifyAll(), sbloccando tutti i thread in attesa per quel topic e consentendo la ripresa delle operazioni.

Un altro problema riscontrato è stato nella gestione degli ID dei messaggi all'interno dei singoli topic. Nel sistema, in cui diversi client hanno la possibilità di pubblicare messaggi contemporaneamente sullo stesso topic, era necessario garantire che l'assegnazione degli ID fosse corretta e thread-safe. Per affrontare questo problema, abbiamo utilizzato un oggetto AtomicInteger, che assicura l'accesso sicuro e coerente al contatore da parte di più thread, evitando così problematiche di race condition. Senza questa soluzione, infatti, l'uso concorrente di ID potrebbe portare a incoerenze nei valori generati. L'adozione di questa struttura dati ha inoltre il vantaggio di preservare l'unicità degli ID anche dopo la rimozione di un messaggio. Questo significa che, anche una volta eliminato un messaggio, l'ID associato rimane univoco, evitando possibili conflitti o duplicazioni durante la gestione dei nuovi messaggi.

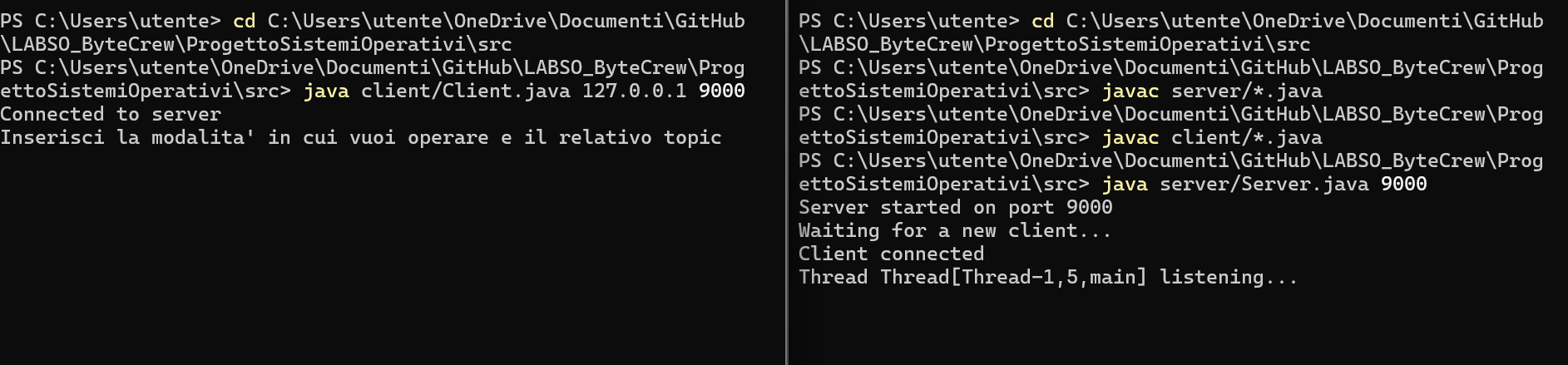
**Descrizione degli strumenti utilizzati:**

Per l'implementazione e lo sviluppo del progetto, abbiamo utilizzato Eclipse come ambiente di sviluppo; la comunicazione tra i membri è stata mantenuta costante e immediata tramite Discord e WhatsApp. Discord è stato utilizzato principalmente per riunioni vocali, discussioni di gruppo e confronti diretti sui dettagli tecnici del progetto, offrendo uno spazio collaborativo per la condivisione in tempo reale di idee e soluzioni. WhatsApp ha invece garantito un flusso continuo di aggiornamenti, utilizzato soprattutto per coordinarsi sugli aspetti organizzativi; per la gestione e la condivisione del codice prodotto abbiamo utilizzato GitHub, piattaforma utilizzata per mantenere il controllo delle versioni del progetto e per facilitare il lavoro collaborativo; infine, per quanto riguarda la pianificazione e la tracciabilità delle attività, abbiamo preferito una gestione informale, sfruttando le comunicazioni su Discord e WhatsApp per discutere e annotare in tempo reale le decisioni prese, il lavoro svolto e le attività da completare. Questo approccio ci ha consentito di organizzare le task in modo flessibile e di adattare rapidamente le priorità in base all'avanzamento complessivo del progetto.

**REQUISITI E ISTRUZIONI PER COMPILARE E USARE LE APPLICAZIONI**

**Compilazione e avvio dell’applicazione**

Per compilare i file .java, aprire il prompt dei comandi e spostarsi nella cartella “src”, contenente a sua volta le cartelle “client” e “server” con all’interno i file dell’applicazione. Utilizzare i comandi “**Javac server/.java***”* e *“****J*avac client/.java**” per compilare i file. Dopodiché, avviare il server con il comando “**Java server/Server.java 9000**” dove 9000 è una porta a piacere. In un’altra console, avviare il client con il comando “**Java client/Client.java 127.0.0.1 9000**” dove il primo numero è l’indirizzo IP (localHost) ed il secondo la porta a piacere, che deve essere uguale a quella del server. A questo punto è possibile utilizzare l’applicazione.



Si possono visualizzare i comandi disponibili in qualsiasi momento all’interno dell’applicazione tramite il comando “**commands**” .

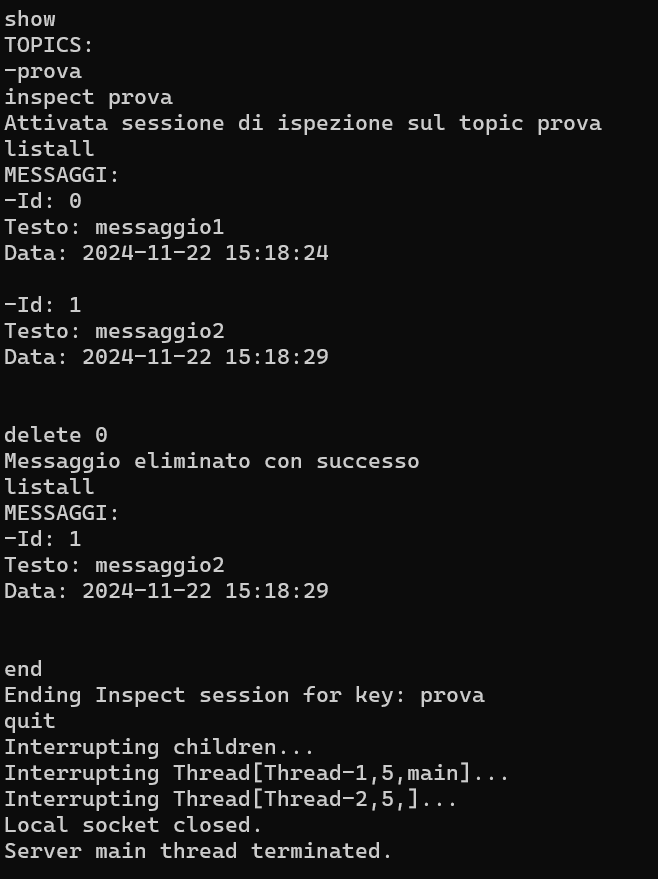
**Comandi del server e output attesi**

I comandi a disposizione del server sono i seguenti:

* ***quit:***Termina l'esecuzione del server. Non produce alcun output specifico, il server si interrompe;
* ***show:***Mostra tutti i topic disponibili nell’applicazione. L'output è una lista restituita dal metodo topics.show();
* ***inspect <topic>:***Avvia una sessione di ispezione per un topic. Se il topic non esiste, viene mostrato “Topic non esistente”. Se non è specificato, viene mostrato: “Necessario specificare il topic da ispezionare”. Se la sessione viene avviata con successo viene stampato il messaggio “Attivata sessione di ispezione sul topic <topic>”.

Durante la sessione di ispezione, sono disponibili i seguenti comandi:

* ***end:***Termina la sessione di ispezione;
* ***listall:***Mostra tutti i messaggi del topic in ispezione.  
  L'output è restituito dal metodo topics.listAll(topic).
* ***delete <id>:***Elimina un messaggio specifico, identificato da id, dal topic in ispezione.
  + Se il messaggio è eliminato con successo: Messaggio eliminato con successo.
  + Se l'ID non esiste o non è trovato: ID messaggio non esistente o non trovato.
  + Se non ci sono messaggi nel topic: Non sono presenti messaggi sul topic da poter eliminare.
  + Se l'ID non è un numero: Error: ID must be a number.
  + Se l'ID non è fornito: Error: delete requires an id argument;
* ***comandi non riconosciuti durante l'ispezione:***Mostra: Unknown cmd: <command>.



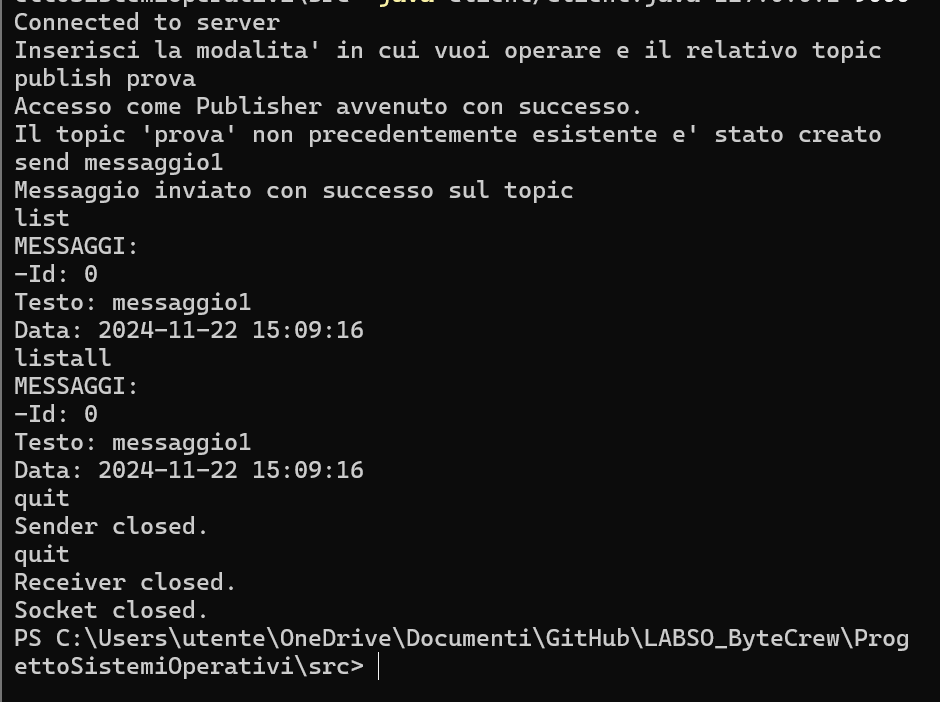
**Comandi del client e output attesi**

Appena connesso, al client viene chiesto se vuole essere publisher o subscriber. I comandi sono i seguenti:

* ***publish [topic]:*** registra il client come publisher per il topic specificato. Se il topic non esiste, viene creato:
  + se il topic non esiste:  
    "Accesso come Publisher avvenuto con successo. Il topic '[topic]' non precedentemente esistente è stato creato";
  + se il topic esiste:  
    "Accesso come Publisher avvenuto con successo al topic '[topic]'";
* ***show*:** mostra tutti i topic disponibili nel sistema. Restituisce l'elenco dei topic esistenti, uno per riga;
* ***subscribe [topic]*:** registra il client come subscriber per il topic specificato:
  + se il topic esiste:  
    "Accesso come Subscriber avvenuto con successo al topic '[topic]'";
  + se il topic non esiste:  
    "Accesso come Subscriber fallito, il topic '[topic]' non esiste".

I comandi a disposizione del publisher sono i seguenti:

* ***send [message]*:** invia un messaggio sul topic scelto. Restituisce "Messaggio inviato con successo sul topic" se l’operazione è andata a buon fine;
* ***list*:** mostra i messaggi inviati dal client corrente sul topic scelto. L’output atteso è l’ elenco dei messaggi inviati dal client sul topic.
* ***listall*:** mostra tutti i messaggi inviati da tutti i publisher sul topic scelto. L’output atteso è l’elenco completo dei messaggi sul topic.



Per quanto riguarda il subscriber, l’unico comando dei precedenti che condivide con il publisher è **listall**. Oltre ai comandi condivisi iniziali, non ne ha altri, viene però stampato nella console di questo ogni messaggio aggiunto sul topic a cui è iscritto.

