

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

LABSO\_EJ

[edoardo.galli3@studio.unibo.it](mailto:edoardo.galli3@studio.unibo.it)

Edoardo Galli - 0001049383

Jean Baptiste Dindane - 0001100694

## 1. Descrizione del Progetto Consegnato

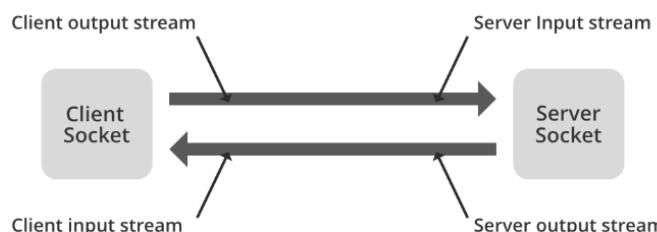
### (a) Architettura Generale

Il progetto ha come obiettivo primario la creazione di un'applicazione che permetta a molteplici client di connettersi ad un server e di comunicare fra l'un l'altro, dopo essersi registrati a un determinato topic.

Il linguaggio di programmazione utilizzato per l'applicazione è Java, il meccanismo di comunicazione di rete di cui ci siamo serviti per stabilire connessioni tra server e client è quello dei Socket.

I socket sono alla base di molte applicazioni online che richiedono una comunicazione in tempo reale tra dispositivi diversi, come ad esempio chat, giochi multiplayer, streaming di dati (audio e video), e sistemi di messaggistica istantanea come WhatsApp, Telegram e Messenger (per inviare notifiche, aggiornamenti di stato e altri tipi di contenuti in tempo reale). In Java sono implementati tramite le classi **Socket** e **ServerSocket**, che fanno parte del pacchetto *java.net*.

#### **Come avviene la comunicazione tra server e client?**



I socket vengono anche definiti come **endpoint** per la comunicazione tra due macchine (client e server). Ogni socket è definito da:

- **Indirizzo IP**: l'indirizzo della macchina a cui si desidera connettersi.
- **Porta**: il numero che identifica in modo univoco un'applicazione specifica sulla macchina destinataria.

In un'applicazione client-server, il client stabilisce una connessione con un server tramite un socket. Una volta connessi, client e server possono scambiare messaggi attraverso il canale di comunicazione creato.

### ***Cos'è il multithreading, come ce ne serviamo in questo progetto?***

Per fare in modo che il server non fosse limitato a gestire un solo client alla volta, abbiamo adottato un approccio multithread, in questo modo, ogni connessione con un client viene trattata autonomamente su un thread dedicato, consentendo al server di ricevere e gestire simultaneamente più richieste provenienti da diversi client e i comandi lato server.

Inoltre, sia il server che ogni client, dispongono internamente di un'architettura I/O che usufruisce del multithreading, perché inizialmente quando ciascun processo viene lanciato, sono creati all'interno **due thread separati**: Uno responsabile per **l'ascolto e gestione di input da console**, da cui sia server che client possono eseguire comandi, ed uno responsabile per **l'ascolto e stampa di output da altri client (output stream)**, come messaggi ricevuti o notifica di cambiamenti fatti dal server.

Il **multithreading** è una tecnica di programmazione che consente di suddividere un programma in più "thread" indipendenti, ovvero unità più piccole di esecuzione, che possono funzionare in parallelo. Ogni thread può eseguire un compito diverso nello stesso programma, il che permette di svolgere più operazioni contemporaneamente.

Quando si parla di **multithreading**, una delle sfide principali riguarda la gestione delle **risorse condivise**. Queste sono risorse, come variabili, oggetti o strutture dati, che possono essere accessibili e modificate da più thread contemporaneamente.

### ***Cosa succede se le risorse condivise non vengono gestite correttamente ?***

L'inconsistenza dei dati è solo uno dei potenziali problemi che possono emergere nel multithreading, una gestione sbagliata delle risorse condivise potrebbe condurre ad altre problematiche come quelle sottoindicate.

#### **Race Condition**

Race Condition (condizione di gara), questa condizione si verifica quando due o più thread accedono e modificano le stesse risorse contemporaneamente.

Poiché l'ordine di esecuzione dei thread non è garantito, il risultato finale può variare e portare a **incoerenze nei dati** o a **comportamenti imprevedibili**.

Un esempio di race condition potrebbe verificarsi quando si parla di “producer consumer problem”, dove il processo produttore produce un'informazione che il processo consumer consuma (entrambi condividono una stessa variabile).

Nell'esempio nella tabella sottostante, abbiamo rappresentato le operazioni eseguite da due processi, Producer e Consumer. Viene data una variabile Counter (che è una risorsa condivisa), entrambi i processi accedono alla variabile ed eseguono un'operazione specifica, Producer incrementa la variabile mentre Consumer la decrementa.

Passo	Operazione	Processo	Valori
T0	Register1 = Counter	Producer	Register1 = 5
T1	Register1 += 1	Producer	Register1 = 6
T2	Register2 = Counter	Consumer	Register2 = 5
T3	Register2 -= 1	Consumer	Register2 = 4
T4	Counter = Register1	Producer	Counter = 6
T5	Counter = Register2	Consumer	Counter = 4

Il valore iniziale di Counter è 5, nella tabella viene fatto notare che se vengo eseguiti simultaneamente i due processi è impossibile definire un output ben preciso, il valore finale di Counter assumerà il valore 4 o 6, mentre invece il risultato corretto dovrebbe essere 5.

Questo fenomeno, chiamato *race condition*, si verifica quando processi concorrenti accedono e modificano simultaneamente una risorsa condivisa senza controllo sull'ordine di esecuzione, portando a risultati imprevedibili

## Deadlock

Deadlock (Stallo): Accade quando due o più thread rimangono bloccati, ciascuno in attesa che una risorsa venga rilasciata dall'altro. Nessuno dei thread coinvolti può continuare l'esecuzione, portando il programma in uno stato di blocco permanente. Un deadlock si verifica spesso in sistemi che richiedono più risorse per completare un'operazione.

Il codice sottostante è un esempio di connessione Client - Server, tramite Socket. Il modo in cui Client e Server sono implementati, crea un Deadlock, il metodo **hasNextLine()** del client ritorna false solo quando il server viene chiuso, perciò non riusciamo mai ad uscire dal ciclo while. Nel lato server invece non riusciamo ad avanzare perché stiamo cercando di ricevere dati da parte del client (il quale è intrappolato nel loop).

```
EDO/Server.java
7  public class Server {
8      public static void main(String args[]) {
9          try {
10              ServerSocket listener = new ServerSocket(1555);
11              System.out.println("Listening");
12              Socket socket = listener.accept();
13              System.out.println("client " + socket + " connected");
14
15              // write to client
16              PrintWriter to = new PrintWriter(socket.getOutputStream(), true);
17              to.println("ciao client io sono server ss500");
18
19              // read from client
20              Scanner from = new Scanner(socket.getInputStream());
21              while (from.hasNextLine()) {
22                  System.out.println(from.nextLine());
23              }
24
25              from.close();
26              to.close();
27              listener.close();
28              socket.close();
29
30          } catch (IOException e) {
31              e.printStackTrace();
32          }
33      }
34  }

EDO/Client.java
6  public class Client {
7      public static void main(String[] args) throws IOException {
8          try {
9              Socket client = new Socket("localhost", 1555);
10             System.out.println("Connected");
11             Scanner fromClient = new Scanner(client.getInputStream());
12
13             // read from server, infinite loop (no break condition)
14             while (!fromClient.hasNextLine()) {
15                 System.out.println(fromClient.nextLine());
16             }
17
18             // will never reach here
19             System.out.println("exited the reading block");
20
21             PrintWriter toServer = new PrintWriter(client.getOutputStream(), true);
22             toServer.println("I am jean");
23
24             toServer.close();
25             fromClient.close();
26             client.close();
27         } catch (IOException e) {
28             e.printStackTrace();
29         }
30     }
31 }
```

Questo è un esempio di Deadlock, abbiamo due thread che rimangono bloccati, in attesa che una risorsa venga rilasciata da un'altro.

## Starvation

Parliamo di starvation (fame) quando un thread rimane costantemente in attesa di una risorsa, senza mai riuscire ad accedervi perché viene sempre assegnata una priorità maggiore ad altre operazioni. Questo può portare a un blocco temporaneo o permanente per il thread affetto.

## Livelock

Il livelock si verifica quando due o più thread sono costantemente impegnati a cambiare stato in risposta alle azioni degli altri thread, senza mai progredire verso una condizione di stallo. In sostanza, i thread sono in un ciclo infinito di tentativi falliti, senza mai raggiungere un obiettivo utile.

## **Quali strategie si sono adottate contro i problemi tipici del multithreading?**

In generale abbiamo visto che una gestione non corretta delle risorse condivise quando si lavora con molteplici thread può portare a diversi problemi. Le soluzioni adottate contro i problemi sopra indicati sono le seguenti:

- **Synchronized:** In Java, il blocco synchronized garantisce che solo un thread alla volta possa accedere a una determinata sezione di codice, a un metodo, o ad un oggetto. Questo aiuta a prevenire le race conditions e a garantire la coerenza dei dati condivisi.

Un esempio di race condition potrebbe verificarsi nella situazione in cui il publisher (avendo già inviato in precedenza alcuni messaggi) richieda una lista dei suoi messaggi con il comando “list”. Durante l'esecuzione di tale comando, vogliamo garantire l'integrità dei dati presenti all'interno di publisherMessages (prevenire che il publisher invii un messaggio con “send”), in modo da non interferire con la collezione e stampa dei messaggi.

```
SISOP-PRJ/ClientHandler.java
127 StringBuilder messageOutput = new StringBuilder();
128 synchronized (publisherMessages) {
129     messageOutput.append("---- LIST: YOU SENT " + publisherMessages.get(topic).size() + " MESSAGES IN '" + topic + "' ---\n\n");
130     for (Message msg : publisherMessages.get(topic)) {
131         messageOutput.append(msg.toString()).append("\n");
132     }
133     messageOutput.append("---- LIST: END OF MESSAGES YOU SENT ---\n");
134     out.println(messageOutput);
135 }
```

- **Concurrent Collections:** In particolare ci siamo serviti delle collection **ConcurrentHashMap** e **ConcurrentLinkedQueue**, appartenenti alla libreria **java.util.Concurrent**, le quali permettono un accesso concorrente sicuro senza la necessità di implementare esplicitamente dei lock.

Nel nostro progetto ci serviamo di queste collection per le variabili “static”, cioè quelle variabili condivise tra tutte le istanze della classe come **clientHandlers** e **topics**, a queste variabili hanno accesso diversi thread.

```
SISOP-PRJ/ClientHandler.java
19 public static ConcurrentHashMap<Integer, ClientHandler> clientHandlers = new ConcurrentHashMap<>(); // userID : ClientHandler
20 public static ConcurrentHashMap<String, ConcurrentLinkedQueue<Message>> topics = new ConcurrentHashMap<>(); // topic : all-messages-of-topic
```

- **Atomic Integer**: Le operazioni su tali variabili sono atomiche, il che significa che vengono eseguite come un'unica operazione indivisibile, senza la possibilità che altri thread possano intervenire o osservare stati intermedi durante l'esecuzione.

Ci siamo serviti di questa classe per gestire **messageCounter** nella classe Message e **clientCounter** nella classe ClientHandler perché volevamo che ogni messaggio e ogni client avessero un ID unico.

```
SISOP-PRJ/Message.java
10 private static final AtomicInteger messageCounter = new AtomicInteger(0); // Unique ID for each message
```

```
SISOP-PRJ/ClientHandler.java
29 private static final AtomicInteger clientCounter = new AtomicInteger(0); // Unique ID for each client
```

- **ExecutorService**: Questa interfaccia appartenente alla libreria **java.util.concurrent**, non richiede di creare thread manualmente, ma ci permette di assegnare task (compiti) servendoci di un pool.

**Executors.newCachedThreadPool()** crea un pool di thread flessibile che può riutilizzare i thread creati in precedenza, se disponibili, oppure crearne di nuovi quando necessario.

Ci serviamo serviti di queste funzionalità nella classe Server, ogni volta che un nuovo client si connette, viene creata una nuova istanza di ClientHandler per gestire le comunicazioni con tale client, e il metodo run() della classe ClientHandler viene eseguito da un nuovo thread, mentre il thread principale rimane in ascolto per altre connessioni.

```
SISOP-PRJ/Server.java
40 while (serverRunning) {
41     try {
42         Socket socket = serverSocket.accept();
43         System.out.println("--- NEW CLIENT CONNECTED ---");
44         ClientHandler clientHandler = new ClientHandler(socket, this);
45         pool.execute(clientHandler);
46     } catch (SocketTimeoutException | SocketException e) {
47         if (!serverRunning) {
48             break;
49         }
50     }
51 }
```

- **StringBuilder**: Inizialmente avevamo strutturato l'output delle funzioni (come "list" o "listall") in una sequenza di print statements (**System.out.println()**), ma ci siamo accorti che a volte capitava che l'output di diversi comandi (sia localmente che fra client) si mescolava, creando confusione e difficoltà nella lettura di output. Questo lo si può immaginare nella seguente situazione: Client A (supponendo che abbia inviato messaggi in precedenza) esegue il comando "list" per ricevere una lista dei suoi messaggi. Durante l'esecuzione del "list" di Client A, un Client B (sullo stesso topic) manda un messaggio con "send".
  - Se l'output del "list" di Client A fosse una sequenza di print distaccati, sulla sua console comparirebbe erroneamente il messaggio di Client B **nel mezzo** dei print del proprio "list".
  - Utilizzando invece StringBuilder, abbiamo modo di comporre una stringa da inizio a fine che racchiude tutti i messaggi di Client A, e invece di avere un miscuglio di print come nel scenario di sopra, Client A vedrebbe nella propria console **prima** il messaggio di Client B (inviauto ad Client A durante la sua esecuzione di "list"), **e dopo** verrebbe stampata **l'intera stringa** del "list" di Client A, composta da StringBuilder.

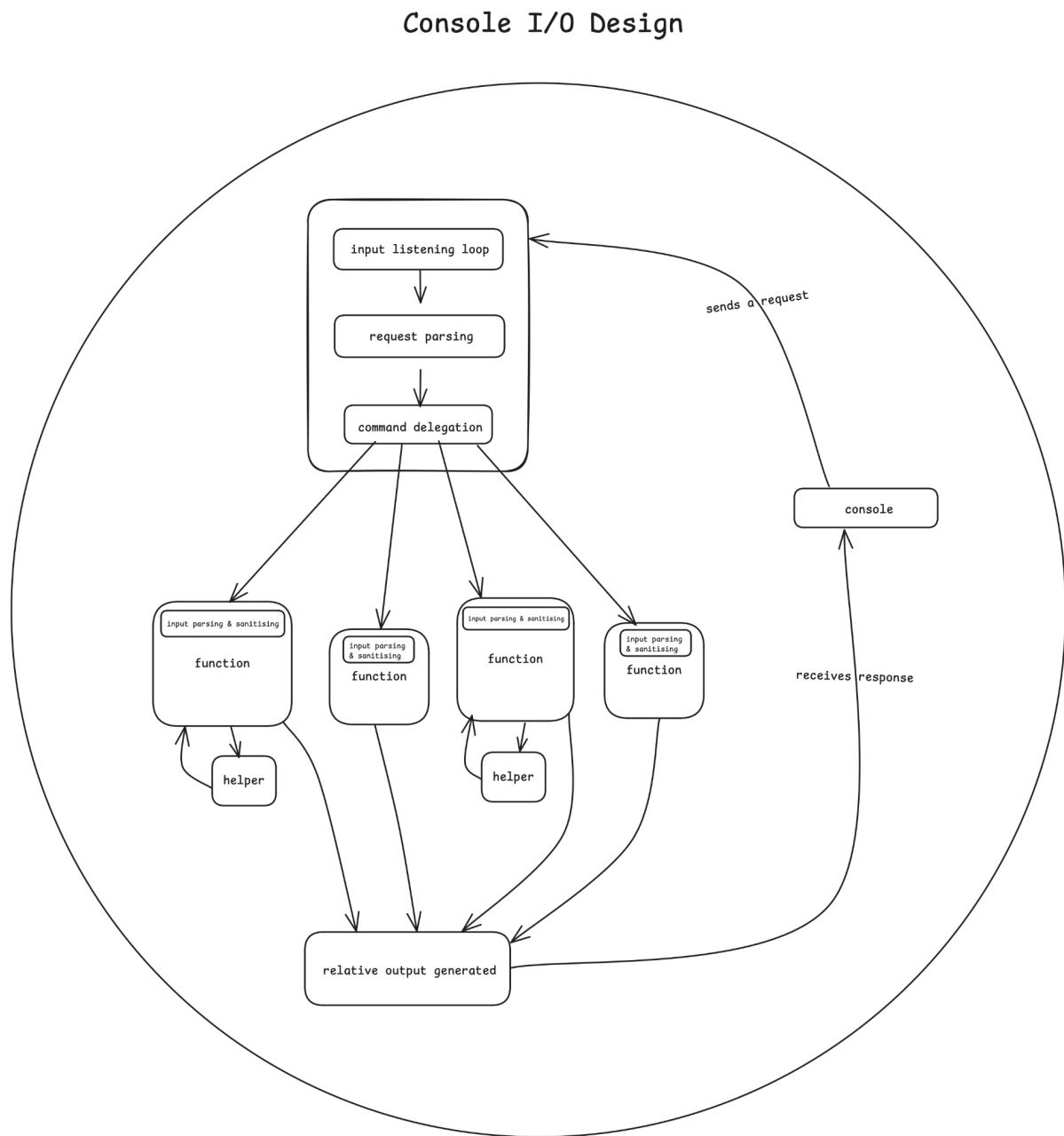
```
SISOP-PRJ/ClientHandler.java
158 StringBuilder messageOutput = new StringBuilder();
159 messageOutput.append("--- LISTALL: ").append(snapshot.size()).append(" MESSAGES IN '").append(topic).append("' ---\n\n");
160 for (Message msg : snapshot) {
161   messageOutput.append(msg.toString()).append("\n");
162 }
163 messageOutput.append("--- LISTALL: END OF MESSAGES IN '").append(topic).append("' ---\n");
164 out.println(messageOutput);
```

- **Snapshot Mechanism:** Per quanto riguarda il comando **client-side** “listall” (indipendente dal “listall” **server-side**, che viene eseguito dal server **solo** durante Inspect mode dove **nessun client può inviare messaggi**), una race condition che abbiamo dovuto affrontare la si può immaginare nella seguente situazione: Client A richiede una lista completa di tutti i messaggi del proprio topic mediante “listall”, e durante l’esecuzione di tale richiesta, Client B invia un messaggio sullo stesso topic.
  - Se Client A stesse raccogliendo tutti i messaggi del topic accedendo direttamente alla collezione “centrale” di tutti i messaggi (collocati nella mappa statica “topics” di ClientHandler), condivisa da tutti i client, avverrebbe una lettura sporca perché in contemporanea ci sarebbe (in fondo) il nuovo messaggio di Client B, che non esisteva inizialmente quando Client A fece la sua richiesta “listall”. Pertanto l’expected output di Client A differirebbe dall’output reale, contenente in questo caso un ulteriore messaggio imprevisto. Questo lo si nota quando viene calcolata e successivamente stampata la dimensione (.size()) della lista di messaggi condivisa, che indicherebbe un numero inferiore di messaggi a quello effettivamente stampato.
  - La soluzione è relativamente semplice, piuttosto di iterare sulla collezione di messaggi condivisa da tutti i client, creo uno “snapshot” della lista di messaggi del topic, nell’istante in cui Client A fece la sua richiesta “listall”, e avendo la garanzia che questa copia non verrà modificata da altri client (essendo locale a Client A), itero su di essa per generare l’output corretto.

```
SISOP-PRJ/ClientHandler.java
154 ArrayList<Message> snapshot;
155 synchronized (messages) { snapshot = new ArrayList<>(messages); }
```

## (b) Descrizione Dettagliata delle Singole Componenti

### Logica di Implementazione



**N.B.** Il diagramma è leggermente semplificato:

- La freccia in uscita da console (“sends a request”) rappresenta il thread che gestisce l’input da tastiera del server/client
- La freccia in entrata da console (“receives response”) rappresenta il thread che gestisce il ricevimento (in input) dell’output generato dal server o altri client

- Nel caso in cui il client faccia un richiesta svolta server-side, la chiamata della relativa funzione (post-delega del comando) non avviene localmente ma invece esternamente attraverso il ClientHandler associato al client

Questa immagine descrive brevemente la logica alla base di ciascuna classe del progetto (utilizzata per gestire l'I/O del server e di ogni client). L'applicazione riceve un input dalla console e lo sottopone a un processo di parsing e sanitizzazione. Successivamente, un'istruzione switch-case viene utilizzata per mappare l'input a una specifica azione.

Se il token è presente nella lista comandi, viene eseguita la funzione associata ad essa, in caso contrario viene stampato un messaggio di default per informare l'utente.

Al termine dell'esecuzione di ogni funzione viene stampato un messaggio che descrive l'output dell'operazione richiesta, oppure un messaggio di errore. Alcune funzioni, al fine di completare correttamente la loro esecuzione, hanno bisogno del supporto di funzioni secondarie ("helper functions") che ci hanno permesso di suddividere problemi complessi in sotto-problemi più semplici e gestibili.

In seguito a questa sezione vengono descritte le classi e i metodi principali che fanno parte del progetto.

## **Come viene implementato il server, quali sono i suoi compiti?**

Nel nostro progetto il server viene implementato attraverso le classi Server e ClientHandler, ed è responsabile per la gestione delle richieste di tutti i client.

### **Server**

Questa classe è il punto di partenza della nostra applicazione, ci permette di eseguire e gestire correttamente tutti i comandi dal lato server. Si fornisce, di seguito, una descrizione più dettagliata di questa classe e dei suoi metodi principali.

Metodo main() classe server

```
SISOP-PRJ/Server.java
594  public static void main(String[] args) {
595      if (args.length != 1) {
596          System.err.println("Usage: java Server <port>");
597          System.exit(1);
598      }
599
600     try (ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]))) {
601         Server server = new Server(serverSocket);
602         // When start() is called on the thread, server.startServer() is called within the thread
603         Thread serverThread = new Thread(server::startServer);
604         serverThread.start();
605         server.processCommand();
606     } catch (IOException e) {
607         System.out.println("> Error starting server: " + e.getMessage());
608     }
609 }
```

I comandi `javac Server.java` e `java Server <portNumber>`, (`portNumber` preferibilmente nel range 1024-65535) lanceranno il server della nostra applicazione, in caso non fossero utilizzati correttamente l'utente verrebbe allertato con un messaggio di errore.

Quando viene lanciata l'applicazione, andremo a creare un oggetto di tipo **ServerSocket**, invece di metterci in ascolto nel thread principale per una connessione da parte di un client, lanceremo il metodo **startServer()** che verrà eseguito in un nuovo thread (`serverThread`).

Il comando successivo è **server.processCommand()**, questo metodo è utilizzato per gestire i comandi lato server, rimarrà quindi in esecuzione fino a quando il server non viene interrotto “`while (serverRunning)`”.

Ogni volta che premiamo il tasto invio, l'input inserito nella tastiera viene controllato, se viene inserita una stringa vuota, il resto del codice viene saltato e si passa all'iterazione successiva (`if (commandLine.isEmpty()) continue;`), in caso contrario, andiamo a processare l'input, separandolo in base agli spazi bianchi, il tutto viene poi gestito attraverso uno switch-case, che (come anche in Client e

ClientHandler) contiene una lista esaustiva di ogni comando disponibile nell'applicazione.

```
SISOP-PRJ/Server.java
75 switch (command) {
76 case "show" -> showTopics();
77 case "quit" -> shutdownServer();
78 case "inspect" -> startInspectMode(tokens);
79 case "end" -> endInspectMode();
80 case "listall" -> listAllMessagesInTopic();
81 case "delete" -> deleteMessage(tokens);
82 case "help" -> showHelp();
83 case "kick" -> kickClient(tokens);
84 case "clear" -> clearTopic(scanner);
85 case "export" -> export(tokens);
86 case "users" -> showAllUsersInformation();
87 case "user" -> showUserInformation(tokens);
88 default -> System.out.println("> Unknown command. Enter 'help' to see the list of available commands.\n");
89 }
```

Se il comando corrisponde ad uno dei “case” definiti cui sopra, viene chiamato il metodo equivalente, altrimenti stampiamo il messaggio di default.

Il metodo `startServer()` rimane in esecuzione fino a quando il server non viene interrotto (`while (serverRunning)`), attraverso questo metodo, rimaniamo in ascolto per connessioni da parte del client, una volta stabilita la connessione, passiamo l'oggetto socket ad un'istanza della classe ClientHandler, assieme ad un riferimento all'istanza corrente del server. Il metodo `run()` di tale classe viene eseguito da un nuovo thread tramite il comando `pool.execute(clientHandler);`.

```
SISOP-PRJ/Server.java
37 private void startServer() {
38     try {
39         System.out.println("--- SERVER STARTED ON PORT " + serverSocket.getLocalPort() + " ---");
40         while (serverRunning) {
41             try {
42                 Socket socket = serverSocket.accept();
43                 System.out.println("--- NEW CLIENT CONNECTED ---");
44                 ClientHandler clientHandler = new ClientHandler(socket, this);
45                 pool.execute(clientHandler);
46             } catch (SocketTimeoutException | SocketException e) {
47                 if (!serverRunning) {
48                     break;
49                 }
50             }
51         }
52     } catch (IOException e) {
53         System.out.println("> Error starting server: " + e.getMessage());
54     } finally {
55         pool.shutdown();
56     }
57 }
```

Il metodo `startInspectMode(String[] tokens)` inizia una sessione interattiva in cui è possibile analizzare un topic, richiede un solo argomento `<topic>`, se il topic non esiste, restituisce un messaggio di errore.

Durante la sessione interattiva, il server ha a disposizione i seguenti comandi:

- > show: Ritorna una lista dei topic esistenti
- > quit: chiude il server, dopo aver disconnesso tutti i client
- > inspect <topicExample> : Inizia una sessione interattiva su topicExample
- > end : termina la sessione interattiva
- > listall: List all messages in the topic
- > delete <messageld>: Delete a message by ID
- > help: Stampa i comandi che il server ha a disposizione

```
SISOP-PRJ/Server.java
174  private void startInspectMode(String[] tokens) {
175      if (isInspecting) {
176          System.out.println("> Command 'inspect' is not available in inspect mode.\n");
177          return;
178      }
179
180      if (tokens.length < 2) {
181          System.out.println("> Usage: inspect <topic>\n");
182          return;
183      }
184
185      String topic = String.join("_", Arrays.copyOfRange(tokens, 1, tokens.length)); // "example topic" -> "example_topic"
186      if (!ClientHandler.topics.containsKey(topic)) {
187          System.out.println("> Topic '" + topic + "' does not exist.\n");
188          return;
189      }
190
191      isInspecting = true;
192      currentInspectTopic = topic;
193      System.out.println(" --- INSPECT MODE STARTED --- ");
194      System.out.println("> Begun inspecting topic '" + topic + "'. Enter 'help' for a list of available commands.\n");
195
196      // Notify clients that the server is inspecting the topic
197      for (ClientHandler clientHandler : ClientHandler.clientHandlers.values()) {
198          if (topic.equals(clientHandler.getTopic())) {
199              clientHandler.setIsServerInspecting(isInspecting);
200          }
201      }
202  }
```

Quando `startInspectMode(String[] tokens)` viene chiamato, per prima cosa ci assicuriamo che il server non stia già ispezionando un altro topic, che gli argomenti della funzione siano stati passati correttamente e che il topic che vogliamo ispezionare esista. Una volta verificate queste condizioni, iteriamo attraverso `clientHandlers`, e se il topic da ispezionare è uguale al topic del client, andiamo a chiamare il metodo `clientHandler.setIsServerInspecting(isInspecting)`. Attraverso questa funzione, il server comunica a ciascun client quali sono i comandi disponibili durante la fase interattiva, e quando quest'ultima viene chiusa.

```

SISOP-PRJ/ClientHandler.java
207 public void setIsServerInspecting(boolean isInspecting) {
208     try {
209         if (isInspecting) {
210             String commands = isPublisher ? "'send', 'list', 'listall'" : "'listall'";
211             out.println("--- SERVER INSPECT STARTED FOR '" + topic + "' ---\n"
212                     + "> Regular functionality has been temporarily suspended. See 'help' for a list of available commands.\n"
213                     + "> Use of " + commands + " will be queued and executed when the server ends Inspect mode.\n";
214     } else
215         out.println("--- SERVER INSPECT ENDED FOR '" + topic + "' ---\n"
216                     + "> Server has exited Inspect mode for topic '" + topic + "'.\n"
217                     + "> Any backlogged commands will now be executed.\n");
218     out.println("IS_SERVER_INSPECTING " + isInspecting);
219 } catch (Exception e) {
220     System.out.println("> Error in setIsServerInspecting(): " + e.getMessage());
221 }
222 }
```

Il messaggio `out.println("IS_SERVER_INSPECTING " + isInspecting);` è un comando speciale che il server invia al client, permette di gestire le richieste inviate da quest'ultimo durante la sessione interattiva.

Questo comando, non viene mai stampato a console, una volta ricevuto, il client chiama il metodo `executeBacklogCommands()` se `isInspecting` ha valore `true`, in caso contrario non viene eseguita nessuna operazione.

```

SISOP-PRJ/Client.java
234 if (tokens[0].equals("IS_SERVER_INSPECTING")) {
235     isServerInspecting = Boolean.parseBoolean(tokens[1]);
236     if (!isServerInspecting) {
237         executeBacklogCommands();
238     }
239     return;
240 }
```

`executeBacklogCommands()`, controlla per prima cosa che l'arraylist `backlog` non sia vuoto, procede poi a riordinare le richieste del client in modo che `list` e `listall` siano eseguite come ultime.

Nel passaggio successivo, tutti i comandi presenti nel backlog vengono passati al metodo `processCommand()` che gestisce le richieste del client prima di inviarle al server (simile come logica al `processCommand` del server). Come ultima cosa ci assicuriamo di ripulire il backlog, servendosi del metodo `clear()`.

L'ultimo metodo della classe Server che descriveremo in modo dettagliato è `shutdownServer()`, viene chiamato attraverso il comando “quit” e permette di chiudere il server dopo aver disconnesso tutti i client.

```
SISOP-PRJ/Server.java
143  private void shutdownServer() {
144      if (isInspecting) {
145          System.out.println("> Cannot shut down server while in inspect mode.\n");
146          return;
147      }
148
149      serverRunning = false;
150      try {
151          System.out.println("> (PRE-QUIT) Connected clients: " + ClientHandler.clientHandlers.size());
152          for (ClientHandler clientHandler : ClientHandler.clientHandlers.values()) {
153              clientHandler.interruptThread();
154          }
155          if (!serverSocket.isClosed()) {
156              serverSocket.close();
157          }
158          pool.shutdownNow();
159          System.out.println("> (POST-QUIT) Connected clients: " + ClientHandler.clientHandlers.size());
160      } catch (IOException e) {
161          System.out.println("> Error shutting down server: " + e.getMessage());
162      }
163
164      System.out.println(" --- SERVER SHUTDOWN --- ");
165      System.exit(0);
166  }
```

Per prima cosa ci si assicura che il server non sia in “inspect mode”, in caso lo fosse, si stampa un messaggio di errore e il metodo viene terminato, in caso contrario si procede con l'esecuzione del resto del codice.

Il valore di `serverRunning` viene impostato a false e per ogni client viene chiamato `interruptThread()`.

Il metodo `interruptThread()` è utile per recuperare e stampare le informazioni riguardanti il client prima di disconnetterlo (role, topic, userID) dal server servendosi di `closeEverything()`.

## Classe ClientHandler

Nel nostro approccio, è stato molto importante dichiarare come variabili statiche i campi `clientHandlers` e `topics`, appartenenti a questa classe. In questo modo, tali variabili risultano accessibili sia dalla classe **ClientHandler** che dalla classe **Server**, oltre che dai diversi thread.

Per evitare i problemi tipici del multithreading già citati (come l'inconsistenza dei dati, i deadlock e le race conditions ecc..), si è deciso di utilizzare le Concurrent Collections messe a disposizione da Java. Si è scelto di usare **ConcurrentHashMap** e **ConcurrentLinkedQueue** al posto dei più comuni **ArrayList** e **HashMap** per via della loro semplicità e della maggiore efficienza quando si lavora con thread multipli.

Come accennato in precedenza, queste Collection consentono di manipolare le risorse condivise garantendo che solo un thread alla volta possa accedervi.

Per motivi di semplicità, si è preferito questo approccio rispetto all'uso di blocchi **synchronized** nella gestione delle risorse condivise, riducendo così al minimo l'utilizzo di questi ultimi.

**Questa classe è fondamentale per gestire le richieste dei singoli client.**

Quando viene stabilita una connessione con un nuovo client, l'oggetto socket generato viene passato a una nuova istanza di **ClientHandler**. In questo modo, ogni client è gestito separatamente e in modo simultaneo dal server.

Il metodo `run()` della classe, viene eseguito su un nuovo thread, consentendo così al server di comunicare con più client contemporaneamente senza interferenze tra le varie connessioni.

Se il metodo `run()` di **ClientHandler** venisse eseguito sullo stesso thread principale, (quello che ascolta per nuove connessioni) il server resterebbe "bloccato" a gestire il singolo client e smetterebbe temporaneamente di ascoltare per nuove connessioni. Questo accadrebbe perché saremmo occupati a comunicare con il client connesso e potremmo tornare ad ascoltare nuove connessioni solo dopo che la connessione con il client attuale viene chiusa. Il punto di partenza di questa classe è il metodo `run()`, di seguito si procederà con la descrizione di alcuni dei metodi più rilevanti all'interno della classe.

Il metodo `run()` è essenziale per le comunicazioni tra client e server. Per prima cosa vengono aperti sull'oggetto socket dei flussi di comunicazione per leggere e scrivere al client (`in` e `out`, che sono rispettivamente `BufferedReader` e `PrintWriter`).

```
SISOP-PRJ/ClientHandler.java
56 while (clientRunning) { // Main loop for handling client messages
57     try {
58         messageFromClient = in.readLine(); // Blocking call
59         if (messageFromClient == null) {
60             break; // Client disconnected
61         }
62         processCommand(messageFromClient);
63     } catch (SocketTimeoutException e) {
64         if (!server.isRunning()) {
65             break;
66         }
67     } catch (IOException e) {
68         break;
69     }
70 }
```

La logica per gestire i comandi del client, è la stessa utilizzata nella classe Server, fino a quando il client è connesso, leggiamo i messaggi ricevuti e li processiamo attraverso il metodo `processCommand()` (essendo la logica uguale al metodo incontrato precedentemente non verrà descritto).

Se non viene rilevato nessun messaggio, significa che il client non è più connesso, perciò usciamo dal ciclo while tramite `break`. Di seguito descriviamo cosa avviene all'interno di `.readLine()` nella situazione in cui restituisce `null`:

1. **Il Client Chiude la Connessione:** Quando il client chiude la connessione del socket (sia chiudendo esplicitamente il socket che terminando il programma), segnala al server che non invierà più dati.
2. **Il Server Rileva la Fine del Flusso:** Sul lato server, quando viene chiamato `readLine()`, internamente viene utilizzato il metodo `read()` dello stream di input sottostante. Se il client ha chiuso la connessione, il metodo `read()` restituirà -1, indicando la fine del flusso.
3. **readLine() Restituisce null:** Dopo aver rilevato -1 da `read()`, il metodo `readLine()` sa che non ci sono più dati da leggere e restituisce `null`.

Il metodo `handleRegistration()` è lanciato attraverso il comando `publish` o `subscribe`, è fondamentale durante la fase di registrazione. Permette di assegnare un valore a variabili come `role` e `topic`, inviare un messaggio di conferma al client, aggiungere il

nuovo topic creato a **topics** e a **publisherMessages** se non è già esistente una chiave corrispondente.

```
SISOP-PRJ/ClientHandler.java
173 private void handleRegistration(String[] tokens) {
174     String role = tokens[0].toLowerCase();
175     topic = String.join("_", Arrays.copyOfRange(tokens, 1, tokens.length)); // "example topic" -> "example_topic"
176     isPublisher = role.equals("publisher"); // Important: Determine if the client is a publisher or subscriber
177
178     System.out.println("> Client (" + userID + ") registered as " + (isPublisher ? "publisher" : "subscriber") + " on '" + topic + "' .");
179     out.println("---- REGISTRATION SUCCESSFUL ---\n"
180             + "> Registered as " + (isPublisher ? "publisher" : "subscriber") + " on topic '" + topic + "' .\n"
181             + "> Enter 'help' for a list of available commands.\n");
182
183     topics.putIfAbsent(topic, new ConcurrentLinkedQueue<>()); // Ensure topic is added to topics map
184     publisherMessages.putIfAbsent(topic, new ArrayList<>()); // Ensure topic is added to client-specific map
185     if (server.isInspectingTopic(topic))
186         setIsServerInspecting(true); // Important: If server inspecting topic, notify client
187 }
```

Il metodo **broadcastMessage()** viene utilizzato per inviare i messaggi di un publisher registrato su un determinato topic a tutti i client iscritti al topic. Viene chiamato quando un client invia un messaggio tramite il comando “send”.

```
SISOP-PRJ/ClientHandler.java
195 private void broadcastMessage(String messageBody) {
196     Message message = new Message(userID, topic, messageBody);
197     topics.computeIfAbsent(topic, msgs -> new ConcurrentLinkedQueue<>()).offer(message); // Noticed NullPointerException without this
198     publisherMessages.computeIfAbsent(topic, msgs -> new ArrayList<>()).add(message); // Important: Store the message in the client's own list
199     clientHandlers.values()
200         .stream()
201         .filter(ch -> topic.equals(ch.topic))
202         .forEach(ch -> ch.out.println((ch != this ? "> MESSAGE RECEIVED:\n" : "> MESSAGE SENT:\n") + message));
203 }
```

Per prima cosa si va a creare un oggetto di tipo **message**, che rappresenta il messaggio da inviare. Il messaggio viene poi registrato nella lista associata al topic e nella lista dei messaggi del publisher, una nuova lista viene creata nel caso non esistesse nessuna lista associata per evitare **NullPointerExceptions**.

Viene ottenuto un oggetto **Stream**, sulla collezione **clientHandlers**.

Questa operazione ci permette di manipolare la collezione in modo più funzionale, dopo aver recuperato tutti i **clientHandler** che hanno il topic che si desidera, viene inviato loro il messaggio.

## Classe Client

Questa classe rappresenta il lato client della nostra applicazione, il client si connette al server, invia messaggi e richieste, e stampa i messaggi ricevuti in output.

Una volta creato un oggetto di tipo Client, vengono aperti sull'oggetto socket dei flussi di comunicazione per leggere e scrivere al server (`in` e `out`, che sono rispettivamente `BufferedReader` e `PrintWriter`).

Il metodo `start()` comunica al client quando viene stabilita una connessione con il server, e chiama i metodi `receiveMessage()` e `processCommand()`, per ricevere messaggi e inviare richieste, rispettivamente, al server.

Il metodo `receiveMessage()` viene eseguito su un thread differente da quello principale **per evitare deadlock** durante la comunicazione.

```
SISOP-PRJ/Client.java
67 private void receiveMessage() {
68     new Thread(() -> {
69         while (running) {
70             try {
71                 String messageFromServer = in.readLine();
72                 if (messageFromServer == null) {
73                     closeEverything();
74                 }
75                 handleMessageFromServer(messageFromServer);
76             } catch (IOException e) {
77                 if (!running) {
78                     break;
79                 }
80                 System.out.println("> Connection lost: " + e.getMessage());
81                 closeEverything();
82             }
83         }
84     }).start();
85 }
```

La logica di questo metodo è molto semplice, fino a quando il client è connesso, i messaggi provenienti dal server vengono letti e passati al metodo `handleMessageFromServer(messageFromServer)`, quest'ultimo metodo, sino a che non viene ricevuto il comando speciale (`"IS_SERVER_INSPECTING"`) come tokens[0], procede semplicemente a stampare il messaggio.

Quando viene ricevuto il comando speciale e il server non è in “inspect mode”, vengono eseguiti i comandi inviati dal client che erano disabilitati durante la modalità

interattiva. Se il client viene disconnesso per qualche motivo (**messageFromServer == null**), ci si assicura che il socket e i flussi di comunicazione (**in** e **out**) vengano chiusi correttamente.

Il metodo **processCommand(String inputLine)**, rimane in esecuzione fino a quando il client è connesso, attraverso questo metodo il client invia richieste al server.

Questa funzione viene richiamata a partire dal metodo **start()**, tutte le volte che si preme il tasto invio (**while (running) processCommand(scanner.nextLine())**).

Se viene passata una stringa vuota, si fa terminare il metodo, altrimenti l'input viene sanificato. Si procede poi accertandosi che il server non sia in “inspect mode”, per assicurarsi che le richieste del client non siano momentaneamente disabilitate.

I comandi messi a disposizione del client sono i seguenti:

- > help: Stampa i comandi messi a disposizione del client
- > show: Stampa la lista dei topic disponibili
- > send <message>: Invia un messaggio sul topic su cui il publisher è registrato
- > list: Stampa tutti i messaggi inviati dal publisher
- > listall: Lista di tutti i messaggi in the topic
- > quit: Termina la connessione con il server
- > publish <topicExample>: registra il client come publisher su topicExample
- > subscribe <topicExample>: registra il client come subscriber su topicExample

```
SISOP-PRJ/Client.java
116 switch (command) {
117 case "help" -> showHelp();
118 case "show" -> out.println("show");
119 case "send" -> handleSendCommand(tokens);
120 case "list" -> out.println("list");
121 case "listall" -> out.println("listall");
122 case "quit" -> {
123     out.println("quit");
124     closeEverything();
125 }
126 case "publish", "subscribe" -> handleRegistration(tokens);
127 default -> System.out.println("> Unknown command. Enter 'help' to see the list of available commands.\n");
128 }
```

Se la richiesta del client appartiene a uno dei case presenti, viene inviata al server, mentre se il comando non viene riconosciuto viene stampato un messaggio di errore.

La funzione `handleRegistration(String [] tokens)`, è abilitata attraverso uno dei due comandi `publish` o `subscribe`.

Viene utilizzata in fase di registrazione per definire il valore di variabili come `topic` e `isPublisher`, che ci permettono di determinare il ruolo (publisher o subscriber) di un client, e il topic al quale è registrato.

Oltre a permettere ad un nuovo client di registrarsi, questa funzione permette a un client già registrato di cambiare il proprio ruolo.

```
SISOP-PRJ/Client.java
191     if (isPublisher != null) { // Allow clients to change their role and topic
192         String newRole = tokens[0].toLowerCase();
193         String newTopic = String.join("_", Arrays.copyOfRange(tokens, 1, tokens.length));
194
195         // Check if the client is already registered as a publisher or subscriber for the same topic
196         if (isPublisher == newRole.equals("publish") && topic.equals(newTopic)) {
197             System.out.println("> You are already a '" + (isPublisher ? "publisher" : "subscriber") + "' for topic '" + topic + "'.\n");
198             return;
199         }
200
201         try {
202             System.out.print("> You are currently a '" + (isPublisher ? "publisher" : "subscriber") + "' for topic '" + topic + "'.\n"
203                         + "> Do you want to change your role and topic? (y/n): ");
204             String response = scanner.nextLine().toLowerCase();
205             if (!response.equalsIgnoreCase("y") && !response.equalsIgnoreCase("yes")) {
206                 System.out.println("> OK. Registration unchanged.\n");
207                 return;
208             }
209         } catch (Exception e) {
210             System.out.println("> Error reading from console: " + e.getMessage());
211             closeEverything();
212         }
213     }
}
```

Ci si serve della variabile booleana `isPublisher` per determinare il ruolo del client (true → publisher, e false → subscriber).

Questa variabile statica, inizialmente ha valore `null`, la condizione che permette di cambiare ruolo e topic ad un client è infatti `isPublisher != null` (significa che il client è già registrato).

Se il client non sta provando a registrarsi con lo stesso ruolo, sullo stesso topic, l'operazione andrà a buon fine dopo una richiesta di conferma, in caso contrario si termina il metodo.

Se il client è alla prima registrazione `isPublisher` avrà valore `null`, il codice all'interno dell'if statement non verrà eseguito e si proseguirà con il resto del codice.

```
SISOP-PRJ/Client.java
216     String role = tokens[0].toLowerCase();
217     isPublisher = role.equals("publish");
218     topic = String.join("_", Arrays.copyOfRange(tokens, 1, tokens.length)); // "example topic" -> "example_topic"
219     out.println(String.join(" ", tokens));
```

Si definiscono i valori di `isPublisher` e `topic`, e si completa la registrazione inviando la richiesta al server.

Si considerano le classi e i metodi illustrati, sufficienti per comprendere a livello concettuale l'architettura del progetto e le relazioni tra le varie componenti.

Lo scopo principale della documentazione è quella di dare un quadro generale che permetta all'utente di capire come utilizzare l'applicazione in modo corretto, e la logica alla base della nostra architettura.

Il motivo per il quale si è deciso di non descrivere in modo dettagliato tutti i metodi è il seguente:

- Si ritiene che la comprensione di alcuni metodi sia intuitiva per l'utente, perciò non necessitano di una descrizione.
- Oltre alle funzionalità richieste dal progetto, dal lato server sono state sviluppate e integrate funzionalità aggiuntive, non previste nella specifica iniziale. I metodi implementati per gestire queste funzionalità non verranno descritti in questo documento, verrà aggiunta però una breve descrizione del comando, in modo che l'utente possa comprendere a livello concettuale il suo funzionamento, e avere un'idea chiara di come utilizzarlo per ottenere i risultati desiderati.

## **Comandi Aggiuntivi Implementati**

- > **kick <idClient>**: Rimuove un client specifico dal server .
- > **export <idClient> o <idTopic>**: Salva su un file messaggi client o topic.
- > **user <idClient>**: Mostra informazioni su un utente specifico.
- > **users**: Mostra informazioni su tutti gli utenti connessi.
- > **clear**: Cancella tutti i messaggi di un topic in ispezione.

## (c) Suddivisione del Lavoro tra i Membri del Gruppo

Per la realizzazione del progetto, si è ritenuto opportuno suddividere equamente il carico di lavoro, in modo da permettere ogni membro del gruppo di lavorare sul progetto, e assimilare alcuni dei concetti fondamentali trattati nel corso.

Nella fase iniziale dello sviluppo dell'applicazione, la modalità di lavoro adottata prevedeva incontri in presenza.

Abbiamo optato per questo tipo di approccio perché siamo giunti alla conclusione che, al fine di lavorare simultaneamente sullo stesso codice, è necessario avere una logica di implementazione ben chiara, che permette ai singoli membri del gruppo di lavorare indipendentemente e allo stesso tempo avere un punto di riferimento su cui appoggiarsi.

Prima degli incontri, dopo aver letto le specifiche del progetto, ogni membro ha cercato attivamente sul web informazioni che potessero essere utili, in modo da confrontare le idee una volta in presenza e decidere come procedere.

Durante gli incontri in presenza, si è stabilita una logica di implementazione (sezione 1b), discusso quale fosse il modo più efficiente per gestire le operazioni su risorse condivise e su come gestire molteplici threads (optato per **java.util.Concurrent, synchronized, e ExecutorService**), deciso quali classi avrebbero fatto parte del progetto e come gestire le comunicazioni tra server e client (si è presa ispirazione dagli esercizi svolti durante il corso e dal materiale trovato sul web, un esempio potrebbe essere il video nel seguente link <https://youtu.be/gLfuZrrfKes?feature=shared>).

Il blocco iniziale del progetto, che permette al server di mettersi in ascolto, stabilire una connessione con un client, ascoltare per comandi, e il metodo `processCommand()` sono stati scritti durante questi incontri, per quanto riguarda i metodi relativi ai comandi presenti nei case della struttura di comando switch, sono stati implementati durante le sessioni interattive di coding (funzionalità messa a disposizione da IntelliJ), e comunicando tramite discord.

I metodi aggiuntivi e alcuni degli ultimi metodi implementati sono stati sviluppati lavorando da casa e comunicando con discord o WhatsApp.

Inizialmente si è utilizzato Eclipse come editor, poi per motivi di semplicità tutto il codice è stato spostato su IntelliJ, ogni modifica successiva nel corso dello sviluppo dell'intero progetto è stata salvata su GitHub.

Essendo il gruppo formato da sole due persone, è risultato molto intuitivo suddividere il carico del lavoro, ogni modifica del codice da parte di un membro, è stato comunicato con un messaggio di commit, discusso negli incontri online, e testato dall'altro membro.

Questo perché per alcuni metodi lavorare in due avrebbe rallentato il processo di lavoro invece di velocizzarlo, visto i diversi approcci e punti di vista, la fase decisionale avrebbe richiesto molto tempo della fase di implementazione.

## 2. Descrizione e Discussione del Processo di Implementazione

### (a) Descrizione dei Problemi

**N.B.** Dentro a questa lista ricadono alcuni dei problemi (concettuali/tecnicici) isolati che abbiamo affrontato e risolto, ma la lista non è esaustiva poiché vengono descritte in altre sezioni della documentazione funzionalità del progetto (e cosa abbiamo dovuto fare per farle funzionare), come per esempio con il sistema del backlog (sezione 3b).

- “**Scope Creep**”

Il termine “scope creep” si riferisce allo scenario in cui vengono aggiunte sempre più funzionalità ad un progetto, che man mano lo fanno fuorviare dal proprio core e scopo originale. Ci sono quasi stati svariati di questi momenti e tentazioni in questo progetto, di cui la più notevole sarebbe stata l’aggiunta di una base di dati inizialmente relazionale, e poi NoSQL (spiegata nella sezione 3b).

La soluzione a questo fenomeno l’abbiamo trovata attenendoci il più possibile alle specifiche del progetto, il rischio sarebbe stato quello di aumentare a dismisura il carico di lavoro e rischiare di non terminare il progetto nei tempi prefissati.

- **Nuovi clienti che si connettono ad un topic già in modalità ispezione**

Inizialmente quando il server ispezionava un topic, gli unici client ad essere sottoposti alle restrizioni poste dall’ispezione erano quelli già connessi al topic da prima. Questo voleva dire che se un nuovo client si connette dopo che il server ha già cominciato l’ispezione, sarebbe stato libero di usare comandi in teoria banditi.

La soluzione a questo problema è stata di introdurre il controllo dello stato di ispezione del topic, oltre al meccanismo pre-esistente spiegato nelle sezioni precedenti, al momento di registrazione del client, catturando così ogni possibile utente. Il tutto è stato agevolato da un flag booleano, **isServerInspecting**, che indirizza l’esecuzione anche del resto dei comandi disponibili al client.

- **Passaggio da CopyOnWriteArrayList a ConcurrentLinkedQueue**

CopyOnWriteArrayList è ideale per applicazioni con un elevato numero di operazioni di lettura. La nostra applicazione combina operazioni di lettura (subscriber/publisher) e di scrittura (publisher), pertanto è più adatta l'adozione di una struttura come ConcurrentLinkedQueue, in quanto non ricrea l'intero array dei messaggi del topic ogni volta che viene utilizzato un comando "send" ("CopyOnWrite"). Inoltre, considerando che i sottoscrittori non possono richiedere messaggi specifici basati sull'ID, e gli UNICI casi in cui ciò avviene è quando il server entra in modalità ispezione e specificamente elimina un messaggio, si annulla l'unico vero vantaggio di efficienza che un array ha rispetto a una struttura dati basata su nodi, come la traversabilità. Considerando che la maggior parte delle operazioni di lettura saranno effettuate da publisher/subscriber utilizzando i comandi "list" e "listall", il costo di runtime di entrambe le operazioni è O(n), indipendentemente dal fatto che si tratti di un array dinamico o di una lista collegata.

- **Conflitto di System.in tra il metodo processCommand() e clearTopic()**

L'errore si è verificato perché si è utilizzata un'istanza di Scanner per leggere da System.in su due punti diversi del tuo codice contemporaneamente. L'istanza di Scanner creata in Server.processCommand() stava già utilizzando System.in, quando Server.clearTopic() tentava di creare un'altra istanza di Scanner sullo stesso System.in, questo conflitto ci ha portato a un **Line not found error**.

Per risolvere questo problema e continuare a leggere dalla console in clearTopic(), abbiamo passato l'istanza esistente di Scanner creata in processCommand a clearTopic() come parametro. In questo modo, esisterà una sola istanza di Scanner che legge da System.in, evitando conflitti e comportamenti imprevedibili.

- **Possibile conflitto di System.in start() e handleRegistration()**

Nonostante ci trovassimo nella stessa situazione del problema appena descritto, ci siamo accorti di questo problema soltanto quando invece di utilizzare il tradizionale try-catch, abbiamo utilizzato un **try-with-resources** (`try (Scanner scanner = new Scanner(System.in))`), questo perché il blocco garantisce che la risorsa (scanner in questo caso) venga automaticamente chiusa al termine del blocco, chiamando il metodo close() sulla risorsa.

Al termine del blocco **try-with-resources**, viene chiamato automaticamente il metodo close() sull'oggetto Scanner, il quale chiude lo stream sottostante, ovvero **System.in**.

Inizialmente abbiamo optato per l'utilizzo del blocco **try-catch** per risolvere il problema, una volta stabilita la natura del problema però, si è introdotta una variabile

**scanner** globale.

Avremmo potuto passare l'istanza di scanner creata nel metodo **start()** al metodo **handleRegistration()** ma abbiamo preferito la prima soluzione, perché riteniamo che renda il nostro codice più robusto.

- Dopo aver cancellato messaggi durante inspect mode non venivano rimossi dai messaggi del client

Ci siamo accorti di questo errore perché una volta cancellati messaggi durante inspect mode, venivano soltanto rimossi dai messaggi sul topic (**topics**) , ma non dai messaggi inviati dal singolo client (**publisherMessages**).

Perciò una volta lanciati i comandi “**list**” e “ **listall**”, messaggi eliminati e non più presenti in **listall** erano presenti in **list**.

Per risolvere questo problema si è introdotto il metodo **deleteFromClient(int msgID)**, questo metodo per ogni **clientHandler** esistente, recupera l'**ArrayList<message>** relativo al topic che si sta ispezionando, se trova un messaggio con lo stesso **id**, lo rimuove e ritorna **true**, altrimenti ritorna **false**.

## (b) Descrizione degli Strumenti Utilizzati per l'Organizzazione

### Sviluppo del progetto: IntelliJ IDEA, Excalidraw

- Separatamente con aggiornamenti e controllo delle versioni mediante GitHub. Ci siamo sentiti quotidianamente via messaggi per tenere traccia dell'andamento del progetto.
- Congiuntamente attraverso la modalità interattiva dell'IDE: *Code With Me*. Di base bisettimanalmente, per aggiornare l'un l'altro sulle proprie aggiunte e/o modifiche del codice, per testare il funzionamento corretto di ciascuna *feature*, e per stabilire come procedere nei seguenti giorni.
- Per la creazione di schemi (alcuni dei quali presenti nella documentazione), abbiamo utilizzato Excalidraw, un'applicazione web gratuita che permette di creare e modificare in diretta disegni e schemi condivisi.

### Comunicazione del gruppo: Discord & WhatsApp

- Via Discord abbiamo comunicato in chiamate vocali durante le sessioni interattive (*Code With Me*) per quanto riguardo lo sviluppo del progetto, ma anche per discussioni riguardanti l'architettura delle classi, e per la stesura della documentazione.
- Via WhatsApp ci siamo tenuti aggiornati quotidianamente sullo status del progetto, e per organizzare date di incontro.

### Condivisione del progetto: GitHub

- Abbiamo condiviso il codice del progetto attraverso GitHub. In particolare, abbiamo tenuto traccia delle versioni del progetto all'interno di IntelliJ IDEA, il quale dispone di un plugin ufficiale che integra le funzionalità di GitHub direttamente nell'IDE.

### Svolgimento del lavoro: GitHub / IntelliJ, Commenti Codice, Discord

- Come menzionato in precedenza, abbiamo utilizzato GitHub per tenere traccia delle versioni del progetto. In particolare, ci siamo affidati ai *commit messages* di ciascuno per avere un'idea generale delle modifiche e/o implementazioni nel codice, e di strumenti di confronto del codice disponibili all'interno di IntelliJ, permettendoci di confrontare il codice più aggiornato con versioni precedenti.

- Abbiamo inoltre mantenuto a capo della classe Server una lista (in continuo aggiornamento tra commit) di punti da svolgere, suddivisa in una lista prioritaria con le cose più importanti da controllare o implementare, e una lista secondaria contenente idee e potenziali espansioni del progetto. Ciascuna lista era organizzata in base all'importanza di ciascun punto, con le entry più importanti in cima alla lista, e quelle meno importanti in fondo.
- Attraverso chiamate su Discord abbiamo avuto molte discussioni e riflessioni sulla progettazione delle classi. La possibilità di condividere i propri schermi è stata particolarmente utile anche per la discussione di frammenti di codice per chiarire certe decisioni sull'implementazione e logica (per esempio) di funzioni.

### **3. Requisiti e Istruzioni per Compilare e Usare le Applicazioni Consegnate**

#### **(a) Esempi degli output attesi per ogni comando**

##### **Server**

---

“show”:

- **Topic esistente:**

```
--- SHOW: EXISTING TOPICS ---  
  
--- TOPIC: topic_1  
> PUB: 1  
> SUB: 2  
> MSG: 3  
  
--- TOPIC: topic_2  
> PUB: 1  
> SUB: 0  
> MSG: 1  
  
--- END OF TOPIC LIST ---
```

- **Nel caso in cui non esistano alcuni topic nella sessione:**  
> No existing topics available.

##### **Dove:**

PUB → Publisher  
SUB → Subscriber  
MSG → Numero di messaggi inviati nel topic

---

“quit”:

```
> (PRE-QUIT) Connected clients: 4  
> Client 0 disconnected. Clients currently connected: 3.  
> Client 1 disconnected. Clients currently connected: 2.  
> Client 2 disconnected. Clients currently connected: 1.  
> Client 3 disconnected. Clients currently connected: 0.  
> (POST-QUIT) Connected clients: 0
```

---

## “inspect”:

- **Topic esistente:**

```
inspect topic 1
--- INSPECT MODE STARTED ---
> Begun inspecting topic 'topic_1'. Enter 'help' for a list of available commands.
```

- **Topic non esistente:**

> Topic 'topic\_1' does not exist.

---

## “end”:

- **Inspect mode:**

```
> Exited inspect mode for topic 'topic_1'.
--- INSPECT MODE ENDED ---
```

- **No inspect mode:**

> Command 'end' is only available in inspect mode.

---

## “listall”:

- **Inspect mode:**

```
--- LISTALL: 3 MESSAGES IN 'topic_1' ---

-----
USER-0 @ 22/11/2024 - 10:16:41
[ID 0 | TOPIC 'topic_1']
BODY: Mess1
-----

-----
USER-0 @ 22/11/2024 - 10:16:46
[ID 1 | TOPIC 'topic_1']
BODY: Mess2
-----

-----
USER-1 @ 22/11/2024 - 10:17:04
[ID 2 | TOPIC 'topic_1']
BODY: Mess3
-----

--- LISTALL: END OF MESSAGES IN 'topic_1' ---
```

- **No inspect mode:**

> Command 'listall' is only available in inspect mode.

---

#### “**delete <messageID>**”:

- **Inspect mode:**

> (SUCCESS) Message with ID 2 deleted.

- **Commando non corretto:**

> Usage: delete <messageID> (see 'listall' for valid id's)

- **No inspect mode:**

> Command 'delete' is only available in inspect mode.

---

#### “**help**”:

- **Inspect mode:**

```
--- HELP: AVAILABLE COMMANDS ---
> kick <userID>: Kick a client by ID
> export user <userID>: Export all messages of a user to logs/user_exports
> export topic <topic>: Export all messages of a topic to logs/topic_exports
> users: Show all connected users and their details
> user <userID>: Show details of a specific user
> listall: List all messages in the topic
> delete <messageId>: Delete a message by ID
> clear: Clear all messages in the topic being inspected
> end: Exit interactive mode

! N.B. Commands 'quit' & 'inspect' are disabled in interactive mode,
      client's 'send', 'list', 'listall' are suspended until the mode is exited.
--- END OF HELP ---
```

- **No inspect mode:**

```
--- HELP: AVAILABLE COMMANDS ---
> kick <userID>: Kick a client by ID
> export user <userID>: Export all messages of a user to logs/user_exports
> export topic <topic>: Export all messages of a topic to logs/topic_exports
> users: Show all connected users and their details
> user <userID>: Show details of a specific user
> show: Show available topics
> inspect <topic>: Open interactive mode to inspect a topic (list all messages, delete messages, etc.)
> quit: Disconnect from the server
--- END OF HELP ---
```

---

#### “kick <userID>”:

- **Comando utilizzato correttamente**

```
kick 0
> Client 0 has been kicked from the server.
> Client 0 disconnected. Clients currently connected: 1.
```

- **Comando non utilizzato correttamente**

```
> Usage: kick <userID>
```

---

#### “clear”:

- **Inspect Mode**

```
> Are you sure you want to clear all messages in topic 'topic_1'? (y/n): y
> All messages in topic 'topic_1' have been cleared.
```

- **No Inspect Mode**

```
> Command 'clear' is only available in inspect mode
```

---

#### “export topic <topic>”:

- **Non ci sono messaggi**

```
> No messages available for topic 'topic_1'.
```

- **Ci sono messaggi**

> Messages in topic 'topic\_1' exported to  
'logs/topic\_exports/export\_20241122\_161205\_topic\_topic\_1.txt'.

---

“**export user <Id>**”:

- **User esistente**

> Messages for user ID 0 exported to  
'logs/user\_exports/export\_20241122\_162230\_user\_0.txt'.

- **User non esistente**

> User ID 1 not found.

---

“**users**”:

- **Users esistenti**

```
--- SHOW: ALL USERS ---  
  
--- USER ID 0 ---  
> CURRENT TOPIC: topic_1  
> CURRENT ROLE: Publisher  
> MESSAGES SENT: 3  
  
--- USER ID 1 ---  
> CURRENT TOPIC: topic_2  
> CURRENT ROLE: Publisher  
> MESSAGES SENT: 0  
  
--- END OF ALL USERS ---
```

- **No users esistente**

> No users connected.

---

“**user <ID>**”:

- **Id esistente**

```
--- SHOW: USER ID 0 ---  
> CURRENT TOPIC: null  
> CURRENT ROLE: Unregistered
```

```
> MESSAGES SENT: 0  
--- END OF USER DETAILS ---
```

- **Id non esistente**

```
> User ID 2 not found.
```

## Client

---

“**help**”:

- **Prima della registrazione**

```
--- HELP: AVAILABLE COMMANDS ---  
> [publish | subscribe] <topic>: Register as publisher (read-write) or subscriber  
(read-only) for <topic>  
> show: Show available topics  
> quit: Disconnect from the server
```

- **Publisher - Inspect Mode**

```
--- HELP: AVAILABLE COMMANDS ---  
* > send <message>: Send a message to the server  
* > list: List the messages you have sent in the topic  
* > listall: List all messages in the topic  
> show: Show available topics  
> quit: Disconnect from the server  
  
* Commands marked with an asterisk(*) are disabled during Inspect mode.  
Any usage of these commands will be queued and will execute once Inspect mode is e  
nded.
```

- **Subscriber - Inspect Mode**

```
help  
--- HELP: AVAILABLE COMMANDS ---  
* > listall: List all messages in the topic  
> show: Show available topics  
> quit: Disconnect from the server  
  
* Commands marked with an asterisk(*) are disabled during Inspect mode.  
Any usage of these commands will be queued and will execute once Inspect mode is e  
nded.
```

- **Publisher - No Inspect Mode**

```
|> send <message>: Send a message to the server  
|> list: List the messages you have sent in the topic  
|> listall: List all messages in the topic  
|> show: Show available topics  
|> quit: Disconnect from the server
```

- **Subscriber - No Inspect Mode**

```
|--- HELP: AVAILABLE COMMANDS ---  
|> listall: List all messages in the topic  
|> show: Show available topics  
|> quit: Disconnect from the server
```

---

“**show**”: Identico a Server

---

“**send <Message>**”:

- **Publisher**

```
|> MESSAGE SENT:  
|-----  
|USER-1 @ 22/11/2024 - 16:55:42  
|[ID 0 | TOPIC 'topic_1']  
|BODY: 2  
|-----
```

- **Subscriber**

> You are registered as a subscriber. You cannot send messages.

---

“**listall**”: Identico a Server

---

“list”:

- **Publisher**

```
--- LIST: YOU SENT 2 MESSAGES IN 'topic_1' ---  
  
-----  
USER-1 @ 22/11/2024 - 16:55:42  
[ID 0 | TOPIC 'topic_1']  
BODY: 2  
-----  
  
-----  
USER-1 @ 22/11/2024 - 16:58:10  
[ID 1 | TOPIC 'topic_1']  
BODY: 3  
-----
```

- **Subscriber**

> You need to register as a publisher first.

- **Publisher - No messaggi inviati**

> You have not sent any messages in 'topic\_1'.

---

“quit”: --- CLIENT SHUTDOWN ---

---

“publish”:

--- REGISTRATION SUCCESSFUL ---

> Registered as 'publisher' on topic 'topic\_1'.  
> Enter 'help' for a list of available commands.

---

“subscribe”:

--- REGISTRATION SUCCESSFUL ---

> Registered as 'subscriber' on topic 'topic\_1'.  
> Enter 'help' for a list of available commands.

---

## (b) Descrizione delle Estensioni Implementate e Come Usarle

### Backlog System

Anche se non fondamentale, abbiamo ritenuto necessario fare sì che quando il server sta ispezionando un topic, e pertanto i client non hanno modo di usare i comandi “send”, “list”, e “listall”, piuttosto che ignorare le loro richieste, abbiamo implementato un sistema client-side che tiene traccia dei comandi tentati dall’utente, eseguendoli automaticamente al termine dell’inspect mode.

All’interno di **Client.processCommand()**, la funzione centrale di Client, prima di eseguire il comando, viene controllato se il topic è attualmente ispezionato dal server (inspect mode), e se sì (dopo aver verificato il ruolo dell’utente), il comando viene aggiunto a **backlog** (un semplice ArrayList di stringhe), notificando l’utente.

```
SISOP-PRJ/Client.java
102 if (isServerInspecting && disabledWhenInspecting.contains(command)) {
103     if (!isPublisher && publisherOnlyCommands.contains(command)) {
104         System.out.println("> You cannot use the command '" + command + "' as a subscriber.\n");
105     } else {
106         System.out.println(command.equals("listall")
107             ? "> Command '" + inputLine + "' will execute last (to avoid inconsistencies) when Inspect mode is ended.\n"
108             : "> Command '" + inputLine + "' has been queued and will execute when Inspect mode is ended.\n");
109         backlog.add(inputLine);
110     }
111 }
112 }
```

Quando il server termina l’ispezione, invia a tutti i membri del topic una stringa speciale (“**IS\_SERVER\_INSPECTING false**”) che non viene stampata sulla console del client, ma che invece fa eseguire con **executeBacklogCommands()** i comandi accumulati dall’utente durante l’ispezione del topic.

```
SISOP-PRJ/Client.java
234 if (tokens[0].equals("IS_SERVER_INSPECTING")) {
235     isServerInspecting = Boolean.parseBoolean(tokens[1]);
236     if (!isServerInspecting) {
237         executeBacklogCommands();
238     }
239     return;
240 }
```

Assumendo che il backlog non sia vuoto, prima di eseguire i comandi, viene posto ogni utilizzo di “list” e “listall” al fondo della lista. L’impiego di questa scelta risolve un problema analogo a quello risolto dall’utilizzo di StringBuilder: L’ordine erroneo di stampa dei messaggi in output. Ponendo questi due comandi al termine del backlog di ogni client, permette prima di far eseguire qualunque “send” richiesto, e solamente

all'ultimo, avendo esaurito ogni possibile invio di messaggi, si stampano liste. Applicando questa logica ad ogni backlog di ogni client, si permette di evitare il più possibile errori di stampa e preservare ogni comando inviato dagli utenti durante l'inspect mode. Questo problema nasce dal fatto che in questa situazione, quando vengono eseguiti i comandi nel backlog, accade tutto, per ogni client, istantaneamente (in media 0.4ms - 0.9ms eseguendo 5 - 10 comandi), e non in secondi come lo si può presumere da umani.

```
SISOP-PRJ/Client.java
255 backlog.sort((a, b) -> { // Sort "list" and "listall" commands to be executed last to avoid interleaving
256     if (a.startsWith("list") && !b.startsWith("list")) {
257         return 1;
258     }
259     if (!a.startsWith("list") && b.startsWith("list")) {
260         return -1;
261     }
262     return 0;
263 });

```

Si stampa la lista di comandi accumulati da eseguire, ed uno ad uno, vengono eseguiti con **processCommand()**, il quale salterà l'if iniziale che controlla se il server sta ispezionando il topic, e entrerà nella già menzionata lista di comandi definiti nel switch-case. Infine, anche se non strettamente necessario, si svuota il backlog nel caso eccezionale (per qualunque ragione) in cui non vengono eseguiti ogni comando nel for-each precedente.

```
SISOP-PRJ/Client.java
265     System.out.println("--- COMMANDS TO BE EXECUTED ---");
266     int i = 1;
267     for (String cmd : backlog) {
268         System.out.println("> " + (i++) + ": " + cmd);
269     }
270     System.out.println();
271
272     for (String cmd : backlog) {
273         processCommand(cmd);
274     }
275     backlog.clear();

```

## Comandi Server-side

Un’ulteriore estensione che abbiamo fatto riguarda i comandi previsti per il server:

- kick <idClient>
- export <[idClient | idTopic]>
- user <userId>
- users
- clear

La motivazione per l’aggiunta di questi comandi risiede nel ragionamento che avendo a disposizione anch’essi, il server ricopre un ruolo ancora più dinamico e attivo nello svolgimento di una sessione.

Con “kick”, il server può scacciare client malintenzionati, e si può immaginare l’utilizzo di questo comando dopo che il server abbia eliminato uno o più messaggi dell’utente.

Con “export”, il server, per qualunque ragione (di sicurezza), ha la possibilità di esportare un log immediato di tutti i messaggi inviati da un utente, oppure di un intero topic.

Con “user” e “users”, il server ha modo di ottenere le statistiche di un utente, oppure della sessione corrente, rispettivamente.

Con “clear”, infine ed in maniera analoga a “delete”, il server ha modo di eliminare un intero topic, piuttosto che limitarsi ad un messaggio alla volta. Anche questo comando è posto solamente all’interno della modalità inspect, essendo una procedura delicata con conseguenze permanenti.

## NoSQL DB

Un’estensione finale che avevamo progettato e su cui avevamo ragionato, ma che non abbiamo avuto tempo di implementare, era quella di connettere il server ad una base di dati NoSQL esterna, come per esempio con un cluster in MongoDB, oppure con un *Firebase Database* di Firebase. Avevamo anche considerato la possibilità di modellare ed implementare una base di dati relazionale in MySQL, ma questa scelta avrebbe allungato ancora di più il nostro percorso, oltre che a complicarlo, perché utilizzando invece per esempio Firebase, avremmo avuto a disposizione una libreria di comandi con i quali abbiamo già un po’ di familiarità.

Abbiamo optato di non implementare una base di dati (a prescindere da SQL o NoSQL) anche perchè avrebbe necessitato che ri-scrivessimo buona parte dei metodi che a quel punto avevamo già finito e testato. In tutti i modi, di seguito viene esplicitato il modo in cui avremo memorizzato i dati generati dall’applicazione in una struttura/collezione NoSQL:

```

Clients {
    user_1: {
        username: STRING
        password: STRING
        isPublisher: BOOLEAN → [null | prev. session role]
        localMessages: {
            topic_1: {
                contents: LISTA (messaggi di user_1 in topic_1)
            },
            topic_2: {...},
            ...,
            topic_m: {...}
        }
    },
    user_2 {...},
    ...
    user_k {...}
},
Messages {
    topic_1: {
        contents: LISTA (ogni messaggio in topic_1)
    },
    topic_2: {...},
    ...
    topic_n: {...}
}

```

La struttura della base di dati è relativamente semplice: Disponiamo di una collezione contenente gli utenti/client registrati sulla piattaforma, ed un'altra contenente i messaggi presenti sulla piattaforma suddivisi nei propri topic.

Seguendo questo approccio, avremmo aggiunto in fase di registrazione per il client anche un campo per lo username ed un altro per la password, in tal modo da poter far autenticare e discernere fra sessioni gli utenti preesistenti, e nuovi, sulla piattaforma.

Un'ulteriore considerazione importante da fare è quella della presenza di **localMessages** in **Clients** (equivalente alla hashmap **publisherMessages** in **ClientHandler** nell'implementazione corrente del progetto) che inizialmente sembra solamente una ridondanza che occupa spazio inutilmente (un raddoppio della complessità spaziale nel database per precisione), ma che invece è fondamentale per l'efficienza dell'esecuzione del comando client-side (dei publisher) “**list**”. Questo perché se non ci fosse **localMessages** nel database, allora per poter eseguire “**list**” l'applicazione dovrebbe andare a scandire **l'intera lista di messaggi del topic** a cui il client appartiene, **e filtrare uno ad uno ogni messaggio** non appartenente al client (con n messaggi nel topic, una complessità temporale pari a  $\Theta(n)$ ). Avendo a

disposizione invece **localMessages**, il costo dell'operazione “**list**” si riduce notevolmente, dovendo semplicemente iterare sulla lista locale di messaggi dell'utente per un determinato topic (con m messaggi inviati dall'utente nel topic, una complessità temporale pari a  $\Theta(m)$ , dove  $m < n$ ). Per convincersi basta immaginare un utente che richiede con “**list**” i 100 messaggi (m) inviati in un topic contenente 1.000.000 messaggi (n).

Per quanto riguarda modifiche all'attuale implementazione del progetto, si può immaginare che metodi come **Server.startServer()** conterrebbero un'apposita sezione dedicata al popolamento dei topic e messaggi pre-esistenti, ed in fase di autenticazione per i client avverrebbe un controllo sulle credenziali rispetto alla lista di utenti presenti nel database.