

Università degli studi di Napoli Parthenope

Laboratorio di Reti di Calcolatori

Anno 2023/2024

Progetto

Server IRC

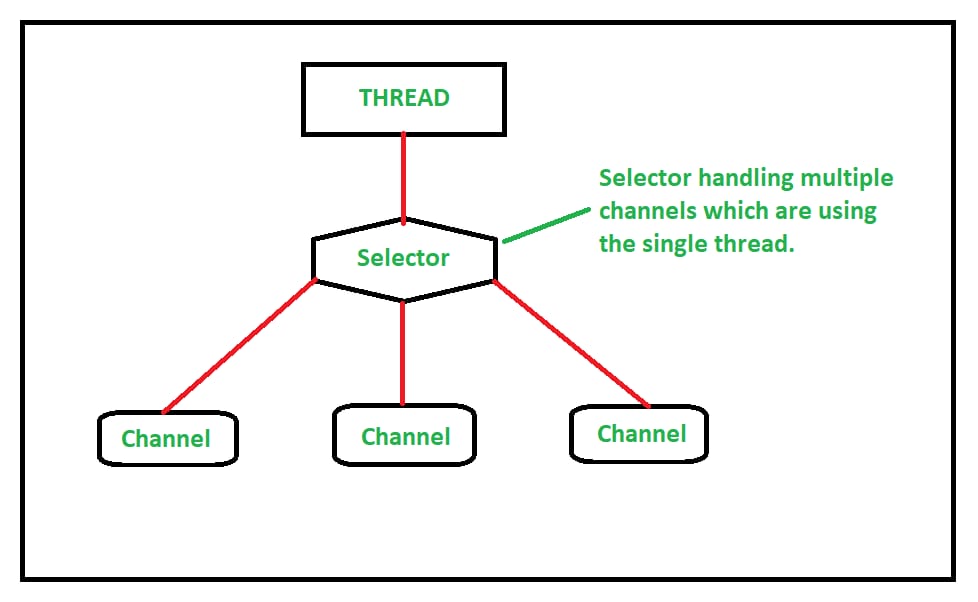
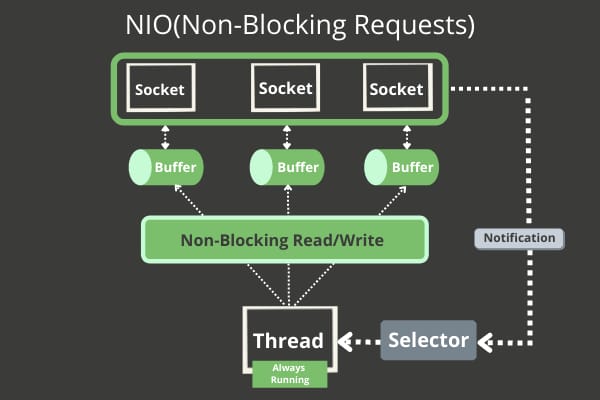
Marzocchi Pasquale 0124/001891

Leonardo Pulzone 0124/002502

Descrizione del progetto;

L'obiettivo del progetto è implementare un server IRC (Internet Relay Chat) che supporta la comunicazione in tempo reale tra più utenti. Un server IRC consente agli utenti di connettersi, unirsi a diversi canali di chat, inviare messaggi pubblici o privati, e gestire i permessi degli utenti (come bannare o espellere un utente). Il progetto implementa un'architettura client-server utilizzando il linguaggio Java, in cui il server gestisce i vari client connessi e i loro comandi.

Descrizione e schema dell'architettura;



Dettagli implementativi dei client/server;

**Server IRC**

Il server IRC è implementato per gestire la comunicazione tra i client connessi. Il server riceve comandi dai client, li elabora, e gestisce l'invio delle risposte o dei messaggi tra i client stessi.

* **Tecnologia Utilizzata**: Il server è scritto in Java e utilizza la libreria java.nio per la gestione delle connessioni di rete non bloccanti (NIO - Non-blocking I/O), che permette al server di gestire molteplici connessioni simultaneamente in modo efficiente.
* **Gestione delle Connessioni**: Il server ascolta le connessioni in entrata su una porta specifica. Utilizza ServerSocketChannel per accettare le connessioni dei client e SocketChannel per la comunicazione con ciascun client connesso.
* **Gestione dei Comandi**: Il server utilizza la classe CommandHandler per instradare i comandi ricevuti dai client alle classi appropriate che implementano l'interfaccia Command.
* **Gestione dello Stato**: Le informazioni sugli utenti connessi, come i loro nomi, i canali a cui sono iscritti, e i loro privilegi, sono mantenute in memoria utilizzando oggetti Java. Le operazioni di gestione (es. login, ban, messaggi privati) sono delegate a classi gestori come GestoreLogin e GestoreBanUtente.

**Client IRC (SimpleIrcClient.java)**

Il client IRC è un'applicazione Java che permette agli utenti di connettersi al server, autenticarsi, e interagire con altri utenti attraverso comandi.

* **Connessione al Server**: Il client si connette al server specificando l'indirizzo IP e la porta del server. Utilizza Socket per creare una connessione TCP.
* **Invio e Ricezione di Messaggi**: Il client invia comandi al server e riceve le risposte tramite flussi di input/output (InputStream e OutputStream).
* **Comandi Utente**: Il client supporta vari comandi, come login, msg (messaggio), join (unirsi a un canale), ban (bannare un utente), ecc. Ogni comando viene inviato al server sotto forma di stringa formattata.

Parti rilevanti del codice sviluppato;

#### ClientWriterImpl

La classe ClientWriterImpl si occupa di gestire l'invio dei messaggi dal server verso i client. Implementa un'interfaccia ClientWriter che definisce i metodi per scrivere dati su una connessione client.

**Funzionalità**:

* **Conversione del messaggio in byte:**  
  Converte il messaggio in un array di byte utilizzando UTF-8 e lo memorizza in un ByteBuffer insieme alla sua lunghezza.
* **Gestione dei dati parzialmente inviati:**  
  Se non è possibile inviare tutti i dati in una sola operazione (ad esempio, a causa di buffer di rete pieni), la classe memorizza i dati rimanenti in pendingData e registra il SocketChannel per ulteriori operazioni di scrittura.
* **Sincronizzazione dell'accesso:**  
  Il metodo writeToClient è synchronized, e garantisce che solo un thread alla volta possa eseguire la scrittura su un determinato SocketChannel, evitando problemi di concorrenza.

**GestioneLetturaClient**

GestioneLetturaClient è responsabile della lettura dei dati inviati dai client al server. Gestisce la ricezione dei comandi e dei messaggi dai client connessi.

**Funzionalità**

* **Gestione delle connessioni e dei messaggi client:**  
  GestioneLetturaClient legge i dati dal client, identifica i messaggi completi, e gestisce le disconnessioni.
* **Supporto per JSON e gestione dei comandi:**  
  Utilizza un parser JSON per interpretare i messaggi e un CommandHandler per gestire i comandi inviati dai client.
* **Robustezza e gestione delle eccezioni:**  
  Gestisce le disconnessioni e gli errori di I/O in modo ordinato, garantendo la stabilità del server anche in caso di errori di rete.

GestioneLetturaClient si occuperà di mandare il comando del client al commandhandler così da occuparsi di gestire il comando venuto dal client

**GestoreAccettazione**

GestoreAccettazione è responsabile di accettare nuove connessioni dai client in entrata. Utilizza un ServerSocketChannel per ascoltare nuove connessioni e crea un nuovo gestore per ciascun client connesso.

**Funzionalità**

* **Accettazione delle connessioni client:** La classe GestoreAccettazione gestisce l'accettazione delle nuove connessioni client tramite il metodo Accettazione. Quando un nuovo client si connette al server, la connessione viene accettata dal ServerSocketChannel.
* **Configurazione del canale del client:** Dopo aver accettato una nuova connessione, la classe configura il SocketChannel del client per funzionare in modalità non bloccante. Questo è essenziale per gestire più connessioni contemporaneamente senza bloccare il thread principale.
* **Registrazione per le operazioni di lettura:** Il SocketChannel del client viene registrato con un Selector per monitorare quando ci sono dati disponibili per la lettura.
* **Inizializzazione del buffer:** Per ogni nuovo client, viene creato e memorizzato un ByteBuffer per gestire i dati che saranno letti dal client.
* **Feedback di connessione:** Fornisce un feedback immediato nel terminale, informando che un nuovo client è connesso al server, inclusa l'informazione sull'indirizzo remoto del client.

**GestoreDisconnesioneClient**

GestoreDisconnesioneClient si occupa di gestire la disconnessione dei client dal server. Questa classe è responsabile di chiudere i socket, rimuovere il client dalla lista di connessioni attive e gestire eventuali risorse o cleanup necessari.

* **Gestione della disconnessione dei client:**  
  Rimuove il client disconnesso da tutte le strutture dati del server, inclusi i buffer di dati in sospeso, le associazioni di canali e utenti, e le informazioni sugli utenti duplicati.
* **Pulizia e liberazione delle risorse:**  
  Cancella la SelectionKey del client e chiude il canale, garantendo che tutte le risorse siano correttamente liberate.
* **Gestione degli ID temporanei:**  
  Mantiene l'integrità delle informazioni sugli utenti, rimuovendo gli ID temporanei associati agli utenti disconnessi e aggiornando i contatori.

Abbiamo implementato un sistema per gestire i nomi duplicati, in cui ogni persona ha un UUID, però visto che era complicato identificare un persona con un UUID, il server implementa una struttura dati che gestisce dinamicamente gli id con gli id temporanei, che sono più semplici da utilizzare.

**Command**

L'interfaccia Command definisce un contratto per una classe che rappresenta un comando eseguibile nel contesto di una connessione di rete tramite un canale di comunicazione SocketChannel. È una parte di un'applicazione Java che gestisce l'interazione tra un client e un server. E’ un’interfaccia che estende gli altri comandi, i quali poi richiama i comandi veri, i concrete command che chiamano i vari gestori

### Scopo dell'interfaccia Command

L'interfaccia Command rappresenta probabilmente un "comando" generico che può essere eseguito in risposta a un evento di rete (come un messaggio ricevuto da un client). Implementando questa interfaccia, diverse classi possono definire azioni specifiche da eseguire quando il metodo execute viene chiamato.

Per il server IRC, viene utilizzato di un pool di thread per la gestione degli invii dei messaggi consente di migliorare il throughput e l'efficienza del server. Invece di inviare messaggi in sequenza, il server utilizza più thread per gestire gli invii in modo concorrente, garantendo che i messaggi raggiungano i client in modo rapido e affidabile.

Questa strategia è particolarmente utile in applicazioni di chat in tempo reale, dove è importante che i messaggi vengano consegnati tempestivamente a un gran numero di utenti.

**Comandi del Client**

Una volta connesso, il client può inviare i seguenti comandi:

* **Login**: /login <username> <password> - Serve ad autenticarsi sul server.
* **Unirsi a un Canale**: /join <canale> - Serve ad unirsi o creare un nuovo canale.
* **Inviare Messaggi**: /msg <canale> <messaggio> - Serve ad inviare un messaggio pubblico a un canale.
* **Messaggio Privato**: /privmsg <utente> <messaggio> - Serve ad inviare un messaggio privato a un altro utente.
* **Lista canali**: /list - Serve a mostrare la lista canali
* **Utenti**: /users - Serve a mostrare gli utenti presenti nel canale
* **Mostra info Utente**: /fusers Serve ad ottenere le informazioni degli utenti del canale
* **Cambio canale**: /switchchannel - Serve a cambiare canale
* **Lista Utenti**: /lu – Serve a mostrare gli utenti presenti in tutti i canali
* **Caccia Utente:** /kick - Serve a cacciare un utente dal canale
* **Ban Utente**: /ban <username> <canale> - Serve a bannare un utente dal canale
* **Sbanna Utente**: /unban - <username> <canale> - Serve a sbannare un utente dal canale
* **Ban definitivo Utente**: /fban <username> - Server a bannare un utente definitivamente dal server
* **Sbanna Utente dal Ban definitivo:** /funban <username> Serve a sbannare un utente dal server
* **Promuovi Utente**: /promote <username> Serve a promuovere un utente del canale con un ruolo di admin
* **Togli la promozione all’ Utente:** /unpromote <username> Serve a togliere la promozione ad un utente del canale

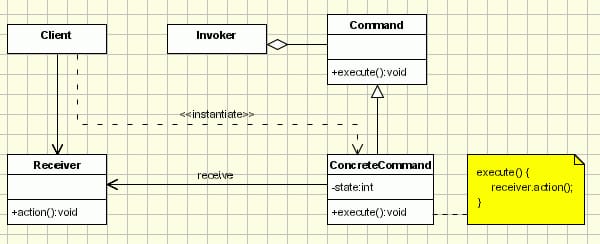
I comandi vengono impartiti tramite JSON. Quindi, ogni client può implementare come lui prevede come inviare il comando. Vale a dire che quello che si dovrà occupare il client è di creare un JSON adatto al server.

I comandi come /join e /list non sono comandi che devono essere necessariamente chiamati così, perché il server non interpreta /join e /list, ma interpreta il comando associato.

Quindi nel client non bisogna per forza fare /join, ma lo si può chiamare anche /cambio o /entrare. Basta che poi dentro al comando, nel JSON, per il cambio canale ci sia "command: parola riservata del server" che serve per fare il cambio canale.

In questo caso, lo si chiama "switch channel", però potremmo chiamarlo join in un'altra maniera. Tuttavia, il server lo interpreta solo quando si scrive "switch channel". Ora, tramite il client si può praticamente dire /cambio, per esempio, non per forza join. Basta che dentro al comando, nel JSON che si invia al server, ci sia "command: (switch channel)".

L'impiego della libreria **Gson** per la serializzazione e deserializzazione dei messaggi JSON rende il progetto più robusto e modulare, favorendo una gestione chiara dei comandi e delle risposte.



**Executor**

Inizialmente avevamo pensato di utilizzare un executor per inviare i messaggi via broadcast, per aumentare il throughput, praticamente in un executor, sarebbe una pull di thread. Il nome dell’executor si chiama broadcast executor, nato con l’intento di mandare messaggi in broadcast. Per aumentare il throughput si sono fatte modifiche, quindi vari gestori utilizzano il medesimo broadcast executor che bisognerebbe chiamarlo sandler executor, ma l’abbiamo chiamato ancora broadcast executor per affezionamento al nome.

**CommandHandler**

La classe CommandHandler implementa un sistema per gestire i comandi inviati dagli utenti in un'applicazione di comunicazione. Si tratta di un invoker all'interno di un pattern **Command**, che organizza i comandi in due categorie principali: **comandi comuni** e **comandi amministrativi**. Vediamo nel dettaglio come funziona:

### **Metodo** handleCommand

Questo metodo si occupa di gestire i comandi inviati dagli utenti:

* Controlla se il comando richiesto è un **comando comune** o un **comando amministrativo**.
* Per i comandi comuni:
  + Se il comando è presente nella mappa commonCommands, viene eseguito chiamando il metodo execute del rispettivo comando.
* Per i comandi amministrativi:
  + Prima di eseguire un comando amministrativo, il sistema verifica che l'utente abbia i privilegi necessari (ovvero che l'utente abbia il ruolo di "admin").
  + Se l'utente non è amministratore, viene negato l'accesso e viene stampato un messaggio di errore.
* Se il comando non è riconosciuto, viene stampato un messaggio che segnala il comando sconosciuto.

### **Pattern Command**

Questa classe segue il **pattern Command**, un pattern di progettazione che permette di incapsulare una richiesta come oggetto. I comandi vengono incapsulati in oggetti specifici (come LoginCommand, SendMessageBroadcastCommand, ecc.), rendendo il codice più modulare e facilmente estendibile. In questo modo, i comandi possono essere aggiunti o modificati senza alterare la logica del gestore principale.

### **Gestione della Sicurezza**

La classe gestisce i privilegi attraverso il controllo del ruolo dell'utente:

* Gli amministratori hanno accesso a comandi più avanzati, come kick, ban, promote, ecc.
* Se un utente tenta di eseguire un comando amministrativo senza i permessi necessari, l'accesso è negato.

Manuale utente con le istruzioni su compilazione ed esecuzione;

**Compilazione**

Per compilare il progetto, è necessario disporre di un ambiente Java configurato (Java JDK). Eseguire i seguenti comandi per compilare le classi:

1. Clona il repository:

git clone https://github.com/leopzoc/ProgettoReti\_IrcServer.git

cd ProgettoReti\_IrcServer

1. Compila il progetto:

javac -d out src/server/\*.java

**Esecuzione**

**Avvio del Server**: Lanciare il server utilizzando il comando:

1. Esegui il server:

java -cp out server.Server

1. Esegui un client per connetterti al server:

java -cp out client.Client