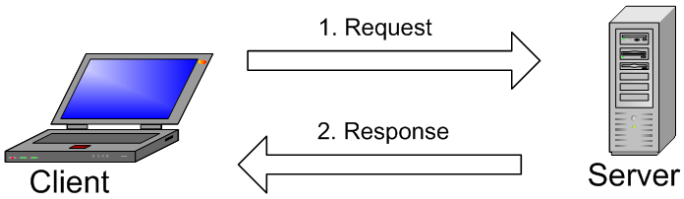
Programmazione avanzata

Lezione 4

Operazioni di base con i Socket per realizzare applicazioni con funzionalità di rete

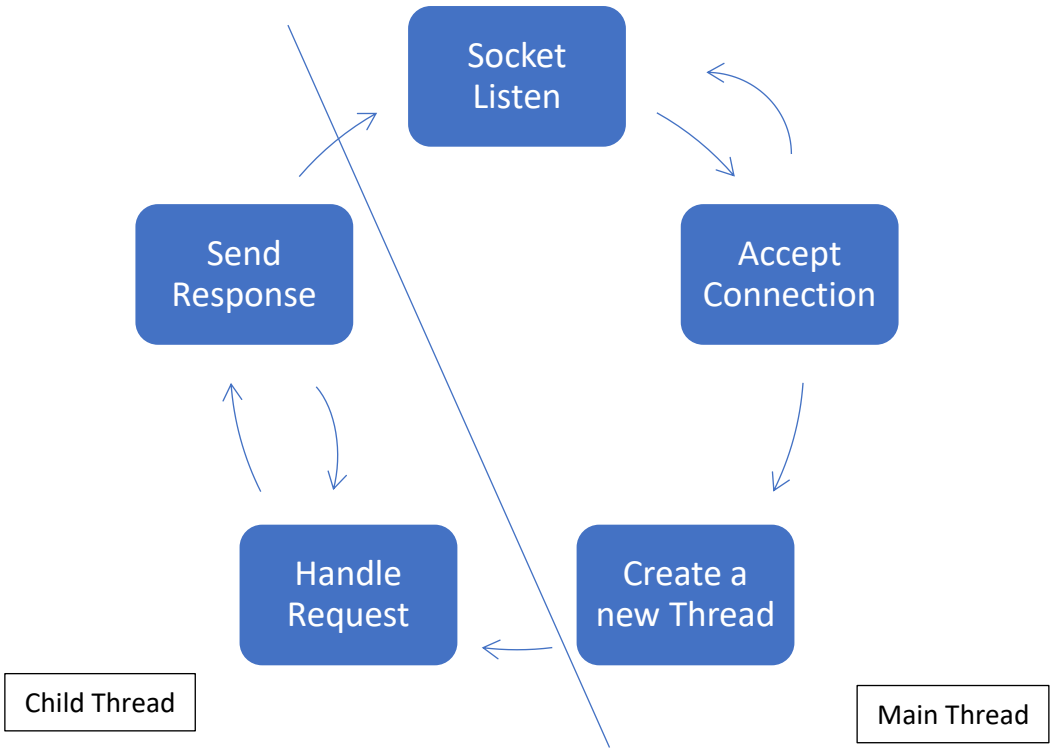
Realizzazione di applicazioni Client/Server



Le applicazioni di tipo Client/Server dividono la logica applicativa in due programmi, uno client e uno server, in esecuzione su due calcolatori diversi. La funzionalità fornita agli utenti viene implementata sfruttando le comunicazioni di rete, ad esempio attraverso una comunicazione di tipo richiesta risposta: il client invia al server una richiesta (ad esempio una richiesta di verifica password o una pagina HTML o semplicemente un dato) e questo risponde con il dato richiesto incapsulato in una risposta.

Implementazione server

L'implementazione di un server ha una struttura ben definita: al fine di poter servire più utenti contemporaneamente, il server utilizza la programmazione concorrente, ad esempio un nuovo Thread o un nuovo processo viene creato ogni volta che il server riceve una richiesta da un determinato client.



Il server è composto da un thread principale che viene eseguito al momento dell'avvio dell'applicazione, e da una serie di thread figli, ciascuno creato al momento dell'arrivo di una connessione. Il thread principale si mette in attesa di una connessione, una volta che una connessione è ricevuta questa viene accettata e viene passata ad un nuovo thread che viene creato appositamente per gestire lo scambio richiesta risposta con uno specifico client. Il thread figlio, continuamente riceve e gestisce le richieste fino a quando tutte le informazioni sono state scambiate e le richieste del client sono state soddisfatte. A quel punto la connessione può essere chiusa e il thread può esaurirsi.

```
1.  public class PrimoServer extends Thread {
2.      private ServerSocket serverSocket;
3.      private boolean running = false;
4.
5.      public void startServer()
6.      {
7.          try
8.          {
9.              serverSocket = new ServerSocket( 5555 );
10.             this.start();
11.         }
12.         catch (IOException e)
13.         {
14.             logger.error(e.getMessage());
15.         }
16.     }
17.
18.     public void stopServer()
19.     {
20.         running = false;
21.         this.interrupt();
22.     }
23.
24.     @Override
25.     public void run()
26.     {
27.         running = true;
28.         while( running )
29.         {
30.             try
31.             {
32.
33.                 Socket socket = serverSocket.accept();
34.                 RequestHandler requestHandler = new RequestHandler( socket );
35.                 requestHandler.start();
36.             }
37.             catch (IOException e)
38.             {
39.                 logger.debug(e.getMessage());
40.             }
41.         }
42.     }
43.
44.
45.     public static void main(String[] args) {
46.
47.         PrimoServer server = new PrimoServer();
48.         server.startServer();
49.
50.     }
51. }
52.
```

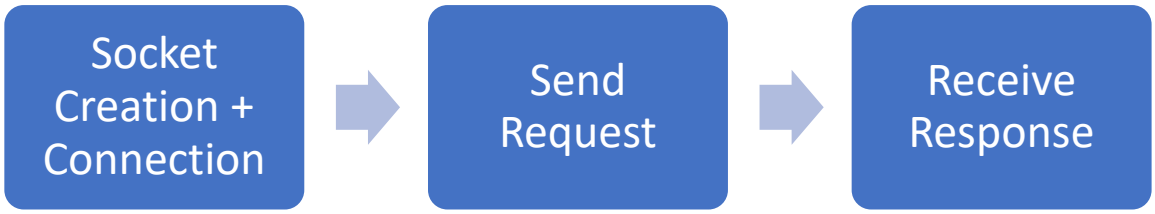
(2) la classe SocketServer implementa le funzionalita' base di un server
(3) la variabile booleana running implementa la variabile di stato che indica se il server e' al momento in esecuzione
(5) la funzione startServer fa partire le funzionalita' di server, creando un nuovo elemento ServerSocket e mettendolo in ascolto sulla porta 5555 (riga 9) e facendo partire il thread (riga 10).
(18) la funzione stopServer interrompe le funzionalità del server, interrompendo anche l'esecuzione del thread (riga 21)
(25) la funzione run contiene il codice del thread, con un ciclo infinito (riga 28) che ciclicamente accetta connessioni (riga 33) e crea una nuova istanza della classe RequestHandler (riga 34) che implementa le funzioni per la gestione delle richieste da parte di un client. Successivamente (riga 35) il thread della classe RequestHandler viene fatta partire.
(45) Nel main viene creata una nuova istanza della classe PrimoServer e viene inizializzato (riga 48)

La classe RequestHandler coneiene il codice che gestisce le richieste dei client. Il tutto è implementato nella run che viene eseguito come corpo del thread (righe 11 – 32). I dati ricevuti vengono gestiti tramite la BufferedRead (in input) e la PrintWriter (in output), per leggere la richiesta (riga 19) e generare la risposta (riga 24).

```
1.  class RequestHandler extends Thread
2.  {
3.      private Socket socket;
4.
5.      RequestHandler( Socket socket )
6.      {
7.          this.socket = socket;
8.      }
9.
10.     @Override
11.     public void run()
12.     {
13.         try
14.         {
15.
16.             BufferedReader in = new BufferedReader( new InputStreamReader(
    socket.getInputStream() ) );
17.             PrintWriter out = new PrintWriter( socket.getOutputStream() );
18.
19.             String line = in.readLine();
20.
21.             // Eseguire operazioni sulla richiesta
22.
23.             // Generare una risposta
24.             out.println( "RISPOSTA" );
25.             out.flush();
26.
27.             in.close();
28.             out.close();
29.             socket.close();
30.
31.         }
32.         catch( Exception e )
33.         {
34.             logger.debug(e.getMessage());
35.         }
36.     }
37. }
38.
```

Implementazione client

La realizzazione di un programma client ha un flusso di lavoro più semplice: si crea un socket e si connette al server, si inviano richieste e si attendono risposte.




```
1.  try
2.  {
3.      Socket socket = new Socket( "127.0.0.1", 5555 );
4.
5.      PrintStream out = new PrintStream( socket.getOutputStream() );
6.      BufferedReader in = new BufferedReader( new InputStreamReader(socket.getInputStream()));
7.
8.      out.println( “RICHIESTA” );
9.
10.     String line = in.readLine();
11.
12.     // Gestisci risposta
13.
14.     in.close();
15.     out.close();
16.     socket.close();
17. }
18. catch( Exception e )
19. {
20.     e.printStackTrace();
21. }
22.
```

(3) la creazione e la connessione del socket al server avviene in maniera contestuale direttamente all’interno del costruttore. Un eventuale server irraggiungibile genera un’eccezione che deve essere gestito nel catch (righe 18-21).

Esercizio

In questo esercizio proviamo a modificare l’applicazione dell’esercitazione precedente per implementare la funzionalità di verifica delle credenziali sul server, trasferendo nome utente e password come testo attraverso un socket. Abbiamo quindi bisogno di modificare l’applicazione in modo da inviare i dati verso un server e di realizzare un semplice server che riceve i dati di accesso, li analizza e risponde con login valido o meno in modo che l’applicazione possa passare alla schermata successiva o meno.

Operazioni bloccanti e interfaccia grafica

Le operazioni legate ai socket come altre operazioni che potenzialmente possono impiegare molto tempo sono operazioni bloccanti che non dovrebbero essere eseguite direttamente nel corpo della funzione corrispondente all'evento dell'interfaccia grafica in quanto fino alla terminazione della funzione handle l'interfaccia grafica rimane non responsiva. Questo può essere verificato inserendo una chiamata alla funzione "Thread.sleep(5000);" per ritardare la terminazione della funzione e vedere il blocco dell'interfaccia.

Al fine di evitare questo problema le operazioni dovrebbero essere inserite in un thread indipendente in modo da sbloccare l'handler immediatamente. Questo può essere utilizzato attraverso l'uso dei Task, costrutto inserito in JavaFX per l'esecuzione di operazioni asincrone[1]..

```
1.  Task task = new Task<Void>() {
2.  @Override public Void call() {
3.      try
4.      {
5.            Socket socket = new Socket( "127.0.0.1", 5555 );
6.
7.            PrintStream out = new PrintStream(socket.getOutputStream());
8.            BufferedReader in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
9.
10.           // Inviare richiesta
11.
12.           String line = in.readLine();
13.
14.           // Leggere risposta
15.
16.           // Modificare interfaccia grafica (esempio riabilitare campi testo)
17.           Platform.runLater(new Runnable() {
18.           @Override public void run() {
19.                 campoUtente.setDisable(false);
20.                 primaryButton.setDisable(false);
21.                 primaryButton.setText("Riprova");
22.           }
23.           });
24.
25.           in.close();
26.           out.close();
27.           socket.close();
28.     }
29.     catch( Exception e )
30.     {
31.           logger.error(e.getMessage());
32.     }
33.     return null;
34. }
35. };
36. new Thread(task).start();
```

La gestione della comunicazione con il server può essere inclusa all'interno di un Task asincrono dichiarato inline (riga 1-2) il quale viene poi lanciato subito dopo (riga 36). Il codice all'interno del Task esegue le operazioni di comunicazione con il server. Un Task non dovrebbe interagire direttamente con l'interfaccia grafica per evitare un accoppiamento tra l'interfaccia e un task che esegue una determinata

[1] https://docs.oracle.com/javase/9/docs/api/index.html?javafx/concurrent/Task.html

funzione. Eventuali modifiche all'interfaccia grafica possono essere effettuate creando e lanciando un'istanza Runnable definita inline (riga 17-18). Anche in questo caso viene definita una funzione run che va a modificare l'interfaccia grafica e viene mandata in esecuzione al primo momento utile direttamente dal sistema tramite la chiamata runLater (riga 17).

All'interno della funzione run si possono vedere alcuni esempi per riabilitare un campo testo (riga 19), per disabilitare un bottone (riga 20) o per cambiare il messaggio sul bottone (riga 21).

Esercizio

Modificare il programma realizzato per includere le operazioni bloccanti all'interno di Task e di un elemento Runnable, andando anche a gestire i bottoni e gli elemeneti di testo durante le operazioni di comunicazione con il server.