2. SOCKET TCP

Nel paradigma di programmazione orientata ad oggetti, la computazione viene effettuata attraverso un insieme di oggetti che contengono uno stato (variabili istanza) ed espongono un comportamento, vale a dire permettono ad altri oggetti di usare i loro metodi. Per poter estendere questo modello di programmazione nell'ambito distribuito, è necessario permettere di invocare un metodo da parte di un oggetto remoto, questo lo si fa mediante i cosiddetti **Socket**.

La comunicazione tra programmi su internet avviene utilizzando la suite di protocolli TCP/IP. La maniera in cui questi protocolli vengono usati è quello di fornire un'astrazione (mediante il software di rete) chiamata socket che permette di ricevere e trasmettere dati.

I **socket TCP** sono degli endpoint di una comunicazione bidirezionale sulla rete che unisce due programmi, ad ogni socket viene assegnato un numero di porta che serve a identificare l'applicazione che è incaricata di dover trattare i dati, che è in esecuzione sul computer che li riceve.

Quindi un socket viene univocamente definito dalla combinazione di **indirizzo IP** e **num di porta**.

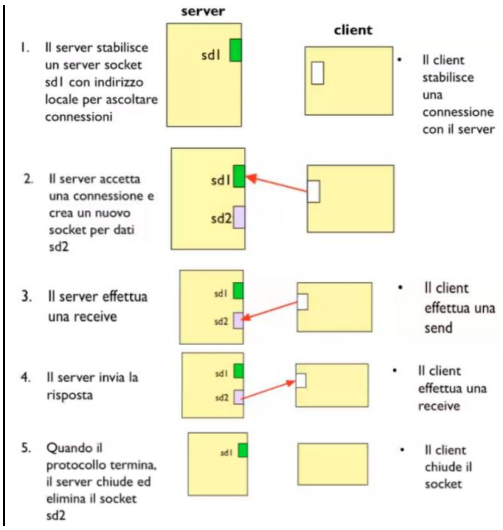
Normalmente, si identificano i due computer coinvolti in un socket, col nome di **client** e **server**.

Il **server** è in esecuzione ed attende che qualche client richieda la connessione. Dal lato client, il programma conosce l'indirizzo della macchina su cui è in esecuzione il server ed il numero di porta, in più il client deve anche comunicare al server il numero di porta locale sul quale riceverà i dati (di solito questo viene assegnato dal sistema).

Il procedimento di connessione prevede che il server debba accettare la connessione e che assegni un **nuovo socket** per la comunicazione bidirezionale tra client e server (*socket di comunicazione*). In questa maniera, il server può tornare ad accettare connessioni da altri client. In questo caso, tipicamente il server lancia un thread per ogni socket stabilito con un client, in modo da permettere la gestione concorrente delle comunicazioni con tutti i client.

Il package **java.net** offre due classi per i socket, proprio per gestire la fase di accettazione da parte dei server e la comunicazione bidirezionale tra client e server:

La **classe Socket** (crea il canale di comunicazione) e la **classe ServerSocket** (accetta connessioni, richiamata dal server), quest'ultima implementa un socket di connessione che attende richieste da parte del client, quando ne riceve una, assegna un socket alla connessione bidirezionale, restituendo l'oggetto Socket che viene usato per la comunicazione tra client e server.



2.1 STREAM

La comunicazione tra client e server avviene attraverso la scrittura e la lettura di **stream** (flussi) associati con il socket e che permettono una facile interazione (gestita dal linguaggio) per poter trasmettere istanze di classi Java (oggetti) tra client e server, attraverso un meccanismo di ***serializzazione***.

Gli stream di I/O sono una utile astrazione che Java fornisce al programmatore per trattare con una sequenza di dati che può essere "diretta a" / "proveniente da" diverse entità, quali file, periferiche, memoria e ovviamente socket.

Gli stream sono presenti sotto numerose forme: la gerarchia delle classi di stream nel package java.io offre un notevole numero.

La sottoclasse più importante che utilizzeremo e **ObjectInputStream** che fornisce il meccanismo di ***deserializzazione*** quando si riceve un oggetto precedentemente serializzato con **ObjectOutputStream**. Gli oggetti che possono essere trasmessi su questo tipo di stream devono implementare l'interfaccia **Serializable**.

La sequenza di istruzioni classicamente utilizzata per accedere agli stream di un socket lato server è:

```
ServerSocket serverSocket = new ServerSocket(9000);
socket = serverSocket.accept();
System.out.println("Accettata una connessione… attendo comandi");
ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
```

Metodi di Socket:

- `ServerSocket(int port)`: crea un server socket su una specifica porta;
- `Socket accept() throws IOException`: aspetta connessioni sul ServerSocket e lo accetta (metodo bloccante fino a nuova connessione);
- `public void close() throws IOException`: chiude il socket;
- `void setSoTimeout(int timeout) throws SocketException`: setta un timeout per call ad accept, se il tempo passa viene lanciata l'eccezione.

Metodi di InputStream, dati che da una sorgente esterna possono essere usati dal programma:

`int`	**`available()`** Restituisce una stima del numero di byte che possono essere letti (o ignorati) da questo flusso di input senza essere bloccati dalla successiva chiamata di un metodo per questo flusso di input.
`void`	**`close()`** Chiude questo flusso di input e rilascia tutte le risorse di sistema associate al flusso.
`void`	**`mark(int readlimit)`** Contrassegna la posizione corrente in questo flusso di input.
`boolean`	**`markSupported()`** Verifica se questo flusso di input supporta i metodi `marke` `reset`.
`abstract int`	**`read()`** Legge il byte di dati successivo dal flusso di input.
`int`	**`read(byte[] b)`** Legge un certo numero di byte dal flusso di input e li memorizza nella matrice di buffer `b`.
`int`	**`read(byte[] b, int off, int len)`** Legge fino a `len`byte di dati dal flusso di input in una matrice di byte.
`void`	**`reset()`** Riposiziona questo flusso nella posizione in cui il `mark`metodo è stato chiamato l'ultima volta su questo flusso di input.
`long`	**`skip(long n)`** Salta e scarta `n`byte di dati da questo flusso di input.

Metodi di OutputStream, dati che il programma può inviare:

`void`	`close()` Chiude questo flusso di output e rilascia tutte le risorse di sistema associate a questo flusso.
`void`	`flush()` Svuota questo flusso di output e forza la scrittura dei byte di output memorizzati nel buffer.
`void`	`write(byte[] b)` Scrive i `b.length`byte dalla matrice di byte specificata in questo flusso di output.
`void`	`write(byte[] b, int off, int len)` Scrive i `len`byte dalla matrice di byte specificata a partire dall'offset `off`in questo flusso di output.
`abstract void`	`write(int b)` Scrive il byte specificato in questo flusso di output.

Esempio con i Socket:

Server.java

```
import java.io.*;
import java.net.*;
import java.util.logging.Logger;

public class Server {
  static Logger logger = Logger.getLogger("global");

  public static void main(String[] args) {
    try {
      ServerSocket serverSocket = new ServerSocket(9000);
      logger.info("Socket istanziato, accetto connessioni...");
      Socket socket = serverSocket.accept();
      logger.info("Accettata una connessione... attendo comandi.");
      ObjectOutputStream outStream = new ObjectOutputStream (socket.getOutputStream());
      ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
      String nome= (String) inStream.readObject();
      logger.info("Ricevuto:" + nome);
      outStream.writeObject("Hello "+ nome);
      socket.close();
    } catch (EOFException e)  {
      logger.severe("Problemi con la connessione:" + e.getMessage());
      e.printStackTrace();
    } catch (Throwable t)  {
      logger.severe("Lanciata Throwable:" + t.getMessage());
      t.printStackTrace();
    }
  }
}
```

Il client si connette al server su *localhost* passando *Giovanni*.
Il server risponde salutando *Giovanni* con *Hello Giovanni*.
Nel try instanziamo su socket di connessione sulla porta 9000 e chiamiamo il metodo bloccante accept(). Da questo metodo si esce solamente quando un client effettua una richiesta di connessione verso il server, ed il metodo restituisce un socket che è stato stabilito tra il server ed il client.
È su questo socket che usiamo gli stream in input e output che servono, prima, per chiamare il metodo (anche esso bloccante) readObject() che restituisce l'oggetto trasmesso (e deserializzato dallo stream) che viene utilizzato per realizzare la risposta.
Ed infine si chiude il socket con close().

Client.java

```
import java.io.*;
import java.net.*;
import java.util.logging.Logger;

public class Client {
  static Logger logger = Logger.getLogger("global");

  public static void main(String args[]) {
    try {
      Socket socket = new Socket ("localhost", 9000);
      ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
      ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
      out.writeObject("Giovanni");
      System.out.println(in.readObject());
      socket.close();
    } catch (EOFException e)  {
      logger.severe("Problemi con la connessione:" + e.getMessage());
      e.printStackTrace();
    } catch (Throwable t)  {
      logger.severe("Lanciata Throwable:" + t.getMessage());
      t.printStackTrace();
    }
    }
  }
}
```

Il client apre il socket verso l'host locale dove abbiamo lanciato il server (prima riga del try), preleva gli stream dal socket e scrive il suo nome sullo stream di output (oggetti out e in).
Il client termina stampando a video quello che ha ricevuto dalla lettura dello stream di input.

NOTA: Particolare attenzione va data all'ordine con il quale si devono aprire gli stream: prima lo stream di output e poi quello di input.

Per concludere, il server può essere facilmente reso iterativo, inserendo la accept() e la risposta verso il client all'interno di un while(true). Poi, si può rendere il server multi-thread, in modo che ad ogni accept, si faccia partire un thread (che gestisce l'invio della risposta ai client) permettendo ai server di tornare subito alla accept() successiva.

Esempio Client-Server coi Socket:

RecordRegistro.java

```
import java.io.Serializable;

public class RecordRegistro implements Serializable {
  private static final long serialVersionUID = -4147133786465982122L;

  // Costruttore
  public RecordRegistro(String n, String i) {
    nome = n;
    indirizzo = i;
  }

  // Metodi accessori
  public String getNome() {
    return nome;
  }
  public String getIndirizzo() {
    return indirizzo;
  }

  // Variabili istanza
  private String nome;
  private String indirizzo;
}
```

L'esempio implementa sul server un registro che contiene record (composti da due campi stringa, nome e indirizzo) che vengono inseriti e possono essere reperiti specificando solamente il nome.
Realizzeremo un server che, in attesa su una porta, riceve le richieste di inserimento (codificate attraverso un oggetto RecordRegistro con tutti i campi riempiti) o le richieste di ricerca (codificate attraverso un oggetto RecordRegistro che ha solo il campo nome riempito).

Iniziamo dalla definizione dei record da memorizzare, un semplice wrapper di due campi stringa, con i relativi metodi di accesso.
Le instanze sono serializzabili (implementa l'interfaccia Serializable), in quanto ci aspettiamo che debbano essere trasmesse su socket TCP in formato binario. In pratica, l'algoritmo di deserializzazione usa questo numero per assicurarsi che la classe appena caricata corrisponde ad un oggetto serializzato. Se questo non corrisponde ai numeri che ha calcolato, allora viene lanciata una eccezione InvalidClassException.

RegistroServer.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.logging.Logger;

public class RegistroServer {
  static Logger logger = Logger.getLogger("global");

  public static void main(String[] args) {
    HashMap<String, RecordRegistro> hash = new HashMap<String, RecordRegistro>();
    Socket socket = null;
    System.out.println ("In attesa di connessioni...");
    try {
      // Creazione ed accept su socket
      ServerSocket serverSocket = new ServerSocket(7000);
      while (true) {
        socket = serverSocket.accept();
        ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
        RecordRegistro record = (RecordRegistro) inStream.readObject();
        if (record.getIndirizzo()!=null) { // si tratta di una scrittura
          logger.info ("Inserisco:" + record.getNome() +", "+record.getIndirizzo());
          hash.put(record.getNome(), record);
        } else { // è una ricerca
          logger.info ("Cerco:" + record.getNome());
          RecordRegistro res = hash.get(record.getNome());
          ObjectOutputStream outStream=new ObjectOutputStream(socket.getOutputStream());
          outStream.writeObject(res); // se non c'è il record, res è null
          outStream.flush();
        }
        socket.close();
      } // fine while
    } catch (EOFException e)  {
      logger.severe("Problemi con la connessione:" + e.getMessage());
      e.printStackTrace();
    } catch (Throwable t)  {
      logger.severe("Lanciata Throwable:" + t.getMessage());
      t.printStackTrace();
    }
    finally { // chiusura del socket e terminazione programma
      try { socket.close();
      } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
      }
    }
  }
}
```

Questo server non fa altro che attendere sulla porta 7000 che ci siano delle richieste di connessione. Il server è iterativo e serve le richieste così come arrivano, quindi senza multi-thread.

Tutti i record che arrivano vanno memorizzati in una HashMap che, come chiave di accesso, utilizza il nome presente nel record.

Nel ciclo infinito il programma accetta connessioni, riceve un oggetto dallo stream in input.

Se l'oggetto ha il campo indirizzo non vuoto, allora è una richiesta di inserimento, che viene effettuata nell'if, altrimenti si tratta di una ricerca, che viene effettuata e restituita sullo stesso socket, ai client.

Se la ricerca è stata infruttuosa, il metodo get() restituisce null che viene restituito ai client.

ShellClient.java

```
import java.io.*;
import java.net.*;
import java.util.logging.Logger;
public class ShellClient {
  static Logger logger = Logger.getLogger("global");

  public static void main(String args[]) {
    String host = args[0];
    String cmd;
    in = new BufferedReader(new InputStreamReader(System.in));
    try {
      while (!(cmd = ask ("Comandi>")).equals("quit")) {
        if (cmd.equals ("inserisci")) {
          System.out.println ("Inserire i dati.");
          String nome = ask("Nome: ");
          String indirizzo = ask ("Indirizzo: ");
          RecordRegistro r = new RecordRegistro(nome, indirizzo);
          Socket socket = new Socket (host, 7000);
          ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
          sock_out.writeObject(r);
          sock_out.flush();
          socket.close();
        } else if (cmd.equals ("cerca")) {
          System.out.println ("Inserire il nome per la ricerca.");
          String nome = ask("Nome: ");
          RecordRegistro r = new RecordRegistro(nome, null);
          // si invia un oggetto con indirizzo vuoto
          Socket socket = new Socket (host, 7000);
          ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
          sock_out.writeObject(r);
          sock_out.flush();
          ObjectInputStream sock_in = new ObjectInputStream(socket.getInputStream());
          RecordRegistro result = (RecordRegistro) sock_in.readObject();
          // se viene ottenuto un risultato, allora si stampa l'indirizzo
          if (result != null)
            System.out.println ("Indirizzo :"+result.getIndirizzo());
          else // altrimenti non esiste un record con quel nome
            System.out.println ("Record non presente");
          socket.close();
        } else System.out.println (ERRORMSG);
      } // end while
    } catch (Throwable t) {
      logger.severe("Lanciata Throwable:" + t.getMessage());
      t.printStackTrace();
      }
    System.out.println ("Bye bye");
  }
  private static String ask(String prompt) throws IOException {
    System.out.print(prompt+" ");
    return (in.readLine());
  }

  static final String ERRORMSG = "Cosa?";
  static BufferedReader in = null;
}
```

Per ogni richiesta dell'utente, apre il socket, fa la richiesta al server e scrive la risposta (se necessario), chiudendo il socket. Questo avviene sia per l'inserimento che per la ricerca.

Per l'inserimento, dopo aver chiesto all'utente di digitare le stringhe per comporre l'oggetto RecordRegistro da inviare, apre il socket, usa Io stream di output, invia l'oggetto e chiude il socket.
Per La ricerca, dopo aver chiesto all'utente il nome dei record da ricercare, il client invia un record con il campo indirizzo a null e si attende un oggetto in risposta. Se l'oggetto non è null, allora la ricerca e andata a buon fine e si stampa l'indirizzo dell'oggetto ricevuto, altrimenti si stampa un messaggio, e si chiude ii socket.

Il metodo ask() serve per fare input da tastiera in maniera semplice, scrivendo un prompt definito nel programma.