Programmazione distribuita in Java: Socket

da materiale di

Carlo Ghezzi e Alfredo Motta

Nodi fisici e nodi logici

- Occorre distinguere tra nodi fisici e logici
- Può essere opportuno progettare ignorando all'inizio il nodo fisico in cui un nodo logico sarà allocato
- Java consente addirittura di vedere tutto attraverso la nozione di oggetti e di invocazione di metodi, dove l'invocazione può essere remota

Architettura client-server

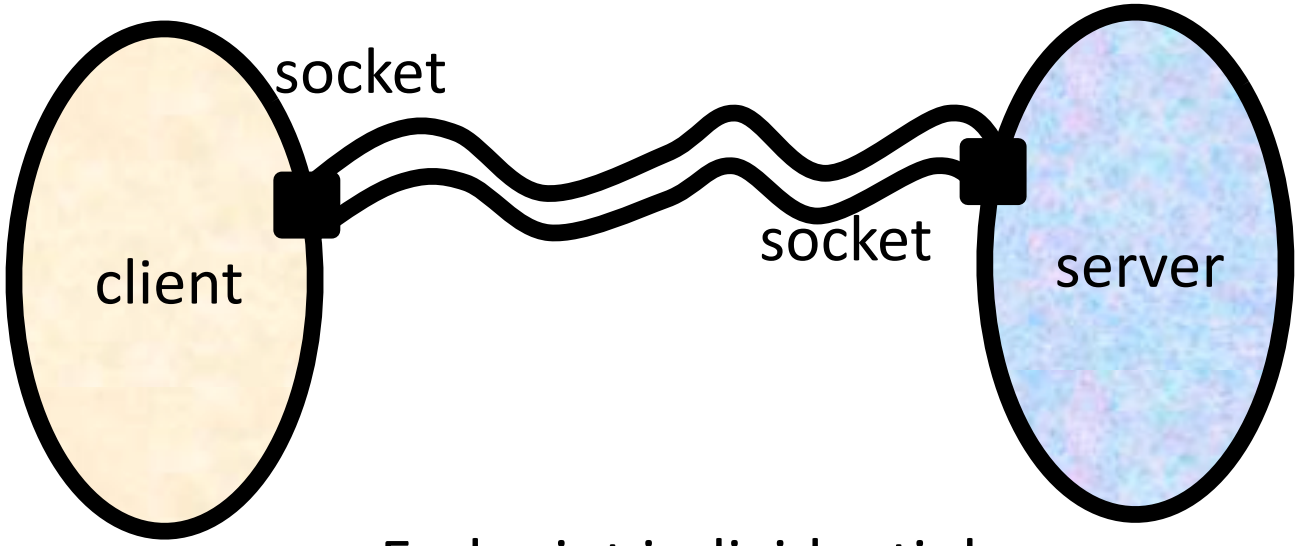
- È il modo classico di progettare applicazioni distribuite su rete
- **Server**
  - Offre un servizio “centralizzato”
  - Attende che altri (client) lo contattino per fornire il proprio servizio
- **Client**
  - Si rivolge ad apposito/i server per ottenere certi servizi

Middleware

- Per programmare un sistema distribuito vengono forniti servizi (di sistema) specifici, come estensione del sistema operativo
- Il **middleware** viene posto tra il sistema operativo e le applicazioni
- In Java il middleware fa parte del linguaggio, diversamente da altre soluzioni
  - Esempio: CORBA

Socket in Java

- Client e server comunicano attraverso **socket** che permettono lo scambio di pacchetti TCP
  - Package java.net, classi: Socket, ServerSocket
  - DatagramSocket (UDP): non considerati in questo corso



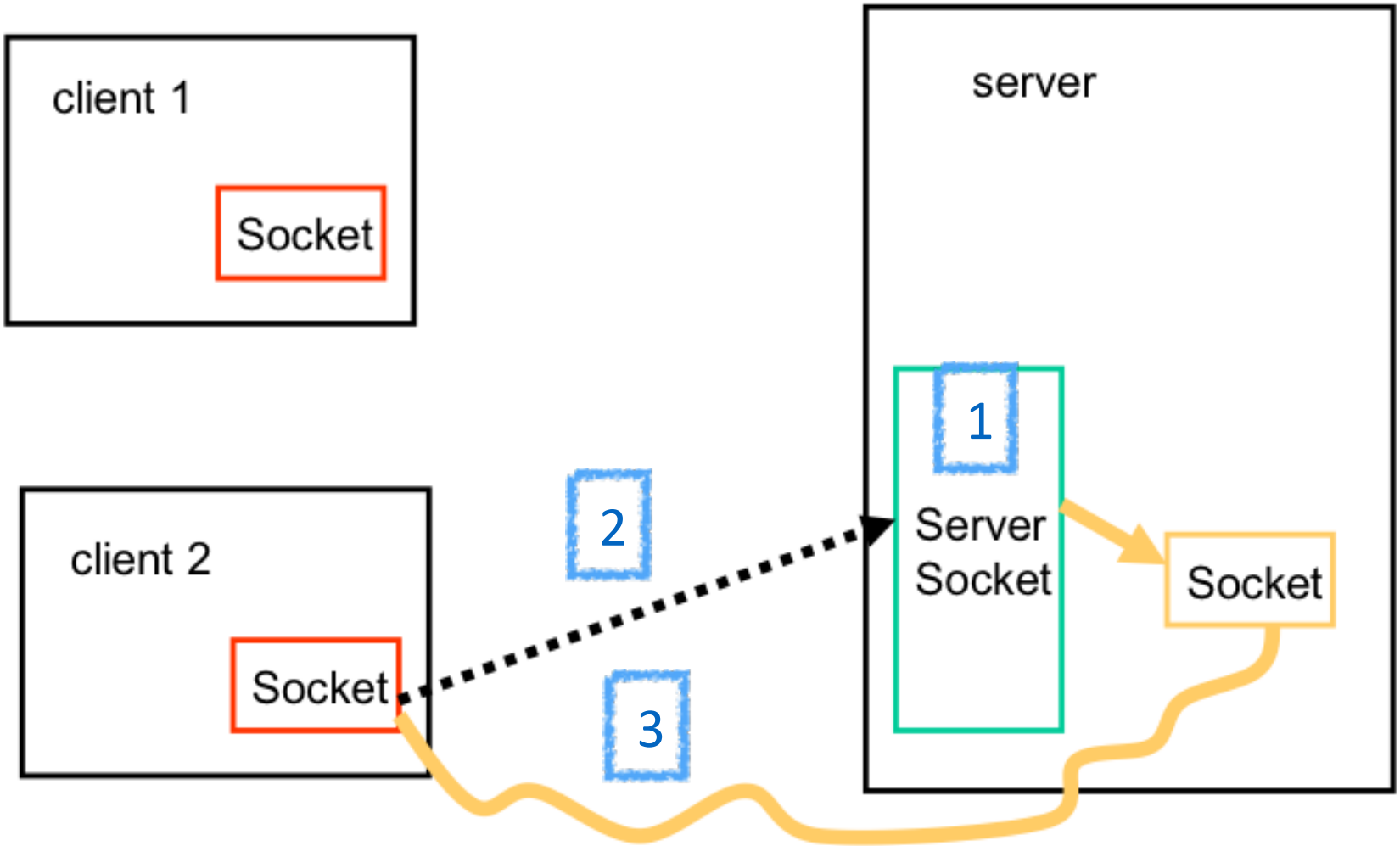
Endpoint individuati da:

- **indirizzo IP**
- **numero di porta**

Socket (dal tutorial Java)

- A socket is one endpoint of a two-way communication link between two programs running on the network
- A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to

Comunicazione client-server



Attesa connessione (lato server)

- Creare un'istanza della classe java.net.ServerSocket specificando il numero di porta su cui rimanere in ascolto
  - La porta non deve essere già in uso
  - `ServerSocket sc = new ServerSocket(4567);`
- Chiamare il metodo accept() che fa in modo che il server rimanga in ascolto di una richiesta di connessione `Socket s = sc.accept();`
- Quando il metodo completa la sua esecuzione la connessione col client è stabilita e viene restituita un'istanza di java.net.Socket connessa al client remoto

Aprire connessione (lato client)

- Aprire un socket specificando l'indirizzo IP e numero di porta del server
  - `Socket sc = new Socket("127.0.0.1", 4567);`
  - Il numero di porta è compreso fra 1 e 65535
    - Le porte inferiori a 1024 sono riservate a servizi standard
- All'indirizzo e numero di porta specificati ci deve essere in ascolto un processo server
  - `Socket ss = serverSocket.accept();`
- Se la connessione ha successo si usano sia dal lato client che dal lato server gli **stream** associati al socket per permettere la comunicazione tra client e server (e viceversa)
  - `Scanner in = new Scanner(sc.getInputStream());`
  - `PrintWriter out = new PrintWriter(sc.getOutputStream());`

Chiusura connessioni

- Per chiudere un ServerSocket o un Socket si utilizza il metodo close()
- Per ServerSocket, close() fa terminare la accept() con IOException
- Per Socket, close() fa terminare le operazioni di lettura o scrittura del socket con eccezioni che dipendono dal tipo di reader/writer utilizzato
- Sia ServerSocket sia Socket hanno un metodo isClosed() che restituisce vero se il socket è stato chiuso

EchoServer

- Si crei un server che accetta connessioni TCP sulla porta 1337
- Una volta accettata la connessione il server leggerà ciò che viene scritto una riga alla volta e ripeterà nella stessa connessione ciò che è stato scritto
- Se il server riceve una riga “quit” chiuderà la connessione e terminerà l’esecuzione

EchoServer

```
public class EchoServer {
    private int port;
    private ServerSocket serverSocket;
    public EchoServer(int port) {
        this.port = port;
    }

    public static void main(String[] args) {
        EchoServer server = new EchoServer(1337);
        try {
            server.startServer();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
```

```
public void startServer() throws IOException {
    // apro una porta TCP
    serverSocket = new ServerSocket(port);
    System.out.println("Server socket ready on port: " + port);
    // resto in attesa di una connessione
    Socket socket = serverSocket.accept();
    System.out.println("Received client connection");
    // apro gli stream di input e output per leggere  e scrivere
    // nella connessione appena ricevuta
    Scanner in = new Scanner(socket.getInputStream());
    PrintWriter out = new PrintWriter(socket.getOutputStream());
    // leggo e scrivo nella connessione finche' non ricevo "quit"
    while (true) {
       String line = in.nextLine();
       if (line.equals("quit")) {
          break;
       } else {
          out.println("Received: " = line);
          out.flush();
         }
    }
    // chiudo gli stream e il socket
    System.out.println("Closing sockets");
    in.close();
    out.close();
    socket.close();
    serverSocket.close();
}
```

E' fondamentale chiamare **flush()** per assicurarsi che lo stream venga svuotato, quindi il contenuto spedito alla destinazione!

LineClient

- Si crei un client che si collega, con protocollo TCP, alla porta 1337 dell'indirizzo IP 127.0.0.1
- Una volta stabilita la connessione il client legge una riga alla volta dallo standard input e invia il testo digitato al server
- Il client inoltre stampa sullo standard output le risposte ottenute dal server
- Il client deve terminare quando il server chiude la connessione

LineClient

```
public class LineClient {
    private String ip;
    private int port;

    public LineClient(String ip, int port) {
        this.ip = ip;
        this.port = port;
    }

    public static void main(String[] args) {
        LineClient client = new LineClient("127.0.0.1", 1337);
        try {
            client.startClient();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
```

```
public void startClient() throws IOException {
    Socket socket = new Socket(ip, port);
    System.out.println("Connection established");
    Scanner socketIn = new Scanner(socket.getInputStream());
    PrintWriter socketOut = new PrintWriter(socket.getOutputStream());
    Scanner stdin = new Scanner(System.in);
    try {
        while (true) {
            String inputLine = stdin.nextLine();
            socketOut.println(inputLine);
            socketOut.flush();
            String socketLine = socketIn.nextLine();
            System.out.println(socketLine);
        }
    }
    catch(NoSuchElementException e) {
        System.out.println("Connection closed");
    }
    finally {
        stdin.close();
        socketIn.close();
        socketOut.close();
        socket.close();
   }
}
```

Architettura del server

- Il server che abbiamo visto accetta una sola connessione alla volta da un solo client
- Un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”
- Idea: **server multi-thread**
  - All’interno del processo Server far eseguire le istruzioni dopo l’accept() in un nuovo thread
  - In questo modo è possibile accettare più client contemporaneamente

EchoServer multi-thread

- Spostiamo la logica che gestisce la comunicazione con il client in una nuova classe ClientHandler che implementa Runnable
- La classe principale del server si occupa solo di istanziare il ServerSocket, eseguire la accept() e di creare i thread necessari per gestire le connessioni accettate
- La classe ClientHandler si occupa di gestire la comunicazione con il client associato al socket assegnato

EchoServer multi-thread

Crea thread quando necessario, ma ri-usa quelli esistenti finchè possibile

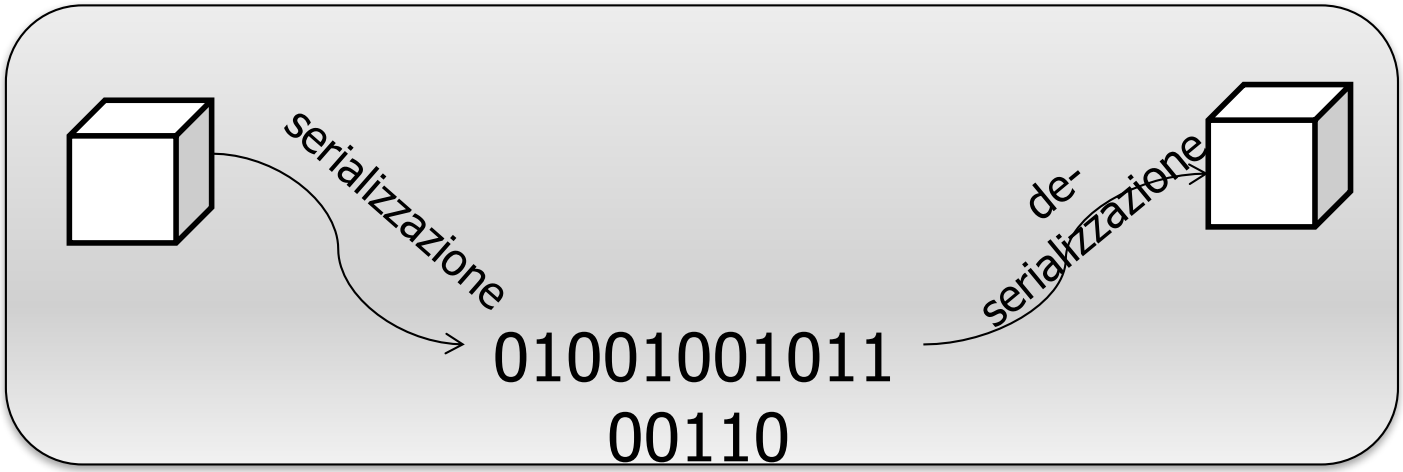
```
public class MultiEchoServer {
        private int port;
        public MultiEchoServer(int port) {
                this.port = port;
        }
        public void startServer() {
                ExecutorService executor = Executors.newCachedThreadPool();
                ServerSocket serverSocket;
                try {
                        serverSocket = new ServerSocket(port);
                } catch (IOException e) {
                        System.err.println(e.getMessage()); // porta non disponibile
                        return;
                }
                System.out.println("Server ready");
                while (true) {
                        try {
                                Socket socket = serverSocket.accept();
                                executor.submit(new EchoServerClientHandler(socket));
                        } catch(IOException e) {
                                break; // entrerei qui se serverSocket venisse chiuso
                        }
                }
                executor.shutdown();
        }
        public static void main(String[] args) {
                MultiEchoServer echoServer = new MultiEchoServer(1337);
                echoServer.startServer();
        }
}
```

EchoServer multi-thread

```
public class EchoServerClientHandler implements Runnable {
	private Socket socket;
	public EchoServerClientHandler(Socket socket) {
		this.socket = socket;
	}
	public void run() {
		try {
			Scanner in = new Scanner(socket.getInputStream());
			PrintWriter out = new PrintWriter(socket.getOutputStream());
			// leggo e scrivo nella connessione finche' non ricevo "quit"
			while (true) {
				String line = in.nextLine();
				if (line.equals("quit")) {
					break;
				} else {
					out.println("Received: " + line);
					out.flush();
				}
			}
			// chiudo gli stream e il socket
			in.close();
			out.close();
			socket.close();
		} catch (IOException e) {
			System.err.println(e.getMessage());
		}
	}
}
```

Le basi della serializzazione

- La **serializzazione** è un processo che trasforma un oggetto in memoria in uno stream di byte
- La **de-serializzazione** è il processo inverso
  - Ricostruisce un oggetto Java da uno stream di byte e lo riporta nello stesso stato nel quale si trovava quando è stato serializzato



Le basi della serializzazione

- Solo le istanze delle classi possono essere serializzate
  - I tipi primitivi non possono essere serializzati
- Affinché sia possibile serializzare un oggetto, la sua classe o una delle sue superclassi deve implementare l'interfaccia **Serializable**
- L'interfaccia Serializable è un interfaccia vuota utilizzata solo come metodo per marcare un oggetto che può essere serializzato
- Per serializzare/de-serializzare un oggetto basta scriverlo dentro un ObjectOutputStream/ObjectInputStream
  - Per scrivere/leggere i tipi primitivi utilizzare i metodi della DataOutput/DataInput interface implementati da ObjectOutputStream/ObjectInputStream

Le basi della serializzazione

Serializzazione

```
FileOutputStream out = new FileOutputStream( "save.ser" );
ObjectOutputStream oos = new ObjectOutputStream( out );
oos.writeObject( new Date() );
oos.close();
```

De-Serializzazione

```
FileInputStream in =  new FileInputStream( "save.ser" );
ObjectInputStream ois = new ObjectInputStream( in );
Date d = (Date) ois.readObject();
ois.close();
```