Comunicazione tra processi tramite socket TCP/IP

Corso di Sistemi Operativi IEL/IDT, a.a. 2002-2003

Ing. Jürgen Assfalg

Su una macchina, un processo può aprire una *server ocket*, ovvero una porta di ascolto, per consentire a processi in esecuzione su altre macchine (o, al limite anche sulla stessa), connesse alla prima attraverso una rete di comunicazione con protocollo IP, di comunicare con il processo stesso. Ad un tale schema si fa spesso riferimento con il termine *client/server*. Infatti, il primo processo solitamente fornisce un servizio, al quale uno o più clienti operanti sulla stessa rete possono accedere. Per poter accedere ad un servizio attivato su una determinata porta, ovvero per poter comunicare con il processo che ha aperto la porta, è necessario specificare l'indirizzo IP della macchina su cui è in esecuzione il processo e la porta sulla quale questo è in ascolto.

Nella figura seguente è illustrata la caso in cui un client invia un messaggio ad un server attraverso il protocollo TCP/IP:

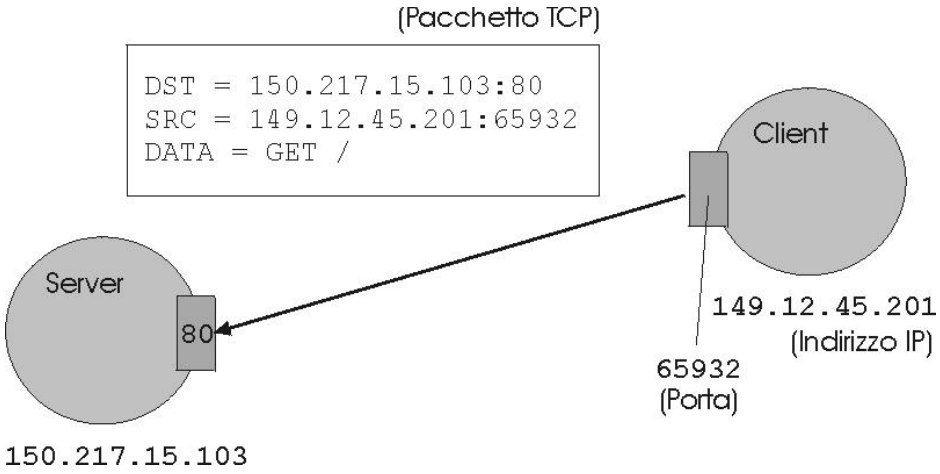

Figura 1: Il client inizia la comunicazione verso il server attivando una porta di comunicazione (*socket*) ed inviando un messaggio alla porta di ascolto sul server (*server socket*)

In particolare, in figura è illustrato l'avvio di una connessione da parte del client (indirizzo e porta della sorgente sono specificati nel pacchetto) verso una porta di ascolto del server. Il processo server si appoggia al sistema operativo per poter aprire una porta in ascolto, specificandone il numero. Tipicamente, per un dato servizio è prevista una porta predefinita. Nel caso dei web server, la porta predefinita è la porta numero 80, come nell'esempio. Anche il client si appoggia al suo sistema operativo per iniziare una comunicazione con un processo remoto. Il client richiede al sistema una connessione. A questo scopo viene aperta dal sistema una porta, il cui numero, però, non deve essere necessariamente fissato. Infatti, la comunicazione può comunque avvenire perché nel pacchetto inviato dal client al server è specificato, oltre all'indirizzo IP della macchina remota (in maniera tale da consentire il corretto instradamento del pacchetto sulla rete IP) e alla porta su cui gira il processo server, anche la porta aperta sulla macchina client appositamente per consentire questa comunicazione.

Il processo server in ascolto sulla porta specificata accetta la connessione entrante e apre una nuova socket per comunicare direttamente con il client. La server socket, infatti, ha il solo scopo di raccogliere le richieste di connessione entranti. Dalla socket così creata, il server legge la richiesta. Nell'esempio questa è rappresentata dalla stringa "GET /",

che, secondo il protocollo HTTP utilizzato per il web, corrisponde alla richiesta della homepage. Il server, una volta interpretata la richiesta, provvede ad inviare la risposta al client. Il messaggio di risposta potrà essere composto da una sequenza di pacchetti TCP (la lunghezza di un pacchetto TCP, infatti, è limitata) indirizzati alla porta specificata dal client al momento della richiesta. Il client potrà quindi leggere i dati e processarli (nel caso di una pagina web, il browser provvederà a visualizzarla).. La risposta è illustrata nella seguente figura:

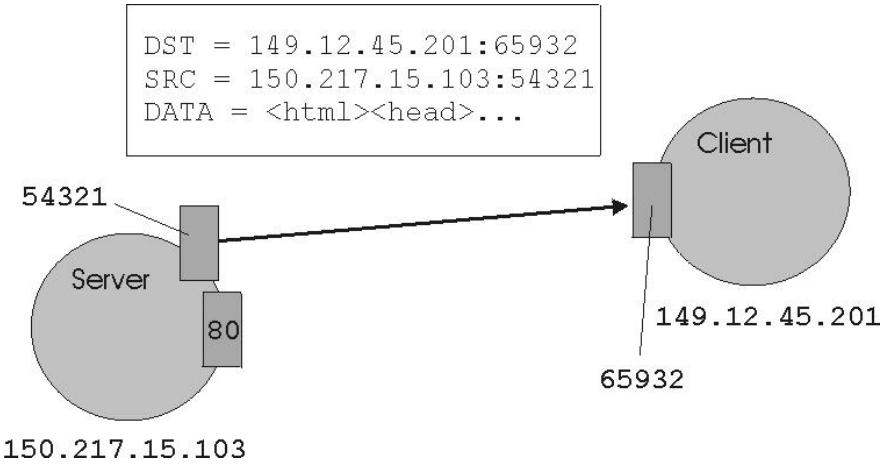

Figura 2: Una volta accettata una richiesta entrante sulla porta di ascolto, il processo server apre una nuova porta per la comunicazione diretta con il client.

Mentre la comunicazione tra server e client procede tra le porte appositamente aperte allo scopo, la porta di ascolto si pone in attesa di altre richieste. Nel caso di un'applicazione ad un singolo thread (o processo), il server non è in grado di soddisfare più di una richiesta per volta. In questo caso è solitamente prevista una coda che raccoglie le richieste entranti.

Al termine della comunicazione, le porte appositamente aperte da server e client devono essere chiuse. Infatti, le porte di comunicazione rappresentano una risorsa limitata per un sistema operativo, e debbono essere a questo restituite quando non sono più necessarie.

Per realizzare una comunicazione tra processi sulla base del protocollo TCP/IP, in Java si possono utilizzare le classi `ServerSocket` e `Socket` (del package `java.net`) per aprire rispettivamente una porta di ascolto e una porta di comunicazione. Due semplici programmi per realizzare un server ed un client sono riportati qui di seguito:

```
package comunicazione.tcp;

/*    classe Java che realizza un semplice server TCP.
      Il server legge i caratteri inviati dal client, e li rimanda indietro.
*/

import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.ServerSocket;

public class TCPEchoServer {
      public static void main( String args[] ) {
            ServerSocket server_socket = null;
```

```
            int porta = Integer.parseInt( args[ 0 ] );
            try {
            /*    crea la porta su cui ascoltare istanziando un oggetto
                  della classe ServerSocket. Al costruttore viene passato
                  l'identificatore della porta su cui si intende porre in
                  ascolto il processo per fornire il servizio
            */
                  server_socket = new ServerSocket( porta );
            } catch ( IOException ioe ) {
                  System.err.println( "impossibile creare socket" );
                  System.exit( 1 );
            }
            System.out.println( "server attivo sulla porta " + porta );
            Socket client_socket = null;
            boolean esegui = true;
            while ( esegui ) {
                  /*    il server si blocca in ascolto sulla porta finche' non
                        arriva una richiesta da parte in un client. Il metodo
                        restituisce quindi un oggetto di tipo socket, attraverso
                        il quale il server puo' comunicare con il client
                  */
                  try {
                        client_socket = server_socket.accept();
                        /*    ottiene lo stream di ingresso collegato alla
                              socket, ovvero il canale attraverso il quale il
                              client invia la richiesta al server
                        */
                        InputStream is = client_socket.getInputStream();
                        /*    ottiene lo stream di uscita collegato alla socket,
                              ovvero il canale attraverso il quale il server
                              invia la risposta al client
                        */
                        OutputStream os = client_socket.getOutputStream();
                        /*    legge i dati sullo stream di ingresso e li ricopia
                              sullo stream di uscita
                        */
                        boolean stop = false;
                        while ( ! stop )
                        {
                              int b = is.read();
                              if ( -1 == b )
                                    stop = true;
                              else
                                    os.write( (byte) b );
                        }
                  } catch ( IOException ioe ) {
                        System.err.println( "errore di I/O" );
                  } finally {
                        try {
                              client_socket.close();
                        } catch ( IOException ioe ) {}
                  }
            }
            /*    si chiude la socket (anche se in pratica questa istruzione non
                  verra' eseguita perche' non e' prevista la possibilita' di
                  uscire dal ciclo infinito di servizio)
            */
            try {
                  server_socket.close();
            } catch ( IOException ioe ) {}
      }
}
```

Per lanciare il server:

```
java -cp . comunicazione.tcp.TCPEchoServer <porta>
```

e per verificarne il funzionamento:

```
telnet localhost <porta>
```

e quindi digitare caratteri. Nota bene: questo server non è in grado di servire più richieste contemporaneamente.

Il seguente programma accede al servizio di echo. In particolare, esso crea una socket per connetterci al servizio e vi invia una stringa (fornita come argomento a linea di comando). La risposta del server viene visualizzata sullo schermo.

```
package comunicazione.tcp;

/**
*           classe Java che realizza un semplice client TCP.
*/

import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;

public class TCPEchoClient {
      public static void main( String args[] ) {
            Socket socket = null;
            try {
                  String host = args[ 0 ];
                  int porta = Integer.parseInt( args[ 1 ] );
                  socket = new Socket( host, porta );
                  InputStream is = socket.getInputStream();
                  OutputStream os = socket.getOutputStream();
                  /*    copia in un vettore di bytes la rappresentazione
                        della stringa che si intende inviare al server
                  */
                  byte[] dati = args[ 2 ].getBytes();
                  /*    invia i dati al server
                  */
                  os.write( dati );
                  /*    chiude lo stream per l'invio dei dati al server
                        (la richiesta e' completata)
                  */
                  socket.shutdownOutput();
                  boolean stop = false;
                  while ( ! stop )
                  {
                        int b = is.read();
                        if ( -1 == b )
                        {
                              /*    e' stata raggiunta la fine dello stream
                              */
                              stop = true;
                        }
                        else
                        {
                              System.out.print( (char) b );
                              System.out.flush();
                        }
                  }
                  socket.close();
```

```
            } catch ( UnknownHostException uhe ) {
                  System.err.println( "host sconosciuto" );
            } catch ( IOException ioe ) {
                  System.err.println( "errore di I/O" );
            } finally {
                  try {
                        if ( null != socket )
                              socket.close();
                  } catch ( IOException ioe ) {}
            }
      }
}
```

Per lanciare il client:

```
java -cp . comunicazione.tcp.TCPEchoServer <host> <porta> <stringa>
```

Come già anticipato, la soluzione descritta sopra per il server presenta uno svantaggio: il server non è in grado di gestire più richieste provenienti da diversi client. Le richieste che pervengono mentre un'altra richiesta viene gestita, sono inserite in una coda, per cui si ha una serializzazione delle richiesta. Per rimediare a tale inconveniente si può allora prevedere l'uso di threads, per cui il programma server accetta le connessioni entranti, e le inoltra ad un certo numero di threads di servizio. Il thread con il server svolge quindi il ruolo di smistatore (*dispatcher*), mentre gli altri threads svolgono il lavoro vero e proprio (*workers*). Il principio di funzionamento è illustrato nella figura seguente:

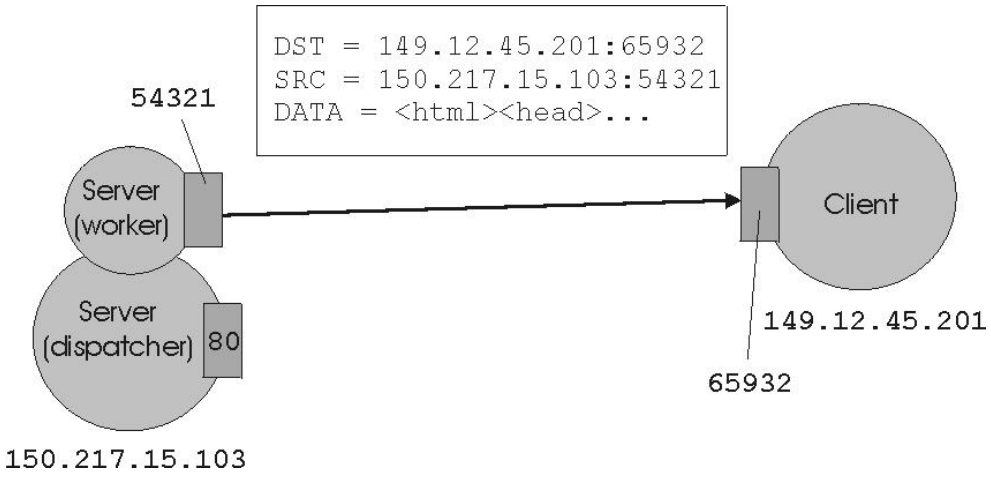

Figura 3: In un server multi-threaded, la gestione della comunicazione con il client viene delegata ad un thread di servizio (*worker*). Il thread di smistatamento (*dispatcher*) può tornare in ascolto e eventualmente avviare nuovi threads di servizio per le nuove richieste entranti.

Mentre una richiesta di un client viene gestita da un thread di lavoro, il thread di smistamento può tornare a gestire le richieste di connessione che pervengono alla porta di ascolto, ed eventualmente avviare nuovi thread di servizio per le nuove richieste. In questo modo i client non si influenzano a vicenda. Infatti, nel caso di un singolo thread, se un client è collegato al server attraverso un collegamento lento, un secondo client che dispone di una notevole larghezza di banda verso il server e che effettui una richiesta successivamente al primo, dovendo attendere che questo sia stato servito, può vedersi ritardare notevolmente la risposta. Il principio è illustrato nella seguente figura:

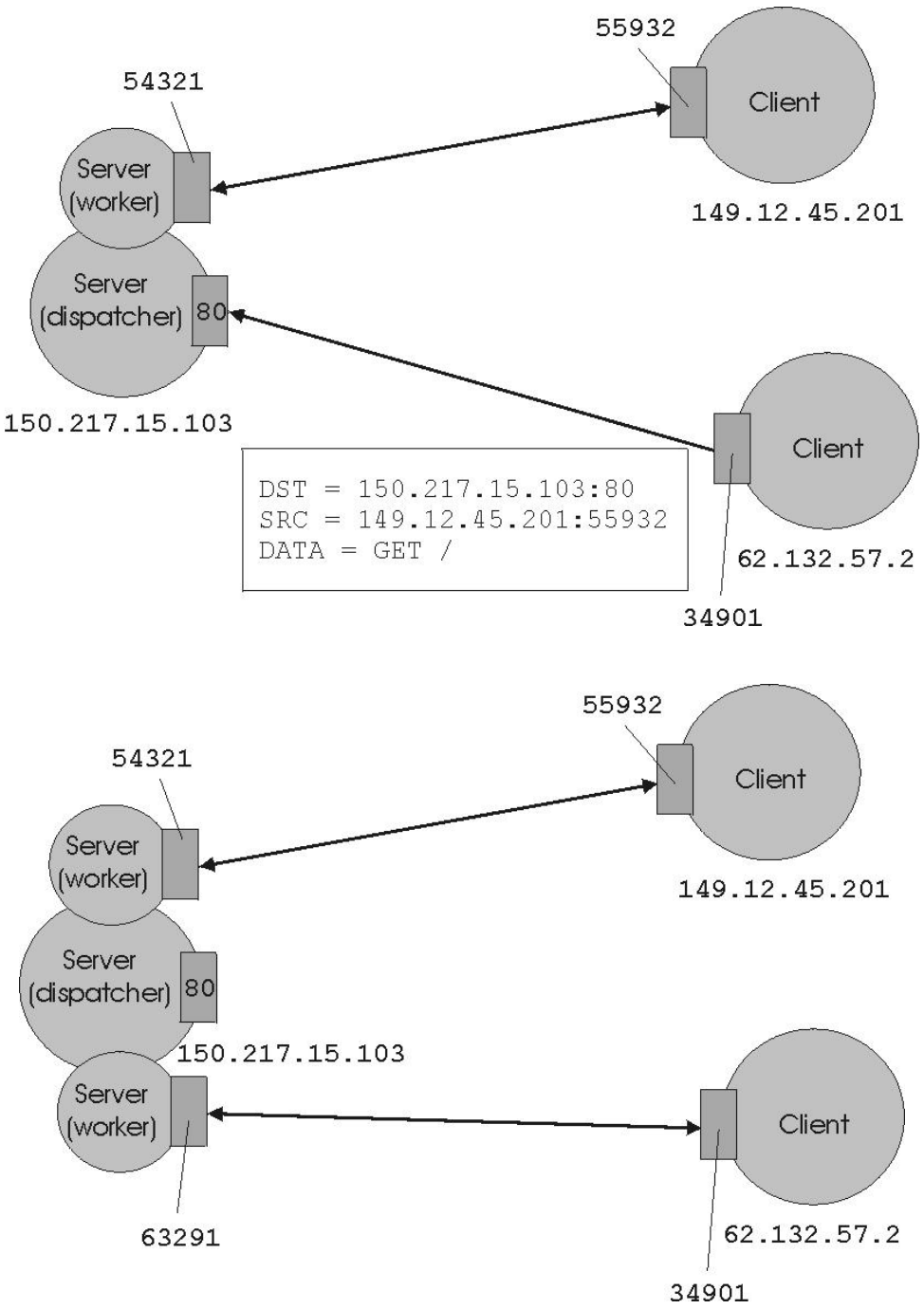

Figura 4: in un server multi-threaded il thread di smistamento si occupa soltanto di accogliere le richieste entranti e generare un thread di servizio per ognuna di esse. In questo modo la gestione delle diverse richieste può svolgersi in parallelo, senza che i singoli client si influenziono a vicenda (velocità di elaborazione, larghezza di banda).

Un server che opera secondo lo schema appena esposto può essere realizzato mediante le seguenti classi (possono essere salvate in un unico file avente nome `TCPMultiThreadedEchoServer.java`):

```
package comunicazione.tcp;

/*    classe Java che realizza un semplice server TCP.
      Il server legge i caratteri inviati dal client, e li rimanda indietro.
*/

import java.io.InputStream;
```

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.ServerSocket;

class Worker extends Thread {
	Worker( Socket clientSocket ) {
		this.clientSocket = clientSocket;
	}

	public void run() {
		try
		{
			/*	ottiene lo stream di ingresso collegato alla
				socket, ovvero il canale attraverso il quale il
				client invia la richiesta al server
			*/
			InputStream is = clientSocket.getInputStream();
			/*	ottiene lo stream di uscita collegato alla socket,
				ovvero il canale attraverso il quale il server
				invia la risposta al client
			*/
			OutputStream os = clientSocket.getOutputStream();
			/*	legge i dati sullo stream di ingresso e li ricopia
				sullo stream di uscita
			*/
			boolean stop = false;
			while ( ! stop )
			{
				int b = is.read();
				if ( -1 == b )
					stop = true;
				else
					os.write( (byte) b );
			}
		} catch ( IOException ioe )
		{
			System.err.println(
				"errori di I/O durante comunicazione con client"
			);
		}
		finally {
			try {
				clientSocket.close();
			} catch ( IOException ioe ) {}
		}
	}

	private Socket clientSocket = null;
}

public class TCPMultiThreadedEchoServer extends Thread {

	public TCPMultiThreadedEchoServer ( int porta ) {
		try {
		/*	crea la porta su cui ascoltare istanziando un oggetto
			della classe ServerSocket. Al costruttore viene passato
			l'identificatore della porta su cui si intende porre in
			ascolto il processo per fornire il servizio
		*/
			serverSocket = new ServerSocket( porta );
		} catch ( IOException ioe ) {
			System.err.println( "impossibile creare socket" );
```

```
                System.exit( 1 );
        }
        System.out.println( "server attivo sulla porta " + porta );
    }

    public void run() {
        while ( true ) {
            /*    il server si blocca in ascolto sulla porta finche' non
                  arriva una richiesta da parte in un client. Il metodo
                  restituisce quindi un oggetto di tipo socket, attraverso
                  il quale il server puo' comunicare con il client
            */
            Socket client_socket = null;
            try {
                  client_socket = serverSocket.accept();
                  Worker worker_thread = new Worker( client_socket );
                  worker_thread.start();
            } catch ( IOException ioe ) {
                  System.err.println( "errore di I/O durante ascolto" );
                  try {
                        client_socket.close();
                  } catch ( IOException ioe2 ) {}
            }
        }
    }

    private ServerSocket serverSocket = null;

    public static void main( String args[] ) {
        int porta = Integer.parseInt( args[ 0 ] );
        TCPMultiThreadedEchoServer dispatcher =
              new TCPMultiThreadedEchoServer( porta );
        dispatcher.start();
    }
}
```