Capitolo 3

Comunicazione tra processi

3.1 Socket TCP/IP

Un *socket* è definito come un punto finale per una comunicazione. Come tale, un socket TCP/IP è caratterizzato da un indirizzo IP (*Internet Protocol*) e da un numero di porta.

Su una macchina, un processo può aprire una server socket, ovvero una porta di ascolto, per consentire a processi in esecuzione su altre macchine (o 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.

In Fig.3.1 è illustrato il caso in cui un client invia un messaggio ad un server attraverso il protocollo TCP/IP.
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 la porta di ascolto del server. Il processo server si appoggia al sistema operativo per poter aprire una porta in ascolto, specificando il numero. Tipicamente, per servizi standard la porta è predefinita (ad esempio, porta 21 per `ftp`, porta 23 per `telnet`). Nel caso dei server `http`, la porta predefinita è la porta numero 80, come nell'esempio. Tutte le porte con numeri compresi tra 0 e 1024 sono considerate corrispondenti a servizi noti. Le altre possono essere istanziate ai processi per instaurare canali di comunicazione.

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 non è prefissato ma viene scelto dal sistema. Infatti, la comunicazione

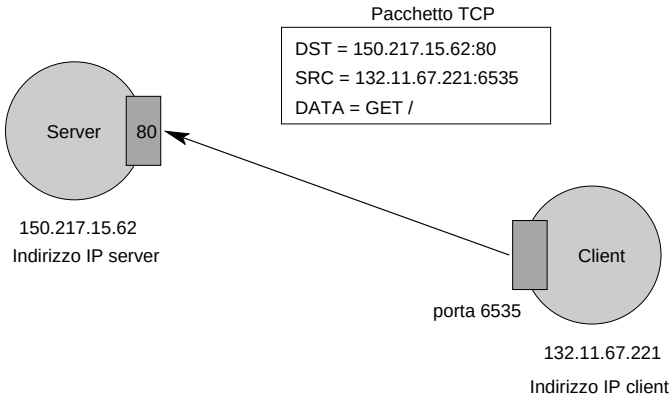

Figura 3.1: Il client inizia la comunicazione verso il server aprendo una porta di comunicazione (*socket*) ed inviando un messaggio alla porta di ascolto sul server (*server socket*).

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 in Fig.3.2.
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 a 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 utilizzando il protocollo

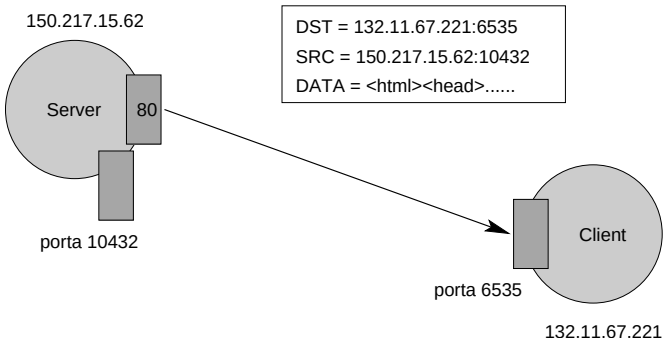

Figura 3.2: Una volta accettata una richiesta entrante sulla porta di ascolto, il processo server apre una nuova porta per la comunicazione diretta con il client.

TCP/IP, in Java si possono utilizzare le classi `ServerSocket` e `Socket` del package `java.net`. Le due classi aprono, rispettivamente, una porta di ascolto su un server ed una porta di comunicazione.

Nel seguito è riportato un semplice esempio di comunicazione client/server basata su socket. In particolare, il server fornisce un servizio di "eco," attraverso il quale è in grado di ricevere una generica stringa da un client e di restituirla allo stesso. Lo stesso problema verrà affrontato fornendo tre distinte soluzioni per il server: server a singolo thread; server multithread; server con pool di thread

3.1.1 Esempio: server a singolo thread

La soluzione è ottenuta attraverso due classi. La classe `TCPEchoClient` (vedi Fig.3.3) modella il comportamento del processo client che accede al servizio di "eco". In particolare, il client si collega al server ed invia una stringa fornita come argomento a linea di comando. La risposta del server (implementato dalla classe `TCPEchoServer`, di Fig.3.4) viene visualizzata sullo schermo.

```
import java.io.*;
import java.net.*;

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();

            // trasforma la stringa data in ingresso in un vettore di byte
            byte[] dati = args[2].getBytes();

            // invia il vettore di byte al server scrivendo sullo stream del socket
            // di comunicazione
            os.write(dati);

            // chiude l'outputstream per l'invio dei dati al server
            socket.shutdownOutput();

            // inizia la lettura dell'eco dei caratteri prodotti dal server
            boolean stop = false;
            while (!stop) {
                int b = is.read();
                if (b==-1)  // condizione di fine stream
                    stop = true;
                else {
                    System.out.print((char) b);
                    System.out.flush();
                }
            }
        }
        catch (UnknownHostException uhe) {
            System.err.println("host sconosciuto");
        }
        catch (IOException ioe) {
            System.err.println("errore di I/O");
        }
        finally {
            try {
                if (socket!=null)
                    socket.close();
            }
            catch (IOException ioe) {
                System.err.println("errore di I/O");
            }
        }
    }
}
```

Figura 3.3: Classe che implementa il client (file `TCPEchoClient.java`).

```
import java.io.*;
import java.net.*;

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) {
            try {
                /*  il server si blocca in ascolto sulla porta fino a quando
                    non arriva una richiesta da parte di un client. Il metodo
                    restituisce un oggetto di tipo socket attraverso il quale
                    il server puo' comunicare con il client
                */
                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 dallo stream di ingresso e li copia
                    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) {
                    System.err.println("errore di I/O");
                }
            }
        }

        /*  chiude la socket (in pratica questa istruzione non
            verra' eseguita in quanto non e' prevista la possibilita'
            di uscire dal ciclo infinito di servizio)
        */
        try {
            server_socket.close();
        }
        catch (IOException ioe) {
            System.err.println("errore di I/O");
        }
    }
}
```

Figura 3.4: Classe che implementa il server TCP (file `TCPEchoServer.java`).

Per lanciare il server:

`java TCPEchoServer` $<$ `porta` $>$

e per verificarne il funzionamento:

`java TCPEchoClient` $<$ `host` $>$ $<$ `porta` $>$ $<$ `stringa` $>$

dove `<host>` è l'indirizzo IP `127.0.0.1` nel caso in cui client e server siano attivi sulla stessa macchina. In alternativa, è possibile collegarsi al server utilizzando il comando telnet:

`telnet localhost` $<$ `porta` $>$

e quindi digitando i caratteri della stringa di cui si vuole ottenere l'eco.

3.1.2 Esempio: server multithread

La soluzione descritta per il server in Sez.3.1.1, presenta uno svantaggio: il server non è in grado di gestire contemporaneamente 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 richieste. Per rimediare a tale inconveniente si può allora prevedere l'uso di più thread, per cui il programma server accetta le connessioni entranti, e le inoltra ad un certo numero di thread di servizio. Il thread del server svolge quindi il ruolo di smistatore (*dispatcher*), mentre gli altri thread svolgono il lavoro vero e proprio (*workers*). Il principio di funzionamento è illustrato in Fig.3.5.

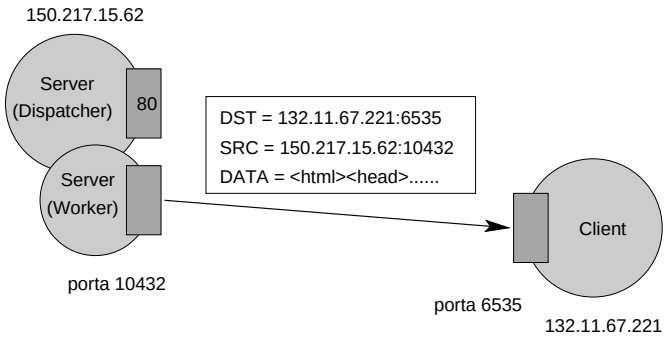

Figura 3.5: In un server multi-threaded, la gestione della comunicazione con il client viene delegata ad un thread di servizio (*worker*). Il thread di smistamento (*dispatcher*) può tornare in ascolto ed eventualmente avviare thread di servizio per le nuove richieste entranti.

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 arrivano 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. Questo può determinare un'attesa dovuta ad un *effetto convoglio*. Il caso di servizio di più richieste da parte di thread separati è illustrato in Fig.3.6. Questa soluzione incrementa anche la velocità di risposta alle richieste e l'utilizzo della larghezza di banda del server.
Un server che opera secondo lo schema appena esposto può essere realizzato mediante le classi di Fig.3.7 (le classi possono essere salvate in un unico file `TCPMultiThreadedEchoServer.java`, in cui è pubblica solo la classe

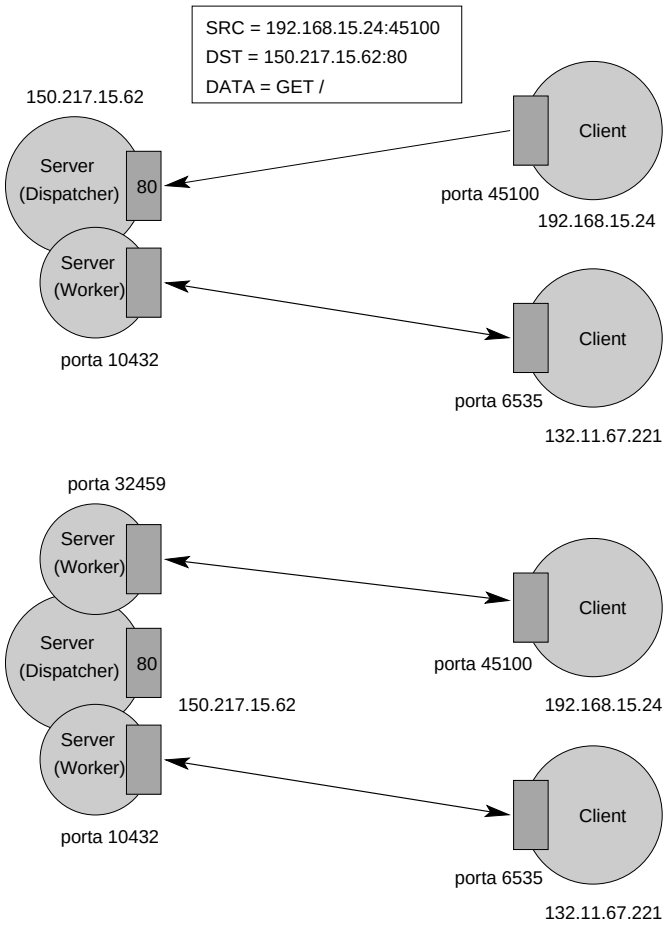

Figura 3.6: In un server multi-threaded il thread di smistamento si occupa soltanto di accogliere le richieste entranti e generare un thread di servizio per ognuna di esse. In questo modo, la gestione delle diverse richieste può svolgersi in parallelo, senza che i singoli client si influenzino a vicenda.

`TCPMultiThreadedEchoServer`).

```
import java.io.*;
import java.net.*;

class Worker extends Thread
{
    private Socket clientSocket = null;

    Worker(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    public void run() {
        try {
            /*  ottiene lo stream di ingresso collegato alla
                socket. E' il canale attraverso il quale il
                client invia la richiesta al server
            */
            InputStream is = clientSocket.getInputStream();

            /*  ottiene lo stream di uscita collegato alla socket.
                E' il canale attraverso il quale il server
                invia la risposta al client
            */
            OutputStream os = clientSocket.getOutputStream();

            /*  legge i dati dallo stream di ingresso e li copia
                sullo stream di uscita
            */
            boolean stop = false;
            while (!stop) {
                int b = is.read();
                if (b==-1)
                    stop = true;
                else
                    os.write((byte)b);
            }
        }
        catch (IOException ioe) {
            System.err.println("errore di I/O durante comunicazione");
        }
        finally {
            try {
                clientSocket.close();
            }
            catch (IOException ioe) {
```

```
                System.err.println("errore di I/O");
            }
        }
    }
}

public class TCPMultiThreadedEchoServer
{
    public static void main(String args[]) {
        ServerSocket serverSocket = null;
        int porta = Integer.parseInt(args[0]);

        try {
            /*  crea la porta di ascolto istanziando un oggetto
                della classe ServerSocket. Al costruttore viene passato
                il numero della porta su cui porre in ascolto il server
            */
            serverSocket = new ServerSocket(porta);
        }
        catch (IOException ioe) {
            System.err.println("impossibile creare socket");
            System.exit(1);
        }
        System.out.println("server attivo sulla porta " + porta);

        while (true) {
            Socket client_socket = null;
            try {
                /*  il server si blocca in ascolto sulla porta in attesa
                    di una richiesta. Il metodo restituisce un oggetto di
                    tipo socket, attraverso il quale il server può comunicare
                    con il client
                */
                client_socket = serverSocket.accept();

                /*  ogni richiesta è gestita da un thread separato
                */
                Worker worker_thread = new Worker(client_socket);
                worker_thread.start();
            }
            catch (IOException ioe) {
                System.err.println("errore di I/O durante ascolto");
            }
        }
    }
}
```

Il maggiore inconveniente della soluzione precedente è rappresentato dalla possibile crescita non controllata del numero di thread in esecuzione per

Figura 3.7: Classe che implementa il server TCP multithreaded (file `TCPMultiThreadedEchoServer.java`).

gestire le richieste. Infatti, con questo schema, per ogni nuova richiesta da parte di un client è generato un nuovo thread gestore della comunicazione.

3.1.3 Esempio: server con thread pooling

Un approccio che mira a risolvere il problema evidenziato nella Sez.3.1.2, è il *thread pooling*. L'idea alla base di questa soluzione è quella di creare all'avvio del server un insieme (*pool*) di thread: se non ci sono richieste da servire i thread si sospendono utilizzando le primitive di sincronizzazione Java. Nel momento in cui una o più richieste arrivano al server, queste sono servite risvegliando i thread creati all'avvio del server. In questo modo, si ottiene un duplice vantaggio: il massimo numero di thread che possono essere contemporaneamente in esecuzione è fissato dal numero di thread del pool; i thread sono creati solo all'avvio del server e dopo quel momento non sono mai distrutti ma solo sospesi e riattivati, riducendo in questo modo l'overhead di gestione dei thread.

In Fig.3.8, sono riportate le classi (`PooledConnectionHandler` e `TCPPooledEchoServer`) che realizzano il server secondo lo schema del pool di thread.

```
import java.io.*;
import java.net.*;

class PooledConnectionHandler implements Runnable
{
    protected Socket connection;

    // la lista è "static" cioè condivisa tra i thread del pool
    // a livello di classe.  Richiede un accesso sincronizzato
    protected static List pool = new LinkedList();

    public PooledConnectionHandler() {
    }

    public void handleConnection() {
        try {
            /*  ottiene lo stream di ingresso collegato alla
                socket, ovvero il canale attraverso il quale il
                client invia la richiesta al server
            */
            InputStream is = connection.getInputStream();

            /*  ottiene lo stream di uscita collegato alla socket,
                ovvero il canale attraverso il quale il server
```

```
            invia la risposta al client
        */
        OutputStream os = connection.getOutputStream();

        /*  legge i dati sullo stream di ingresso e li copia
            sullo stream di uscita
        */
        boolean stop = false;
        while (!stop) {
            int b = is.read();
            if (b==-1)
                stop = true;
            else
                os.write((byte)b);
        }
    }
    catch (IOException ioe) {
        System.err.println("errori I/O durante comunicazione con client");
    }
    finally {
        try {
            connection.close();
        }
        catch (IOException ioe) {
            System.err.println("errori I/O");
        }
    }
}

public static void processRequest(Socket requestToHandle) {
    synchronized(pool) {
        // appena arriva una richiesta i thread del pool sospesi sono
        // risvegliati
        pool.add(pool.size(),requestToHandle);
        pool.notifyAll();
    }
}

public void run() {
    while (true) {
        synchronized(pool) {
            while (pool.isEmpty()) {
                try {
                    // se non ci sono richieste da servire il thread si sospende
                    pool.wait();
                }
                catch (InterruptedException e) {
                    return;
                }
```

```
                }
                // rimuove un elemento dalla lista delle richieste
                connection = (Socket) pool.remove(0);
            }
            // chiama il metodo per la gestione della richiesta
            handleConnection();
        }
    }
}

public class TCPPooledEchoServer
{
    protected int maxConnections;
    protected int listenPort;
    protected ServerSocket serverSocket;

    public TCPPooledEchoServer(int port,int max) {
        listenPort = port;
        maxConnections = max;
    }

    public void setUpHandlers() {
        for (int i=0;i<maxConnections;i++) {
            PooledConnectionHandler currentHandler = new PooledConnectionHandler();
            new Thread(currentHandler, "Handler" + i).start();
        }
    }

    public void acceptConnections() {
        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(listenPort,5);
        }
        catch (IOException ioe) {
            System.err.println("impossibile creare socket");
            System.exit(1);
        }
        System.out.println("server attivo sulla porta " + listenPort);

        while (true) {
            Socket client_socket = null;
            try {
                /*  il server si blocca in ascolto sulla porta finché non
                    arriva una richiesta da parte di un client. Il metodo
                    restituisce quindi un oggetto di tipo socket, attraverso
```

```
                    il quale il server può comunicare con il client.
                    La richiesta è gestita dal pool di thread
                */
                client_socket = serverSocket.accept();
                PooledConnectionHandler.processRequest(client_socket);
            }
            catch (IOException ioe) {
                System.err.println("errore di I/O durante ascolto");
            }
        }
    }

    public static void main(String args[]) {
        TCPPooledEchoServer server = new
            TCPPooledEchoServer(Integer.parseInt(args[0]),Integer.parseInt(args[1]));

        // crea e inizializza il pool di thread
        server.setUpHandlers();

        // accetta e gestisce la connessione
        server.acceptConnections();
    }
}
```

Figura 3.8: Classe che implementa il server TCP con pool di thread (file `TCPPooledEchoServer.java`).

Osservare che nella classe `TCPPooledEchoServer`, l'oggetto di tipo `ServerSocket` è istanziato utilizzando il costruttore `ServerSocket(listenPort,5)`, che oltre alla porta di ascolto riceve in ingresso la lunghezza della coda associata alla porta di ascolto. Il valore di default per la lunghezza di questa coda è 50. Nell'esempio, la lunghezza della coda è limitata a 5, numero di thread del pool.

Bibliografia

[1] S. Oaks, H. Wong. “Java Threads,” second edition, *O’Reilly*, 2003.

[2] *Java sockets 101*: http://ibm.com/developerWorks.