
RA3. Programa mecanismes de comunicació en xarxa emprant sockets

Índex

1. Introducció	1
2. Escenaris de comunicació en xarxa	1
3. El model client-servidor	2
4. Classes Java per a comunicacions en xarxa	2
API de baix nivell	3
API d'alt nivell	3
4.1. La classe InetAddress	3
4.2. La classe URL	4
4.3. La classe URLConnection	4
5. Els sockets: concepte i tipus	5
Funcionament general	5
Tipus de sockets	6
6. Sockets orientats a connexió (TCP)	6
6.1. Classe ServerSocket	6
6.2. Classe Socket	7
6.3. Esquema general de funcionament	7
6.4. Exemple: servidor i client bàsics (TCP)	8
6.5. Exemple: invertir text amb PrintWriter i BufferedReader	9
7. Sockets no orientats a connexió (UDP)	11
7.1. Classe DatagramPacket	11
7.2. Classe DatagramSocket	11
7.3. Esquema de funcionament	12
7.4. Exemple: servidor i client UDP	12
8. MulticastSocket	14
Esquema d'un servidor multicast	14
Esquema d'un client multicast	14
9. Enviament d'objectes a través de sockets	15
9.1. Objectes en sockets TCP	15
9.2. Objectes en sockets UDP	16
10. Atenció a múltiples clients amb fils	16
10.1. Exemple: servidor multiclient (retorna la longitud de cada línia)	17
10.2. Consideracions importants	18
10.3. Cas d'ús típic: servidor de xat	18
11. Models de comunicació en arquitectures distribuïdes	19
12. Depuració i bones pràctiques	19
Índex de figures	21

Cicle: Desenvolupament d'Aplicacions Multiplataforma (DAM)

Mòdul: 0490. Programació de serveis i processos

1. Introducció

Antigament, programar aplicacions que es comuniquessin entre màquines diferents era una tasca complexa: calia conèixer detalls de les capes del protocol de xarxa i, fins i tot, del maquinari. Java va simplificar enormement aquesta feina gràcies al paquet **java.net**, que proporciona classes per establir connexions, crear servidors, enviar i rebre dades i gestionar la resta d'operacions pròpies de la comunicació en xarxa.

En aquest RA aprendrem a:

- Identificar quan una aplicació necessita comunicar-se amb una altra a través de la xarxa.
- Distingir els rols de **client** i **servidor**.
- Utilitzar les classes de `java.net` per crear **sockets**.
- Desenvolupar aplicacions client-servidor amb **TCP** i **UDP**.
- Atendre múltiples clients simultàniament fent servir **fils (threads)**.

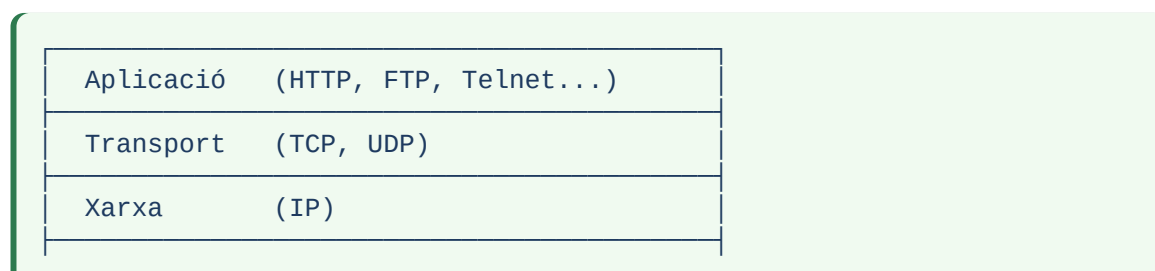
2. Escenaris de comunicació en xarxa

Moltes aplicacions necessiten intercanviar informació amb altres aplicacions que s'executen en màquines diferents (o fins i tot a la mateixa màquina, en processos diferents). Alguns escenaris típics:

Escenari	Exemple
Aplicació web	Navegador (client) que sol·licita una pàgina a un servidor HTTP
Missatgeria instantània / xat	Diversos clients que envien missatges a través d'un servidor central
Joc en xarxa	Diversos jugadors que sincronitzen l'estat de la partida
Accés a bases de dades remotes	Una aplicació client consulta dades a un servidor de BD
Transmissió multimèdia	Àudio/vídeo en temps real (streaming)
Serveis de xarxa	DNS, NFS, SNMP...

En tots aquests casos cal establir un **canal de comunicació** entre dos (o més) processos, que poden estar a la mateixa màquina o en màquines diferents connectades per una xarxa (LAN, Internet...).

El model TCP/IP defineix 4 capes o nivells d'abstracció:



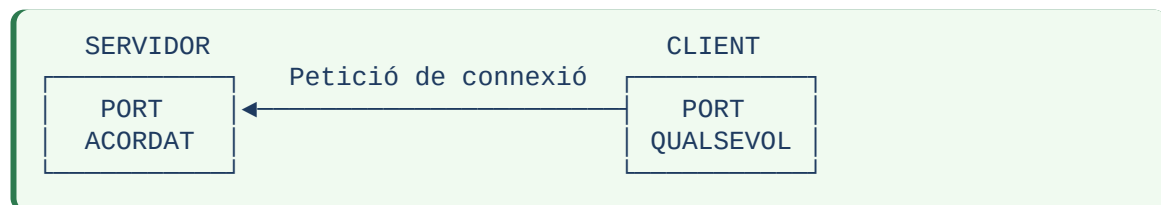
Enllaç (controlador de dispositiu)

Quan programem en Java amb el paquet `java.net`, treballem a la **capa d'aplicació**, sense haver de preocupar-nos directament de com es fragmenten o encaminen els paquets a les capes inferiors.

3. El model client-servidor

En una comunicació en xarxa cal distingir dos rols:

- **Servidor**: s'executa en una màquina concreta i queda a l'espera ("escoltant") peticions de connexió en un port determinat.
- **Client**: coneix el nom (o IP) de la màquina servidora i el port pel qual escolta, i inicia la comunicació sol·licitant la connexió.



Característiques bàsiques:

- El **servidor** ha de conèixer i publicar un port fix (el "port acordat") pel qual escolta.
- El **client** pot utilitzar qualsevol port lliure assignat pel sistema per connectar-se.
- Un cop establerta la connexió, el servidor sol crear un **nou punt de comunicació** (nou socket) per atendre aquest client concret, mentre l'original queda lliure per acceptar noves connexions.
- Els dos extrems, un cop connectats, poden llegir i escriure dades en ambdues direccions.

4. Classes Java per a comunicacions en xarxa

El paquet `java.net` es pot dividir en dos grans blocs:

API de baix nivell

- **Adreces:** identificadors de xarxa (adreces IP) → classe `InetAddress`.
- **Sockets:** mecanisme bàsic de comunicació bidireccional → `Socket`, `ServerSocket`, `DatagramSocket`.
- **Interfícies de xarxa.**

API d'alt nivell

- **URI:** identificadors de recursos universals.
- **URL:** localitzadors de recursos → classe `URL`.
- **Connexions:** representen la connexió a un recurs apuntat per un URL → `URLConnection`.

4.1. La classe `InetAddress`

Representa una adreça IP. Té dues subclasses (`Inet4Address` i `Inet6Address`), però normalment no cal utilitzar-les directament.

Mètode	Missió
<code>static InetAddress getLocalHost()</code>	Adreça IP de la màquina local
<code>static InetAddress getByName(String host)</code>	Adreça IP a partir d'un nom de màquina, domini o IP
<code>static InetAddress[] getAllByName(String host)</code>	Totes les adreces IP assignades a una màquina
<code>String getHostAddress()</code>	Adreça IP en forma de cadena
<code>String getHostName()</code>	Nom del host

```
import java.net.*;

public class ProvaInetAddress {
    public static void main(String[] args) {
        try {
            InetAddress local = InetAddress.getLocalHost();
            System.out.println("Nom: " + local.getHostName());
            System.out.println("IP: " + local.getHostAddress());

            InetAddress remot =
                ↪ InetAddress.getByName("www.google.com");
            System.out.println("IP de Google: " +
                ↪ remot.getHostAddress());
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

4.2. La classe URL

Representa un punter a un recurs de la Web. Es divideix en protocol, nom de màquina, port (opcional) i ruta del recurs:

```
http://www.exemple.cat:80/carpeta/pagina.php
|      |      |      |
protocol màquina port fitxer
```

Alguns constructors i mètodes:

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

public class ProvaURL {
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://docs.oracle.com/javase/7");
        System.out.println("Protocol: " + url.getProtocol());
        System.out.println("Host: " + url.getHost());
        System.out.println("Port: " + url.getPort());        // -1 si no
        ↪ s'indica
        System.out.println("Path: " + url.getPath());

        // Llegir el contingut de la URL
        BufferedReader in = new BufferedReader(
            new InputStreamReader(url.openStream()));
        String linia;
        while ((linia = in.readLine()) != null) {
            System.out.println(linia);
        }
        in.close();
    }
}
```

4.3. La classe URLConnection

Permet interactuar amb el recurs apuntat per un URL (llegir capçaleres, enviar dades a un script del servidor, etc.). S'obté invocant `openConnection()` sobre un objecte URL:

```
URL url = new URL("http://localhost/servei.php");
URLConnection con = url.openConnection();
con.setDoOutput(true);    // permetre enviar dades

PrintWriter out = new PrintWriter(con.getOutputStream());
out.write("nom=Marta&curs=DAM2");
out.close();

BufferedReader in = new BufferedReader(
    new InputStreamReader(con.getInputStream()));
String resposta;
```

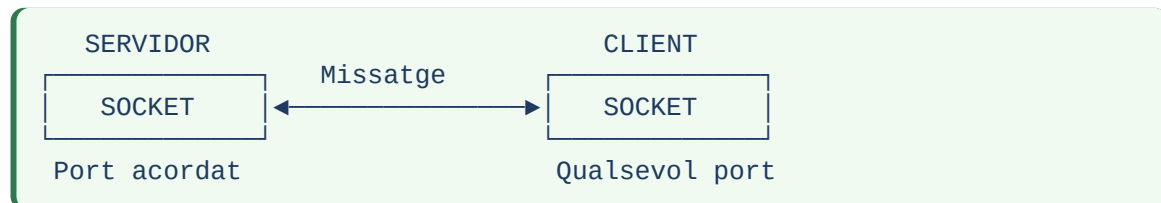
```
while ((resposta = in.readLine()) != null) {  
    System.out.println(resposta);  
}  
in.close();
```

Aquesta API d'alt nivell és útil quan volem parlar amb un servei web existent, però **no ens serveix per crear el nostre propi servidor**. Per això necessitem l'API de baix nivell: els **sockets**.

5. Els sockets: concepte i tipus

Un **socket** és l'abstracció que representa el punt de connexió entre dos processos que es comuniquen en xarxa. Per als processos receptors, el socket va associat a:

- L'**adreça IP** del host on s'executa l'aplicació.
- El **port local** a través del qual es comunica.



Funcionament general

1. El **servidor** crea un socket lligat a un port específic i queda "escoltant" (`accept()`).
2. El **client** coneix el nom de la màquina i el port del servidor, i sol·licita la connexió.
3. Si tot va bé, el servidor **accepta** la connexió i obté un **nou socket** per atendre aquest client concret (el socket original continua escoltant nous clients).
4. Un cop connectats, ambdós processos poden llegir i escriure a través dels seus respectius sockets.

Tipus de sockets

	TCP (orientat a connexió)	UDP (no orientat a connexió)
Fiabilitat	Garanteix l'entrega i l'ordre dels paquets	No garanteix ni l'entrega ni l'ordre
Connexió	Cal establir-la abans de comunicar-se (stream)	No cal connexió; s'envien datagrames independents
Velocitat	Més lent (confirmació de recepció, reintents)	Més ràpid, menys sobrecàrrega
Ús típic	Web (HTTP), correu (SMTP), FTP, Telnet	Streaming d'àudio/vídeo, DNS, NFS, SNMP
Classes Java	Socket, ServerSocket	DatagramSocket, DatagramPacket

Un **stream** és una seqüència ordenada d'unitats d'informació (bytes, caràcters...) que flueix en una direcció: cap fora del procés (sortida) o cap dins (entrada). TCP es basa en aquest concepte de flux continu de dades.

6. Sockets orientats a connexió (TCP)

El paquet `java.net` proporciona dues classes principals: **ServerSocket** (per al servidor) i **Socket** (per al client, i també per representar la connexió acceptada al servidor).

6.1. Classe ServerSocket

Constructor	Missió
<code>ServerSocket()</code>	Socket de servidor sense port associat
<code>ServerSocket(int port)</code>	S'enllaça al port indicat
<code>ServerSocket(int port, int max)</code>	Indica el màxim de peticions en cua
<code>ServerSocket(int port, int max, InetAddress addr)</code>	Especifica també l'adreça IP local

Mètode	Missió
<code>Socket accept()</code>	Espera i accepta una connexió; retorna un <code>Socket</code> per comunicar-se amb el client
<code>void close()</code>	Tanca el <code>ServerSocket</code>
<code>int getLocalPort()</code>	Port local al qual està enllaçat

6.2. Classe Socket

Constructor	Missió
Socket()	Sense port associat
Socket(InetAddress addr, int port)	Es connecta a la IP i port indicats
Socket(String host, int port)	Es connecta al nom de host i port indicats

Mètode	Missió
InputStream getInputStream()	Flux d'entrada per llegir dades del socket
OutputStream getOutputStream()	Flux de sortida per escriure dades
void close()	Tanca el socket
InetAddress getInetAddress()	IP remota a la qual està connectat
int getPort()	Port remot
int getLocalPort()	Port local

6.3. Esquema general de funcionament

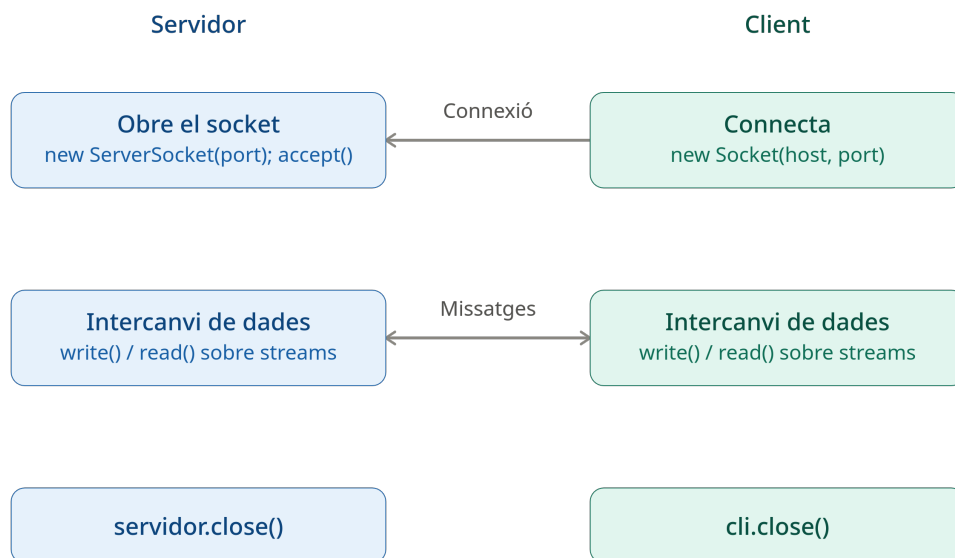


Figura 1: Esquema general de funcionament

IMPORTANT

Ordre de tancament: primer s'han de tancar els fluxos (streams) associats al socket, i després el propi socket.

6.4. Exemple: servidor i client bàsics (TCP)

Servidor (ServidorQuadrat.java): rep un nombre enter del client i li retorna el seu quadrat.

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

public class ServidorQuadrat {
    public static void main(String[] args) throws IOException {
        int numeroPort = 5500;
        ServerSocket servidor = new ServerSocket(numeroPort);
        System.out.println("Esperant client al port " + numeroPort +
            ↪ "...");

        Socket clientConnectat = servidor.accept();
        System.out.println("Client connectat!");

        DataInputStream fluxEntrada =
            new DataInputStream(clientConnectat.getInputStream());
        int valor = fluxEntrada.readInt();
        System.out.println("Valor rebut: " + valor);

        DataOutputStream fluxSortida =
            new DataOutputStream(clientConnectat.getOutputStream());
        fluxSortida.writeInt(valor * valor);

        fluxEntrada.close();
        fluxSortida.close();
        clientConnectat.close();
        servidor.close();
    }
}
```

Client (ClientQuadrat.java):

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

public class ClientQuadrat {
    public static void main(String[] args) throws IOException {
        String host = "localhost";
        int port = 5500;

        Socket client = new Socket(host, port);
        System.out.println("Client iniciat...");

        DataOutputStream fluxSortida =
            new DataOutputStream(client.getOutputStream());
        fluxSortida.writeInt(7);

        DataInputStream fluxEntrada =
            new DataInputStream(client.getInputStream());
    }
}
```

```

        System.out.println("El servidor retorna: " +
            ↪ fluxEntrada.readInt());

        fluxSortida.close();
        fluxEntrada.close();
        client.close();
    }
}

```

AVÍS

Primer cal executar el programa **servidor** (que es queda esperant amb `accept()`), i tot seguit el programa **client**.

6.5. Exemple: invertir text amb `PrintWriter` i `BufferedReader`

En lloc de `DataInputStream/DataOutputStream` també és molt habitual utilitzar `PrintWriter` i `BufferedReader`, que treballen amb línies de text. En aquest exemple el servidor no retorna la mateixa línia, sinó que la **capgira** (posa-la del revés) abans de respondre:

Servidor:

```

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

public class ServidorCapgirat {
    public static void main(String[] args) throws IOException {
        int port = 7000;
        ServerSocket servidor = new ServerSocket(port);
        System.out.println("Esperant connexió al port " + port +
            ↪ "...");

        Socket clientConnectat = servidor.accept();
        System.out.println("Client connectat.");

        PrintWriter sortida =
            new PrintWriter(clientConnectat.getOutputStream(),
                ↪ true);
        BufferedReader entrada = new BufferedReader(
            new InputStreamReader(clientConnectat.getInputStream()));

        String linia;
        while ((linia = entrada.readLine()) != null) {
            if (linia.equalsIgnoreCase("sortir")) break;    // condició
            ↪ de fi
            String capgirada = new
                ↪ StringBuilder(linia).reverse().toString();
            sortida.println(capgirada);
            System.out.println(linia + " -> " + capgirada);
        }
    }
}

```

```

    }

    entrada.close();
    sortida.close();
    clientConnectat.close();
    servidor.close();
}
}

```

Client:

```

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

public class ClientCapgirat {
    public static void main(String[] args) throws IOException {
        Socket client = new Socket("localhost", 7000);

        PrintWriter sortida = new
            ↪ PrintWriter(client.getOutputStream(), true);
        BufferedReader entrada = new BufferedReader(
            new InputStreamReader(client.getInputStream()));
        BufferedReader teclat = new BufferedReader(
            new InputStreamReader(System.in));

        String cadena;
        System.out.print("Escriu una paraula (\\"sortir\\" per acabar):
            ↪ ");
        while ((cadena = teclat.readLine()) != null &&
            ↪ !cadena.equalsIgnoreCase("sortir")) {
            sortida.println(cadena);
            System.out.println(" => " + entrada.readLine());
            System.out.print("Escriu una paraula (\\"sortir\\" per
                ↪ acabar): ");
        }

        sortida.close();
        entrada.close();
        teclat.close();
        client.close();
    }
}

```

Els ports TCP van de 0 a 65535: de 0 a 1023 estan reservats per a serveis privilegiats (HTTP=80, FTP=21, SMTP=25...), de 1024 a 49151 per a aplicacions concretes (MySQL=3306, Oracle=1521...) i de 49152 a 65535 són lliures.

7. Sockets no orientats a connexió (UDP)

Els sockets UDP no estableixen connexió prèvia: cada vegada que s'envia un datagrama cal indicar explícitament l'adreça IP i el port de destinació, i el receptor extreu aquesta informació del mateix paquet rebut.

7.1. Classe DatagramPacket

Representa el paquet de dades. Conté: la cadena de bytes amb el missatge, la longitud, l'adreça IP i el port de destinació (o d'origen, si s'ha rebut).

Constructor	Ús
<code>DatagramPacket(byte[] buf, int length)</code>	Per rebre un datagrama
<code>DatagramPacket(byte[] buf, int length, InetAddress addr, int port)</code>	Per enviar un datagrama

Mètode	Missió
<code>InetAddress getAddress()</code>	IP d'origen/destí
<code>byte[] getData()</code>	Contingut del missatge
<code>int getLength()</code>	Longitud de les dades
<code>int getPort()</code>	Port d'origen/destí

7.2. Classe DatagramSocket

Constructor	Missió
<code>DatagramSocket()</code>	El sistema tria un port lliure
<code>DatagramSocket(int port)</code>	S'enllaça al port indicat

Mètode	Missió
<code>void send(DatagramPacket p)</code>	Envia el paquet
<code>void receive(DatagramPacket p)</code>	Es bloqueja fins a rebre un paquet
<code>void close()</code>	Tanca el socket
<code>void setSoTimeout(int ms)</code>	Temps màxim d'espera abans de llançar <code>InterruptedIOException</code>

7.3. Esquema de funcionament

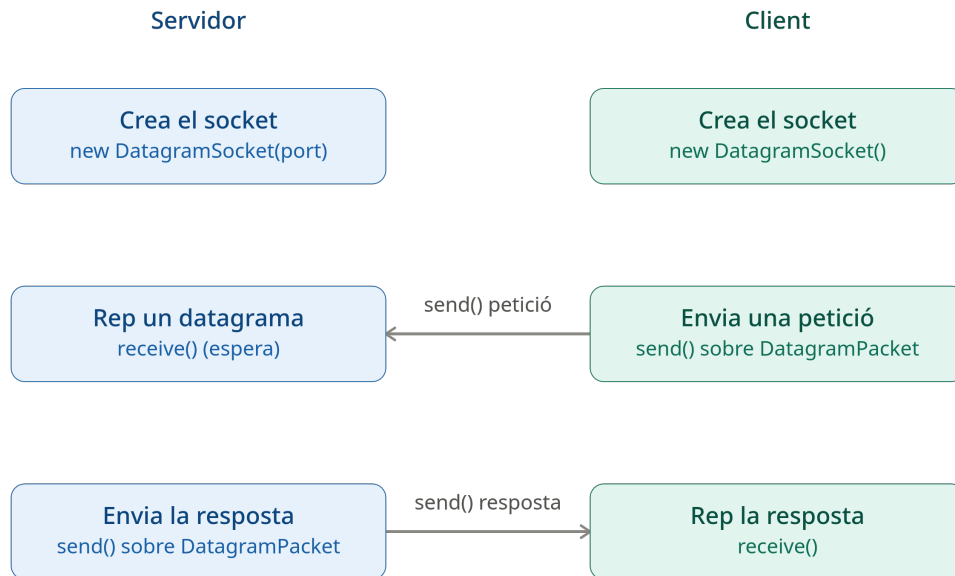


Figura 2: Esquema general de funcionament

7.4. Exemple: servidor i client UDP

Servidor (rep un missatge i respon amb el nombre de vocals que conté):

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

public class ServidorUDP {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket(5001);
        byte[] rebuts = new byte[1024];

        while (true) {
            DatagramPacket paquetRebut = new DatagramPacket(rebuts,
                ↪ rebuts.length);
            socket.receive(paquetRebut);

            String missatge = new String(paquetRebut.getData(), 0,
                ↪ paquetRebut.getLength());
            InetAddress ipOrigen = paquetRebut.getAddress();
            int portOrigen = paquetRebut.getPort();
            System.out.println("Rebut de " + ipOrigen + ":" +
                ↪ portOrigen + " -> " + missatge);

            long vocals = missatge.toLowerCase().chars()
                .filter(c -> "aeiouàèéíòóú".indexOf(c) >= 0)
        }
    }
}
```

```

        .count();
        byte[] enviats = (String.valueOf(vocals) + "
        ↪ vocals").getBytes();
        DatagramPacket paquetEnviar =
            new DatagramPacket(enviats, enviats.length,
            ↪ ipOrigen, portOrigen);
        socket.send(paquetEnviar);

        if (missatge.trim().equals("fi")) break;
    }
    socket.close();
}
}

```

Client:

```

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

public class ClientUDP {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket();
        InetAddress ipServidor = InetAddress.getByName("localhost");
        int port = 5001;

        BufferedReader teclat = new BufferedReader(new
            ↪ InputStreamReader(System.in));
        System.out.print("Missatge a enviar: ");
        String text = teclat.readLine();
        byte[] enviats = text.getBytes();

        DatagramPacket enviament =
            new DatagramPacket(enviats, enviats.length,
            ↪ ipServidor, port);
        socket.send(enviament);

        byte[] rebuts = new byte[1024];
        DatagramPacket recepcio = new DatagramPacket(rebuts,
            ↪ rebuts.length);
        socket.receive(recepcio);

        String resposta = new String(recepcio.getData(), 0,
            ↪ recepcio.getLength());
        System.out.println("Resposta del servidor: " + resposta);

        socket.close();
    }
}

```

NOTA

Com que UDP no garanteix l'entrega, en aplicacions reals sol combinar-se amb `setSoTimeout()` per detectar pèrdua de paquets.

8. MulticastSocket

La classe **MulticastSocket** permet enviar un mateix paquet a **múltiples destinataris** simultàniament, formant un **grup multicast**: un conjunt d'adreces IP (classe D, de 224.0.0.0 a 239.255.255.255) que comparteixen el mateix número de port.

Mètode	Missió
<code>joinGroup(InetAddress grup)</code>	El socket s'uneix al grup
<code>leaveGroup(InetAddress grup)</code>	El socket abandona el grup
<code>send(DatagramPacket p)</code>	Envia el datagrama a tots els membres
<code>receive(DatagramPacket p)</code>	Rep el datagrama del grup

Esquema d'un servidor multicast

```
MulticastSocket ms = new MulticastSocket();
int port = 9000;
InetAddress grup = InetAddress.getByName("230.0.0.5");

String msg = "Avís general per al grup";
DatagramPacket paquet = new DatagramPacket(
    msg.getBytes(), msg.length(), grup, port);
ms.send(paquet);
ms.close();
```

Esquema d'un client multicast

```
MulticastSocket ms = new MulticastSocket(9000);
InetAddress grup = InetAddress.getByName("230.0.0.5");
ms.joinGroup(grup);

byte[] buf = new byte[1000];
DatagramPacket rebut = new DatagramPacket(buf, buf.length);
ms.receive(rebut);

ms.leaveGroup(grup);
ms.close();
```

El multicast és molt útil per a xats senzills, notificacions a diversos clients o transmissió d'àudio/vídeo en temps real a diversos receptors alhora.

9. Enviament d'objectes a través de sockets

Fins ara hem intercanviat cadenes de text, però els streams també admeten **objectes**. Per fer-ho, les classes involucrades han d'implementar la interfície **Serializable**.

```
import java.io.Serializable;

public class Producte implements Serializable {
    private String codi;
    private double preu;

    public Producte() {}
    public Producte(String codi, double preu) {
        this.codi = codi;
        this.preu = preu;
    }
    public String getCodi() { return codi; }
    public void setCodi(String codi) { this.codi = codi; }
    public double getPreu() { return preu; }
    public void setPreu(double preu) { this.preu = preu; }
}
```

9.1. Objectes en sockets TCP

S'utilitzen **ObjectOutputStream** i **ObjectInputStream**, amb els mètodes `writeObject()` i `readObject()`:

```
// Sortida (escriure un objecte)
ObjectOutputStream outObjecte = new
    ↳ ObjectOutputStream(socket.getOutputStream());
Producte prod = new Producte("REF-118", 24.90);
outObjecte.writeObject(prod);

// Entrada (llegir un objecte)
ObjectInputStream inObjecte = new
    ↳ ObjectInputStream(socket.getInputStream());
Producte rebut = (Producte) inObjecte.readObject();
```

9.2. Objectes en sockets UDP

Com que UDP treballa amb arrays de bytes, cal serialitzar l'objecte manualment amb `ByteArrayOutputStream/ByteArrayInputStream`:

```
// Convertir objecte a bytes
Producte prod = new Producte("REF-204", 9.50);
ByteArrayOutputStream bs = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bs);
out.writeObject(prod);
out.close();
byte[] bytes = bs.toByteArray();
// ... enviar 'bytes' dins un DatagramPacket

// Convertir bytes rebuts a objecte
ByteArrayInputStream bais = new ByteArrayInputStream(rebuts);
ObjectInputStream in = new ObjectInputStream(bais);
Producte obj = (Producte) in.readObject();
in.close();
```

10. Atenció a múltiples clients amb fils

Un servidor TCP “simple” com els vistos fins ara només pot atendre **un client alhora**. Per atendre'n diversos simultàniament cal combinar sockets amb **fils (threads)**:

main() del Servidor

<pre>while (true) { Socket cli = servidor.accept(); ↪ un client FilClient f = new FilClient(cli); h.start(); ↪ independent }</pre>	<pre> ← es bloqueja fins que arriba ← cada client en un fil</pre>
--	--

D'aquesta manera, mentre un fil atén un client concret (llegint/escrivint al seu socket), el fil principal del servidor pot continuar acceptant noves connexions.

10.1. Exemple: servidor multiclient (retorna la longitud de cada línia)

Fil d'atenció:

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

public class FilClient extends Thread {
    private BufferedReader entrada;
    private PrintWriter sortida;
    private Socket socket;

    public FilClient(Socket s) throws IOException {
        socket = s;
        entrada = new BufferedReader(new
            ↪ InputStreamReader(socket.getInputStream()));
        sortida = new PrintWriter(socket.getOutputStream(), true);
    }

    @Override
    public void run() {
        String cadena;
        try {
            while (!(cadena =
                ↪ entrada.readLine()).trim().equalsIgnoreCase("adeu"))
                ↪ {
                    System.out.println("Client " + socket + " -> " +
                        ↪ cadena);
                    sortida.println("Longitud: " + cadena.length());
                }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                entrada.close();
                sortida.close();
                socket.close();
            } catch (IOException e) { /* ignorem */ }
        }
    }
}
```

Servidor principal:

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

public class ServidorMultiClient {
    public static void main(String[] args) throws IOException {
        ServerSocket servidor = new ServerSocket(8000);
        System.out.println("Servidor iniciat...");

        while (true) {
```

```

        Socket client = servidor.accept(); // espera un client
        System.out.println("Nou client connectat: " + client);
        FilClient fil = new FilClient(client);
        fil.start(); // atendre'l en el seu propi fil
    }
}

```

Amb aquest esquema, cada client que es connecti serà atès en paral·lel per un fil diferent, sense bloquejar la resta.

10.2. Consideracions importants

- Si diversos fils accedeixen a **recursos compartits** (per exemple, un array de sockets connectats, un fitxer, una connexió a base de dades), cal sincronitzar l'accés amb `synchronized` per evitar condicions de carrera.
- És habitual mantenir una **estructura compartida** (llista o array de sockets) per poder retransmetre missatges a tots els clients connectats (com en una aplicació de xat).
- Cal controlar el **nombre màxim de connexions** simultànies si el servidor ha de limitar la càrrega.

10.3. Cas d'ús típic: servidor de xat

Un servidor de xat és l'exemple paradigmàtic de servidor multiclient:

1. El servidor manté una **llista de sockets** dels clients connectats.
2. Quan arriba un client nou, s'afegeix a la llista i es llança el seu fil.
3. Cada fil llegeix els missatges del seu client i els **retransmet a la resta de sockets** de la llista (broadcast).
4. Quan un client es desconnecta, s'elimina de la llista i es tanca el seu socket.

```

private void difondreATothom(String text, List<Socket> connectats,
    ↪ Socket remitent) {
    Iterator<Socket> it = connectats.iterator();
    while (it.hasNext()) {
        Socket s = it.next();
        if (s == remitent) continue; // no cal reenviar-li'l a qui
        ↪ l'ha escrit
        try {
            DataOutputStream sortida = new
                ↪ DataOutputStream(s.getOutputStream());
            sortida.writeUTF(text);
        } catch (IOException e) {
            it.remove(); // el client ja no està disponible: el
            ↪ traiem de la llista
        }
    }
}

```

11. Models de comunicació en architectures distribuïdes

A més del clàssic client-servidor, hi ha altres models de comunicació habituals:

Model	Descripció	Exemple
Client-servidor (unicast)	Un client es comunica amb un únic servidor	Aplicació web, consulta a BD
Peer-to-peer (P2P)	Els processos actuen alhora com a client i servidor, sense un node central	BitTorrent, xarxes de compartició de fitxers
Multicast	Un emissor envia a un grup de receptors interessats	Streaming a diversos clients, notificacions
Broadcast	Un emissor envia a tots els nodes de la xarxa	Descobriments de dispositius (DHCP)
Publicació-subscripció (pub/sub)	Els emissors publiquen missatges en "temes" i els receptors se subscriuen als que els interessen	Sistemes de missatgeria (MQTT, JMS)
RPC / RMI	El client invoca mètodes que s'executen remotament, com si fossin locals	Java RMI, gRPC

En les aplicacions client-servidor amb sockets que hem vist:

- El model **TCP** és adequat quan cal **fiabilitat** i **ordre** en l'entrega (transferència d'arxius, protocols d'aplicació, bases de dades).
- El model **UDP** és adequat quan interessa més la **rapidesa** que la fiabilitat total (àudio/vídeo en temps real, jocs en xarxa).
- El **multicast** s'utilitza quan un mateix contingut ha d'arribar a diversos receptors sense haver-lo d'enviar individualment a cadascun.

12. Depuració i bones pràctiques

A l'hora de desenvolupar i depurar aplicacions client-servidor amb sockets:

1. **Ordre d'execució:** sempre cal engegar primer el **servidor** (que queda esperant amb `accept()` o `receive()`) i després el **client**.
2. **Gestió d'excepcions:** totes les operacions de xarxa poden llançar `IOException` (o subclasses com `UnknownHostException`, `SocketException`, `MalformedURLException`). Cal capturar-les i tractar-les adequadament (registre, missatges a l'usuari, tancament net de recursos).
3. **Tancament de recursos:** sempre s'han de tancar primer els streams i després els sockets, preferiblement dins un bloc `finally` o utilitzant *try-with-resources*.
4. **Ports ocupats:** si un `ServerSocket` no es pot enllaçar a un port, sol ser perquè ja hi ha un altre procés escoltant-hi (`BindException`). Cal comprovar-ho amb eines com `netstat` o canviar de port.
5. **Proves en local vs. xarxa:** es pot provar primer amb `"localhost"` (o `127.0.0.1`) abans de provar-ho entre màquines diferents de la xarxa, indicant la IP real del servidor al client.
6. **Registre (logging):** és molt útil mostrar per consola (o en un `JTextArea` si hi ha interfície gràfica) les connexions, desconnexions i missatges intercanviats, per poder seguir el flux de l'aplicació.

7. **Documentació:** cal documentar el protocol propi de l'aplicació (format dels missatges, cadenes de control com "*" per finalitzar, codis d'error, etc.), ja que sockets i datagrames no imposen cap format: el dissenya el programador.
8. **Tallafocs i xarxa:** si les proves es fan entre màquines diferents, cal assegurar-se que el port utilitzat no estigui bloquejat pel tallafocs.

Índex de figures

1	Esquema general de funcionament	7
2	Esquema general de funcionament	12

Versions d'aquest document

- [HTML](#)
- [PDF](#)
- [ODT](#)
- [MD](#)

[Domini Públic \(CC0\)](#)