
RA4. Generació de serveis en xarxa

Índex

1. Introducció als serveis i protocols de xarxa	1
1.1 Capes TCP/IP i protocols d'aplicació	1
1.2 Per a què utilitzar protocols estàndard?	2
1.3 Serveis web	2
2. Apache Commons Net: la llibreria per als clients	2
2.1 Còdis de resposta	3
3. Client FTP	3
3.1 Connexió bàsica	3
3.2 Login i navegació	4
3.3 Pujar un fitxer	5
3.4 Baixar un fitxer	5
3.5 Registre de la conversa amb el servidor	5
3.6 Creació d'un client FTP complet	5
4. Client Telnet	6
4.1 Instal·lació d'un servidor Telnet (per fer proves)	6
4.2 La classe TelnetClient	6
4.3 Lectura i escriptura de dades	7
4.4 Client Telnet interactiu amb fil de lectura	7
5. Client SMTP (enviament de correu)	9
5.1 El protocol SMTP en cru amb Telnet	9
5.2 La classe SMTPClient	9
5.3 Capçalera correcta amb SimpleSMTPHeader	10
5.4 Autenticació SMTP segura (STARTTLS)	11
6. Client POP3 (recepció de correu)	12
6.1 Classes de Commons Net	12
6.2 Recuperar la capçalera o el contingut d'un missatge	12
7. Programació d'un servidor propi amb sockets	13
7.1 Disseny de la comunicació	13
7.2 La classe d'estat DirectoriInfo	14
7.3 El servidor	15
7.4 El fil d'atenció al client	16
7.5 El client	17
8. Comunicació simultània amb diversos clients	18
8.1 Alternativa: pools de fils	19
9. Monitoratge i verificació de la disponibilitat del servei	19
9.1 Clients de comunicacions genèrics	19
9.2 Comprovació programàtica	20
9.3 Registres (logs)	20
9.4 Gestió d'excepcions com a indicador de disponibilitat	20
10. Bones pràctiques i depuració	21

Cicle: Desenvolupament d'Aplicacions Multiplataforma (DAM)

Mòdul: 0490. Programació de serveis i processos

1. Introducció als serveis i protocols de xarxa

Un **servei** és un programa auxiliar que gestiona un conjunt de recursos i n'ofereix la funcionalitat a altres usuaris o aplicacions mitjançant un conjunt d'operacions concretes. Per exemple, un servei d'impressió permet gestionar i compartir una impressora a la xarxa; un servei de fitxers ofereix operacions de lectura, escriptura i esborrat.

Tots els serveis d'Internet segueixen el model **client-servidor**: el servidor espera peticions en un port conegut i els clients s'hi connecten per sol·licitar operacions.

1.1 Capes TCP/IP i protocols d'aplicació

El model TCP/IP es divideix en quatre capes. La capa d'**aplicació** és la que implementa els protocols d'alt nivell que farem servir:

Capa	Protocols habituals
Aplicació	SMTP, Telnet, FTP, HTTP, NFS, SNMP, DNS
Transport	TCP, UDP
Internet	IP
Interfície de xarxa	Protocols de subxarxa

Alguns dels protocols estàndard més utilitzats:

- **Telnet**: emulació de terminal remot. Permet accedir a una màquina remota i treballar-hi com si hi fóssim asseguts al davant. El seu principal problema és la **seguretat**: l'usuari i la contrasenya viatgen en text pla. Port TCP 23.
- **SMTP** (*Simple Mail Transfer Protocol*): protocol per a la transferència de correu electrònic entre servidors. Port 25.
- **FTP** (*File Transfer Protocol*): protocol fiable orientat a connexió per transferir fitxers. Utilitza dues connexions TCP separades: una de **control** (port 21) i una de **dades** (port 20).
- **TFTP** (*Trivial FTP*): versió simplificada d'FTP sense autenticació, sobre UDP.
- **HTTP**: protocol de petició-resposta utilitzat pels navegadors web.
- **NFS**: sistema de fitxers en xarxa, permet accedir a fitxers remots com si fossin locals.
- **SNMP**: protocol per intercanviar informació de gestió entre dispositius de xarxa.
- **DNS**: sistema distribuït que resol noms d'ordinador en adreces IP.

1.2 Per a què utilitzar protocols estàndard?

- **Interoperabilitat:** qualsevol client que implementi l'estàndard es pot comunicar amb el nostre servei, i viceversa.
- **Documentació i suport:** estan definits en RFC públics (per exemple, FTP a la RFC 959, SMTP a la RFC 2821, Telnet a la RFC 854), amb comunitats i eines de diagnòstic ja existents.
- **Reutilització:** hi ha llibreries madures i provades (com Apache Commons Net) que ja implementen el protocol, evitant haver de programar-lo des de zero.
- **Manteniment i evolució:** les millores de seguretat o rendiment del protocol beneficien totes les aplicacions que el fan servir.
- **Menys errors:** es redueix el risc de bugs de baix nivell en la implementació del protocol.

1.3 Serveis web

A banda dels protocols “clàssics” (FTP, Telnet, SMTP...), avui en dia molts serveis en xarxa s'exposen com a **serveis web**, és a dir, aplicacions accessibles mitjançant HTTP/HTTPS que intercanvien dades normalment en format XML o JSON (per exemple, API REST). Encara que en aquesta unitat ens centrarem en protocols estàndard “tradicionals” amb Apache Commons Net, val la pena tenir present que:

- Els serveis web reutilitzen el mateix model client-servidor i els mateixos conceptes de ports, sockets i connexions que veurem aquí.
- Java ofereix llibreries (per exemple `HttpClient`, `URLConnection`, o frameworks com Spring) per consumir i publicar serveis web.
- Els criteris d'eficiència, disponibilitat i gestió de clients simultanis que aprendrem amb el nostre servidor de fitxers casolà (apartat 7) són exactament els mateixos que calen per dissenyar un servei web robust.

2. Apache Commons Net: la llibreria per als clients

Apache Commons Net™ és una llibreria Java que implementa el costat client de molts protocols bàsics d'Internet: FTP, SMTP, Telnet, TFTP, POP3, entre d'altres.

- Es descarrega des de http://commons.apache.org/proper/commons-net/download_net.cgi.
- S'ha d'afegir el `.jar` (per exemple `commons-net-3.2.jar`) al classpath del projecte.

Totes les classes client de la llibreria deriven de **SocketClient**, que proporciona la funcionalitat comuna:

Mètode	Missió
<code>connect(String host)</code>	Obre la connexió amb el servidor
<code>disconnect()</code>	Tanca la connexió i restaura els paràmetres
<code>isConnected()</code>	Indica si hi ha una connexió activa
<code>getReplyCode()</code>	Codi numèric de l'última resposta
<code>getReplyString()</code>	Text complet de l'última resposta

El patró de treball és sempre el mateix:

1. Crear l'objecte client (FTPClient, TelnetClient, SMTPClient, POP3Client...).
2. Connectar-se amb connect().
3. Comprovar el codi de resposta.
4. Fer les operacions pròpies del protocol.
5. Desconnectar amb disconnect()/logout().

2.1 Codis de resposta

Molts d'aquests protocols (FTP, SMTP) responen amb codis numèrics de tres xifres, on el primer dígit indica el tipus de resposta:

Primer dígit	Significat
1yz	Resposta preliminar positiva
2yz	Terminació positiva (èxit)
3yz	Intermèdia positiva, cal més informació
4yz	Terminació negativa transitòria
5yz	Terminació negativa permanent

Les classes FTPReply i SMTPReply ofereixen constants i el mètode estàtic isPositiveCompletion(int codi) per comprovar-ho de manera còmoda.

3. Client FTP

3.1 Connexió bàsica

La classe **FTPClient** encapsula tota la funcionalitat per emmagatzemar i recuperar fitxers d'un servidor FTP.

```
import java.io.IOException;
import java.net.SocketException;
import org.apache.commons.net.ftp.*;

public class ClientFTP1 {
    public static void main(String[] args) throws SocketException,
        ↪ IOException {
        FTPClient client = new FTPClient();
        String host = "ftp.rediris.es";
        client.connect(host);

        int codi = client.getReplyCode();
        if (!FTPReply.isPositiveCompletion(codi)) {
            client.disconnect();
            System.out.println("Connexió rebutjada: " + codi);
            System.exit(0);
        }

        client.disconnect();
        System.out.println("Connexió finalitzada.");
    }
}
```

```
}  
}
```

3.2 Login i navegació

Amb **login(usuari, clau)** ens autèntiquem (l'accés **anònim** es fa amb l'usuari **anonymous**). Un cop dins, podem llistar fitxers amb **listFiles()**, que retorna un array de **FTPFile**:

```
FTPClient client = new FTPClient();  
client.connect("ftp.rediris.es");  
boolean sessioIniciada = client.login("anonymous", "anonymous");  
  
if (sessioIniciada) {  
    System.out.println("Directori actual: " +  
        ↪ client.printWorkingDirectory());  
    FTPFile[] fitxers = client.listFiles();  
    for (FTPFile f : fitxers) {  
        String tipus = f.isDirectory() ? "Directori" : (f.isFile() ?  
            ↪ "Fitxer" : "Enllaç");  
        System.out.println(f.getName() + " => " + tipus);  
    }  
    client.logout();  
}  
client.disconnect();
```

Mètodes habituals de **FTPClient** (tots poden llançar **IOException**):

Mètode	Missió
login(user, pass)	Inicia sessió
logout()	Tanca sessió
changeWorkingDirectory(path)	Canvia de directori
changeToParentDirectory()	Puja al directori pare
makeDirectory(path)	Crea un directori
removeDirectory(path)	Elimina un directori (buit)
setFileType(int tipus)	FTP.BINARY_FILE_TYPE, FTP.ASCII_FILE_TYPE...
storeFile(nom, InputStream)	Puja un fitxer
retrieveFile(nom, OutputStream)	Baixa un fitxer
deleteFile(path)	Elimina un fitxer
rename(antic, nou)	Reanomena un fitxer

AVÍS

Cal fixar sempre el tipus de transferència amb **setFileType(FTP.BINARY_FILE_TYPE)** abans de pujar o baixar fitxers binaris (imatges, documents...), o es poden corrompre.

3.3 Pujar un fitxer

```
client.changeWorkingDirectory("/NUEVODIREC");
client.setFileType(FTP.BINARY_FILE_TYPE);

BufferedInputStream in = new BufferedInputStream(
    new FileInputStream("D:\\CAPIT4\\TEXT01.txt"));
client.storeFile("TEXT01.txt", in);
in.close();
```

3.4 Baixar un fitxer

```
client.changeWorkingDirectory("/NUEVODIREC/NUEVO");
BufferedOutputStream out = new BufferedOutputStream(
    new FileOutputStream("D:\\CAPIT4\\TEXT02nuevo.txt"));

if (client.retrieveFile("TEXT02.txt", out))
    System.out.println("Recuperat correctament...");
out.close();
```

3.5 Registre de la conversa amb el servidor

Per veure els comandaments i respostes que s'intercanvien client i servidor, s'utilitza `addProtocolCommandListener()` (heretat de `SocketClient`):

```
client.addProtocolCommandListener(
    new PrintCommandListener(new PrintWriter(System.out)));
client.connect(servidor);
```

3.6 Creació d'un client FTP complet

Un client FTP amb interfície gràfica es pot estructurar amb un `JFrame` que mostri un `JList` amb el contingut del directori actual i botons per pujar, baixar, crear i eliminar fitxers/carpets. Els punts clau del disseny són:

- Es guarda l'estat de navegació en variables com `carpetaActual` (directori de treball actual) i `elementSeleccionat` (últim fitxer o carpeta seleccionat).
- Cada vegada que es canvia de directori, es torna a cridar `listFiles()` i s'omple de nou el `JList` amb un `DefaultListModel`.
- Per distingir un directori d'un fitxer a la llista, es prefixa el nom amb `(DIR)`.
- Fer clic sobre el primer element de la llista (que representa el directori actual) permet pujar al directori pare amb `changeToParentDirectory()`.
- Per pujar o baixar fitxers s'utilitza `JFileChooser`, que ofereix una finestra estàndard de selecció de fitxers/directoris.

```

private static void actualitzarLlista(FTPFile[] fitxers, String
↪ directoriActual) {
    DefaultListModel model = new DefaultListModel();
    model.addElement(directoriActual);          // primer element =
    ↪ directori actual
    for (FTPFile f : fitxers) {
        if (!f.getName().equals(".") && !f.getName().equals("..")) {
            String nom = f.getName();
            if (f.isDirectory()) nom = "(DIR) " + nom;
            model.addElement(nom);
        }
    }
    llistaVisual.setModel(model);
}

```

4. Client Telnet

4.1 Instal·lació d'un servidor Telnet (per fer proves)

A Windows es pot activar des de *Panell de control* → *Programes* → *Activar o desactivar característiques de Windows*, marcant **Servidor Telnet** i **Client Telnet**. Cal donar accés al servei als usuaris afegint-los al grup TelnetClients:

```

net localgroup TelnetClients /add usuari2
net localgroup TelnetClients

```

A Linux es pot instal·lar un dimoni telnet (per exemple `telnetd`) o, més recomanable, provar-ho contra un servei propi que escolti en un port TCP (com el servidor de fitxers de l'apartat 7).

4.2 La classe TelnetClient

TelnetClient (que estén **SocketClient**) implementa el terminal virtual de xarxa (NVT) segons la RFC 854.

```

TelnetClient telnet = new TelnetClient("xterm"); // o new
↪ TelnetClient() -> VT100
telnet.connect("localhost");

OutputStream out = telnet.getOutputStream();
InputStream in = telnet.getInputStream();

```

A diferència d'FTP i SMTP, Telnet **no té una estructura de comandaments/respostes fixa**: el que rebem i enviem depèn del programa que hi ha a l'altra banda (un shell, un login...), per això cal fer **lectures i escriptures manuals** dels streams.

4.3 Lectura i escriptura de dades

```
static void lecturaDatos() throws IOException {
    byte[] buff = new byte[1024];
    int nbytes = in.read(buff);
    System.out.print(new String(buff, 0, nbytes));
}

static void envioDatos(String cad) {
    out.println(cad);
    out.flush();
}
```

Seqüència típica per fer login i executar un comandament:

```
lecturaDatos();           // missatge de benvinguda + "login:"
envioDatos(login);
lecturaDatos();           // "password:"
envioDatos(password);
lecturaDatos();           // presentació + prompt
envioDatos("DIR");
lecturaDatos();           // sortida del comandament
telnet.disconnect();
```

4.4 Client Telnet interactiu amb fil de lectura

En un client real, l'usuari escriu per teclat mentre el servidor pot enviar dades en qualsevol moment. La solució és llançar un **fil (Thread)** dedicat a llegir contínuament el que arriba del servidor. En comptes de limitar-se a imprimir cada tros rebut, aquest fil pot acumular-lo en un `StringBuilder` compartit i marcar quan hi ha novetats, cosa que permetria, per exemple, refrescar una àrea de text en una interfície gràfica:

```
class MonitorSortida extends Thread {
    private final TelnetClient connexio;
    private final StringBuilder acumulat = new StringBuilder();

    public MonitorSortida(TelnetClient connexio) {
        this.connexio = connexio;
    }

    public synchronized String contingutActual() {
        return acumulat.toString();
    }

    @Override
    public void run() {
        InputStream entrada = connexio.getInputStream();
        byte[] buff = new byte[512];
        int nbytes;
        try {
```

```

        while ((nbytes = entrada.read(buff)) > 0) {
            String tros = new String(buff, 0, nbytes);
            System.out.print(tros);
            synchronized (this) {
                acumulat.append(tros);
            }
        }
    } catch (IOException e) {
        System.err.println("El servidor ha tancat la connexió: " +
            ↪ e.getMessage());
    }
}

```

```

TelnetClient telnet = new TelnetClient("xterm");
telnet.connect("localhost");

MonitorSortida monitor = new MonitorSortida(telnet);
monitor.start();

OutputStream sortida = telnet.getOutputStream();
BufferedReader teclat = new BufferedReader(new
    ↪ InputStreamReader(System.in));
String ordre;

while ((ordre = teclat.readLine()) != null &&
    ↪ !ordre.equalsIgnoreCase("sortir")) {
    sortida.write((ordre + "\n").getBytes());
    sortida.flush();
}
telnet.disconnect();

```

Aquest patró de “fil de lectura + bucle d'escriptura al fil principal” el reutilitzarem després per al nostre propi servidor.

5. Client SMTP (enviament de correu)

5.1 El protocol SMTP en cru amb Telnet

Abans de programar-ho, val la pena entendre el protocol enviant un correu manualment via Telnet al port 25:

Ordre	Missió
HELO/EHLO	Obre la sessió
MAIL FROM: origen	Indica el remitent
RCPT TO: destí	Indica el destinatari
DATA	Comença el cos del missatge (acaba amb una línia amb només un punt .)
QUIT	Tanca la sessió

```
HELO
MAIL FROM: jo@brianda.net
RCPT TO: jo@brianda.net
DATA
from: jo@brianda.net
to: jo@brianda.net
subject: proves

Hola, això és una prova.
Adeu.
.
QUIT
```

5.2 La classe SMTPClient

SMTPClient (estén SMTP) encapsula tota aquesta lògica:

```
import org.apache.commons.net.smtp.*;

SMTPClient client = new SMTPClient();
client.connect("localhost");           // port 25 per defecte

int respuesta = client.getReplyCode();
if (!SMTPReply.isPositiveCompletion(respuesta)) {
    client.disconnect();
    System.exit(1);
}
```

Mètodes principals:

Mètode	Missió
login() / login(host)	Envia HELO
setSender(address)	Envia MAIL FROM
addRecipient(address)	Envia RCPT TO
sendMessageData()	Envia DATA i retorna un <code>Writer</code> per escriure la capçalera i el cos

<code>completePendingCommand()</code>	Finalitza la transacció i comprova l'èxit
<code>sendSimpleMessage(remitent, destinatari, missatge)</code>	Enviament ràpid d'un missatge senzill
<code>logout()</code>	Envia QUIT
<code>verify(username)</code>	Comprova si un usuari/adreça és vàlida (comandament VRFY)

Enviament senzill:

```
client.login();
client.sendSimpleMessage("jo@localhost.es", "algu@exemple.com",
    "Hola.\nEnviant salutacions.\nAdeu.");
client.logout();
```

5.3 Capçalera correcta amb SimpleSMTPHeader

Un missatge enviat sense capçalera adequada pot arribar com a spam o ser rebutjat. La classe **SimpleSMTPHeader** construeix una capçalera mínima vàlida:

```
SimpleSMTPHeader cabecera =
    new SimpleSMTPHeader("jo@localhost.es", "desti1@exemple.com",
        ↪ "Assumpte");
cabecera.addCC("desti2@exemple.com");

client.setSender("jo@localhost.es");
client.addRecipient("desti1@exemple.com");
client.addRecipient("desti2@exemple.com");

Writer writer = client.sendMessageData();
if (writer == null) { System.out.println("Fallada en enviar DATA.");
    ↪ System.exit(1); }
writer.write(cabecera.toString());
writer.write("Hola.\nEnviant salutacions.\nAdeu.");
writer.close();

if (!client.completePendingCommand()) {
    System.out.println("Fallada en finalitzar la transacció.");
}
client.logout();
```

5.4 Autenticació SMTP segura (STARTTLS)

Per enviar correu amb un proveïdor real (per exemple Gmail: smtp.gmail.com, port 587 amb STARTTLS, o 465 amb SSL directe) calen classes amb suport de seguretat:

- **SMTPSClient**: SMTP sobre SSL/TLS. Constructor SMTPSClient() (mode explícit, negocia STARTTLS després de connectar) o SMTPSClient(true) (mode implícit, SSL/TLS des del principi).
- **AuthenticatingSMTPClient**: afegeix el mètode auth() per autenticar-se.

```
AuthenticatingSMTPClient client = new AuthenticatingSMTPClient();
String server = "smtp.gmail.com";
int puerto = 587;

KeyManagerFactory kmf = KeyManagerFactory.getInstance(
    KeyManagerFactory.getDefaultAlgorithm());
kmf.init(null, null);
KeyManager km = kmf.getKeyManagers()[0];

client.connect(server, puerto);
client.setKeyManager(km);

client.ehlo(server);

if (client.execTLS()) {
    if (client.auth(AuthenticatingSMTPClient.AUTH_METHOD.PLAIN,
        "usuari@gmail.com", "clau")) {

        SimpleSMTPHeader cabecera = new SimpleSMTPHeader(
            "usuari@gmail.com", "desti@exemple.com", "Prova amb
            ↪ Gmail");
        client.setSender("usuari@gmail.com");
        client.addRecipient("desti@exemple.com");

        Writer writer = client.sendMessageData();
        writer.write(cabecera.toString());
        writer.write("Hola.\nUsant Gmail.\nAdeu.");
        writer.close();

        client.completePendingCommand();
    }
}
client.disconnect();
```

Mètodes d'autenticació disponibles a AUTH_METHOD: CRAM_MD5 (contrasenya xifrada), LOGIN/PLAIN (text pla sobre canal ja xifrat per TLS) i XOAUTH (basat en OAuth).

6. Client POP3 (recepció de correu)

POP3 (*Post Office Protocol*) permet accedir als missatges emmagatzemats en un servidor de correu. Port 110 (o 995 amb SSL).

Comandament	Missió
USER login	Nom d'usuari
PASS contrasenya	Contrasenya
STAT	Nombre de missatges i mida total
LIST	Llistat de missatges (número - mida)
RETR n	Recupera el missatge n
DELE n	Esborra el missatge n
TOP n x	Mostra les x primeres línies del missatge n (x=0 retorna només la capçalera)
QUIT	Tanca la connexió

6.1 Classes de Commons Net

- **POP3Client**: implementació del client POP3 (RFC 1939).
- **POP3SClient**: versió amb suport SSL (mode implícit/explicit igual que SMTPSClient).
- **POP3MessageInfo**: informació sobre un missatge (identifier, number, size).

```
import org.apache.commons.net.pop3.*;

POP3SClient pop3 = new POP3SClient();           // mode explícit
pop3.connect("localhost", 110);

if (pop3.login("usu1", "usu1")) {
    POP3MessageInfo[] missatges = pop3.listMessages();
    System.out.println("Nombre de missatges: " + missatges.length);

    pop3.logout();
}
pop3.disconnect();
```

6.2 Recuperar la capçalera o el contingut d'un missatge

```
BufferedReader reader = (BufferedReader)
    pop3.retrieveMessageTop(missatges[0].number, 0); // 0 => només
    ⇨ capçalera

String linia;
while ((linia = reader.readLine()) != null) {
    System.out.println(linia);
}
reader.close();
```

Per recuperar el missatge sencer (i que desaparegui de la bústia):

```
BufferedReader reader = (BufferedReader)
    pop3.retrieveMessage(missatges[0].number);
```

NOTA

Per connectar amb un servidor real com Gmail (pop.gmail.com, port 995, SSL):
POP3SClient pop3 = new POP3SClient(true);

7. Programació d'un servidor propi amb sockets

Fins ara hem creat **clients** de protocols ja existents. Ara construirem el nostre propi **servei en xarxa**: un servidor de fitxers senzill que permet als clients llistar, pujar i baixar fitxers d'un directori.

7.1 Disseny de la comunicació

La comunicació client-servidor es farà **per objectes serialitzats** sobre `ObjectInputStream/ObjectOutputStream` en comptes de text pla. En lloc de definir una classe diferent per a cada tipus de missatge, farem servir una **única classe embolcall** (`MissatgeProtocol`) amb un camp tipus (un enum) i una càrrega útil (payload) genèrica; és una alternativa més compacta al clàssic “una classe per missatge + instanceof”:

```
SERVIDOR  --- MissatgeProtocol(ARBRE)    ---> CLIENT  (en connectar)
SERVIDOR  <--- MissatgeProtocol(BAIXAR)  ---  CLIENT  (vol baixar un
↳ fitxer)
SERVIDOR  --- MissatgeProtocol(DADES)    ---> CLIENT
SERVIDOR  <--- MissatgeProtocol(PUJAR)   ---  CLIENT  (vol pujar un
↳ fitxer)
SERVIDOR  --- MissatgeProtocol(ARBRE)    ---> CLIENT  (arbre
↳ actualitzat)
```

```
import java.io.Serializable;

public class MissatgeProtocol implements Serializable {
    public enum Tipus { ARBRE, BAIXAR, DADES, PUJAR }

    private final Tipus tipus;
    private final Object payload;

    public MissatgeProtocol(Tipus tipus, Object payload) {
        this.tipus = tipus;
        this.payload = payload;
    }

    public Tipus getTipus() { return tipus; }
    public Object getPayload() { return payload; }
}
```

7.2 La classe d'estat DirectoriInfo

Aquesta classe representa, de manera recursiva, l'arbre d'un directori (nom, ruta absoluta, si és una carpeta i els seus fills):

```
import java.io.File;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class DirectoriInfo implements Serializable {
    private final String nom;
    private final String rutaAbsoluta;
    private final boolean esDirectori;
    private final List<DirectoriInfo> fills = new ArrayList<>();

    public DirectoriInfo(File origen) {
        this.nom = origen.getName();
        this.rutaAbsoluta = origen.getAbsolutePath();
        this.esDirectori = origen.isDirectory();

        if (esDirectori) {
            File[] continguts = origen.listFiles();
            if (continguts != null) {
                for (File element : continguts) {
                    fills.add(new DirectoriInfo(element));
                }
            }
        }
    }

    public String getNom() { return nom; }
    public String getRutaAbsoluta() { return rutaAbsoluta; }
    public boolean esDirectori() { return esDirectori; }
    public List<DirectoriInfo> getFills() { return fills; }
    public int quantitatFills() { return fills.size(); }
}
```

Els altres dos tipus de càrrega útil (la petició de baixada/pujada i les dades transferides) són encara més senzills: n'hi ha prou amb una parella `nom fitxer / directori` per a les peticions i un array de `byte[]` per al contingut.

7.3 El servidor

El servidor:

1. Escull el directori que oferirà (per exemple amb `JFileChooser`).
2. Obre un **ServerSocket** en un port pactat.
3. Es queda en bucle infinit esperant connexions amb `accept()`.
4. Per **cada** client acceptat, crea i llança un **fil** (Thread) dedicat, de manera que el servidor pugui seguir acceptant nous clients sense bloquejar-se.

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

public class ServidorFitxersCompartits {
    private static final int PORT = 5050;

    public static void main(String[] args) throws IOException {
        JFileChooser selector = new JFileChooser();
        selector.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        selector.setDialogTitle("Tria el directori que vols
        ⇨ compartir");

        if (selector.showOpenDialog(null) !=
        ⇨ JFileChooser.APPROVE_OPTION) {
            System.out.println("Cal seleccionar un directori.
            ⇨ Sortint...");
            return;
        }
        File arrel = selector.getSelectedFile();

        try (ServerSocket servidor = new ServerSocket(PORT)) {
            System.out.println("Servidor escoltant al port " + PORT);
            while (true) {
                Socket connexio = servidor.accept();
                System.out.println("Nou client: " +
                ⇨ connexio.getInetAddress());
                new GestorClient(connexio, arrel).start(); // un fil
                ⇨ per client
            }
        }
    }
}
```

7.4 El fil d'atenció al client

Cada instància de `GestorClient` gestiona **un** client de manera independent, dins del seu propi mètode `run()`. Per llegir i escriure el contingut dels fitxers s'utilitzen els mètodes de conveniència de `java.nio.file.Files` en lloc d'un bucle manual byte a byte:

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

public class GestorClient extends Thread {
    private final Socket connexio;
    private final File directoriArrel;
    private ObjectOutputStream sortida;
    private ObjectInputStream entrada;

    public GestorClient(Socket connexio, File directoriArrel) {
        this.connexio = connexio;
        this.directoriArrel = directoriArrel;
    }

    @Override
    public void run() {
        try {
            sortida = new
                ObjectOutputStream(connexio.getOutputStream());
            entrada = new ObjectInputStream(connexio.getInputStream());

            sortida.writeObject(new MissatgeProtocol(
                MissatgeProtocol.Tipus.ARBRE, new
                DirectoriInfo(directoriArrel)));

            while (true) {
                MissatgeProtocol missatge = (MissatgeProtocol)
                    entrada.readObject();
                switch (missatge.getTipus()) {
                    case BAIXAR -> processarBaixada((String)
                        missatge.getPayload());
                    case PUJAR -> processarPujada((Object[])
                        missatge.getPayload());
                    default -> System.out.println("Missatge inesperat:
                        " + missatge.getTipus());
                }
            }
        } catch (IOException e) {
            tancar();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    private void processarBaixada(String rutaFitxer) throws
        IOException {
```

```

        byte[] dades = Files.readAllBytes(Path.of(rutaFitxer));
        sortida.writeObject(new
            ↪ MissatgeProtocol(MissatgeProtocol.Tipus.DADES, dades));
    }

    // payload = { nomFitxer, directoriDesti, dades }
    private void processarPujada(Object[] payload) throws IOException
    ↪ {
        String nomFitxer = (String) payload[0];
        String directoriDesti = (String) payload[1];
        byte[] dades = (byte[]) payload[2];

        Path desti = Path.of(directoriDesti, nomFitxer);
        Files.write(desti, dades);

        // enviem l'arbre actualitzat del directori afectat
        sortida.writeObject(new MissatgeProtocol(
            MissatgeProtocol.Tipus.ARBRE, new DirectoriInfo(new
            ↪ File(directoriDesti))));
    }

    private void tancar() {
        try {
            entrada.close();
            sortida.close();
            connexio.close();
            System.out.println("Client desconnectat");
        } catch (IOException e) { e.printStackTrace(); }
    }
}

```

NOTA

En aquest exemple es llegeix el fitxer sencer a memòria amb `Files.readAllBytes()`. Per a fitxers grans això pot provocar `OutOfMemoryError`. En un servei real, cal **troseejar** la transferència (per exemple, en blocs de 4.000 bytes), enviant diversos objectes seguits fins a completar el fitxer.

7.5 El client

El client és bàsicament simètric al servidor: obre un Socket cap al host/port del servidor, n'obté els fluxos d'objectes i:

- En connectar, **llegeix** el `MissatgeProtocol` de tipus ARBRE inicial i n'omple una llista visual (`JList`).
- Quan l'usuari vol baixar un fitxer, **envia** un missatge BAIXAR i espera rebre'n un de tipus DADES.
- Quan l'usuari vol pujar un fitxer, **envia** un missatge PUJAR i espera rebre l'ARBRE actualitzat.

```

Socket connexio = new Socket("localhost", 5050);
ObjectOutputStream sortida = new
    ↪ ObjectOutputStream(connexio.getOutputStream());
ObjectInputStream entrada = new
    ↪ ObjectInputStream(connexio.getInputStream());

// 1) rebem l'estat inicial
MissatgeProtocol inicial = (MissatgeProtocol) entrada.readObject();
DirectorInfo arrel = (DirectorInfo) inicial.getPayload();

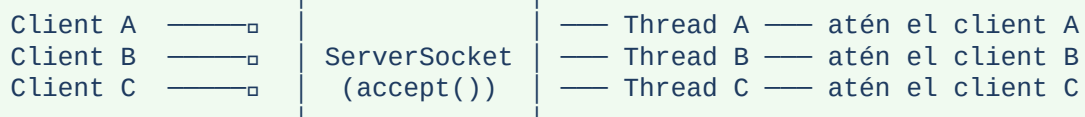
// 2) demanem un fitxer
sortida.writeObject(new MissatgeProtocol(
    MissatgeProtocol.Tipus.BAIXAR, "/Fotos/foto1.jpg"));

MissatgeProtocol resposta = (MissatgeProtocol) entrada.readObject();
if (resposta.getTipus() == MissatgeProtocol.Tipus.DADES) {
    byte[] contingut = (byte[]) resposta.getPayload();
    Files.write(Path.of("foto1.jpg"), contingut);
}

```

8. Comunicació simultània amb diversos clients

El disseny de l'apartat anterior ja incorpora el mecanisme clau per atendre múltiples clients alhora: **un fil per connexió**.



Punts clau d'aquest disseny:

- El **bucle principal** del servidor (`while(true) { accept(); ... }`) només fa una cosa: acceptar connexions i llançar fils. No fa cap operació d'E/S lenta que el pugui bloquejar.
- Cada **fil** (`GestorClient extends Thread`) té el seu propi Socket i els seus propis fluxos d'entrada/sortida, de manera que els clients no interfereixen entre si pel que fa a la connexió.
- Si diversos fils **comparteixen recursos** (per exemple, el mateix directori de fitxers al disc), cal vigilar la **concurrència**: dos clients pujant un fitxer amb el mateix nom alhora, o llegint mentre un altre escriu, poden generar condicions de carrera. Es pot mitigar amb:
 - Blocs `synchronized` sobre l'objecte o recurs compartit.
 - Estructures de dades thread-safe (`ConcurrentHashMap`, etc.) si es manté estat compartit en memòria (per exemple, una llista de clients connectats).
 - Bloquejos a escala de fitxer (`FileLock`) si cal garantir exclusivitat sobre un fitxer concret.
- Una limitació del disseny bàsic vist a l'apartat 7 és que **els canvis fets per un**

client no es notifiquen als altres fins que aquests tornen a demanar l'estructura. Una millora habitual (proposada com a activitat) és afegir un nou valor REFRESCAR a l'enum Tipus de MissatgeProtocol perquè el client pugui demanar l'arbre actualitzat sota demanda.

8.1 Alternativa: pools de fils

Crear un Thread nou per cada client és senzill, però pot ser costós si s'esperen **moltes** connexions simultànies. Una alternativa més eficient és utilitzar un **ExecutorService** (pool de fils) de `java.util.concurrent`:

```
ExecutorService pool = Executors.newFixedThreadPool(20); // màxim 20
    ↪  fils actius

while (true) {
    Socket connexio = servidor.accept();
    pool.submit(new GestorClient(connexio, arrel)); // GestorClient
    ↪  implements Runnable
}
```

Això limita el nombre de fils simultanis i reaprofitja els fils ja creats, evitant la sobrecàrrega de crear-ne un de nou per cada connexió.

9. Monitoratge i verificació de la disponibilitat del servei

Per comprovar que el nostre servei (o qualsevol servei estàndard) està disponible i funciona correctament, disposem de diverses eines i tècniques:

9.1 Clients de comunicacions genèrics

- **Telnet** a un port concret és la manera més ràpida de comprovar si un servei TCP està escoltant i respon: `telnet <host> <port>`.
- **netcat/nc**: alternativa a telnet, més flexible (`nc -vz host port` per comprovar si un port està obert).
- El nostre propi **client** (FTP, Telnet, SMTP, POP3, o el client del servidor de fitxers) també és una eina de verificació: si es connecta i respon amb el codi esperat, el servei és disponible.

9.2 Comprovació programàtica

Des de Java mateix podem escriure petits “health checks” que provin de connectar-se i mesurin el temps de resposta:

```
public static boolean serveiDisponible(String host, int port, int
↪ timeoutMs) {
    try (Socket s = new Socket()) {
        s.connect(new InetSocketAddress(host, port), timeoutMs);
        return true;
    } catch (IOException e) {
        return false;
    }
}
```

Per als clients de Commons Net, comprovar sempre el codi de resposta després de `connect()` (`FTPReply.isPositiveCompletion(...)`, `SMTPReply.isPositiveCompletion(...)`) és, de fet, la primera comprovació de disponibilitat del servei remot.

9.3 Registres (logs)

- Activar els *listeners* de comandaments (`addProtocolCommandListener`) en els clients de Commons Net permet registrar tota la conversa client-servidor, útil per detectar errors intermitents.
- En el nostre servidor propi, cal afegir traces (`System.out/Logger`) en punts clau: connexió/desconnexió de clients, errors d'E/S, peticions rebudes. Això facilita tant el **monitoratge** en producció com la **depuració** durant el desenvolupament.
- Eines externes de monitoratge (Zabbix, Nagios...) es poden configurar per comprovar periòdicament que el port del servei respon, i alertar si deixa de fer-ho.

9.4 Gestió d'excepcions com a indicador de disponibilitat

Si el servidor tanca una connexió per inactivitat (habitual en FTP després d'uns 900 segons), es produirà una `FTPConnectionClosedException`. Detectar i tractar aquestes excepcions específiques ---en comptes de només capturar `IOException` genèricament--- permet distingir entre “el servei no està disponible” i “la connexió ha caducat per inactivitat”, i reaccionar-hi de manera diferent (per exemple, reconnectant automàticament).

10. Bones pràctiques i depuració

Per completar l'aplicació de manera professional:

- **Tancar sempre els recursos:** sockets, streams i clients s'han de tancar en un bloc *finally* (o amb *try-with-resources*), encara que es produeixi una excepció.
- **Comprovar codis de resposta** després de cada operació de protocol, no assumir mai que ha anat bé.
- **Separar responsabilitats:** una classe/mètode per a cada operació del protocol (connectar, llistar, pujar, baixar...), en comptes d'un mètode *main()* gegant.
- **Documentar el codi:** javadoc a les classes de missatge (*DirectoriInfo*, *MissatgeProtocol*...) i comentaris que expliquin el flux de comunicació client-servidor.
- **Provar amb clients externs:** abans de donar per bo un servidor propi, provar-lo amb *telnet*, nc o un client de propòsit general, per assegurar-nos que la conversa de baix nivell és correcta.
- **Gestionar la mida de les dades:** no assumir que un fitxer sempre cap en memòria; pensar en transferències per blocs per a fitxers grans.
- **Seguretat bàsica:** recordar que *Telnet*, *FTP* i *SMTP* “clàssics” viatgen en text pla; per a serveis reals, preferir les variants amb *TLS/SSL* (*FTPSCient*, *SMTPSCient*, *POP3SCient*) o protocols moderns (*SFTP*, *IMAPS*).

Versions d'aquest document

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

[Domini Públic \(CC0\)](#)