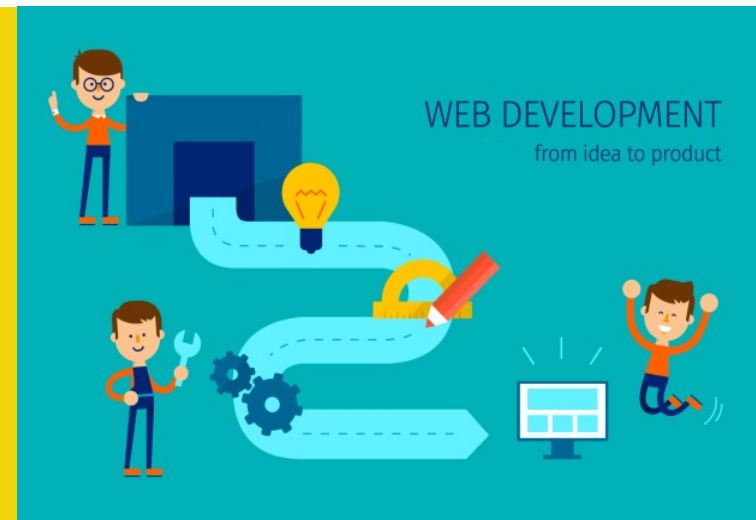


# Módulo 9

Código: 490

## Programación de Servicios y Procesos

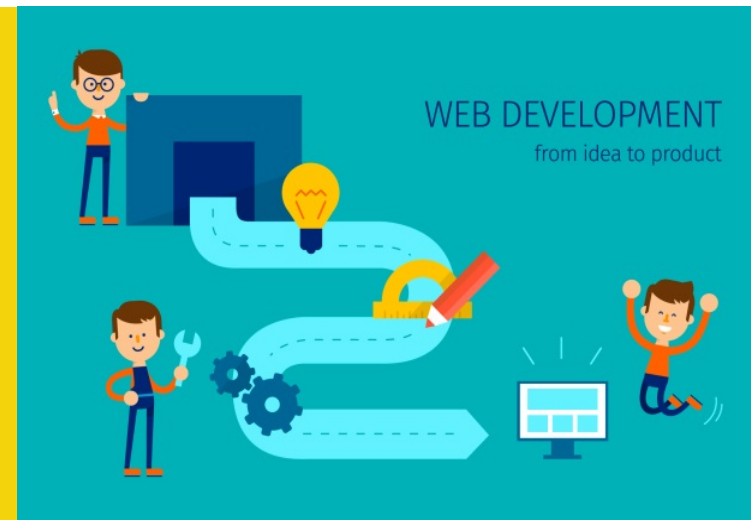
Técnico Superior en Desarrollo de  
Aplicaciones Multiplataforma



# UNIDAD 4

## Generación de servicios en red

Contenidos teóricos



1. Objetivos generales de la unidad
2. Competencias y contribución en la unidad
3. Contenidos conceptuales y procedimentales
4. Evaluación
5. Distribución de los contenidos
6. Sesiones

# 1. Objetivos generales de la unidad

## Objetivos curriculares

---


1. Utilizar librerías Java para implementar protocolos estándar de comunicación en red

2. Programar clientes de protocolos estándar de comunicación en red.

3. Probar servicios de comunicación en red.

4. Crear servicios que gestionen varios clientes.

## 2. Competencias y contribución en la unidad



a) Desarrollar aplicaciones capaces de ofrecer servicios en red empleando mecanismos de comunicación.

	A través del uso de librerías Java para implementar protocolos estándar de comunicaciones y creando clientes para acceder a servidores SMTP.	
--	--	--

### 3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Protocolos estándar de comunicación en red	<ul style="list-style-type: none"><li>• Servicios de emulación de terminal: Telnet/SSH</li><li>• Servicios de transferencia de archivos: FTP, TFTP.</li><li>• Protocolo de transferencia de hipertexto: HTTP.</li><li>• Protocolos de correo electrónico: POP3, SMTP.</li><li>• Sistema de nombres de dominio: DNS.</li><li>• Protocolo de configuración dinámica de host: DHCP.</li></ul>	<ul style="list-style-type: none"><li>• Comprender el concepto de servicio.</li><li>• Estudiar los diferentes protocolos de comunicación en red a nivel de aplicación.</li></ul>

### 3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Librerías de clases para comunicar con los protocolos estándar	<ul style="list-style-type: none"><li>• JavaMail.</li><li>• Apache Commons Email.</li></ul>	<ul style="list-style-type: none"><li>• Crear programas en Java que utilicen librerías de clases y componentes, para implementar protocolos estándar de comunicación.</li></ul>
Crear clientes de protocolos estándar de comunicación en red	<ul style="list-style-type: none"><li>• Clientes SMTP.</li></ul>	<ul style="list-style-type: none"><li>• Utilizar librerías Java para crear clientes que accedan a servidores de correo electrónico SMTP.</li></ul>
Programación de servidores	<ul style="list-style-type: none"><li>• Servidores Java y clientes multihilo.</li></ul>	<ul style="list-style-type: none"><li>• Crear programas servidor utilizando Java, que ofrezcan servicio a múltiples clientes de forma concurrente.</li></ul>

## 4. Evaluación

Se han analizado librerías que permitan implementar protocolos estándar de comunicación en red.

Se han programado clientes de protocolos estándar de comunicaciones y verificado su funcionamiento .

Se han desarrollado y probado servicios de comunicación en red.

Se han analizado los requerimientos necesarios para crear servicios capaces de gestionar varios clientes concurrentes.

Se han incorporado mecanismos para posibilitar la comunicación simultánea de varios clientes con el servicio.

Se han depurado y documentado las aplicaciones desarrolladas.



## 1. CONCEPTO DE SERVICIO

- Los **servicios** son “programas” utilizados en redes de ordenadores con el objetivo de gestionar recursos y prestar funcionalidad (servicio, de ahí su nombre) a los usuarios del sistema y/o aplicaciones.
- Por ejemplo, cuando dentro de una red enviamos un documento a una impresora, estamos usando el servicio de impresión, el cual, permite gestionar y compartir la impresora en la red.
- También hay servicios para mantener la seguridad y la operativa amigable de los recursos de la red.
- Por ejemplo, las redes locales corporativas usan servicios de red como DNS (Domain Name System) para dar nombres a las direcciones IP y MAC y DHCP para asegurar que todos en la red tienen una dirección IP válida.
- Realizar tareas de administración de red sin estos servicios sería una tarea muy complicada.

## 1. CONCEPTO DE SERVICIO

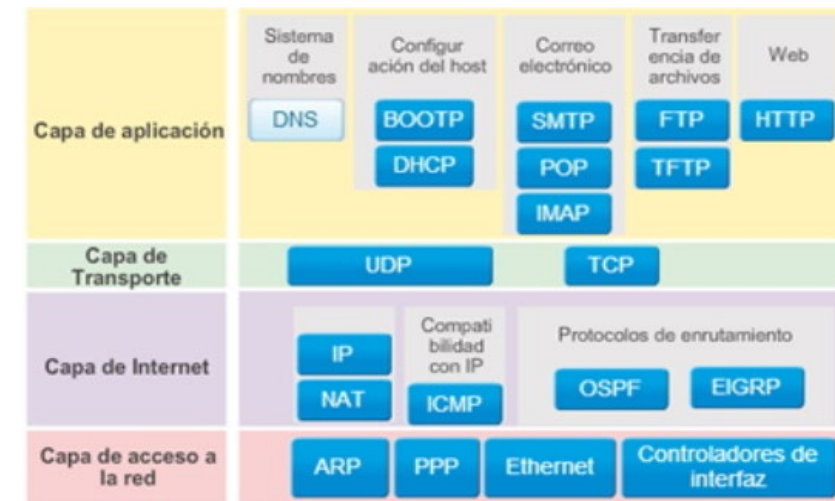
- Nosotros como usuarios, podemos acceder al servicio, a través de las operaciones que ofrece.
- Todos los servicios en una red, sea local o Internet utilizan una arquitectura cliente-servidor.
- Vamos a ver como podemos acceder desde nuestros programas Java a los servicios de red más habituales.



<https://alejandromunozsegrera.wordpress.com/2014/02/18/servicios-en-red/>

## 2. PROTOCOLOS ESTÁNDAR DE COMUNICACIÓN

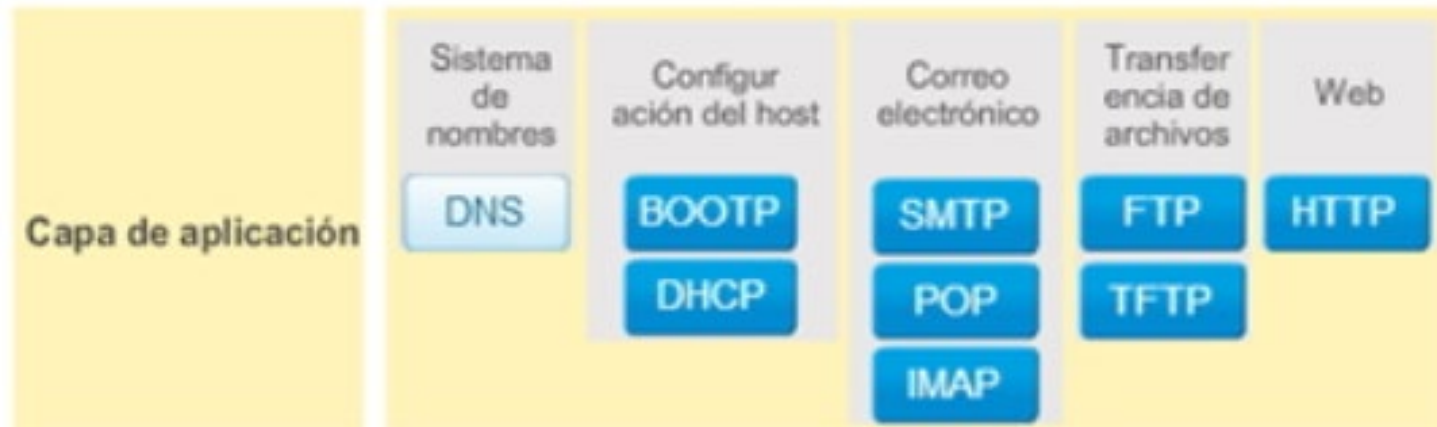
- Un **protocolo de comunicación** es un conjunto de reglas que regulan la comunicación entre los dispositivos de una red.
- Los protocolos se organiza en capas, las cuales no pueden funcionar independientemente sin las demás.
- Las capas inferiores ofrecen servicios a las capas superiores. Normalmente, las capas inferiores se encargan de enviar el mensaje mientras que las capas superiores de mostrarlo a los destinatarios.
- Las redes, utilizan la **familia de protocolos TCP/IP**, cuyas capas se pueden ver en la imagen:



<https://yaniraticsaifa.wordpress.com/2016/09/26/2-1-protocolos-tcpip/>

## 2. PROTOCOLOS ESTÁNDAR DE COMUNICACIÓN

- La **capa de aplicación** define las aplicaciones y servicios estándar que pueden usar los usuarios de una red de ordenadores.
- Estos servicios y aplicaciones se basan en el modelo cliente-servidor.
- A lo largo de esta unidad vamos a ir viendo el funcionamiento de estos protocolos y las clases Java para acceder a los servicios de red más habituales.



## 3. SERVICIOS DE EMULACIÓN DE TERMINAL

### Telnet/SSH

- Hoy en día prácticamente obsoleto, de hecho, no aparece en la imagen anterior.
- Fue un servicio que permitía conectarse a máquinas UNIX desde un sitio remoto, permitiendo enviar comandos y ver el resultado de dichos comandos en nuestro terminal como si estuviésemos en la máquina UNIX real.
- Su gran problema era la seguridad. Telnet no enviaba los datos cifrados por lo que cualquier persona con un *sniffer* podía capturar el tráfico de red y ver no solamente los comandos sino también los usuarios y contraseñas enviados a través de la red.
- Por ello, Telnet prácticamente no se usa hoy en día y ha sido sustituido por **SSH**, que hace lo mismo pero cifrando la comunicación con criptografía de clave pública.

## 4. SERVICIOS DE TRANSFERENCIA DE ARCHIVOS

### FTP

- Significa *File Transfer Protocol*.
- Es un protocolo orientado a comandos pensado para descargar ficheros.
- También tiende a desaparecer, ya que cada vez más a menudo se usa el navegador (con HTTP) para descargar ficheros.
- Algunos navegadores, como Mozilla Firefox siguen implementando el protocolo FTP, permitiendo así el descargar ficheros como si en realidad estuviésemos usando comandos.
- Los sitios FTP son lugares desde los cuales podemos descargar o enviar archivos.
- FTP tiene dos modalidades de uso: pasivo y activo.  
La diferencia está en quien inicia las conexiones.

## 4. SERVICIOS DE TRANSFERENCIA DE ARCHIVOS

### TFTP

- Sus siglas significan *Trivial File Transfer Protocol*.
- Es un protocolo trivial de transferencia de ficheros, muy simple, semejante a una versión básica de FTP, de ahí su adjetivo "trivial".
- Fue definido para aplicaciones que no necesitaban tanta interacción entre el servidor y el cliente.
- A menudo se usa para transferir archivos entre los ordenadores de una red en los que no es necesaria una autenticación.
- Es un protocolo no orientado a conexión que utiliza el protocolo UDP.

## 5. PROTOCOLO DE TRANSFERENCIA DE HIPERTEXTO

### HTTP

- *HyperText Transfer Protocol.*
- Este protocolo inicialmente se diseñó para que los navegadores (clientes web) se conectasen a servidores y descargasen archivos HTML.
- Sin embargo, su uso se ha popularizado en otros ámbitos como son la creación de aplicaciones web, es decir aplicaciones pensadas para ser manejadas desde un navegador.
- Es decir utilizando este protocolo, los navegadores web realizan peticiones a los servidores web y reciben sus respuestas.
- Es un protocolo basado en texto.



## 6. PROTOCOLOS DE CORREO ELECTRÓNICO

### POP3

- Significa *Post Office Protocol version 3*.
- Está pensando fundamentalmente para descargar correo desde un servidor de correo al ordenador de un cliente que habitualmente utiliza Outlook, Mozilla Thunderbird o algún otro cliente de correo.
- Las versiones más antiguas no usaban mecanismos de cifrado, pero al igual que Telnet, POP3 ha necesitado cambiar para manejar correctamente la seguridad (aunque Telnet ha sido sustituido por otro protocolo y POP3 lo que ha hecho ha sido incorporar extensiones).

## 6. PROTOCOLOS DE CORREO ELECTRÓNICO

### SMTP

- *Simple Mail Transfer Protocol.*
- Al contrario que POP3, está pensado sobre todo para enviar correo.
- Es decir, SMTP es utilizado por máquinas que son clientes de correo o servidores de correo:
  - ✓ **Cliente de correo** de un servidor de correo (por ejemplo hotmail.es). Quiere enviar un mensaje a un usuario que tiene su cuenta en gmail.com. Esto significa que este cliente de correo primero tiene que enviar el mensaje a hotmail.es y pedirle que lo entregue. La subida del correo a hotmail.es se hace mediante SMTP.
  - ✓ **Servidor de correo** que quiere enviar mensajes a otro servidor. En el ejemplo anterior hotmail.es recibe un correo que debe entregar a su destinatario, pero como dicho destinatario está en otro servidor debe entregarlo a gmail.com que comprobará si tiene el usuario destino del mensaje, y si es así, recibirá el mensaje. Todo este proceso también se hace mediante SMTP.

## 7. SISTEMA DE NOMBRES DE DOMINIO

### DNS

- *Domain Name System.*
- El sistema de nombres de dominio es un sistema que usa servidores distribuidos a lo largo de la red para resolver el nombre de un host IP (nombre de ordenador + nombre de subdominio + nombre de dominio) en una dirección IP.
- De esta manera no es necesario recordar su dirección IP, sino que se usa su nombre DNS.

## 8. PROTOCOLO DE CONFIGURACIÓN DINÁMICA DE HOST

### DHCP

- *Dynamic Host Configuration Protocol.*
- Protocolo de red de tipo cliente/servidor mediante el cual un servidor asigna dinámicamente una dirección IP y otros parámetros de configuración de red a cada dispositivo en una red para que puedan comunicarse con otras redes IP.
- Este servidor posee una lista de direcciones IP dinámicas y las va asignando a los clientes conforme estas van quedando libres, sabiendo en todo momento quién ha estado en posesión de esa IP, cuánto tiempo la ha tenido y a quién se la ha asignado después.
- De esta forma, los clientes de una red pueden conseguir sus parámetros de configuración automáticamente.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Lo que vamos a ver en las sesiones siguientes, es como crear un programa Java para comunicarnos con un servidor de correo SMTP.
- Vamos a utilizar un servidor de correo público: **Gmail**.
- Usaremos una **cuenta de prueba**, que configuraremos para permitir el acceso a aplicaciones desconocidas (la aplicación desconocida será nuestro programa).
- Una vez que hayamos hecho esto en nuestra cuenta de Gmail, el programa Java podrá enviar y recibir correos electrónicos con nuestra cuenta de correo.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

Google Cuenta

### ← Acceso de aplicaciones poco seguras

Algunos dispositivos y aplicaciones utilizan una tecnología de inicio de sesión menos segura, lo cual hace que tu cuenta sea más vulnerable, por lo que te recomendamos que **desactives** el acceso de estas aplicaciones. Si, a pesar del riesgo que ello supone, quieres utilizarlas, puedes **activar** el acceso. [Más información](#)

Permitir el acceso de aplicaciones poco seguras: Sí

#### Acceso de aplicaciones poco seguras

Tu cuenta es vulnerable porque permites el acceso de aplicaciones y dispositivos que utilizan una tecnología de inicio de sesión menos segura



! Activado

[Desactivar acceso \(opción recomendada\)](#)

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Librerías que vamos a usar:

JavaMail	Apache Commons Email
Biblioteca de libre distribución que proporciona los objetos necesarios para enviar y recibir correos desde cualquier aplicación Java.	Biblioteca de libre distribución que facilita el desarrollo de aplicaciones que necesiten enviar o recibir email.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Todos los objetos que vamos a usar se encuentran en estas librerías.
- A medida que vayamos viendo los diferentes ejemplos, iremos comentando las clases que intervienen en el proceso.
- Estas librerías proporcionan de manera transparente el proceso correcto para el establecimiento y finalización de conexiones.
- Igualmente, toda la transmisión de información es controlada por las bibliotecas, liberando al programador de tareas de control.



## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

Ejemplo de envío de correo electrónico con texto.

- Usamos la clase JavaMail:

```
public class EnviarCorreoTexto {  
  
    public static void main(String[] args) {  
        try {  
            // Propiedades de la conexión  
            Properties props = new Properties();  
            // Nombre del host de correo  
            props.setProperty("mail.smtp.host", "smtp.gmail.com");  
            // TLS si está disponible  
            props.setProperty("mail.smtp.starttls.enable", "true");  
            // Puerto de gmail para envío de correos  
            props.setProperty("mail.smtp.port", "587");  
            // Nombre del usuario  
            props.setProperty("mail.smtp.user", "procesos.daml9@gmail.com");  
            // Si requiere o no usuario y password para conectarse.  
            props.setProperty("mail.smtp.auth", "true");  
  
            // Creamos la sesion  
            Session session = Session.getDefaultInstance(props);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

```
// Construimos el mensaje
MimeMessage message = new MimeMessage(session);
// Quién manda el mensaje
message.setFrom(new InternetAddress("XXXXXX@hotmail.com"));
// A quién va dirigido el mensaje.
message.addRecipient(
    Message.RecipientType.TO,
    new InternetAddress("procesos.daml9@gmail.com"));
// Asunto del mensaje
message.setSubject("EJEMPLO CON JAVA MAIL");
// Texto del mensaje
message.setText(
    "Hola este es un mensaje enviado desde Java con Java Mail.");

// Enviamos el mensaje
Transport t = session.getTransport("smtp");
t.connect("procesos.daml9@gmail.com", "XXXXXX");
t.sendMessage(message, message.getAllRecipients());

// Cierre de la conexión
t.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Clases implicadas de JavaMail:

Session	Representa la conexión con el servidor Gmail de correo. Hay que obtener la sesión pasándola los parámetros de configuración del servidor de correo.
Transport	Clase para el envío de mensajes. Se obtiene llamando al método <i>getTransport()</i> de la clase Session.
MimeMessage	Clase que representa al mensaje.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Configuración de la sesión:
  - ✓ Creación de un objeto **Properties** con las propiedades de la conexión.
  - ✓ Para establecerlas usamos el método *setProperty*.
  - ✓ Las Properties que necesita Session son específicas. Puedes consultarlas en el API.

```
// Propiedades de la conexión
Properties props = new Properties();
// Nombre del host de correo
props.setProperty("mail.smtp.host", "smtp.gmail.com");
// TLS si está disponible
props.setProperty("mail.smtp.starttls.enable", "true");
// Puerto de gmail para envío de correos
props.setProperty("mail.smtp.port", "587");
// Nombre del usuario
props.setProperty("mail.smtp.user", "procesos.daml9@gmail.com");
// Si requiere o no usuario y password para conectarse.
props.setProperty("mail.smtp.auth", "true");
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación de la sesión:
  - ✓ Creación de un objeto `Session` con el método *`getDefaultInstance`*.

```
// Creamos la sesion  
Session session = Session.getDefaultInstance(props);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación del mensaje de correo:
  - ✓ Creación de un objeto **MimeMessage** pasándole el objeto Session creado anteriormente.
  - ✓ Configuramos este objeto/mensaje:
    - **FROM** del mensaje: método *setFrom*. La dirección de correo origen se crea con un objeto **InternetAddress**.
    - **TO** del mensaje: método *addRecipient*.  
Este método admite dos parámetros, una constante para indicar el tipo de destinatario y el objeto **InternetAdress** con la dirección de correo destino.  
El tipo de destinatario puede ser:
      - Message.RecipientType.TO*: destinatario principal.
      - Message.RecipientType.CC*: destinatario al que se envía copia del mensaje.
      - Message.RecipientType.BCC*: destinatario al que se envía copia, pero sin que los demás destinatarios puedan verlo.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación del mensaje de correo:
  - ✓ Configuramos este objeto/mensaje:
    - **ASUNTO**: método *setSubject*.
    - **TEXTO**: método *setText*.

Este mensaje es de texto plano. Pero si se quiere formatear se puede enviar por ejemplo en formato HTML.

En este caso debemos pasar al método `setText` una serie de parámetros:

El texto, en formato html.

El juego de caracteres a utilizar. Por ejemplo, "UTF-8".

Y el formato a utilizar: html en nuestro caso.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación del mensaje de correo:

```
// Construimos el mensaje
MimeMessage message = new MimeMessage(session);
// Quién manda el mensaje
message.setFrom(new InternetAddress("XXXXXX@hotmail.com"));
// A quién va dirigido el mensaje.
message.addRecipient(
    Message.RecipientType.TO,
    new InternetAddress("procesos.daml9@gmail.com"));
// Asunto del mensaje
message.setSubject("EJEMPLO CON JAVA MAIL");
// Texto del mensaje
message.setText(
    "Hola este es un mensaje enviado desde Java con Java Mail.");
```



## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Envío del mensaje de correo:
  - ✓ Creación de un objeto **Transport**. Hay que pasarle como parámetro el nombre del protocolo a usar, en nuestro caso para Gmail, SMTP.
  - ✓ Establecemos la conexión con el método *connect*, al que le pasamos la cuenta de correo electrónico y la contraseña.
  - ✓ Mandamos el mensaje con el método *sendMessage*. A este método le pasamos el mensaje y la lista de receptores del mismo. Para ello con el método *getAllRecipients* obtenemos el array de direcciones destinatarias de mensaje.
  - ✓ Finalmente cerramos la conexión con el método *close*.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Envío del mensaje de correo:

```
// Enviamos el mensaje
Transport t = session.getTransport("smtp");
t.connect("procesos.daml9@gmail.com", "XXXXXXXX");
t.sendMessage(message, message.getAllRecipients());

// Cierre de la conexión
t.close();
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

### Ejemplo de lectura de correo electrónico.

- Usamos la clase JavaMail:

```
public class LeerCorreo {  
  
    public static void main(String[] args) {  
  
        // Configuramos la sesión  
        Properties prop = new Properties();  
        // Deshabilitamos TLS  
        prop.setProperty("mail.pop3.starttls.enable", "false");  
        // Hay que usar SSL  
        prop.setProperty(  
            "mail.pop3.socketFactory.class", "javax.net.ssl.SSLSocketFactory");  
        prop.setProperty("mail.pop3.socketFactory.fallback", "false");  
        // Puerto 995 para conectarse.  
        prop.setProperty("mail.pop3.port", "995");  
        prop.setProperty("mail.pop3.socketFactory.port", "995");  
  
        Session sesion = Session.getInstance(prop);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

```
try {  
    // Obtención de la carpeta de la bandeja de entrada  
    Store store = sesion.getStore("pop3");  
    store.connect(  
        "pop.gmail.com", "procesos.daml9@gmail.com", "XXXXXXX");  
    Folder folder = store.getFolder("INBOX");  
    folder.open(Folder.READ_ONLY);  
  
    // Se obtienen los mensajes.  
    Message[] mensajes = folder.getMessages();  
  
    // Obtenemos el origen y asunto de cada mensaje  
    for (Message mensaje : mensajes) {  
        System.out.println("From:" + mensaje.getFrom()[0].toString());  
        System.out.println("Subject:" + mensaje.getSubject());  
        // Se visualiza el contenido de cada mensaje  
        mostrarMensaje(mensaje);  
    }  
  
    folder.close(false);  
    store.close();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

```
private static void mostrarMensaje(Part unaParte) {  
    try {  
        // Si es multipart, se analiza cada una de sus partes recursivamente.  
        if (unaParte.isMimeType("multipart/*")) {  
            Multipart multi;  
            multi = (Multipart) unaParte.getContent();  
            for (int j = 0; j < multi.getCount(); j++) {  
                mostrarMensaje(multi.getBodyPart(j));  
            }  
        } else // Si es texto, se escribe el texto.  
        if (unaParte.isMimeType("text/*")) {  
            System.out.println("Texto " + unaParte.getContentType());  
            System.out.println(unaParte.getContent());  
        } else // Si es imagen, se guarda en fichero y se visualiza en JFrame  
        if (unaParte.isMimeType("image/*")) {  
            System.out.println(  
                "Imagen " + unaParte.getContentType());  
            System.out.println("Fichero=" + unaParte.getFileName());  
  
            guardarImagen(unaParte);  
            mostrarImagen(unaParte);  
        } else { // Si no es de ninguno de los tipos anteriores mostramos el tipo  
            System.out.println("Tipo: " + unaParte.getContentType());  
        }  
    } catch (MessagingException | IOException e) {  
        e.printStackTrace();  
    }  
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

```
private static void mostrarImagen(Part unaParte)
    throws IOException, MessagingException {
    JFrame v = new JFrame();
    ImageIcon icono = new ImageIcon(
        ImageIO.read(unaParte.getInputStream()));
    JLabel imagen = new JLabel(icono);
    v.getContentPane().add(imagen);
    v.pack();
    v.setVisible(true);
}
```

```
private static void guardarImagen(Part unaParte)
    throws FileNotFoundException, MessagingException, IOException {
    FileOutputStream fichero = new FileOutputStream(
        "D:/" + unaParte.getFileName());
    InputStream imagen = unaParte.getInputStream();
    byte[] bytes = new byte[1000];
    int leidos = 0;

    while ((leidos = imagen.read(bytes)) > 0) {
        fichero.write(bytes, 0, leidos);
    }
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Clases implicadas de JavaMail:

Session	Representa la conexión con el servidor Gmail de correo. Hay que obtener la sesión pasándole los parámetros de configuración del servidor de correo.
Store y Folder	Clases que representan el almacén del servidor con los correos y la carpeta dentro de ese almacén donde el servidor guarda los correos que queremos leer.
MimeMessage	Clase que representa los mensajes que leemos.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Configuración de la **sesión**:
  - ✓ Creación de un objeto **Properties** con las propiedades de la conexión.
  - ✓ Para establecerlas usamos el método *setProperty*.
  - ✓ Las Properties que necesita Session son específicas. Puedes consultarlas en el API.
    - Para usar SSL debemos decirle a JavaMail como obtener un socket SSL.
    - Ponemos la propiedad *mail.pop3.socketFactory.fallback* a false, deshabilitando la posibilidad de que JavaMail intente con sockets normales si falla el socket SSL, Ya que Gmail solo admite sockets SSL.
  - ✓ Indicamos el puerto 995 tanto para JavaMail como para la clase que crea sockets SSL.



## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Configuración de la sesión:

```
// Configuramos la sesión
Properties prop = new Properties();
// Deshabilitamos TLS
prop.setProperty("mail.pop3.starttls.enable", "false");
// Hay que usar SSL
prop.setProperty(
    "mail.pop3.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
prop.setProperty("mail.pop3.socketFactory.fallback", "false");
// Puerto 995 para conectarse.
prop.setProperty("mail.pop3.port", "995");
prop.setProperty("mail.pop3.socketFactory.port", "995");
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación de la **sesión**:
  - ✓ Creación de un objeto `Session` con el método *`getDefaultInstance`*.

```
// Creamos la sesion  
Session session = Session.getDefaultInstance(props);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Obtención de la carpeta de la bandeja de entrada:
  - ✓ Creación de un objeto **Store** con el método *getStore*.
  - ✓ Establecemos la conexión con el servidor de correo con el método *connect*, indicando servidor, cuenta de correo y contraseña.
  - ✓ Pedimos la carpeta de la bandeja de entrada con el método *getFolder*. Este método devuelve un objeto **Folder**.
  - ✓ El Folder que representa a la carpeta de la bandeja de entrada la abrimos para solo lectura.

```
// Obtención de la carpeta de la bandeja de entrada
Store store = sesion.getStore("pop3");
store.connect(
    "pop.gmail.com", "procesos.daml9@gmail.com", "XXXXXX");
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Obtención de los mensajes nuevos:
  - ✓ Llamada al método *getMessages* del objeto Folder.
  - ✓ Recorremos el array de mensajes, y para cada uno de ellos mostramos el origen y el asunto.
  - ✓ Como el método *getFrom* devuelve un array de direcciones, obtenemos la primera dirección.

```
// Se obtienen los mensajes.  
Message[] mensajes = folder.getMessages();  
  
// Obtenemos el origen y asunto de cada mensaje  
for (Message mensaje : mensajes) {  
    System.out.println("From:" + mensaje.getFrom()[0].toString());  
    System.out.println("Subject:" + mensaje.getSubject());  
    // Se visualiza el contenido de cada mensaje  
    analizaParteDeMensaje(mensaje);  
}  
  
folder.close(false);  
store.close();
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Obtención de los mensajes nuevos:
  - ✓ Mostramos el contenido del mensaje con el método *mostrarMensaje*.

```
// Se obtienen los mensajes.
Message[] mensajes = folder.getMessages();

// Obtenemos el origen y asunto de cada mensaje
for (Message mensaje : mensajes) {
    System.out.println("From:" + mensaje.getFrom()[0].toString());
    System.out.println("Subject:" + mensaje.getSubject());
    // Se visualiza el contenido de cada mensaje
    mostrarMensaje(mensaje);
}

folder.close(false);
store.close();
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Mostrar el mensaje:
  - ✓ Vemos de qué tipo es el mensaje. Ya que puede ser simplemente de texto plano o con formato html, incluso puede tener ficheros adjuntos.
  - ✓ Para ello usamos el método *isMimeType* de la interfaz **Part** que implementa **Message**.

```
// Si es multipart, se analiza cada una de sus partes recursivamente.  
if (unaParte.isMimeType("multipart/*")) {  
  
} else // Si es texto, se escribe el texto.  
if (unaParte.isMimeType("text/*")) {  
  
    } else // Si es imagen, se guarda y se visualiza en JFrame  
    if (unaParte.isMimeType("image/*")) {  
  
        } else { // Si no es de ninguno de los tipos anteriores mostramos el tipo  
        System.out.println("Tipo: " + unaParte.getContentType());  
    }  
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Mensaje de texto:
  - ✓ Vamos a suponer que es text/plain.
  - ✓ Para otros tipos, como text/html o text/rtf, necesitaremos otra forma de visualizarlo como por ejemplo usar un

```
// Si es texto, se escribe el texto.  
if (unaParte.isMimeType("text/*")) {  
    System.out.println("Texto " + unaParte.getContentType());  
    System.out.println(unaParte.getContent());  
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Mensaje multipart:
  - ✓ Extraemos las partes que componen el mensaje compuesto con el método *getContent*.
  - ✓ Iteramos por todas ellas, accediendo a cada parte con el método *getBodyPart*.
  - ✓ Con cada parte hacer un tratamiento similar, es decir, hay que analizarla, ya que puede ser una parte solo texto, una imagen o multipart, por eso el método es recursivo.

```
// Si es multipart, se analiza cada una de sus partes recursivamente.  
if (unaParte.isMimeType("multipart/*")) {  
    Multipart multi;  
    multi = (Multipart) unaParte.getContent();  
    for (int j = 0; j < multi.getCount(); j++) {  
        mostrarMensaje(multi.getBodyPart(j));  
    }  
}
```



## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Imagen:
  - ✓ Guardamos en el disco la imagen con el método *guardarImagen*.
  - ✓ La mostramos en un JFrame con el método *mostrarImagen*.

```
if (unaParte.isMimeType("image/*")) {  
    System.out.println(  
        "Imagen " + unaParte.getContentType());  
    System.out.println("Fichero=" + unaParte.getFileName());  
  
    guardarImagen(unaParte);  
    mostrarImagen(unaParte);  
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Imagen:
  - ✓ Método *guardarImagen*:
    - Leemos la imagen de 1000 en 1000 bytes usando un stream de bytes.
    - Y la almacenamos en la unidad D:

```
private static void guardarImagen(Part unaParte)
    throws FileNotFoundException, MessagingException, IOException {
    FileOutputStream fichero = new FileOutputStream(
        "D:/" + unaParte.getFileName());
    InputStream imagen = unaParte.getInputStream();
    byte[] bytes = new byte[1000];
    int leidos = 0;

    while ((leidos = imagen.read(bytes)) > 0) {
        fichero.write(bytes, 0, leidos);
    }
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Imagen:
  - ✓ Método *mostrarImagen*:
    - Mostramos la imagen usando un objeto `ImageIcon` que colocamos en una etiqueta.

```
private static void mostrarImagen(Part unaParte)
    throws IOException, MessagingException {
    JFrame v = new JFrame();
    ImageIcon icono = new ImageIcon(
        ImageIO.read(unaParte.getInputStream()));
    JLabel imagen = new JLabel(icono);
    v.getContentPane().add(imagen);
    v.pack();
    v.setVisible(true);
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

Ejemplo de envío de correo electrónico de texto con archivo adjunto.

- Usamos la clase JavaMail:

```
public class EnviarCorreoAdjunto {  
  
    public static void main(String[] args) {  
        try {  
            // Propiedades de la conexión  
            Properties props = new Properties();  
            props.put("mail.smtp.host", "smtp.gmail.com");  
            props.setProperty("mail.smtp.starttls.enable", "true");  
            props.setProperty("mail.smtp.port", "587");  
            props.setProperty("mail.smtp.user", "procesos.daml9@gmail.com");  
            props.setProperty("mail.smtp.auth", "true");  
  
            // Creamos la sesion  
            Session session = Session.getDefaultInstance(props, null);  
  
            // Formamos el texto del mensaje  
            BodyPart texto = new MimeBodyPart();  
            texto.setText("Texto del mensaje");  
  
            // Formamos el adjunto (imagen) del mensaje  
            BodyPart adjunto = new MimeBodyPart();  
            adjunto.setDataHandler(  
                new DataHandler(new FileDataSource("d:/imagen.gif"))  
            );  
            adjunto.setFileName("imagen.gif");  
        }  
    }  
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

Ejemplo de envío de correo electrónico de texto con archivo adjunto.

- Usamos la clase JavaMail:

```
// Agrupamos texto e imagen.
MimeMultipart multiParte = new MimeMultipart();
multiParte.addBodyPart(texto);
multiParte.addBodyPart(adjunto);

// Construimos el mensaje
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress("XXXXXXX@hotmail.com"));
message.addRecipient(
    Message.RecipientType.TO,
    new InternetAddress("procesos.daml9@gmail.com"));
message.setSubject("EJEMPLO DE CORREO CON ADJUNTO CON JAVA MAIL");
message.setContent(multiParte);

// Lo enviamos.
Transport t = session.getTransport("smtp");
t.connect("procesos.daml9@gmail.com", "XXXXXXX");
t.sendMessage(message, message.getAllRecipients());

//Cierre
t.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Clases implicadas de JavaMail:

Session y Transport	Session representa la conexión con el servidor Gmail de correo y la clase <b>Transport</b> se usa para el envío del mensaje.
MimeMessage MimeMultipart y MimeBodyPart	Clase que vamos a utilizar para construir el mensaje.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Configuración de la sesión:
  - ✓ Creación de un objeto **Properties** con las propiedades de la conexión.
  - ✓ Para establecerlas usamos el método *setProperty*.
  - ✓ Las Properties que necesita Session son específicas. Puedes consultarlas en el API.

```
// Propiedades de la conexión
Properties props = new Properties();
props.put("mail.smtp.host", "smtp.gmail.com");
props.setProperty("mail.smtp.starttls.enable", "true");
props.setProperty("mail.smtp.port", "587");
props.setProperty("mail.smtp.user", "procesos.daml9@gmail.com");
props.setProperty("mail.smtp.auth", "true");
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación de la sesión:
  - ✓ Creación de un objeto `Session` con el método *`getDefaultInstance`*.

```
// Creamos la sesion  
Session session = Session.getDefaultInstance(props);
```



## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Construcción del mensaje de texto con adjunto:
  - ✓ Creación de dos objetos **MimeBodyPart** para construir las dos partes del mensaje: texto e imagen.
    - **Texto**: con el método *setText* asignamos el texto plano al mensaje.
    - **Imagen**: usamos el método *setDataHandler* al que le pasamos un objeto **DataHandler** al que le damos la fuente de datos. en este caso una imagen que se encuentra almacenada en la unidad

```
// Formamos el texto del mensaje
BodyPart texto = new MimeBodyPart();
texto.setText("Texto del mensaje");

// Formamos el adjunto (imagen) del mensaje
BodyPart adjunto = new MimeBodyPart();
adjunto.setDataHandler(
    new DataHandler(new FileDataSource("d:/imagen.gif")));
adjunto.setFileName("imagen.gif");
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Construcción del mensaje de texto con adjunto:
  - ✓ Ahora juntamos las dos partes en una única parte.
  - ✓ Para ello creamos un objeto **MimeMultipart**.
  - ✓ Con el método *addBodyPart* podemos ir añadiendo al mensaje los archivos que queramos.

```
// Agrupamos texto e imagen.  
MimeMultipart multiParte = new MimeMultipart();  
multiParte.addBodyPart(texto);  
multiParte.addBodyPart(adjunto);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación del mensaje de correo:
  - ✓ Creación de un objeto **MimeMessage** pasándole el objeto Session creado anteriormente.
  - ✓ Configuramos este objeto/mensaje:
    - **FROM** del mensaje: método *setFrom*. La dirección de correo origen se crea con un objeto **InternetAddress**.
    - **TO** del mensaje: método *addRecipient*.  
Este método admite dos parámetros, una constante para indicar el tipo de destinatario y el objeto **InternetAdress** con la dirección de correo destino.  
El tipo de destinatario puede ser:
      - Message.RecipientType.TO*: destinatario principal.
      - Message.RecipientType.CC*: destinatario al que se envía copia del mensaje.
      - Message.RecipientType.BCC*: destinatario al que se envía copia, pero sin que los demás destinatarios puedan verlo.

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Creación del mensaje de correo:
  - ✓ Configuramos este objeto/mensaje:
    - ASUNTO: método *setSubject*.
    - TEXTO + IMAGEN: método *setContent*.

```
// Construimos el mensaje
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress("XXXXXX@hotmail.com"));
message.addRecipient(
    Message.RecipientType.TO,
    new InternetAddress("procesos.daml9@gmail.com"));
message.setSubject("EJEMPLO DE CORREO CON ADJUNTO CON JAVA MAIL");
message.setContent(multiParte);
```

## 9. JAVA PARA COMUNICAR CON UN SERVIDOR DE CORREO ELECTRÓNICO

- Envío del mensaje de correo:
  - ✓ Creación de un objeto `Transport`. Hay que pasarle como parámetro el nombre del protocolo a usar, en nuestro caso para Gmail, SMTP.
  - ✓ Establecemos la conexión con el método `connect`, al que le pasamos la cuenta de correo electrónico y la contraseña.
  - ✓ Mandamos el mensaje con el método `sendMessage`. A este método le pasamos el mensaje y la lista de receptores del mismo. Para ello con el método `getAllRecipients` obtenemos el array de direcciones destinatarias de mensaje.
  - ✓ Finalmente cerramos la conexión con el método `close`.

```
// Lo enviamos.  
Transport t = session.getTransport("smtp");  
t.connect("procesos.daml9@gmail.com", "XXXXXXX");  
t.sendMessage(message, message.getAllRecipients());  
  
//Cierre  
t.close();
```

## 10. PROGRAMACIÓN DE SERVIDORES

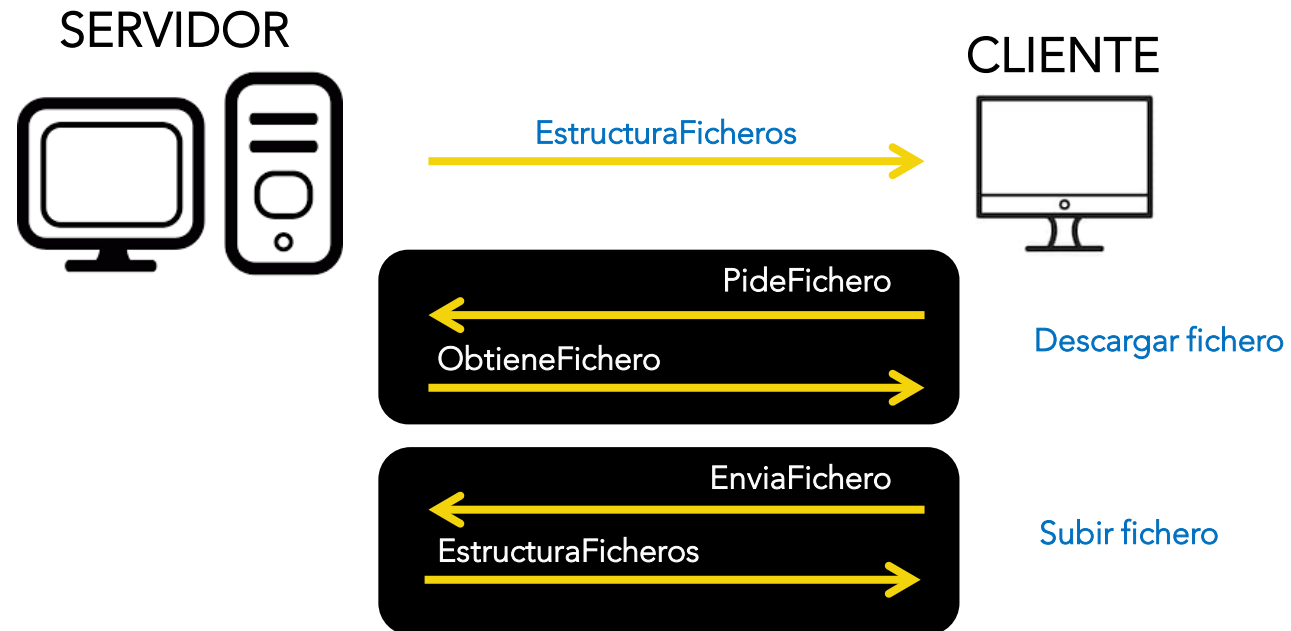
- Un **servidor** no es más que un programa que ofrece sus servicios a otros programas, los clientes.
- Así se crean las aplicaciones llamadas "**cliente/servidor**".
- En estas aplicaciones se determina un protocolo de comunicaciones entre el cliente y el servidor.
- Posteriormente el cliente solicita operaciones al servidor y éste le devuelve resultados.
- Hay que tener en cuenta que el cliente y el servidor:
  - ✓ Pueden estar en sitios distintos y comunicarse a través de protocolos de red.
  - ✓ Pueden estar programados en distintos lenguajes de programación.

## 10. PROGRAMACIÓN DE SERVIDORES

- Lo que vamos a ver a continuación, es un ejemplo de programación de un servidor utilizando Java.
- También programaremos el cliente para completar la funcionalidad.
- **Planteamiento:**
  - ✓ Servidor de ficheros que proporciona acceso a carpetas y ficheros, para su carga y descarga.
- **Protocolo de comunicación:**
  - ✓ El servidor proporciona a los clientes una carpeta a la que tener acceso.
  - ✓ Los clientes se conectan al servidor para cargar/descargar ficheros en esa carpeta.
  - ✓ No podrán navegar por el árbol de carpetas del servidor.
  - ✓ La comunicación cliente-servidor se realiza mediante el intercambio de objetos.

## 10. PROGRAMACIÓN DE SERVIDORES

- El esquema de comunicación entre el servidor y el cliente sería:





## 10. PROGRAMACIÓN DE SERVIDORES

- Explicación del flujo de comunicación:
  - ✓ En el servidor se selecciona el directorio a la que los clientes podrán acceder para subir/descargar ficheros.  
NOTA: Los clientes no podrán navegar por este directorio.
  - ✓ Se establece el puerto de comunicación 1234.
  - ✓ Cuando se conecta el cliente se crea un objeto **EstructuraFicheros** con la información del directorio del servidor al que tendrá acceso.  
El servidor manda al cliente este objeto.

## 10. PROGRAMACIÓN DE SERVIDORES

- Explicación del flujo de comunicación:
  - ✓ **Descargar fichero:** el cliente hace la petición al servidor mandándole un objeto **PideFichero** con el nombre del fichero a descargar. El servidor le contesta enviándole un objeto **ObtieneFichero** con el contenido del fichero en bytes y el tamaño del fichero.
  - ✓ **Subir fichero:** el cliente hace la petición al servidor enviándole un objeto **EnviaFichero**, con el nombre del fichero, contenido en bytes y tamaño. El servidor crea el fichero en el directorio y en respuesta manda al cliente un objeto **EstructuraFicheros** con la nueva estructura del árbol de directorios del directorio del servidor.

# Sesiones 6 y 7. Contenidos teóricos

## 10. PROGRAMACIÓN DE SERVIDORES

Clase	Descripción
EstructuraFicheros	<ul style="list-style-type: none"><li>• Clase que define la estructura del directorio seleccionado en el servidor, con sus directorios y ficheros.</li><li>• Se irá actualizando a medida que los cliente "suban" nuevos ficheros.</li></ul>
Servidor	<ul style="list-style-type: none"><li>• Al arrancar el servidor se selecciona el directorio de trabajo, y se queda a la espera de conexiones de clientes.</li><li>• A cada cliente que se conecte, se le manda un socket y un objeto EstructuraFicheros.</li></ul>

# Sesiones 6 y 7. Contenidos teóricos

## 10. PROGRAMACIÓN DE SERVIDORES

Clase	Descripción
HiloServidor	<ul style="list-style-type: none"><li>• Para cada cliente conectado se crea un objeto de esta clase.</li><li>• En esta clase se define las operaciones de carga y subida de ficheros, que solicita el cliente.</li></ul>
PideFichero y ObtieneFichero	<ul style="list-style-type: none"><li>• Estas clase representan la petición de un cliente de descarga de un fichero del servidor.</li><li>• <b>PideFichero</b> manda al servidor el nombre del fichero a descargar y <b>ObtieneFichero</b> devuelve al cliente el contenido del fichero.</li></ul>
EnviaFichero	<ul style="list-style-type: none"><li>• Clase que modela la petición de subida de un fichero por parte de un cliente. Este objeto envía al servidor el nombre, directorio y contenido del fichero a cargar en el servidor.</li></ul>

# Sesiones 6 y 7. Contenidos teóricos

## 10. PROGRAMACIÓN DE SERVIDORES

Clase	Descripción
HiloCliente	<ul style="list-style-type: none"><li>• Clase que modela a cada cliente.</li><li>• En realidad se trata de un hilo por cliente.</li><li>• JFrame que muestra una lista el contenido del directorio disponible del servidor para subir/descargar fichero, y botones para realizar las operaciones.</li></ul>
AplicacionCliente	<ul style="list-style-type: none"><li>• Clase que lanza cada HiloCliente.</li></ul>

## 10. PROGRAMACIÓN DE SERVIDORES

### EstructuraFicheros

```
public class EstructuraFicheros implements Serializable {  
  
    private String nombre; //nombre del directorio  
    private String path;   //nombre completo  
    private boolean esDir; //indica si es o no un directorio  
    private int numFich;   //número de ficheros que tiene el directorio  
    private EstructuraFicheros[] lista; //array de objetos EstructuraFicheros  
                                     //que contiene los ficheros y subdirectorios  
                                     //del directorio seleccionado  
  
    //Constructor que es llamado desde el servidor.  
    //Recibe como parámetro el directorio seleccionado  
    public EstructuraFicheros(String dir) {  
        File f=new File(dir);  
        this.nombre=f.getName();  
        this.path=f.getPath();  
        this.esDir=f.isDirectory();  
        this.lista=getListaFiles();  
        if(f.isDirectory()){  
            File[] ficheros=f.listFiles();  
            if(ficheros!=null) this.numFich=ficheros.length;  
        }  
    }  
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### EstructuraFicheros

```
//Constructor que es llamado desde el método getListFiles()
public EstructuraFicheros(String nombre, String path, boolean esDir, int numFich) {
    this.nombre = nombre;
    this.path = path;
    this.esDir = esDir;
    this.numFich = numFich;
}

//Getter
public String getNombre() {
    String nombre_dir=this.nombre;
    if(this.esDir){
        int pos=this.path.lastIndexOf(File.separator);
        nombre_dir=this.path.substring(pos+1,this.path.length());
    }
    return nombre_dir;
}

public String getPath() {
    return path;
}

public boolean isEsDir() {
    return esDir;
}

public int getNumFich() {
    return numFich;
}
```

```
public EstructuraFicheros[] getList() {
    return lista;
}

@Override
public String toString() {
    String nombre=this.nombre;
    if(this.esDir) nombre="(DIR) " + this.nombre;
    return nombre;
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### EstructuraFicheros

```
private EstructuraFicheros[] getListaFiles(){
    EstructuraFicheros[] lista=null;
    String sDirectorio=this.path;
    File f=new File(sDirectorio);
    File[] ficheros=f.listFiles();
    int longitud=ficheros.length;
    if(longitud>0){
        lista=new EstructuraFicheros(longitud);
        for(int i=0; i<ficheros.length;i++){
            EstructuraFicheros elemento;
            String nombre=ficheros[i].getName();
            String path=ficheros[i].getPath();
            boolean esDir=ficheros[i].isDirectory();
            int num=0;
            if(esDir){
                File[]fich=ficheros[i].listFiles();
                if(fich!=null) num=fich.length;
            }
            elemento=new EstructuraFicheros(nombre,path,esDir,num);
            lista[i]=elemento;
        } //for
    } //if
    return lista;
}
```



## 10. PROGRAMACIÓN DE SERVIDORES

### Servidor

```
public class Servidor {  
  
    private final int puerto=1234;  
    private EstructuraFicheros NF;  
    private ServerSocket servidor;  
  
    public void conectarServidor(){  
        JFileChooser jfc=new JFileChooser();  
        jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);  
        jfc.setDialogTitle("Selecciona la carpeta donde se encuentran los ficheros");  
        int resp=jfc.showDialog(null, "Seleccionar");  
        File f=null;  
        if(resp==JFileChooser.APPROVE_OPTION){  
            f=jfc.getSelectedFile();  
        }  
        if(f==null){  
            System.out.println("Debe seleccionar una carpeta");  
        }else{  
            try {  
                servidor=new ServerSocket(this.puerto);  
                while(true){  
                    Socket cliente=servidor.accept();  
                    System.out.println("Cliente en linea...");  
                    NF=new EstructuraFicheros(f.getAbsolutePath());  
                    HiloServidor hiloServidor=new HiloServidor(cliente,NF);  
                    Thread hilo=new Thread(hiloServidor);  
                }  
            } catch (IOException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloServidor

```
public class HiloServidor implements Runnable {  
  
    private Socket hiloServidor;  
    private ObjectOutputStream salidaCliente;  
    private ObjectInputStream entradaCliente;  
    private EstructuraFicheros NF;  
  
    public HiloServidor(Socket hiloServidor, EstructuraFicheros NF) throws IOException {  
        this.hiloServidor = hiloServidor;  
        this.NF = NF;  
        this.salidaCliente = new ObjectOutputStream(hiloServidor.getOutputStream());  
        this.entradaCliente = new ObjectInputStream(hiloServidor.getInputStream());  
    }  
  
    private void enviarFichero(PideFichero fich){  
        try {  
            File f=new File(fich.getNombreFichero());  
            FileInputStream fis=null;  
            fis=new FileInputStream(f);  
            long bytes=f.length();  
            byte[]bufer=new byte[(int)bytes];  
            int i=0;  
            int j=0;  
            while((i=fis.read()) != -1){  
                bufer[j]=(byte) i;  
                j++;  
            }  
            fis.close();  
            Object ob=new ObtieneFichero(bufer);  
            this.salidaCliente.writeObject(ob);  
        } catch (FileNotFoundException ex) {  
            Logger.getLogger(HiloServidor.class.getName()).log(Level.SEVERE, null, ex);  
        } catch (IOException ex) {  
            Logger.getLogger(HiloServidor.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloServidor

```
public void run() {
    try {
        //mandamos al cliente el objeto EstructuraFicheros
        salidaCliente.writeObject(NF);
        while (true) {
            //obtengo la entrada del cliente
            Object peticion = entradaCliente.readObject();
            //si el cliente quiere descargar un fichero
            if (peticion instanceof PideFichero) {
                PideFichero fichero = (PideFichero) peticion;
                enviarFichero(fichero);
            }
            //si el cliente quiere subir un fichero
            if (peticion instanceof EnviaFichero) {
                EnviaFichero fichero = (EnviaFichero) peticion;
                File dir = new File(fichero.getDirectorio());
                File f = new File(dir, fichero.getNombre());

                //se crea el fichero en el directorio
                //con los bytes transferidos
                FileOutputStream fos = new FileOutputStream(f);
                fos.write(fichero.getContenidoFichero());
                fos.close();

                //se crea la nueva estructura de ficheros
                EstructuraFicheros ef=new EstructuraFicheros(fichero.getDirectorio());
                salidaCliente.writeObject(ef);
            }
        }
    } catch (IOException ex) {
        Logger.getLogger(HiloServidor.class.getName()).log(Level.SEVERE, null, ex);
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(HiloServidor.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

- En el método *run()* se manda al cliente un objeto **EstructuraFicheros** con el arbol de directorios del servidor.
- Después se comprueba la petición del cliente: subida o descarga de fichero. Éstas llegan como objetos **PideFichero** o **EnviaFichero**.

## 10. PROGRAMACIÓN DE SERVIDORES

### PideFichero

```
public class PideFichero implements Serializable {  
    private String nombreFichero;  
  
    public PideFichero(String nombreFichero) {  
        this.nombreFichero = nombreFichero;  
    }  
  
    public String getNombreFichero() {  
        return nombreFichero;  
    }  
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### ObtieneFichero

```
public class ObtieneFichero implements Serializable {  
  
    private byte[] contenidoFichero;  
  
    public ObtieneFichero(byte[] contenidoFichero) {  
        this.contenidoFichero = contenidoFichero;  
    }  
  
    public byte[] getContenidoFichero() {  
        return contenidoFichero;  
    }  
}
```

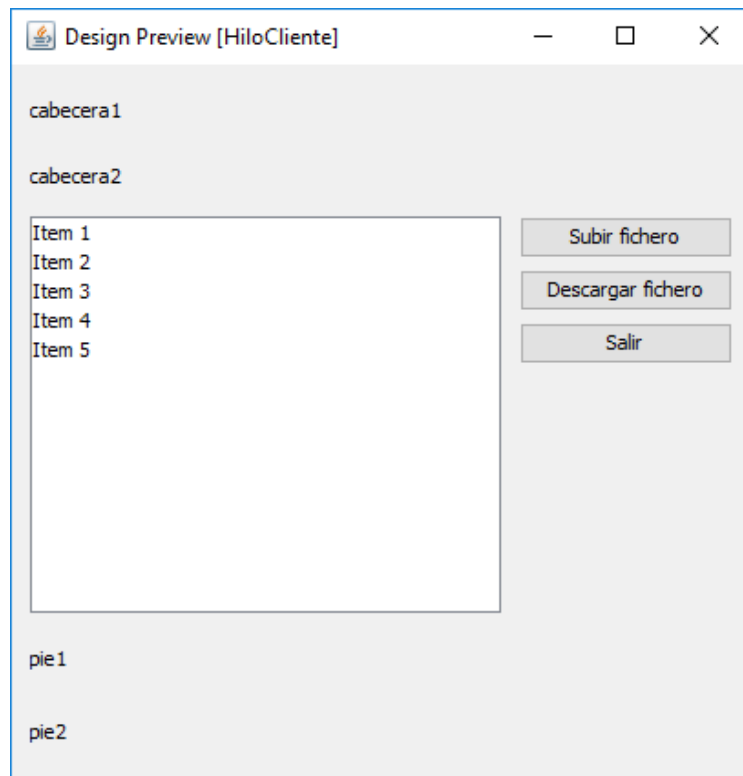
## 10. PROGRAMACIÓN DE SERVIDORES

### EnviaFichero

```
public class EnviaFichero implements Serializable {  
  
    byte[] contenidoFichero;  
    String nombre;  
    String directorio;  
  
    public EnviaFichero(byte[] contenidoFichero, String nombre, String directorio) {  
        this.contenidoFichero = contenidoFichero;  
        this.nombre = nombre;  
        this.directorio = directorio;  
    }  
  
    public byte[] getContenidoFichero() {  
        return contenidoFichero;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public String getDirectorio() {  
        return directorio;  
    }  
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente



```
public class HiloCliente extends javax.swing.JFrame implements Runnable {

    private Socket cliente; //Socket del cliente
    private EstructuraFicheros nodo; //objeto actual
    private EstructuraFicheros raiz; //raiz
    private ObjectOutputStream salidaServidor;
    private ObjectInputStream entradaServidor;
    private String dirSelec;
    private String fichSelec;
    private String pathSelec;

    public HiloCliente(Socket hilo) throws IOException {
        initComponents();
        setFrame();
        this.cliente = hilo;
        this.salidaServidor = new ObjectOutputStream(cliente.getOutputStream());
        this.entradaServidor = new ObjectInputStream(cliente.getInputStream());
    }
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente

```
@Override
public void run() {
    try {
        this.cabecera1.setText("CONECTANDO CON EL SERVIDOR.....");
        this.raiz = (EstructuraFicheros) entradaServidor.readObject();
        EstructuraFicheros[] dir = this.raiz.getList();
        this.dirSelec = this.raiz.getPath();
        llenarLista(dir, this.raiz.getNumFich());
        this.cabecera2.setText("RAIZ: " + this.dirSelec);
        this.cabecera1.setText("CONECTADO AL SERVIDOR DE FICHEROS");
        this.pie2.setText("Nº de ficheros en el directorio: " +
            this.raiz.getNumFich());
    } catch (IOException | ClassNotFoundException ex) {
        ex.printStackTrace();
    }
}
```

- En el método *run()* se lee el objeto **EstructuraFicheros** que manda el servidor, se carga en un array de nodos y se manda al método *llenarLista()* que lo que hace es llenar el **Jlist** con objetos **EstructuraFicheros**, que representan ficheros o directorios, para mostrarlos por pantalla.



## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente

```
private void listaValueChanged(javax.swing.event.ListSelectionEvent evt) {  
    if (this.lista.getValueIsAdjusting()) {  
        this.nodo = (EstructuraFicheros) this.lista.getSelectedValue();  
        if (nodo.isEsDir()) {  
            this.piel.setText("FUNCIÓN NO IMPLEMENTADA.....");  
        } else {  
            this.fichSelec = nodo.getNombre();  
            this.pathSelec = nodo.getPath();  
            this.piel.setText("FICHERO SELECCIONADO: " + this.fichSelec);  
        }  
    }  
}
```

- Al seleccionar un elemento del Jlist, se comprueba si es un directorio o un fichero.
- En el caso de ser un directorio no se hace nada, solo se muestra el mensaje "Función no implementada."
- Si se trata de un fichero, se actualizan las variables de clase para tener acceso al fichero seleccionado.

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente

```
private void subirActionPerformed(java.awt.event.ActionEvent evt) {  
    JFileChooser jfc = new JFileChooser();  
    jfc.setFileSelectionMode(JFileChooser.FILES_ONLY);  
    jfc.setDialogTitle("Selecciona el fichero a subir al servidor");  
    int resp = jfc.showDialog(this, "Subir");  
    if (resp == JFileChooser.APPROVE_OPTION) {  
        File f = jfc.getSelectedFile();  
        String nombreArchivo = f.getName();  
        BufferedInputStream bis;  
        try {  
            bis = new BufferedInputStream(new FileInputStream(f.getAbsolutePath()));  
            byte[] buffer = new byte[(int) f.length()];  
            int i, j = 0;  
            while ((i = bis.read()) != -1) {  
                buffer[j] = (byte) i;  
                j++;  
            }  
            bis.close();  
        }  
    }  
}
```

- Al pulsar el botón Subir fichero el cliente elegirá el fichero a subir al servidor a través de un JFileChooser.
- Éste será modelado a través de un objeto EnviaFichero.
- El servidor responde con la estructura del directorio del servidor actualizada, mandando un objeto EstructuraFicheros que lo representa.

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente

```
Object ob = new EnviaFichero(buffer, nombreArchivo, dirSelec);
this.salidaServidor.writeObject(ob);
JOptionPane.showMessageDialog(null, "FICHERO SUBIDO");

this.nodo = (EstructuraFicheros) this.entradaServidor.readObject();
EstructuraFicheros[] dir = this.nodo.getList();
this.dirSelec = this.nodo.getPath();
llenarLista(dir, this.nodo.getNumFich());
this.pie2.setText("Nº de ficheros en el directorio: " + dir.length);

} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (IOException ex) {
    Logger.getLogger(HiloCliente.class.getName()).log(Level.SEVERE, null, ex);
} catch (ClassNotFoundException ex) {
    Logger.getLogger(HiloCliente.class.getName()).log(Level.SEVERE, null, ex);
}
}
```

- Para mostrar la estructura actualiza en el Jlist se llama al método *llenarLista()*.

```
private void llenarLista(EstructuraFicheros[] files, int num) {
    if (num == 0) {
        return;
    }
    DefaultListModel modeloLista = new DefaultListModel();
    this.lista.removeAll();
    for (EstructuraFicheros file : files) {
        modeloLista.addElement(file);
    }
    this.lista.setModel(modeloLista);
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente

```
private void descargarActionPerformed(java.awt.event.ActionEvent evt) {  
    if (this.pathSelec.equals("")) {  
        return; //no se ha seleccionado nada  
    }  
    try {  
        //Sino, pedimos ese fichero  
        PideFichero pidoFich = new PideFichero((this.pathSelec));  
        salidaServidor.writeObject(pidoFich);  
        //creamos un fichero con el nombreseleccionado en el dir. actual  
        FileOutputStream fos=new FileOutputStream(this.fichSelec);  
        //recibo el fichero del servidor  
        Object obtengoFich=entradaServidor.readObject();  
        if(obtengoFich instanceof ObtieneFichero){  
            ObtieneFichero fic=(ObtieneFichero) obtengoFich;  
            fos.write(fic.getContenidoFichero()); //escribo los bytes del  
            fos.close();  
            JOptionPane.showMessageDialog(null, "FICHERO DESCARGADO");  
        }  
    } catch (IOException ex) {  
        ex.printStackTrace();  
    } catch (ClassNotFoundException ex) {  
        ex.printStackTrace();  
    }  
}
```

- Al pulsar el botón Descargar fichero se crea un objeto PideFichero con el nombre del fichero a descargar, que se corresponde con el seleccionado en el JList.
- Copiamos el fichero que recibimos a través del socket del servidor, modelado como un objeto ObtieneFichero, en nuestro directorio de trabajo actual, con el mismo nombre que el seleccionado del JList.

## 10. PROGRAMACIÓN DE SERVIDORES

### HiloCliente

- El botón Salir finaliza el programa.

```
private void salirActionPerformed(java.awt.event.ActionEvent evt) {  
    try {  
        this.cliente.close();  
        System.exit(0);  
    } catch (IOException ex) {  
        Logger.getLogger(HiloCliente.class.getName()).log(Level.SEVERE, null, ex);  
    }  
}
```

## 10. PROGRAMACIÓN DE SERVIDORES

### AplicacionCliente

```
public class AplicacionCliente {  
  
    private static final int PUERTO = 1234; //Puerto para la conexión  
    private static final String HOST = "localhost"; //Host para la conexión  
  
    public static void main(String[] args) {  
        try {  
            Socket s=new Socket(HOST,PUERTO);  
            HiloCliente hiloCliente=new HiloCliente(s);  
            Thread hilo=new Thread(hiloCliente);  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

- Lanza un **HiloCliente**, de tal forma que este momento solo hay un cliente conectado al servidor de ficheros.
- Se podría hacer que varios clientes estuvieran conectados lanzando tantos hilos como fuera necesario.