# TEMA 4. GENERACIÓN DE SERVICIOS EN RED

PROGRAMACION DE SERVICIOS Y PROCESOS.

2023-2024

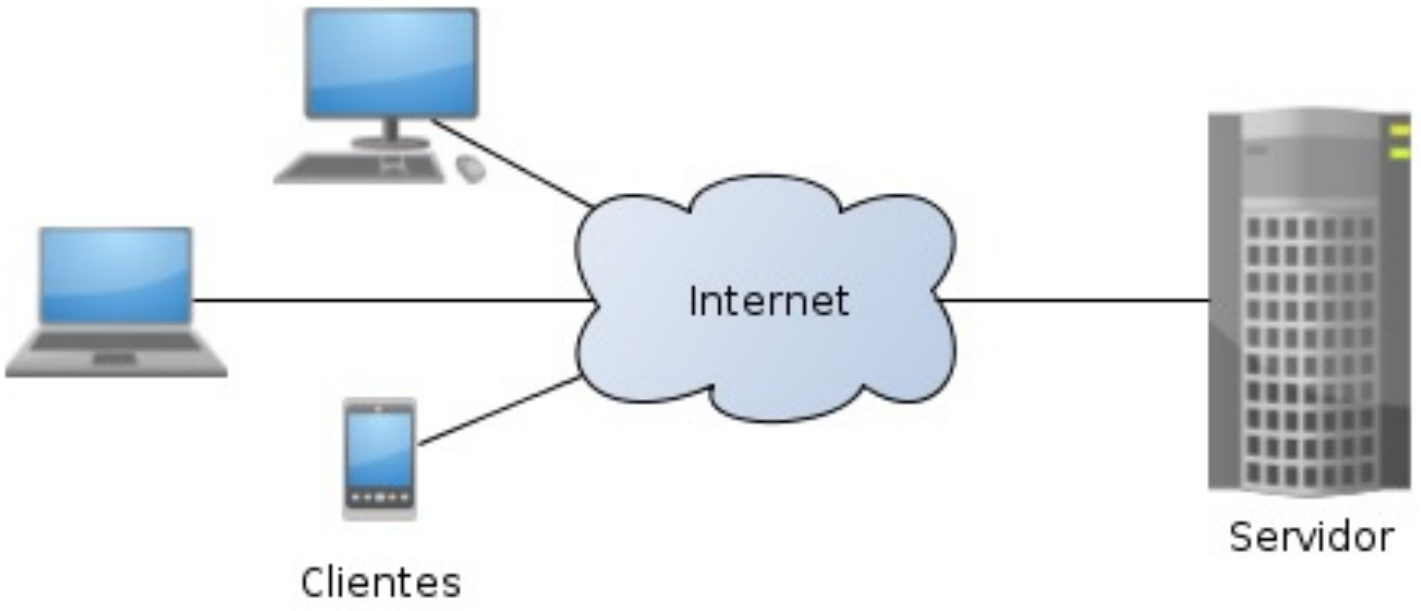
# Índice

# 1. Introducción

- **Definición de servicio**
  - Un servicio es un programa auxiliar que gestiona recursos.
  - El servicio presta funcionalidades a otras aplicaciones o a los usuarios.
  - El único acceso que tenemos a un servicio lo conforman las operaciones que este ofrece.
    - Operaciones de lectura/escritura, transferencia, borrado, etc.
  - Los servicios de Internet implementan una relación cliente-servidor.
  - Los servicios están implementados en la actualidad siguiendo diversos protocolos estándar.

# 1. Introducción

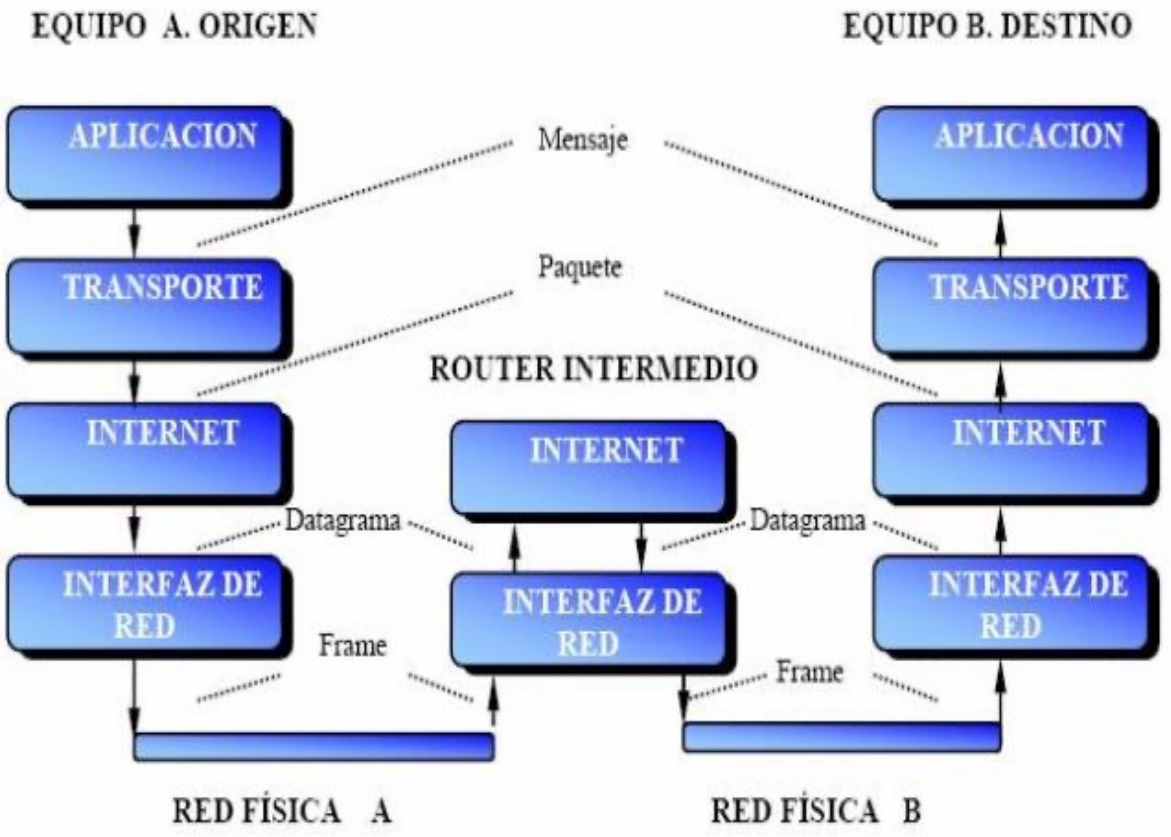
- La red **cliente-servidor**



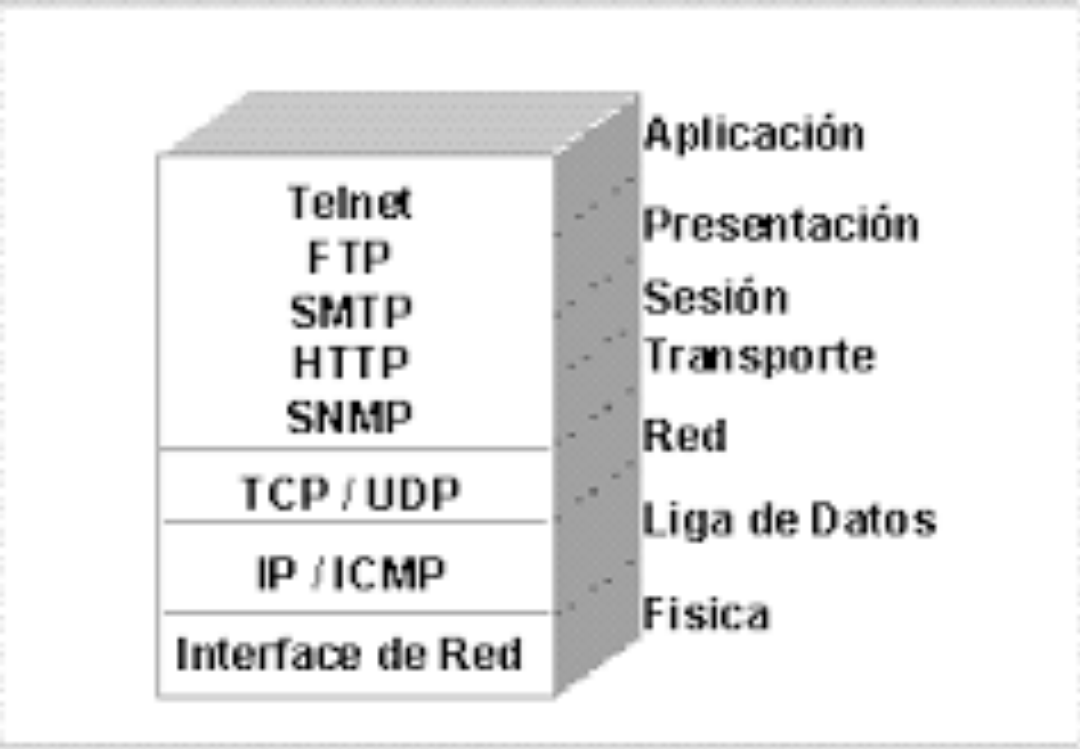
# 2. Clasificación de los servicios por su funcionalidad

- Administración/Configuración.
  - Ejemplos: DHCP y DNS.
- Acceso y control remoto
  - Ejemplos: Telnet y SSH.
- De ficheros
  - Ejemplo: FTP
- De impresión
- Información
  - Ejemplos: HTTP y servidores de BBDD.
- Comunicación
  - Ejemplo: Correo electrónico (SMTP y POP3).

# 3. Comunicación entre aplicaciones

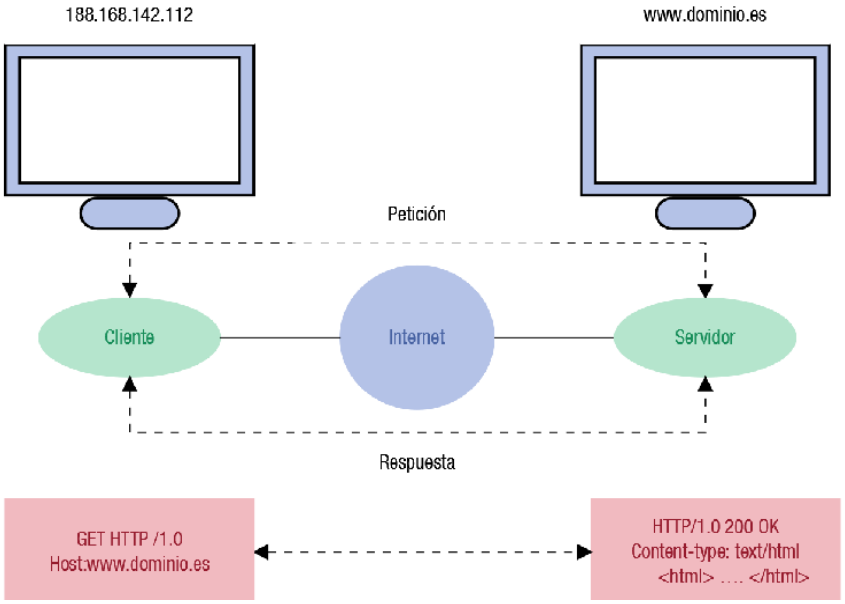



# 4. Protocolos estándar de comunicación a nivel de aplicación



# 4.1 Protocolo HTTP

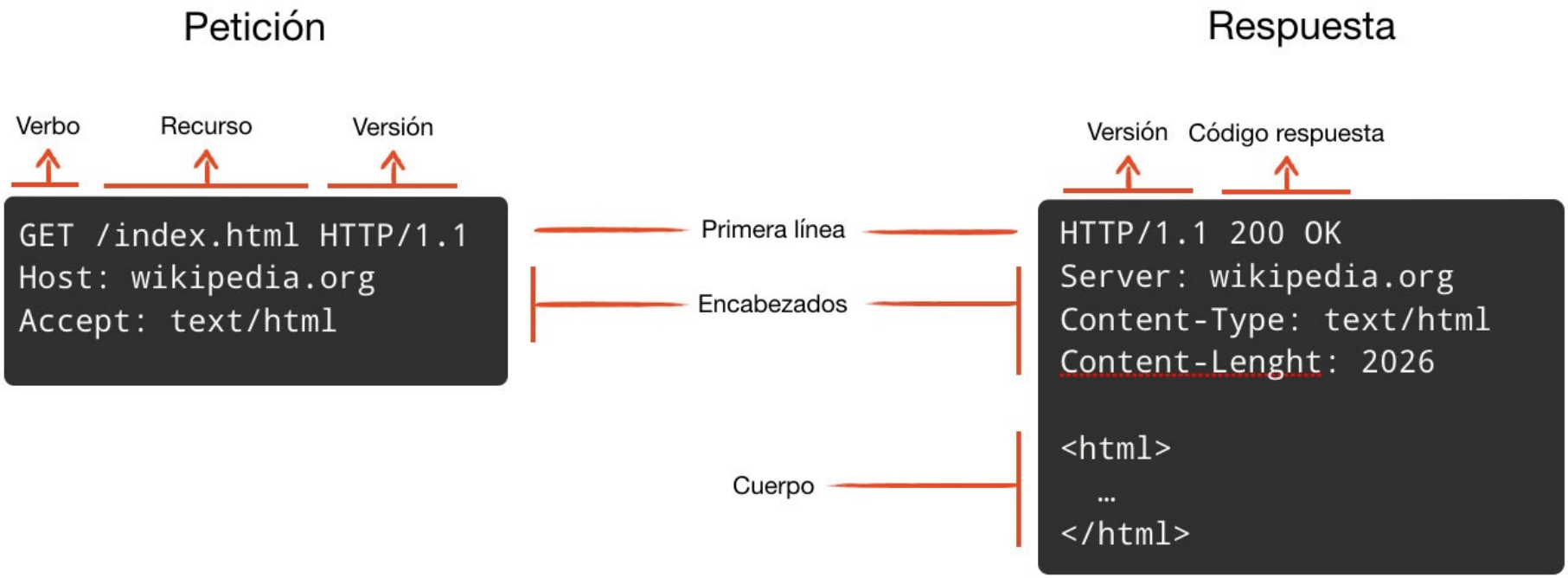
- El protocolo HTTP o *Protocolo de Transferencia de Hipertexto* es un conjunto de normas que permiten que una aplicación en un cliente y otra en un servidor (servidor web) se comuniquen e intercambien información en formato HTML.
- El servidor HTTP escucha en el puerto **80** por defecto o **443** si se utiliza el protocolo HTTPS (HTTP seguro).
- La información transmitida se identifica mediante una **URL** (Localizador Uniforme de Recursos).



# 4.1 Protocolo HTTP

- La mayoría de implementaciones d este protocolo se han realizado sobre TCP.
- La comunicación se estructura siempre en dos fases: petición (la hacen los clientes) y respuesta (la hacen los servidores).
- Tanto las peticiones como las respuestas son mensajes de texto estructurados en dos bloques y separados entre ellos con una línea.

# 4.1 Protocolo HTTP



# 4.1 Protocolo HTTP

- **Métodos de envío de datos**
  - GET: Solicita un documento al servidor. Se pueden enviar datos en la URL.
  - HEAD: Similar a GET, pero sólo pide las cabeceras HTTP.
  - POST: Manda datos al servidor para su procesado. Similar a GET, pero además envía datos en el cuerpo del mensaje.
  - PUT: Almacena el documento enviado en el cuerpo del mensaje.
  - DELETE: Elimina el documento referenciado en la URL.
  - …

# 4.1 Protocolo HTTP

- Código de estados
  - **1xx**: Mensaje informativo.
  - **2xx:** Exito
    - 200 OK
    - 201 Created
    - 202 Accepted
    - 204 No Content
  - **3xx:** Redirección
    - 300 Multiple Choice
    - 301 Moved Permanently
    - 302 Found
    - 304 Not Modified
  - **4xx:** Error del cliente
    - 400 Bad Request
    - 401 Unauthorized
    - 403 Forbidden
    - 404 Not Found
  - **5xx:** Error del servidor
    - 500 Internal Server Error
    - 501 Not Implemented
    - 502 Bad Gateway
    - 503 Service Unavailable

# 4.1 Protocolo HTTP

- **Cabeceras**
  - Server: indica el tipo de servidor HTTP empleado.
  - Age: indica el tiempo que ha estado el objeto servido almacenado en un proxy cache (segundos).
  - Content-Encoding: se indica el tipo de codificación empleado en la respuesta.
  - Location: usado para especificar una nueva ubicación en casos de redirecciones.
  - Set-Cookie: sirve para crear cookies.

# 4.1.2 Programación con URL

- La programación de URL se produce a un nivel más alto de abstracción que la programación de sockets y esto facilita la codificación de aplicaciones que acceden a recursos en una red.
- Un recurso puede ser desde un archivo o un directorio hasta una referencia a un objeto más complicado, como una consulta a una base de datos, el resultado de la ejecución de un programa, etc.
- La estructura de una URL se puede dividir en varias partes:
  - Protocolo://nombrehost(:puerto)ruta(#referencia)
- Ejemplo:

http://aglinformatica.es:6080/citymap/index.php?option=eventos

# 4.1.2 Programación con URL (en desuso)

- Constructores
  - public URL(String spec)
    Por ejemplo: URL direccion = new URL("http://www.it.uc3m.es");
  - public URL(String protocol, String host, String file)
    Por ejemplo: URL direccion = new URL("http","www.it.uc3m.es","index.html");
  - public URL(String protocol, String host, int port, String file)
    Por ejemplo: URL direccion = new URL("http","www.it.uc3m.es", 80, "index.html");

# 4.1.2 Programación con URL (en desuso)

- Métodos para obtener campos de una URL
  - String getProtocol()
  - String getHost()
  - int getPort()
  - String getFile()

# 4.1.2 Programación con URL (en desuso)

- El método InputStream openStream() nos permite recuperar los datos del recurso representado por la URL. Este método nos devuelve un InputStream del cual podremos leer el contenido del recurso que identifica la URL.

- El método URLConnection openConnection() devuelve objeto URLConnection que representa una nueva conexión al recurso remoto referido en la URL.
- El objeto **URLConnection** para obtener información sobre la cabecera del mensaje de respuesta del servidor.
- A partir de este objeto se puede obtener también un InputStream para leer el recurso a través del método InputStream getInputStream()

# 4.1.3 Programación con HttpClient

- Esta clase apareció a partir del JDK 11.
- La API *HttpURLConnection* tenía numerosos problemas:
  - Complejidad y verbosidad.
  - Poca flexibilidad en cuanto a configuraciones.
  - Manejo de redirecciones manual.
  - Manejo de cookies manual.
  - Manejo difícil de peticiones HTTP asíncronas.
- La API ***HttpClient*** es más moderna, robusta y fácil de usar.
- Además, ofrece mecanismos de solicitud **síncronos** y **asíncronos**.

# 4.1.3 HTTP Descripción general

- La API consiste en tres clases principales:
  - *HttpRequest*: representa una solicitud que va a ser enviada vía *HttpClient*.
  - *HttpClient*: se comporta como un contenedor de información de configuración común a múltiples solicitudes.
  - *HttpResponse*: representa el resultado de una llamada a *HttpRequest*.

- APIs oficiales de consulta:
  - https://docs.oracle.com/en/java/javase/17/docs/api/java.net.http/java/net/http/HttpRequest.html
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html
  - https://docs.oracle.com/en/java/javase/17/docs/api/java.net.http/java/net/http/HttpResponse.html

# 4.1.3 HttpRequest

- Esta tiene métodos para configurar una solicitud HTTP.
- Las configuraciones más comunes de una solicitud son:
  - Establecer una URL (método *uri(uri unaURL)*).
  - Establecer el método de envío de datos
    - GET().
    - POST(BodyPublisher body)
    - PUT(BodyPublisher body)
    - DELETE()

# 4.1.3 HttpRequest

- Ejemplo:

```
HttpRequest request= HttpRequest.newBuilder()
        .uri(new URI( str: "https://www.google.es"))
        .GET()
        .build();
```

- Esta solicitud tiene todos los parámetros requeridos por *HttpClient*.
- Sin embargo, algunas veces necesitamos añadir parámetros adicionales como:
  - La versión del protocolo.
  - Cabeceras de solicitud.
  - Tiempo de espera máximo.

# 4.1.3 HttpRequest

- Ejemplo de request GET compleja:

```
HttpRequest request= HttpRequest.newBuilder()
        .uri(new URI( str: "https://postman-echo.com/get"))
        .version(HttpClient.Version.HTTP_2)
        .header( name: "key1",  value: "value2")
        .header( name: "key2",  value: "value2")
        .timeout(Duration.of( amount: 10, SECONDS))
        .GET()
        .build();
```

# 4.1.3 HttpRequest

- Para los métodos POST(BodyPublisher body) y PUT(BodyPublisher body) podemos añadir un *body* (cuerpo de la petición HTTP).
- Esta API provee varias implementaciones de ***BodyPublisher*** para especificar la forma de publicar el cuerpo (body) enviado al servidor:
  - HttpRequest.BodyPublishers.ofString
  - HttpRequest.BodyPublishers.ofInputStream
  - HttpRequest.BodyPublishers.ofByteArray
  - HttpRequest.BodyPublishers.ofFile
  - HttpRequest.BodyPublishers.noBody

# 4.1.3 HttpRequest

- Ejemplo de *request* con POST y un *body* tipo String

```
HttpRequest request = HttpRequest.newBuilder()
        .uri(new URI( str: "https://postman-echo.com/post"))
        .headers("Content-Type", "text/plain;charset=UTF-8")
        .POST(HttpRequest.BodyPublishers.ofString( body: "Ejemplo de POST con body"))
        .build();
```

# 4.1.3 HttpClient

- Todas las solicitudes son enviadas usando *HttpClient*.
- Un objeto HttpClient se crea con la siguiente línea de código:

```
HttpClient client = HttpClient.newHttpClient();
```

- De manera similar a similar a los BodyPublishers, tenemos métodos para manejar los *body* de respuesta HTTP.

# 4.1.3 HttpClient

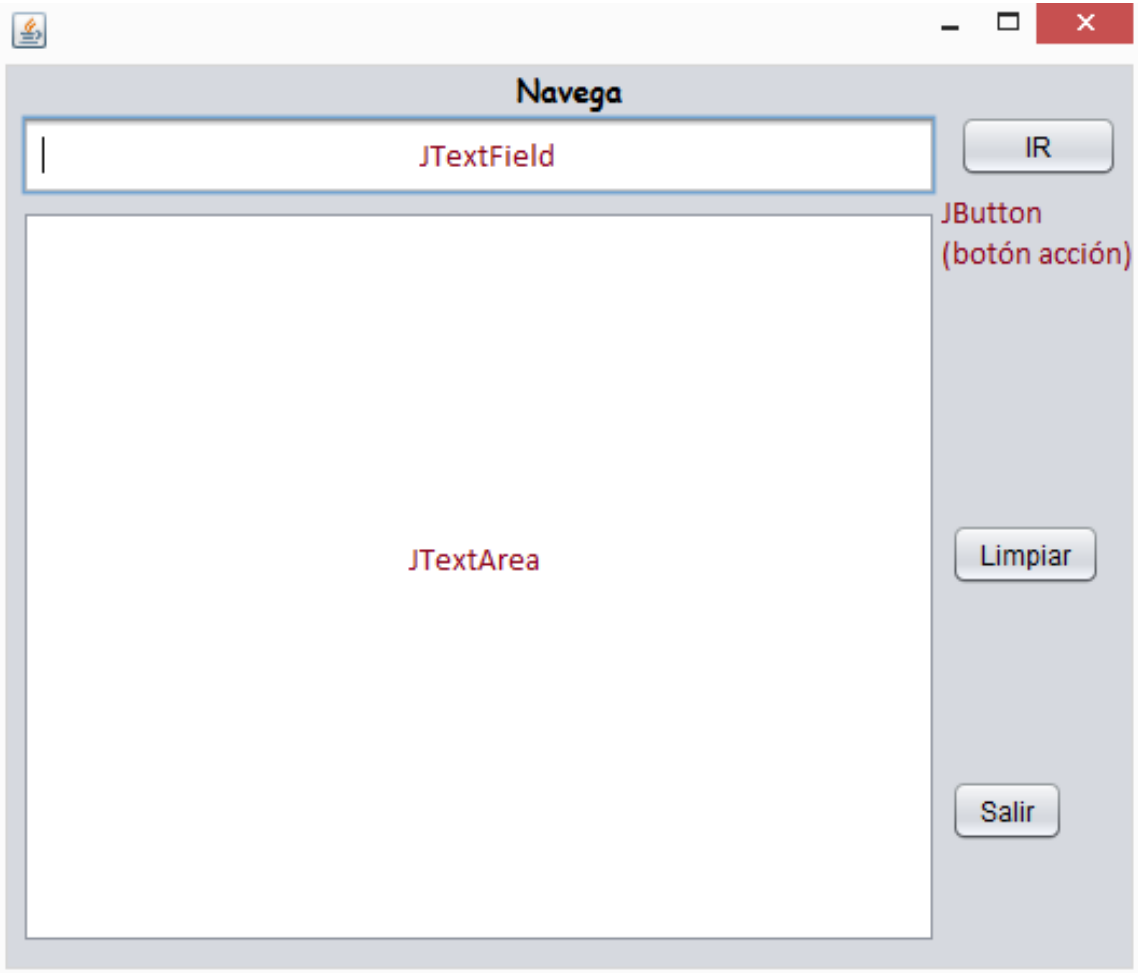
- Manejadores de respuestas posibles:
  - HttpResponse.BodyHandlers.ofByteArray
  - HttpResponse.BodyHandlers.ofString
  - HttpResponse.BodyHandlers.ofFile.
  - …
- El método ***send*** sirve para enviar una request previamente creada al servidor**.**

```
HttpResponse<String> response= client.send(request, HttpResponse.BodyHandlers.ofString());
```

# 4.1.3 HttpResponse

- Esta clase representa la respuesta del servidor y provee varios métodos útiles pero dos son los más importantes:
  - *statusCode*(): el valor de estado de la respuesta como un tipo *int*.
  - *body*(): retorna el body de la respuesta (el tipo depende del parámetro *BodyHandler* pasado al método *send*().
- Otros métodos importante son: *uri*(), *headers*(), *version*().

# 4.1.3 Programación de un cliente HTTP



# 4.1.4 Programación de un servidor HTTP

- El servidor debe atender a multitud de peticiones de clientes.
- Son importantes los tiempos de respuesta del servidor.
- Hay que conocer bien el protocolo.
- La definición de los protocolos de Internet están definidos en los RFCs. Su gestión se realiza a través del IETF.
- El RFC de HTTP ES EL **RFC 2616**.

# 4.1.4 Programación de un servidor HTTP

- Ejemplo para la URL http://www.example.com/index.html
  - El cliente envía

  - El servidor responde

GET /index.html HTTP/1.1

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221

<html lang="eo">
<head>
<meta charset="utf-8">
<title>Título del sitio</title>
</head>
<body>
<h1>Página principal de tuHost</h1>
(Contenido)
  .
```

# 4.1.5 Servicios REST

- REST es un estilo arquitectónico para servicios que utiliza estándares web. Sus principios fundamentales son:
  - Todo recurso puede ser identificado por una URI.
  - Un recurso puede ser representado en múltiples formatos, definido por un *media type*.
  - Se usan métodos HTTP estándar para interactuar con el recurso: principalmente GET, POST, PUST y DELETE.
  - La comunicación entre el cliente y el *endpoint* es sin estado.

# 4.1.5 Servicios REST (JSON)

- REST puede devolver los resultados con distintos tipos de archivo: XML, HTML o JavaScript Object Notation (JSON).
- JSON: es un formato de texto pensado para el intercambio de datos.
- Para ver más información sobre su formato:
  - https://www.mclibre.org/consultar/informatica/lecciones/formato-json.html

# 4.1.5 Servicios REST (JSON)

- Algunos ejemplos de páginas que nos permite obtener información en el formato JSON son:
  - Open Movie Database: https://www.omdbapi.com
  - El País- Consulta de la lotería de Navidad: https://servicios.elpais.com/sorteos/loteria-navidad/api/
  - Agencia estatal de datos de Meteorología (AEMET): https://www.aemet.es/es/datos_abiertos/AEMET_OpenData
- Para trabajar con JSON existen muchas librerías que se pueden importar con Maven para Java:
  - https://www.baeldung.com/java-json#jackson
  - https://www.baeldung.com/java-org-json
  - https://www.baeldung.com/java-json#gson

# 4.1.5 Servicios REST (JSON)

- Los pasos para usar Jackson son:
  - Importar la dependencia Maven.

```xml
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.14.2</version>
</dependency>
```

  - Crear un Objeto ObjectMapper.

```java
// Create a new object mapper
ObjectMapper objectMapper = new ObjectMapper();
```

  - Leer el JSON de la URL.

```java
// Read the JSON data from the URL
JsonNode root = objectMapper.readTree(new URL(url));
```

  - Trabajar con los datos

# 4.1.5 Servicios REST (JSON)

- Varios tutoriales sobre la API Jackson
  - https://github.com/FasterXML/jackson-docs

# 4.2 DNS y resolución de nombres

- Es un servicio que traduce los nombres de dominios aptos para la lectura humana a direcciones ip aptas para lectura por parte de las máquinas.

# 4.2 DNS y resolución de nombres

- La clase **InetAddress** se usa para encapsular tanto la dirección IP numérica como el nombre de dominio para esa dirección.
- Dos subclases
  - Inet4Address para IPV4
  - Inet6Address para IPv6

# 4.2 DNS y resolución de nombres

| | |
|---|---|
| **InetAddress getLocalHost ()** | Devuelve objeto InetAddres que representa la IP de la máquina donde se está ejecutando. |
| **InetAddress getByName (String host)** | Devuelve objeto InetAddres que representa la IP del host pasado como parámetro.<br>String host puede ser: IP, nombre maquina o nombre de un dominio |
| **InetAddress getAllByName (String host)** | Devuelve array de objetos InetAddress.<br><br>Sirve para averiguar las direcciones IP que tiene asignadas la máquina host pasada como parámetro. |
| **String getHostAddress ()** | Devuelve la IP de un objeto InetAddress en formato cadena o String |
| **String getHostName ()** | Devuelve el nombre del host de un objeto InetAddress |
| **String getCanonicalHostName()** | Obtiene el nombre canónico completa(dirección real del host) de un objeto InetAddress |

# 4.3 Protocolo FTP

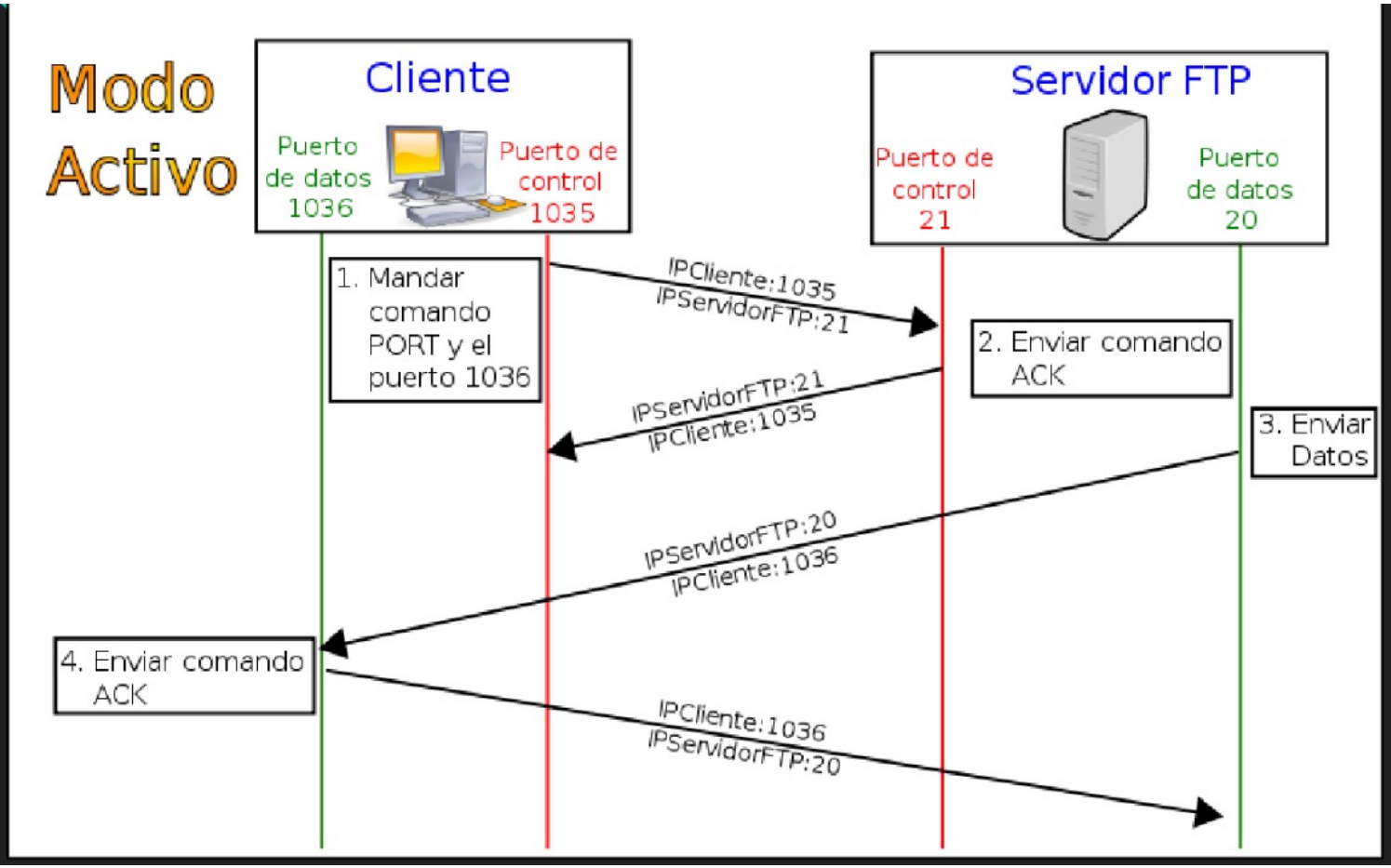
- Este servicio, tal como indica su nombre, permite gestionar la transferencia de ficheros entre dos lugares situados en diferentes dispositivos.
- FTP (File Transport Protocol), se encuentra especificado en el **RFC 959**.
- El protocolo corre sobre TCP.
- FTP es que trabaja sobre dos puertos distintos: el puerto de datos (**20** en modo activo o aleatorio en modo pasivo) y el puerto de comandos/control (**21**):


- Existen **versiones seguras** del protocolo **FTPS** (over SSL) y **SFTP** (con SSH).

# 4.3 Protocolo FTP

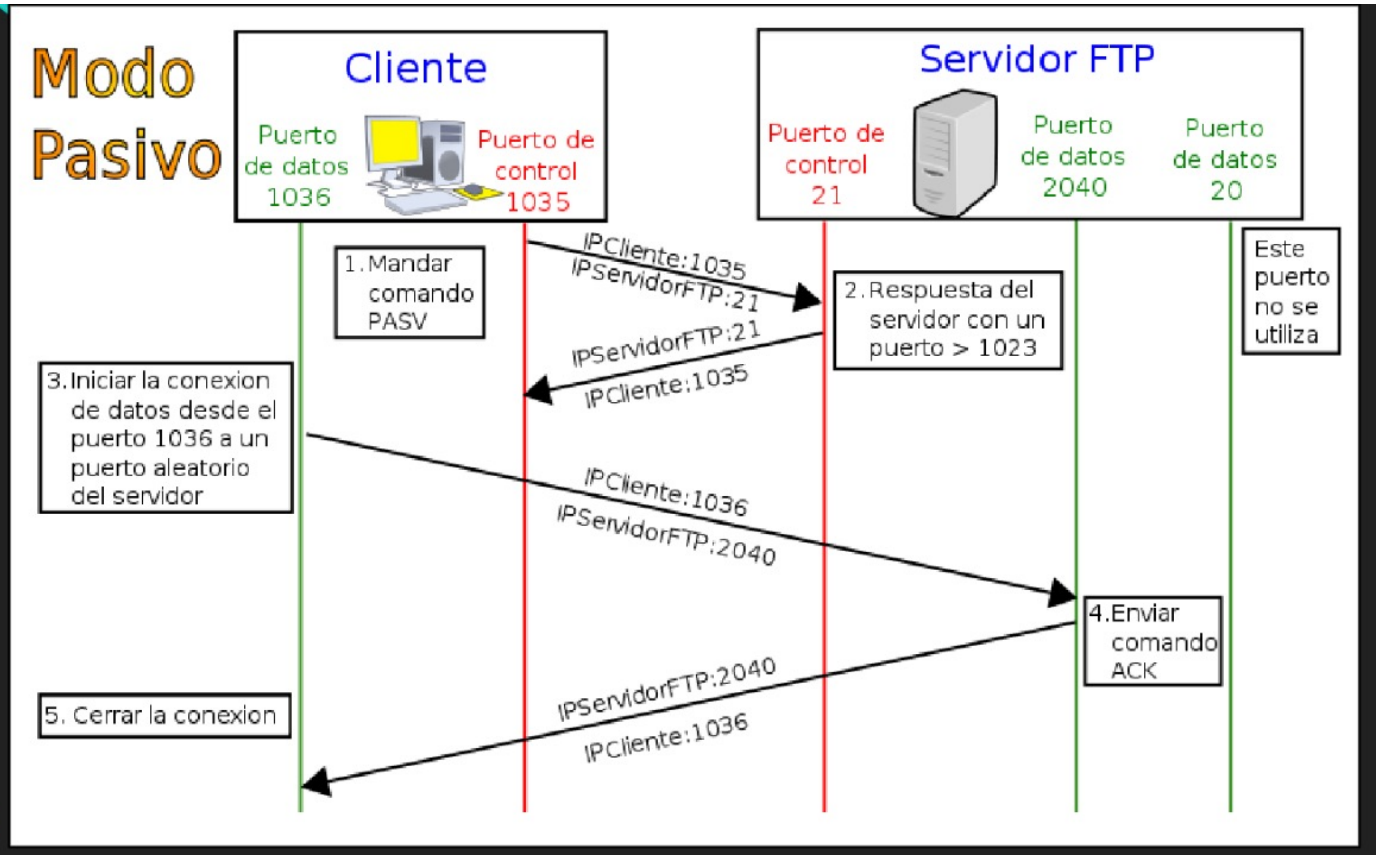
- **Modo activo (PORT):** el servidor inicia la conexión de datos y el firewall del cliente puede bloquearla. El puerto de datos por defecto es el 20.
- **Modo pasivo (PASV):** es el cliente quien inicia la conexión de datos. El servidor tiene un rango de puertos superior a 1023 al cual el cliente se puede conectar.

# 4.3 Protocolo FTP




Fuente: Gabiwxp

# 4.3 Protocolo FTP




Fuente: **Gabiwxp**

# 4.3 Protocolo FTP

| Comando | Parámetros | Acción |
|---|---|---|
| **PWD** | | Nombre del directorio actual. |
| **CWD** | Nombre_directorio | Cambia al directorio indicado como parámetro. |
| **PORT** | Byte_1_IP,<br>Byte_2_IP,<br>Byte_1_port,<br>Byte_2_port | Se quiere iniciar una conexión en modo activo a la IP y puertos indicados. |
| **PASV** | | Se quiere iniciar una transferencia en modo pasivo. |
| **LIST** | | Envía por el canal de datos la lista de ficheros del directorio actual y sus atributos. |
| **STOR** | Nombre_fichero | Almacenará los datos recibidos a través del canal de datos en un fichero que se llamará como indica el parámetro. |
| **RETR** | Nombre fichero remoto | Envía por el canal de datos el fichero remoto. |
| **DELE** | Nombre fichero | Elimina el fichero indicado por parámetro. |
| **QUIT** | | Se abandona el servicio y se cierra la conexión. |

# 4.3.1 Bibliotecas para programar un cliente FTP

- ***Apache Software Foundation*** ofrece un API para programar clientes de algunos protocolos.
- El paquete para trabajar con FTP es: **org.apache.commons.net.ftp**.
- Muy interesante ver la documentación oficial de la librería:
  - https://commons.apache.org/proper/commons-net/apidocs/index.html
- Dependencia Maven

```
<dependency>
    <groupId>commons-net</groupId>
    <artifactId>commons-net</artifactId>
    <version>3.9.0</version>
</dependency>
```

# 4.3.1 Bibliotecas para programar un cliente FTP

- **Clase FTP**. Proporciona las funcionalidades básicas para implementar un cliente FTP. Esta clase hereda de SocketClient.
- **Clase FTPReply**. Permite almacenar los valores devueltos por el servidor como código de respuesta a las peticiones del cliente.
- **Clase FTPClient**. Encapsula toda la funcionalidad que necesitamos para almacenar y recuperar archivos de un servidor FTP, encargándose de todos los detalles de bajo nivel para su interacción con el servidor. Esta clase hereda de SocketClient.
- **Clase FTPClientConfig**. Proporciona una forma alternativa de configurar objetos FTPClient.
- **Clase FTPSClient**. Proporciona FTP seguro, sobre SSL. Esta clase hereda de FTPClient.

# 4.3.1 Bibliotecas para programar un cliente FTP

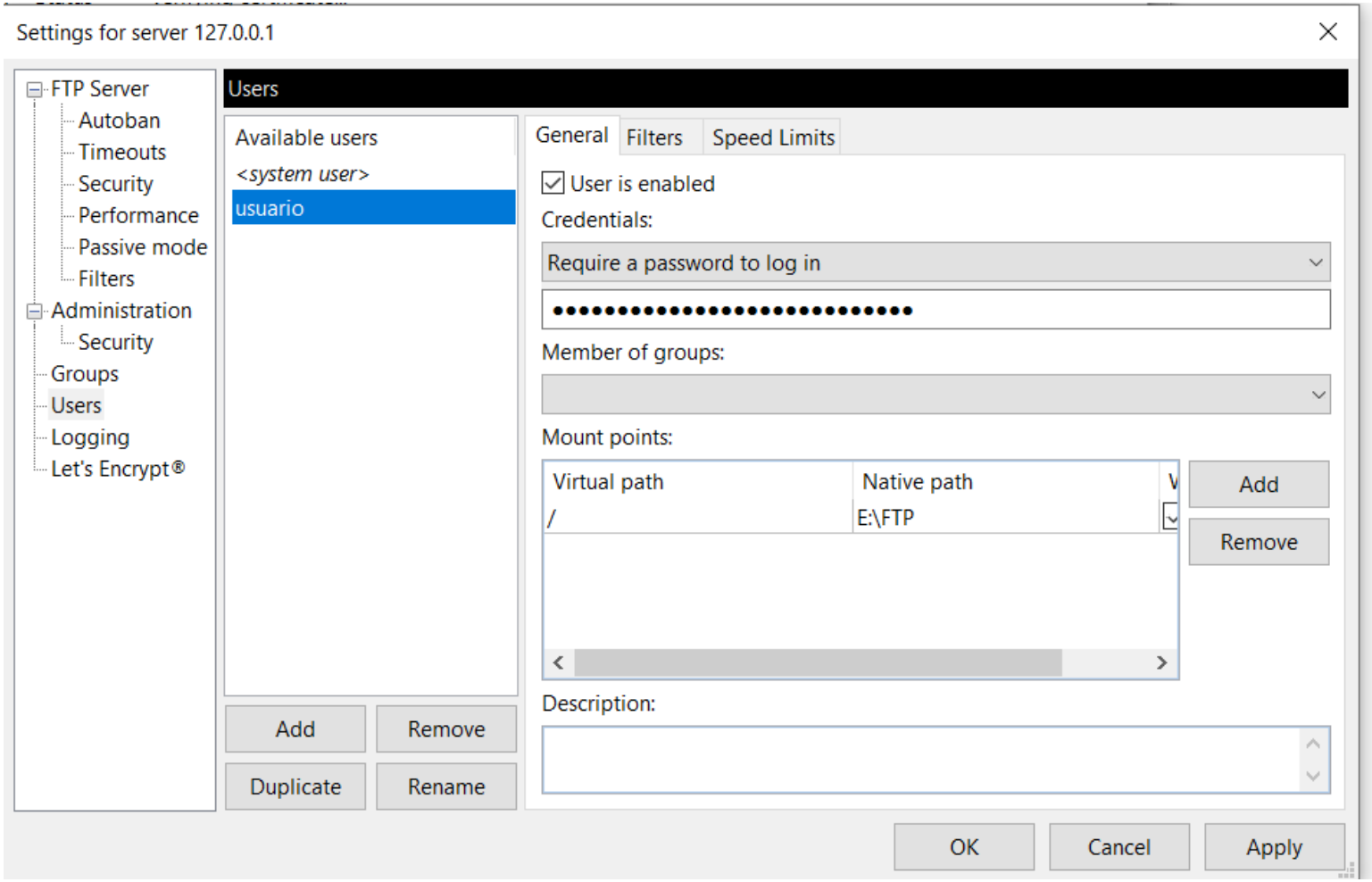
- **Una forma sencilla de crear un cliente FTP es mediante la clase FTPClient**.
- Pasos:
  - **Realizar la conexión** del cliente con el servidor utilizando el método *connect*.
  - **Comprobar la conexión** utilizando el método *getReplyCode().*
  - **Validar usuario** utilizando el método *login(String usuario, String password).*
  - **Realizar operaciones contra el servidor, ejemplos** (listNames(), retrieveFiles(), etc).
  - **Desconexión del servidor** utilizando el método disconnect() o logout().

# 4.3.2 Programación de un cliente FTP

Ejemplo de descarga del fichero utilizando el método *retrieveFile*.

```
//nombre que el que va a recuperarse
ficheroObtenido = new FileOutputStream(nombreFichero);
//mensaje
System.out.println("\nDescargando el fichero " + nombreFichero + " de "
        + "la carpeta " + rutaFichero);
//recupera el contenido del fichero en el Servidor, y lo escribe en el
//nuevo fichero del directorio del proyecto
clienteFTP.retrieveFile("/" + rutaFichero + "/"
        + nombreFichero, ficheroObtenido);
//cierra el nuevo fichero
ficheroObtenido.close();
//cierra la conexión con el Servidor
clienteFTP.disconnect();
```

# 4.3.3 Configuración de Filezilla Server



# 4.3.2 Programación de un cliente FTP

- Ejercicio: programa un cliente FTP y prueba distintas funciones de la clase FTPClient. Como mínimo de deberás listar los ficheros y directorios y descargar uno de ellos.
- Es importante leer la documentación de la API oficial en:
  - https://commons.apache.org/proper/commons-net/apidocs/org/apache/commons/net/ftp/FTPClient.html

# 4.4 Servicio de correo electrónico

- Cuando hablamos de servicios de **correo electrónico**, básicamente nos referimos a dos servicios diferentes que colaboran para conseguir transmitir un mensaje digital desde el dispositivo del **autor** (remitente) en el dispositivo del **destinatario**.
- Se utilizan los **buzones de correo** para que autores y destinatarios no tengan que tener su dispositivo permanentemente conectado.

# 4.4.1 Formato del mensaje

- Se encuentra especificado el RFC 2822
- Los mensajes constan de dos partes:
  - Cabecera.
    - Date: fecha en la que ha sido enviado el mensaje.
    - From: dirección del remitente.
    - To: Dirección del destinatario (nombre opcional).
    - Cc: dirección de destinatarios (nombres opcionales).
    - Bcc: igual que Cc pero el valor de este campo no llegará nunca a los usuarios.
    - Subject: Texto explicativo.
    - Reply –to: dirección a la que contestar el correo en caso de que no coincida con From.

# 4.4.1 Formato del mensaje

- Cuerpo
  - El mensaje o texto escrito.

Date: 10 Jun 2020 18:01:23 +0200

From: professor@educantabria.es

To: Tu nombre <your@domain.org>, Más gente <another@domain.org>

Subject: Ejemplo que ilustra un mensaje de correo electrónico

Cc: El jefe <boss@domain.org>

Bcc: El Eljefe mayor <big_boss@domain.org>

Esto es un ejemplo de mensaje de correo electrónico

Un saludo,

Joaquín.

# 4.4.1 Formato del mensaje

- Se pueden adjuntar ficheros a los mensajes de correo electrónico.
- Esto se consigue con el estándar **MIME** (Multipurpose Internet Mail Extension).
- Este estándar nombra los tipos de formatos existentes.
  - https://en.wikipedia.org/wiki/Media_type

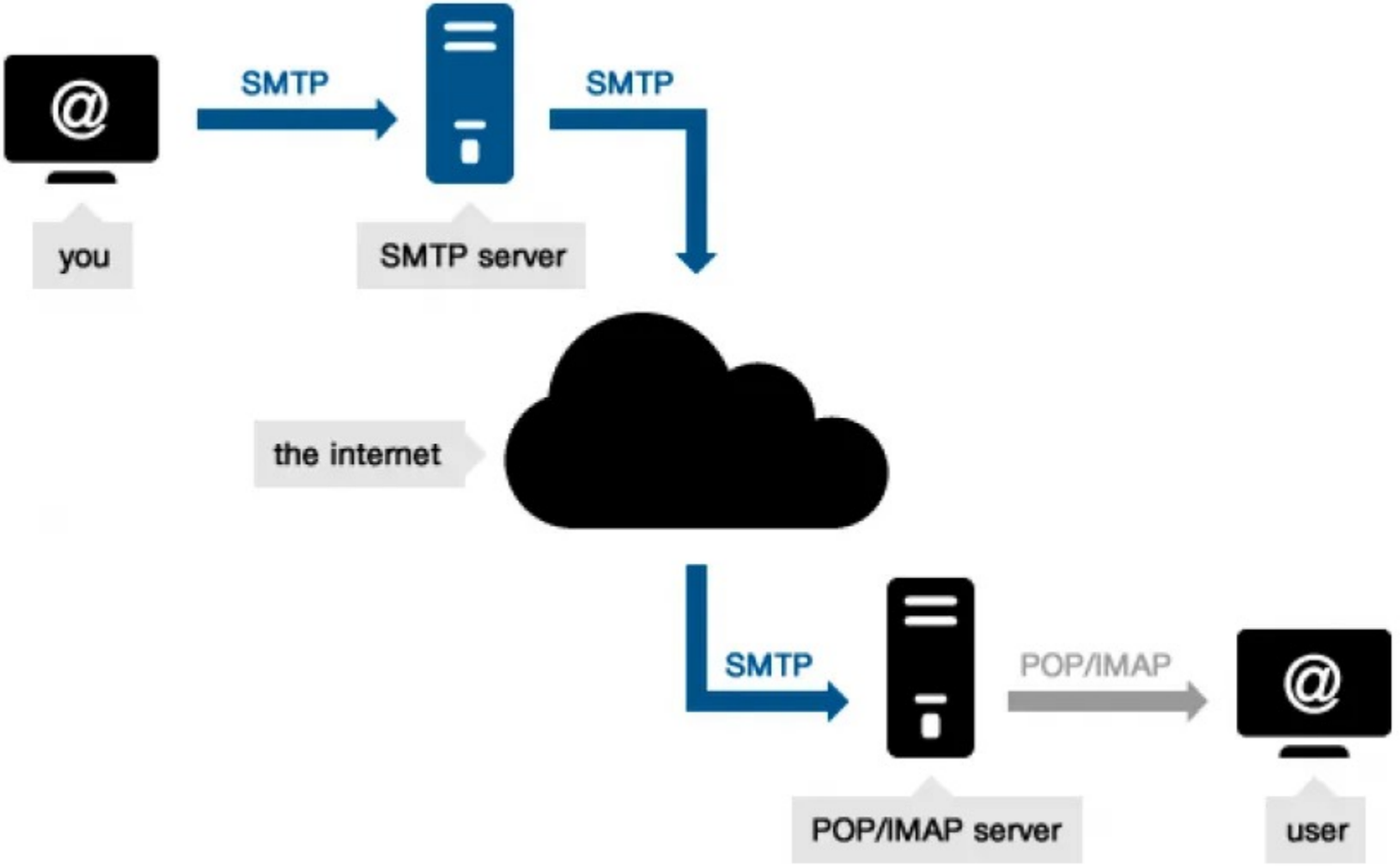
# 4.4.2 SMTP y POP3

- SMTP (Simple Mail Transfer Protocol) es el servicio encargado de transmitir los mensajes de correo electrónico.
- El puerto asignado es el 25.
- RFC 5321
- 3 actores diferentes:
  - Usuario remitente
  - Servidor de correos
  - Proveedor de correos del destinatario.

# 4.4.2 SMTP y POP3

- POP3 es el protocolo para la recepción de correos.
- Permite el acceso a los buzones leer los correos almacenados.
- Actualmente muchos proveedores ofrecen servicio a través de web. De tal manera que no hace falta que el cliente se descargue el correo. Ver protocolo IMAP.
- Algunos comandos: USER, PASS, LIST, RETR, etc.

# 4.4.2 SMTP y POP3



# 4.4.3 Programación de un cliente SMTP

- La API **javax.mail** proporciona las clases necesarias para implementar un sistema de correo y reconoce los protocolos SMTP y POP3.
- https://javaee.github.io/javamail/

# 4.4.3 Programación de un cliente SMTP

- Clases y métodos del paquete javax.mail para crear un cliente de correo:
  - Clase **Session**: representa una sesión de correo.

  Esta clase se configura a través de las **Properties**.

  http://connector.sourceforge.net/doc-files/Properties.html

```
Properties props = new Properties();
props.put("mail.smtp.host", hostName);
props.put("mail.smtp.port", 459);
//Crea una instancia que almacena les datos de configuración
Session session = Session.getInstance(props);
```

# 4.4.3 Programación de un cliente SMTP

- El método getDefaultInstance() obtiene la sesión por defecto. El parámetro que se le pasa debe tener al menos las propiedades: protocolo y servidor smtp, puerto para el socket de sesión, y usuario y puerto smtp.
- Clase **Message:** modela un mensaje de correo electrónico. En Java utilizaremos la clase ***MimeMessage***.
  - Método setFrom(): asigna la dirección del emisor al mensaje.
  - Método setRecipients(): asigna el tipo y direcciones de los destinatarios.

# 4.4.3 Programación de un cliente SMTP

- Método setSubject(): para indicar asunto del mensaje.
- Método setText(): asigna el cuerpo del mensaje.

```
Message emailMessage = new MimeMessage(session);
emailMessage.setFrom(new InternetAddress("lameva@ioc.com"));
InternetAddress[] address = {new InternetAddress("secretaria@ioc.com" ),
new InternetAddress("direccionfp@ioc.com")};
emailMessage.setRecipients(Message.RecipientType.TO, address);
emailMessage.setRecipient(Message.RecipientType.CC,
new InternetAddress("secretaria@ioc.com" ) );
 emailMessage.setSubject("Prueba de contrucción de mensaje con JavaMail");
emailMessage.setSentDate(new Date());
emailMessage.setText("Contenido del mensaje con JavaMail");
```

# 4.4.3 Programación de un cliente SMTP

- Clase **Transport**: representa el transporte de mensajes.
  - Método *send():* realiza el envío del mensaje a todas las direcciones indicadas. Si alguna dirección de destino no es válida se lanza una excepción *SendFailedException*.

```
//Objeto para enviar el mensaje
Transport t = session.getTransport("smtp");
//Conexión con usuario y contraseña
//Enviamos el mensaje
t.sendMessage(message, message.getAllRecipients());
t.close();
```

# 4.4.3 Programación de un cliente POP3

- Para leer un correo habrá que hacer uso de la clase **Store** que representa el buzón remoto.
- Habrá que pasar por parámetro el nombre de usuario y contraseña para autenticarse en el servidor.
- A partir de un objeto Store conseguimos un objeto **Folder** que representa una carpeta del buzón.

# 4.4.3 Programación de un cliente POP3

- Creación del objeto store

```
//Obtenemos el almacén de correos
Store store= session.getStore("pop3");
store.connect("pop.gmail.com", "joaq.franco@gmail.com",
//Pedimos la carpeta INBOX para solo lectura
```

- Establecer un objeto folder de sólo lectura

```
//Pedimos la carpeta INBOX para solo lectura
Folder folder= store.getFolder("INBOX");
folder.open(Folder.READ ONLY);
```

- Obtener un array de objetos Message

```
//Obtengo los mensajes nuevos. Si no funciona enviarme uno a mi mismo
Message [] mensajes= folder.getMessages();
```

# 4.4.4 Programación de un cliente electrónico

- Programa un cliente de correo electrónico siguiendo la siguiente guía:

- https://www.campusmvp.es/recursos/post/como-enviar-correo-electronico-con-java-a-traves-de-gmail.aspx

# 4.5 Protocolo Telnet

- Protocolo para establecer conexiones remotas con otros ordenadores, servidores y dispositivos.
- Se utiliza el puerto 23 de manera predeterminada.
- Se necesita un cliente y un servidor.
- Se utiliza en modo comando.
- Problema: en telnet la información viaja en texto plano.
- Solución: SSH que utiliza el puerto 22.
- **Ejemplo** de programación de un cliente Telnet en Java
  - https://hablandojava.blogspot.com/2010/01/cliente-telnet-java.html

# 4.6 Protocolo SSH (instalación)

- Para instalar el servicio en Ubuntu.
  - **sudo apt install openssh-server**
- Comprobar el estado del servidor SSH
  - **sudo systemctl status ssh**
- Activar y reiniciar el servidor ssh.
  - **Sudo systemctl enable ssh**
  - **Sudo systemctl start ssh**
- Abrir el puerto SSH.
  - **sudo ufw allow ssh**

# 4.6 Protocolo SSH (configuración)

- Configuración del servidor SSH
  - Se puede cambiar la configuración estándar, por ejemplo, si quieres comunicarte con otro puerto, elegir otra versión del protocolo o desactivar el TCP forwarding.
- El archivo de configuración se llama **sshd_config.**
  - **sudo nano /etc/ssh/sshd_config**

# 4.6 Protocolo SSH (conexión con usuario y contraseña)

- El comando de conexión con el servidor remoto es el siguiente:
  - **ssh usuario@192.168.1.46**
- La primera vez que nos conectamos, el servidor nos devuelve una huella digital de su clave pública. Nos pregunta si aceptamos a ese servidor o no.
  - Si aceptamos se guardará la información del servidor en el fichero *~/.ssh/known_hosts.*
- A continuación, ponemos el *password* de un usuario del equipo.

```
The authenticity of host '192.168.1.46 (192.168.1.46)' can't be established.
ED25519 key fingerprint is SHA256:F3he+NK3CYOTKYyu58j72j1g6DO+DOOXOrtJAID9OrU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.1.46' (ED25519) to the list of known hosts.
usuario@192.168.1.46's password:
Permission denied, please try again.
usuario@192.168.1.46's password:
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-47-generic x86_64)
```

# 4.6 Protocolo SSH (conexión con claves pública/privada)

- Con este mecanismo nos autenticamos con el servidor con una clave privada.
- Creación de la clave privada con la herramienta ***ssh-keygen***.
  - Se debe especificar con el parámetro –t el algoritmo a utilizar (dsa, ecdsa, ed25519, rsa, rsa1).
  - **ssh-keygen –t rsa**
- Se nos creará un diálogo el cual nos pedirá una frase de paso para proteger la clave privada. Lo ignoraremos en este momento.
- Si aceptamos todas las opciones por defecto, se nos crearán dos ficheros:
  - ~/.ssh/id_rsa (clave privada)
  - ~/.ssh/id_rsa.pub (clave pública)
- Es interesante mirar el contenido de estos ficheros con un editor de texto.

# 4.6 Protocolo SSH (conexión con claves pública/privada)

- Ahora hay que copiar la clave pública en la cuenta que tiene el usuario en el equipo remoto.
- Concretamente dentro del fichero **~/.ssh/authorized_keys**.
- Para ello, vamos a utilizar la herramienta ***ssh-copy-id.***
  - **ssh-copy-id –i ~/.ssh/id_rsa usuario@192.168.1.46**

```
$ ssh-copy-id -i ~/.ssh/id_rsa usuario@192.168.1.46
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/c/Users/joaqf/.s
sh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter
out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompt
ed now it is to install the new keys
usuario@192.168.1.46's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'usuario@192.168.1.46'"
and check to make sure that only the key(s) you wanted were added.
```

# 4.6 Protocolo SSH (conexión con claves pública/privada)

- Ahora, no nos pedirá el usuario y contraseña para entrar a la máquina.
- Si hubiéramos puesto un nombre a los ficheros de clave diferentes a los de por defecto el comando de conexión sería el siguiente:
  - **ssh –i ~/.ssh/miclave usuario@192.168.1.46**
- **Truco** para obtener la clave pública a partir de la privada:
  - **ssh-keygen –y –f clave >> clave.pub**

# 4.6.1 Conexión a servidor SSH mediante Java

- Para la conexión mediante Java, vamos a utilizar la librería **JSCH**.
  - Esta librería nos permite conectarnos a un servidor SSH y hacer todos los comandos y acciones que podríamos hacer desde una terminal.
  - Además, nos permite utilizar el protocolo **SFTP** que nos permite descargar o subir ficheros al estilo del protocolo FTP pero utilizando SSH.
  - Esta librería está descontinuada desde 2018 y es incompatible con los nuevos algoritmos de encriptación. Así que, vamos a utilizar un ***fork*** de JSCH más actualizado.
    - **https://github.com/mwiede/jsch**
  - **La documentación oficial está en el siguiente enlace:**
    - **https://epaul.github.io/jsch-documentation/javadoc/**

# 4.6.1 Conexión a servidor SSH mediante Java

- **Para añadir esta librería hay que añadir la siguiente dependencia Maven al proyecto:**

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/com.jcraft/jsch -->
    <dependency>
        <groupId>com.github.mwiede</groupId>
        <artifactId>jsch</artifactId>
        <version>0.2.5</version>
    </dependency>

</dependencies>
```

# 4.6.1 Conexión a servidor SSH mediante Java

- Conexión al servidor SSH

```
System.out.println("----- INICIO");

JSch jsch = new JSch();

jsch.setKnownHosts("~/.ssh/known_hosts");
jsch.addIdentity(privateKey);
System.out.println("identity added ");

Session session = jsch.getSession(user, host, port);
System.out.println("session created.");


//session.setPassword(pass);
session.connect();
System.out.println("session connected.....");
```

# 4.6.1 Conexión a servidor SSH mediante Java

- Obtener sesión de comandos:
  - ChannelExec channelExec = (ChannelExec)session.openChannel("exec");
- Obtener sesión SFTP
  - ChannelSftp sftp = (ChannelSftp)session.openChannel("sftp");

# 4.6.2 Ejercicio con conexión SSH desde Java

- Con la librería JSCH, conéctate a una máquina con servidor SSH activado y haz distintas operaciones sobré él.
  - Con el canal de comandos (exec), haz distintos comandos sobre el servidor Linux. Busca qué comandos se pueden hacer en el servidor y haz al menos 5.
  - Con el canal SFTP (sftp), prueba a cambiar de directorio, a subir un fichero y a descargar un fichero.