GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unidad 2.1

Programación de comunicaciones en red

Índice

Sumario

Licencia

1 REPASO DE CONCEPTOS

Puedes ampliar este tema con el documento **Tema 2-Protocolos Modelo OSI**

1.1 DNS

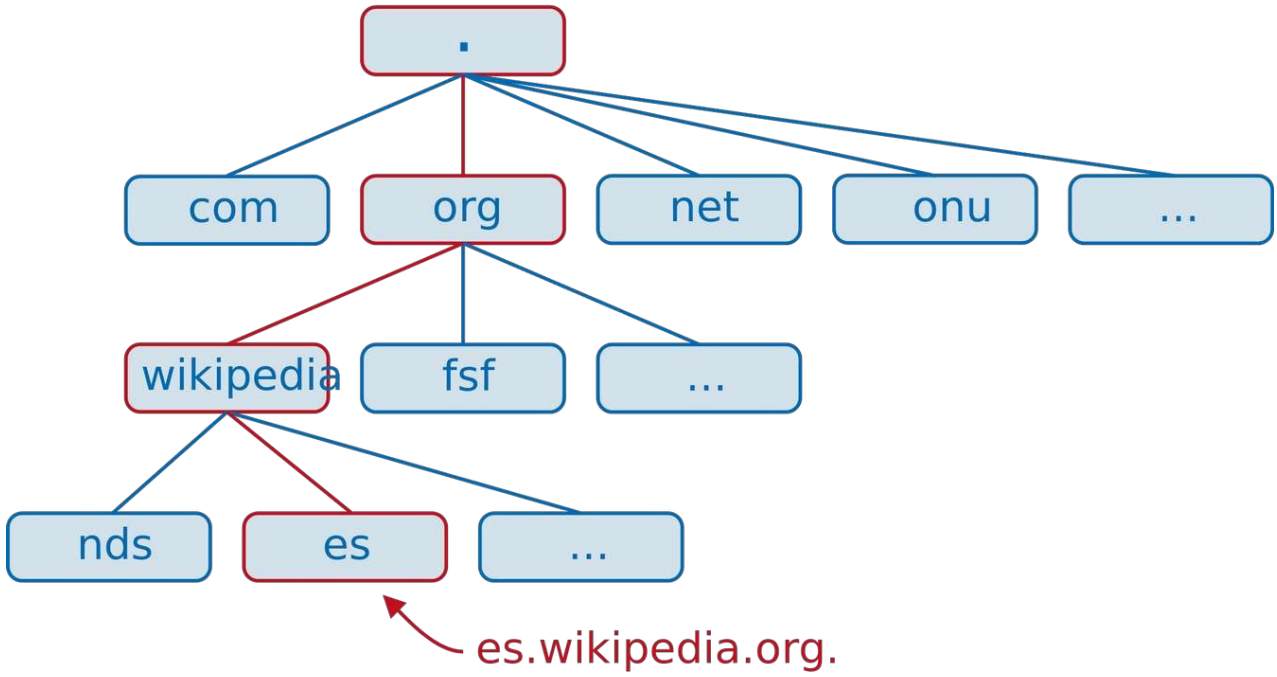
El sistema de nombres de dominio (Domain Name System o DNS) es un sistema que maneja unas tablas o base de datos distribuidas y jerárquicas, para asociar nombres de dominios en redes (como los que hay en Internet), a una dirección IP.

Por ejemplo: La dirección ip del dominio Google.com es: 216.58.210.163. Pero a nosotros nos cuesta recordar estos números, así que las personas humanas usan este nombre para referirse a Google: www.google.com y no la dirección IP.

Para las personas es más fácil recordar el nombre de un sitio web que su IP, además de ser más fiable. La dirección numérica podría cambiar por muchas razones, sin que tenga que cambiar el nombre del sitio web.

DNS actúa como un intérprete de nombres de dominio a direcciones IP para que la comunicación en internet sea posible.

El espacio de nombres de los DNS se organiza de forma jerárquica separando cada palabra por punto.



Esta jerarquía ayuda a gestionar de manera eficiente la resolución de nombres de dominio.

La organización jerárquica se compone de los siguientes elementos clave:

- **Raíz (Root)**: En la parte más alta de la jerarquía se encuentra el servidor raíz, representado por un punto (.). Este servidor raíz no almacena información sobre dominios específicos, pero contiene información sobre los servidores de nombres de nivel superior (TLD).
- **Dominios de Nivel Superior (TLD)**: Justo debajo de la raíz, hay los TLD, como .com, .org, .net, .gov, .edu, entre otros. Cada uno de estos TLD se gestiona por una entidad específica que es responsable de administrar los registros de nombres de dominio en su respectivo TLD. Por ejemplo, la Corporación de Internet para la Asignación de Nombres y Números (ICANN) supervisa la gestión de TLD a nivel global.
- **Dominios de Segundo Nivel (SLD)**: Los dominios de segundo nivel son el siguiente nivel en la jerarquía, como "ejemplo" en "ejemplo.com". Los propietarios de dominios de segundo nivel compran o registran estos nombres de dominio a través de registradores acreditados.
- **Subdominios**: Bajo los dominios de segundo nivel, se pueden crear subdominios, como "blog.ejemplo.com" o "tienda.ejemplo.com". Cada uno de estos subdominios se puede tratar como un dominio independiente y puede tener su propio conjunto de registros DNS.
- **Registros DNS**: Cada nivel en la jerarquía tiene su propio servidor de nombres que almacena información sobre los dominios en ese nivel. Los registros DNS contienen información que asocia un nombre de dominio con una dirección IP, lo que permite la resolución de nombres.

Cuando un usuario ingresa una URL en su navegador, el sistema DNS trabaja de arriba hacia abajo en la jerarquía para encontrar la dirección IP correspondiente al nombre de dominio. Comienza consultando el servidor raíz para determinar la ubicación del servidor de nombres del TLD apropiado y, a medida que desciende en la jerarquía, resuelve la dirección IP completa.

Esta organización jerárquica garantiza una administración eficiente y descentralizada de nombres de dominio en internet y permite que los servidores de nombres se especialicen en su nivel específico de la jerarquía.

1.2 Capa TCP/UDP

El **nivel de transporte** o **capa de transporte** es el cuarto nivel del modelo OSI, y está encargado de la transferencia libre de errores de los datos entre el emisor y el receptor, aunque no estén directamente conectados, así como de mantener el flujo de la red.

En las capas predecesoras los elementos identificativos básicos eran:

- La dirección MAC (capa 2 OSI: **Enlace de datos**)
- La dirección IP (capa 3 OSI: **Red**)

Uno de los conceptos clave **en la capa de transporte** es el uso de **puertos**, que son números que se utilizan para identificar procesos y servicios específicos en un dispositivo dentro de una red. Los puertos permiten que múltiples aplicaciones o servicios se ejecuten en un mismo dispositivo y se comuniquen a través de una sola dirección IP.

1.3 Los puertos de red o port

Un puerto es un número de 16 bits, por lo que existen 65536 puertos, numerados del 0 al 65535.

Los puertos TCP se utilizan para identificar diferentes servicios y aplicaciones en una red. Aunque no hay una clasificación rígida en tipos de puertos TCP, es posible agruparlos en algunas categorías generales según su uso:

- **Puertos Conocidos**: Estos son los puertos que están reservados para servicios y aplicaciones ampliamente reconocidos y estandarizados. Van desde el puerto 0 hasta el puerto 1023. Ejemplos incluyen el puerto 80 para HTTP, el puerto 443 para HTTPS y el puerto 25 para SMTP.
- **Puertos Registrados:** Los puertos registrados (1024-49151) se utilizan para aplicaciones y servicios específicos que no son tan ampliamente reconocidos como los de los puertos bien conocidos. Estos puertos están registrados en la Autoridad de Asignación de Números de Internet (IANA).
- **Puertos Dinámicos o Privados**: Los puertos dinámicos (49152-65535) se utilizan para la comunicación temporal y efímera entre dispositivos. Estos puertos no están reservados y se asignan dinámicamente para la comunicación de corta duración.

Algunos ejemplos de puerto:

- **Puertos conocidos o well-known**, con valores inferiores a 1024.

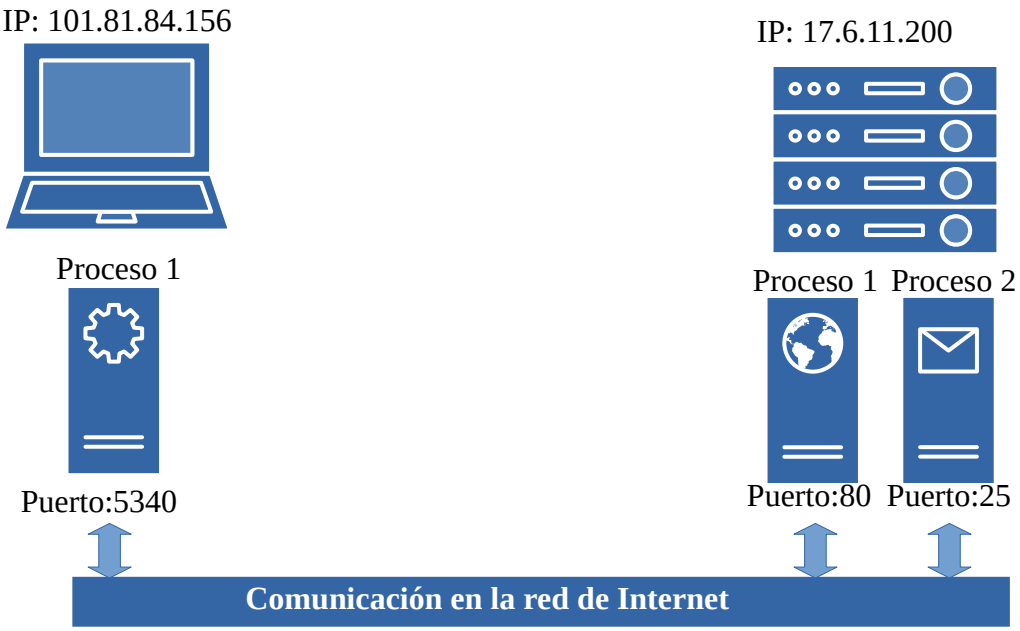
  Son **puertos reservados** por el Sistema Operativo:
  - Echo: 7 (Protocolo Echo (Eco) Responde con eco a llamadas remotas)
  - FTP: 20 y 21
  - SSH: 22
  - Telnet: 23
  - SMTP: 25
  - Time: 37
  - DNS: 53
  - HTTP: 80
  - POP3: 110
  - IMAP3: 220
  - LDAP: 389
  - HTTPS: 443
  - SMTP: 587

1-255 = aplicaciones públicas
255-1023 = para determinadas aplicaciones que necesiten privilegios de superusuario

- **Puertos registrados**, con valores entre 1024 y 49151.

  Están registrados en una lista pública de la IANA:
  - 1194: OpenVPN
  - 1433: Microsoft-SQL-Server
  - 1521: Oracle
  - 2095: WebMail
  - 3306: base de datos MySQL
  - 4662: eMule
  - 6881: BitTorrent
  - 25565: para el servidor de juego Minecraft
  - 26000: para servidores de juego Quake
- **Puertos privados**, para el resto de valores. También se conocen como puertos dinámicos, dado que se suelen cread de manera dinámica y temporal por el sistema operativo. Casi siempre están en la parte cliente y sirven para iniciar la conexión.
  - > 5000 = usado solo por procesos de usuario

1.4 Capa TCP/UDP



Para establecer una comunicación con un punto remoto, a nivel de aplicación, es necesario conocer la IP y el puerto del destino. Este puerto debe tener un valor fijo en la parte que queremos conectar y puede tener un valor variable en el host que inicia la comunicación.

Las aplicaciones de cliente y servidor (los procesos), desarrolladas por los usuarios, se encuentran en la capa de Aplicación y utilizan la API de sockets para interactuar con la capa de Transporte.

La capa de Transporte se encarga de tomar los datos provenientes de la capa de Aplicación (emisor) y dividirlos en segmentos. Estos segmentos se asocian con el número de puerto utilizado por la aplicación y luego se envían a la capa de Red. La elección de los protocolos específicos depende de la aplicación y del tipo de comunicación que se requiere.

Internet tiene dos protocolos estandarizados en la capa de transporte:

- Uno no orientado a la conexión, **UDP**
- Otro orientado a la conexión, **TCP**.

Ambos utilizan el protocolo IP:

- **TCP (Protocolo de Control de Transmisión)**: TCP es un protocolo orientado a la conexión y confiable. Establece una conexión entre dos dispositivos y garantiza la entrega ordenada y sin errores de los datos. Utiliza números de secuencia y acknowledgments para asegurarse de que los datos se transmitan de manera confiable. Los puertos TCP se utilizan para identificar procesos específicos en un dispositivo. Por ejemplo, el puerto 80 se utiliza comúnmente para el tráfico web HTTP, el puerto 25 para el correo electrónico SMTP, etc.

- **UDP (Protocolo de Datagramas de Usuario)**: UDP, a diferencia de TCP, es un protocolo sin conexión y no garantiza la entrega confiable de datos. Es más rápido pero menos confiable que TCP. Los puertos UDP también se utilizan para identificar aplicaciones y servicios en dispositivos. Se utiliza en aplicaciones donde la velocidad es más importante que la integridad de los datos, como la transmisión de video en tiempo real o juegos en línea.

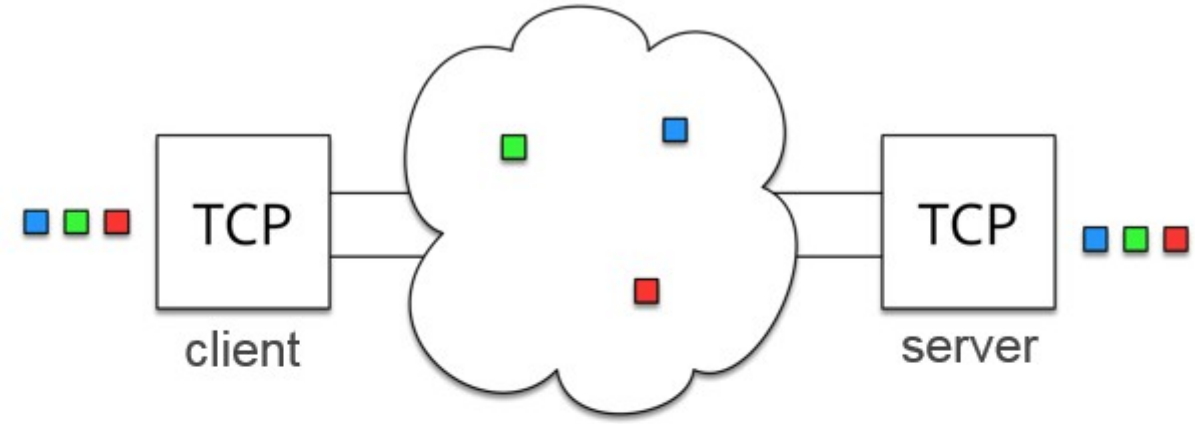
1.5 TCP (Protocolo de Control de Transmisión)

Protocolo orientado a conexión:

En este caso, para realizar un intercambio de información entre dos entidades se pueden distinguir las siguientes fases:

- Establecimiento de la conexión
- Intercambio de información
- Cierre de la conexión

Una de las características de un servicio orientado a conexión es que una vez establecida la conexión, los mensajes llegan al receptor en el mismo orden en que fueron enviados por el emisor.



Un ejemplo de este procedimiento es la comunicación a través del teléfono:

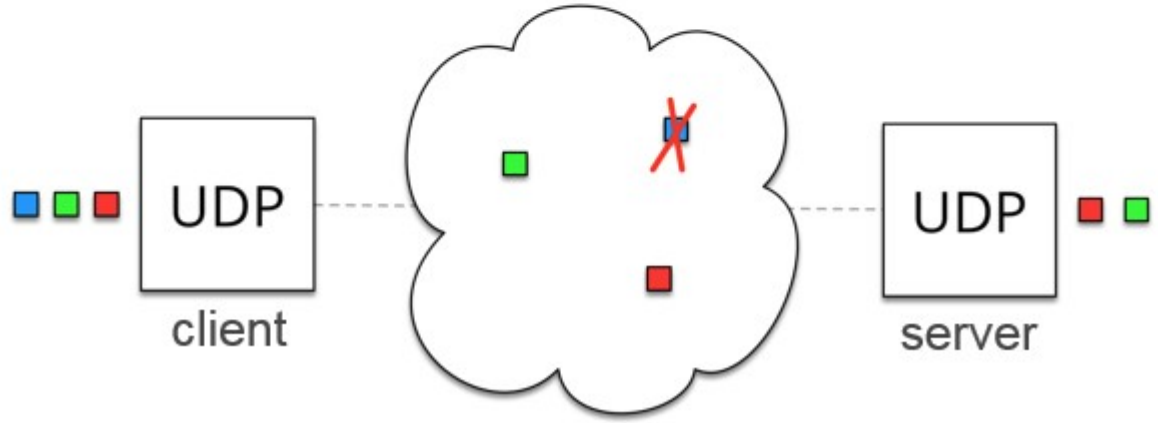
- **Establecimiento de la conexión**: La persona llamante, inicia el establecimiento de la conexión descolgando el teléfono y marcando el número del abonado llamado. El abonado llamado descuelga el teléfono y la conexión ya está realizada.
- **Intercambio de información**: Ambos abonados, llamante y llamado, conversan a través del teléfono.
- **Cierre de la conexión**: Los abonados cuelgan el teléfono dando por finalizada la llamada.

1.6 UDP (Protocolo de Datagramas de Usuario)

Protocolo No orientado a conexión:

En este caso, el intercambio de información no necesita de las fases de establecimiento y cierre de la conexión. La entidad transmisora envía el mensaje sin que la receptora sepa que va a recibir ese mensaje.

Otro parámetro que se utiliza para clasificar un protocolo es la fiabilidad del protocolo en cuanto a la entrega de la información.



Un ejemplo de un servicio no orientado a conexión es el sistema postal:

- En este caso, el remitente envía una carta al destinatario.
- El destinatario sabe que el remitente le envió una carta en el momento en que la recibe.

2 Sockets

Un socket es una abstracción de programación que se utiliza para establecer y gestionar la comunicación de datos en una red, ya sea a nivel local o a través de Internet. Un socket combina una dirección IP y un número de puerto para identificar una conexión entre dos dispositivos en una red.

El socket lleva implícito:

- Protocolos de sincronización
- Una estructura de recepción de datos (buffer)
- Mecanismos de generación de eventos del sistema operativo
- Dispositivo de interrupción del hardware

2.1 Elementos básicos de un socket (socket address)

Hay dos elementos básicos en los sockets:

- **Dirección IP** del host en el que es creado:

  Una dirección IP identifica de manera única un dispositivo en una red. En el contexto de sockets, una dirección IP especifica la ubicación de un dispositivo al que se desea enviar o desde el cual se desea recibir datos.
- **Número de Puerto** al cual está ligado:

  Los números de puerto se utilizan para identificar aplicaciones o servicios específicos en un dispositivo. Los puertos permiten que múltiples aplicaciones se ejecuten en un mismo dispositivo y se comuniquen a través de una sola dirección IP. Hay miles de números de puerto disponibles, y algunos de ellos están reservados para aplicaciones comunes, como el puerto 80 para HTTP o el puerto 25 para SMTP.

Por ejemplo en una conexión entre dos procesos que utilizan el protocolo TCP (Protocolo de Control de Transmisión), se requieren dos sockets para establecer y gestionar la comunicación bidireccional de datos de manera confiable. Habrá que crear un socket en el lad del cliente y otro socket en el lado del servidor :

- **Socket address origen (cliente)**:
  - dirección **IP** del origen
  - **puerto** del origen
- **Socket address destino (servidor):**
  - dirección **IP** del destino
  - **puerto** del destino

2.2 Tipos de socket según el protocolo que implementan:

Hay diferentes tipos de sockets, que se adaptan a las necesidades específicas de las aplicaciones y los protocolos que implementan. La elección del tipo de socket dependerá de factores como la confiabilidad, la velocidad, el control y el tipo de comunicación requerida en una aplicación.

Tipos de socket:

- **Stream Sockets (Sockets de Flujo)**:
  - Protocolo Principal: **TCP** (Protocolo de Control de Transmisión).
  - Características: Los sockets de flujo se utilizan para establecer conexiones orientadas a la conexión y confiables. Proporcionan una transferencia de datos bidireccional con control de errores y retransmisiones en caso de pérdida de datos. Se garantiza que los datos lleguen en orden y sin duplicados.
  - Uso Común: Los sockets de flujo son ideales para aplicaciones que requieren una comunicación confiable y sin pérdida de datos, como la transferencia de archivos, el correo electrónico y las conexiones de servidor-cliente.
- **Datagram Sockets (Sockets de Datagrama)**:
  - Protocolo Principal: **UDP** (Protocolo de Datagramas de Usuario).
  - Características: Los sockets de datagrama se utilizan para comunicaciones sin conexión y no confiables. No garantizan la entrega de datos ni el orden de llegada. Son más rápidos y eficientes en términos de latencia que los sockets de flujo.
  - Uso Común: Los sockets de datagrama son adecuados para aplicaciones en tiempo real que priorizan la velocidad, como VoIP, streaming de video y juegos en línea.
- **Raw Sockets**:
  - Protocolo Principal: Pueden ser utilizados con varios protocolos de bajo nivel, como ICMP (Protocolo de Mensajes de Control de Internet).
  - Características: Los Raw Sockets permiten un mayor control sobre la comunicación a nivel de paquete. Se utilizan para enviar y recibir paquetes de datos directamente sin el procesamiento habitual de las capas superiores del modelo OSI.
  - Uso Común: Los Raw Sockets son utilizados principalmente para tareas de diagnóstico, análisis de redes, y a menudo requieren privilegios de administrador debido a su capacidad para manipular paquetes a nivel de red.

- **WebSockets**:
  - Protocolo Principal: WebSocket, que se ejecuta sobre TCP.
  - Características: WebSockets proporcionan una comunicación bidireccional y de bajo retardo, lo que los hace ideales para aplicaciones web en tiempo real. Establecen una conexión persistente y permiten la transmisión de datos en ambas direcciones sin la necesidad de una solicitud-respuesta como en HTTP.
  - Uso Común: Los WebSockets se utilizan en aplicaciones web interactivas, como chats en línea, juegos multijugador, actualizaciones en tiempo real y cualquier aplicación que requiera una comunicación bidireccional a través del navegador web.

3 Arquitectura cliente-servidor

La arquitectura cliente-servidor es un modelo de diseño ampliamente utilizado en informática y redes de computadoras. Se basa en la idea de que un sistema o aplicación se divide en dos componentes principales: el cliente y el servidor. Cada uno de estos componentes tiene un papel específico y se comunican entre sí para proporcionar servicios o recursos a los usuarios.

- **Cliente**: El cliente es el componente que solicita servicios o recursos al servidor. Los clientes suelen ser aplicaciones o dispositivos que se ejecutan en los equipos de los usuarios finales. Su función principal es interactuar con los usuarios, recopilar sus solicitudes y enviarlas al servidor.
- **Servidor**: El servidor es el componente que proporciona los servicios o recursos solicitados por los clientes. Está diseñado para escuchar las solicitudes entrantes de los clientes, procesarlas y responder adecuadamente. Los servidores pueden ser máquinas físicas o software que se ejecuta en máquinas dedicadas.

3.1 Características de la Arquitectura Cliente-Servidor:

- **Distribución de Tareas**: La arquitectura cliente-servidor permite la distribución de tareas entre los componentes. Los servidores se encargan de procesar y gestionar los recursos, mientras que los clientes se centran en proporcionar interfaces de usuario amigables y recopilar las solicitudes de los usuarios.
- **Comunicación**: La comunicación entre clientes y servidores se realiza a través de redes, utilizando protocolos de comunicación estándar, como HTTP, TCP/IP o UDP. Esto permite la interacción entre dispositivos en redes locales o a través de Internet.
- **Escalabilidad**: La arquitectura cliente-servidor es escalable, lo que significa que se pueden agregar más clientes y servidores según las necesidades. Esto facilita la adaptación a un aumento en la demanda de servicios.
- **Centralización de Recursos**: Los servidores almacenan y gestionan recursos compartidos, como bases de datos, archivos, servicios y aplicaciones. Esto asegura la integridad y el control de los datos y recursos.
- **Seguridad**: La arquitectura cliente-servidor permite implementar medidas de seguridad en el servidor para proteger los datos y recursos compartidos. Los servidores suelen contar con mecanismos de autenticación y autorización.

Ejemplos de Aplicaciones Cliente-Servidor:

- Navegadores web y servidores web: Los navegadores son clientes que solicitan páginas web a servidores web. El servidor entrega las páginas web al cliente para su visualización.
- Correo electrónico: Los clientes de correo electrónico solicitan y envían mensajes a servidores de correo para su almacenamiento y entrega.
- Bases de datos: Las aplicaciones de bases de datos se comunican con servidores de bases de datos para almacenar y recuperar información.

- Juegos en línea: Los juegos en línea utilizan una arquitectura cliente-servidor para permitir la interacción entre jugadores en diferentes ubicaciones.
- Redes sociales: Plataformas como Facebook, Twitter y LinkedIn son ejemplos de aplicaciones cliente-servidor donde los clientes interactúan con servidores para compartir contenido y comunicarse.

4 Sockets: Stream Socket (TCP)

Los "Stream Sockets" son un tipo de sockets que se utilizan para comunicación en tiempo real y confiable en un entorno cliente-servidor. En este escenario, un servidor crea un socket de escucha que espera conexiones entrantes de clientes. Cuando un cliente se conecta al servidor, se establece una conexión bidireccional, y ambos extremos del socket pueden enviar y recibir datos de manera confiable y ordenada.

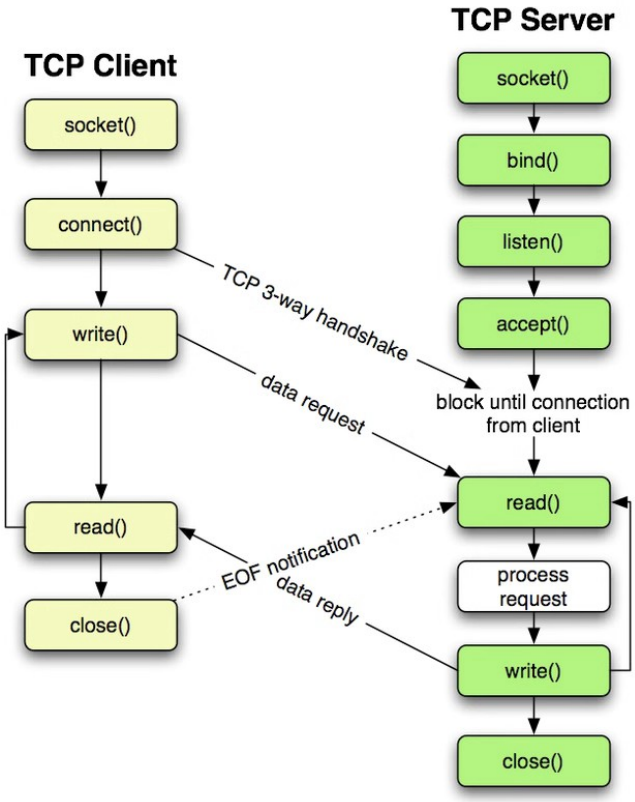
Puntos a remarcar:

- **Define un servicio orientado a conexión confiable** (sobre TCP por ejemplo). Al conectar, se realiza una búsqueda de un camino libre entre origen y destino. Se mantiene el camino durante toda la conexión.
- Los datos se envían sin errores o duplicación y se reciben en el **mismo orden** de como fueron enviados.
- El control de flujo está integrado para evitar overruns.
- **No se imponen límites en el intercambio de datos**, que se considera flujo de bytes.
- Ejemplo de aplicación que use sockets de flujo: la transferencia de archivos (FTP), el chat en línea o la transmisión de video en tiempo real.

4.1 Funciones en Python para Crear Comunicación TCP Cliente-Servidor:

En Python, la biblioteca socket proporciona las funciones necesarias para crear una comunicación TCP entre un cliente y un servidor.

- En la **parte de cliente** las funciones para crear, mantener y finalizar las comunicación son:
  - Socket()
  - Connect()
  - Send()
  - Receive()
  - Close()
- En la **parte del servidor** las funciones para crear, mantener y finalizar las comunicación son:
  - Socket()
  - Bind()
  - Listen()
  - Accept()
  - Receive()
  - Send()
  - Close()



4.2 Descripción de las funciones en la parte de cliente TCP

4.2.1 Socket(): El cliente crea un objeto de tipo socket

Al crear un objeto tipo socket(), hay que definir varios parámetros:

- **family**: Es el tipo de dirección de socket y protocolo que se utilizará.
  - **AF_INET**: crea un socket con una dirección de protocolo de Internet versión 4 (IPv4).
  - **AF_UNIX** para la dirección UNIX.
  - **AF_INET6** para la dirección del protocolo de Internet versión 6 (IPv6).
- **type**: Es el tipo de socket
  - **SOCK_STREAM** denotando que el socket seguirá el protocolo **TCP**: orientado a la conexión.
  - **SOCK_DGRAM** denotando que el socket seguirá el protocolo **UDP**: no orientado a la conexión.

Para los ejercicios que iremos realizando nos centraremos en los ockets de Dominio de Internet (AF_INET), dado que son el tipo más común de socket.

Nos centraremos en los sockets de Dominio de Internet (AF_INET)

Como estamos en una comunicación TCP (orientado a la conexión), la creación del objeto socket será:

```
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

4.2.2 Connect()

Sirve para conectar el socket cliente al puerto en el que el servidor de dicha dirección IP está escuchando. A connect() se le pasan el hostname y el puerto como parámetros. El cliente debe conocer esta información.

4.2.3 Send()

Para enviar datos al servidor, se suele usar los métodos: send() y sendall()

4.2.4 Receive()

Para recibir los datos que nos envía el servidor. Se suele usar el método: recv()

4.2.5 Close()

Cierre el socket. Si todavía quedan datos por enviar en los buffers, TCP trata de enviarlos antes de indicar el cierre. A partir de ahora si el servidor le envía algo, fallará. El servidor deja de recibir datos del cliente.

4.3 Descripción de las funciones en la parte del servidor TCP

4.3.1 Socket(): El sevidor crea un objeto de tipo socket

Al crear un objeto tipo socket(), hay que definir varios parámetros:

- **family**: Es el tipo de dirección de socket y protocolo que se utilizará.
  - **AF_INET**: crea un socket con una dirección de protocolo de Internet versión 4 (IPv4).
  - **AF_UNIX** para la dirección UNIX.
  - **AF_INET6** para la dirección del protocolo de Internet versión 6 (IPv6).
- **type**: Es el tipo de socket
  - **SOCK_STREAM** denotando que el socket seguirá el protocolo **TCP**: orientado a la conexión.
  - **SOCK_DGRAM** denotando que el socket seguirá el protocolo **UDP**: no orientado a la conexión.

En este caso creamos el mismo tipo de socket() que en el cliente:

```
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

4.3.2 Bind():

Tras crear el socket, este método lo vincula a una dirección (hostname, número de puerto).

4.3.3 Listen():

Una vez vinculado el socket, con este método listen() el servidor escucha las solicitudes de conexión de los cliente que quieran sus servicios.

Nota: **El servidor se queda aquí parado esperando a que un cliente se conecte.**

4.3.4 Accept():

Cuando un cliente intenta conectarse al servidor, la función accept() espera y acepta la conexión entrante. Una vez que se acepta la conexión, accept() crea un nuevo socket que se utilizará para comunicarse exclusivamente con ese cliente.

El método accept() devuelve una tupla (cliente, dirección):

- **Cliente**: se crea un nuevo socket, que representa la conexión con el cliente. Este socket se utilizará para enviar y recibir datos con ese cliente específico. Este socket tendrá en principio la misma IP del servidor, pero con un puerto diferente.
- **Dirección**: Contiene la dirección IP y el número de puerto desde el que se originó la conexión. (los datos del socket cliente)

Esta función es esencial en la programación de servidores para manejar múltiples conexiones de clientes. Permite que el servidor acepte conexiones de manera secuencial, creando un nuevo socket para cada cliente que se conecta. Esto facilita la comunicación simultánea con varios clientes.

4.3.5 Receive():

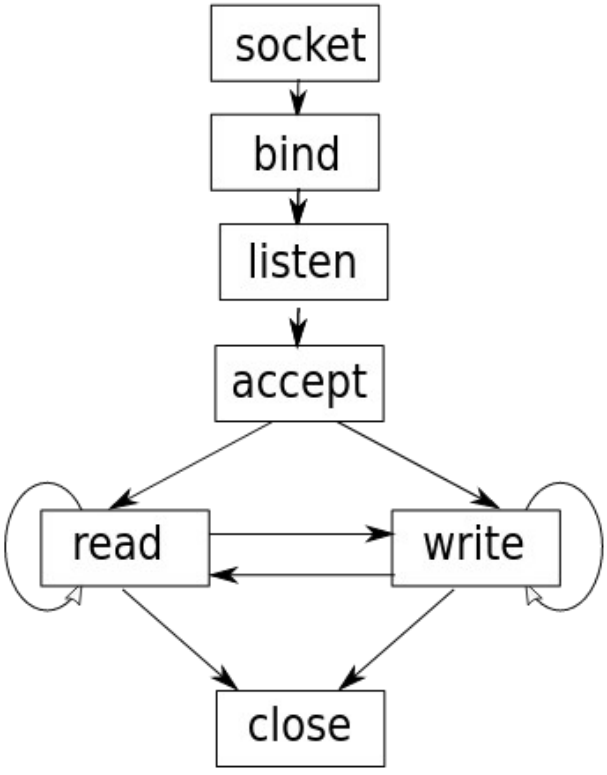
Para recibir los datos que nos envía el cliente. Se suele usar el método: recv()

4.3.6 Send():

Para enviar datos al cliente, se suele usar los métodos: send() y sendall()

4.3.7 Close():

Cierre el socket. Si todavía quedan datos por enviar en los buffers, TCP trata de enviarlos antes de indicar el cierre. A partir de ahora si el cliente le envía algo, fallará. El cliente deja de recibir datos del servidor. Los sockets se cierran automáticamente cuando el proceso padre finaliza, pero se recomienda cerrarlos() explícitamente, o usar una instrucción with alrededor de ellos.



4.4 Nota importante

- Se usa un socket para crear un vínculo entre el servidor y el cliente. Se pueden enviar los datos sin identificar el destino (la asociación está establecida).
- El servidor puede empezar la conversación enviando un send().
  - Ejemplo: el servidor de tiempo. Cuando un cliente se conecta a él, este envía la información sin esperar que el cliente diga nada.
- El cliente puede empezar la conversación enviando un send().
  - Ejemplo: El cliente solicita al servidor que le envíe la página web de su sitio, para poder leerla y mostrarla al usuario (cliente) en su navegador.

5 Sockets: Datagram Socket (UDP)

Los "Datagram Sockets" son un tipo de sockets que se utilizan para comunicación en tiempo real en un entorno cliente-servidor, especialmente cuando la velocidad y la baja latencia son más importantes que la confiabilidad de la entrega de datos. A diferencia de los "Stream Sockets" (sockets de flujo) que utilizan TCP y ofrecen una comunicación confiable, los "Datagram Sockets" utilizan UDP (Protocolo de Datagramas de Usuario), que es un protocolo sin conexión y no garantiza la entrega de datos ni el orden de llegada.

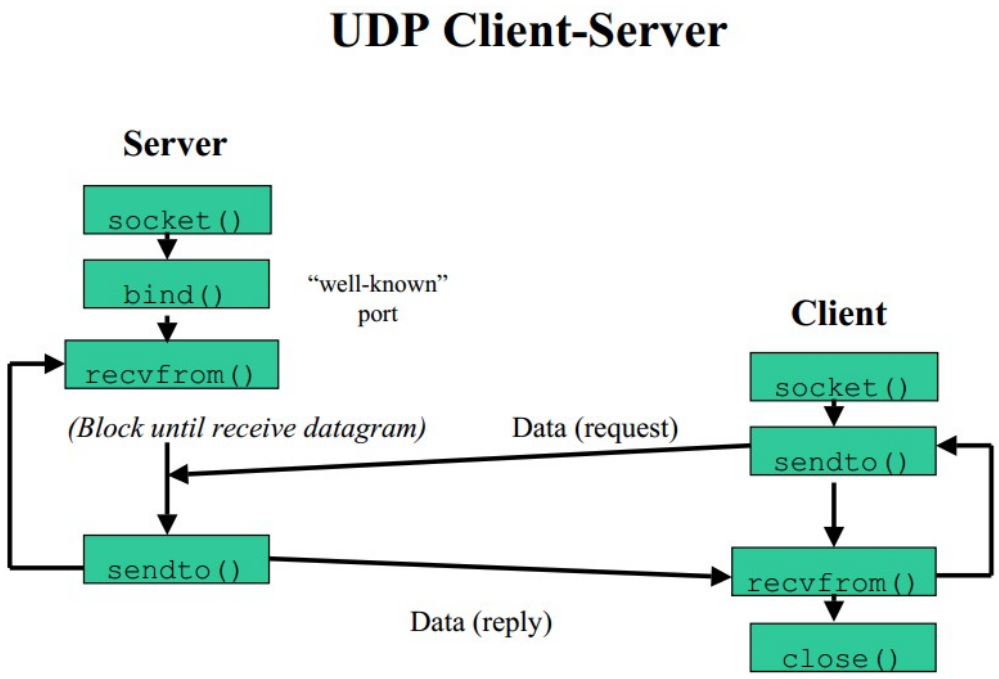
Puntos a remarcar:

- Define un servicio **no orientado a conexión** (sobre UDP por ejemplo). No se fija un camino. Cada paquete podrá ir por cualquier sitio. No se garantiza la recepción secuencial.
- Los datagramas se envían como **paquetes independientes**.
- El servicio no proporciona garantías; los datos se pueden perder o duplicar y los datagramas pueden llegar fuera de orden.
- No realiza desensamblado y reensamblado de paquetes.
- Ejemplo de aplicación que utilice sockets de datagrama: transmisión de video en tiempo real, VoIP y juegos en línea.

5.1 Funciones en Python para Crear Comunicación UDP Cliente-Servidor:

En Python, la biblioteca socket proporciona las funciones necesarias para crear una comunicación UDP entre un cliente y un servidor.

- En la **parte de cliente** las funciones para crear, mantener y finalizar las comunicación son:
  - Socket()
  - Sendto()
  - Recvfrom()
  - Close()
- En la **parte del servidor** las funciones para crear, mantener y finalizar las comunicación son:
  - Socket()
  - Bind()
  - Recfrom()
  - Sendto()
  - Close()



5.2 Descripción de las funciones en la parte de cliente UDP

5.2.1 Socket(): El cliente crea un objeto de tipo socket

Al crear un objeto tipo socket(), hay que definir varios parámetros:

- **family**: Es el tipo de dirección de socket y protocolo que se utilizará.
  - **AF_INET**: crea un socket con una dirección de protocolo de Internet versión 4 (IPv4).
  - **AF_UNIX** para la dirección UNIX.
  - **AF_INET6** para la dirección del protocolo de Internet versión 6 (IPv6).
- **type**: Es el tipo de socket
  - **SOCK_STREAM** denotando que el socket seguirá el protocolo **TCP**: orientado a la conexión.
  - **SOCK_DGRAM** denotando que el socket seguirá el protocolo **UDP**: no orientado a la conexión.

Seguimos con la familia AF_INET y como estamos en una comunicación UDP, la creación del socket es:

```
socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

5.2.2 Sendto():

El cliente debe saber el hostname o la IP del servidor UTP y su puerto, para enviarle información. En este caso, es siempre el cliente el que empieza la conversación. Nunca el servidor.

5.2.3 Recvfrom():

Para recibir los datos que nos envía el servidor. Se guarda la información en una tupla (mensaje, addr), aunque en este caso la addr ya la conocemos (es la del servidor).

5.2.4 Close():

Cierre el socket. A partir de ahora si el servidor le envía algo, fallará. El servidor deja de recibir datos del cliente.

5.3 Descripción de las funciones en la parte del servidor UDP

5.3.1 Socket(): El servidor crea un objeto de tipo socket

Al crear un objeto tipo socket(), hay que definir varios parámetros:

- **family**: Es el tipo de dirección de socket y protocolo que se utilizará.
  - **AF_INET**: crea un socket con una dirección de protocolo de Internet versión 4 (IPv4).
  - **AF_UNIX** para la dirección UNIX.
  - **AF_INET6** para la dirección del protocolo de Internet versión 6 (IPv6).
- **type**: Es el tipo de socket
  - **SOCK_STREAM** denotando que el socket seguirá el protocolo **TCP**: orientado a la conexión.
  - **SOCK_DGRAM** denotando que el socket seguirá el protocolo **UDP**: no orientado a la conexión.

Seguimos con la familia AF_INET y como estamos en una comunicación UDP, la creación del socket es:

```
socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

5.3.2 Bind():

Tras crear el socket, este método lo vincula a una dirección (hostname, número de puerto).

5.3.3 Recvfrom():

Una vez vinculado el socket, el servidor se pone a la espera de recibir información de algún cliente.

Cuando un cliente envíe algo, recvfrom() devolverá una tupla (mensaje, addr):

- mensaje: El mensaje son los datos que nos envía el cliente
- addr: La dirección del remitente-cliente.

Nota: El servidor se queda aquí parado esperando a que un cliente se conecte.

Recvfrom() es una función clave para la recepción de datos UDP en el servidor. Permite al servidor recibir los datos enviados por el cliente y, al mismo tiempo, identificar la dirección del cliente para poder responder si es necesario.

5.3.4 Sendto():

Envía datos a la dirección addr (cliente).

5.3.5 Nota importante

En UDP no se crea un vínculo entre cliente-servidor. Por ello no hay control ni verificación de los datos. Al no haber este vínculo, es importante usar la dirección “remitente” del cliente para mantener la conversación. Si no la guardáramos al recibir los datos, el servidor no tendría manera de saber a quien enviar la respuesta.