

1. Protocolos de aplicación

1.1. Arquitecturas de Red

1.1.1. Cliente servidor

Servidor:

- Servidor siempre en un host
- Dirección IP del servidor tiene que ser estática
- Si quieres más capacidad tienes que comprar más servidores

Cliente:

- No tiene que estar siempre conectado
- Puede tener IP dinámica
- No se comunica con otros clientes

$$t_{C/S} = \max\left\{\frac{Nf}{u_s}, \frac{F}{\min d_i}\right\}$$

1.1.2. Peer to peer (P2P)

- Todos los nodos son como clientes y servidores
- Arquitectura distribuida: no hay posibilidad de control central, resistencia a las averías
- Redundancia de la información, equilibrio de la carga...
- No cuesta nada hacer crecer la red
- Fórmulas para la carga en arquitecturas P2P:

$$t_{P2P} = \max\left\{\frac{F}{u_s}, \frac{F}{\min d_i}, \frac{NF}{u_s + \sum u_i}\right\}$$

1.1.3. Híbridas P2P - C/S

- Problema de P2P: localización de un nodo o de la información
- Solución: añadir un servidor central que solo se encarga de poner en contacto a nodos de la red
 - Permite que algunos clientes que estén detrás de NAT puedan utilizar la red P2P
 - Skype

1.2. QoS en la comunicación entre procesos

- Distintas aplicaciones tienen distintos requisitos: velocidad, integridad, seguridad, retardo...
- Los **protocolos** definen tipos de mensajes (sintaxis y semántica) y garantizan distintos requisitos
- Dos protocolos principales a nivel 4 que determinan que requisitos puede ofrecer un protocolo: TCP y UDP

| TCP | UDP |
|---|--|
| Orientado a conexión cada parte fija IP y puertos durante el establecimiento de la conexión. | No orientado a conexión Cada parte especifica IP y puerto de destino cada vez que envía datos (uso de sendto y recvfrom) |
| Transferencia confiable se garantiza la llegada de toda la información en orden. | Transferencia no confiable no se garantiza nada |
| Control de flujo el emisor no sobrecarga al receptor | No hay control de flujo el emisor no sabe como está el receptor |
| No provee garantías de ancho de banda | No provee garantías de ancho de banda, pero la transferencia de mensajes cortos suele ser más rápida (DNS). |

1.3. API de sockets en C

Todo apartado que se conecte a la red utilizará estas dos funciones

- `int socket(direccionamiento, tipo, protocolo)` crea un nuevo socket
 - **direccionamiento** determina el formato de las direcciones, en internet IPv4 es `AF_INET`
 - **tipo** determina TCP (`SOCK_STREAM`) o UDP (`SOCK_DGRAM`)
 - **protocolo** especifica el protocolo para un tipo (normalmente solo hay uno)
- `int bind(fd, struct sockaddr *local_addr, int addr_len)` registra un socket en un puerto y una IP (un servidor podría tener varias tarjetas de red y por tanto varias IPs)

- `local_addr` especifica IP y puerto, si se quiere escuchar en cualquier IP se pone 0.0.0.0
- `addr_len` especifica la longitud de la dirección (importante para soportar varios tipos)

En este punto un *cliente UDP* ya está listo para enviar con `sendto(fd, data, data_len, flags, dest_addr, dest_addr_len)` y recibir con `recvfrom(fd, buf, buflen, flags, source_addr, source_addr_len)`.

Un servidor UDP solo tendría que hacer `listen(fd, queue_size)` y proceder a recibir datos como un cliente (utilizar `recvfrom`).

Un cliente TCP tendría que conectarse con `connect(fd, dest_addr, dest_addr_len)` y luego ya podría enviar y recibir bytes como si de un fichero se tratara con `read` y `write` o con `send` y `recv`.

Un servidor TCP tendría que esperar conexiones con `accept(fd, source_addr, source_addr_len)` que es bloqueante.

- Se rellena la estructura `source_addr` con la dirección y puerto del cliente.

Tanto cliente como servidor cierran conexiones con `close(fd)`.

1.3.1. Arquitectura de servidores

- **Iterativos:** solo pueden trabajar con una conexión a la vez.
- **Paralelos:** trabajan con más de una conexión a la vez. Se pueden implementar de varias formas:
 - Hacer `fork()` o `pthread_create()` cuando llega una nueva conexión
 - Tener un *pool* de procesos o hilos creado previamente y asignar cada conexión entrante a un miembro del pool. El pool puede redimensionarse dinámicamente.

1.4. Protocolos con arquitectura Cliente/Servidor

| Protocolo | Puerto(s) | Uso |
|------------|-----------------------------|---|
| HTTP/HTTPS | 80/443 seguro | Páginas web |
| SSH | 22 | Terminal remota |
| FTP | 21 (control) / otro (datos) | Transferencia de archivos |
| SMTP | 25 | Envío de correo (servidor - servidor) |
| IMAP | 143/993 seguro | Consulta de correo (cliente - servidor) |
| POP3 | 110/995 seguro | Consulta de correo (cliente - servidor) |
| DNS | 53 | Traducción dominios -> IPs |
| DHCP | 67 servidores / 68 clientes | Gestión de redes LAN (asignación de IPs dinámicas, configuración de DNS, gateways...) |

1.5. Protocolos con arquitectura P2P (peer to peer)

| Protocolo | Puerto(s) | Uso |
|------------|------------|---------------------------|
| BitTorrent | aleatorios | Transferencia de archivos |
| Chord | | Tablas hash distribuidas |

1.6. HTTP

- Protocolo de transferencia de hipertexto.
- Transfiere archivos identificados por URLs o URIs a través de internet.
- Modelo de transferencia: Una **petición - respuesta** por cada recurso que se quiera obtener. La petición siempre la manda el cliente y la respuesta el servidor.
- Normalmente a partir de un fichero principal con formato HTML se generan más peticiones para otros archivos (imágenes, audio) que debe ir incluidos al dibujar la página.
- **Modo no persistente** cada conexión solo se utiliza para una petición/respuesta.
 - Es el por defecto en la versión 1.0.
 - *Overhead* innecesario de establecimiento de conexión para cada recurso que se necesite.
- **Modo persistente** cada conexión se puede utilizar para más de una petición/respuesta.
 - Es el por defecto en la versión 1.1.

- Por separado de lo anterior, los navegadores pueden decidir establecer múltiples conexiones para acelerar la descarga de grupos grandes de recursos. Estas conexiones pueden utilizar cualquier versión.
- Por separado de lo anterior, los navegadores pueden utilizar *pipelining* que consiste en enviar a través de una misma conexión nuevas peticiones sin que se haya recibido la respuesta de la anterior. Es bueno porque aprovecha mejor los paquetes TCP.

Formato de la **cabecera de petición**:

```
VERB /resource?query_string HTTP/VERSION\r\n
Header-Name: header-value\r\n
[more headers]\r\n
\r\n
```

Como mínimo es necesario enviar las cabeceras de **Host** y **Allow**.

Formato de la **cabecera de respuesta**:

```
HTTP/VERSION STATUS-CODE STATUS-MESSAGE\r\n
Header-Name: header-value\r\n
[more headers]\r\n
\r\n
```

Como mínimo es necesario incluir la cabecera **Content-Type** y suelen enviarse además **Last-Modified**, **Server**, **Date** y **Content-Length**.

1.7. FTP

- Transferencia de archivos entre ordenadores.
- Utiliza dos conexiones:
 - Control: la crea el cliente y se envían comandos para autenticarse, inspeccionar y modificar directorios e iniciar transferencias de archivos.
 - Datos: se envían los datos que dicten los comandos enviados por la conexión de control.
 - Modo activo (por defecto): el servidor inicia las conexiones de datos. Da problemas de seguridad y no es posible si el cliente está tras un proxy, firewall o NAT.
 - Modo pasivo (lo solicita el cliente): el cliente inicia las conexiones de datos. Requiere que se habilite un rango de puertos en el servidor para poder recibirlas.
- Servicio de reinicio: algunos servidores soportan la creación de *marcadores* para que en caso de fallo durante la transferencia de un archivo largo, sea posible transmitir de nuevo solo los datos a partir del último marcador recibido por el destinatario.

1.8. SMTP

- *Simple mail transfer protocol*: envío de correo entre hosts.
- Basado en ASCII, muy similar a HTTP.
- Cada email tiene una serie de cabeceras y diferentes *adjuntos*. El cuerpo del mensaje se codifica como un adjunto.

1.9. POP3 e IMAP

- Protocolos de consulta y gestión del correo.
- POP3: Más antiguo y menos sofisticado, permite la lectura y borrado de mensajes (no mantiene el estado fuera del borrado de mensajes).
- IMAP: Más moderno, permite la organización de mensajes en carpetas, marcar como leído/no leído... (en este sentido mantiene el estado).

1.10. DNS

- Resolución de nombres de dominio
- Sobre UDP entre cliente-servidor y sobre TCP entre servidor-servidor.
- Recursivo: el servidor resuelve las consultas adicionales necesarias para obtener la IP de un dominio.
- Iterativo: el cliente debe preguntar a los distintos servidores (local, root, TLD, dominio) para obtener la IP del dominio.