

Apuntes ASOR

Redes

Tema 1 – IPv4. Protocolo DHCP

Def. Protocolo IP: Protocolo de red de internet, proporciona un servicio básico de entrega de paquetes, está no orientado a conexión (no fiable), esto quiere decir que:

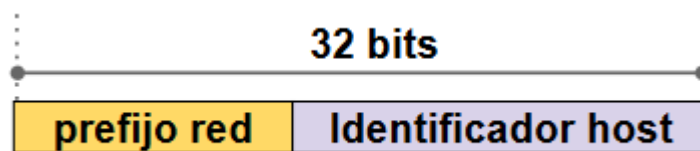
- No realiza detección ni recuperación de paquetes perdidos o erróneos.
- No garantiza que los paquetes lleguen en orden.
- No garantiza la detección de paquetes duplicados.

Las funciones básicas de este protocolo son:

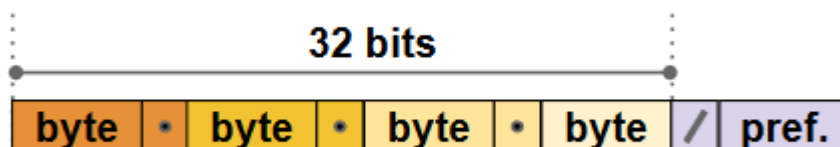
- Direccionamiento (Necesidad de seguir un esquema global).
- Fragmentación y reensamblaje de paquetes (tamaño aceptable por la red).
- Encaminamiento de datagramas (reenvío de paquetes según la info. de la tabla de rutas y uso de protocolos para la construcción de estas tablas (RIP, OSPF, etc)).

Direccionamiento

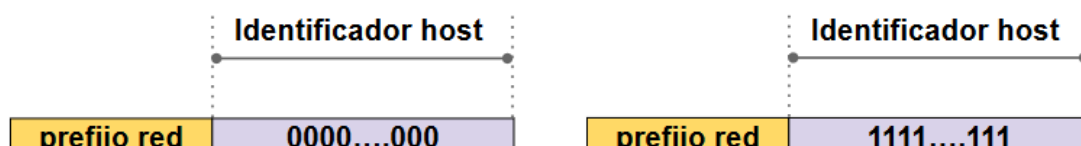
Las direcciones IP constan de 4B (32b), para expresarlas se usa la notación de punto, los primeros bits de la dirección denotan en prefijo de red, los últimos el host.



Con el uso de CIDR se alivia el problema del agotamiento de direcciones y se elimina la estructura fija basada en clases, ahora el espacio de dir. se divide en bloques de tamaño arbitrario y se usa una notación barra que incluye la long. del prefijo de red de una dir. IP, determinado por la máscara de red.



Distinguimos entre dir. de red, que se usan para representar una red completa en las tablas de encaminamiento y nunca se usan como dir. destino ni se asignan a un host, y dir. broadcast, usadas para enviar un paquete a todas las máquinas de una red local.



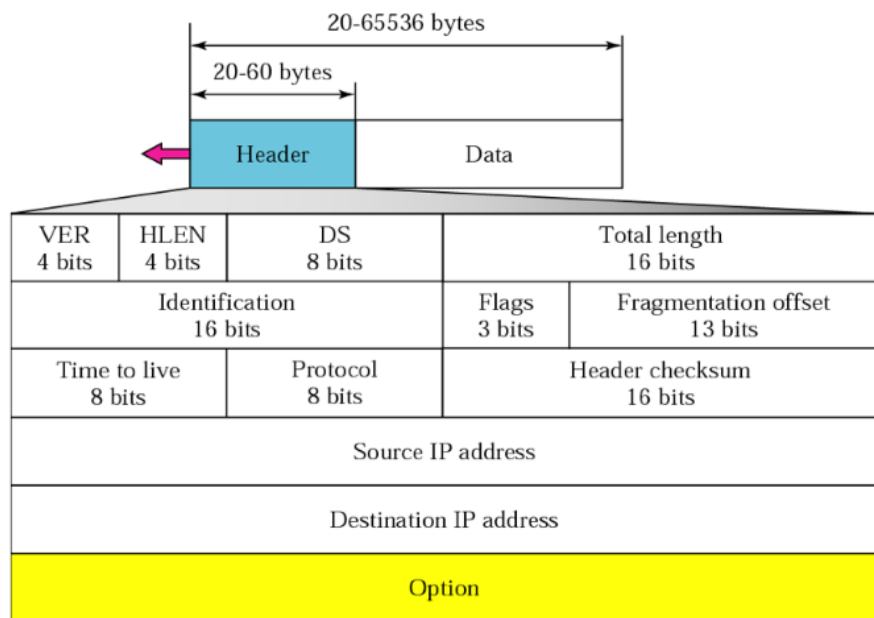
Def. Direcciones privadas: cjto de dir. reservadas para uso privado, no son válidas para su uso en internet, los rangos de direcciones privadas son:

- 10.0.0.0 – 10.255.255.255 (1 red privada de clase A).
- 172.16.0.0 – 172.31.255.255 (16 redes privadas de clase B).
- 192.168.0.0 – 192.168.255.255 (256 redes privadas de clase C).

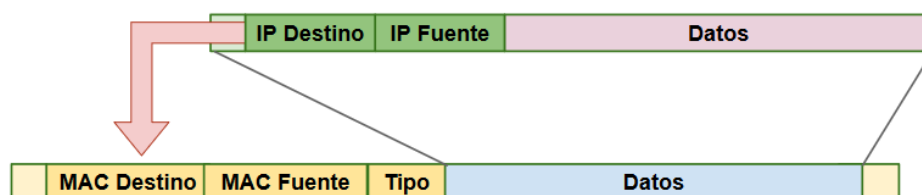
Def. Direcciones multicast: Identifican de forma lógica un grupo de hosts en el segmento de red, su formato es 224.0.0.0/4, por ejemplo:

- 224.0.0.1 (todos los hosts).
- 224.0.0.2 (todos los routers).
- 224.0.0.251 (mDNS).

Formato de datagrama IP



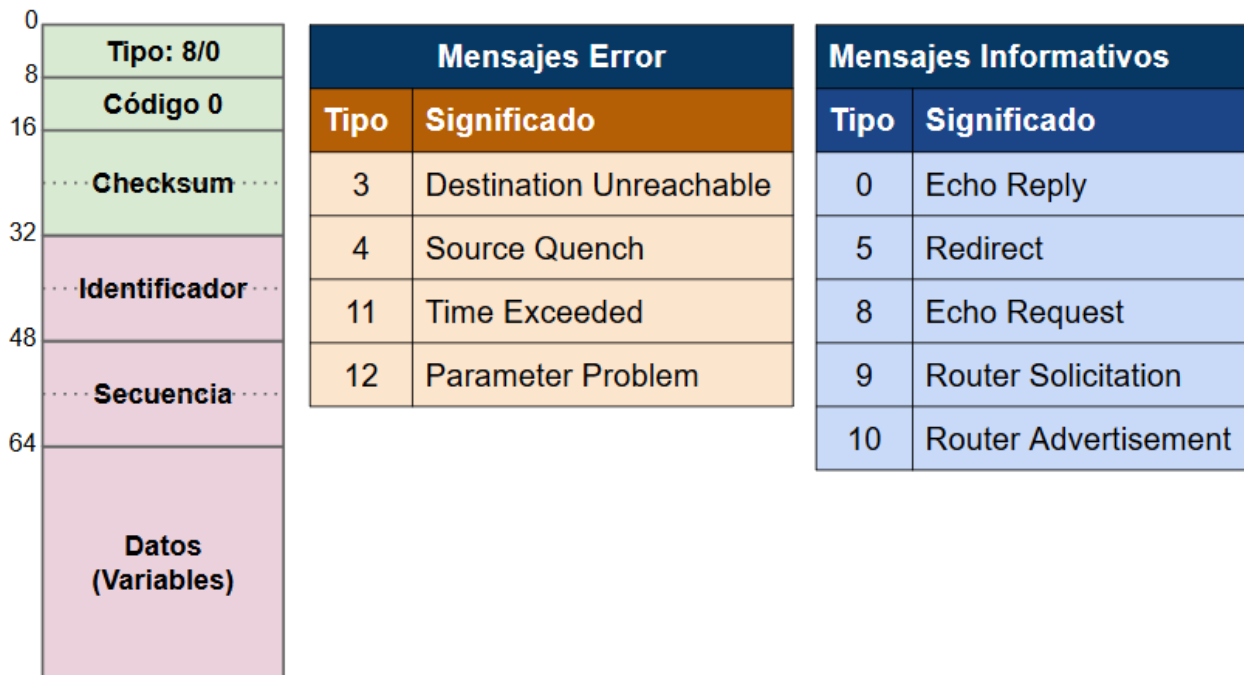
Es importante resaltar la existencia de un protocolo de traducción de dir. llamado ARP cuyo objetivo es establecer la correspondencia entre dir. IP y MAC, este protocolo tiene una tabla ARP en la cual mantiene las dir. IP de las últimas maquinas con las que nos hemos comunicado y las dir. Ethernet (físicas) asociadas.



Net to Media Table				
Device	IP Address	Mask	Flags	Phys Addr
1e0	147.96.48.203	255.255.255.255		00:00:b4:c3:c8:f4
1e0	147.96.37.196	255.255.255.255		00:a0:24:57:78:3e
1e0	147.96.48.217	255.255.255.255		00:20:18:2f:1d:60

Otro protocolo importante es ICMP (protocolo de control de msgs de internet) cuya principal labor es el intercambio de msgs de control en la red, los msgs ICMP se pueden clasificar en dos tipos:

- Error (informan de situaciones de error en la red).
- Informativos (Informan sobre la presencia o estado de un determinado sist.).



Por ejemplo el Echo Reply se utiliza para ver si un computador es alcanzable, su formato es como el que se muestra en la imagen superior, con un campo identificador, que permite establecer la correspondencia entre solicitud (request) y respuesta (reply), otro secuencia (también usado para la correspondencia cuando se envían varios echo requests consecutivos con el mismo identificador) y datos (nº de bytes).

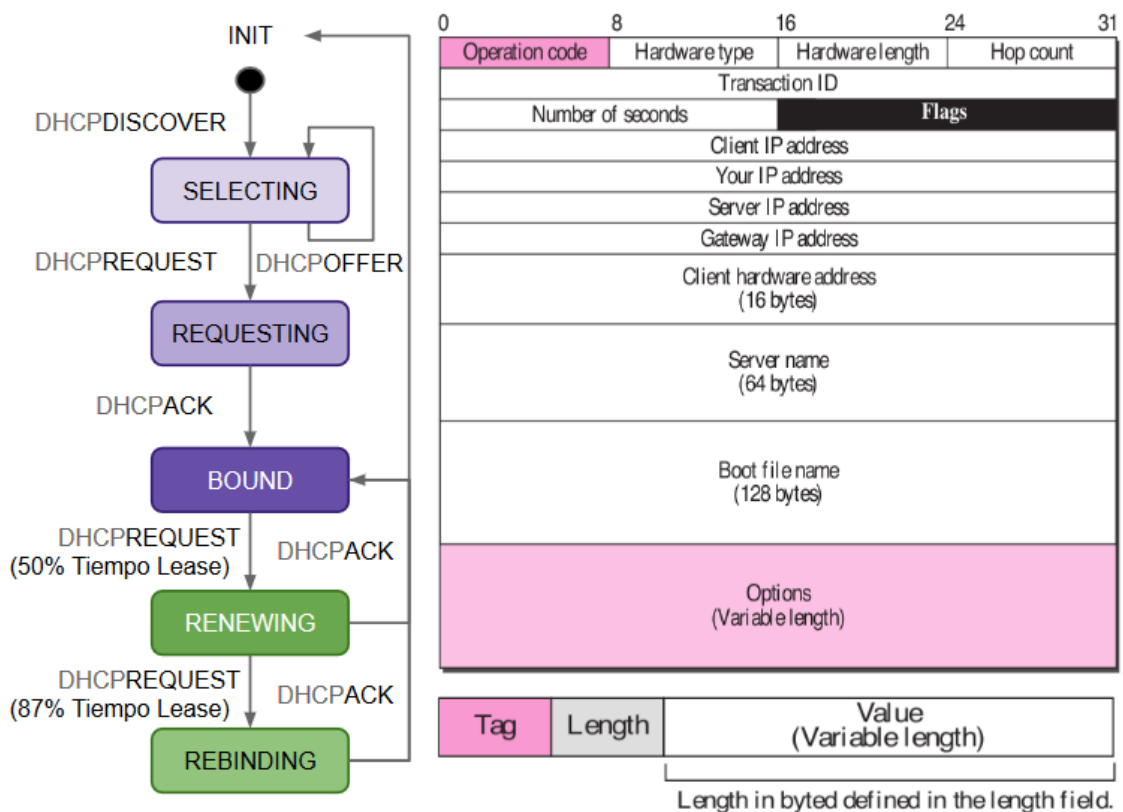
Protocolo DHCP

DHCP (Protocolo de configuración dinámica de hosts), se usa para la configuración automática de los parámetros de la red (dir. IP y máscara de red, router predeterminado, servidores DNS, etc), sus características son:

- Protocolo cliente/servidor sobre UDP en los puertos 67 (server) y 68 (client).
- Control de errores basado en checksum, temporizadores y retransmisiones.
- Protocolo TFTP para la transferencia de ficheros con info. adicional.
- DHCP Relay Agent para servidores/clientes en diferentes redes.

Existen distintos tipos de msgs usados en este protocolo:

- DHCPDISCOVER: Msg del cliente (broadcast) para descubrir los servidores disponibles.
- DHCPOFFER: Respuesta de los servidores con una oferta de parámetros de configuración.
- DHCPREQUEST: Petición de oferta del cliente (broadcast, para notificar a todos los servidores), el servidor seleccionado se especifica con una opción.
- DHCPACK: Msg de confirmación (broadcast) y cierre desde el servidor hacia el cliente indicando los parámetros definitivos.
- DHCPRELEASE: Msg del cliente para informar al servidor de que ha finalizado el uso de la dir. IP.



Tema 2 – Protocolo TCP, conceptos avanzados

Def. Protocolo TCP: Protocolo orientado a conexión que proporciona fiabilidad mediante control de errores a través de acciones correctoras.

Define una serie de fases para la transmisión:

- Establecimiento de la conexión (cliente-servidor).
- Transferencia de datos: Control de errores mediante de ventana deslizante.
- Cierre de conexión.

La unidad de transferencia de este protocolo es el segmento de datos TCP, que contiene un nº de bytes y al cual se le añade una cabecera.

Se usan una serie de mecanismos de control de error de tipo ventana deslizante:

- Checksum (CRC) tanto en los datos como en la cabecera.
- Numeración de segmentos (para una posible reordenación en caso de desorden o para confirmar la falta de segmentos, el nº de segmento viene dado por el nº del 1er byte que contiene el segmento)
- Confirmaciones selectivas y acumulativas (si se confirma la llegada de un segmento se confirma a su vez la llegada de todos los anteriores).
- Retransmisión de segmentos perdidos o erróneos (Si es erróneo se descarta y la entidad correspondiente se encarga de reenviarlo).
- Temporizadores.

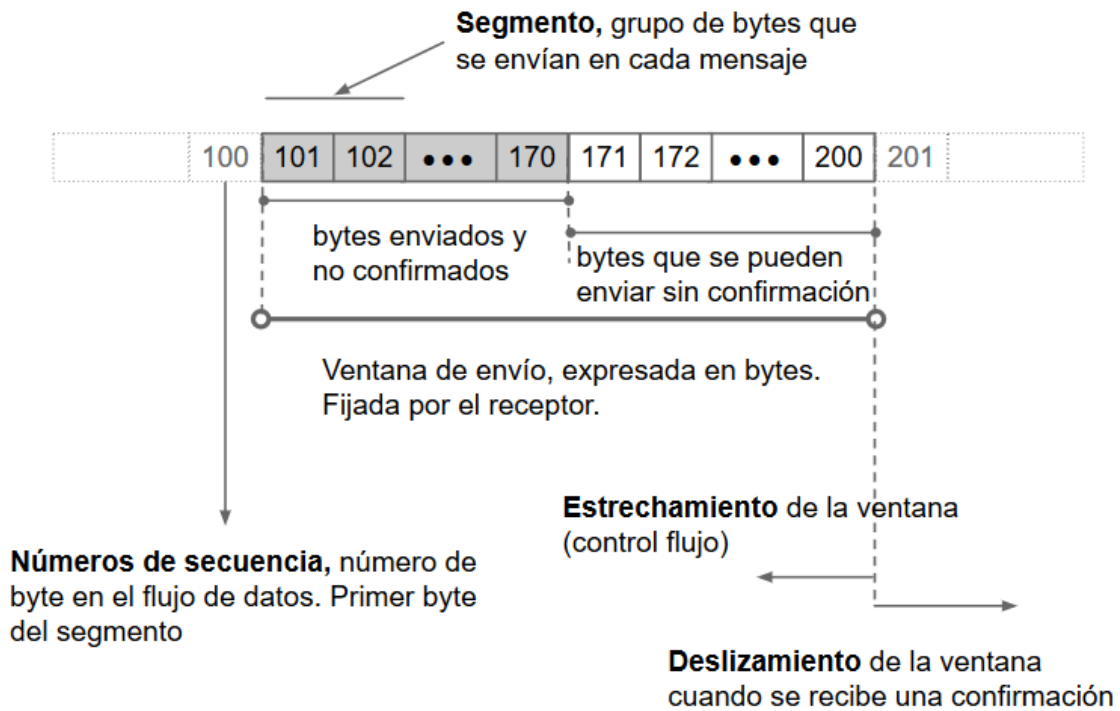
El protocolo TCP ofrece distintos servicios:

- Comunicación lógica proceso-proceso, usando nº de puerto.
- Flujo de datos (stream) para el envío y recepción.
- Transmisión orientada a conexión y fiable.
- Comunicación full-duplex (en ambos sentidos) y multiplexación (el mismo canal puede ser usado para diversas aplicaciones).

TCP hace uso de dos ventanas deslizantes distintas:

- Ventana de envío (emisor): Número de bytes que se pueden enviar a la otra máquina sin confirmación, cuando se establece la conexión el receptor indica el tamaño de su ventana de recepción por lo que la ventana de envío tendrá el mismo tamaño en bytes.

El nº de segmento lo indica el 1er byte contenido, los bytes se marcan como bytes que se pueden enviar sin esperar confirmación o como enviados, conforme los enviados se van confirmando (nº de confirmación) y van saliendo de la ventana por la izquierda, ésta se reduce por la derecha.

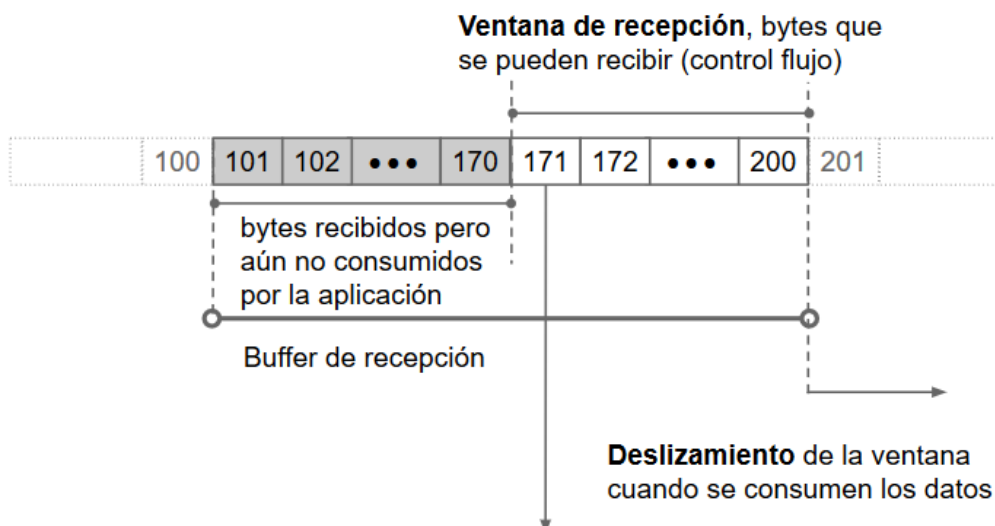


- Ventana de recepción (receptor): El receptor tiene un buffer de recepción en el cual va almacenando los bytes que ha recibido pero que todavía no han sido consumidos por la aplicación, el resto de bytes del buffer (reservados en mem.) es el espacio que tiene ocupado en la ventana de recepción, el resto de bytes son los que podría recibir, el espacio que se anuncia como ventana de recepción, a medida que este espacio se va estrechando o ampliando el receptor lo va anunciando.

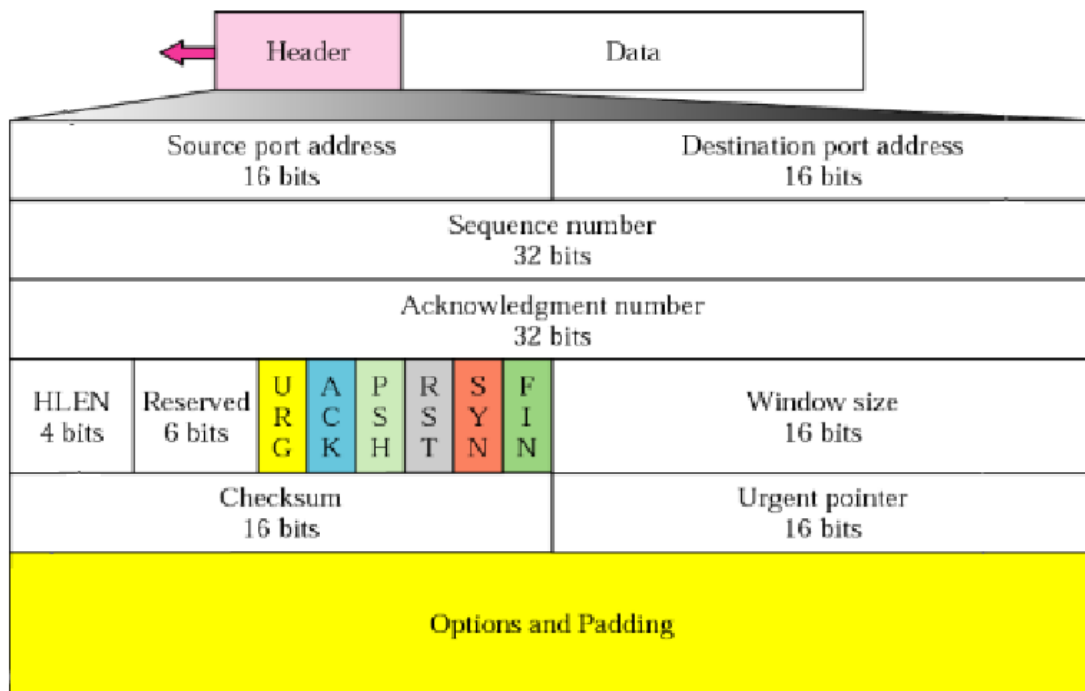
La aplicación siempre va consumiendo los bytes en orden, a medida que se consumen los datos se desliza la ventana a la derecha.

Los ACK vienen determinados por el nº del 1er byte en el flujo de datos que se espera recibir, los ACK se solapan con el envío de datos.

El receptor puede tomar medidas de control de flujo, como anunciar un estrechamiento de la ventana de recepción (y por consiguiente la de envío), caso en el cual la ventana se deberá reducir por la derecha, esto normalmente suele deberse a que no puede consumir los datos tan rápido como los está enviando el emisor, también puede posteriormente volver a ensanchar la ventana.



Formato de datagrama TCP

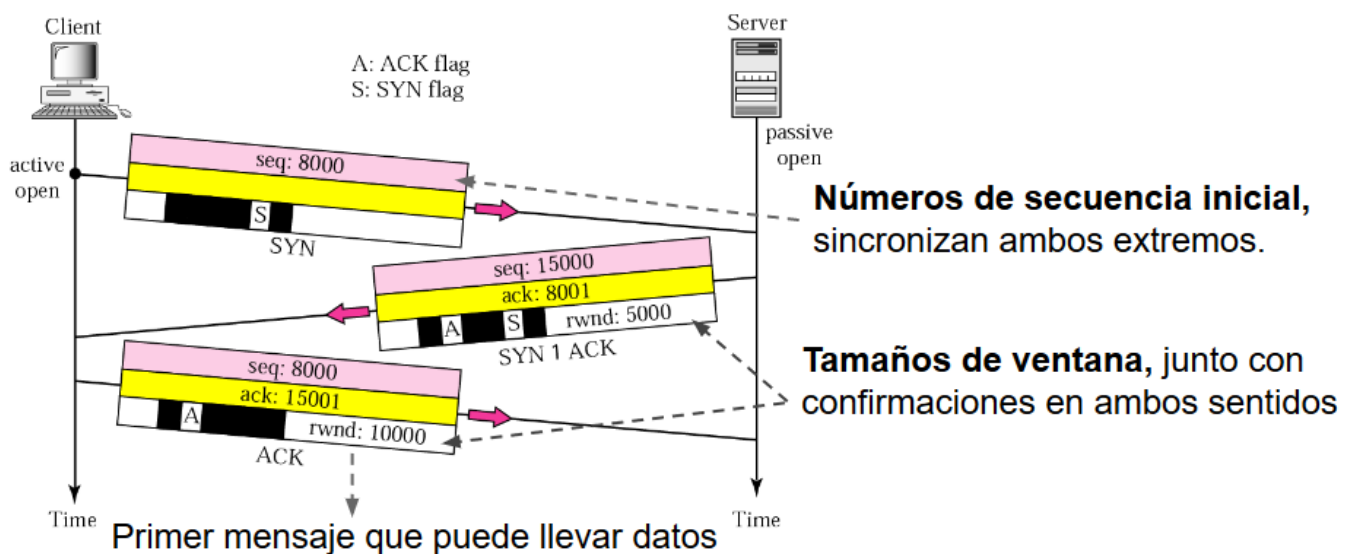


TCP especifica en su datagrama los puertos (origen y dest.), los nº de secuencia (1er byte de datos del segmento), los nº confirmación (ACK), la long. de la cabecera (20-60 bytes), un campo de control que contiene 6 flags que se usan para implementar acciones de control, el tamaño de la ventana de recepción (16b), un checksum y finalmente un puntero asociado a flags.

En cuanto a los flags existentes en el campo de control existen 6 distintos:

- SYN: Usado en el establecimiento de la conexión y para sincronizar los nº de secuencia iniciales (ya que son aleatorios al principio por motivos de seguridad).
- FIN: Usado para finalizar la conexión, cuando una de las partes quiere finalizar la conexión envía un msg con el flag FIN a 1.
- ACK: Indica que el segmento contiene un nº de confirmación válido, en TCP son todos los segmentos salvo el primero.
- RST: Usado para abortar una conexión, por ejemplo, al intentar conectarnos a un puerto cerrado el SO nos mandaría un msg con el flag RST activado (1).
- PSH: Según su valor, indica que los datos deben ser entregados inmediatamente a la aplicación sin almacenarse a la espera de más datos (1) o que pueden almacenarse en el buffer (0).
- URG: Indica que el segmento transporta datos urgentes (1), esto se hace para indicar que estos datos se deben procesar en el momento, antes que todos los bytes enviados anteriormente. Se usa un puntero llamado urgent pointer para indicar en qué punto del segmento acaban los datos urgentes a procesar.

En cuanto a la fase de establecimiento de conexión se usa el mecanismo 3-way handshake, en el cual se usan 3 msgs para establecer la conexión, el cliente (inicio activo de la conexión) envía un segmento con el flag SYN a 1 (indica un nº de secuencia generado aleatoriamente) la otra parte, el servidor (inicio pasivo) pasa a un estado de espera de conexiones (Listen) en la cual le llegan las peticiones de conexión, al aceptar una envía otro msg anunciando por una parte el flag de sincronización SYN, generando otro nº de secuencia aleatorio y por otra el flag ACK confirmando que le ha llegado el msg anterior del cliente, finalmente el cliente envía otro msg ACK con un byte más que el nº de secuencia del msg SYN-ACK del servidor anunciando el tamaño de su ventana de recepción.

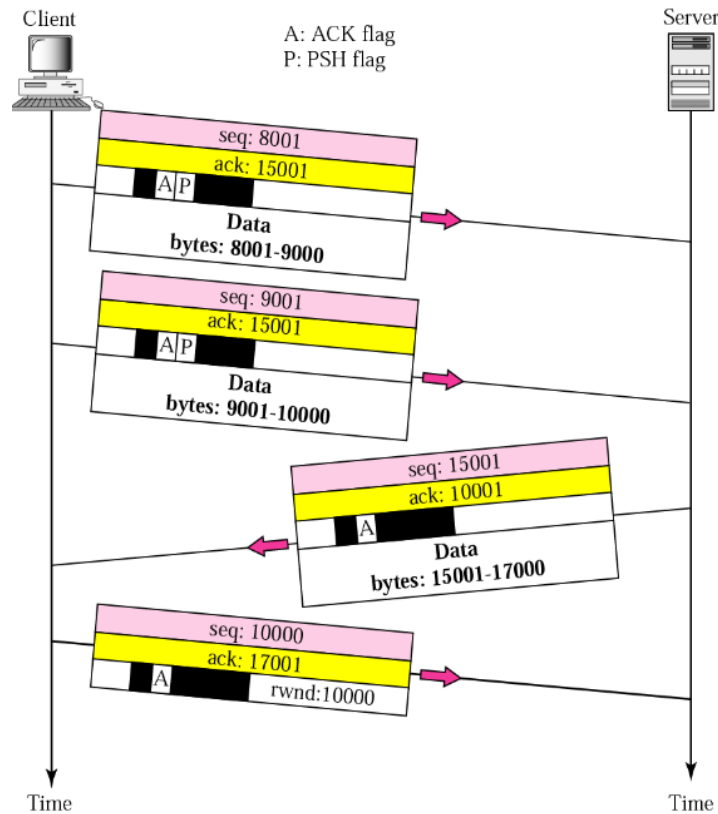


En TCP existe una vulnerabilidad llamada SYN Flood que puede provocar un ataque DDOS, consiste en que uno o varios clientes envían una cantidad ingente de msgs de inicio de conexión con el flag SYN activo (1), el servidor suele tener una lista de conexiones sin terminar (SYN backlog) y suele reservar un nº de conexiones que pueden estar en proceso, si muchas máquinas inician una conexión pero no la terminan pueden llegar a llenar la tabla del SYN backlog y cuando un cliente quiera establecer una conexión de manera normal no podrá hacerlo, a todo esto se suma que al iniciar una conexión se suelen reservar recursos, por lo que se puede llegar incluso a saturar el servidor.

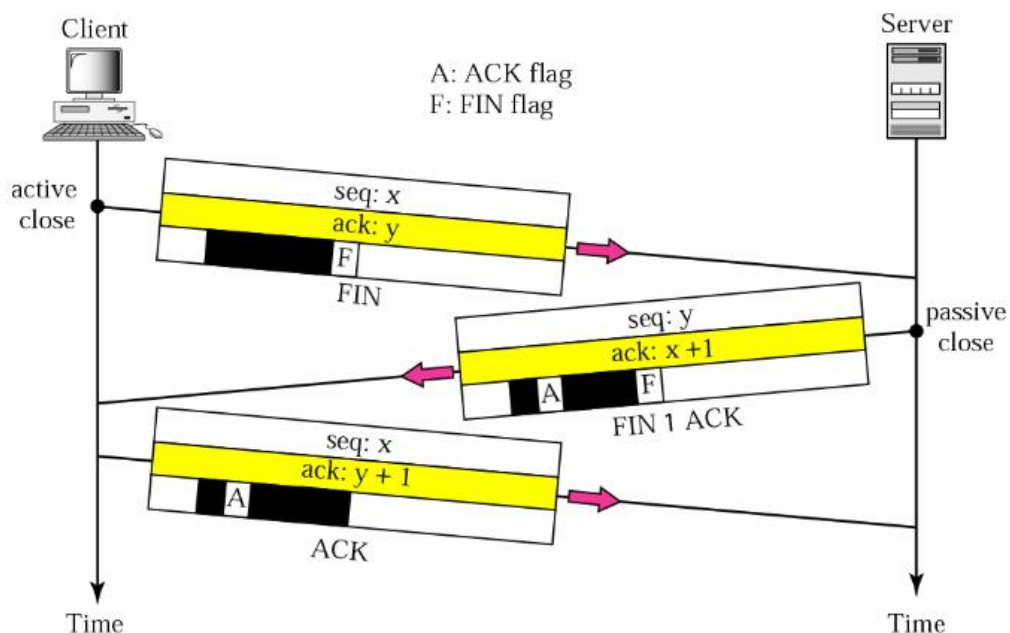
Existen varias soluciones a este problema, pueden ser limitar el nº de conexiones, aceptar conexiones solo de IP's fiables (cortafuegos) o la más idónea es retrasar la asignación de recursos usando el mecanismo SYN cookies que permite aceptar más conexiones de las que se reservan y evita usar una entrada en la SYN, la contrapartida de este mecanismo es su consumo de CPU.

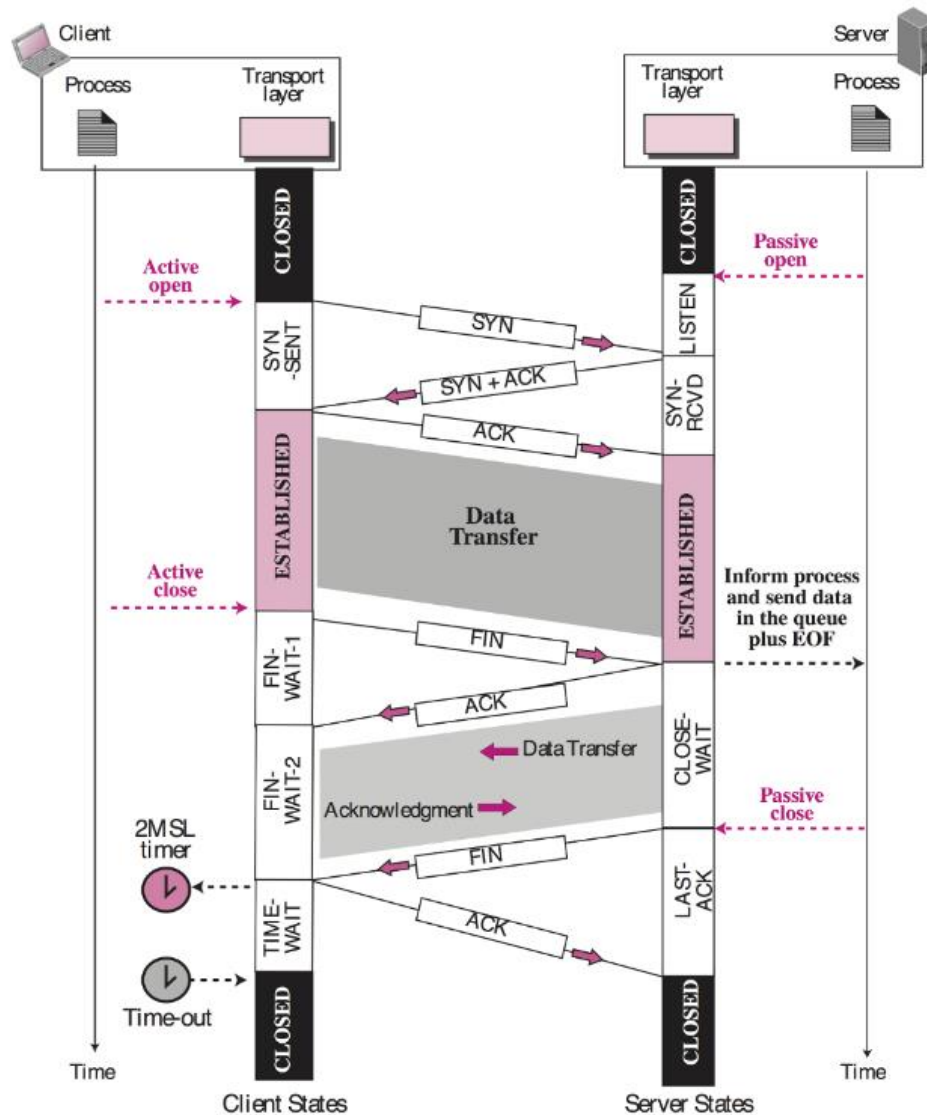
En la fase de transferencia una vez hecha la conexión se envían segmentos de datos con el flag ACK activo, y con el flag PSH (activo o no), además del nº de SEQ.

En la imagen inferior se ve el retraso de la confirmación hasta el msg correspondiente al 2do segmento de datos (ACK's acumulativos), el tamaño max. de segmento lo fija cada extremo en los msgs SYN, en un campo opción del datagrama TCP.



La fase de finalización se puede realizar de varias maneras, mediante la finalización con 4 msgs (4-way) o mediante una optimización de este protocolo de 4 vías en la cual una parte solicita el cierre de conexión, la otra esta de acuerdo en cerrarla y entonces aprovecha para enviar en el mismo msg los flags FIN y ACK, por lo que los msgs acaban siendo 3 (3-way).



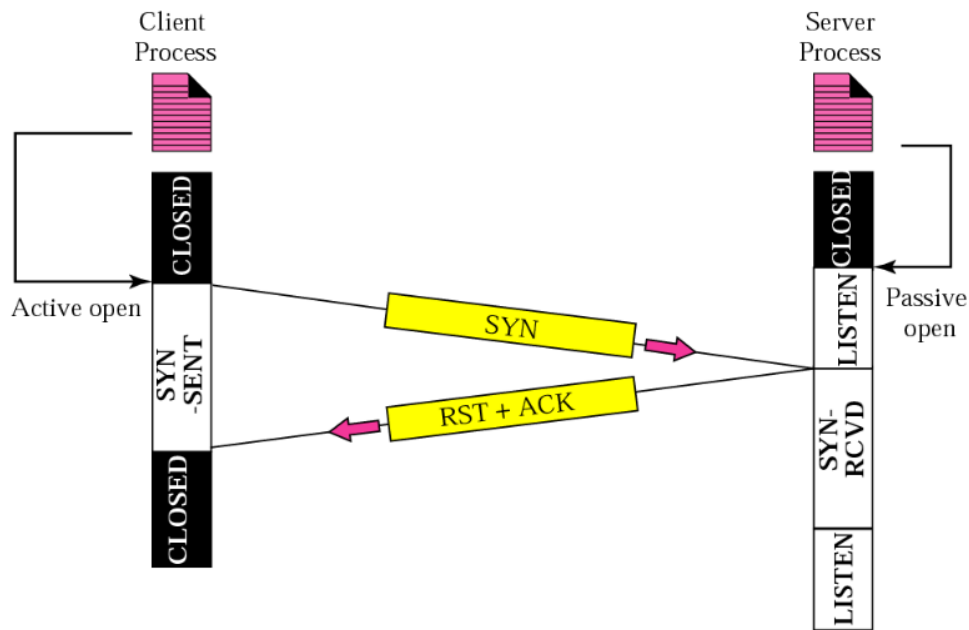


Inicialmente la conexión está cerrada (CLOSED), normalmente los servidores hacen un inicio pasivo y entran en estado LISTEN escuchando peticiones de clientes, por otra parte los clientes hacen la apertura activa, enviando un msg SYN al servidor y pasando al estado SYN-SENT, el servidor al recibir el msg SYN envía un msg de respuesta con SYN y ACK y pasa al estado SYN-RCVD, el cliente que estaba en SYN-SENT recibe el msg con ACK y pasa al estado ESTABLISHED, a su vez responde al servidor con un msg ACK y entonces el servidor pasa también al estado ESTABLISHED.

Si el servidor después de recibir el SYN no recibe el ACK puede decidir tras un tiempo abortar la conexión con un RST (time-out)).

Por último, para el cierre, desde el estado ESTABLISHED tanto el servidor como el cliente pueden iniciar el cierre activo, el cliente enviaría el FIN y pasa al estado FIN-WAIT1 y el servidor enviaría el ACK pasando al estado CLOSE-WAIT, el cliente recibe el ACK y pasa al estado FIN-WAIT2, después solo queda que el servidor cierre enviando un FIN, que lo reciba el cliente y mande un ACK pasando al estado temporal TIME-WAIT, el servidor se pasaría al estado LAST-ACK esperando que el cliente cierre la conexión enviando un último ACK, es mejor que el cliente cierre la conexión para que él se quede en un estado TIME-WAIT.

Esquema aborto de conexión por parte del servidor, que envía RST + ACK.



El protocolo TCP hace uso de un mecanismo de control de errores para gestionar la recepción de segmentos duplicados, erróneos, perdidos o fuera de orden.

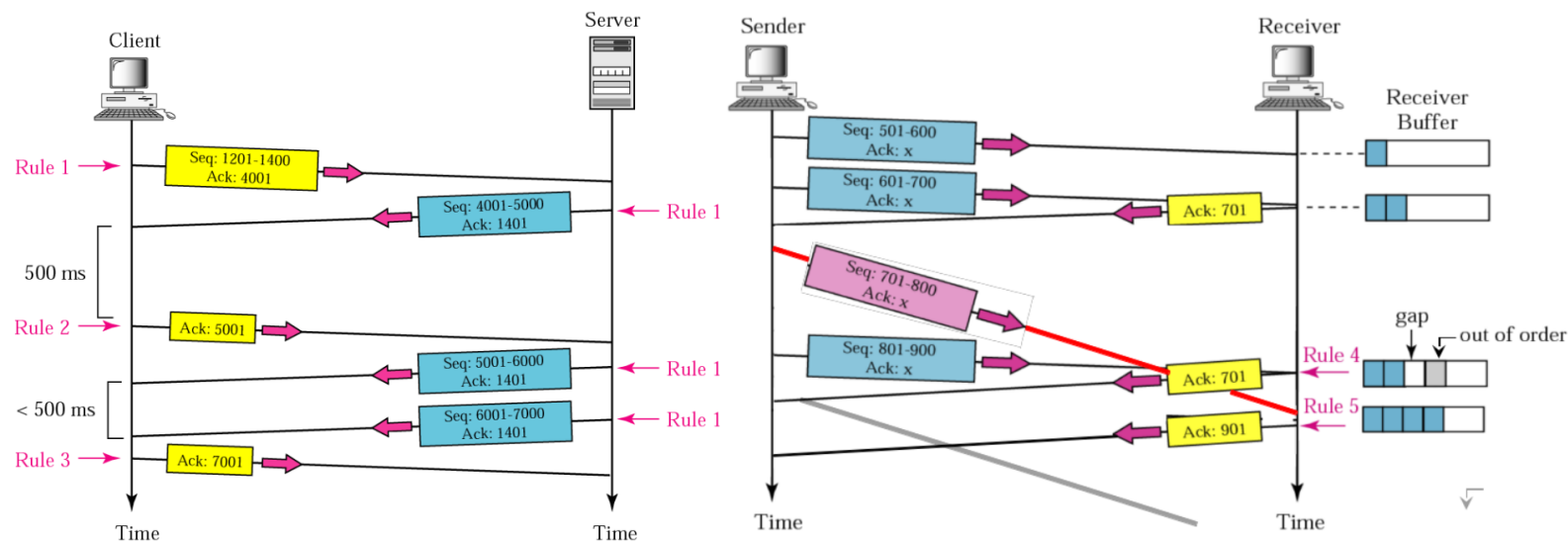
Este mecanismo se basa principalmente en las confirmaciones (ACK) que indican que un segmento de bytes se ha recibido, en otro caso el segmento se ha descartado y habrá que volver a retransmitirlo.

En TCP existen una serie de características de los ACK's:

- Son acumulativos.
- Se pueden retrasar (en caso de que no haya datos que enviar) para darle más tiempo al receptor para que genere algún dato y lo envíe.
- No puede haber más de 2 segmentos sin confirmar.
- Los segmentos fuera de orden, los que completan huecos y los duplicados se confirman inmediatamente para prevenir pérdidas de ACK's.

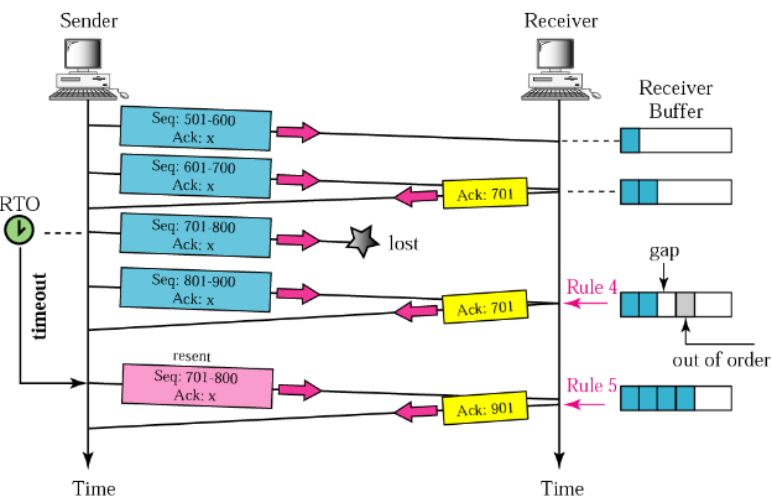
Transmisión sin error

Recepción fuera de orden

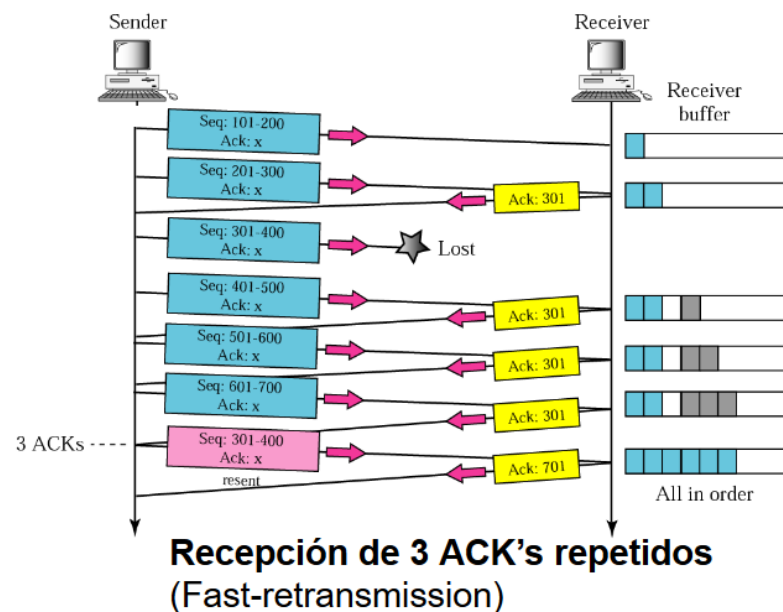


La capacidad para transmitir un segmento cuando no se recibe o se recibe erróneamente es el núcleo del control de errores de TCP, que dispone de dos mecanismos de retransmisión:

- Temporizador de Retransmisión (RTO):
 - Cada conexión tiene asociado un único temporizador.
 - Cuando el temporizador expira se envía el primer segmento sin confirmar de la ventana de envío.
 - Para fijar el RTO (valor dinámico) existen diversos algoritmos, y siempre debe ser mayor que el RTT (round-trip time), que es el tiempo que tarda un segmento de datos en ser enviado y recibir un ACK de vuelta.
- Retransmisión por recepción de 3 ACK's duplicados.
 - Mecanismo de retransmisión más rápido, no requiere que expire el RTO.



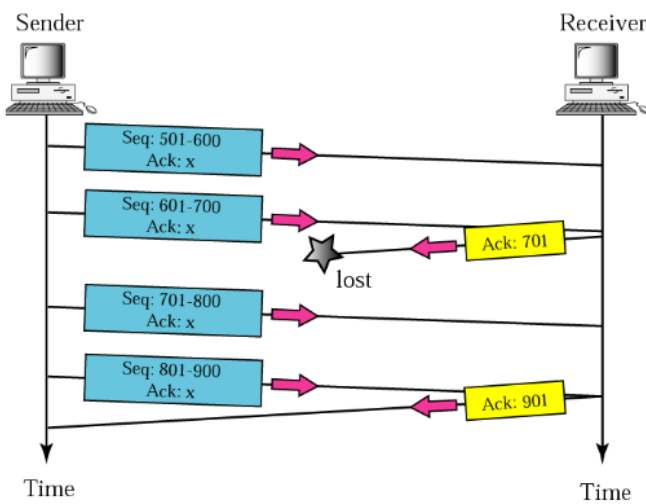
Temporizador RTO expirado



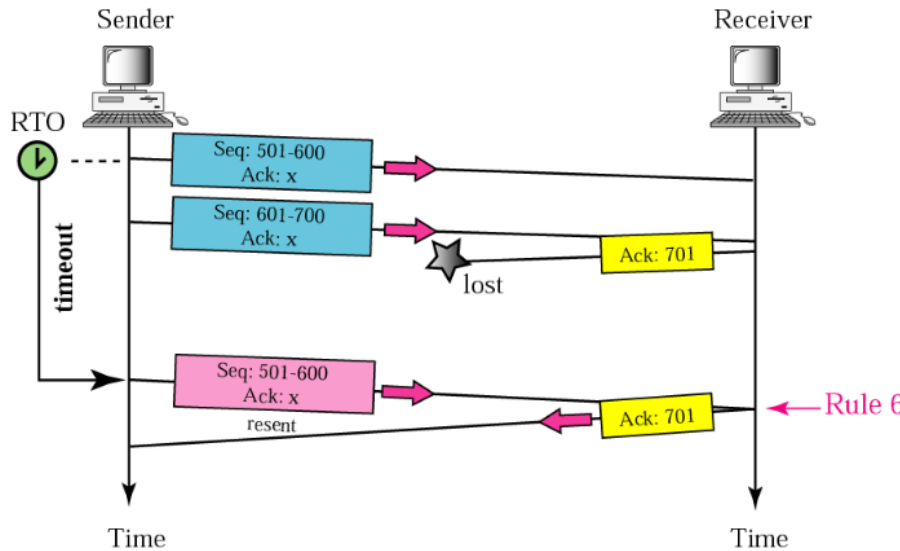
Recepción de 3 ACK's repetidos (Fast-retransmission)

Se puede perder un segmento en dos situaciones:

- En el caso del RTO el cliente deja de enviar datos y el temporizador expira, esto provoca el reenvío del primer segmento sin confirmar (701).
- En el otro caso el cliente sigue enviando datos sin confirmar por lo que al recibir 3 ACK's duplicados se retransmite el 1er segmento de datos sin confirmar.



Sin expirar el temporizador RTO



Temporizador RTO expirado

Se puede perder un segmento de confirmación (ACK) en dos situaciones:

- Que el cliente siga enviando datos, por lo cual el receptor los seguirá confirmando y al ser los ACK's acumulativos el siguiente ACK confirma los anteriores.
- Que el cliente no envíe más datos, por lo que acaba expirando el RTO y el receptor debe enviar una confirmación inmediata de ese segmento para evitar más retransmisiones.

TCP usa además del RTO otros tres temporizadores:

- Keepalive: Se usa para no tener conexiones inactivas indefinidamente, si uno de los extremos no envía datos durante un tiempo el otro envía una serie de pruebas para ver si el otro responde, si no lo hace el primero de los dos extremos acaba cerrando la conexión.
- TIMEWAIT: Controla el tiempo en el que se permanece en este estado cuando se cierra la conexión, se usa por dos razones, una, si hay que volver a enviar el último ACK porque el otro extremo retransmite FIN y otra, para prevenir colisiones de dos conexiones que usen los mismos parámetros (nº de secuencia), esto puede ser problemático si es el servidor el que cierra la conexión porque genera muchas conexiones en estado TIME-WAIT, como remedio se puede reducir el MSL.
- Temporizador de persistencia: Se usa cuando el receptor anuncia un tamaño de ventana 0, con lo cual el emisor deja de enviar, el receptor consume datos y anuncia un ACK junto con un tamaño de ventana mayor que 0, si ese ACK enviado se pierde, la conexión queda bloqueada al creer el emisor que la ventana es 0 y esperar el receptor alguna señal del emisor, para evitar esto se envía una sonda que fuerza el envío de este ACK, así se evita el interbloqueo.

La elección del tiempo de vencimiento del RTO está basada en los retardos observados en la red, que pueden variar dinámicamente, por lo que el RTO debe adaptarse a esta situación, por ello el RTO se basa en el RTT (Round trip-time) que refleja estos retardos

Primero se calcula el RTTm, que mide el tiempo transcurrido desde que se envía hasta que se recibe el ACK, este valor del RTTm puede variar mucho, por lo que también se calcula el RTTs, que es una media ponderada entre el RTTm y la última media calculada, además se calcula la desviación (RTTd) para ver si es muy grande o pequeña.

La ambigüedad en el cálculo del RTTm se resuelve con el uso de TCP timestamps (el emisor coloca una marca de tiempo (timestamp) en cada segmento de datos y el receptor refleja estas marcas en los ACK's, así tan solo con una resta le da al emisor un valor del RTT para cada segmento de ACK).

TCP también usa mecanismos para el control de flujo, que es el encargado de controlar la tasa de envío de datos del emisor para no saturar la del receptor, evitando así la sobrecarga, este control de flujo se realiza mediante la ventana de recepción, anunciada en cada ACK.

Existen dos casos en los que puede darse una situación llamada síndrome de la ventana trivial:

- Cuando la aplicación emisora genera datos a un ritmo muy lento (ej. TELNET) byte a byte hasta un total de más de 160B (un tiempo muy largo) lo que acaba resultando absurdo, para evitarlo se usa una técnica de buffering llamada algoritmo de Nagle, en el cual el emisor envía un primer msg y los siguientes se retrasan hasta que, o se recibe un ACK del receptor, o se acumulan MSS bytes de la aplicación o se expira el temporizador RTO.
- Otro cuando la aplicación consume datos a un ritmo muy lento, en este caso el receptor va anunciando tamaños de ventana muy pequeños, para evitar esto se hace uso del algoritmo de Clark, con el cual se anuncia un tamaño de ventana = 0 que no se puede cambiar hasta que se pueda recibir un segmento completo (MSS) o hasta que se haya liberado la mitad del buffer de recepción. Otra opción es activar los ACK retardados.

Otro mecanismo de TCP es el control de la congestión, que tiene en cuenta la capacidad de la red y se centra en los encaminadores, que tienen la capacidad de detectar cuando no son capaces de encaminar los paquetes al mismo ritmo que los reciben, entonces evitan llegar al punto de saturarse y empiezan a descartar paquetes (incluidos los ACK's).

Para hacer el control de la congestión se usa el ritmo de llegada de los ACK's para regular el ritmo de envío de paquetes, la ventana de congestión (CW) controla la saturación en la red, empezando en un valor muy bajo y aumentando según va recibiendo ACK's, según se detecta tráfico en la red se va disminuyendo su tamaño, su tamaño máximo (en caso de no haber congestión en la red) es el de la ventana de recepción.

Se dice que una red está sin congestión cuando no se pierden o se retrasan segmentos.

La CW empieza con un valor = 1 y va aumentando pasando por tres fases:

- Arranque lento: La CW va incrementando su tamaño por cada segmento enviado y confirmado formando un crecimiento exponencial (arranque no lento), hasta llegar al umbral SST (Slow Start Threshold), que suele ser de 64 KB.
- Evitación de congestión: La CW crece pero ya no de forma exponencial sino lineal (crece en 1 cuando recibe todos los ACK's de los segmentos que ha enviado), hasta llegar a su tamaño max. (tamaño de la ventana de recepción).
- Constante: La CW se mantiene a un valor constante (CW = ventana de recepción).

Si se da en la red una situación de congestión se detecta indirectamente, ya los routers notifican de la detección de niveles altos de congestión, en este caso se pueden tomar dos medidas distintas:

- Recepción de 3 ACK's duplicados: Se asocia a un nivel de congestión leve, se activa el método de recuperación rápida, en el cual se divide el valor de la CW a la mitad y se ejecuta la fase de evitación de congestión.
- Expiración del temporizador de transmisión (RTO): Se asocia a un nivel de congestión elevado (tráfico interrumpido) y se toman acciones más drásticas, se inicializa la CW a 1, se reduce el umbral SST a la mitad del valor de la CW antes de producirse el time out y se ejecuta la fase de arranque lento de la CW.

Tema 3 – Servicios de red: Filtrado de paquetes

Def. Firewall: Componente de seguridad HW-SW que analiza el tráfico de red y determina si debe permitir su paso, entre sus funciones está el filtrado de paquetes de red, el registro de actividad (ficheros de log, generar alarmas...) y la traducción de direcciones.

Existen distintos tipos de firewalls:

- Basados en host: Analizan el tráfico que entra o sale del host para decidir si lo bloquean o permiten su paso, en este tipo de firewalls lo que se desea es controlar los paquetes que entran o salen del host.
- Basados en red: Analizan el tráfico que entra o sale de la red para decidir si se bloquea o si se permite su paso, conectan varias redes con distintos niveles de seguridad, también suelen proporcionar traducción de dir. privadas a dir. públicas.

También se distinguen por otros criterios:

- Si son capaces o no de manejar el estado de las conexiones:
 - Stateless: El filtrado se basa únicamente en el contenido de los paquetes de red, no tiene en cuenta el estado de las conexiones.
 - Stateful: Además de tener en cuenta el contenido de los paquetes también tiene en cuenta si el paquete está intentando iniciar una nueva conexión o si la conexión ya está iniciada y es un paquete que forma parte de esa conexión, para esto se mira si el paquete lleva el flag SYN activado.
- En función de la capa en la que trabajan:
 - De red: Permiten construir reglas para inspeccionar los paquetes.
 - SPI: Solo inspecciona la info. contenida en las cabeceras de los paquetes de los protocolos TCP, UDP, ICMP, etc, pero sin inspeccionar la carga del paquete (los datos que contiene).
 - DPI: Inspecciona en profundidad el paquete, por ejemplo, el protocolo de aplicación, en el caso de que sea HTTP se usan WAF que detectan cualquier tipo de amenaza o ataque que pueda ir en este protocolo y también se usan IDS que son sist. que pueden funcionar internamente o a nivel de firewall haciendo un sniffing de todo el tráfico analizándolo en busca de intrusiones o paquetes sospechosos.
 - De aplicación
 - De transporte (SOCS)

Def. Iptables: Herramienta de filtrado de paquetes de Linux, es una tabla que contiene todas las reglas organizadas en tablas a su vez y en cadenas, con este comando se pueden manipular distintas reglas del firewall que filtran qué paquetes pueden entrar o salir y cuales pueden atravesar el firewall en un sentido u otro.

Iptables está compuesta de:

- Reglas: Definen qué hacer con un paquete (descartarlo o aceptarlo), se pueden usar multitud de criterios como la dir. origen, la dir. destino, el protocolo, etc.
- Cadenas: Son listas de reglas que se van aplicando a un paquete, todo paquete atraviesa siempre por lo menos una cadena, ya que en caso de que no se le aplique ninguna de las reglas se usa la política por defecto de la cadena, que puede ser descartar o aceptar, se aconseja que sea descartar.
- Tablas: Son cjtos de cadenas que se agrupan en función de la funcionalidad que proporcionan.

Existen una serie de tablas y cadenas predefinidas:

- Tabla Filter: Bloquea o permite el tránsito de un paquete (filtrado de paquetes), todos los paquetes del sist. atraviesan esta tabla (tabla por defecto). Está compuesta por tres cadenas:
 - INPUT: paquetes destinados al sistema.
 - OUTPUT: paquetes generados en el propio sistema.
 - FORWARD: paquetes que lo atraviesan (encaminadores).
- Tabla NAT: Reescribe las dir. origen/destino y puertos de un paquete (traducción de direcciones), está compuesta por tres cadenas:
 - PREROUTING: paquetes de entrada justo antes de pasar por la tabla de encaminamiento local (de entrar a la máquina) usada en la traducción de la dir. destino (DNAT).
 - POSTROUTING: paquetes de salida después de la decisión de encaminamiento (antes de que salgan de la máquina) usada en la traducción de la dir. origen (SNAT).
 - OUTPUT: Paquetes de salida generados localmente (en la máquina).
- Tabla Mangle: Sirve para manipular paquetes, cambia campos u opciones de los paquetes, en el caso de TCP por ejemplo se puede modificar el campo MSS, tiene 5 cadenas: INPUT, OUTPUT, FORWARD, PREROUTING Y POSTROUTING.

Las reglas se definen en función del estado del paquete o de la conexión, normalmente se incluye la cadena a la que se añade la regla (opción -A (append)), se añade al final de la cadena, la tabla se indica con la opción -T, si no indicamos tabla se añade a la tabla por defecto (Filter), se puede filtrar por dir. origen y destino (-s y -d respectivamente), por protocolo (-p) y por muchas más opciones.

Opción/Ejemplo	Significado
-A INPUT	Añade regla a cadena de entrada
-A OUTPUT	Añade regla a cadena de salida
-A FORWARD	Añade regla a la cadena forward (sólo en caso de routers)
-s 192.168.1.1	Filtrado por dirección IP origen
-d 140.10.15.1	Filtrado por dirección IP destino
-p tcp	Filtrado de paquetes TCP
-p udp	Filtrado de paquetes UDP
-p icmp	Filtrado de paquetes ICMP
--sport 3000	Filtrado por nº de puerto origen (TCP o UDP)
--dport 80	Filtrado por nº de puerto destino (TCP o UDP)
--icmp_type 8	Filtrado por código del paquete ICMP (sólo para ICMP)
-i eth0	Filtrado por interfaz de red de entrada
-o eth1	Filtrado por interfaz de red de salida

Para definir reglas según el estado de la conexión iptables proporciona módulos, que son infraestructuras flexibles que funcionan de manera modular, se indican con la opción -m, el módulo state por ejemplo permite controlar el estado de las conexiones, para cada módulo en concreto se pueden indicar distintos estados:

- NEW: Se aplica a paquetes que inician una nueva conexión.
- ESTABLISHED: Se aplica a paquetes que pertenecen a conexiones establecidas.
- RELATED: Se aplica a paquetes que pertenecen a conexiones que están relacionadas con conexiones ya establecidas, esto se hace porque hay protocolos que usan más de una conexión (Ej: FTP).
- INVALID: Se aplica a paquetes que no pertenecen a ninguno de los estados anteriores, por ejemplo, en TCP el 1er msg con los flags SYN y ACK activos.

Opción	Significado
-m state --state NEW	Filtrado de paquetes correspondientes a conexiones nuevas (el primer paquete)
-m state --state ESTABLISHED	Filtrado de paquetes correspondientes a conexiones ya establecidas
-m state --state RELATED	Filtrado de paquetes relacionados con otras conexiones existentes
-m state --state INVALID	Filtrado de paquetes que no pertenecen a ninguno de los estados anteriores

Además de los estados también se especifica el objetivo de la regla, es decir, qué se hace con ese paquete que cumple esos requisitos (opción -j):

- DROP: Descartar el paquete
- ACCEPT: Acepta el paquete
- REJECT: Se descarta pero se notifica al origen.
- LOG: Se registra ese paquete en un fichero de log y se indica que se ha recibido.

```
# Establecer política por defecto para cadenas INPUT, OUTPUT y FORWARD
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP
# Dejar entrar o salir cualquier paquete correspondiente a
# conexiones establecidas o relacionadas
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
# Permitir conexiones entrantes SSH (tcp/22) desde pc-oficina
iptables -A INPUT -s 200.1.1.1 -p tcp --dport 22 -m state \
    --state NEW -j ACCEPT
# Permitir conexiones web salientes (tcp/80) a cualquier destino
iptables -A OUTPUT -p tcp --dport 80 -m state --state NEW -j ACCEPT
# Permitir conexiones pop3 salientes (tcp/110) con servidor de correo
iptables -A OUTPUT -d 22.1.1.1 -p tcp --dport 110 -m state \
    --state NEW -j ACCEPT
# Permitir conexiones DNS salientes (udp/53) con servidor DNS
iptables -A OUTPUT -d 22.1.1.2 -p udp --dport 53 -m state \
    --state NEW -j ACCEPT
```

Def. Traducción de direcciones (NAT): Se usa para permitir el acceso a internet o desde internet a máquinas que están configuradas con direcciones de IPv4 privadas, esto se hace traduciendo la dir. DHCP que te asigna el router a una dir. pública (normalmente la del router), de manera que el servidor ya le puede mandar al router una respuesta y éste tendrá una tabla de correspondencia donde verá qué respuesta va dirigida a qué máquina de la red local. Existen varios tipos de traducción, estática o dinámica.

Hoy en día se usa un mecanismo que traduce además de la dir. también el puerto (NAPT), normalmente se asignan N dir. privadas a una dir. pública (la del router) y se diferencian los clientes que se conectan a internet mediante puertos.

Usando el objetivo SNAT (tabla NAT) se puede cambiar la dir. origen de un paquete y el puerto (se aplica a la cadena POSTROUTING) y permite implementar NATP usando una IP pública fija.

```
iptables -t nat -A POSTROUTING -o ppp0 -j SNAT --to 175.20.12.1
```

En caso de usar una IP dinámica (como la del router), NAT dispone de un objetivo especial llamado MASQUERADE que hace uso de la IP del interfaz y que mantiene pista de las conexiones activas para aplicar el cambio de una conexión a otra.

```
iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE
```

Existe el caso contrario en el cual se quiere que desde internet se tenga acceso a las máquinas internas, en este caso se asigna una dir. pública a N dir. privadas, y para diferenciar el servidor se usa un puerto distinto (virtual servers).

Usando el objetivo DNAT (tabla NAT) se puede modificar la dir. de destino de un paquete y el puerto (se aplica a las cadenas OUTPUT y PREROUTING) y se aplica a todos los paquetes posteriores de la conexión.

```
iptables -t nat -A PREROUTING -d 175.20.12.1 -p tcp --dport 80 \
-j DNAT --to 192.168.1.1:80
iptables -t nat -A PREROUTING -d 175.20.12.1 -p tcp --dport 25 \
-j DNAT --to 192.168.1.2:25
iptables -t nat -A PREROUTING -d 175.20.12.1 -p tcp --dport 20 \
-j DNAT --to 192.168.1.3:20
iptables -t nat -A PREROUTING -d 175.20.12.1 -p tcp --dport 21 \
-j DNAT --to 192.168.1.3:21
```

Tema 4 – Servicios de red: DNS

Def. DNS: Protocolo que mantiene la correspondencia entre nombres de dominio y direcciones IP, funciona como una BD distribuida, ya que cada sitio solo guarda info. propia e intercambia y comparte la info. con otros sitios, realizando y recibiendo consultas sobre los nombres de dominio. Es un sist. muy complejo con múltiples implementaciones con diferentes funcionalidades, por ejemplo, BIND es la más usada y es open-source. También se usa para almacenar los servidores de correo de los dominios o incluso para almacenar claves criptográficas públicas.

Existe una clasificación de las zonas y dominios:

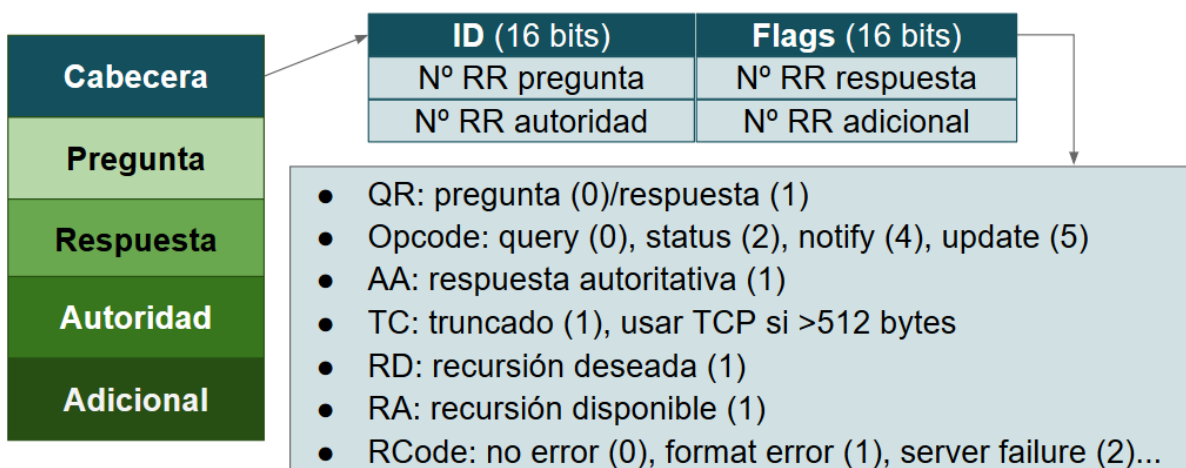
- Servidores raíz: Contienen referencias a los servidores de nombres de los dominios de 1er nivel, son 13 servidores de nombres (anycast, estas dir. pueden apuntar a más de una máquina, redirigen a la más cercana).
- Top Level Domains (TLDs): Gestionados por la ICANN, cada zona incluye los servidores de nombre autorizados y los servidores de nombre de los subdominios.
- generic (gTLD), country code (ccTLD), com, gov, net, uk, eu...etc.

Def. Nombre de dominio completamente cualificado (FQDNs): nombre completo del nodo, todos los nombres de dominio separados por el punto hasta llegar al raíz. (Ej: www.ucm.es.)

La BD se estructura en registros de recursos (RR), los servidores guardan esos RR en ficheros de texto (zone files) y DNS gestiona diferentes tipos de RR para almacenar servidores de nombres, asignaciones nombre-IP e IP-nombre, etc. Los RR son estándar e independientes de la implementación, son la info. básica que se intercambia y cachea.

DNS como protocolo de transporte

DNS usa normalmente UDP por eficiencia (puerto 53 del servidor) si se usa TCP puede ser por dos motivos, uno la transferencias de zonas (sirven para que los servidores transfieran toda la info. del fichero de una zona a otro servidor) otro en caso de hacer la pregunta o respuesta demasiado grande como para ser encapsulada en un segmento UDP (+12B).



La cabecera tiene un identificador de 16 bits que se usa para emparejar la pregunta y la respuesta, también tiene una serie de flags:

- QR: Para indicar si es una pregunta (0) o una respuesta (1).
- OpCode: Código de operación, usa 4 bits, normalmente es Query (0).
- TC: Indica que el msg ha sido truncado (puede no estar completo).
- RD: Solicita recursión.
- RA: Recursión disponible.
- RCode: No error (0).

Además de los flags hay otros 4 nº de 16 bits que indican el nº de registros en cada una de las secciones.

Existen un tipo de servidores llamados recursivos en los cuales cuando se les hace una pregunta, si no tienen la respuesta la buscan empezando por el dominio raíz, avanzando y preguntando de servidor en servidor hasta encontrarla.

Normalmente se suele cachear la info. de la resolución de direcciones, ya que no suele cambiar demasiado, así en caso de que se haga repetidamente la misma petición se pasa la misma respuesta que se tiene guardada, esto mejora notablemente la eficiencia, según la frecuencia con la que vayan a cambiar se asignan distintos niveles con distintos valores TTL.

Existen distintos tipos de servidores de nombres:

- Autoritativos: Representan oficialmente a la zona, si son primarios tienen la copia oficial de la BD en disco, si son secundarios obtienen la copia de los primarios, siempre debe haber un servidor primario y otro secundario por zona.
- De caché: Parten de una lista de servidores raíz y guardan los resultados de las búsquedas realizadas, no tienen ningún registro DNS propio, reducen la latencia de las consultas y el tráfico DNS en la red.
- No recursivos: Cuando no disponen del registro de la consulta devuelven una referencia al servidor de nombres que puede tenerlo.
- Recursivos: Resuelven cada referencia hasta devolver la respuesta al cliente, aunque no dispongan del registro, suelen usarse en la configuración de los clientes.

La BD de DNS guarda archivos de texto (zone files) mantenidos en el servidor primario de cada zona, se usan comandos que especifican cómo interpretar los registros, éstos están asociados con la zona y contienen info. como el nombre que identifica al registro, su TTL en segundos, su clase, su tipo y finalmente los datos.

Existen una serie de registros especiales en la BD de DNS:

- SOA (Start Of Authority): que marca el comienzo de definición de una zona, ésta incluye los registros dentro del espacio de nombres DNS, un servidor normalmente tiene dos zonas, una directa (nombres -> IP) y otra inversa (IP -> nombres).

```
example.com.  IN      SOA      ns.example.com. hostmaster.example.com. (
                                2003080800 ; sn = serial number
                                172800    ; ref = refresh = 2d
                                900       ; ret = update retry = 15m
                                1209600   ; ex = expiry = 2w
                                3600)     ; nx = nxdomain ttl = 1h
```

- NS (Name Server): especifica los servidores autoritativos para la zona, además se incluyen los servidores de nombres de los subdominios delegados a otras organizaciones, normalmente se añaden después del registro SOA.

```
NS ns.example.com.
NS ns1.example.com.
NS ns-ha.example.com.
NS ns.sub.example.com.
NS ns.example.com.
```

- A (Address, IPv4) y AAAA (IPv6): son la base de DNS e incluyen la traducción directa del nombre a la dir. IP.

```
ns                IN  A      63.175.177.1
                  IN  A      63.175.177.4
                  IN  AAAA    2001:501:2f::a01b
ns1.example.com.  IN  A      63.175.177.2
```

- PTR (Pointer): Contiene la traducción inversa de la dir. IP al nombre.

```
1.177            IN  PTR    ns.example.com.
```

- MX (Mail eXchanger): Usado por los sist. de correo para examinar los msgs eficientemente, permite recibir de forma centralizada el correo de una organización y realizar operaciones centralizadas (ej: filtrar spam).

```
example.com.     IN  MX      10 mail
                  IN  MX      20 mail2.example.com.
```

- CNAME: Se usa para definir un nombre canónico de un dominio, que permite definir alias para el nombre canónico, deben apuntar siempre a un dominio.

```
informatica.ucm.es. 86400  IN  CNAME  ucm.es.
ucm.es.             86400  IN  A      147.96.1.15
```

Tema 5 – Protocolo IPv6

Este protocolo surge de una serie de problemas del protocolo IPv4:

- Espacio de direcciones limitado de IPv4: a pesar de que se han ideado otras soluciones (CIDR, NAT, DHCP).
- Formato complejo de la cabecera del paquete: Tamaño variable (campo opciones) e info. de fragmentación no siempre necesaria.
- Seguridad limitada: No incluye soporte para autenticación, se han buscado soluciones (IPsec).
- Soporte limitado para prioridad de tráfico: En la mayoría de routers esta funcionalidad no está implementada.
- Multicast limitado: No se ha llegado a usar de forma completa y eficaz.

IPv6 tiene una serie de características:

- Direcciones de 128 bits: espacio de direcciones mayor que en IPv4.
- Formato de cabecera fijo (más simple): Mayor velocidad de procesamiento de los routers y mejora en el rendimiento de los protocolos de routing.
- Posibilidad de autoconfiguración de direcciones: Ya no es necesario DHCP.
- Mejor soporte para opciones adicionales: Estas opciones no se codifican en la cabecera sino en el cuerpo del paquete IP (payload), no hay limitaciones de espacio
- Opciones de seguridad tanto para autenticación como para cifrado: IPsec.
- Soporte para tráfico en tiempo real.
- Encaminamiento jerárquico basado en prefijos.
- Mecanismos de transición desde IPv4: A base de túneles.

Característica	IPv4	IPv6
Longitud de direcciones	32 bits	128 bits
Clases de direcciones	Clase A, Clase B, Clase C o CIDR	Direcciones sin clase (Classless)
Tipo de direcciones	Unicast, Multicast, Broadcast	Unicast, Multicast, Anycast
Configuración de dirección	Estática (a través de ficheros de configuración) o por DHCP	Estática (a través de ficheros de configuración), <u>autoconfiguración (plug and play)</u> o por DHCP
Formato cabecera	Complejo. Longitud variable	Simple. Longitud fija
Calidad de servicio	Sí, aunque no soportado totalmente por routers	Sí
Soporte tráfico en tiempo real	No	Sí
Seguridad	No (extensión IPsec)	Sí

Direccionamiento

En IPv6 existen distintos tipos de direcciones:

- Unicast: Identifican a un único interfaz en la red, un paquete dirigido a una dir. unicast se entregará únicamente al interfaz identificado con dicha dir. IP.
- Multicast: Identifica a un grupo de interfaces (asignadas a más de un interfaz), un paquete dirigido a una dir. multicast se entrega a todos los interfaces identificados con esa dirección, no existe dir. de broadcast.
- Anycast: Parecidas a las multicast, pero cuando se envía un paquete a una de estas dir. no hay que entregarlo a todos sino solamente a uno (al más cercano).

n	128 - n
Prefijo de la subred	Identificador del interfaz

Las dir. tienen una long. de 128b (16B), se usa una notación hexadecimal, en la que la dir. se divide en 8 grupos de 16 bits, cada grupo se escribe en hexadecimal con 4 dígitos, separados por “:”. Ej: FE80:0000:0000:0000:0008:0800:200C:741A

Normalmente se suele usar una notación abreviada, en cada grupo los 0's a la izq. se pueden omitir, las cadenas de 0's seguidos se pueden comprimir con el símbolo “::” y éste solo puede aparecer una vez, la dir. anterior abreviada: FE80::8:800:200C:741A

Se utiliza una notación CIDR sin clases para soportar el direccionamiento jerárquico, por lo que las direcciones se dividen en prefijo y sufijo, el tamaño del prefijo se denota en CIDR, ej: FE80::8:800:200C:741A/64 (64b de prefijo de red y 64b de id. de interfaz).

Def. Ámbito (scope): Determina la región de la topología de red en la que la dir. es válida, puede ser de tres tipos distintos:

- Enlace local (link-local): Dir. válida dentro del enlace en el que está conectado la interfaz de red (ej: LAN).
- Sitio local (site-local): Dir. válida dentro de un sitio, que puede estar formado por una o varias redes interconectadas mediante routers (ej: campus universitario).
- Global: Dir. válida en todo internet.

Def. Zona (scope zone): Región conexas de la red de un ámbito determinado, la unicidad de las dir. solo se garantiza dentro de su zona.

Casi todas las dir. son de enlace local (descubrimiento de vecinos, parecido a ARP) o globales, en caso de ambigüedad se usan índices de zona, indicados <dir>%<idzona>.

IPv6 permite una jerarquía flexible, es decir, acomoda diferentes tipos de direcciones, cada una de ellas comienza con un prefijo de formato de long. variable.

Tipo de dirección	FP (binario)	FP (hexadecimal)
<i>Reserved Address</i>	0000 0000	0000::/8
<i>Global Unicast Address</i>	001	2000::/3
<i>Link-Local Unicast Address</i>	1111 1110 10	FE80::/10
<i>Site-Local Unicast Address</i> (en desuso)	1111 1110 11	FEC0::/10
<i>Unique Local Address</i> (ULA)	1111 110	FC00::/7
<i>Multicast Address</i>	1111 1111	FF00::/8

Las dir. IPv6 de enlace local son dir. unicast privadas que se asignan a un enlace (link), una zona de enlace local consiste en un único enlace con todos los interfaces conectados a él, nunca se encaminan fuera de la zona de ámbito del enlace, es un espacio de dir. plano y su principal uso es la autoconfiguración y el descubrimiento de vecinos, su prefijo de formato siempre es “fe80::”.

Las dir. ULA (Unique Local Adresses) son dir. unicast privadas que pueden usarse en intranets jerárquicas, nunca se encaminan fuera del sitio (aunque su ámbito es global) y sustituyen a las antiguas dir. de sitio local, su formato es:

- Prefijo de formato es “fc00::”.
- El bit 8 a 1 indica que el prefijo se asigna localmente (ID global).
- Identificador global (40bits): debe ser generado aleatoriamente (evitar colisiones)
- Identificador de subred (16bits).
- Identificador de interfaz (64bits).

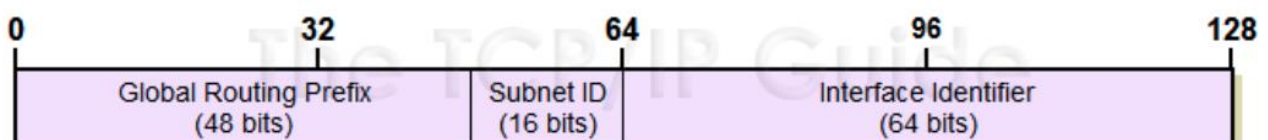
Ejemplo: fd12:A128:e8e1:1:FEDC:BA98:7865:4321/64

8 bits	40 bits	16 bits	64 bits
1111 1101	ID global (aleatorio)	ID subred	ID interfaz

Las dir. IPv6 de enlace global permiten la autoconfiguración, su formato es:

- Prefijo global de encaminamiento (48 bits).
- Identificador de subred.
- Identificador de interfaz.

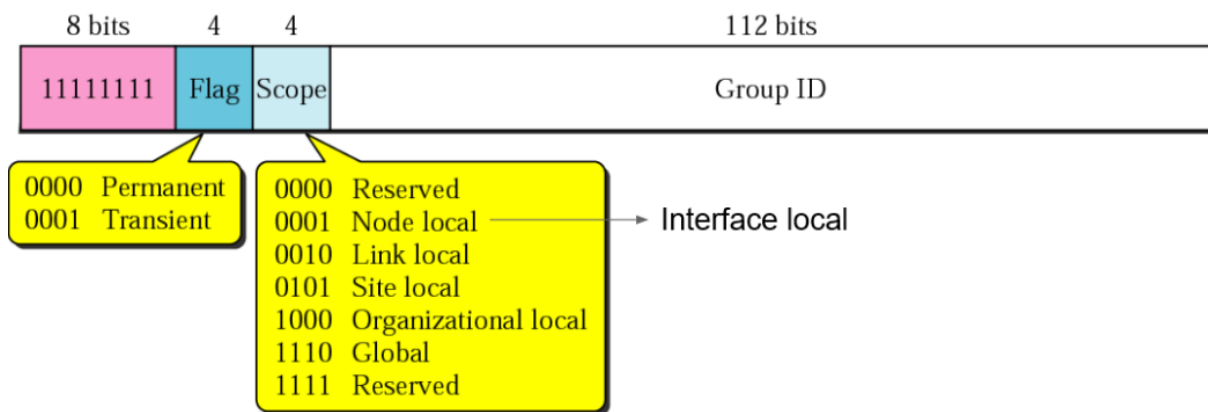
Ejemplo: 2004:A128::32:FEDC:BA98:7865:4321/64



Los 64 bits menos significativos determinan el identificador del interfaz, normalmente se usa el 64-bit unique identifier que permite relacionar la dir. IP de red con la dir. MAC de enlace (dividiendo la MAC por la mitad y colocando en medio el valor FF FE, e invirtiendo el bit 7 ya que si está a 0 indica que la dir. MAC es globalmente única) esto puede suponer un problema de privacidad ya que se podrían rastrear desde donde se conectan los clientes, para evitarlo existe una alternativa que es usar las extensiones de privacidad de IPv6 que generan un identificador de interfaz pseudoaleatorio y temporal.

Las dir. IPv6 multicast definen un grupo de interfaces en un ámbito determinado, su formato es:

- Prefijo de formato: “FF::/8” (8 bits), FF <- Multicast.
- Flags: Indican si es una dir. permanente (0000) o temporal (0001) para una comunicación (4 bits).
- Ámbito: Puede ser interfaz local (01), enlace local (02), enlace global (0E), etc. (4 bits)
- Identificador de grupo (112 bits), Grupo 2 <- todos los routers del sitio local.



- Direcciones para los computadores

Dirección	Ámbito	Significado
FF01::1	Interface local	Un datagrama dirigido a esta dirección se envía a una interfaz del host
FF02::1	Link local	Un datagrama dirigido a esta dirección se envía a todos los interfaces del enlace local, pero nunca se encamina

- Direcciones para los encaminadores

Dirección	Ámbito	Significado
FF05::2	Site local	Un datagrama dirigido a esta dirección se envía a todos los routers del sitio local, por tanto se reexpide a todas las subredes a través de los routers internos
FF02::9	Link local	Un datagrama dirigido a esta dirección se envía a todos los routers del enlace local que realizan encaminamiento RIP

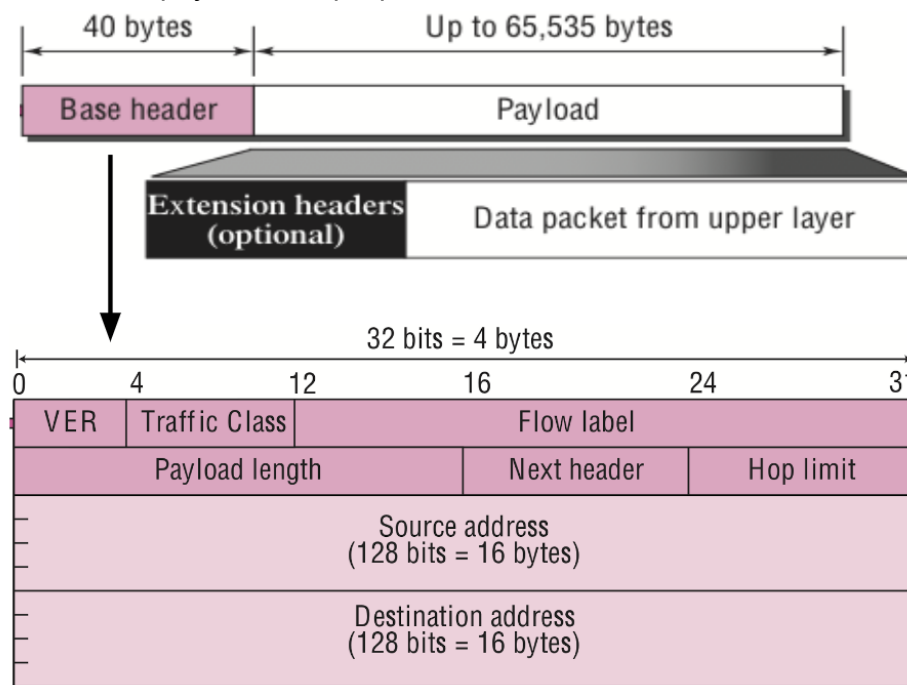
Def. Dir. multicast de nodo solicitado: Se usan en el protocolo de descubrimiento de vecinos (que implementa la funcionalidad que antes nos daba ARP) con el cual averiguamos la dirección MAC asociada a una dir. IPv6, al no haber brdcst en IPv6 se usa esta dirección como dest, se calcula a partir de la dir. unicast del nodo, añadiendo los 24b menos significativos de la dir. al rango FF02::1:FF, ejemplo:

Dir. unicast: 2037::01:800:200E:8C6C

Dir. multicast de nodo solicitado: FF02::1:FF0E:8C6C

Otras posibles dir. en IPv6 son la dir. sin especificar (::) y la dir. de loopback (::1).

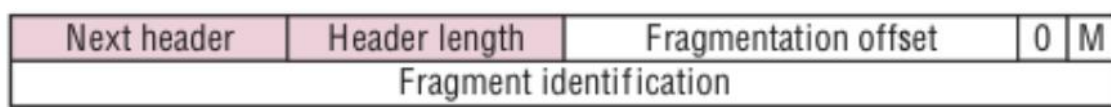
En IPv6 el datagrama se ha simplificado, eliminando campos relacionados con la fragmentación y añadiendo algunos otros, hay una cabecera fija de 40B y si hace falta alguna extensión se codifica en cabeceras de extensión opcionales que se añaden al payload del paquete.



- Traffic Class: Distingue distintos requisitos de entrega del datagrama, tiene dos subcampos, DSCP (prioriza el tráfico e intenta reducir la latencia) y ECN (permite a los nodos notificar situaciones de congestión en la red sin descartar paquetes).
- Flow Label: Etiqueta al paquete como perteneciente a un flujo, por lo que mejora el procesamiento hecho por los nodos de red, así solo tienen que consultar la tabla de rutas con el 1er paquete del flujo.
- Payload length: Long. de los datos sin contar la cabecera.
- Hop Limit: Similar al campo TTL, pero medido en segundos.
- Source/Dest Address: Dir. origen y destino.
- Next Header: Siguiendo cabecera en el datagrama (extensión o protocolo).

Fragmentacion

En IPv6 se realiza en origen (los routers nunca fragmentan), para evitar estos problemas se usa el alg. Path MTU Discovery según el cual la MTU del camino es la mínima MTU de un enlace entre origen y destino, el origen va adaptando el tamaño de la MTU según se vaya encontrando con MTU's de distinto tamaño, ya que si lo encuentra el router manda un msg ICMP informando del problema.



- Header length: Reservado inicializado a 0's (8 bits).
- Offset: Desplazamiento respecto al inicio de la parte fragmentable del datagrama original en unidades de 8 bytes (13 bits).
- M: Similar al MF de IPv4, indica si hay más fragmentos o no.
- Identification: Permite identificar a los fragmentos del mismo programa.

ICMPv6

ICMPv6 asume el papel de otros protocolos auxiliares como IGMP o ARP, es el protocolo de msgs de control de internet para IPv6, proporcionando mensajes de error, de info. y además msgs para el descubrimiento de vecinos y para la pertenencia y gestión de grupos multicast (Multicast Listener Discovery, MLD), todos los msgs tienen un formato común, un tipo (8b), un código (8b), checksum (16b), etc.

Los msgs de error usan los tipos del 0 al 127 e incluyen errores relativos a destino inalcanzable, datagrama demasiado grande, tiempo excedido, etc, este tipo de datagramas no se pueden encaminar por lo que se descartan, los msgs de info. usan los tipos del 128 al 255, ej: echo request y echo reply.

ICMPv6 es un protocolo multifunción que también permite realizar oper. de configuración de hosts y routers dentro de una red:

- Descubrimiento de vecinos: Resolución de dir. (ARP), detección de dir. duplicadas y comprobación de si un vecino sigue siendo alcanzable (Neighbour solicitation y Neighbour advertisement).
- Descubrimiento de router: Permite descubrir encaminadores y además estos pueden anunciar prefijos u otra info. de la red (Router solicitation y Router advertisement).
- Redirección: Notificar a un host la ruta más adecuada para alcanzar un determinado destino (ICMPv6 Redirect).

Neighbour solicitation: Este msg se puede generar por diversos motivos:

- Para averiguar la MAC asociada a una dir. IP usando como dest. la dir. multicast de nodo solicitado.
- Para determinar si un nodo vecino sigue siendo alcanzable.
- Para detectar si la dir. IP está duplicada (proceso autoconfiguración DHCP).

Neighbour advertisement: Este msg se puede generar por diversos motivos:

- Responder a un msg de neighbour solicitation usando la dir. unicast del destinatario como vecino.
- Anunciar un cambio en la dir. de enlace del interfaz usando como destino FF02::1.

Router solicitation: Se genera cuando un interfaz se activa para detectar los nodos y realizar la autoconfiguración del interfaz, con la dir. multicast FF02::2 como destino.

Router advertisement: Lo envían los nodos para anunciar su presencia en la red periódicamente con la dirección multicast FF02::1 como destino o como respuesta a un msg de router solicitation de un host.

Tema 6 – Encaminamiento en internet

Def. Encaminamiento: Mecanismo de búsqueda del camino más óptimo en una red, desde el origen al destino, a través de routers intermedios.

El camino más optimo suele ser el más corto o el de menor coste, para establecerlo se usan una serie de métricas:

- Nº de saltos: Nº de encaminadores intermedios que se atraviesan.
- Distancia geográfica: Distancia en Km que tiene que recorrer el paquete.
- Retardo promedio: Retardo de las líneas, es proporcional a la distancia.
- Ancho de banda: Velocidad de transmisión de las líneas por las que va el paquete.
- Nivel de tráfico: Nivel de uso de las líneas, usar las de menor nivel de saturación.

Def. Retransmisión de paquetes: Mecanismo según el cual cuando un encaminador recibe un paquete lo retransmite (forward) por el enlace adecuado para alcanzar el destino, la elección de este enlace se hace en base a las tablas de encaminamiento, que usan el campo dir. destino del paquete IP y buscan en la tabla un destino que concuerde con la dir. aplicando la máscara de red y calculando la dir. de red aplicada al destino, si coincide con alguna entrada lo envía a esa dir., si no coincide se usa una entrada por defecto que siempre coincide con todos.

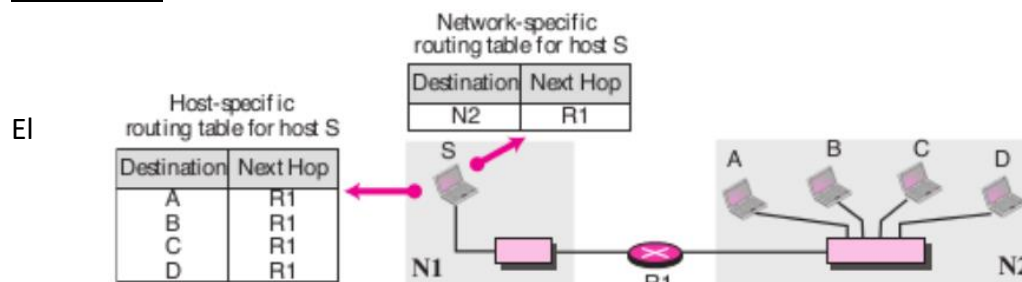
Cada datagrama IP se etiqueta y conmuta según su etiqueta (orientado a conexión), en IPv6 el equivalente es el campo Flow Label, esto reduce la complejidad de la tabla de encaminamiento, siempre se usa el mismo circuito (entrega en orden).

A la hora de decidir cuál es el destino del siguiente salto se usa el principio de optimalidad de Bellman, según el cual, dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima, es decir, todos los pasos que llevan a una solución óptima han de ser óptimos, por lo tanto, los encaminadores solo necesitan saber la identidad del siguiente encaminador inmediato, y no de toda la ruta.

Una tabla de encaminamiento tiene info. sobre:

- Destino.
- Máscara o prefijo de red (CIDR).
- Siguiente salto.
- Coste asociado al camino.

Las entradas destino corresponden con el host específico, con la red o con el destino por defecto.

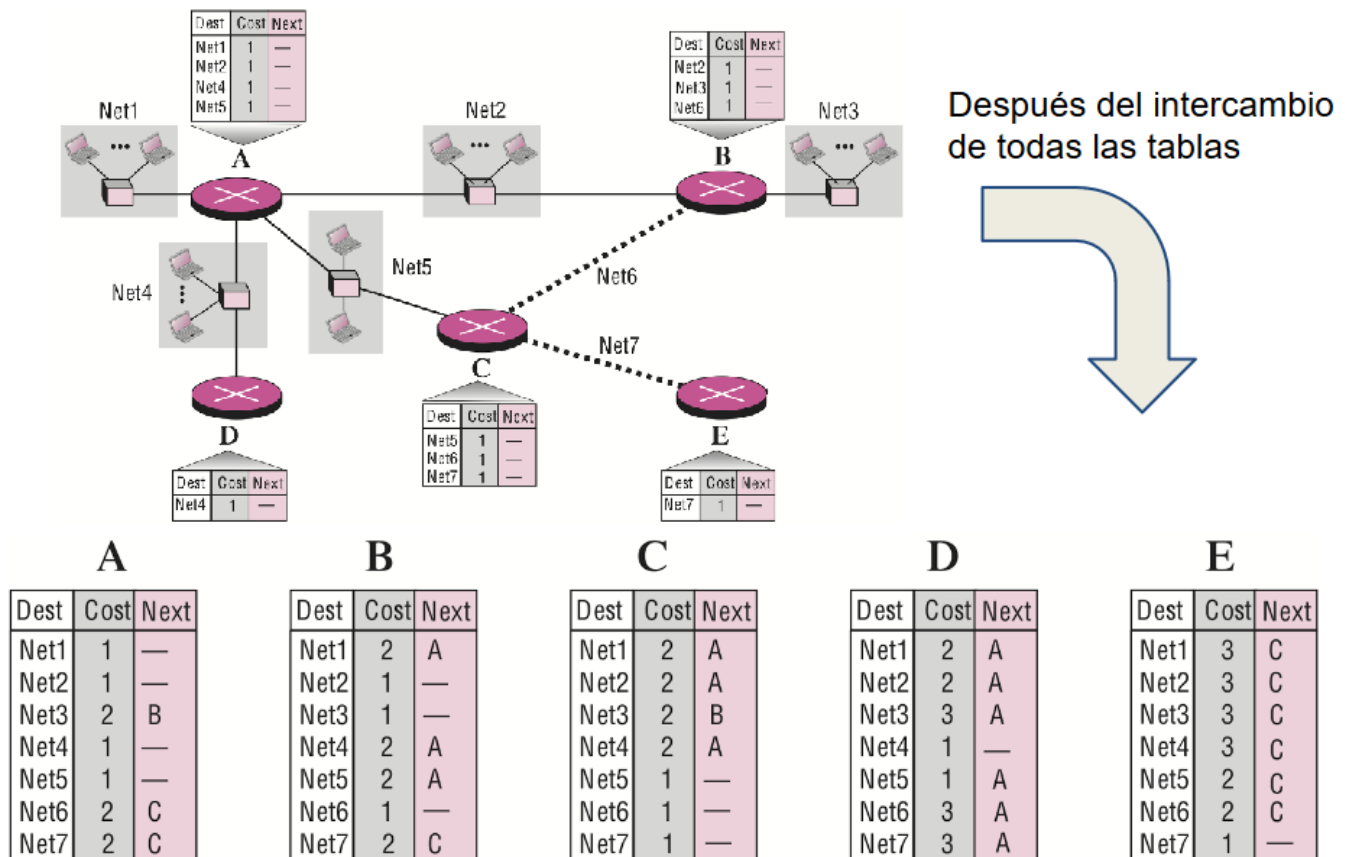


encaminamiento en internet depende de controlar el tamaño de las tablas de rutas de los encaminadores, el encaminamiento con clase no es viable debido al gran nº de redes de internet, por lo que se basa en CIDR y en encaminamiento jerárquico.

Existen distintos tipos de técnicas de encaminamiento:

- Encaminamiento local: No tiene en cuenta la topología de la red, únicamente usa info. local.
- Encaminamiento estático: Las decisiones de encaminamiento consideran la topología de la red, las tablas de encaminamiento se construyen manualmente (ip route o route) y no se adaptan a cambios de la red, válido en redes muy simples.
- Encaminamiento dinámico: Las tablas de encaminamiento se construyen de forma automática mediante el intercambio periódico de info. entre los encaminadores, esto permite adaptar automáticamente el encaminamiento a los cambios en la topología de la red, las técnicas más comunes son RIP y OSPF.

Cada encaminador mantiene una tabla de encaminamiento con una entrada por cada posible destino (host o red), cada entrada contiene el destino, el tamaño de la máscara de red, el siguiente nodo para alcanzar dicho destino y la distancia hasta ese destino, para construir la tabla de encaminamiento los nodos intercambian periódicamente vectores de distancia con sus vecinos, actualizando su tabla de rutas periódicamente, este proceso iterativo de intercambio de tablas converge idealmente a los caminos óptimos (alg. de Bellman-Ford), el coste es igual al nº de saltos de la red, el protocolo RIP (Routing Info. Protocol) es un ejemplo de protocolo que hace uso de estas técnicas.



Existe un problema de convergencia con los vectores de distancia, ya que si uno de los enlaces se rompe por cualquier motivo, en su tabla de rutas pondría que está a una distancia infinita del destino, esto provoca múltiples problemas ya que los cambios en las tablas de rutas de todos los encaminadores tardan en propagarse.

Existen diversas soluciones para este problema:

- Establecer el infinito a un número pequeño: En RIP por ejemplo se pone a 16 saltos, por lo que las rutas tienen un límite de 15 saltos.
- Horizonte dividido: Los destinos aprendidos a través de un determinado enlace nunca se difunden a través de dicho enlace.
- Horizonte dividido con ruta inversa envenenada: Los destinos aprendidos por un enlace si se anuncian a través de ese enlace, pero con una distancia infinita.

- Actualizaciones forzadas: Cuando un encaminador detecta un cambio en su tabla de rutas lo difunde inmediatamente, así los cambios se propagan rápidamente.

Existe otro problema de convergencia, en redes con bucles puede que el algoritmo no converja, en este caso las técnicas de horizonte dividido no son una solución.

Existen protocolos basados en estado del enlace en los cuales cada encaminador mantiene un BD con la info. sobre la topología exacta de la red, para construirla cada nodo identifica a sus vecinos y la distancia a la que están, una vez conoce esta info. la anuncia a todos los nodos de la red (inundación) y con ella (grafo de la red) cada uno construye un mapa de rutas (árbol) desde su posición en la red, OSPF (Open Shortest Path First) por ejemplo usa estas técnicas.

Internet está organizada en sist. autónomos (AS) que son colecciones de redes y routers gestionados y administrados por una misma autoridad, cada uno se identifica mediante un nº AS, y existen tanto encaminadores internos, que interconectan redes dentro del propio AS, como externos, que interconectan varios ASs, cada uno con sus características propias.

Los encaminadores externos usan protocolos externos (EGP) como EGP o BGP, mientras que los encaminadores internos usan una serie de protocolos internos (IGP) para su encaminamiento, como RIP, OSPF o IGRP, y no usan ni protocolos de vectores de distancia ni basados en el estado del enlace, por los problemas que ocasionan o por la info. que requieren, usan un protocolo basado en vectores de rutas, que intenta resolver los problemas anteriores, está basado en los vectores de distancia y cada encaminador obtiene a partir de la info. sobre los destinos alcanzables en el AS y mediante un proceso de intercambio, la lista de destinos alcanzables y la ruta completa al destino (AS que atravesar), además este protocolo hace uso de CIDR para agregar dir. de red en las tablas de rutas, tiene detección de bucles sencilla y permite implementar políticas comprobando si un determinado AS es parte de la ruta.

Protocolo RIP

Es un protocolo de encaminamiento interior (IGP) basado en vectores de distancia, estos incluyen la lista de destinos alcanzables por cada encaminador junto con la distancia (nº de saltos) de esos destinos, los msgs se encapsulan en datagramas UDP dirigidos al puerto 520, el límite se establece en 16 saltos y puede usar distintas soluciones para el problema de la convergencia.

El msg puede ser REQUEST (solicitud, se envía cuando se conecta a la red (0.0.0.0) o cuando caduca una entrada en la tabla) o RESPONSE (respuesta, que se difunden periódicamente (broadcast) con los vectores de distancia o en respuesta a una solicitud, también usando actualización forzada cuando cambia la distancia a la red).

RIP usa además una serie de temporizadores:

- Temporizador periódico: Intervalo de envío de msgs RESPONSE para anunciar los vectores de distancia, RIP establece un valor de 30s para este temporizador.
- Temporizador de expiración: Controla el periodo de validez de una entrada en la tabla de encaminamiento, si no se recibe actualización de la entrada durante 180s la entrada deja de considerarse valida.
- Temporizador de “recolección de basura”: Cuando una entrada de la tabla de rutas expira el encaminador no la elimina inmediatamente, sino que se sigue anunciando con métrica 16 (dest. inalcanzable) durante un periodo adicional de 120s.

RIP versión 1 tiene algunas limitaciones, como la gran cantidad de tráfico broadcast, no tener soporte para CIDR, no autenticar a info. de encaminamiento (seguridad), no admitir caminos alternativos, el tiempo de propagación de cambios, etc. En su segunda versión corrige algunas de estas limitaciones añadiendo soporte para máscaras de red, multicast y autenticación.

Existe una adaptación del protocolo RIP-2 para IPv6 llamada RIPng que se diferencia con RIP-2 en que los msgs se encapsulan en datagramas UDP dirigidos al puerto 521 y se difunden a la dirección IPv6 multicast FF02::9, los vectores de distancia de los msgs RESPONSE anuncian prefijos de red IPv6, no usa info. de autenticación ya que en su lugar usa mecanismos de cifrado y autenticación disponibles en IPv6 la info. de ruta contenida en un vector de distancia no incluye el campo Next Hop sino que en su lugar se incluye una entrada específica Next Hop que afecta a las entradas siguientes.

Protocolo OSPF

Es un protocolo de encaminamiento interior (IGP) basado en el estado de los enlaces, se desarrolló como alternativa a RIP para aliviar sus limitaciones, teniendo soporte para máscaras de long. variable (VLSM) y CIDR, para autenticación, particionado lógico de la red, convergencia más rápida, etc.

Usa un protocolo propio de encapsulado y direcciones multicast, en este protocolo los encaminadores intercambian info. extra además de los destinos y la distancia, de manera que al final conocen la topología de red completa y en función de eso calculan rutas óptimas.

OSPF hace uso de áreas que son mecanismos para hacer un particionado lógico de la red y reducir el intercambio de info., existe un área especial llamada backbone que siempre está conectada a otras áreas.

Los encaminadores tienen un identificador único (RID) de 32 bits e intercambian info. dependiendo del tipo de encaminador:

- Intra-Area Routers (IA): Localizado en un área, mantiene solo info. de la topología de su área.
- Area Border Routers (ABR): Conectado a dos o más áreas, mantiene una BD por cada una de las áreas a las que está conectado.
- AS Boundary Routers (ASBR): Situado en la frontera del AS intercambia rutas entre la red OSPF y otros sist., son los responsables de transmitir rutas externas a la red OSPF y pueden inyectar en OSPF rutas aprendidas mediante otros protocolos.

Las redes definen la frecuencia y el tipo de comunicaciones entre los encaminadores, OSPF define distintas redes, punto-a-punto, multi-acceso con brdcst, multipunto, etc.

En OSPF decimos que dos encaminadores son vecinos si comparten un enlace común, pertenecen a la misma área y usan el mismo mecanismo de autenticación, dos encaminadores son adyacentes si son vecinos y además intercambian info. de estado de los enlaces entre ellos, esto permite limitar la info. intercambiada entre encaminadores, las relaciones de adyacencia se desarrollan según el tipo de red.

El proceso de distribución de la info. de enlaces es una optimización de la estrategia de inundación, en los enlaces multi-acceso el encaminador designado (DR) y el de respaldo (BDR) son los de mayor prioridad, los msgs del DR no son inmediatos, para solapar el envío de múltiples actualizaciones, estos msgs se confirman (fiabilidad).

OSPF hace uso del protocolo OSPF Hello para el descubrimiento y el mantenimiento de los vecinos, cada nodo envía periódicamente msgs Hello a través de todos sus enlaces, en este se incluye el RID de todos los msgs Hello que se han recibido a través de ese enlace, cuando un nodo recibe un msg Hello de otro con su RID se hacen vecinos.

Se eligen los encaminadores designado y de respaldo (DR y BDR) según la info. de prioridad contenida en los paquetes Hello, en caso de empate se elige el de mayor RID.

Para el establecimiento de las adyacencias se sigue un proceso de dos fases:

- Intercambio de BD: Después de establecer la vecindad se intercambian las BD con el estado de los enlaces, de forma que los nodos van detectando info. no contenida en su BD u obsoleta, el intercambio sigue un patrón maestro-esclavo, cada msg es confirmado con otro msg que incluye la info. del esclavo.
- Generación de la BD: Se solicita al encaminador adyacente una copia de los msgs obsoletos o no presentes, siguiendo un mecanismo similar al de inundación usado para comunicar los cambios en la red.

Una vez el encaminador dispone de la info. del estado de los enlaces construye su árbol de rutas, que incluye tanto encaminadores (RID) como redes (IP) y el coste asociado, según este árbol se construye una tabla de encaminamiento que contiene el destino (red o host), el siguiente salto y el coste total asociado.

Protocolo BGP

BGP es un protocolo de encaminamiento exterior basado en vectores de rutas, la función principal de estos sist. es intercambiar info. sobre las redes alcanzables con otros sist. BGP, esta info. incluye una lista de ASs atravesados en la ruta y basta para construir un grafo de conectividad de ASs libre de bucles para las redes alcanzables, cada AS aplica políticas para aceptar y publicar las rutas recibidas.

Existen distintos tipos de AS:

- Stub: Conectado únicamente a otro AS, es destino u origen para el tráfico (ISP).
- Multihomed: Conectado a varios AS, es destino y origen del tráfico de red.
- Tránsito: Es como un AS multihomed pero permite el tráfico de tránsito.

La comunicación entre encaminadores en BGP se realiza mediante TCP, puerto 179, estos intercambian a tabla de rutas cuando establecen la conexión inicial y periódicamente se envían actualizaciones incrementales sobre la tabla inicial.

Existen distintos tipos de msgs BGP:

- OPEN: Establecimiento de la sesión BGP, contiene el identificador del AS y el encaminador, así como parámetros de configuración.
- UPDATE: Actualización incremental de la info. de encaminamiento, cada msg puede incluir una red alcanzable en CIDR con sus atributos (ruta, redes retiradas...).
- NOTIFICATION: Se envía a los vecinos cuando se detecta un error, implica un cierre de sesión y las rutas asociadas se invalidan.
- KEEPALIVE: Se usa para asegurar que la sesión permanezca activa en respuesta a un msg open y periódicamente para informar de la presencia del encaminador.

Los msgs UPDATE incluyen las redes alcanzables y los atributos de ruta y permiten evaluar caminos alternativos al mismo destino. Existen distintos tipos de atributos:

- Bien conocidos: Deben ser admitidos por todas las implementaciones BGP, pueden ser obligatorios (se deben incluir en cada actualización) o discrecionales.
- Opcionales: Son específicos de cada implementación, pueden ser transitivos (se debe incluir en las actualizaciones) o no.

Ejemplos de atributos conocidos y obligatorios:

- ORIGIN: Origen de la info. de ruta, no debe modificarse por otro nodo BGP.
- AS_PATH: La ruta como cito o secuencia de ACs.
- NEXT_HOP: Dirección IP del siguiente salto para alcanzar el destino.

Sistemas Operativos

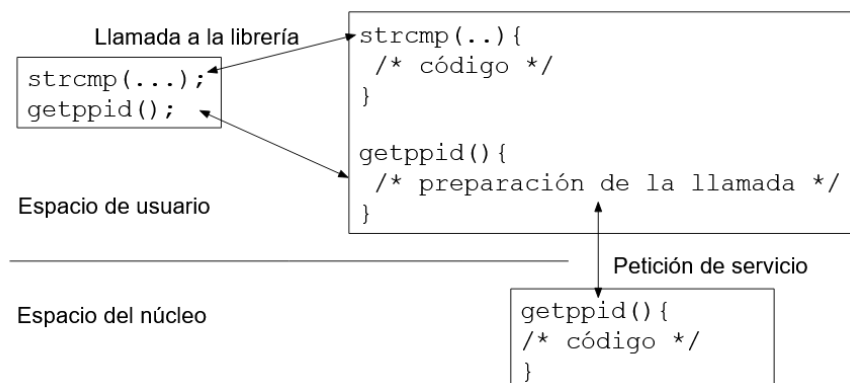
Tema 1 – Introducción a la programación de sistemas

Existen distintos tipos de estándares de programación:

- ANSI-C/ ISO-C: Estándar más general, adoptado por ANSI e ISO.
- BSD: Sus contribuciones más importantes son los enlaces simbólicos, sockets, la función *select*, etc.
- SVID: Descr. formal de las distribuciones comerciales de UNIX por AT&T, su principal contribución son los mecanismos IPC.
- POSIX: Describe llamadas al sistema y librerías de C, especifica la semántica detallada de la Shell y un cjto mínimo de comandos, además de interfaces detallados para varios lenguajes de programación.
- GNU: SO de tipo UNIX de SW libre con licencia GNU GPL, Linux utiliza este tipo de SW junto con un kernel propio de Linux.

Diferencias entre llamadas al sist. y a librerías:

Desde el punto de vista del programador no hay diferencia, pero una llamada al sist. es una función de la librería C que solicita un servicio del sist., y cuya solicitud se resuelve en el núcleo del SO (trap), mientras que una llamada a una librería estándar no interacciona de forma directa con el sist., a pesar de que debe usar llamadas al sist.



Las funciones de sistema y librería están documentadas en las secciones 2 y 3 del manual respectivamente, el formato general es *man [secciónManual] comando*.

	Llamadas al Sistema	Llamadas a Librerías
Sección de manual	2	3
Área de ejecución	Usuario/Kernel	Usuario
Espacio de parámetros	No se reserva	Dinámico/Estático
Código de error	-1 + errno	NULL + no errno

Argumentos de un programa:

El formato de la cabecera principal de un programa puede ser:

```
int main(int argc, char **argv); o int main(int argc, char *argv[]);
```

POSIX se compone de una serie de reglas a seguir:

- Los argumentos se consideran opciones si empiezan con un guion.
- Los nombres de las opciones son un único carácter alfanumérico.
- Se pueden indicar varias opciones tras un guion en un solo elemento si las opciones no toman argumentos, Ej: *-abc* es igual que *-a -b -c*.
- Las opciones que requieren un argumento pueden tener o no un espacio entre la opción y el argumento, Ej: *-o foo* y *-ofoo* es equivalente.
- El argumento *-* termina las opciones, lo que sigue no se trata como opciones, incluso si empiezan por guion.
- Las opciones largas consisten en dos guiones seguidos de un nombre compuesto por caracteres alfanuméricos y guiones, Ej: *--name=value*.

Para procesar los argumentos de un programa se usa la función `int getopt(int argc, const char *argv[], const char *options)`; la cual cuenta con las variables `extern char *optarg`; `extern int optind, opterr, optopt`; y funciona de manera que si se encuentra alguna opción entonces se devuelve, tras tratar todas, se devuelve -1. Si alguna opción tiene un argumento, entonces se establece en el puntero `optarg`. Si se encuentra una opción no valida o a la cual le falta un argumento se devuelve el carácter '?' y establece `optopt` a la opción incorrecta. Si `opterr` no es 0 se muestra un msg de error en la salida de error estándar.

API del sistema:

Def: *API (Application Programming Interface)*: Cjto de funciones y rutinas agrupadas con un propósito común.

A la hora de usar una API conviene tener en cuenta una serie de consideraciones generales, como qué fichero de cabecera necesito (`#include`), qué tipos de datos devuelve la función, cuáles son sus argumentos, qué significado tienen si se pasan por valor o por referencia (entrada/salida), qué significado tiene el valor de retorno, cómo se ha de gestionar la mem. de las variables, etc.

Para comprobar la traza de las llamadas al sist. realizadas por un programa se ejecuta el comando *strace [opciones] comando [argumentos]*, esto intercepta las llamadas al sist. que realiza el comando y las señales que recibe, permitiendo analizar el comportamiento de programas de que no disponemos de su código fuente, en cada línea se muestra la llamada al sist. realizada, los argumentos y el valor de retorno. Existen diversas opciones que muestran info. adicional, *-c*, *-f*, *-T*, *-e*.

Gestión de errores:

Para la gestión de errores se suele usar la función `void perror(const char *s)`; la cual imprime por pantalla un msg de error perteneciente a la última llamada al sist. realizada, su formato de salida es *cadena : msg de error \n*, en la cadena debe incluirse el nombre de la función que produjo el error, el código de error se obtiene de la variable `int errno`, que se fija cuando se produce un error.

La función `char *strerror(int errnum)` devuelve una cadena que describe el nº de error, por convenio las llamadas al sist. devuelven -1 cuando se ha producido un error, también lo hacen algunas llamadas de librería.

Información del sistema:

Para obtener info. sobre el kernel actual se usa la función `int uname(struct utsname *buffer)`; la cual devuelve la info. en la estructura *buffer* de la forma:

```
struct utsname {  
    char sysname[];  
    char nodename[];  
    char release[];  
    char version[];  
    char machine[];  
}
```

Si se recibe el código de error *EFAULT* significa que el buffer no es válido.

Información del SO:

Para obtener info. acerca del SO se usa la función `long sysconf(int name)`; donde el argumento name puede ser:

- `_SC_ARG_MAX`: Long. max. de los argumentos de las funciones `exec()`.
- `_SC_CLK_TCK`: Nº de ticks de reloj por segundo (Hz).
- `_SC_OPEN_MAX`: Nº máx. de ficheros que puede abrir el proceso.
- `_SC_PAGESIZE`: Tamaño de página en bytes.
- `_SC_CHILD_MAX`: El nº máximo de procesos simultáneos por usuario.

La función devuelve el valor del parámetro en cuestión o -1 en caso de error, en este caso no se instancia la variable `errno`.

Información del sist. de ficheros:

En caso de querer obtener info. sobre el sist. de ficheros se usa `long pathconf(char *path, int name)`; y `long fpathconf(int filedes, int name)`, donde el argumento name puede ser:

- `_PC_LINK_MAX`: Nº máx. de enlaces al archivo/directorio.

- `_PC_NAME_MAX`: Long. máx. del nombre de archivo en el directorio indicado por *path*.
- `_PC_PATH_MAX`: Long. máx. de la ruta relativa.
- `_PC_CHOWN_RESTRICTED`: Devuelve un valor no nulo si no puede efectuarse un cambio de permisos sobre un archivo.
- `_PC_PIPE_BUF`: Tamaño de la tubería asociada a *path* o *files*.

La función devuelve el límite asociado con el parámetro, -1 en caso de error o de que no exista, solo instancia `errno` en el primer caso.

Información del usuario:

Los procesos disponen de un identificador de usuario (UID) y de grupo (GID), que corresponden a los identificadores del usuario que posee el proceso, se denominan UID y GID reales `uid_t` `getuid(void)`; y `gid_t` `getgid(void)`;

Además los procesos disponen de un identificador de usuario efectivo (EUID) y grupo efectivo (EGID), que son los que se comprueban para conceder permisos, generalmente los identificadores UID y EUID coinciden, sin embargo, cuando se ejecuta un programa con el bit *setuid* activo, el proceso hereda los privilegios del propietario y grupo del archivo de programa.

Si se desea obtener info. de usuario accediendo a base de datos de contraseñas se hace uso de `struct passwd *getpwnam(const char *name)`; o `struct passwd *getpwuid(uid_t uid)`;

```
struct passwd {
    char *pw_name; // Nombre de usuario
    char *pw_passwd; // Contraseña
    uid_t pw_uid; // Id. de usuario
    gid_t pw_gid; // Id. de grupo
    char *pw_gecos; // Nombre real de usuario
    char *pw_dir; // Directorio "home"
    char *pw_shell; // Shell
}
```

Devuelve NULL si no encuentra al usuario o si se produce algún error.

Información de la hora del sist.:

La función `time_t time(time_t *t)`; devuelve el tiempo en segundos desde el Epoch que se refiere al 01-01-1970 00:00:00 +0000 UTC, si *t* no es NULL el resultado también se almacena en la variable apuntada por *t*.

Para obtener y fijar la hora del sist. se usa `int gettimeofday(...)` e `int settimeofday(...)` respectivamente, la estructura pasada como primer argumento está obsoleta y el puntero de la misma debe ponerse a NULL de forma que ni se modifica ni se retorna.

Para convertir la info. temporal a una cadena se usa `char *ctime(const time_t *time)`; que devuelve un puntero a datos estáticos, si se desea en cambio la cadena a medida se usa `size_t strftime(char *s, size_t max, const char *format, const struct tm *tm)`; el parámetro *format* es una cadena dónde se admiten diversas opciones, *%a, %A, %b, %B, %d, %H, %I, %M, %S, %p, %r, etc.*

Tema 2 – Sistema de ficheros

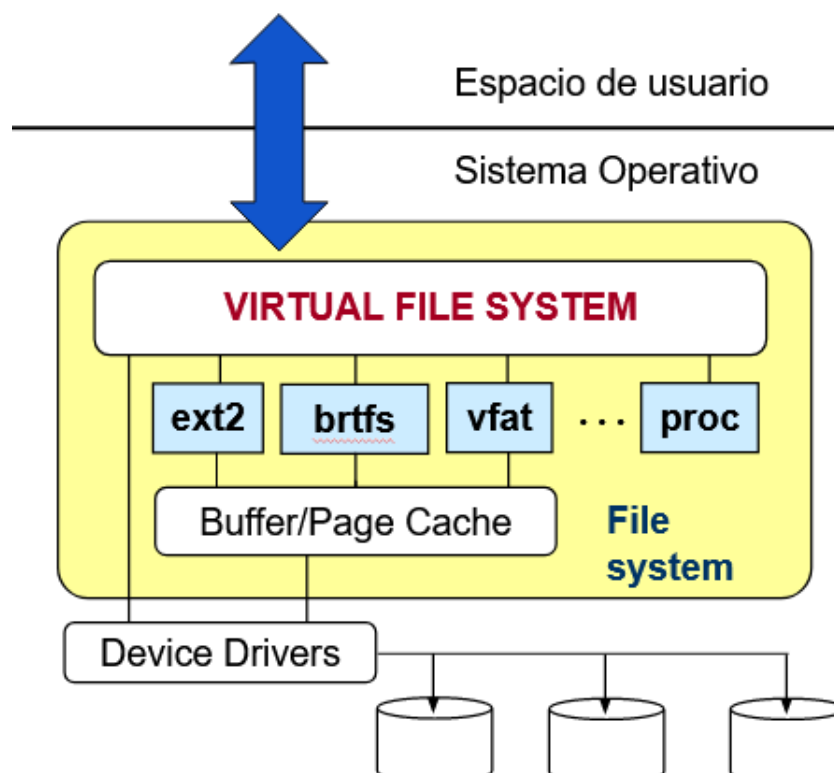
Los sist. de ficheros tienen una serie de características que dependen del punto de vista, desde el del usuario son una colección de ficheros y directorios usados para guardar y organizar info., mientras que desde el del SO son un cjto de tablas y estructuras que permiten gestionar los ficheros y directorios.

Tipos de sist. de ficheros:

- Basados en disco: Implementados en discos duros, Ej: ext2-3-4, FAT, NTFS, etc.
- Basados en red (o distribuidos): Se usan para acceder a sist. de ficheros remotos, normalmente se acceden como NFS (Network File System).
- Basados en memoria (o pseudomemoria): Residen en mem. principal mientras el sist. operativo se está ejecutando, Ej: procs, tmpfs.

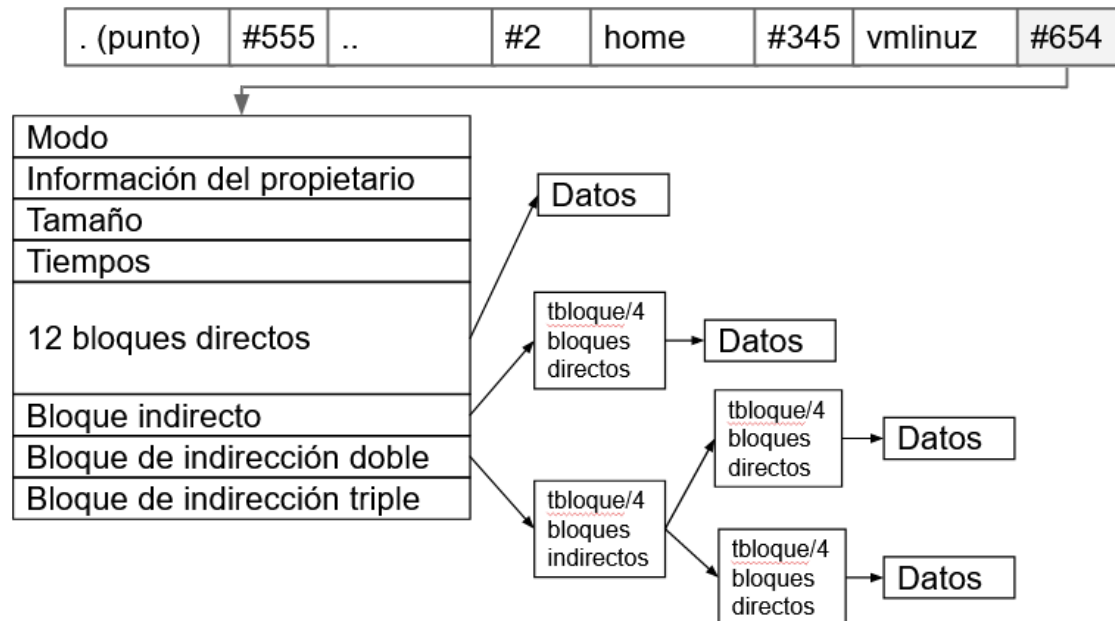
Virtual File System:

El sist. de ficheros usa la capa VFS (Virtual File System) para establecer un enlace entre el kernel del SO y los distintos sist. de ficheros, este enlace proporciona las diferentes llamadas para la gestión de archivos, independientemente del sist. de ficheros, de esta manera permite acceder a múltiples sist. de ficheros distintos y optimiza la entrada / salida por medio de la caché de i-nodos y entradas de directorio del VFS y la caché de buffers/páginas.



Estructura del sist. de ficheros ext2:

Prácticamente todos los sist. de ficheros se basan en el FFS (Fast File System), que era el sist. de ficheros de UNIXBSD, a partir de este y del UFS se desarrolló ext2, este sist. de ficheros tiene una estructura en la cual los bloques de datos están cerca de sus i-nodos y estos a su vez están cerca de su directorio de i-nodos correspondiente, cada directorio se gestiona como una lista enlazada de entradas con longitud variable, tienen un nº de i-nodo, un tamaño, un nombre y un fichero.



Journaling:

En el caso de que un sist. de ficheros tradicional no se apague correctamente, el SO debe comprobar la integridad y consistencia del sist. de ficheros en el siguiente arranque, esto implica recorrer toda la estructura del disco en búsqueda de i-nodos huérfanos, lo cual puede llevar mucho tiempo en sist. grandes, en ocasiones no es posible reparar automáticamente la estructura y se debe hacer de forma manual.

Los sist. de ficheros más modernos, como el ext3 y 4, incorporan un archivo especial llamado bitácora o *journal*, que evita la corrupción de datos escribiendo en este archivo los cambios en el disco, así, en caso de apagado brusco del sistema, se puede usar este archivo para devolver el sist. de ficheros a un estado coherente, a esta funcionalidad se le llama *journaling*.

Tipos de Journaling:

Los metadatos siempre se escriben en el archivo de bitácora de forma inmediata, dependiendo de cómo se escriben los datos existen tres variantes de *journaling*:

- Writeback mode: Los bloques de datos se escriben en el disco mediante el proceso estándar, los datos solo se pueden escribir en disco después de que sus metadatos hayan sido escritos (no se preserva el orden). Es el modo más rápido, pero puede ocasionar pérdidas de datos si el fallo se produce entre la actualización de la bitácora y la escritura de los datos.
- Ordered mode: Primero se escriben los datos en el disco y luego se actualiza el archivo de bitácora (se preserva el orden). Modo por defecto en ext3.
- Journal mode: Los datos también se escriben en el archivo de bitácora, esto ofrece mayor protección frente a fallos pero se degrada el rendimiento al tener que escribir los datos dos veces en el disco.

La consolidación de los datos puede hacerse periódicamente o cuando el archivo de bitácora se llena hasta cierto punto.

Apertura de ficheros

Para la apertura y/o creación de un archivo se usa `int open(const char *path, int flags);` o `int open(const char *path, int flags, mode_t mode);` donde path es un puntero que apunta al principio del descriptor de fichero. Los flags disponibles son:

- `O_RDONLY` (solo lectura)
- `O_WRONLY` (sólo escritura)
- `O_RDWR` (lectura y escritura).

mode determina los permisos con los que se creará el archivo, que se ven modificados por el *umask* del proceso y puede ser:

- `O_CREAT` (si el archivo no existe lo crea).
- `O_EXCL` (provoca un error si el archivo ya existe).
- `O_TRUNC` (una vez abierto el archivo puede ser modificado).
- `O_APPEND` (antes de realizar cualquier escritura se sitúa el puntero de archivo a la última posición del fichero).
- `O_NONBLOCK` (abre el archivo en modo no bloqueante).
- `O_SYNC` (abre el archivo en modo síncrono, bloqueando las llamadas *write* hasta que los datos sean físicamente escritos).

Atributos de ficheros:

Existen funciones cuyo objetivo es la obtención de info. de ficheros, para ejecutarlas no se necesitan permisos sobre el archivo, pero sí para buscar en PATH.

`int stat(const char *file_name, struct stat *buf);` sirve para obtener el estado de un fichero.

`int lstat(const char *file_name, struct stat *buf);` para obtener el estado de ficheros y enlaces.

`int fstat(int filedes, struct stat *buf);` para obtener el estado de ficheros mediante descriptores.

Los posibles errores que pueden dar estas funciones son:

- EBADF: Descriptor no valido.
- ENOENT: Path incorrecto o nulo.
- ENOTDIR: Componente del PATH no es un directorio.
- ELOOP: Demasiados links en la búsqueda.
- EFAULT: Dir. no válida.
- EACCES: Permiso denegado.
- ENAMETOOLONG: Nombre de archivo muy largo.

La estructura `stat` devuelta tiene la forma:

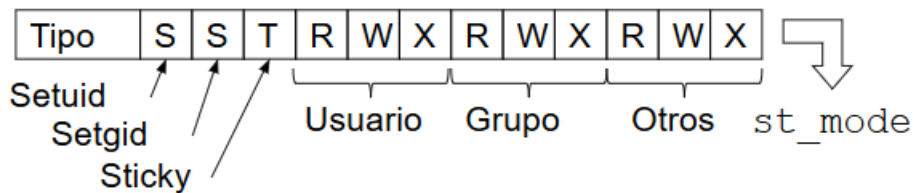
```
struct stat {
    dev_t st_dev; // Dispositivo que contiene el inodo
    ino_t st_ino; // I-nodo
    mode_t st_mode; // Permisos
    nlink_t st_link; // Numero de enlaces duros (hard)
    uid_t st_uid; // UID del propietario
    gid_t st_gid; // GID del propietario
    dev_t st_rdev; // Dispositivo, si fichero especial
    off_t st_size; // Tamaño (bytes)
    unsigned long st_blksize; // Tamaño de bloque E/S
    unsigned long st_blocks; // Bloques reservados
    time_t st_atime; // Último acceso
    time_t st_mtime; // última modificación
    time_t st_ctime; // Último cambio (I-nodo)
}
```

El estándar POSIX ofrece una serie de flags para comprobar el tipo de archivo y permisos:

- `S_ISLNK`: es un enlace simbólico.
- `S_ISREG`: es un fichero normal.
- `S_ISDIR`: es un directorio.
- `S_ISCHR`: es un dispositivo por caracteres.
- `S_ISBLK`: es un dispositivo por bloques.
- `S_ISFIFO`: es un FIFO o pipe.
- `S_ISSOCK`: es un socket.

Permisos:

Para cambiar el tipo de permisos se utilizan `int chmod(const char *path, mode_t mode);` e `int fchmod(int filedes, mode_t mode);`



La modificación de los permisos ha de hacerse mediante operaciones lógicas (bitwise), el UID efectivo del proceso debe ser 0 (root) o coincidir con el del propietario del fichero. El bit T, *Sticky* se usa para que otros usuarios no borren tus ficheros, indicando que se ha de fijar en el permiso de escritura del fichero, no del directorio.

Para comprobar el tipo de permisos sobre un fichero se usa `int access(const char *path, int mode);` el modo es una combinación de los flags:

- *R_OK*: El archivo existe y tenemos permisos de lectura.
- *W_OK*: El archivo existe y tenemos permisos de escritura.
- *X_OK*: El archivo existe y tenemos permisos de ejecución.
- *F_OK*: El archivo existe.

En la comprobación de los permisos se tiene en cuenta la ruta completa, la comprobación se realiza con los identificadores de usuario y grupo reales, a diferencia de la escritura o lectura. La llamada a la función falla si alguno de los permisos no se cumple.

El argumento de permisos que se usa en la llamada `open()`, no es necesariamente los que recibe el archivo creado. Para establecer la máscara de permisos para a apertura de ficheros se usa `mode_t umask(mode_t mask);` de manera que la función `open` establece los permisos del nuevo archivo de la forma:

`Permisos = mode & (!umask).`

Es útil para especificar permisos que nunca se concederán a los archivos creados en el programa, la función siempre se ejecuta correctamente devolviendo la mask anterior.

Duplicación de descriptores:

Para crear un descriptor que se refiera a otro archivo previamente abierto se usa `int dup(int old_fd);` o `int dup2(int old_fd, int new_fd);` ambos descriptors comparten los cerrojos, punteros en el archivo y opciones del fichero, de forma que puede intercambiarse su uso. El descriptor devuelto por `dup()` es el menor disponible en el sistema. Con `dup2()`, `new_fd` referirá a `old_fd` y si `new_fd` está abierto, se cerrará.

Se puede producir un error EBADF si `old_fd` no corresponde a un fichero abierto o `new_fd` está fuera de rango. Otro error posible es EMFILE si se alcanza el nº máx. de archivos abiertos.

Si se ejecuta el comando `ls > out` se redirige la salida de `ls` a un fichero `out`, de forma que la shell abre el fichero `out` (`fd = open("out", ...)`), cierra la salida estándar (`close(1)`), duplica el descriptor de fichero `fd` (`dup(fd)`) y cierra el fichero `fd` (`close(fd)`).

Lectura y escritura en fichero:

Para escribir, leer, modificar, o cerrar un fichero se usan las funciones:

```
ssize_t write(int fd, void *buffer, size_t count);,
ssize_t read(int fd, void *buffer, size_t count);
off_t lseek(int fd, off_t offset, int where);
int close(int fd);
```

Es importante no mezclar las funciones de librería con las llamadas al sist. `fopen`, `open`, `fread`, `read`, `fwrite`, `write` o clases `fstream` en C++.

La escritura de los ficheros se realiza mediante buffers de sist. intermedios y accesos eficientes. Para evitar la pérdida de info. se usa el flag `O_SYNC` en `open()`, que sincroniza fichero y disco en cada escritura, además se realiza la sincronización explícita mediante una llamada `int fsync(int fd)`, así la llamada no retorna hasta que termina la sincronización.

Enlaces simbólicos y duros:

Para crear un enlace duro:

```
int link(const char *exist, const char *new);
```

Únicamente se realiza sobre archivos en el mismo sist. de ficheros, si el nuevo archivo no existe no será sobrescrito.

Para crear un enlace simbólico:

```
int symlink(const char *exist, const char *new);
```

Pueden realizarse entre archivos en distintos sist. de ficheros, o a pesar de que el archivo original no exista, si el nuevo archivo existe no será sobrescrito.

Para leer el contenido de la ruta de un enlace simbólico se usa:

```
int readlink(const char *path, char *b, size_t tb);
```

El tamaño del enlace puede determinarse con *lstat*, la cadena *b* no contiene el carácter fin de cadena, con *lstat* conseguimos info del enlace y no del fichero enlazado, *stat* no serviría, ya que no brinda info. sobre enlaces.

Borrado de ficheros:

Para eliminar un nombre de fichero y el fichero se usa

```
int unlink(const char *name);
```

Borra la entrada del dir. y decrementa el nº de referencias del i-nodo, si este nº se reduce a 0 y no hay ningún proceso que mantenga el archivo (fifo, socket o dispositivo) abierto entonces se elimina, devolviendo el espacio al sist.

Control de ficheros:

Para manipular un descriptor de fichero se usa:

```
int fcntl(int fd, int cmd); e int fcntl(int fd, int cmd, long argv);
```

cmd determina la operación que se realizará sobre el archivo, pudiendo ser:

- F_DUPFD: Duplica el descriptor en la forma de *dup()*.
- F_GETFD: Obtiene los flags del descriptor.
- F_SETFD: Fija los flags del descriptor.
- F_GETFL: Obtención de los flags del archivo como se fijaron con *open()*.
- F_SETFL: Fija los flags (*argv*) del descriptor: O_APPEND, O_NONBLOCK y O_ASYNC.

Cerrojos

Para bloquear las regiones de un fichero se usa:

```
int fcntl(int fd, int cmd, struct flock *lock);
```

El argumento *lock* (cerrojo) es una estructura de la forma:

```
struct flock {  
    short int l_type; // F_RDLCK, F_WRLCK o F_UNLCK  
    short int l_whence; // SEEK_SET, SEEK_CUR o SEEK_END  
    off_t l_start; // Offset de la región bloqueada  
    off_t l_len; // Long. de la región, 0 = hasta el final  
    pid_t l_pid; // Proceso que mantiene el cerrojo (F_GETLK)  
}
```

Existen dos tipos de cerrojos:

- De lectura o compartido (F_RDLCK): El proceso está leyendo el área bloqueada por lo que no puede ser modificada. Pueden establecerse varios cerrojos sobre una misma región.
- De escritura o exclusivo (F_WRLCK): El proceso está escribiendo, por lo que ningún otro debe leer o escribir del área bloqueada, solo puede haber un cerrojo.

cmd determina qué operación se hará sobre el cerrojo, pudiendo ser:

- F_GETLK: Si se puede activar el cerrojo *lock*, establece el campo *l_type* a F_UNLCK. Si uno o más cerrojos incompatibles impiden que se pueda activar, devuelve los detalles de uno de ellos en los campos *l_type*, *l_whence*, *l_start* y *l_len*, y establece el campo *l_pid* al PID del proceso que lo mantiene.
- F_SETLK: Activa (si *l_type* es F_WRLCK o F_RDLCK) o libera (si *l_type* es F_UNLCK) el cerrojo *lock*. Si otro proceso mantiene un cerrojo incompatible, devuelve -^o y pone *errno* a EACCESS o EAGAIN.
- F_SETLKW: Igual que F_SETLK, pero si hay un cerrojo incompatible espera a que sea liberado.

Los cerrojos son consultivos, es decir, *read* y *write* no comprueban su existencia.

Además, se asocian al proceso, por lo que no se heredan con *fork*. Los cerrojos activos se pueden consultar en */proc/locks*.

flock(2) proporciona los antiguos cerrojos de ficheros de UNIX (no POSIX), esta función aplica o elimina un bloqueo de aviso en un archivo abierto.

lockf(3) proporciona un interfaz para cerrojos de regiones basado en *fcntl*, esta función aplica, chequea o elimina un bloqueo POSIX en un archivo abierto.

Acceso a directorios:

Para el acceso a estos directorios se usan funciones de biblioteca, no llamadas al sistema.

Para abrir un directorio se usa: `DIR *opendir(const char *name);`

Esta función devuelve un puntero al flujo de directorio, posicionado en la primera entrada del mismo. El tipo de datos DIR se usa de forma similar al tipo *FILE* especificado por la librería de entrada/salida estándar.

Para leer un directorio se usa: `struct dirent *readdir(DIR *dir);`

Retorna una estructura *dirent* que apunta a la siguiente entrada en el directorio, devuelve NULL si llega al final u ocurre un error.

Para cerrar un directorio se usa: `int closedir(DIR *dir);`

Cierra el directorio definido por el descriptor, haciéndolo inaccesible a subsecuentes llamadas.

Creación y borrado de directorios:

Se usan las funciones:

```
int mkdir(const char *path, mode_t mode);  
int rmdir(const char *path);
```

mode indica los permisos con los que se crea el directorio, modificados en la forma habitual (*mode & !unmask*). En la creación de los directorios el usuario y grupo del nuevo directorio serán los efectivos del proceso.

La función `int rename(const char *old, const char *new);` cambia el nombre de un fichero o directorio, si *new* existe, se elimina antes de asociarlo con *old*. Si *new* es un directorio, ha de estar vacío. Tanto *old* como *new* han de ser del mismo tipo (ficheros o directorios) y pertenecer al mismo sist. de ficheros. Si *old* es un enlace simbólico, será renombrado. Si *new* es un enlace simbólico, será sobrescrito.

Tema 3 – Gestión de procesos

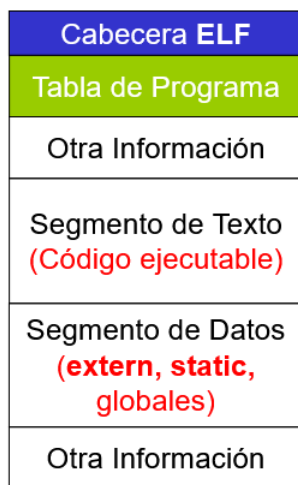
Estructura de un programa:

Def Programa: Entidad pasiva, cjto de instrucciones en código máquina y datos, almacenados en una imagen ejecutable en disco.

Un programa está compuesto de:

- Una cabecera que contiene la info. imprescindible para ejecutarse.
- Una tabla de programa, que contiene una serie de info. de cabeceras de programa que incluyen a su vez info. acerca del contenido de ese fichero ejecutable, son:
 - LOAD: incluye segmentos de texto o datos que hay que cargar en memoria.
 - DYNAMIC: incluye info. acerca del enlace dinámico, para enlazarlo en tiempo de ejecución.
 - PHDR: La propia cabecera del programa
- Segmento de texto que contiene el código ejecutable con la sección *.text* y la sección *.rodata* (datos de solo lectura). Necesita permisos de R y X.
- Segmento de datos que contiene la sección *.data* que contiene a su vez variables inicializadas (variables estáticas y globales), y la sección *.bss* (datos estáticos globales inicializados a 0). Necesita permisos de R y W.

Executable and Linking Format (ELF)



Organización y Atributos

```
typedef struct{
    Elf32_Half e_type;
    Elf32_Half e_machine;
    ...
} Elf32_Ehdr;
```

- Objeto
- Ejecutable
- Objeto Dinámico
- CORE
- EM_SPARC
- EM_386
- EM_IA_64
- ...

Información para la ejecución

```
typedef struct{
    Elf32_Word p_type;
    Elf32_Addr p_vaddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    ...
} Elf32_Phdr;
```

- PT_LOAD
- PT_DYNAMIC
- PT_PHDR
- ...

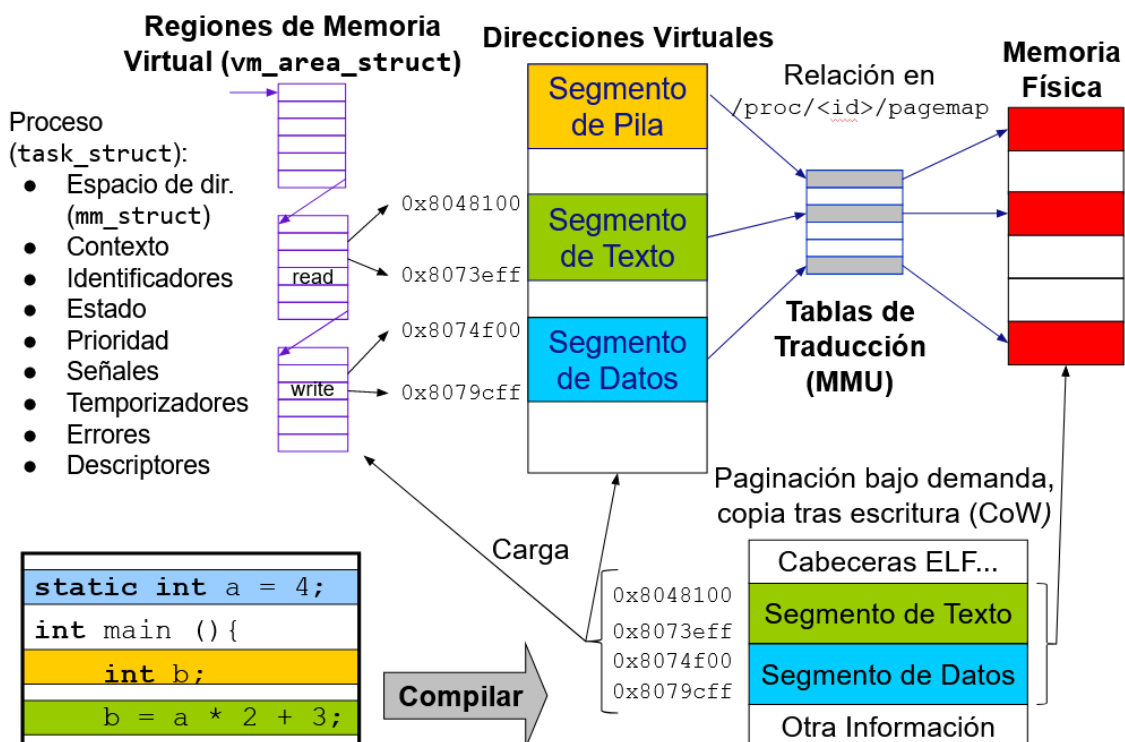
Dirección Virtual del segmento

Estructura de un proceso:

Con la info. del fichero *.elf* se lee esta información y se crean los distintos segmentos de texto, datos, la pila, etc.

La estructura del proceso se almacena en *task_struct*, el espacio de direcciones se almacena en una lista enlazada de segmentos que incluyen info. de las dir. virtuales de memoria en las que termina o empieza el segmento en cuestión, e incluyen un puntero a la siguiente sección.

El espacio continuo virtual se mapea y se traduce a través de unas tablas de traducción (MMU) a páginas físicas de mem. que no serían contiguas, esto se hace con un proceso de paginación bajo demanda.

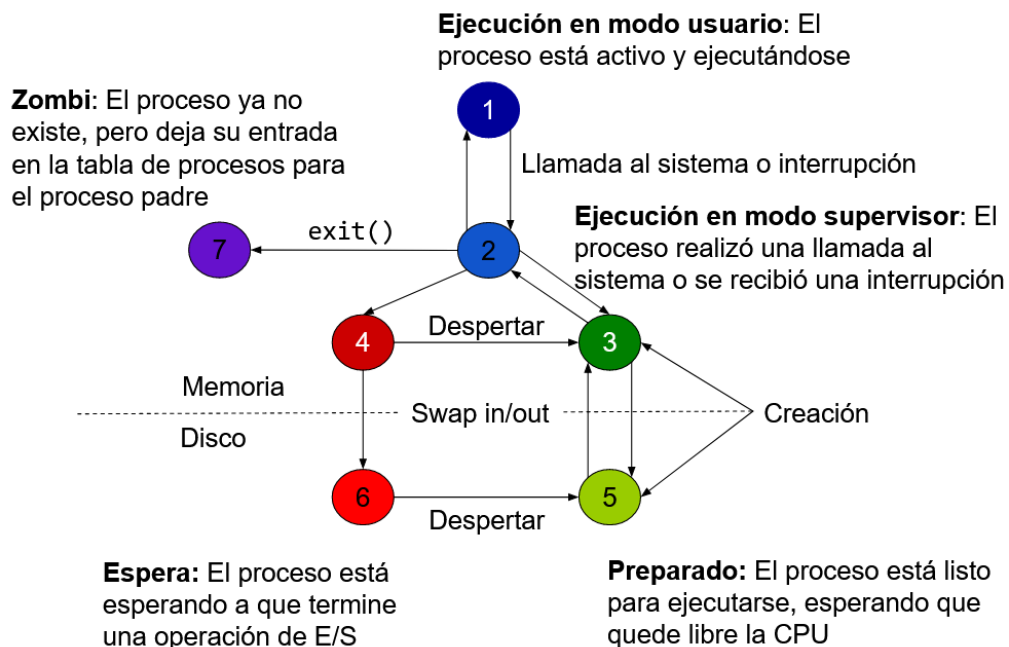


Cuando se hace una copia de un proceso (por ejemplo un *fork()*) ambos procesos padre e hijo comparten toda la info. y la mem. mientras accedan solamente para leer, en el momento que alguno de los dos accede para escribir en memoria es cuando se hace una copia de la info. y dejan de compartirla, por esto se llama copia tras escritura.

Para ver info. del mapa de mem. se mira en el directorio */proc/*, por ejemplo se puede acceder a *proc/\$\$* y en el fichero maps: "*more maps*" está el mapa de mem. del proceso.

Estados de un proceso:

Existen distintos estados para un proceso, en este diagrama se muestran los estados estándar de un proceso, es decir, los procesos típicos de UNIX.



Linux no los implementa así, ya que:

- No implementa el *swapping* de procesos (funcionalidad según la cual se saca un proceso de mem. siguiendo algún criterio concreto, debido a que el paginador no es capaz de conseguir marcos libres a la velocidad necesaria).
- No distingue entre procesos en mem. y en disco, pero si hace paginación, es decir, mueve una página de mem. y la lleva a disco.
- Tampoco diferencia entre procesos en ejecución y aquellos preparados para ejecutarse (todos se consideran *running*), el proceso puede estar ejecutándose en la CPU o esperando para ejecutarse porque está preparado para ello.

Existe también el estado *waiting* con tres tipos de espera:

- Interrumpible: enviando una señal se puede despertar al proceso.

- No interrumpible: el proceso espera a que una operación de E/S termine.
- No interrumpible con respuesta a señales fatales: Surgió debido a que se puede producir una situación en la cual un proceso se bloquea esperando en disco y el disco no responde, por lo tanto el proceso no se puede matar. En este estado se da la posibilidad de enviar una señal a este tipo de procesos y matarlos.

Por último existen también el estado *stopped*, que normalmente se usa para trazar un proceso (depuradores), y el estado *zombie* (proceso que ha terminado pero sigue apareciendo en la tabla de procesos porque el padre no ha hecho el *wait*).

Planificador:

Def Planificador: Es un algoritmo del núcleo del sistema que determina el orden de ejecución de los procesos, para ello tiene en cuenta la clase de planificación y la prioridad, es un sist. expropiativo (puede quitar unos procesos para poner otros).

Existen varios tipos de políticas de planificación:

- SCHED_OTHER: Política estándar, la prioridad estática es 0, la prioridad dinámica se determina mediante el valor de *nice* del proceso, que especifica un reparto de la CPU, ya que en función del valor el porcentaje de uso de la CPU de un proceso va a ser distinto a otro. (mayor prioridad -> -20, 19 menor prioridad).
- SCHED_FIFO: Prioridad estática mayor que 0, entre 1 y 99, la tarea de prioridad menor siempre se ejecuta antes, y siempre expropia a los procesos de la clase SCHED_OTHER, además a estas tareas (SCHED_FIFO) no se las puede echar de la CPU.
- SCHED_RR: Igual que la anterior, pero a cada proceso se le asigna un cuanto de tiempo de ejecución.

Las llamadas afectan realmente a la estructura del *thread*, ya que el planificador los maneja. Las llamadas `fork()` heredan los atributos de planificación.

El comando *chrt* nos ofrece acceso a varias funcionalidades del sistema:

- *sched_setscheduler*: permite establecer la política de planificación y/o la prioridad.
- *sched_getscheduler*: permite consultar tanto la política como la prioridad.
- *sched_setparam*: permite fijar una nueva prioridad, pero no cambiar la política.
- *sched_getparam* permite consultar los parámetros y la prioridad.
- *sched_get_priority_max*: permite consultar la prioridad máxima.
- *sched_get_priority_min*: permite consultar la prioridad mínima.

Para fijar y consultar la prioridad (*nice*) de un proceso se usan las funciones:

```
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

which: hace referencia a la clase que verá modificada su prioridad, (PRIO_PROCESS, PRIO_PGRP o PRIO_USER).

who: en función de la clase es un ident. de un proceso, un ident. de grupo de procesos, o un ident. de usuario.

prio: valor de la nueva prioridad (0 por defecto) $19 > prio > -20$, valores más bajos corresponden a mayor prioridad.

Los comandos *nice* y *renice* permiten acceder a esta interfaz. Consultar *errno* si hay errores.

Atributos de un proceso:

Cada proceso tiene un identificador único llamado *Process Identifier*, *PID*, y registra el proceso que lo inició *Parent PID*, *PPID*. Estos identificadores se pueden consultar con:

```
pid_t getpid(void); e pid_t getppid(void);
```

Los procesos pertenecen a un grupo de procesos, con un identificador de grupo *PGID*, que es igual al ident. de procesos del proceso líder (el que creó el grupo), el principal uso de los grupos de procesos es la distribución de señales. Se consultan con:

```
int setpgid(pid_t pid, pid_t pgid) y pid_t getpgid(pid_t pid);
```

```
int setpgrp(void) y pid_t getpgrp(void);
```

Los grupos de procesos se pueden agrupar en sesiones, que se usan para gestionar el acceso al sistema, el caso típico es cuando entramos a un sistema y el proceso login crea una sesión, todos los procesos y grupos del usuario pertenecen a esa sesión, luego en la desconexión todos los procesos asociados a la sesión de ese usuario reciben la señal *SIGHUP*, (colgar), por lo que terminan y se eliminan.

Un proceso sólo puede crear una sesión nueva y un grupo nuevo, de los que se convertirá en el líder y tendrá un grupo igual, para esto se usa las llamadas:

```
pid_t setsid(void) y pid_t getsid(pid_t pid)
```

Para obtener una lista de procesos se usa el comando *ps*, que se puede consultar por *pid*, por *ppid*, por sesión, por estados de procesos, etc.

Recursos de un proceso:

Los procesos tienen impuestos unos determinados límites en el uso de los recursos del sistema, para acceder y modificar esos límites se usan:

```
int getrlimit(int resource, struct rlimit *rlim)
int setrlimit(int resource, const struct rlimit *rlim)
```

resource indica el recurso a consultar o establecer, y el segundo parámetro es el límite.

```
resource:
  o RLIMIT_CPU: Max. tiempo de CPU (segundos)
  o RLIMIT_FSIZE: Max. tamaño de archivo (bytes)
  o RLIMIT_DATA: Max. tamaño del heap (bytes)
  o RLIMIT_STACK: Max. tamaño de pila (bytes)
  o RLIMIT_CORE: Max. tamaño de archivo core (bytes)
  o RLIMIT_NPROC: Max. número de procesos
  o RLIMIT_NOFILE: Max. número de descriptores de archivo

  struct rlimit{
      int rlim_cur; /* Límite actual */
      int rlim_max; /* Límite máximo */
  };

  o RLIM_INFINITY indica ilimitado
```

También se puede acceder a estos recursos usando *ulimit -a*, que nos muestra los límites que están establecidos, estos límites y su prioridad se heredan entre procesos. Además, se puede tener acceso al uso de los recursos del sist. utilizados por un proceso, esto se hace con:

```
int getrusage(int who, struct rusage *usage)
```

Esta función nos devuelve un puntero a una estructura con mucha info. acerca de los recursos que ha usado el proceso, *who* puede ser *RUSAGE_CHILDREN* (los hijos del proceso) o *RUSAGE_SELF* (el proceso y los hijos del mismo). Para tener acceso a esto se puede usar *usr/bin/time -v sleep 1*, el parámetro *time* se usa para calcular el tiempo usado por un programa, con la opción *-v*.

```
struct rusage {
    struct timeval ru_utime; // struct con time_t
    struct timeval ru_stime; // tv_secs y tv_usecs
    long ru_maxrss;
    ...
}
```

Para conocer el uso de CPU, como los ticks de reloj respecto a un punto de referencia, normalmente el arranque del sistema se usa:

```
clock_t times(struct tms *buffer);
```

El directorio de trabajo de un proceso se usa para resolver las rutas relativas usadas, se puede consultar con `getcwd`:

```
char *getcwd(char *buffer, size_t size);
```

Buffer cuya longitud es *size*, considerar el carácter '\0' fin de cadena.

Para cambiar el directorio de trabajo: `int chdir (const char *path);`

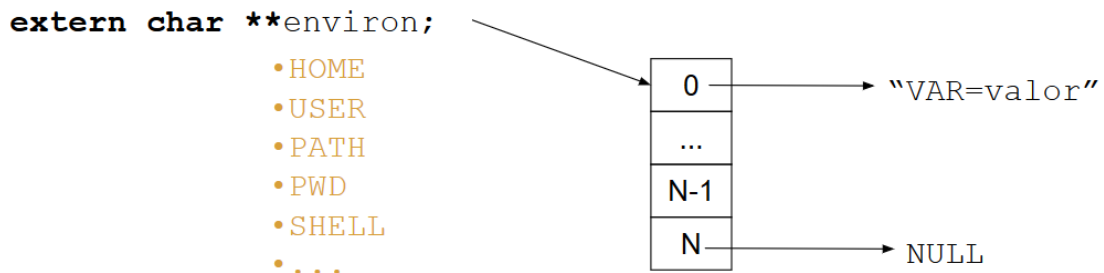
Todos los procesos se ejecutan en un entorno que normalmente es heredado del proceso padre, menos en casos especiales.

El entorno es un cjto de cadenas de forma "VARIABLE=valor, algunas aplicaciones limitan el entorno que pasan a los procesos por ejemplo *sudo*.

Desde un programa el entorno se apunta con la variable *environ*** que es un array de cadenas que termina con el valor NULL.

Para consultar se usa *getenv()*, para modificar *setenv()* y para eliminar *unsetenv()*.

Crear una variable de entorno: `export PRUEBA=prueba.`



Creación de procesos:

Para la ejecución de programas normalmente podemos usar dos formas:

- Combinación de *fork()* y *exec()*:
 - *fork()* crea un nuevo proceso que es una copia del actual, de manera que los dos procesos ejecutan el mismo código y comparten info. como los descriptores de fichero, etc. Es la única llamada que retorna dos veces, una en el padre y otra en el hijo, en función del valor de retorno:
 - -1: ha habido fallo y no se ha creado el hijo.
 - 0: Se ha ejecutado el proceso hijo.
 - >0: Se ha ejecutado el proceso padre, el valor devuelto es el PID del proceso hijo.

El hijo no hereda cerrojos o alarmas, y su cjto de señales pendientes es nulo.

`pid_t fork(void); // -1 fallo, 0 ejecuta hijo, >0 ejecuta padre.`

- *exec()* ejecuta el programa que le pasamos en el proceso actual, que se sustituye por la imagen del nuevo programa.

```
int execl(const char *path, const char *a0, ...); // Listas
```

```
int execlv(const char *path, char *const argv[], ...); // Vectores
```

La otra más sencilla es usar la llamada `system()` que ejecuta el comando especificado por string mediante `bin/sh -c string`, bloqueando la llamada y retornando al flujo del programa cuando termina, si no se produce un error entonces `system` devuelve el código de salida correspondiente.

```
int main() {
    pid_t mi_pid, pid;
    pid = fork();
    switch (pid) {
        case -1:
            perror("fork");
            exit(-1);

        case 0:
            pid_t mi_pid = getpid();
            printf("Proceso hijo %i (padre: %i)\n", mi_pid, getppid());
            break;

        default:
            mi_pid = getpid();
            printf("Proceso padre %i (hijo %i)\n", mi_pid, pid);
            break;
    }
    return 0;
}
```

Finalización de procesos:

La finalización de un proceso puede ocurrir por dos motivos:

- Llamar voluntariamente a `exit()` o `return` desde `main()`.
- Recibiendo una señal, múltiples causas.

`void exit(int status);`

status es el código de salida, que debe ser un nº < 255. Por convenio si el código de salida es 0 -> éxito y 1 -> error. No se debe usar `exit(-1)` sino `exit(1)` o `exit_failure`.

La función `wait()` es la forma que tiene un proceso padre de esperar la finalización de algún hijo y que no se quede zombie, liberando entonces sus recursos asociados, hay dos formas de usarla:

```
pid_t wait(int *estado);
pid_t waitpid(pid_t pid, int *estado, int op);
```

pid puede ser <-1, entonces se espera la finalización de un hijo cuyo *PGID* es *-pid*, -1 igual al *wait()*, 0 espera la finalización de un hijo del grupo de procesos del padre y >0 que espera la finalización de un hijo con identificador *pid*.

op puede ser *WNOHANG*, que retorna sin esperar si hay hijos que no hayan terminado, *WUNTRACED*, que retorna si el proceso ha sido detenido y *WCONTINUED*, si un hijo detenido ha sido reanudado.

La primera opción es la función sin más, que devuelve un estado, la segunda es la función *waitpid* en la que podemos indicar a qué hijo o hijos hay que esperar, y algunas opciones, nos devuelve el estado, que es un argumento de 16 bits cuyos 8 menos significativos son el estado de terminación tal como lo da la llamada *exit()*.

Señales:

Las señales son interrupciones SW generadas por un proceso que informan a otro de la ocurrencia de un evento de forma asíncrona.

Las opciones ante la ocurrencia del evento son bloquear la señal, ignorarla, invocar una rutina de tratamiento que generalmente termina con la ejecución del proceso, o invocar a una rutina de tratamiento propia.

Existen distintos tipos de señales, de terminación de procesos, excepciones, llamadas de sistema, interacción con el terminal, traza de proceso, etc.

Señales: Envío

Para enviar una señal a un proceso se usa: `int kill(pid_t pid, int signal);`

pid identifica el proceso que recibirá la señal, >0 es el ident. del proceso, 0 se envía a todos los procesos del grupo, -1 se envía a todos los procesos (de mayor a menor), <-1 se envía a todos los procesos del grupo con *PGID* igual a *-pid*.

signal es la señal que se enviará, que puede ser:

- **SIGHUP:** Desconexión de terminal, terminación de proceso.
- **SIGINT:** Interrupción, equivale a Ctrl + C.
- **SIGQUIT:** Finalización, equivale a Ctrl + \.
- **SIGILL:** Instr. ilegal (punteros a funciones mal gestionados).
- **SIGTRAP:** Ejecución paso a paso, enviada después de cada instr.
- **SIGKILL:** Terminación brusca, no puede ignorarse.
- **SIGBUS:** Error de acceso a memoria (alineación o dir. no válida).
- **SIGSEGV:** Violación de segmento de datos.
- **SIGPIPE:** Intento de escritura en una tubería sin lectores.

- **SIGALARM:** Despertador, contador a 0.
- **SIGTERM:** Terminar proceso, el sist. advierte que el proceso debe terminar su ejec.
- **SIGUSR1, SIGUSR2:** Señales de usuario, terminación.
- **SIGCHILD:** Terminación del proceso hijo.

Las llamadas *raise()* y *abort()* son equivalentes a la llamada *kill()*.

```
raise(signal) ⇒ kill(getpid(), signal)
abort() ⇒ kill(getpid(), SIGABRT)
```

Señales: Cjtos de señales

La gestión de señales se puede realizar individualmente por su nombre o ident. o conjuntamente usando cjtos de señales *sigset_t* donde cada bit representa una señal.



Existen métodos para inicializar el cjto a vacío o lleno (todas las señales incluidas), para añadir y eliminar señales o para comprobar si una señal pertenece al cjto.

```
#include <signal.h>
...
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGQUIT);
...
```

Es posible proteger regiones del código contra la recepción de una señal, de esta manera se quedarán bloqueadas hasta que sean desbloqueadas explícitamente:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

how define el comportamiento de la función, puede ser:

- **SIG_BLOCK:** Añade el cjto *set* al cjto de señales actualmente bloqueadas.
- **SIG_UNBLOCK:** El cjto *set* se retira del cjto de señales bloqueadas.
- **SIG_SETMASK:** Reemplaza el cjto de señales actuales por *set*.

oset almacena el cjto previo de señales bloqueadas.

Para la comprobación de señales pendientes se utiliza:

```
int sigpending(const sigset_t *set);
```

```
sigset_t blk_set;

sigemptyset(&blk_set);
sigaddset(&blk_set, SIGINT);
sigaddset(&blk_set, SIGQUIT);

sigprocmask(SIG_BLOCK, &blk_set, NULL);

/* Actualización de la base de datos, evitando la
   corrupción de los datos */

sigprocmask(SIG_UNBLOCK, &blk_set, NULL);
```

Señales: Bloqueo

Para el control de las señales se modifica el comportamiento por defecto, que suele ser la finalización del proceso o ignorar la señal, instalando una función llamada *handler* que se ejecute en la recepción de la señal evitando la finalización del proceso.

```
int sigpending(const sigset_t *set); int sigaction(int signal, const
struct sigaction *act, struct sigaction *oldact);
```

signal especifica la señal, a excepción de SIGKILL y SIGSTOP.

act instala el nuevo *handler* para la señal si no es NULL.

oldact almacena el antiguo *handler* de la señal si no es NULL.

```
struct sigaction {
    void (*sa_handler) (int);
    sigset_t sa_mask;
    int sa_flags;
}
```

sa_handler es un puntero a función que constituye el nuevo *handler* para la señal, su valor puede ser SIG_DFL para el *handler* por defecto, SIG_IGN para no atender la señal o un puntero a una función: *void handler(int signal)*;

sa_mask es un cjo de señales que serán bloqueadas durante el tratamiento de la señal, además se bloquea la señal en cuestión.

sa_flags modifica el comportamiento del proceso de gestión de la señal.

El uso de *sigaction* suspende toda ejecución y llama al *handler*, restaurando la ejecución en el punto donde se produjo la señal, es importante no usar variables *extern* o *static* ni funciones como *malloc* o *free*, además de guardar y restaurar el valor de *errno*. Es importante definir como *volatile* las variables al fijar los flags.

Para esperar la ocurrencia de alguna señal suspendiendo la ejec. del proceso se usa:

```
int sigsuspend(const sigset_t *set);
```

La máscara de señales bloqueadas se sustituye temporalmente por el cjo *set*, el proceso se suspende hasta que una señal que no esté en la máscara se produzca, entonces se ejecuta el *handler* asociado a la señal y continúa la ejec. del proceso.

La forma más sencilla para suspender un proceso es:

```
unsigned int sleep(unsigned int segundos);
```

Esta función suspende la ejec. del proceso durante los *segundos* especificados o hasta que se reciba una señal que deba ser tratada.

Señales: Alarmas

Para fijar una alarma se usa: `unsigned int alarm(unsigned int secs);`

Usa el temporizador ITIMER_REAL para programar una señal SIGALARM, *secs* es el nº de segundos a los que se fija el temporizador, = 0 no se planifica ninguna nueva alarma

Esta función devuelve el valor de segundos restantes para que se produzca el final de la cuenta, debe instalarse previamente el controlador.

Las alarmas asociadas a otros temporizadores se obtienen y se fijan con:

```
int getitimer(int which, struct itimerval *value);
int setitimer(int which, struct itimerval *value, struct itimerval
*ovalue);
```

which puede ser ITIMER_REAL es el tiempo real, ITIMER_VIRTUAL es el tiempo de CPU en modo usuario, y ITIMER_PROF es el tiempo de CPU en modo usuario y sist.

```
struct itimerval {
    struct timeval it_interval; /* Intervalo */
    struct timeval it_value;    /* Tiempo que queda */
}
```

Tema 4 – Comunicación entre procesos: Tuberías

Entre los mecanismos de sincronización existentes se encuentran aquellos que ejecutan procesos/threads en el mismo sistema:

- Señales (procesos).
- Ficheros con cerrojos.
- Mutex y variables de condición (threads de un proceso).
- Semáforos.
- Colas de mensajes.

Y los que ejecutan procesos/threads en distintos sistemas:

- Paso de mensajes, colas de mensajes. Están basados en sockets.

En cuanto a la compartición de datos entre procesos existen aquellos que ejecutan procesos en el mismo sistema:

- Memoria compartida.
- Tuberías sin nombre (pipes).
- Tuberías con nombre (FIFOs).
- Colas de mensajes.
- Basados en ficheros.

Y los que ejecutan procesos en distintos sistemas:

- Basados en sockets.

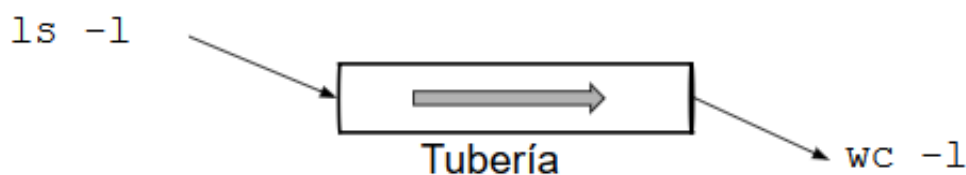
Tuberías sin nombre (pipe):

Def. Tubería sin nombre (pipe): Se trata de un mecanismo de compartición de datos entre procesos que soporta comunicación unidireccional entre ellos.

El sistema trata los pipes como ficheros, ya que tienen un i-nodo asociado, un descriptor de fichero y permiten operaciones de E/S típicas (read, write, etc) pero no de posicionamiento (*lseek*).

Los datos se van añadiendo por el final y siempre se lee primero el dato más antiguo (el que está al principio). Se heredan de padres a hijos (*fork()*), ya que la única forma de que dos procesos compartan un pipe es mediante herencia. La comunicación mediante pipes se realiza únicamente entre procesos con relación de parentesco.


La sutil diferencia entre un fichero y un pipe es que el último no ocupa espacio en disco, sino que reside en mem. principal.



Para crear un pipe se usa la llamada (ejemplo en *man pipe*):

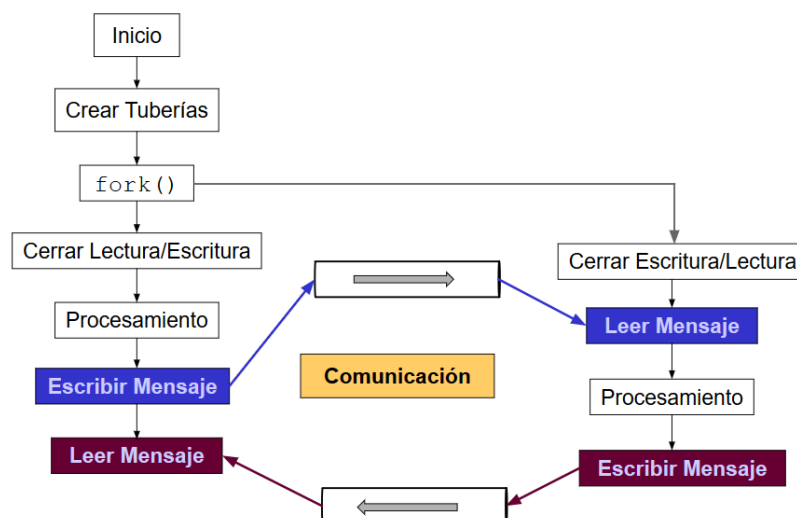
```
int pipe(int descriptor[2]);
```

A la función se le pasa un vector con dos enteros, el primero es el descriptor de fichero para lectura y el segundo el descriptor de fichero para escritura, la función crea la tubería y devuelve dos descriptors, que apuntan a cada extremo, para comunicarse con esa tubería ambos procesos tienen que estar relacionados (ser padre e hijo), solo mediante herencia. Las llamadas *read()* mientras el extremo de escritura este abierto se quedan bloqueadas, solo se desbloquean si hay datos o si se desconectan todos los escritores, entonces se devuelve un 0 (fin de fichero). En caso de que el *pipe* se llene, las llamadas a *write()* se quedarán bloqueadas.

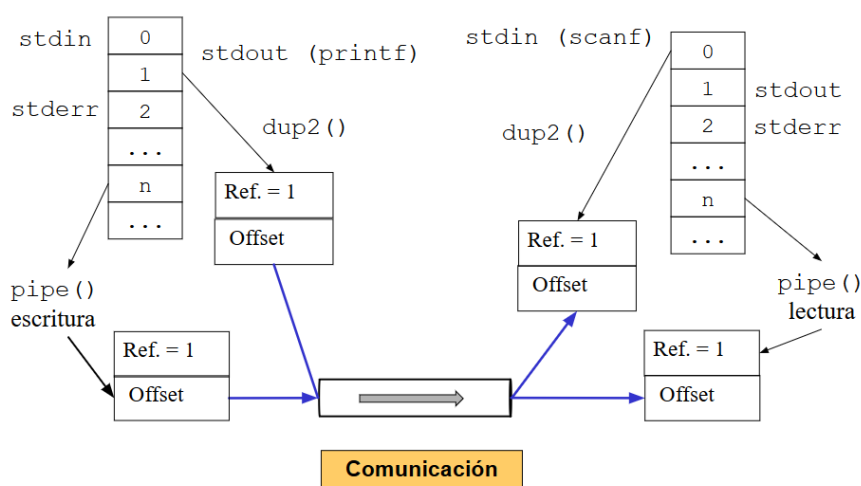
Escritura: `descriptor[1]`  Lectura: `descriptor[0]`

Los posibles errores son:

- EMFILE: Demasiados descriptors.
- ENFILE: Demasiados ficheros en el sistema.
- EFAULT: Array de descriptors no válido.



Ejemplo del uso de *pipes* en *man pipe*, modificarlo para hacer redirección, cerrar los descriptors y hacer el *exec()* para ejecutar los comandos.



Tuberías con nombre (FIFO):

Def. Tubería con nombre (FIFO): Funciona de manera semejante a un *pipe* no hace falta que exista una relación de parentesco entre los dos procesos para comunicarse entre sí. Es un tipo especial de fichero que ocupa una entrada en el directorio (nombre que apunta a un i-nodo que no tiene datos asociados, hace referencia a un *pipe*).

No se crean con la llamada *pipe()* sino que deben abrirse con la llamada *open()* para lectura o escritura, el núcleo del sist. almacena los datos internamente, sin escribirlos en el sist. de ficheros. Para que se puedan enviar datos deben abrirse ambos extremos (lectura y escritura) sino la llamada *open()* se queda bloqueada hasta que abramos el otro extremo.

Para la creación de FIFO's se pueden usar las siguientes llamadas:

- `int mknod(const char *filename, mode_t mode, dev_t dev);`
Se usa para crear algún tipo de fichero especial, *filename* es el nombre de fichero (archivo, dispositivo, tubería) que se creará.
mode especifica los permisos y el tipo de archivo que se creará, se fija mediante una OR lógica de permisos, y su tipo ha de ser:
 - S_IFREG: Archivo regular
 - S_IFCHR: Dispositivo de caracteres.
 - S_IFBLK: Dispositivo de bloques.
 - S_IFIFO: Tubería con nombre.
- `int mkfifo(const char *filename, mode_t mode);`
Específica para crear *FIFOs*, con *-m* podemos indicar permisos especiales.
filename es el nombre de la tubería que se creará.
mode determina los permisos de la tubería, modificados por el *umask* del proceso.

La apertura de tuberías bloquea el proceso hasta que otro se conecte al otro lado de la tubería, el *flag* O_NONBLOCK permite una apertura no bloqueante, si se hace una lectura con el *FIFO* vacío y hay un escritor conectado la operación se bloquea, si no hay escritor nos devuelve un 0 (fin de fichero).

Si el *FIFO* está lleno e intentamos escribir en él se bloquea, y si no hay lector conectado y se intenta escribir se recibe una señal SIGPIPE.

Operación E/S	Situación	Resultado
Lectura	FIFO vacía, con escritor	Se bloquea
Lectura	FIFO vacía, sin escritor	Devuelve 0
Escritura	FIFO llena, con lector	Se bloquea
Escritura	No hay lector conectado	Recibe SIGPIPE

Ejemplos de creación y uso de *FIFOs*:

mkfifo fifo

ls -l fifo (vemos que crea un fichero especial, permisos r,w)

echo prueba > fifo (escribimos en el *fifo* y se bloquea la apertura del fichero, hasta que no se abra el *fifo* por el extremo de lectura se va a quedar bloqueado).

En otro terminal escribimos:

cat fifo (leemos del *fifo*, al abrirse el extremo de lectura se escribe la cadena y se lee, el escritor escribe la cadena y cierra el *fifo*, el lector recibe un fin de fichero y termina).

Otro ejemplo:

strace cat > fifo (Lanzamos primero al escritor, se bloquea)

cat fifo (Abrimos el extremo lector y cerramos con Ctrl+C, si luego el escritor intenta escribir recibirá una señal SIGPIPE).

Es posible tener varios lectores o escritores, pero solamente uno de los lectores recibe el mensaje.

Sincronización de E/S:

Cuando un proceso gestiona la sincronización de varios canales de E/S, debe seleccionar los que están listos en cada momento para realizar la operación, esto aplica a cualquier canal de E/S, pueden ser *pipes*, *sockets*, etc.

Para gestionar qué canales escuchamos y que la llamada *read()* no se quede bloqueada existen varias alternativas:

- E/S no bloqueante (encuesta): Añadirles la opción *O_NONBLOCK* con el comando *fcntl()* de manera que todas las llamadas son no bloqueantes.
- E/S conducida por eventos (notificación asíncrona): Usar la opción *O_ASYNC* con el comando *fcntl()* si ya está abierto o con *open()* al abrirlo, de esta manera se envía la señal *SIGIO* cuando el descriptor de fichero está preparado, y se puede usar un *handler* para atender la señal.
- E/S síncrona: Se usa la interfaz *POSIX_AIO* que proporciona una versión de *read()* y *write()* que se llaman *aio_read()* y *aio_write()*, las cuales solicitan la lectura o escritura pero no se bloquean (se realizan en segundo plano).
- Multiplexación de E/S síncrona: Lo proporciona la llamada *select()* que permite monitorizar varios canales de E/S durante un cierto tiempo o de forma indefinida, pasado ese tiempo la llamada nos indica qué canales están preparados para realizar la operación de E/S (lectura o escritura), de esta manera se pueden sincronizar varios canales simultáneamente.

Por lo tanto, se selecciona en cada momento qué descriptor de fichero está listo para realizar la operación de E/S, permitiendo realizarla de manera síncrona.

Para la selección del canal se utiliza la llamada (ejemplo en *man select*):

```
int select(int n, fd_set *Rset, fd_set *Wset, fd_set *Eset, struct
timeval *tout);
```

n: mayor de los descriptors en cualquiera de los tres cjtos, más 1.

Rset: cjto de descriptors de lectura, se comprobará si hay datos disponibles.

Wset: cjto de descriptors de escritura, se comprobará si es posible escribir de forma inmediata.

Eset: cjto de descriptors de excepción, se comprobará si hay alguna condición especial.

tout: tiempo máximo en el que retornará la función, si es 0 retorna inmediatamente, si es NULL se bloquea hasta que se produce un cambio.

Existen una serie de marcos para la manipulación de los conjuntos de descriptors:

- `void FD_ZERO(fd_set *set)`: Inicializa set como cjto vacío.
- `void FD_SET(int fd, fd_set *set)`: Añade fd a set.
- `void FD_CLEAR(int fd, fd_set *set)`: Elimina fd de set.
- `void FD_ISSET(int fd, fd_set *set)`: Comprueba si fd está en set.

La función *select()* devuelve:

- Mismo número de variables que se pasaron como argumento, para indicar en esos cjtos qué descriptors están listos, para revisar si han sufrido algún cambio se usa `FD_ISSET`.
- Número de descriptors que han experimentado un cambio de estado, un nº mayor que 0 significa que algún descriptor está preparado, por lo que se debería buscar con `ISSET`, si devuelve 0 no hay ningún proceso preparado.

Si se produce un error devuelve -1, los cjtos no se modifican y *tout* queda indeterminado. Lo normal es usar *select()* cuando recibimos datos por varios descriptors (si hemos creado varios *FIFOs* y leído alternativamente de ellos) o cuando un proceso recibe datos por distintos *sockets* o por la red y el terminal, etc.

```
...
FD_ZERO(&conjunto);
FD_SET(0, &conjunto);

timeout.tv_sec = 2;
timeout.tv_usec = 0;

cambios = select(1, &conjunto, NULL, NULL, &timeout);
if (cambios == -1)
    perror("select()");
else if (cambios) {
    printf("Datos nuevos.\n");
    scanf("%s", buffer);
    printf("Datos: %s\n", buffer);
} else {
    printf("Ningún dato nuevo en 2 seg.\n");
}
...
```


Tema 5 – Programación con sockets

Def. Socket: Canal de comunicación establecido entre el cliente y el servidor, permite intercambiar datos de forma bidireccional entre ellos. Cada aplicación servidor o cliente se identifica por un nº de puerto.



Se manejan de forma semejante a las tuberías (*pipe* o *FIFO*), diferenciándose en que son mejores en algunos contextos, ya que proporcionan info. bidireccional y además pueden usarse para comunicar procesos en una misma máquina o en distintas. También usan la señal SIGPIPE cuando el otro extremo pierde la conexión, si no se captura ni se bloquea la señal el proceso finaliza como acción por defecto.

Además, proporcionan abstracción para crear extremos de comunicación, que pueden tener distintas semánticas, identificadas por los tipos de sockets:

- **SOCK_STREAM:** Orientado a conexión, representa un flujo de bytes fiable (mensajes duplicados, retransmisión...) y comunicación full-duplex (bidireccional). Además, como el flujo de bytes es continuo este tipo de sockets no proporcionan fronteras entre los mensajes, al igual que un *pipe*, por lo que debe marcarse el principio y el final del mensaje (detectando carácter de fin de línea). TCP.
- **SOCK_DGRAM:** Comunicación basada en datagramas, no hay conexión ni fiabilidad, no se comprueba la correcta recepción del mensaje, comunicación bidireccional, los datagramas tienen una long. máx. fija. UDP.
- **SOCK_RAW:** Permite acceder directamente al protocolo internet.

Protocolos de soporte:

Estos tipos de sockets se implementan sobre unos protocolos de soporte, cada uno se implementa usando una funcionalidad proporcionada por un dominio y un protocolo que se ajusta a la semántica específica del tipo, no todos los *sockets* se soportan en todos los dominios.

Def. Dominios: Familia de protocolos que comparten un esquema de direccionamiento. Ej: AF_INET, AF_INET6, AF_UNIX se suele usar para comunicar dos procesos en una misma máquina.

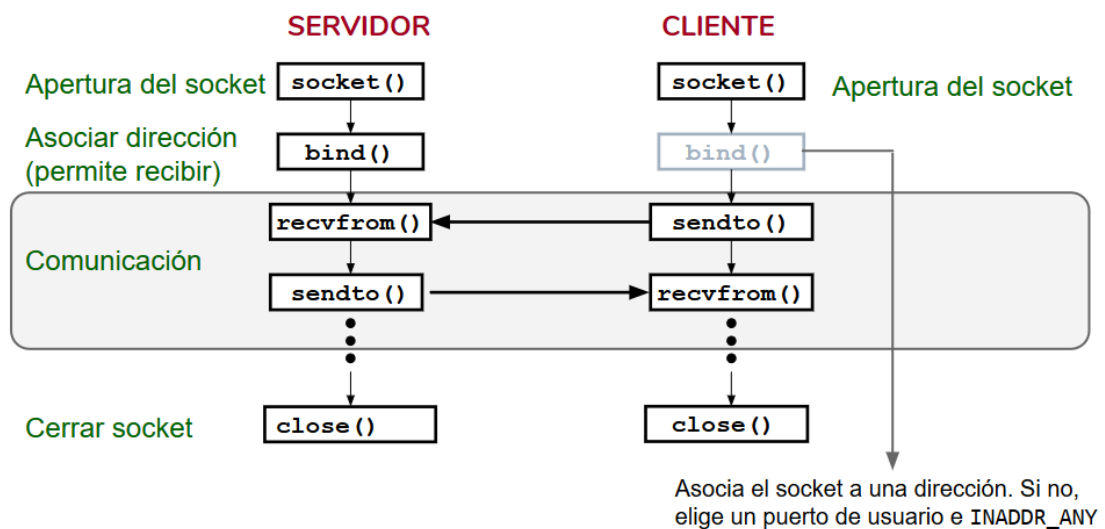
Def. Protocolo: Protocolo concreto del dominio que se usará para el socket, normalmente cuando se crea un socket especificando el dominio y el tipo ya se determina el protocolo, por lo que se suele indicar como 0.

Para crear un socket se utiliza la llamada: `int socket(int domain, int type, int protocol);`

La llamada nos devuelve un descriptor de fichero que nos da acceso al *socket*, suele ser el nº de descriptor de fichero más bajo disponible, podemos leer de él o escribir en él como si fuera un fichero, usando funciones específicas como *receive()* y *send()*.

Patrón de comunicación:

- UDP: Generalmente los servidores UDP no hacen *fork()*, al ser transacciones independientes inician varios procesos que leen del mismo *socket*, usaremos este protocolo si usamos *sockets* SOCK_DGRAM para la familia AF_INET y AF_INET6.



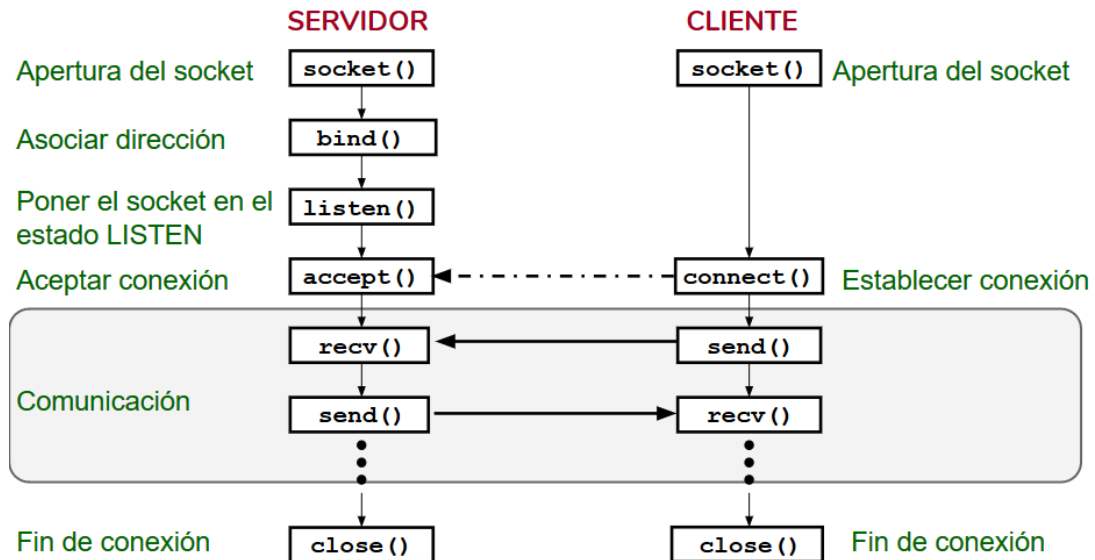
Primero se crea el socket con la llamada *bind()* que vincula el socket a una dir. y a un puerto, es opcional que el cliente haga el *bind()*, si no lo hace se asocia el socket a una dir. .ANY (todo 0's) y se usa un puerto efímero.

Normalmente el cliente inicia la comunicación usando la llamada *sendto()* que es equivalente a la llamada *write()*, la diferencia es que nos permite especificar una dir. destino a la que enviar el mensaje, este se recibe en el servidor por *receivefrom()* que es similar a *receive()*, pero nos devuelve la dir. del otro extremo, para saber con quién nos comunicamos. El servidor puede responder con *sendto()* y el cliente hacer el *receivefrom()*, después cualquiera de los dos puede cerrar el *socket* con *close()*.

- TCP: usaremos este protocolo si usamos *sockets* SOCK_STREAM para la familia AF_INET y AF_INET6, el tratamiento de la conexión se hace en un proceso distinto. En este caso el servidor y el cliente crean el *socket*, tras esto el servidor vincula el socket a una dir. y un puerto local, luego pone el *socket* en estado de escucha llamando a la función *listen()*, a partir de entonces, el *socket* es capaz de aceptar conexiones entrantes, para establecerlas el cliente usa la llamada *connect()* mientras el servidor llama a la función *accept()* para aceptarla, esta llamada

bloquea al servidor a la espera de que se establezca una conexión, y devuelve un nuevo *socket* con el cual se puede realizar la comunicación.

En el caso de los servidores con *socket* *SOCK_STREAM* existen dos tipos de *sockets*, de escucha (*listen*) y conectados (para realizar la comunicación). Normalmente el cliente inicia la conexión con *send()* y el servidor da feedback con *receive()*, no se usan las llamadas *sendto()* ni *receive()* porque ya está establecida la conexión. Una vez termina el intercambio de datos se cierra la conexión con *close()*.



El tratamiento de la conexión, sobre todo en el servidor, normalmente se realiza en procesos distintos, existe un servidor que acepta conexiones y para cada una recibida puede crear procesos que atienden a esas conexiones (*accept-and-fork*), esto se suele usar para conexiones largas como *telnet* o *ssh*. Para conexiones más cortas como *http* se usa el modelo *pre-fork*, en el cual antes de hacer el *accept()* el servidor hace varios *fork()* y todos los procesos creados se bloquean en *accept()*.

Apertura de sockets:

Para realizar la apertura de un socket se utilizan las llamadas:

```
tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

Def. Dirección del socket: Compuesta por una dir. IP de 32 bits de la capa de red y un puerto de 16 bits de la capa de transporte.

```

struct sockaddr_in {
    sa_family_t    sin_family; // fijar a AF_INET
    in_port_t      sin_port;   // puerto
    struct in_addr sin_addr;    // dirección IP
};

struct in_addr {
    uint32_t       s_addr;      // 32 bits dirección IP
};

```

La estructura *sockaddr_in* contiene una variable para el dominio, que se suele fijar a *AF_INET*, otra *in_port_t* para el puerto, donde los puertos < 1024 son privilegiados y solo se usan para procesos con privilegios, además suelen ir asociados a protocolos superiores TCP y UDP, y una estructura *struct in_addr* que es la dir. asociada a la interfaz, puede ser *INADDR_ANY*, *INADDR_BROADCAST* o *INADDR_LOOPBACK*.

Gestión de direcciones y nombres de red:

Las direcciones se generan a través de un nombre de la máquina y un puerto:

```
int getaddrinfo(const char *node, const char *service, const struct
addrinfo *hints, struct addrinfo **res);
```

Esta llamada hace la traducción de un nombre y un puerto a una dirección, se le pasa como argumentos las cadenas *node*, *service* y una serie de criterios de búsqueda *hints*, y nos devuelve un resultado *res* que es una lista enlazada de direcciones a las que nos podemos conectar, sus longitudes, en ocasiones los nombres de las máquina y un puntero *ai_next* que apunta al siguiente elemento de la lista.

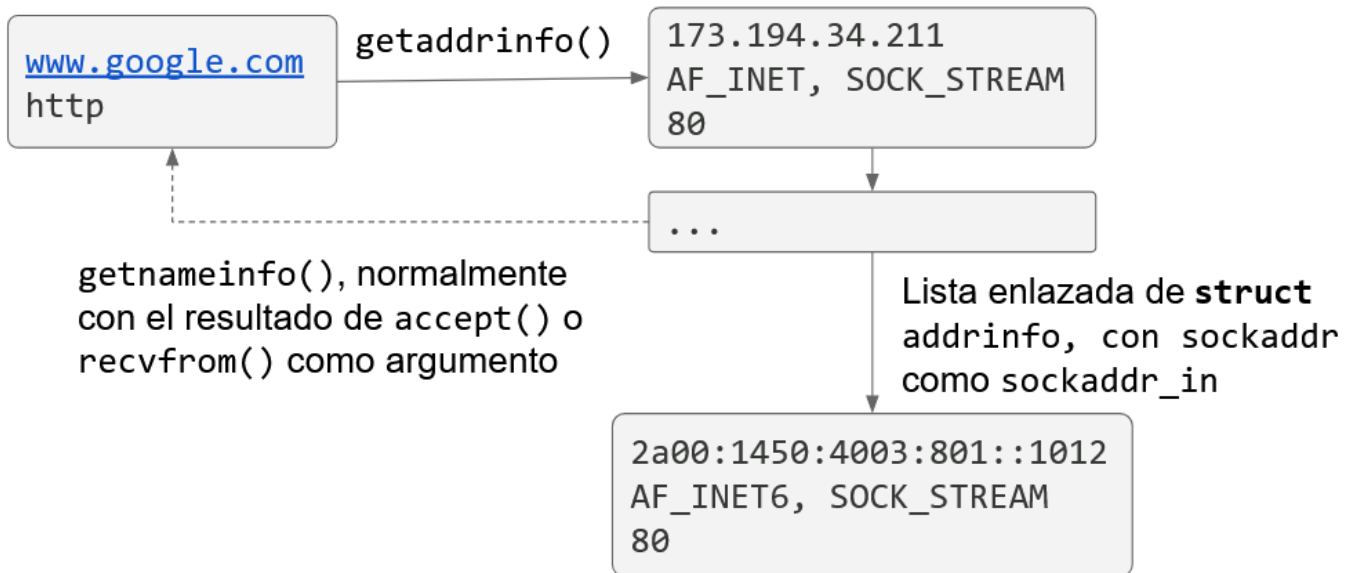
node hace referencia al host, *service* al puerto, puede ser un nombre de servicio, como *http*, un entero o NULL. La función devuelve la info. sobre la dirección en una *struct*:

```
struct addrinfo {
    int          ai_flags;      // Opciones para filtrado (hints)
    int          ai_family;     // Familia ej. AF_INET
    int          ai_socktype;   // ej. SOCK_STREAM
    int          ai_protocol;   // Protocolo, igual que socket(2)
    socklen_t    ai_addrlen;    //
    struct sockaddr *ai_addr;    // Resultado, NULL o 0 en hints
    char         *ai_canonname; // addr, depende de ai_family
    struct addrinfo *ai_next;
};
```

ai_family puede ser *AF_INET*, *AF_INET6* o *AF_UNSPEC* para fijar IPv4, IPv6 o ambos, *ai_socktype* y *ai_protocol* se usan para indicar los tipos preferidos, con *ai_flags* podemos fijar algunos criterios de búsqueda (*hints*), como por ejemplo *AI_PASSIVE*, que indica que los *sockets* pueden ser pasivos (escuchan esperando a recibir un dato o conexión) o activos, en caso de no usar este *flag* (inicia la conexión o comunicación).

Para realizar el proceso inverso se utiliza la llamada *getnameinfo()*, a la cual le pasamos una dirección y la traduce devolviéndonos un nombre y un puerto.

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
socklen_t hostlen, char *serv, socklen_t servlen, int flags)
```



```
struct sockaddr *addr;      /* input */
socklen_t addrlen;         /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo(addr, addrlen, hbuf, sizeof(hbuf), sbuf,
    sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV) == 0)
    printf("host=%s, serv=%s\n", hbuf, sbuf);
```

Para hacer la resolución de una dir. en binario a una dir. en cadena se usa la función *inet_ntop()*, para el proceso inverso (dir. en cadena a binario) se usa *inet_pton()*.

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t
size);
```

```
int inet_pton(int af, const char *src, void *dst);
```

af se refiere a la familia de protocolos, puede ser `AF_INET` o `AF_INET6`.

*const void *src, void *dst*, son las estructuras de la dir. en cada caso.

Las cadenas *char ** son la representación de la dirección.

Gestión de la conexión:

Para la gestión de la conexión se utilizan las llamadas:

```
int bind(int sock, const struct sockaddr *addr, socklen_t addrlen);
int listen(int sock, int backlog);
int accept(int sock, struct sockaddr *addr, socklen_t *addrlen);
int connect(int sock, const struct sockaddr *addr, socklen_t addrlen);
```

bind() sirve para asociar el *socket* a una dirección IP y un puerto (dir. de *socket* local) en la que se escuchará.

listen() pone el *socket* en modo pasivo y le permite escuchar conexiones entrantes.

accept() sirve para aceptar conexiones e incluir los datos del cliente de la conexión, la llamada se queda bloqueada hasta que recibe una conexión (se puede usar *select()*), tras esto devuelve un *socket* nuevo que hace referencia a la conexión establecida, esta llamada es sólo para *sockets* orientados a conexión (SOCK_STREAM). Al ser bloqueante se puede usar *select()*.

connect() es equivalente a *accept()* pero de forma activa en vez de pasiva, define la dirección del servidor.

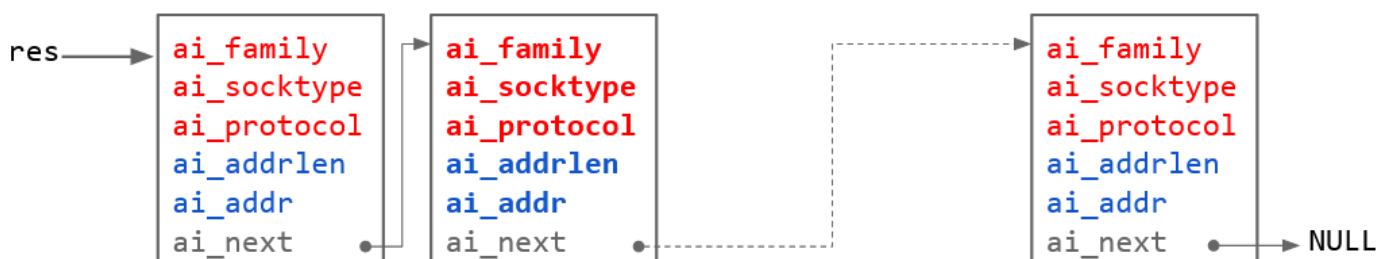
sock es el descriptor creado con la llamada *socket()*

struct sockaddr es un tipo genérico para acomodar la dir. de cada familia.

addrlen es la long. de la dir., que depende del tipo, se recomienda usar *sizeof()*.

backlog es un nº de conexiones pendientes en la cola del *socket*.

Normalmente se usan los campos de la estructura *res* obtenida de la llamada a *getaddrinfo()* para crear el *socket*, asociarlo y realizar la conexión.



```
sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sock, (struct sockaddr *) res->ai_addr, res->ai_addrlen);
connect(sock, (struct sockaddr *) res->ai_addr, res->ai_addrlen);
```

Envío y recepción de información:

Para el envío y recepción de la información se utilizan las llamadas:

```
ssize_t send(int sock, const void *buffer, size_t length, int flags);
ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

send() se puede usar con `SOCK_STREAM` y `SOCK_DGRAM`, con el primero se usa *connect()* para fijar la dir. del servidor, o *sendto()* que es igual pero requiere especificar la dir. en cada llamada, al poder ser bloqueante se puede usar *select()* o *fcntl(sd, F_SETFL, O_NONBLOCK)*.

recv() normalmente se usa con `SOCK_STREAM` pero vale para ambos, recibe hasta *length* bytes en *buffer*.

```
ssize_t sendto(int socket, const void *message, size_t length,
               int flags, const struct sockaddr *dest_addr,
               socklen_t addrlen);
ssize_t recvfrom(int socket, void *buffer, size_t length,
                 int flags, struct sockaddr *src_addr,
                 socklen_t *addrlen);
```

Con `SOCK_DGRAM` lo normal es usar *sendto()* y *recvfrom()*, ya que devuelven un *struct sockaddr* con los datos del otro extremo, además se debe leer el mensaje en una operación *recv()* para no perder datos.

Sockets, opciones:

- IP: Para consultar o fijar opciones en un *socket* IP se usan las llamadas:

```
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
int getsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

- TCP: Se pueden configurar distintas opciones de los *sockets* TCP:
 - El tamaño de los buffers de envío y recepción con `SO_SNDBUF` y `SO_RCVBUF`, esto permite gestionar la ventana de TCP de manera más eficiente.
 - Mensajes urgentes, con los *flags* `URG` en la cabecera TCP, añadiendo la opción `MSG_OOB` en *send()*. Para que el proceso dueño del socket reciba SIGURG se usa *fcntl(socket_fd, F_SETOWN, pid_servidor)*;

Sockets IPv6:

Para crear un *socket* IPv6 se utilizan las llamadas:

```
tcp_socket = socket(AF_INET6, SOCK_STREAM, 0);  
udp_socket = socket(AF_INET6, SOCK_DGRAM, 0);
```

Al igual que en AF_INET, SOCK_STREAM y SOCK_DGRAM están basados en TCP y UDP.

La implementación de IPv6 es casi totalmente compatible con el protocolo IPv4.

```
struct sockaddr_in6 {  
    sa_family_t    sin6_family; // Fijar a AF_INET6  
    in_port_t      sin6_port;   // Número de puerto  
    uint32_t        sin6_flowinfo; // Id del flujo  
    struct in6_addr sin6_addr;   // Dir. IPv6 de 128 bits  
    uint32_t        sin6_scope_id; // sólo si link-local  
};  
  
struct in6_addr {  
    unsigned char   s6_addr[16];  
};
```

Se ha definido una nueva estructura *sockadd_storage* que permite almacenar tanto la *sockaddr_in* como la *sockaddr_in6*.

```
struct sockaddr_storage addr;  
socklen_t addrlen = sizeof(addr);  
accept(sockfd, (struct sockaddr *)&addr, &addrlen);
```

DAVID ZAMORA REY

Notas Prácticas:

Redes:

Práctica 1. Protocolo IPv4. Servicio DHCP

Para eliminar la dirección de red de los interfaces:

```
ip addr flush <ethX>
```

```
ip addr del <dirIp/mask> dev <ethX>
```

Para configurar *dhcp* en el servidor (router) escribir en *etc/dhcp/dhcpd.conf*

```
subnet <dirIP> netmask <mask> {  
    range <dirIPInicioRango> <dirIPFinRango>;  
    option routers <dirIPRouter>;  
    option broadcast-address <dirBroadcast>;  
}
```

Y arrancar el servicio con *service isc-dhcp-server start*.

En el cliente (VM), en cada máquina se puede o bien arrancar el cliente *dhcp*:

```
dhclient -d <ethX>
```

o bien configurar el fichero *etc/network/interfaces* escribiendo:

```
iface <ethX> inet dhcp
```

y escribiendo luego en cada una:

```
ifup dev <ethX>
```

Práctica 2. Conceptos avanzados de TCP

Para arrancar un servidor TCP:

```
nc -l -p <puerto> (con la opción -s se escucha en todas las IP)
```

Para comprobar que el servidor este en estado LISTEN:

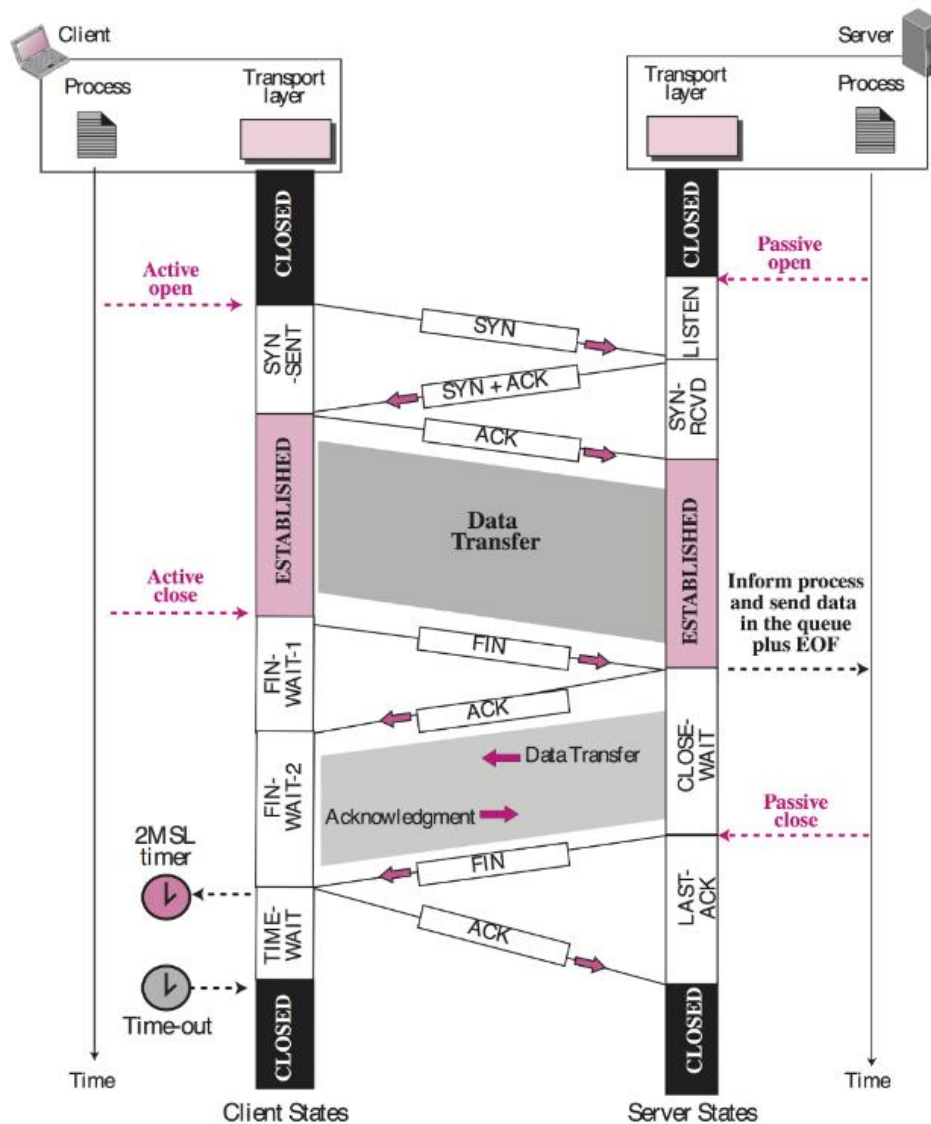
`netstat -tln` (t indica el protocolo TCP, l indica el estado LISTEN y n el número).

Para conectarse al servidor se escribe desde el cliente:

```
nc <dirIPservidor> <puerto>
```

Para ver el temporizador TIMEWAIT, desde donde se cierre la conexión:

```
netstat -o | grep tcp
```



Para fijar una regla en que permita filtrar conexiones en el servidor se usa el comando `iptables`.

En caso de querer dejar al cliente en el estado SYN-SENT:

```
iptables -A OUTPUT -p TCP --tcp-flags ALL SYN,ACK -j DROP
```

```
iptables -A INPUT -p TCP --tcp-flags ALL SYN -j DROP
```