CAPA DE TRANSPORTE

Funciones y servicios de la capa de transporte

- Comunicación extremo a extremo (end-to-end)
- Multiplexación/demultiplexación de aplicaciones:
    - Demultiplexación: Entregar los datos de un segmento de la capa de transporte al socket correcto.
    - Multiplexación: Reunir los fragmentos de datos en el host de origen desde los diferentes sockets, encapsulando cada fragmento de datos con la información de cabecera (que se usará en la demultiplexación) para crear los segmentos y pasarlos a la capa de red.
    -

La operación de multiplexación requiere que los sockets tengan identificadores únicos y que cada segmento tenga campos especiales que indiquen al socket al que tiene que entregarse (campo nº de puerto de origen y campo nº de puerto destino). Los nº de puerto del 0 al 1023 son bien conocidos y reservados para protocolos.

Protocolo UDP (User Datagram Protocol)

Hace casi lo mínimo que un protocolo de transporte debe hacer. Aparte de la multiplexación/demultiplexación no añade nada a IP (funcionalidad best-effort).
Es un servicio no orientado a conexión (no hay handshaking previo al envío de segmentos, por lo que no hay retardos por establecimiento de conexión). Cada TPDU es independiente (stateless). Ofrece un servicio no fiable, por lo que puede haber pérdidas, no hay garantía de entrega ordenada, no hay control de flujo ni de congestión.
DNS usa UDP para evitar los retardos de establecimiento de conexión de TCP. UDP se usa frecuentemente en aplicaciones multimedia, que son tolerantes a fallos y sensibles a retardos. Otros servicios que usan UDP son: RIP (información de encaminamiento) o TFTP (transferencia simple de ficheros).
Cada segmento UDP se encapsula en un datagrama IP y la capa de red hace su mejor esfuerzo por entregarlo al host receptor. Multiplexación/demultiplexación: transportar las TPDU al proceso correcto.

Protocolo TCP (Transmission Control Protocol)

Es un servicio orientado a conexión porque exige un estado común entre el emisor y el receptor (handshaking: Los procesos se envían segmentos preliminares para definir los parámetros de la transferencia de datos). Es un servicio punto a punto, no sirve para comunicaciones multicast, proporciona un servicio de transmisión full-dúplex, garantiza la entrega ordenada de las secuencias de bytes generadas por la aplicación (stream oriented), tiene mecanismo de detección y recuperación de errores (ARQ (Automatic Repeat re-Quest)) con confirmaciones positivas ACK acumulativas y timeouts adaptables, es un servicio fiable con control de congestión y flujo con ventanas deslizantes de tamaño máximo adaptable y usa piggybacking (en vez de enviar ACK en un paquete individual lo incluye en el siguiente paquete).
La conexión TCP se identifica por el puerto e IP origen y puerto e IP destino (y opcionalmente protocolo). Algunos servicios que usan TCP son: HTTP (80), FTP (21), SSH (22), TELNET (23), SMTP (25) y DNS (53) (DNS normalmente es UDP pero también puede usarse TCP).
El intercambio de información tiene 3 fases:

1. **Establecimiento de conexión (three-way-handshake):**
   Sincronizar nº de secuencia y reservar recursos.
   TCP del cliente envía un segmento TCP especial a TCP del servidor y en la cabecera de dicho segmento, el bit SYN se pone a 1. El cliente selecciona un nº de secuencia (al azar) y lo pone en dicho segmento, que se encapsula en un datagrama IP y se envía al servidor.
   El nº de secuencia es un campo de 32 bits que cuenta bytes en módulo 2^32. Normalmente no empieza en 0, sino en un valor ISN (Initial Sequence Number) elegido al azar para evitar confusiones con solicitudes anteriores. El mecanismo de selección de ISN es fiable para proteger coincidencias, pero no ante sabotajes (es fácil averiguar el ISN de una conexión e interceptarla suplantando a uno de los participantes). TCP incrementa el nº de secuencia de cada segmento según los bytes los bytes del segmento anterior excepto cuando los bits SYN y FIN están puestos, que incrementan en 1 el nº de secuencia.
   Cuando el datagrama IP llega al servidor, este extrae el segmento SYN, asigna los buffers y variables TCP a la conexión y envía un segmento de conexión concedida al cliente. En él incluye su nº de secuencia y el segmento se conoce como SYNACK.
   Cuando el cliente recibe el SYNACK asigna buffers y variables a la conexión y envía otro segmento de confirmación poniendo SYN a 0, porque la conexión ya está establecida.

2. **Intercambio de datos (full-duplex)**

3. **Cierre de conexión (liberar recursos)**
   Cualquiera de los dos procesos puede terminar la conexión. Al terminarse, se liberan los buffers y variables de los hosts.
   El que quiere cerrar envía un segmento especial con el bit FIN a 1. Cuando el otro lo recibe devuelve segmento de reconocimiento y su propio segmento de desconexión

con FIN a 1. El primero reconoce este segmento y los recursos de ambos host se liberan.
Para evitar bloqueos por pérdidas, una vez comenzado el procedimiento de cierre se usan timeouts (Maximum Segment Life = 2 min).

Debido a que va sobre IP, no es posible garantizar un establecimiento/cierre fiable de la conexión.

Control de errores y flujo → para mejorar el rendimiento se usan ventanas deslizantes.

Control de errores

Se usa el esquema ARQ con confirmaciones positivas y acumulativas. Hay varios campos involucrados:

- campo secuencia: offset (en bytes) dentro del mensaje.
- Campo acuse: nº de bytes esperado en el receptor.
- Bit A (ACK) del campo de control.
- Campo comprobación: checksum de todo el segmento y uso de pseudo-cabecera TCP.

Hay varios escenarios de error que se solucionan:

- **Caso pérdida de ACK**: un host A envía un segmento a un host B y espera el ACK de este, pero se pierde. En este caso, se produce un suceso de fin de temporización y A retransmite el mismo mensaje. Cuando llega a B, este comprueba que ya lo había recibido y lo descarta, pero envía el ACK
- **Caso timeout prematuro y ACK acumulativo**: A envía 2 segmentos seguidos y B envía 2 ACK, sin embargo, ninguno llega antes del fin de temporización. Al acabarse, A reenvía el primer segmento y reinicia el temporizador, recibiendo ahora los 2 ACK. Si el ACK del segundo segmento llega antes de otro fin, el segundo segmento no es retransmitido.
- **ACK acumulativo evita retransmisión**: A envía 2 segmentos seguidos y el ACK del primero se pierde, pero antes del fin de temporización recibe el del segundo. De esta forma, A sabe que lo ha recibido todo y no retransmite nada.

Generación de ACK por parte del receptor

- Llegada ordenada de segmento, sin discontinuidad y con todo lo anterior ya confirmado → retrasar ACK 500ms y si no llega el siguiente se envía.
- Llegada ordenada de segmento, sin discontinuidad y un ACK retrasado → enviar un ACK acumulativo
- Llegada desordenada de segmento con nº de secuencia mayor que el esperado, discontinuidad detectada → enviar ACK duplicado con el nº de secuencia del siguiente byte esperado.
- Llegada de un segmento que completa una discontinuidad parcial o totalmente → confirmar ACK si el segmento comienza en el extremo inferior de la discontinuidad.

Los timeouts (fin de temporización) deben ser mayores que el tiempo de ida y vuelta (RTT) porque si son demasiado pequeños habrá timeouts prematuros y habrá que hacer

repeticiones innecesarias, pero si son demasiado grandes y un segmento se pierde, este no se retransmitirá rápido y habrá retardos en la transferencia de datos. Para situaciones cambiantes la mejor solución es la adaptable:

- RTTmedido = tiempo desde la emisión de un segmento hasta la recepción del ACK
- RTTnuevo = $(1 - \alpha)$*RTTviejo + α * RTTmedido , $\alpha \in [0,1]$
- Desviación_nueva = $(1-\beta)$*Desviación_vieja + β*|RTTmedido – RTTnuevo|, $\beta \in [0,1]$
- Timeout = RTTnuevo + 4 * Desviación_nueva

Algoritmo de Karn: Actualizar el RTT cuando no haya ambigüedad por ACK repetidos, pero si hay que repetir un segmento incrementar el timeout al final.

$tout_{nuevo} = \gamma . tout_{viejo}$, $\gamma = 2.$

Control de flujo

Procedimiento para evitar que el emisor sature al receptor con el envío de demasiada información y/o demasiado rápido. Es, por tanto, un servicio de adaptación de velocidades. Es un esquema crediticio, el receptor informa al emisor sobre los bytes autorizados a emitir sin esperar respuesta. Se usa el campo ventana:

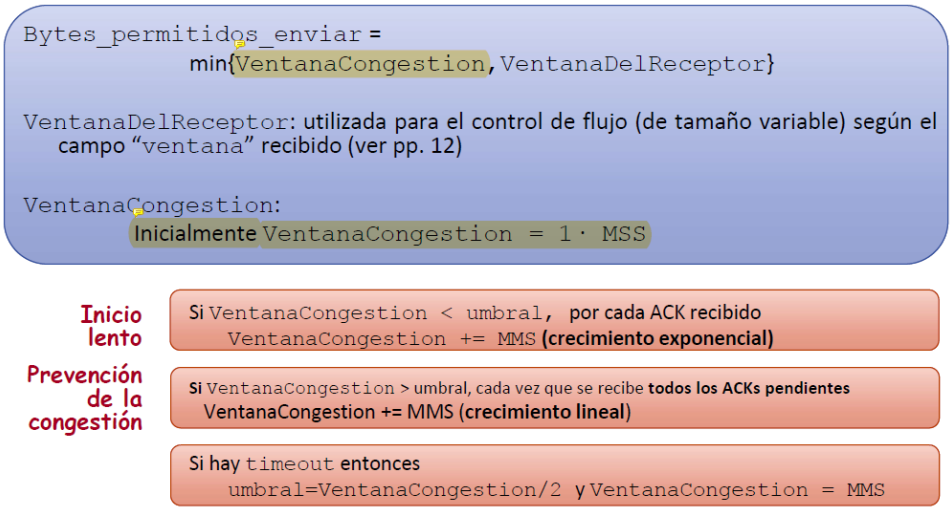
- ventana útil emisor = ventana ofertada receptor - bytes en tránsito.

Un posible problema si los segmentos son muy pequeños es el síndrome de la ventana tonta (el receptor recoge los bytes de uno en uno → el emisor envía datos y su ventana se llena, entonces lee 1 byte y permite que se envíe otro. De esta forma los segmentos tienen muy poca información útil y la eficiencia baja).
Una posible mejora es la ventana optimista. Es posible hacer entregas “no ordenadas” : bit U (UGR), campo puntero.
Solicitar una entrega inmediata a la aplicación: bit P(PSH).

Control de congestión

La congestión es un problema debido a la insuficiencia de recursos (ancho de banda de las líneas como buffer en routers y sistemas finales). Es un problema diferente al control de flujo, ya que involucra a la red y a los sistemas finales. Puede tener naturaleza adelante-atrás y se manifiesta en pérdidas y/o retrasos en los ACK. La solución es limitar en la fuente de forma adaptable el tráfico generado. La limitación se hace por el tamaño de la ventana de emisión (importancia del producto BandWidth-Delay (RTT)).
En el emisor se usan dos ventanas y un umbral (Bytes_permitidos_enviar : min {VentanaCongestion, VentasnaDelReceptor }):

- VentanaCongestion (CongWin): inicialmente vale 1 * MSS (maximun segment size)
- VentasnaDelReceptor: usada para el control del fujo (de tamaño variable) según el campo ventana recibido.



Proceso inicio lento:

- Envias un segmento y paras hasta que llega ACK
- Cuando vuelve ADC CongWin vale 2, por lo que puedes enviar 2 y esperar dichos ACK.
- CongWing crece en tantos ACK te llegan (Si mandas 3 segmentos y vuelven 3 ACK crecería en 3 por lo que podrias enviar 6)

Cuando hay timeout es porque hay congestión, por eso, a partir de ese momento se entra en prevención de congestión y la ventana se aumenta 1 MSS cada RTT.

Extensiones TCP

TCP se define con múltiples sabores: Tahoe, Reno, SACK…
Los diferentes sabores no afectan a la interoperabilidad entre los extremos. Desde cualquier versión de Linux con kernel > 2.6.19 se usa por defecto TCP cubic (la ventana va creciendo cubicamente con el time out).

Adaptaciones de TCP a redes actuales:

- Ventana escalada: Opción TCP en segmentos SYN → hasta 2^14 * 2^16 bytes autorizados.
- Estimación RTT: Opción TCP de sello de tiempo en todos los segmentos.
- PAWS (Protect Against Wrapped Sequence Numbers). Sello de tiempo y rechazo de segmentos duplicados.