3. Capa de Transporte

Se encuentra entre la capa de aplicacion y la capa de red.

Un **protocolo** de la **capa de transporte** provee una **comunicacion logica** entre los procesos de las aplicaciones corriendo en distintos hosts. Se utiliza esto para mandar mensajes entre los hosts.
Por **comunicación lógica** se quiere decir que desde la perspectiva de la aplicación, es como si los hosts estuvieran directamente conectados, cuando en realidad están conectados por muchos routers de por medio.

Los protocolos de la capa de transporte están implementadas en los sistemas terminales, NO en los routers.

3.1 La capa de transporte y sus servicios

Desde el lado **emisor**, la capa de transporte convierte los mensajes de la capa de aplicación en **segmento.** Esto lo hace dividiendo los mensajes de aplicacion en fragmentos más pequeños y agregándoles una cabecer. Luego la capa de transporte pasa el segmento a la capa de red (que encapsula el segmento en un **datagrama**) y lo envía a destino.

Del lado del **receptor** la capa de red extrae el segmento del datagrama y lo pasa a la capa de transporte. La capa de transporte luego procesa el segmento recibido, haciendo que los datos estén disponibles para la aplicación receptora.

3.1.1 Relacion entre la capa de transporte y la capa de red

La **capa de transporte** provee protocolos que generan comunicación lógica entre procesos corriendo en diferentes hosts mientras que un protocolo de **capa de red** provee comunicación lógica entre hosts.

Los servicios que un protocolo de la capa de transporte puede ofrecer son construidos arriba de los servicios que la capa de red ofrece. Aunque hay algunos servicios que puede ofrecer la capa de transporte a pesar de que el protocolo de red no lo ofrezca como el asegurar la entrega de datos fiables.

3.1.2 Capa de transporte en internet

Como se mencionó antes, en internet hay dos protocolos:

UDP	TCP
Sin conexión	Con conexión (3 way handshake)
No fiable (sin garantía de entrega)	Fiable (usa técnicas de control de flujo, números de secuencia, ACK y temporizadores). Garantiza que todos los datos sean entregados y en orden
Comprobación de errores (integridad de los datos)	Comprobación de errores (integridad de los datos)
-	Control de congestión (evita que cualquier conexión TCP inunde con tráfico los enlaces y routers entre los hosts que están comunicándose)

Mux/Demux	Mux/Demux

El **protocolo de capa de red de internet** es el protocolo **IP**. Este proporciona comunicación lógica entre hosts. El modelo de servicio de IP es un servicio **best effort**

La entrega del mejor esfuerzo significa que IP hace lo mejor que puede para entregar los segmentos, pero no garantiza nada. **No garantiza**:

- entrega de segmentos,
- que lleguen ordenados
- la integridad de los segmentos (que no estén corruptos)

Es por estas razones IP es un servicio **no confiable.**

La responsabilidad más fundamental de UDP y TCP es ***ampliar el servicio de entrega de IP entre dos sistemas terminales a un servicio de entrega entre dos procesos que estén ejecutándose en los sistemas terminales.***

Extender este servicio de "host a host" hacía "proceso a proceso" es denominado **multiplexing y demultiplexing** de capa de transporte.

UDP y TCP también proveen chequeos de integridad incluyendo detecciones de errores en los headers de los segmentos.
TCP ademas ofrece un servicio de transferencia de datos fiable y control de congestion. Los mecanismos de control de congestión de tcp evitan que cualquier conexión tcp Inunde con una cantidad de tráfico excesiva los enlaces y routers existentes entre los hosts que están comunicándose. Eso se consigue regulando la velocidad a la que los lados emisores de las conexiones tcp pueden enviar tráfico a la red

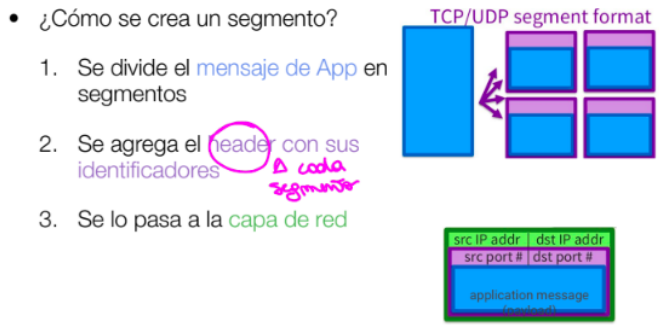
3.2 Multiplexación y Demultiplexacion

Definición: Es la ampliación del servicio de entrega host a host proporcionado por la capa de red a un servicio de entrega proceso a proceso para las aplicaciones que se ejecutan en los hosts.

- **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 para crear los segmentos y pasarlos a la capa de red. Ej: Cuando Ann recopila las cartas de sus hermanos y las entrega al cartero.
- **Demultiplexación**: Entregar los datos contenidos en un segmento de la capa de transporte al socket correcto. Ej: Cuando Bill recibe cartas y las reparte a sus hermanos.

La capa de transporte del host, no entrega los datos directamente a un proceso, sino a un socket intermedio. Un proceso, puede tener uno o más sockets (puertas por las que pasan los datos de la red al proceso).

Puertos: 0-**65535 (2^16-1)**. Del 0 al 1023 son restringidos.



La multiplexacion requiere de dos cosas:

- sockets con identificador unico

- Cada segmento tenga lugares especiales para indicar el **puerto de origen** y **puerto destino.**

Con estos datos, queda claro cómo funciona el demultiplexado, lee campo de puerto de destino y envía el segmento al socket destino.

3.3 Transporte sin conexiones: UDP

UDP proporciona el servicio de multiplexacion y demuliplexacion (que es lo mínimo que debe hacer por ser un protocolo de la capa de transporte). También ofrece un chequeo de errores.

Si se elige UDP sobre TCP es como estar hablandole casi al protocolo IP, ya que udp toma los mensajes de la capa de aplicacion agrega unos capos y lo manda directamente a la capa de red. Es por eso que a veces se le llama datagramas a los segmentos de UDP

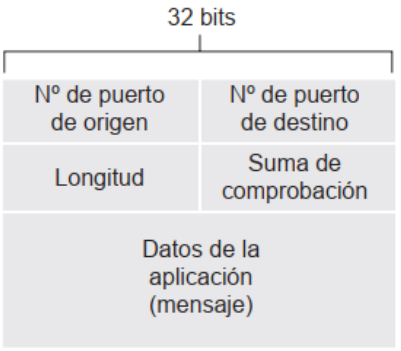
UDP es sin conexion: no establece una fase de conexion entre dos hosts.
Algunas razones para elegir UDP por encima de TCP pueden ser:

- **Menos latencia**: UDP envía el segmento y **no se preocupa por saber si llegó**, por lo que para las aplicaciones real time como STREAMS que no tienen mucho problema con perder datos es util ya que de esa manera no se genera latencia (es decir que NO TIENE QUE ESPERAR A QUE RESPONDA CON EL ACK).
- **Sin conexion**: TCP antes de empezar a transferir datos realiza un handshaking de tres caminos. UDP directamente envía los datos sin preliminares. Es por esto que **UDP no tiene delay por establecer una conexión**; es por esto que DNS usa UDP antes que TCP.
  Nota: hay protocolos de la capa de aplicacion que utilizan UDP y proben un servicio de fiabilidad encima de UDP (QUIC)
- **No hay estado de conexión:** UDP no mantiene un estado de conexion, Por esta razón un servidor dedicado a una aplicación concreta suele **poder soportar más clientes activo**s cuando la aplicación se ejecuta sobre UDP Porque para tcp necesito mantener vivas las conexiones y utiliza un monton de recursos (ej: los buffers y otras variables)
- **Poca sobrecarga debido a que el header es chico:** en UDP son 8 bytes de overhead, en TCP 20 bytes. Entonces el procesamiento/envio es más rapido.

Quienes utiliza UDP:

- ➔ DNS utiliza para evitar demoras establecidas por TCP
- ➔ Se utiliza en la administracion de red (SNMP) ya que las aplicaciones de red a menudo se tienen que ejecutar cuando la red se encuentra en un estado de sobrecarga.
- ➔ Algunas aplicaciones multimedia(telefonia e internet) ya que pueden tolerar un poco de pérdida de paquetes, por lo que la transferencia confiable de datos no es crítica para que la aplicación funcione.
- ➔ Algunas aplicaciones de tiempo real ya que si utilizan TCP van a reaccionar probremente al control de congestion.
- ➔ También puede ser utilizada en una aplicación que requiera de transferencia de datos confiable. Esto puede ser logrado si la confiablilidad es implementada en la aplicación por sí misma. Ej: el protocolo QUIC que utiliza Google Chrome, implementando la confiabilidad en el protocolo de capa de aplicación, y utilizando enlaces UDP.

3.3.1 Estructura de segmentos UDP



El header tiene **4** campos, cada uno de 2 bytes: los puertos, un largo (header + data), checksum, usado por el host que recibe para ver si hubo errores en el segmento. Y EL PAYLOAD

Multiplexacion y Demultiplexación sin conexion (UDP)

Cuando un socket UDP es creado, la capa de transporte asigna automaticamente un número de puerto al socket que actualmente no esté siendo utilizado en ese host por ningun otro puerto.

Un **socket UDP es identificado** por una tupla: (**direccion IP destino**, **puerto destino)**

¿Para qué sirve el puerto de origen? para saber a quién responder.

3.3.2 Por qué UDP provee checksum

La razon para que UDP provea checksum siendo que ya existen en otros protocolos de la capa de enlace, es que **No están garantizadas ni la fiabilidad enlace a enlace, ni la detección de errores durante el almacenamiento en memoria de los routes**, UDP tiene que proporcionar un mecanismo de detección de errores en la capa de transporte terminal a terminal

Es por esto que UDP debe proveer detección de errores en la capa de transporte. Pero no hace nada para recuperar el error.

3.4 Principios de RELIABLE DATA TRANSFER

Esto quiere decir que **ningún bit será corrompido o perdido**, además serán enviados en el **orden correcto**. Los protocolos de transferencia de datos fiables basados en retransmisiones se conocen como protocolos ARQ. Estos protocolos tienen 3 **capacidades** para gestionar la presencia de errores de bit:

- **Detección de errores**: requiere que el emisor envíe al receptor bits adicionales. como en el caso de checksum
- **Realimentacion del receptor**: el receptor mande respuestas del tipo reconocimiento positivo (ACK) o negativo (NAK)
- **Retransmisión**: si se recibe un paquete con errores, este será retransmitido.

Resumen de las variables y los mecanismos del flujo confiable de datos :

- Checksum ó CRC (Cyclical Redundancy Check) : verificación de integridad
- Temporizador (**Timer**): para detectar paquetes perdidos (puede haber duplicados)
- Número de **secuencia**: para mantener un flujo de paquetes y detectar los perdidos
- Acuse de Recibo (**ACK**): para avisar qué paquetes se han recibido
- Acuse Negativo de Recibo (**NACK**): para avisar que el paquete llegó corrupto
- **Ventana** deslizante: para implementar un flujo de datos con alto desempeño

Protocolos**:**

- Sobre un canal totalmente fiable: rdt1.0 (reliable data transf)
- Sobre un canal con errores de bit: **Stop and wait rdt2.0**

Se requieren 3 capacidades adicionales para los errores de bit: detección de errores, realimentación del receptor y retransmisión.
El emisor envía un segmento al receptor y se queda a la espera de recibir un paquete de reconocimiento ACK o NAK. Si recibe un NAK vuelve a retransmitir el paquete. El emisor no envía ningun paquete hasta estar seguro de que el receptor recibio bien el paquete.

- si el paquete ACK o NAK es corrupto? **rdt2.2** : Una solucion es que se reenvie el paquete si recibe un NAK o ACK corrupto, pero esto genera **paquetes duplicados** , donde el que recibe no sabe si el NAK o ACK que envió llegó bien. Entonces se agrega **un sequence number al paquete**, permitiendo que el receptor pueda determinar si es una retransmision. (Además también podemos deshacernos del paquete NAK utilizando solo ACK ya que el emisor se da cuenta de que un paquete llegó con errores si se reciben respuestas ACK duplicadas del mismo paquete)
- Sobre un canal con **pérdidas y errores de bit: rdt3.0**
  Se selecciona un tiempo tal que sea probable que un paquete se haya perdido, aunque no seguro. Si en ese intervalo no se recibe un ack, ocurre un **TIMEOUT** y el paquete se retransmite. Esto da la posibilidad de que existan paquetes duplicados. Este tiempo es mínimo un RTT + Tproc.

3.4.2 Transferencia de datos fiable con procesamiento en cadena

El protocolo de stop and wait tiene una forma de enviar datos bastante ineficiente xq justamente espera a recibir un ack y no hace nada mientras lo espera.

Para solucionar este tema de performance el emisor podria enviar varios paquetes sin tener que esperar a los acknowledgments. Esto se llama **pipelining.**

Para poder utilizarlo:

- Se debe incrementar el rango del número de secuencia dado que **cada paquete** en tránsito debe **tener un único número**
- **Ambos** lados de la comunicación deben tener **buffers** para más de un paquete. Del lado del que envía, para los que fueron **transmitidos pero aguardan respuesta**. Y del lado receptor un Buffer **para paquetes recibidos correctamente.**

Existen dos formas de implementar esto: **Go-Back-N** y **Selective-Repeat**

3.4.3 Go Back N

En este protocolo el emisor puede transmitir multiples paquetes sin tener que esperar a que sean reconocidos pero **está restringido a no tener más de un número N de paquetes no reconocidos en el canal**.

El rango de los números de secuencia permitidos para los paquetes **transmitidos pero todavía no reconocidos** puede visualizarse como una **ventana** de tamaño N sobre el rango de los números de secuencia. Cuando el protocolo opera, esta ventana se desplaza hacia adelante sobre el espacio de los números de secuencia. Por esta razón, N suele denominarse tamaño de ventana y el propio protocolo GBN se dice que es un protocolo de ventana deslizante

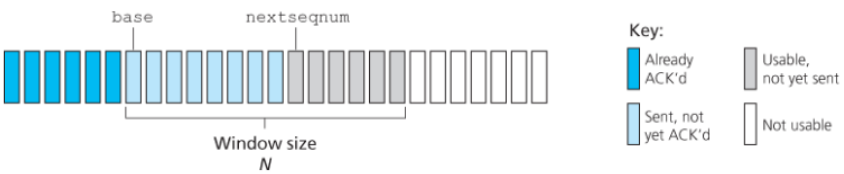


Figure 3.19 Sender's view of sequence numbers in Go-Back-N

N está limitado por el control de flujo

El emisor GoBackN debe responder a 3 tipos de eventos:

- Invocación desde la capa superior: **chequea si la ventana está llena**. Si no lo está, crea el paquete y actualiza variables. Si está llena, devuelve la data a la capa superior, indicando que tendrá que intentar de nuevo
- Recepcion de un mensaje ACK: al recibir una respuesta de un paquete con número de secuencia n, se **asume que los paquetes con número hasta n han sido correctamente recibidos** (Acumulative acknowledgement)
- Timeout event: se usa timer para detectar paquetes perdidos, pero en este caso **reenvia todos los paquetes que no hayan sido reconocidos**. Si se recibe un paquete ACK pero existen más paquetes transmitidos adicionales no reconocidos, entonces se reinicia el temporizador.

Los paquetes recibidos fuera de orden se descartan. El que recibe solo debe mantener es el número de secuencia siguiente paquete. Si no recibe un paquete con ese número, lo descarta.

Problemas:
- Sufre de problema de performance porque cuando se pierde un paquete, este protocolo envía todos los paquetes no reconocidos de nuevo.
- Se descartan muchos paquetes y se reetransmiten muchos.

3.4.4 Selective Repeat

Evita la retrasmicion innecesaria retransmitiendo solo los paquetes que considera que hubo un error. Esto requerirá que el receptor **confirme individualmente** qué paquetes ha recibido correctamente.
Los paquetes no recibidos en orden se almacenarán en el buffer hasta que se reciban los paquetes que faltan

Hay una ventana tanto en el emisor como en el receptor.

Eventos del SR que envía:
- Receive data de la capa superior: chequea el próximo número de secuencia. Si está en la ventana, arma el paquete y envía. Sino igual que GBN
- Timeout: cada paquete tendrá su propio timer lógico (se puede simular con un timer de hardware)
- recibe ACK: marca el paquete como recibido. Si el número de secuencia es igual a send_base, mueve la ventana. Si al moverse la ventana, caen paquetes que no fueron transmitidos, se transmiten

Eventos del SR que recibe:
- El paquete con número de secuencia [rcv_base, rcv_base+N-1] es correctamente recibido: el paquete está en la ventana, se envía entonces un ACK. Si no había sido recibido previamente, se lo pone en un buffer. Si el número de secuencia es igual a la base, el paquete y cualquiera previo en el buffer se envía a la capa superiro, actualizando la ventana
- Paquete con número de secuencia [rcv_base-N, rcv_base-1] es correctamente recibido: se genera un ACK, a pesar de que fue recibido previamente
- Otro rango: se ignora el paquete

Las ventanas del que envía y del que recibe no siempre estarán igual. Trae consecuencias. El ancho de la ventana debe ser menor o igual al rango de los números de secuencia.

3.5 Transporte Orientado a Conexiones: TCP

TCP está **orientado a conexión** porque antes de que un proceso de aplicación pueda comenzar a enviar datos a otro, los dos procesos primero deben realizar un "**handshake**" entre ellos; es decir, **se envían segmentos preliminares para establecer parametros de la transferencia de datos**.

Proporciona un servicio **full-duplex**: si hay una conexion TCP entre proceso A y el proceso B en distintos host, los datos pueden fluir desde el proceso A al proceso B **en el mismo instante** que fluyen del proceso B al proceso A

TCP es casi siempre una conexión punto a punto, es decir, entre un unico emisor y un unico receptor. No es posible con TCP el broadcast

Una vez que se ha establecido una conexión TCP, los dos procesos de aplicación pueden enviarse datos entre sí. El proceso cliente pasa un flujo de datos a través del socket (la puerta del proceso). Una vez que los datos atraviesan la puerta, se encuentran en manos del protocolo TCP que se ejecuta en el cliente. TCP dirige estos datos al buffer de emisión de la conexión, que es uno de los buffers que se definen durante el proceso inicial del acuerdo en tres fases. De vez en cuando, TCP tomará fragmentos de datos del buffer de emisión y los pasa a la capa de red

3.5.1 La conexión TCP

Como parte de la conexión, ambos lados inicializan variables de estado asociadas con la conexión TCP.

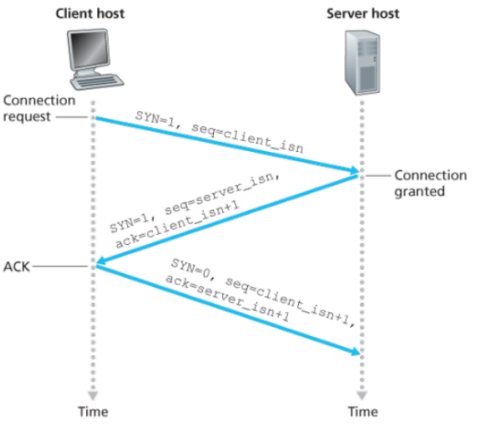
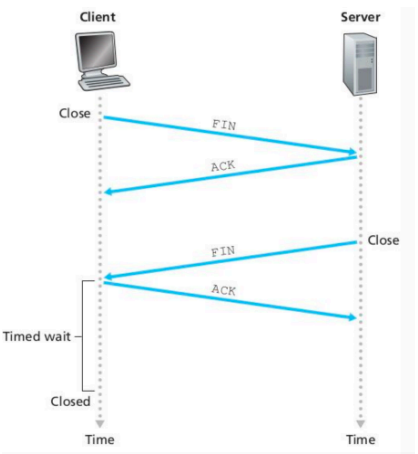


Figure 3.39 TCP three-way handshake: segment exchange

Procedimiento para establecer una conexión TCP:
El proceso de aplicación cliente informa en primer lugar al cliente TCP que desea establecer una conexión con un proceso del servidor.

- Paso 1. El cliente envía un segmento con el bit SYN en 1 y un número de secuencia inicial (random) al servidor. TCP procede a establecer una conexión TCP con el servidor
- Paso 2. El servidor envía un **SYNACK** 1, asigna los **buffers y variables** TCP a la conexión y envía un segmento de conexión concedida
- Paso 3. El cliente recibe el segmento SYNACK, asigna buffers y variables a la conexion y responde con un **ACK** (el cliente ya puede enviar datos)

Una vez que la conexión se establece, los dos procesos pueden enviarse datos entre sí.



Cierre de conexion

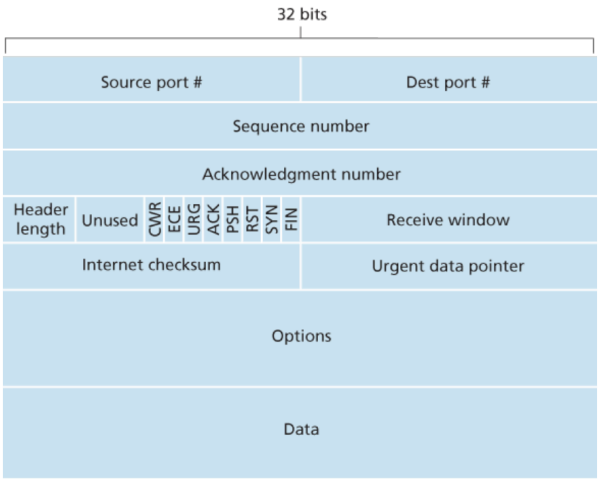
Cuaquiera de los dos procesos pueden terminar la conexion. El proceso de la aplicacion cliente ejecuta el comando cierre, se envía un segmento FIN con el bit fin en 1. El servidor recibe el segmento FIN, devuelve al cliente un segmento de reconocimiento (ACK) y envía su propio segmento de desconexion con el bit FIN en 1. Finalmente los recursos de ambos son liberados .

Ataque SYN

Consiste en que los atacantes envían un gran número de segmentos SYN TCP y no completan el tercer paso del three way handshake. Con esta gran cantidad de segmentos SYN los recursos de la conexion pueden agotarse rapidamente y de esta manera el servidor no puede atender a clientes verdaderos. Para evitar esto existe una defensa llamada **SYN COOKIES.**
Cuando el servidor recibe un segmento SYN crea un número de secuencia TCP inicial por medio de una funcion hash llamado cookie. El servidor manda un paquete SYNACK con el número de secuencia de la cookie. El servidor no puede recordar la cookie.
Si el cliente es legítimo, devolverá un segmento ACK con el número de secuencia cookie +1. De esta forma el servidor puede ejecutar la misma funcion hash utilizando la direccion ip de origen y destino, los puertos y el número secreto. Si el resultado de la funcion + 1 es igual al número de secuencia que mandó el cliente entonces efectivamente es un cliente legítimo y recien ahi el servidor reserva los recursos.

3.5.2 Estructura de segmentos TCP

La cantidad máxima de datos que se pueden poner en un segmento está limitada por el **Maximum Segment Size (MSS)**, el cual queda limitado por el **MTU**: **la longitud de la trama más larga** de la capa de enlace que pueda ser enviado por el host local (Maximum Transmission Unit, MTU), y luego se settea el MSS para asegurar que el segmento TCP (encapsulado), y su cabecera TCP/IP (40 bytes) entren en un único frame de capa de enlace.



Tiene un header y un campo de datos. El MSS determina el límite máximo de este último campo. El header tiene **número de puerto destino y origen**, también tiene un **cheksum**.

El header además tiene:

- campo de **número de secuencia** y campo de **número de acknowledgment**
- **receive window** para control de flujo.
- el campo del **largo del header**
- campo de **opciones**: para negociar MSS o escalar ventana entre end-systems. es de tamaño variable.
- **Campo de flags** de 6 bits.
    - ACK bit indica que el campo de acknowledgment es válido (el segmento tiene un acknowledgment para un segmento que fue recibido correctamente).
    - RST, SYN y FIN para el setup y cierre de la coneccion.
    - CWR y ECE para notificaciones de congestion.
    - PSH indicaría que la data debe ser enviada a la capa de arriba inmediatamente.
    - URG indicaría que la capa de aplicación marcó como urgente. La ubicación del último byte la determina el urgent data pointer field.

En TCP los números de secuencia hacen referencia al flujo de bytes transmitido y no a la serie de segmentos transmitidos. El número de secuencia de un segmento es por tanto el número del primer byte del segmento dentro del flujo de bytes.

El número de reconocimiento que el host A incluye en su segmento es el número de secuencia del siguiente byte que el host A está esperando del host B. Como TCP solo confirma los bytes hasta el

primer byte que falta en el flujo, se dice que TCP proporciona **reconocimiento acumulativo**

Multiplexacion y Demultiplexacion orientado a conexión(TCP)

Los sockets TCP se identifican por una tupla de 4 elementos: (**Direccion IP origen, puerto origen, Dirreccion IP destino, puerto destino**)

Entonces, cuando un segmento TCP llega desde la red a un host, el host usa los 4 valores para dirigir el segmento al socket apropiado (Demultiplexar).

En contraste con UDP, dos segmentos TCP con diferente direccion IP o puerto origen y misma direcion ip y puerto destino son dirigidos a 2 sockets distintos.

3.5.3 Estimación del Round-Trip Time y Timeout

TCP usa un mecanismo de timeout/retransmision para recuperar segmentos perdidos.

Estimar RTT

TCP toma una medida por vez. De esta manera, el RTT de muestra es tomado una vez cada RTT. Nunca se computa un RTT de muestra para un segmento transmitido. TCP a su vez mantiene un promedio de RTT.
Cuando ocurre un timeout, el valor se duplica para evitar otro subsecuente. Cuando un segmento es recibido nuevamente y actualizado el RTT estimado, el timeout es calculado nuevamente.

TIMEOUT= RTTEstimado + 4 RTTDev(Intervalo>= RTTEstimado, o se producirán retransmisiones innecesarias)

3.5.4 Transferencia de datos fiable

TCP crea un servicio de transferencia de datos confiable sobre el servicio best effort de IP.
Este servicio garantiza que los segmentos recibidos no estén corrompidos, no contengan huecos ni duplicados y estén en orden.
El protocolo TCP utiliza un unico temporizador para la implementación de este servicio.

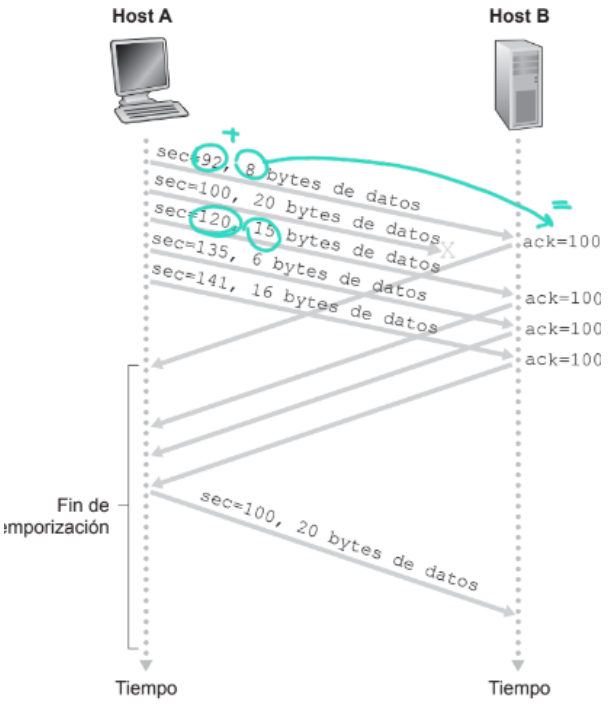
TCP responde a tres eventos principales:

- Cuando TCP **recibe datos de la capa de aplicación**, encapsula los datos en un segmento y pasa el segmento a IP. Cada segmento incluye un número de secuencia que es el número del primer byte de datos del segmento, dentro del flujo de datos. Si el temporizador no está funcionando para algún otro segmento, TCP lo inicia en el momento de pasar el segmento a IP
- **timeout**: retransmite el segmento que no haya sido acknowledged con el seqNum más chico. A continuación reinicia el temporizador
- **recibe ACK**: si es mayor al último que fue reconocido entonces el ACK está confirmando uno o más paquetes no reconocidos anteriormente y actualiza su variable BaseEmisión. Si actualmente aun existen segmentos no reconocidos, reinicia el temporizador

Cada vez que TCP retransmite por un timeout, resetea el timer al doble del valor que tenía, así sucesivamente hasta que recibe data de arriba o recibe un ACK y el timeInterval se calcula nuevamente. Este mecanismo sirve para control de congestion, retransmitiendo entre intervalos más

largos de tiempo.

Retransmision Rapida:



Un ACK duplicado es un ACK que vuelve a reconocer un segmento para el que el emisor ya ha recibido un reconocimiento anterior.

Cuando un receptor TCP recibe un segmento con un número de secuencia que es mayor que el siguiente número de secuencia en orden esperado, detecta que falta un segmento. Dado que TCP no utiliza paquetes NAK, el receptor no puede devolver al emisor un mensaje de reconocimiento negativo explícito. En su lugar, genera un ACK duplicado al último byte de datos recibido. Si el emisor TCP recibe tres ACK duplicados para los mismos datos, toma esto como una indicación de que el segmento que sigue al segmento que ha sido reconocido tres veces se ha perdido. En el caso de que se reciban tres ACK duplicados, el emisor TCP realiza una **retransmisión rápida** reenviando el segmento que falta antes de que caduque el temporizador de dicho segmento.

GBN o SR

Podemos concluir que TCP es un hibrido entre Go Back N y SR.
Los reconocimientos de TCP son acumulativos y los segmentos correctamente recibidos pero no en orden son reconocidos. El emisor solo debe mantener el número de secuencia del menor paquete no reconocido y el número de secuencia del siguiente paquete que va a mandar. Se parece a GBN pero permite almacenar en sus buffers paquetes recibidos en distinto orden. También se parece a Selective Repeat porque no retransmite los segmentos que ya fueron reconocidos de forma selectiva por el receptor.

3.5.5 Control de flujo

TCP proporciona un servicio de control de flujo a sus aplicaciones para eliminar la posibilidad de que el emisor desborde el buffer del receptor.
El control de flujo **adapta la velocidad a la que el emisor está transmitiendo frente a la velocidad a la que la aplicación receptora está leyendo**.

El control de flujo se hace haciendo que **el emisor mantenga una variable Receive window**: le da una idea de cuánto **espacio libre** hay disponible en el buffer del receptor.
Puesto que TCP es una conexión full-duplex, el emisor de cada lado de la conexión mantiene una ventana de recepción diferente.

El tamaño del buffer siempre va a ser mayor al Num último byte leído - el Num último byte que fue puesto en el buffer. LastByteRcvd - LastByteRead <= RCVBuffer

La ventana de recepción, llamada RWDN, se hace igual a la cantidad de espacio libre disponible en el buffer: ReciveWinDoN = RcvBuffer – [LastByteRcvd–LastByteRead]
Dado que el espacio libre varía con el tiempo, rwnd es una variable dinámica.

Un host le comunica al otro cuánto espacio le queda, colocando el valor de la variable en el campo de la ventana **en cada segmento que envía**.

Si la ventanta llega a cero, pero el que enviaba no tiene nada que enviar, entonces va a quedar bloqueado. TCP obliga al que envía a enviar constantemente segmentos con UN byte de data cuando la ventana tiene valor cero. Estos segmentos son acknowledged por el que recibe y en algún momento se liberará la ventana

(PD: UDP no proporciona ningún mecanismo de control de flujo => los segmentos pueden perderse en el receptor a causa del desbordamiento del buffer)

3.6 Principios del control de congestión

Causas y consecuencias de una red congestionada:

- Los **grandes retardos de cola** son experimentados cuando la tasa de llegada de los paquetes se aproxima a la capacidad del enlace.
- El emisor tiene que realizar **retransmisiones** para poder compensar los paquetes descartados (perdidos) a causa de un desbordamiento de buffer.
- Las retransmisiones innecesarias del emisor causadas por retardos largos pueden llevar a que un router **utilice el ancho de band**a del enlace para reenviar copias **innecesarias** de un paquete.
- Cuando un paquete se **descarta** a lo largo de una ruta, la capacidad de transmisión empleada en cada uno de los enlaces anteriores para encaminar dicho paquete hasta el punto en el que se ha descartado termina por **desperdiciarse**.

Las pérdidas de paquetes son normalmente el resultado de un desbordamiento de los buffers de los routers a medida que la red se va congestinando.

3.6.2 Métodos para controlar la congestión

- Control de congestion terminal a terminal:
    - TCP tiene que aplicar este metodo debido a que IP no proporciona ningun mecanismo para la congestion de la red
    - La pérdida de segmentos TCP (dada por RTO o ACK duplicados), se toma como indicador de que hay congestion en la red por lo que TCP reduce el tamañano de su ventana.
- Control de congestion asistido por la red:
    - Los routers proporcionan una realimentacion explícita al emisor y/o receptor informando del estado de congestion de la red. Esto se puede hacer marcando un campo de un paquete para indicar que hay congestion

3.7 Mecanismo de control de congesion de TCP

TCP utiliza un control de congestion terminal a terminal.
Su metodo consiste en que **cada emisor limite la velocidad a la que transmite** el trafico a traves de su conexion **en funcion de la congestion de red percibida**.

- Aumenta su velocidad si apenas hay congestion
- Disminuye su velocidad si hay congestion

Esto lo logra por medio de la **ventana de congestion**. La cant de datos no reconocidos no puede

exceder el min de (rcwn, winCongestion). Esto **limita la cantidad de datos no reconocidos** por el emisor y por lo tanto limita de manera indirecta la velocidad de transmisión del emisor

Algortimo de control de congestion TCP

¿Cómo percibe TCP que la red está congestionada? Lo que pasa cuando se congestiona es que se pierden paquetes en el camino, entonces un emisor TCP percibe que **existe congestión** con el **Timeout** (fin de temporización) o a la recepción de **tres paquetes ACK duplicados**.

A su vez, **si no hay congestión** lo va a percibir al recibir todos los ACK, y ante esto puede **incrementar la congestion window**. Como TCP utiliza los ACK para triggerear el aumento o decremento de la ventana, se dice que es **self-clocking**.

Este algortimo tiene 3 principales componentes:

- Slow start
- Congestion avoidance
- fast recovery

Slow Start

cwnd (ventana de congestion) : es la máxima cantidad de bytes que puede haber en vuelo
El valor de la ventana de congestion se inicializa normalmente a 1 MSS, y se va incrementado exponencialmente es decir

$$cwnd(n+1) = cwnd(n) + \#(ACK)$$

Si se produce una pérdida de paquetes indicado por el fin de un temporizador (**RTO**):

se reduce el ssthresh a la mitad
el emisor TCP establece el valor de la ventana en 1
volvemos a Slow start

- ssthresh = cwnd(n) / 2
- cwnd(n+1) = 1 (1 es LW [loss window])
- Empieza slow start de nuevo

Cuando la VENTANA = SSTRHESH, se termina la etapa de slow start y se pasa a la congestion avoidance

Congestion Avoidance

Ahora el crecimiento de la ventana va a ser lineal: $cwnd(n+1) = cwnd(n) + \#(ACK)/cwnd(n)$

Si se da el lugar de un fin de temporizacion sucede:

- ssthresh = cwnd(n) / 2
- cwnd(n+1) = 1 (1 es LW [loss window])

Tanto en slow start como en congestion avoidance, si llegan 3 ACK duplicados implicando una pérdida de paquete se pasa a la etapa de Fast Retrasmit.

Recuperacion Rapida (Fast Retrasmit)

El valor de cwnd se incrementa en 1 MSS por cada ACK duplicado recibido correspondiente al segmento que falta.
Cuando llega un ACK para el segmento que falta, TCP entra de nuevo en el estado de evitación de la congestión después de disminuir el valor de cwnd.

¿Qué pasaría si se pierde **un** paquete?

TCP Tahoe:

- establece el tamaño de la ventana de congestión en 1 MSS [cwnd(n+1) = 1]

- entra en el estado de **slow start**
- ssthresh= cwnd(n-1) /2

TCP Reno:

- se sigue con la fase de congestion avoidance (recuperación rápida)
- donde el cwnd(n+1) = cwnd(n) / 2
- sshtresh = cwnd(n) / 2