Unidades 7 y 8: Nivel de Transporte y Congestión

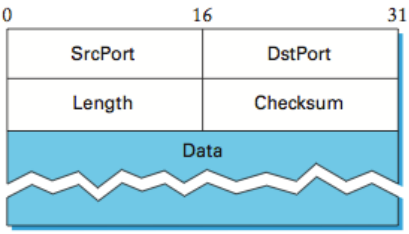
Hasta ahora la capa de red provee la funcionalidad de conectar dos host a través de una red. Debido a la dificultad del problema, provee un servicio best effort.

El objetivo final es comunicar procesos que requieren ciertas garantías sobre el canal de comunicación. El nivel de transporte se encargará de proveer la **demultiplexación a nivel proceso** y un conjunto de **garantías** útiles para las aplicaciones.

En Internet se utilizan básicamente dos protocolos de nivel de transporte: **UDP** (sin conexión y no confiable) y **TCP** (con conexión y confiable).

UDP

UDP (User Datagram Protocol) es un protocolo de transporte que ***sólo provee demultiplexación entre procesos*** y ***chequeo de integridad***, sin agregar garantías sobre la trasmisión de paquetes.



- Para identificar procesos podría usarse el *pid* asignado por el sistema operativo, haría el protocolo muy dependiente de cómo el SO asigna números de proceso.
- En vez de esto, se utilizan direcciones más abstractas, **puertos** que funcionan a modo de casillas.
- Un proceso envía datos a un puerto en el host de destino, y el proceso receptor recibirá la información de ese puerto.
- La clave de demultiplexación es, entonces, el par <host, port>

Alternativas para conocer el puerto destino:

- Convención de números de puertos (ejemplo: 80 para web)
- Tener un puerto conocido para negociar otro puerto
- Port Mapper: servicio escuchando en un único puerto conocido, con protocolo para averiguar el puerto donde reside el servicio buscado

El chequeo de integridad es un checksum sobre:

- Header UDP
- Contenido del mensaje
- Pseudoheader (campos *protocol version*, *source address* y *destination address* del header IP).

Observar cómo aquí se rompe la capa de abstracción, acoplando UDP a IP. El chequeo de los campos de IP es para descartar mensajes dirigidos a otro destinatario que puedan haber llegado erróneamente.

TCP

TCP (Transmission Control Protocol) es un protocolo de nivel de transporte con las siguientes características:

- Garantiza la transmisión ordenada de un stream de bytes
- Provee demultiplexación (como UDP)
- Orientado a conexión full duplex
- Mecanismo de **flow control** (para que no se transmita más de lo que puede procesar el receptor)
- Mecanismo de **congestion control** (para que no se sature la red)

Características de la comunicación entre procesos

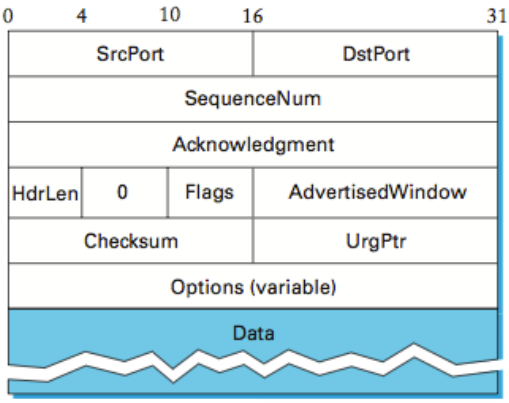
TCP debe lidiar con algunos de los problemas que también se atacaron en la capa física (control de errores, secuenciamiento, control de flujo). Sin embargo, en esta capa se deben tener en cuenta ciertas consideraciones:

- **Establecimiento de conexiones**: Una conexión TCP puede realizarse con cualquier otro host de Internet, a diferencia de contar con un canal fijo (como puede ser un cable).
- **Timeout adaptativo**: El delay de la transmisión es desconocido y variable, a diferencia de los físicos. Es más complicado saber cuándo declarar timeout para retransmitir un paquete.
- **Almacenamiento de la red**: Un paquete puede quedar almacenado en la red y llegar a destino varios segundos más tarde.
- **Necesidad de flow control**: No se tiene garantías de la cantidad de recursos que dispone el receptor para la conexión, por lo cuál el protocolo se debe encargar de averiguar cuánto se puede enviar en un determinado momento.
- **Necesidad de congestion control**: No se sabe conoce la capacidad de la red, por lo que el protocolo debe saber cómo prevenir que la red se inunde de mensajes que no llega a transmitir.

Formato de segmento

Observaciones:

- TCP ofrece una interfaz de byte stream, pero lo que efectivamente se envían son segmentos.
- El host emisor guarda en un buffer una cantidad suficiente de bytes que luego son enviados juntos.
- El receptor, a su vez, guarda en buffers los segmentos recibidos que se van vaciando a medida que el proceso lee.



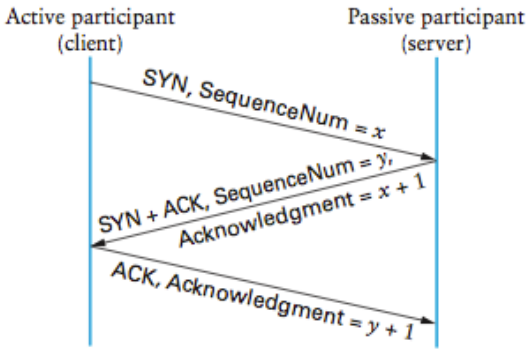
- SrcPort, DstPort: demultiplexación
- SequenceNum, Acknowledgement, AdvertisedWindow: usados para sliding window
- HdrLen: longitud del header (variable)
- Flags: información de control
- UrgPtr: indica dónde empiezan los datos no urgentes (si está habilitado el flag URG)
- Checksum: control de integridad

Conexión

El setup de la conexión se inicia cuando el caller hace un **active open** a un callee que haya hecho un **pasive open** (osea, que esté escuchando en ese puerto). Una vez establecida la conexión, puede ocurrir que sólo un extremo la cierre y que el otro siga mandando datos.

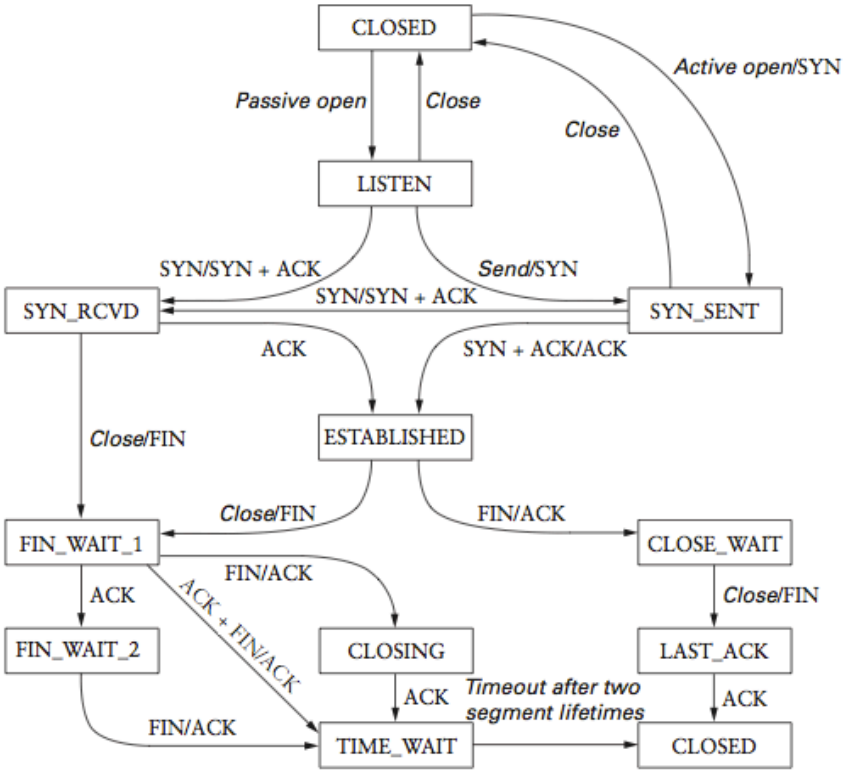
Three way handshake

TCP requiere que se utilicen números de secuencia aleatorios para minimizar el riesgo de que interfieran segmentos de otra encarnación de la conexión (una conexión previa con mismo par de puertos entre los dos hosts).



Al iniciar la conexión, ambos lados realizan un intercambio conocido como **Three way handshake** para ponerse de acuerdo en estos parámetros. Cada mensaje del intercambio tiene un timer asociado, y es retransmitido en caso de que no llegue un acknowledge a tiempo.

Estados de la conexión



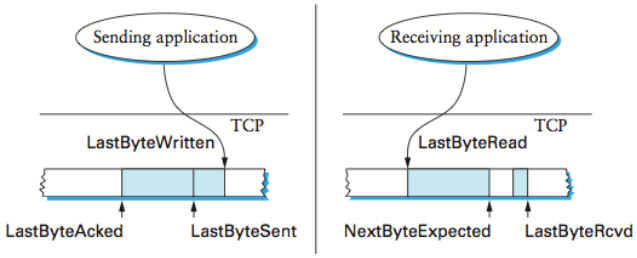
Notar:

- Si se puerde el último ACK del three way handshake no pasa nada. El servidor quedará en estado SYN_RCVD y el cliente en ESTABLISHED (osea que ya puede enviar). En el primer segmento de datos que envíe el cliente habrá también un ACK, lo que hará que el servidor pase a ESTABLISHED.
- La mayoría de los mensajes tiene un timeout y se retransmite si no llega a tiempo el ACK. Si tras varios intentos no llega, se corta la conexión.
- Ambos extremos deben cerrar su lado de la conexión (en caso contrario pueden seguir enviando).

Sliding window en TCP

Garantía de entrega y orden de paquetes

Se logra de forma similar a como ocurre en la capa física, teniendo encuenta también que se bufferean los datos generados y no enviados (en el lado del emisor) y los recibidos en orden pero no aún leídos (en el lado del receptor.)



Control de flujo

Sliding window también se utiliza para evitar que el sender envíe más datos de lo que el receiver puede guardar en buffer. Sin embargo, el tamaño de la ventana no depende solamente del tamaño del buffer:

- Si la aplicación no consume, se debe seguir guardando en buffer.
- Si llega un segmento pero no están todos los anteriores, no se envia ACK y se guarda en buffer.

Esto hace que el temaño de la ventana (es decir, la cantidad de bytes que puede haber enviados sin haber recibido ACK) no sea fijo sino que dependa del estado del buffer, el orden de los datos que llegan y el ritmo al cual la aplicación consume.

Lo que se hace es que, con cada ACK, el receiver comunica al sender el tamaño de ventana actual. En caso de publicarse una ventana de tamaño 0, el sender envía períodicamente segmentos de 1 byte esperando que alguno sea aceptado.

Invariantes

El tamaño del buffer del receptor limita la cantidad de bytes no leídos almacenados:

$$LastByteRcvd - LastByteRead \leq MaxRcvBuffer$$

No puede haber más bytes sin ACK de lo que indica la ventana publicada:

$$LastByteSent - LastByteAcked \leq AdvertisedWindow$$

La aplicación que envía no puede escribir más de lo que soporta su buffer (en caso de querer escribir más, se bloquea):

$$LastByteWritten - LastByteAcked \leq MaxSendBuffer$$

Posibles problemas

Wraparound del SequenceNumber

Dado que el campo SequenceNumber sólo tiene 32 bits, con altas tasas de transferencia podría ocurrir un wraparound de este campo en un tiempo menor al máximo permitido para que un paquete TCP expire. Sería posible, entonces, que llegue un paquete viejo con un sequence number de una "vuelta anterior" y se confunda con los transmitidos actualmente.

Para solucionar esto, se envía un timestamp en la sección opcional del header y efectivamente se chequean con los 32 bits del sequence number y los 32 del timestamp. A fines de ordenar paquetes, sólo se utiliza el número de secuencia.

AdvertisedWindow muy pequeña

Dado que el campo AdvertisedWindow tiene sólo 16 bits, el máximo tamaño de ventana publicable es 64KB. Esto puede resultar muy poco para mantener lleno un enlace de alta capacidad, provocando subutilización de la infraestructura.

Para solucionar esto, se utiliza otro campo opcional en el header que funciona a modo de factor de escala del AdvertisedWindow. El mismo indica que el AdvertisedWindow no contará cantidad de bytes sino porciones

más grandes.

Retransmisión adaptativa

Para garantizar el envío de datos, TCP reenvía segmentos cuando considera que pueden no haber llegado a destino. La variabilidad de los RTT entre las redes y a través del tiempo hace que no se pueda definir un timeout fijo a partir del cuál reenviar. Por esto, TCP necesita algún criterio adaptativo que defina en un determinado instante cuál es el timeout a asignarle a un segmento transmitido.

Algoritmo original

El algoritmo original para calcular el timeout funcionaba así:

- Se lee un timestamp del reloj del sistema operativo y se envía dentro de la sección opcional del header.
- Al enviar el ACK, el receptor copia el mismo timestamp
- Al llegar el ACK del mismo, se calculaba un SampleRTT con el timestamp de emisión y un nuevo timestamp.
- Se calcula el nuevo EstimatedRTT como un promedio ponderado con el estimado anterior:

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$

$$Timeout = 2 \times EstimatedRTT$$

Algoritmo de Karn/Patridge

Dado un ACK, es imposible saber a qué transmisión de ese dato corresponde. Esto puede hacer que tomemos SampleRTT no representativos. Para solucionar esto, el algoritmo de Karn/Patridge propone tomar samples sólamente de paquetes que no requirieron retransmisión.

Además de esto se utiliza exponential backoff para retransmisiones, con el objetivo de evitar sobrecongestionar la red.

Algoritmo de Jacobson/Karels

El algoritmo anterior no soluciona por completo el problema de congestión. Entre otras mejoras para reducir la congestión, Jacobson y Karels propusieron un algoritmo para calcular el timeout.

Este algoritmo tiene en cuenta la variación de RTT.

$$Difference = SampleRTT - EstimatedRTT$$

$$EstimatedRTT = EstimatedRTT + (\delta \times Difference)$$

$$Deviation = Deviation + \delta(|Difference| - Deviation)$$

$$TimeOut = \mu \times EstimatedRTT + \phi \times Deviation$$

Con valores cercanos a $\mu = 1, \phi = 4$. Con varianza pequeña, el timeout es cercano al EstimatedRTT. Varianzas mayores hacen que el término de desvío domine el timeout.

Delimitación de segmentos

Queda el problema de decidir en qué momento dejar de bufferear y enviar un segmento., Asumiendo una ventana arbitrariamente grande, tres eventos pueden hacer que esto ocurra:

- Se alcanza el **MSS** (Maximum Segment Size, ~ MTU inmediato).
- Push por parte de la aplicación.
- Timer (ver adelante)

Silly window sindrome

El **silly window sindrome** es un problema que ocurre cuando la capacidad de recepción es muy pequeña y el sender es muy agresivo. Es decir, intenta enviar constantemente apenas se abre una pequeña ventana de transmisión.

Cuando esto ocurre, cualquier espacio de ventana pequeño que se utilice será ocupado enseguida. Luego, el receptor volverá a publicar un pequeño espacio de ventana, que nunca será combinado para lograr un tamaño de ventana mayor.

Hay dos soluciones parciales a este problema:

- Haciendo que el receptor aguarde a que haya una capacidad considerable (~MSS) para abrir la ventana luego de que se cierre completamente.
- En caso de que el emisor genere muchos segmentos pequeños (por ejemplo, por uso de *push*) se pueden retrasar los ACK. Es decir, esperar varios segmentos y enviar un ACK por todos ellos al final.

Sin embargo, no se puede retrasar indefinidamente el envío del emisor sólo para evitar este problema. Hace falta que el mismo emisor se limite a no enviar paquetes pequeños cuando esto es posible.

Algoritmo de Nagle

Si bien enviar paquetes muy pequeños puede hacernos caer en el silly window sindrome, esperar mucho para enviar puede perjudicar aplicaciones interactivas. Esto se podría solucionar con un *timer*.

El algoritmo de Nagle describe un mecanismo por el cual se utilizan ACKs que llegan como timers que disparan el envío de segmentos con todos los datos que se hayan acumulado hasta el momento.

Idea:

- Se puede enviar si hay un segmento completo y la ventana lo permite.
- Si no hay datos en tránsito, se puede enviar segmentos pequeños.
- Si hay datos en tránsito, se debe esperar a que lleguen los ACK para enviar.

Esto hace que la misma red actue como reloj: el buffer se vaciaría en intervalos de tiempo 2*RTT (con lo que se haya acumulado hasta el momento).

Congestión

El problema

El tráfico que fluye en una red requiere que se destinen recursos en los dispositivos a lo largo de toda la misma. En cada paso de transmisión, cada paquete compite con otros (posiblemente de otros hosts) por ancho de banda en el enlace siguiente o por espacio de buffer en caso de ser encolado. Cuando un

router/switch no está en condiciones de seguir almacenando paquetes debe descartarlos y en este caso se dice que está **congestionado**.

A diferencia del control de flujo, en donde se busca evitar que el emisor transmita más de lo que el receptor puede leer/almacenar, en control de congestión el recurso limitante es la capacidad de la red.

Taxonomía de los métodos

Los métodos para lidiar con congestión se pueden clasificar de la siguiente forma:

- Lazo abierto: no utilizan feedback de la red
- Lazo cerrado: utilizan feedback de la red.
    - Feedback explícito: la red envía señales a los host (implementación más costosa)
    - Feedback implícito: cambia el comportamiento de la red y el host infiere qué sucede

Flujos

En redes con conexión se puede optar reservar recursos en la etapa de setup. Esto lleva a subutilización de la infraestructura.

Tal como ocurre en IP, se considera que a nivel de red no se tienen conexiones. Sin embargo, si bien los datagramas pueden ser ruteados por canales distintos, en la práctica ocurre que muchas veces hay **flujos** de paquetes que atraviesan el mismo conjunto de routers. Distinguir estos flujos permite mantener cierto **soft state** para tomar decisiones en los mecanismos de control.

Encolamiento

El algoritmo de encolamiento elegido para los switches de la red es fundamental ya que afecta directamente el delay. Puede hacer que un paquete sea descartado, o determinar cuánto tarda en transmitirse por el enlace.

FIFO es la estrategia más utilizada, lo que implica que se delega toda la responsabilidad del control de congestión a los extremos: esta estrategia no tiene en cuenta el flujo al que pertenece un paquete ni ningún parámetro sobre la congestión de la red más que la capacidad actual de su buffer.

Delegar toda la responsabilidad a los extremos tiene un problema grande: por más de que haya una estrategia que supera el problema de congestión (por ejemplo, la de TCP) no nos podemos asegurar que los hosts efectivamente la utilicen.

FQ (fair queueing) es una alternativa que básicamente mantiene colas distintas por cada flujo y las atiende utilizando *round robin*. En la implementación, para mantener el *fairness* requiere un mecanismo para aproximar el round robin bit a bit, ya que el tamaño de paquetes no es fijo.

TCP Congestion Control

Características generales:

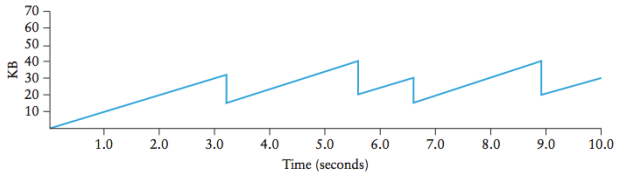
- Se realiza desde los hosts, enviando paquetes sin reserva y reaccionando al comportamiento de la red.
- No se asume nada especial sobre la política de encolamiento de routers.
- Esencialmente, un host evalúa cuántos paquetes puede enviar, e interpreta cada ACK como un

permiso para enviar otro en reemplazo del que acaba de salir de la red.

Hay tres mecanismos básicos que definen el control de congestión de TCP:

- Additive Increase / Multiplicative Decrease
- Slow Start
- Fast Retransmit - Fast Recovery

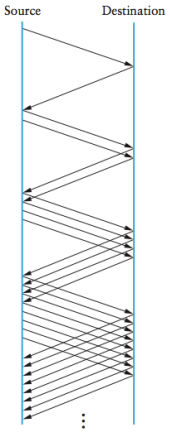
Additive Increase / Multiplicative Decrease



- Se define nueva variable **CongestionWindow** (contraparte de AdvertisedWindow) y se modifica funcionamiento de TCP para que tome en cuenta el mínimo entre CongestionWindow y AdvertisedWindow.
- El valor de CongestionWindow se reduce cuando se detecta una suba en la congestión, y se aumenta cuando baja.
- Se toman los timeouts como indicadores de congestión (despreciando los errores de transmisión).
- Cada vez que ocure un timeout, se baja CongestionWindow a la mitad (con un mínimo en el MSS).
- Cada vez que llega un ACK, la ventana se incrementa por una fración del MSS.
- La idea es que tener una CongestionWindow demasiado grande cuando hay congestión produce más congestión, por lo cual es necesario salir rápido de esa situación.

Slow Start

El comportamiento de AIMD sirve cuando se está operando cerca de los límites permitidos por la red, pero es muy conservador arrancar todas las conexiones transmitiendo con ventana mínima y subiendo linealmente. **Slow Start** hace este incremento exponencial.



- Se incremente en uno el CongestionWindow por cada ACK. Esto tiene el impacto de incrementar exponencialmente el tamaño de ventana cada RTT.
- El nombre se debe a que, a pesar de subir exponencialmente, se arranca desde un valor pequeño, tanteando la capacidad de la red para no enviar un paquete del tamaño máximo permitido por

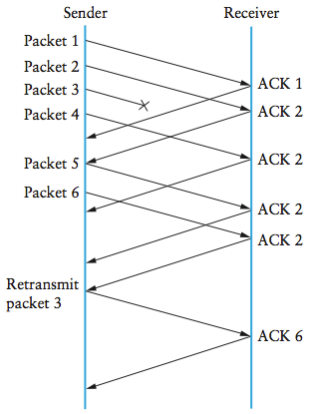
AdvertisedWindow de entrada (como ocurría anteriormente)

Además de en el inicio de la conexión, el mecanismo se utiliza cuando se comienza a enviar de nuevo tras la pérdida de paquetes. Escenario:

- Se pierden un segmento, y *source* llegó al punto que ya agotó todo el AdvertisedWindow por lo que no puede seguir enviando.
- Dado que el ACK nunca llega, eventualmente ocurre un timeout, y *source* considera que ya puede comenzar a enviar nuevamente un segmento del tamaño de AdvertisedWindow.
- En vez de hacer esto realiza slow start, comenzando del mínimo y aumentando hacia el último valor conocido de CongestionWindow (y luego siguiendo con additive increase).
- Para implementar esto, se introduce una nueva variable **CongestionThreshold**, que se setea igual a CongestionWindow tras cada multiplicative decrease.
- Slow start no se utiliza con timeouts ocurridos por fast retransmit (ver siguiente)

Fast Retransmit - Fast Recovery

- El mecanismo tradicional de timeouts de TCP (self-clocked por RTT) daba lugar a varios períodos de inactividad en los cuales no ocurría nada hasta que se disparara un timeout.
- **Fast retransmit** es un mecanismo adicional que puede disparar retransmisión antes que el timeout ocurra.
- Se implementa haciendo que el receptor envíe un ACK por cada paquete que llega (aunque sea fuera de orden y tenga que mandar un ACK duplicado de un paquete anterior).
- Cuando el emisor detecta cierta cantidad de ACK duplicados, retransmite el segmento con número siguiente al ACK (sin necesidad de esperar el timeout).



Observar que cuando se dispara un timeout por fast retransmit todavía pueden quedar ACKs en camino. **Fast recovery** utiliza estos ACK para proceder con la dinámica común de AIMD, evitando la fase de slow start. En definitiva, slow start sólo se utilizará al principio de la conexión y cuando ocurra un timeout regular.

Congestion Avoidance

TCP está diseñado para reducir la congestión una vez que ocurre. Incluso, para detectar el estado de la red aumenta el tráfico hasta que se detectan pérdidas de paquetes.

No incluye, sin embargo, mecanismos para sensar la capacidad de la red y controlar el tráfico emitido antes de que la misma se congestione. Los siguientes son mecanismos con ideas que podrían usarse también en TCP para evitar la congestión.

DECbit

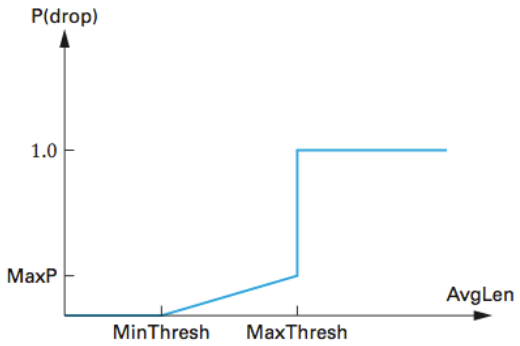
- Cada router monitorea su estado y notifica a los hosts cuando está congestionado.
- La notificacións se implementa seteando un bit en todos los paquetes que pasan por el router. El router destino copia el valor del bit en los ACK que envía.
- Los hosts deciden, en función del porcentaje de paquetes que activaron el bit, cuándo achicar su ventana de congestión con un mecanismo similar a AIMD.

RED (Random Early Detection)

- Los routers también monitorean su estado y notifican a los hosts.
- En este caso, la notificación es **implícita**: se dropean paquetes y el host reacciona ante este evento.
- Se produce antes de llegar a la situación de congestión: preferible dropear unos pocos paquetes en un momento para evitar dropear muchos más después.
- "Colabora" con el mecanismo de TCP: lo que efectivamente reduce la ventana es el mecanismo de control de TCP, disparado por el timeout de RED.
- Cuando se excede un threshold de longitud en las colas, el router dropea paquetes aleatoriamente.
- Los flujos que ocupen mayor ancho de banda tendrán más paquetes dropeados.

Detalles:

- Se calculan longitud promedio de las colas para decidir en base a congestión "estable" y no por ráfagas de tráfico.
- Si la longitud promedio supera cierto intervalo, se descartarán paquetes aleatoriamente (con probabilidad creciente a medida que crecen las colas)
- Cuando se supera el threshold superior del intervalo, se dropean todos los paquetes.



FRED (Flow Random Early Detection)

- Si un flujo que consume mucho no baja su taza de emisión, RED empieza a droppear y termina castigando a otros flujos más pequeños.
- FRED mantiene estadísticas similares a las de RED pero por flujo. Por ende, mantiene estado y requiere más cómputo por parte de los routers.
- El objetivo es ser más agresivo con los flujos que ocupan más lugar de buffer, manteniendo distintas probabilidades.
- Por ejemplo, permite que se dropeen todos los paquetes de un flujo "bandido" (habiendo superado el threshold) pero sin afectar a otros flujos.