TCP

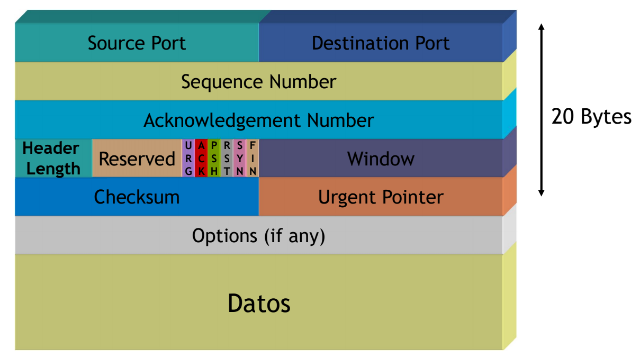
Protocolo de control de la transmisión (RFC 793/1122). Capa 4 (Transporte).

Notas tomadas de [video](https://drive.google.com/drive/folders/1LwzHpjmQy6WfanClurL-K6K2Jqisg2pg).

# TCP. Características

* **Protocolo punto a punto.** Sirve exclusivamente para conectar dos hosts
  + Para más de 2 hosts, se debe usar otro protocolo. Ej: UDP sirve para multi y broadcast
* Orientado a la conexión. Cada uno de los dos extremos tiene que mantener el estado de la conexión (qué se envió, qué hay que enviar, etc)
* Utiliza “segmentos” generalmente contenidos en un único datagrama IP.
  + Segmentos TCP es como se llama a la PDU de capa 4.
  + Numero de secuencia en octetos (bytes).
* Confiabilidad se alcanza mediante:
  + Confirmaciones.
  + Timeouts.
  + Retransmisiones.
  + Checksum **de la cabecera y el cuerpo.** (IP solo chequeaba cabecera)
* El control se hace extremo a extremo es decir entre los dos extremos de la conexión y no en los nodos intermedios.

# Formato TCP

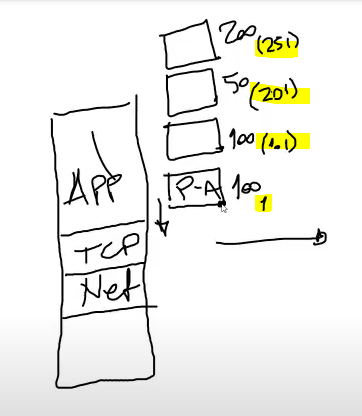


La cabecera TCP tiene un mínimo de 20 bytes, a lo que se le suman las opciones si las hay.

### Source y destination port (2 bytes cada uno)

* En cada host (dirección IP de capa 3) puede haber varias aplicaciones corriendo. El puerto indica para qué aplicación es el mensaje.
* TCP está orientado a la conexión. Una conexión se realiza entre 2 extremos o endpoints. Cada endpoint está conformado por dirección IP + puerto, y la combinación de dos extremos forma una conexión.
  + End-point: Dirección IP + TCP Port.
  + Conexión: 2 endpoints (o sea, IP:puerto origen + IP:puerto destino)
    - Cada conexión tendrá una combinación única de 4 valores.
    - Es posible que 2 conexiones sólo varíen en uno de los puertos, que normalmente es el del puerto cliente.
    - Ej: en 2 pestañas diferentes de un browser me conecto a google.com. Las dos conexiones van a compartir IP origen, IP destino y puerto destino. Pero van a tener puerto de origen diferente, porque la computadora va a asignar un puerto diferente a cada conexión.
* Al haber 16 bits, el puerto máximo es 16535

### Sequence number / Número de secuencia (4 bytes)

* **Número de orden del primer byte en el mensaje.** TCP cuenta en octetos (bytes), no segmentos ni mensajes.
* El número de secuencia indica el orden que tienen los datos dentro del mensaje que vino de la capa superior. (En la imagen, en amarillo está el número de secuencia. Sin resaltar, está el número de bytes de ese mensaje).
* TCP va a establecer una **conexión** con el otro extremo remoto, y luego va a numerar o **secuenciar** toda la información (segmentos) que recibe de la capa superior (está ultima envía varios mensajes de distintas longitudes) conformando una larga secuencia de octetos. TCP va a ir **entregando** los mensajes uno a uno en orden, de manera que la aplicación reciba los mensajes **en el orden en que fueron transmitidos**. Y luego va **verificando** que el extremo receptor le confirme cada octeto.   
  Al recibir un mensaje, TCP va a mandar una **confirmación** indicando que espera recibir el próximo número de secuencia al recibido (es decir, si recibió 2 mensajes de 100 y uno de 50, indica: 251, que es el último byte recibido + 1 – de ahí se deduce que se recibió correctamente hasta el 250). Este número próximo que se espera, se indica en el campo acknowledgement Number.

### Acknowledgement number / Número de confirmación (4 bytes)

* **Próximo número de secuencia que el emisor de este segmento espera recibir.**
* **Válido solamente cuando el flag ACK=1**. El flag ACKnowledge indica que el mensaje es una confirmación, y que hay que prestar atención al campo de número de confirmación.
* Las confirmaciones son acumulativas y secuenciales.
  + El receptor no tiene la obligación de confirmar por separado cada segmento, si le llegan dos juntos puede enviar un único mensaje de confirmación ACK con el próximo número inicial de secuencia.
  + El receptor sólo puede confirmar un mensaje si recibió ese y todos los anteriores. Si uno de los anteriores no está, no se confirma.

### Header Length (4 bits)

* Cantidad de palabras de 4 bytes que hay en la cabecera.
* Se necesita para saber si las opciones se utilizan.
* Normalmente el valor es 5, porque suelen ser 20 bytes (sin opciones).

### Window Size (2 bytes)

* Cantidad de bytes, comenzando por el indicado en el campo ACK, que el emisor está dispuesto a recibir.
* Se utiliza para el Control de flujo.

### Checksum (16 bits)

* Código de **detección de errores**. Me garantiza la entrega libre de errores.
* Se hace sobre la **cabecera y el campo datos**.

### Urgent Pointer (16bits)

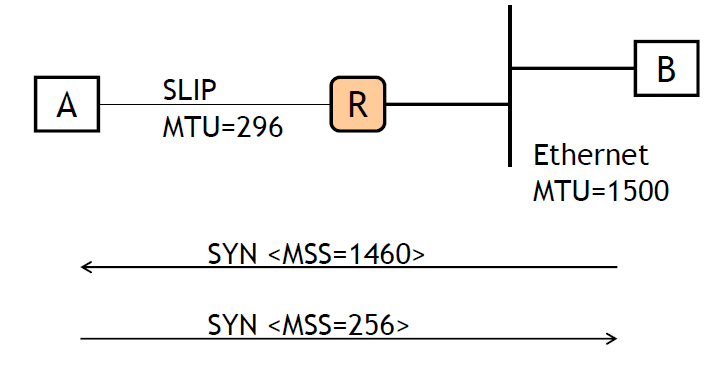
* TCP contempla añadir un mensaje urgente al mensaje normal.
* Le permite a la capa de transporte enviar un mensaje al extremo remoto fuera de la secuencia. Por lo tanto permite enviar bytes junto con un segmento TCP pero indicándole al extremo receptor que esos bytes no forman parte de la secuencia.
* Cómo funciona?
  + Hay un flag URG. Si URG = 1, significa que hay un mensaje urgente añadido.
  + En la porción de datos del PDU, los datos urgentes van primero.
  + **El puntero de urgente, apunta al último byte de la secuencia de datos urgentes**, para indicar dónde terminan esos datos urgentes y empiezan los normales.
* Ejemplo: Permite intercalar un mensaje para cancelar una descarga grande, por la mitad de la descarga.

### Options (variable, opcional)

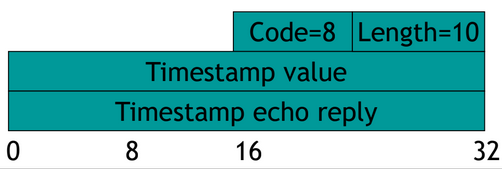
* Se usan muy frecuentemente (en IP casi no se usaban).
* La opción más común es el **MSS** (Maximum Segment Size), **Timestamp**, **Window Scale** **Factor**.

#### Opción: MSS (Maximum Segment Size)

* Es declarada al establecimiento de la conexión (segmentos con flag SYN=1). No puede modificarse durante el intercambio de segmentos.
* Determina el tamaño máximo del segmento de datos que es capaz de aceptar. e busca que sea lo más grande posible para más eficiencia (mayor relación datos/bytes posible) y menor fragmentación.
* Tanto cliente como servidor dos se comunican el MSS, y se va a usar el menor de los dos para la comunicación entre ambos.
* Debe ser: MTU de la interfaz menos 40 bytes. Los 40 bytes surgen porque la cabecera IP ocupa 20 bytes y la TCP otros 20 bytes. Si el MTU de Ethernet es 1500 y cada cabecera es de 20, el MSS es de 1460 (0x5B4). En vez de Ethernet, podría haber otro protocolo de nivel 2, con otro MTU (ej: SLIP, con MTU de 296 y MSS de 256).
* El MSS no soluciona problemas de MTU de nodos intermedios.



#### Opción: Timestamp

* El origen pone el stamp, el destino responde con un stamp al confirmar (ACK).
* Permite calcular de manera más precisa el roundtrip tipe (RTT) por cada segmento.
* Sin esta opción, el RTT se calcula cada ventana. Ineficiente para ventanas grandes.

#### Opción: window scale factor

* La window size determina cuántos bytes se pueden transmitir en la ventana deslizante. Permite 16 bits, por lo que el tamaño máximo original de la window es 2^16 = 64KB.
* Si se quisieran transmitir más de 64KB por ventana, se puede usar la opción window scale e indicar otra escala de ventana, en vez de bytes como unidad de medida. Esto permite ventanas de hasta 1GB (2^30).
* Entiendo que lo que hace es multiplicar el window size por 2^scaling\_factor. Máximo: 14. Entonces: 2^16 (window) \* 2^14 (scaling factor) = 2^30 = 1GB

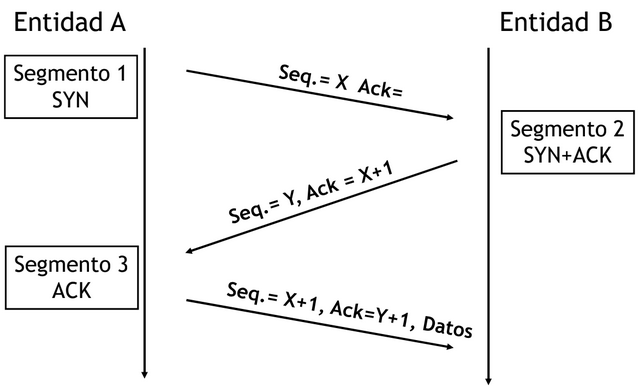
### Flags (6 bits)

* **URG:** El contenido de Urgent pointer es válido y relevante. Existen datos urgentes, y el urgent pointer apunta al último byte de datos urgentes.
* **ACK:** El contenido del campo ACK es válido y relevante. Estoy confirmando la recepción de otro mensaje.
* **PSH:** Push. Procesar tan pronto como pueda. Obliga al receptor a enviar la confirmación.
  + Sin esto, el receptor puede elegir si responder inmediatamente o esperar a tener que mandar otro mensaje y agregar la confirmación ahí (piggy-backing).
  + Útil para aplicaciones que necesitan procesamiento inmediato del mensaje, como Telnet.
* 3 flags usados en [establecimiento y fin de una conexión](#_Three-way_Handshake).
* **RST:** Esta conexión debe reiniciarse. Sucede cuando:
  + No se acepta un pedido de conexión.
  + Uno de los dos extremos perdió el estado de la conexión y necesita reiniciar.
* **SYN:** Sincronizar números de secuencia.
* **FIN:** El emisor no tiene más datos para enviar. Se solicita cerrar la conexión (lo tiene que enviar el cliente y recibir ACK, y después el servidor y recibir ACK – 4 pasos de cierre).

# Three-way Handshake

El establecimiento de la conexión se realiza mediante el Three-way Handshake (intercambio de tres pasos).

1. **Intención de establecer conexión. “Segmento SYN”.** El cliente envía un segmento TCP al servidor indicando su intención de establecer una conexión. Se envía un segmento sin datos con el flag SYN encendido. Flag ACK = 0, campo ACK vacío. Envía el número de secuencia (un número aleatorio no igual a 0 – ISN: initial sequence number).
2. **Aceptación de conexión. “Segmento SYN+ACK”.** El servidor, si lo acepta, le envía un mensaje al cliente indicándole que acepta la conexión. Mensaje con los flag SYN y ACK encendidos, sin datos. Envía el número de secuencia (ISN del servidor), y confirma con ACK el número de secuencia ISN enviado por la entidad A en el primer mensaje.
   * **Si no aceptara la conexión.** Se envía flag RST = 1 (reset).
3. **Confirmación. “Segmento ACK”** El cliente envía una confirmación, e incluso también podría enviar datos en ese mensaje. Es un mensaje con el flag ACK=1, que puede o no tener datos (habitualmente SIN datos). A partir de ese momento se pueden enviar datos.



En segmento SYN y SYN+ACK: Seq = ISN (initial sequence Number). X es ISN de cliente, Y es ISN de servidor. No guardan relación entre sí.

ISN es un número aleatorio diferente a 1, por seguridad, que se va a tomar como cero relativo.

X es el valor de secuencia del emisor. Y es el valor de secuencia del receptor. X e Y no tienen relación. X e Y son los ceros relativos (es un número absoluto que se toma como cero relativo), el otro extremo le envía la confirmación de que espera X / Y + 1.

~~Sobre el mecanismo de handshake opera el firewall. Podría decidir:~~

1. ~~que puede conectarse desde adentro hacia afuera, y no al revés (no permite SYNs desde afuera hacia adentro).~~
2. ~~Que si llega un SYN+ACK, sólo se acepte si antes hubo un SYN.~~

## Diagrama de estados: apertura y cierre normal de una conexión

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **Mensajes** |  | **Estado Cliente** | **Estado servidor** |
| Inicial |  |  |  | CLOSED | LISTEN |
| Apertura | C | ------SYN ----- > | S | SYN\_SENT |  |
| C | < -- SYN/ACK – | S |  | SYN\_RECEIVED |
| C | ------ACK ----- > | S | ESTABLISHED | ESTABLISHED |
| Transmisión de datos | C | < -------------- > | S | ESTABLISHED | ESTABLISHED |
| Cierre | C | ------FIN ----- > | S | FIN\_WAIT\_1 |  |
|  | C | < ----ACK------- | S | FIN\_WAIT\_2 | CLOSE\_WAIT |
|  | C | < ---- FIN ------- | S |  | LAST\_ACK |
|  | C | ------ACK ----- > | S | TIME\_WAIT | CLOSED |

### Punto de vista del cliente

(Estados en azul)

1. Estado inicial: **CLOSED**.
2. Un cliente se va conectar con un servidor, se produce lo una apertura activa (la aplicación le dice a TCP que se conecte a determinada IP y puerto) y TCP envía un SYN.

Desde el lado del cliente esa solicitud se queda como **SYN\_SENT** esperando que el servidor acepte la conexión. Está esperando un mensaje SYN ACK.

1. Una vez que el servidor responde SYN ACK, el cliente envía el ACK y la conexión queda **ESTABLISHED**.
2. Fluye la información.
3. Cuando la aplicación quiere cerrar la conexión, se lo indica a TCP y este envía un segmento FIN (segmento vació sin datos que tiene el flag FIN encendido) al servidor, y pasa a un estado **FIN\_WAIT** porque le indicó que quiere cerrar la conexión.
4. Cuando recibe el ACK del servidor, lo que queda cerrado es una mitad de la conexión (la que va del cliente al servidor).

El cliente pasa a un estado **FIN\_WAIT\_2** donde va a esperar que el servidor cierre su mitad de la conexión.

1. Cuando reciba el FIN del servidor, enviara un ACK y ahí si quedará cerrada la conexión. El estado será **TIME\_WAIT**.

**Un cierre de comunicación se realiza en 4 pasos. La apertura se realiza en 3.**

### Punto de vista del servidor

1. Estado inicial: **CLOSED**.
2. La aplicación hace una apertura pasiva (le indica a TCP que se ponga escuchar en un puerto determinado a la espera de conexión de los clientes) y pasa a modo de escucha (**LISTEN**).

En el modo escucha, TCP está escuchando, esperando segmentos SYN en ese puerto para que se le conecten los clientes.

1. Cuando recibe un SYN, envía un SYN ACK y pasa al estado **SYN\_RCVD** (SYN recibido).
2. Cuando el cliente confirma con el ACK pasa al **ESTABLISHED**.
3. Intercambia la información.
4. Recibe el fin del cliente y envía el ACK, pasa a **CLOSE\_WAIT** porque cerró la mitad del cliente.
5. Envía un FIN y recibe el ACK, pasa al estado **LAST\_ACK**.
6. Acá pasa a CLOSED y luego a LISTEN, porque va a seguir escuchando si vienen nuevas conexiones.

# Control de flujo

El control de flujo es un mecanismo que le permite a uno de los dos extremos apaciguar al otro, es decir, al transmisor. Permite controlar la cantidad de información que el otro extremo le envía.

¿Cómo se implementa (en general)?

* + Stop and wait. Mecanismo más sencillo. A le envía mensaje a B, y espera confirmación de B antes de mandar a otro. Mientras B no confirma, A no puede enviar. Es half-duplex, nunca envían los 2 a la vez. Inconveniente: Es ineficiente en el uso del enlace de datos.
  + Ventana deslizante / Sliding window. Se aumenta la cantidad de mensajes que A le puede enviar a B sin esperar confirmación. El enlace se usa más eficientemente.

## TCP: Esquema de otorgamiento de créditos

En TCP, el control de flujo se hace con el “Esquema de Otorgamiento de Créditos”, que es una variante del mecanismo de ventana deslizante.

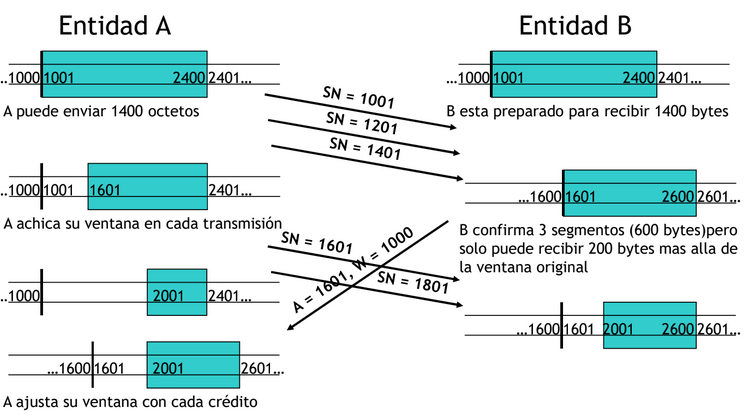
* Un extremo le indica al otro cuántos bytes (o “créditos”) tiene permitido enviarle antes de requerir una confirmación. Estos bytes están relacionados con el tamaño del buffer asignado a una conexión determinada.
* TCP Confirma la recepción de datos con un mensaje de la forma (A=i; W=j). Son dos campos de la cabecera TCP donde:
  + A = ACK = i. Se confirma la recepción de todos los octetos hasta i-1, se espera recibir el i. Sirve para confirmar.
  + W = Window = j. Se permite enviar una nueva ventana de datos (W= j octetos). Esto es desde i hasta i + j – 1. Si W = 0 no permite enviar más. Esto significa que el otro tiene permitido mandar j bytes más, a partir del último confirmado.
* Tanto cliente como servidor mandan una ventana, y puede tener diferente tamaño. Cada ventana aplica para las comunicaciones en una de las dos direcciones.

El protocolo usa un mecanismo de la forma “ventana deslizante” tal como HDLC (High-level Data Link Control).   
A diferencia de HDLC, separa la confirmación de datos recibidos del permiso para enviar más. Es decir separa el mecanismo de control de errores del de control de flujo.

* Cada Octeto de datos se considera que tiene un número de secuencia.
* El receptor confirmará los segmentos y otorgará más crédito, solo si tiene espacio disponible en el buffer.
* La tasa a la cual se envían los segmentos está determinada por la tasa a la cual se reciben las confirmaciones de los segmentos enviados.

### Ejemplo

En el siguiente ejemplo se muestra la ventana que le otorgó B a A. La de A a B no está graficada pero ocurre lo mismo.



Explicación:

* **Estado inicial:** 
  + A puede enviarle 1400 octetos (bytes) a B comenzando desde la posición 1001.
  + B puede recibir 1400 bytes.   
    Esto se acordó cuando establecieron la conexión.
* **SN = 1001 / 1201 / 1401.** 
  + La capa de aplicación le pasa a A una secuencia de 1201 bytes, por lo que A lo encapsula en un segmento TCP, escribe un numero de secuencia 1001 y se lo envía a B. Hace lo mismo 2 veces más.
  + Entre los 3 suman 600 bytes y A tenía un crédito de 1400 bytes por lo que podía enviarlos sin problema.
  + Con esto, desde el punto de vista de A la ventana se redujo. Le quedan 800 bytes de crédito. Sus créditos van desde 1601 a 2400.
* **(A=1601, W=1000)**. B recibe correctamente los tres segmentos. Envía un segmento TCP con (A=1601, W=1000), es decir que recibió correctamente hasta el 1601 (A), también le indica a A que le otorga un crédito de **1000 a partir del 1601 (W).**
* **(SN = 1601 / SN = 1801).** Antes de recibir la confirmación de B, A envía dos segmentos más de 200 bytes. Se reduce la ventana (conocida hasta el momento por A) a 400 bytes.
* B recibió los dos nuevos mensajes y se ve que a A le quedan 600 bytes de crédito.
* A recibe la confirmación de B, entonces desde 1601 **suma 1000 créditos (hasta 2601),** de los cuales ya consumió 400 de los dos mensajes anteriores. Quedan 600 créditos (de 2001 a 2600).

**La cuenta se lleva de los dos lados y en ambos sentidos.**

Cliente y servidor, cada uno lleva el estado de cuánto recibió, cuánto confirmó, cuánto le queda por enviar, etc.

## Mecanismo desde el punto de vista del transmisor

Los valores de datos confirmados, no confirmados y transmisión permitida (crédito) se calculan a partir de:

* NS/SN inicial,
* NS/SN en último ACK recibido,
* próximo NS/SN a enviar
* y fin de ventana.

# Control de errores

* TCP incluye un mecanismo de control de errores. **OSI plantea que el control de errores se hace en capa 2 y 4**.
* Si la información no llega o llega mal, se retransmite. La estrategia de corrección de errores es ARQ (Automatic Retransmission Request). El host que envía, retransmite automática ante la ausencia de confirmación.   
  TCP se basa en la confirmación positiva de la recepción y retransmite cuando la confirmación no llega dentro de un período determinado (conocido como RTO - retransmission time out).
* El tiempo del RTO depende de lo que debe tardar en llegar el mensaje y la confirmación del mismo. Se calcula en base al RTT (round trip time) que es lo que tarda en volver un ¿ping?. Se espera por lo menos un RTT de ping + un poco más para el router.
  + El valor del RTO debe estar en el orden del RTT
  + Un RTO mal estimado:
    - demasiado corto, hace que se me cancele antes de tiempo la conexión y no voy a poder comunicar. Va a llevar a retransmitir demasiadas veces.
    - RTO demasiado largo: voy a esperar demasiado tiempo cuando hay un error.
* Con el checksum detecto errores. **Si hay errores,** **descarta el mensaje y no lo confirma**. Por lo que pasado RTO, el transmisor lo retransmite.

Comparación con otros protocolos:

* No existe en TCP una confirmación de rechazo, tal como REJ o SREJ en HDLC. Es decir lo único que puede recibir del receptor es la confirmación, no hay mensaje de rechazo.

# Control de la congestión

## Congestión

La congestión se produce en una red cuando el tráfico que generan los clientes es superior a la capacidad que tiene la red de cursar el tráfico (más tráfico que el que la red puede soportar).

La congestión se debe evitar: Si se envía un mensaje a una red saturada, el mensaje va a ser descartado. Pero luego va a ser retransmitido, y empeora el problema exponencialmente. En una red saturada se ve muchas copias del mismo mensaje.

La idea es aplicar un mecanismo de control cuando se está acercando al umbral de congestión.

## Mecanismos de control: Manejo de la ventana

Cuando se acerca una congestión, se le indica a los nodos que están usando la red que transmitan menos o dejen de transmitir para evitar la congestión.

El mecanismo de control de congestión le permite a un host reaccionar frente a evidencias de congestión.

**Cómo funciona:**

Al comienzo de la transmisión, no se tiene información acerca del estado de la red. Es necesario determinar cuántos segmentos pueden enviarse.

Se define una ventana permitida, que es el mínimo entre la ventana de congestión y el crédito.

**anwd = MIN(cnwd, credit).**

Donde:

* Anwd (allowed window): ventana permitida, en segmentos. Cantidad de segmentos que se pueden enviar ahora, sin esperar ACK.
* Cnwd (congestion window): ventana de congestión, en segmentos. Usada por TCP en el comienzo y durante periodos de congestión. Inicialmente vale 1.
* credit: cantidad de bytes permitidos por el destino, en segmentos   
  ( = Window / Tamaño de segmento MSS).

### Mecanismo 1: Slow start

Es la primera aproximación que se hizo en TCP para manejar la congestión.

* Cuando se inicia una nueva conexión, se inicializa cnwd = 1. Cada vez que se recibe una confirmación, se incrementa en 1.
* Cuando un segmento se pierde (caduca RTO), cnwd vuelve a valer 1 y comienza nuevamente.

Cada vez se pueden enviar más mensajes, y así se va testeando el estado de la red.

Inconveniente: **“Sincronización global”:** Si hay múltiples clientes conectados al mismo router, todos inician la conexión y empiezan a transmitir al mismo tiempo, llenando el enlace. Cuando se descarta un paquete, todos vuelven a 1 (tienen sincronización global) y otra vez empiezan a transmitir lentamente.



### Mecanismo 2: Fast retransmit

Refina el mecanismo del RTO y las retransmisiones. **No soluciona la sincronización global.**

Cuando la fuente recibe un ACK duplicado, significa:

* El segmento fue demorado pero finalmente llegará.
* El segmento se perdió, deberá retransmitirse.

En lugar de esperar a que se caduque el RTO, si el transmisor recibe 3 ACK duplicados, directamente retransmite el segmento perdido.

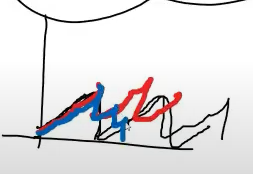
El motivo por el que llegarían múltiples ACK repetidos es que se envían ACKs cada cierto periodo de tiempo y si no recibió nada más, envía el mismo que el anterior.

### Mecanismo 3: Fast recovery

Soluciona el problema de sincronización global. Fusiona las dos técnicas anteriores.

Cuando se recibe el 3er ACK duplicado, se setea cwnd = cwnd / 2 (en vez de 1 como en slow start). Considera que la red no necesariamente necesita volver a 1, sino que va bajando escalonadamente.

Esta variante permite al retransmisor evitar volver al slow start cuando se pierde un segmento.



* En rojo: lo que pasa si bajé el cnwd al medio y no había congestión.
* En azul, lo que ocurre si había congestión (sigue dividiendo el cnwd).

# Well-known Ports

**Server ports, well known ports < 1024**. Originalmente, los bien conocidos son menores al 1024. Son los puertos a los que todos los browsers tratan de conectarse si no se aclara (son por defecto para cada protocolo)

El cliente transmite por puerto mayor a 1024.

Hoy por hoy ya todos los puertos conocidos están ocupados. Por eso un servicio más nuevo como H.323 ya no tenía puerto menor a 1024 y usa el 1720.

# TCP vs UDP

|  |  |
| --- | --- |
| TCP | UDP |
| Protocolo punto a punto | Usado especialmente multi- y broadcast. O para peticiones simples de request / reply |
| Orientado a la conexión (mantiene estado de conexión)   * Establecimiento: 3-way handshake * Cierre de conexión | NO orientado a la conexión. |
| Confiable (gracias a confirmaciones, timeouts – retransmisiones, y CRC). | No confiable (no reordena, no confirma) |
| Cabecera de 20 bytes o más (más, si hay opciones) |  |
|  |  |
| **Control de flujo** por sistema de otorgamiento de créditos |  |
| **Detección de errores** con CRC de datagrama completo | Hay un campo de checksum opcional sobre el datagrama completo. Este checksum genera el pseudo-header, lo calcula e incluye en checksum, pero no lo envía. |
| **Control de congestión** con manejo de ventana | Sin control de flujo |