Resumen Capa de Transporte

Introducción

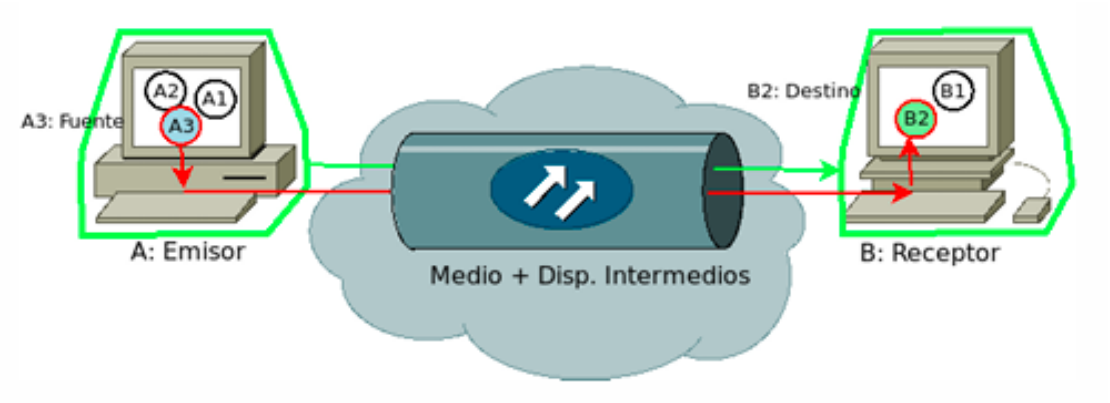
- En la **Capa de Red** se utiliza el protocolo IP que es débil y no fiable, es decir, a la hora del envío de paquetes o datagramas por la red estos mismos pueden ser descartados, desordenados, retardados, duplicados, corrompidos, etc. Para solucionar estos problemas se usa la **Capa de Transporte**.

Servicios de la Capa de Transporte

- Proporcionar una **comunicación lógica** entre procesos de aplicación que se ejecutan en hosts diferentes. Por **comunicación lógica** queremos decir que, desde la perspectiva de la aplicación parezca que los hosts que ejecutan los procesos estén conectados directamente, aunque no necesariamente lo estén.
    - Los **procesos de aplicación** usan esta comunicación para enviar mensajes sin preocuparse de la infraestructura física (Capa de Red) utilizada para transportar esos mensajes.
    - Los **procesos que se comunican** se ven identificados por un **número de puerto**. Estos puertos actúan como puntos finales en una comunicación y

permiten que múltiples aplicaciones en una misma computadora o dispositivo se comuniquen simultáneamente a través de la red.

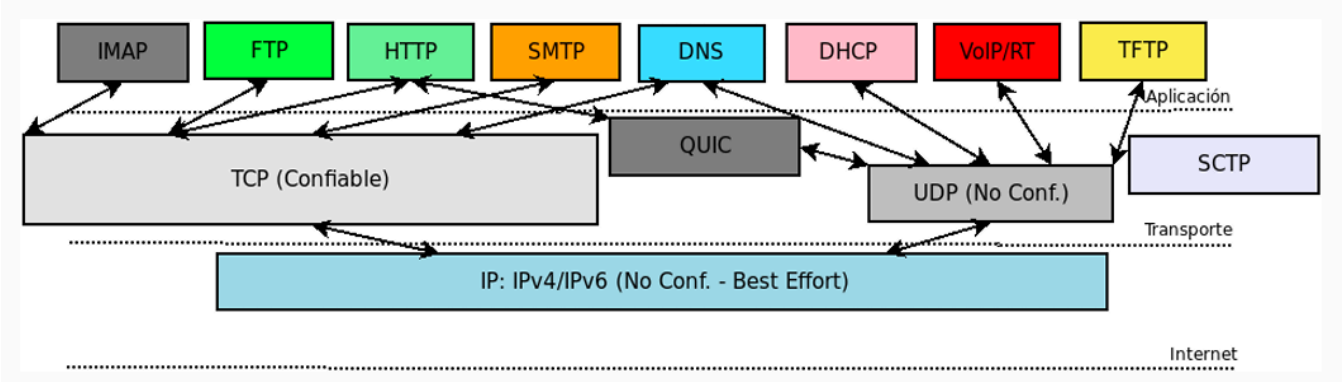
- Los protocolos de esta capa están implementados en los **sistemas terminales**, pero no en los routers de la red.
    - En el **lado emisor**, la capa de transporte convierte los mensajes que recibe procedentes de un proceso de aplicación emisor en paquetes de la capa de transporte, conocidos como **segmentos** de la capa de transporte en la terminología de Internet.
    - En el **lado receptor**, la capa de red extrae el **segmento** de la capa de transporte del datagrama y lo sube a la capa de transporte. A continuación, esta capa procesa el segmento recibido, poniendo los datos del segmento a disposición de la aplicación de recepción.



Direccionamiento a nivel Transporte

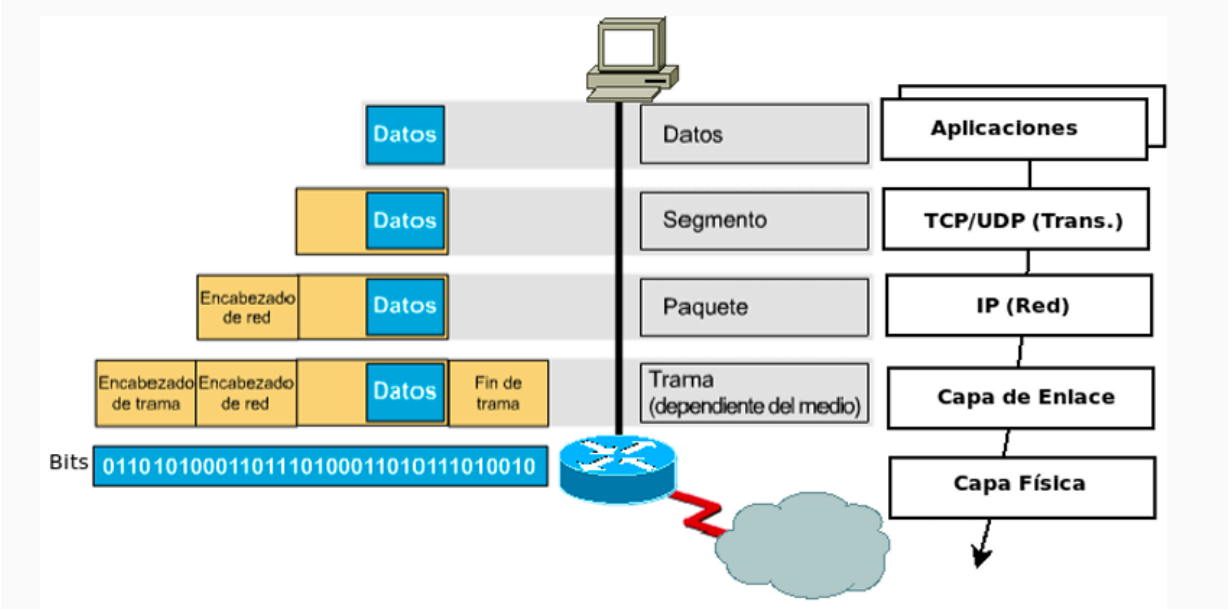
Funciones de la Capa de Transporte

- Multiplexación y Demultiplexación de conversaciones.
- Encapsulación:
    - Define PDU (Protocol Data Unit) donde se envían los mensajes de la aplicación.
- Soporte de datos de tamaños arbitrarios.
- Control y Detección de Errores (pérdida, duplicación, corrupción. etc).
- Control de Flujo y Control de Congestión (TCP).



Encapsulación

- En esta capa los **Datos** de la capa de aplicación se ven encapsulados en lo que se denomina **Segmento** para **TCP** o **Datagrama** en el caso de **UDP**. Estos **Segmentos** son los que viajan por la red dentro de un **Paquete IP.**



Encapsulación

Puertos

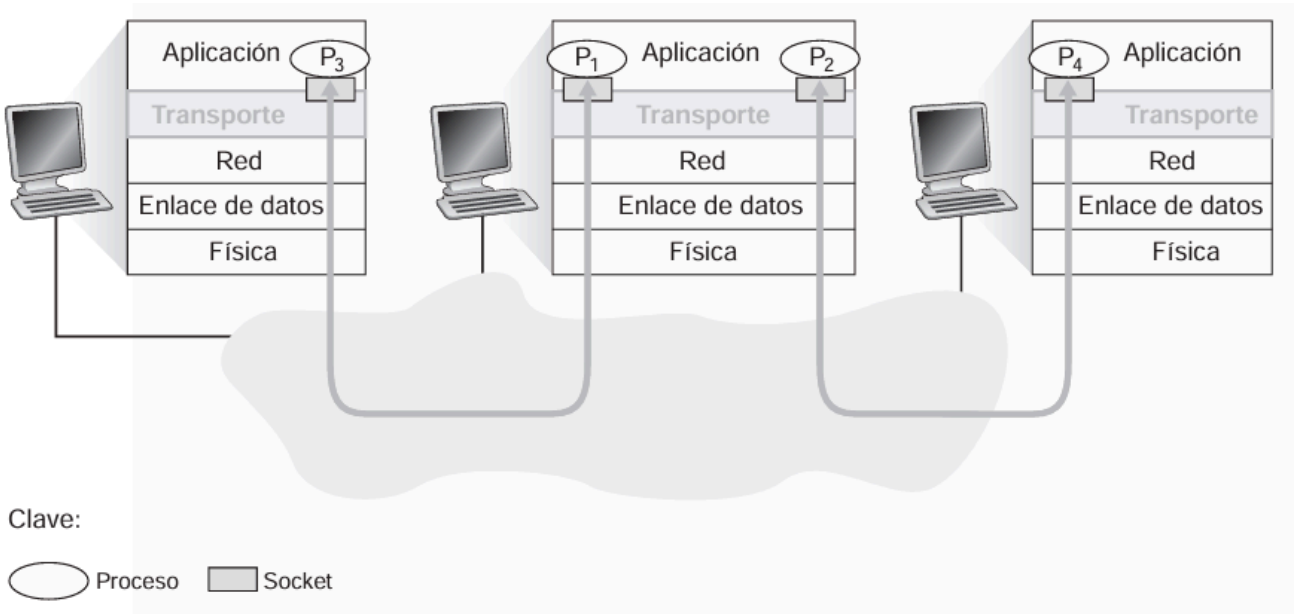
- Número de 16 bits comprendido en el rango de 0 a 65535. Los números de puerto pertenecientes al rango de 0 a 1023 se conocen como **números de puertos bien conocidos** y son restringidos, lo que significa que están reservados para ser empleados por los protocolos de aplicación bien conocidos, como por ejemplo HTTP o FTP. Al desarrollar una nueva aplicación, es necesario asignar un número de puerto a la aplicación.

Sockets

- Cuando estamos desarrollando nuestra aplicación tenemos que definir el uso de uno de los protocolos de transporte (TCP o UDP) cuando creamos los Sockets ya que el acceso a los servicios de transporte se hace mediante **API: Network Socket**. La elección va a variar dependiendo las necesidades de la aplicación.
- Un proceso (como parte de una aplicación de red) puede tener uno o más **sockets**, cada uno con un identificador único, que actúan como puertas por las que pasan los datos de la red al proceso, y viceversa. La **capa de transporte** del host receptor realmente no entrega los datos directamente a un proceso, sino a un **socket**

intermedio. Dado que en cualquier instante puede haber más de un socket en el host receptor.

- El **formato** del **identificador del socket** depende de si se trata de un **socket UDP** o de un **socket TCP**.
    - Un **socket UDP** queda completamente identificado por una tupla que consta de una dirección IP de destino y un número de puerto de destino.
    - Un **socket TCP** queda completamente identificado por una tupla que consta de una dirección IP de origen, número de puerto de origen, dirección IP de destino y número de puerto de destino.
- **Demultiplexación:**
    - Cada segmento de la capa de transporte contiene un conjunto de campos destinados para la dirección del segmento al socket apropiado. En el extremo receptor, la capa de transporte examina estos campos para identificar el socket receptor y, a continuación, envía el segmento a dicho socket. Esta tarea de entregar los datos contenidos en un segmento al socket correcto es lo que se denomina **demultiplexación.**
- **Multiplexación:**
    - La tarea de reunir los fragmentos de datos en el host de origen desde los diferentes sockets, encapsulando cada fragmento de datos con la información de cabecera (la cual se utilizará después en el proceso de demultiplexación) para crear los segmentos y pasarlos a la capa de red es lo que se denomina **multiplexación**.
    - **Requiere:**
        - Que los sockets tengan identificadores únicos. **Campo de número de puerto origen.**
        - Que cada segmento tenga campos especiales que indiquen el socket al que tiene que entregarse el segmento. **Campo de número de puerto de destino.**



Multiplexación y Demultiplexación

UDP - User Datagram Protocol

- Protocolo **minimalista y Full Duplex** que genera **menos Overhead**.
- Orientado a **Packets/Datagramas.**
- Solo provee **Multiplexación** y **Demultiplexación** de conversaciones.
- El **PDU** utilizado en este protocolo se denomina **datagrama de usuario**.
- Proporciona un servicio que no requiere establecimiento de conexión y que es **no fiable** a la aplicación que lo invoca. Esto quiere decir que UDP no garantiza que los datos enviados por un proceso lleguen intactos (o que ni siquiera lleguen) al proceso destino.
    - Es posible que una aplicación disponga de un **servicio fiable de transferencia de datos** utilizando **UDP**. Esto puede conseguirse si las características de fiabilidad se incorporan a la propia aplicación.
- Tiene como **responsabilidad principal,** al igual que TCP, ampliar el servicio de entrega de IP entre dos sistemas terminales (host a host) a un servicio de entrega entre 2 procesos que estén ejecutándose en los sistemas terminales.
- Tiene aplicaciones dentro del área de video/voz streaming/TFTP/DNS/Bcast/Mcast.
- UDP proporciona un **mecanismo de comprobación de errores**, pero no hace nada para recuperarse del error. Algunas implementaciones de UDP simplemente descartan el segmento dañado y otras lo pasan a la aplicación junto con una advertencia.

¿Por qué elegir UDP frente a TCP?

- **Mejor control en el nivel de aplicación sobre qué datos se envían y cuándo:**
    - Con UDP, tan pronto como un proceso de la capa de aplicación pasa datos a UDP, UDP los empaqueta en un segmento UDP e inmediatamente entrega el segmento a la capa de red. Por el contrario, TCP dispone de un mecanismo de control de congestión que regula el flujo del emisor TCP de la capa de transporte cuando uno o más de los enlaces existentes entre los hosts de origen y de destino están excesivamente congestionados. Puesto que las aplicaciones en tiempo real suelen requerir una velocidad mínima de transmisión, no permiten un retardo excesivo en la transmisión de los segmentos y pueden tolerar algunas pérdidas de datos, el modelo de servicio de TCP no se adapta demasiado bien a las necesidades de este tipo de aplicaciones.
- **Sin establecimiento de la conexión:**
    - UDP no añade ningún retardo a causa del establecimiento de una conexión para la transferencia de datos. Probablemente, esta es la razón principal por la que DNS opera sobre UDP y no sobre TCP.
- **Sin información del estado de la conexión:**
    - TCP mantiene información acerca del estado de la conexión en los sistemas terminales, cosa que en UDP no pasa. Por esta razón, un servidor dedicado a una aplicación concreta suele poder soportar más clientes activos cuando la aplicación se ejecuta sobre UDP que cuando lo hace sobre TCP.
- **Poca sobrecarga debida a la cabecera de los paquetes:**

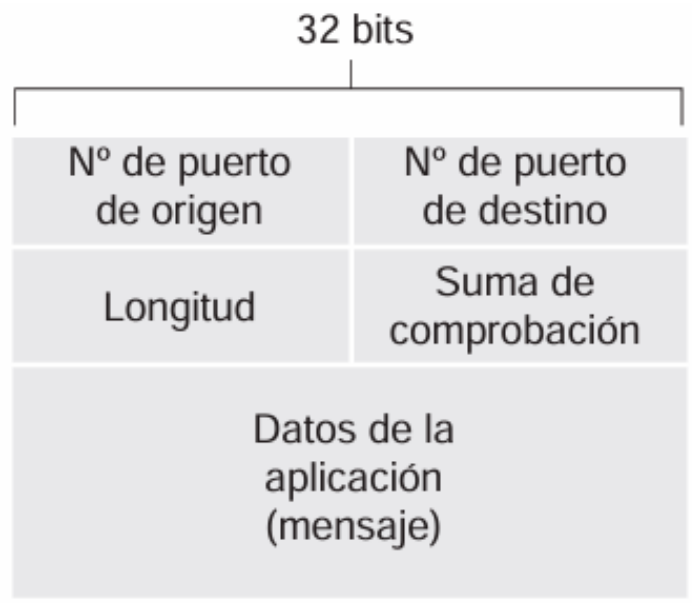
    - Los segmentos TCP contienen 20 bytes en la cabecera de cada segmento, mientras que UDP solo requiere 8 bytes.

Header UDP

- La cabecera UDP solo tiene **cuatro campos**, cada uno con una **longitud de dos bytes:**
    - **Números de puerto de origen y de destino:** Permiten al host de destino pasar los datos de la aplicación al proceso apropiado que está ejecutándose en el sistema terminal de destino (demultiplexación).
    - **Longitud:** Especifica el número de bytes del segmento UDP (la cabecera más los datos). Es necesario un valor de longitud explícito ya que el tamaño del campo de datos puede variar de un segmento UDP al siguiente.
    - **Suma de Comprobación/Checksum:** El host receptor utiliza la suma de comprobación para detectar si se han introducido errores en el segmento.
- Provee **Multiplexación** y **Demultiplexación** de aplicación y **detección de errores** (no obligatorio).

Estructura de los segmentos UDP

- Un segmento/datagrama UDP se conforma de la siguiente manera:
    - **Una cabecera UDP.**
    - **Datos de la aplicación (mensaje):** Estos datos ocupan el campo de datos del segmento UDP.



Estructura de un segmento UDP

Checksum

- **La suma de comprobación de UDP proporciona un mecanismo de detección de errores.** Es decir, se utiliza para determinar si los bits contenidos en el segmento UDP han sido alterados según se desplazaban desde el origen hasta el destino.
- UDP en el **lado del emisor** calcula el **complemento a 1 de la suma de todas las palabras de 16 bits del segmento**, acarreando cualquier desbordamiento obtenido durante la operación de suma sobre el bit de menor peso. Este resultado se almacena en el campo **suma de comprobación del segmento UDP**.
- En el **receptor**, las cuatro palabras de 16 bits se suman, incluyendo la suma de comprobación. **Si no se han introducido errores en el paquete**, entonces la suma en el receptor tiene que ser 1111111111111111. **Si uno de los bits es un 0, entonces sabemos que el paquete contiene errores.**
- UDP proporciona una suma de comprobación ya que no existe ninguna garantía de que todos los **enlaces** existentes entre el origen y el destino proporcionen un **mecanismo de comprobación de errores**; es decir, uno de los enlaces puede utilizar un **protocolo de la capa de enlace** que **no** proporcione comprobación de errores. Además, incluso si los segmentos se transfieren correctamente a través del enlace, **es posible que se introduzcan errores de bit cuando un segmento se almacena en la memoria de un router.** Dado que **no están garantizadas ni la fiabilidad enlace a enlace, ni la detección de errores durante el almacenamiento en memoria,** UDP tiene que proporcionar un mecanismo de detección de errores en la capa de transporte, **terminal a terminal**, si el servicio de transferencia de datos terminal a terminal ha de proporcionar la de detección de errores.

TCP - Transport Control Protocol

- Protocolo fiable, ordenado, Full Duplex, con buffering, control de errores, **control de flujo y control de congestión.**
- Orientado a **Streams**.
- El **PDU** utilizado en este protocolo se denomina **segmento**.
- Incrementa el **Overhead** en la **comunicación** para ofrecer **confiabilidad** en la misma.
- Provee servicios de **Multiplexación** y **Demultiplexación**.
- Requiere el **establecimiento** y el **cierre** de **conexión**.
- Tiene aplicaciones en transferencia de archivos, FTP/HTTP/SMTP/acceso remoto(SSH, telnet,...)/Unicast.
- TCP entrega y envía los datos agrupados o separados de forma **dis-asociada de la aplicación,** es decir, quizás la aplicación puede enviar 3000 bytes en un write y TCP lo podría enviar en 3 segmentos separados de 1000 bytes c/u. **(Basado el TCP).**

Header TCP

- La cabecera TCP es de 20 bytes (12 más que la de UDP) e incluye los siguientes campos:

    - **Número de puerto de origen y de destino:** Se utilizan para multiplexar y demultiplexar los datos de y para las aplicaciones de la capa superior.
    - **Suma de comprobación/Checksum.**
    - **Número de secuencia y Número de reconocimiento:** Ambos campos de 32 bits, son utilizados por el emisor y el receptor de TCP para implementar un servicio de transferencia de datos fiable.
    - **Ventana de recepción:** Campo de 16 bits que se utiliza para el control de flujo (se emplea para indicar el número de bytes que un receptor está dispuesto a aceptar).
    - **Longitud de cabecera:** Campo de 4 bits que especifica la longitud de la cabecera TCP en palabras de 32 bits. La cabecera TCP puede tener una longitud variable a causa del **campo opciones de TCP** (normalmente, este campo está vacío, por lo que la longitud de una cabecera TCP típica es de 20 bytes).
    - **Opciones:** Campo opcional y de longitud variable. Se utiliza cuando un emisor y un receptor negocian el **tamaño máximo de segmento (MSS)** o como un factor de escala de la ventana en las redes de alta velocidad.
    - **Indicador:** Campo de 6 bits → **En la práctica el PSH, URG no se utilizan:**
        - El **bit ACK** se utiliza para indicar que el valor transportado en el campo de reconocimiento es válido; es decir, el segmento contiene un reconocimiento para un segmento que ha sido recibido correctamente.
        - Los **bits RST, SYN y FIN** se utilizan para el establecimiento y cierre de conexiones.
        - Los **bits CWR y ECE** se emplean en la notificación de congestión explícita.
        - La activación del **bit PSH** indica que el receptor deberá pasar los datos a la capa superior de forma inmediata.
        - El **bit URG** se utiliza para indicar que hay datos en este segmento que la entidad de la capa superior del lado emisor ha marcado como “urgentes”.
    - La posición de este último byte de estos datos urgentes se indica mediante el **campo puntero de datos urgentes** de 16 bits. TCP tiene que informar a la entidad de la capa superior del lado receptor si existen datos urgentes y pasarle un puntero a la posición donde finalizan los datos urgentes → **En la práctica no se usa.**
- El encabezado TCP provee: Multiplexación, Demultiplexación, detección de errores, sesiones, control de errores, control de flujo y control de congestión.

Números de secuencia y números de reconocimiento

- Son una parte crítica de TCP. Este protocolo percibe los datos como un **flujo de bytes no estructurado pero ordenado**. El uso que hace TCP de los números de secuencia refleja este punto de vista, en el sentido de que **los números de secuencia hacen referencia al flujo de bytes transmitido y no a la serie de segmentos transmitidos**.

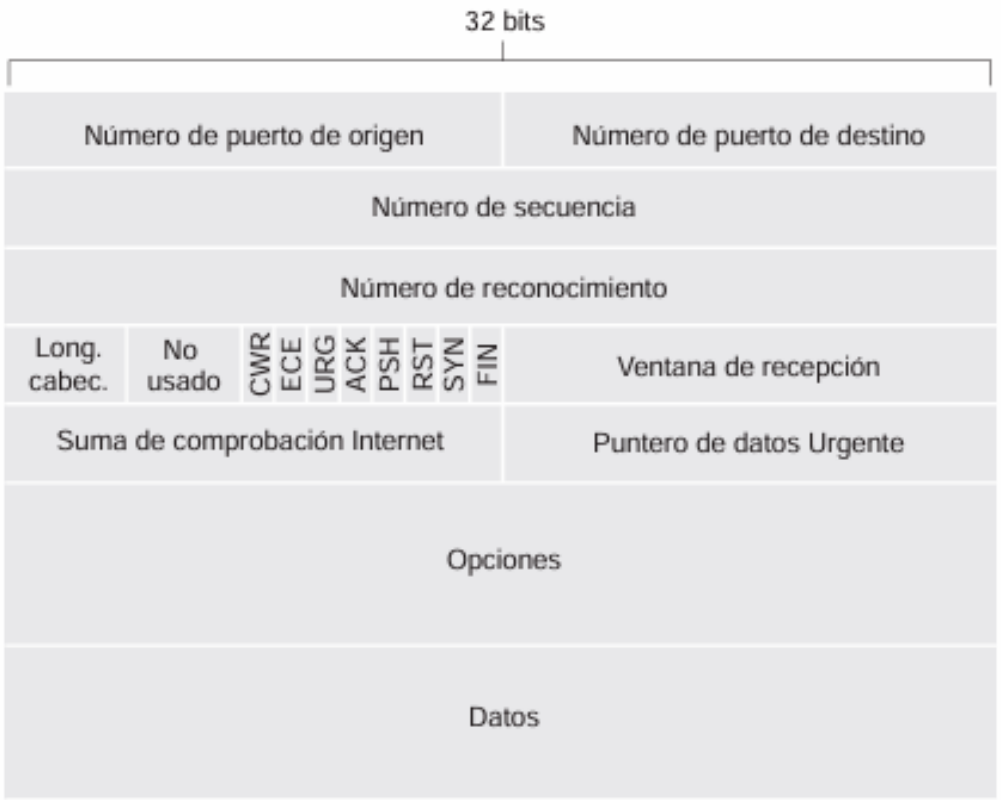
- El **número de secuencia de un segmento** es por tanto el número del primer byte del segmento dentro del flujo de bytes. Este campo es utilizado para mantener un seguimiento del orden de los segmentos TCP en una comunicación. **Cada segmento TCP se etiqueta con un número de secuencia único.**
- El **número de reconocimiento** que un host A incluye en su segmento es el número de secuencia del siguiente byte que el host A está esperando del host B en una conexión TCP. Este campo ayuda a establecer que los datos se han recibido de manera confiable.

Checksum

- **El cálculo de Checksum se da de la misma forma que en UDP pero para TCP es obligatorio.**
- Si se descubre un error, TCP podría pedir una **retransmisión** en la que se descarta el segmento con error y se espera el vencimiento del **RTO (Retransmission Timer).**
    - **En el lado del remitente**, si este timer se vence sin haber recibido un ACK eso quiere decir que el segmento no llegó y se procede a transmitirlo nuevamente.
    - Alternativamente, si el **remitente** recibe múltiples ACKs duplicados (tres ACKs duplicados para el mismo número de secuencia), se interpreta como una pérdida de segmento, y se desencadena una retransmisión rápida (sin esperar a que el temporizador expire).
    - **En el lado del receptor,** se mantienen los bytes no ordenados (si es que se recibieron bien algunos) y se espera a que lleguen los bytes que faltan con el fin de rellenar los huecos.

Estructura del segmento TCP

- El segmento TCP consta de **campos de cabecera** y un **campo de datos.**
    - El **campo de datos** contiene un fragmento de los datos de la aplicación. El **MSS** limita el tamaño máximo del campo de datos de un segmento. Si se enviara un archivo muy grande, este se vería dividido en varios fragmentos de tamaño **MSS.**



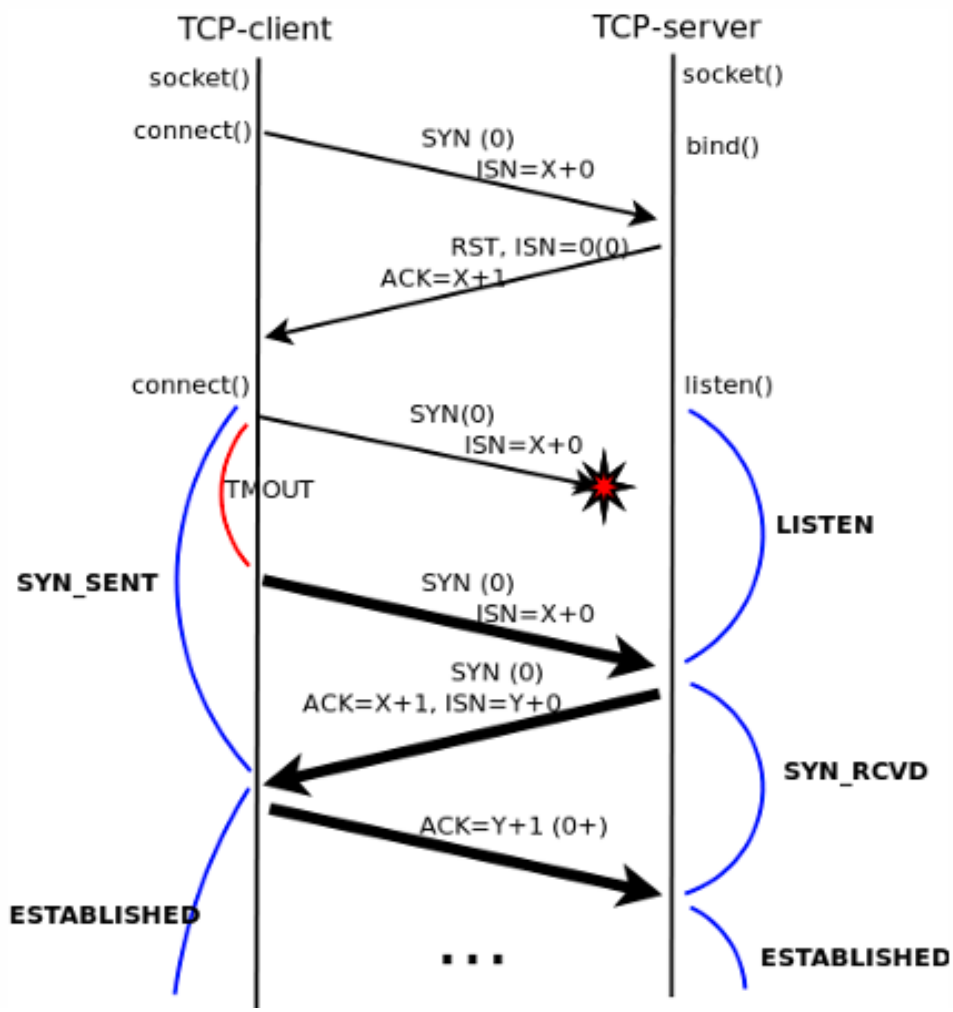
Estructura del segmento TCP

Conexión TCP

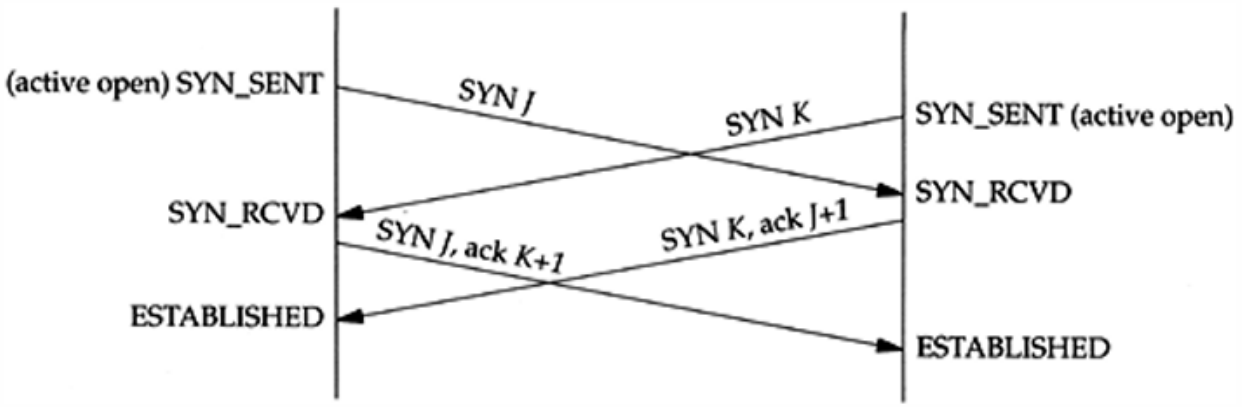
- Se dice que TCP está **orientado a la conexión** porque antes de que un proceso de la capa de aplicación pueda comenzar a enviar datos a otros, los dos procesos deben primero "establecer una comunicación" entre ellos; es decir, tienen que enviarse ciertos segmentos preliminares para definir los parámetros de la transferencia de datos que van a llevar a cabo a continuación.
- **La "conexión" TCP no es un circuito terminal a terminal** con multiplexación TDM o FDM. En su lugar, la "conexión" es una **conexión lógica**, con un estado común que reside solo en los niveles TCP de los dos sistemas terminales que se comunican.
- Al ejecutarse en sistemas terminales, los elementos intermedios de la red como los routers no mantienen el estado de la **conexión TCP.** Los routers intermedios son completamente inconscientes de las conexiones TCP; **los routers ven los datagramas, no las conexiones.**
- Una **conexión TCP** casi siempre es una **conexión punto a punto**, es decir, entre un único emisor y un único receptor. La "multidifusión", es decir, la transferencia de datos desde un emisor a muchos receptores en una única operación de envío, no es posible con TCP. **Con TCP, dos hosts son compañía y tres multitud. (Once again basadísimo mi amigo el TCP).**

Establecer la conexión - Three Way Handshake

- El three-way handshake es el proceso que TCP utiliza para establecer una conexión fiable entre el cliente y el servidor. **Este proceso garantiza que ambos extremos estén listos para recibir y enviar datos.**
- Durante este proceso se intercambian segmentos **SYN** y **ACK** antes de que comience la comunicación de datos real.
    1. **Paso 1 (SYN)**
        - El **cliente** envía un segmento con el **bit SYN activado** y el **número de secuencia inicial (ISN),** este último es un número aleatorio que servirá como punto de partida para la transmisión de datos, además, ayuda a rastrear y ordenar los segmentos de datos enviados en la conexión.
    2. **Paso 2 (SYN-ACK)**
        - El **servidor** recibe el **segmento SYN** y responde con un segmento que tiene **los bits SYN y ACK activados**, a su vez, el servidor también envía su propio **número de secuencia inicial (ISN).**
        - La parte **ACK** del segmento enviado contiene el **número de secuencia del cliente incrementado en uno (ISN del cliente + 1)** para indicar que se recibió correctamente el **SYN del cliente**.
        - Se envía **RST** si no hay proceso en estado **LISTEN**.
    3. **Paso 3 (ACK)**
        - Cuando el **cliente** recibe el **segmento SYN-ACK del servidor**, responde con un segmento que tiene el **bit ACK activado**. **En este segmento ya se puede enviar información.**
        - La parte **ACK** del segmento enviado contiene el **número de secuencia del servidor incrementado en uno (ISN del servidor + 1)** para indicar que se recibió correctamente el **SYN-ACK del servidor.**
        - En este punto, **ambos lados han confirmado la conexión, y la conexión TCP se considera establecida.**



Three Way Handshake



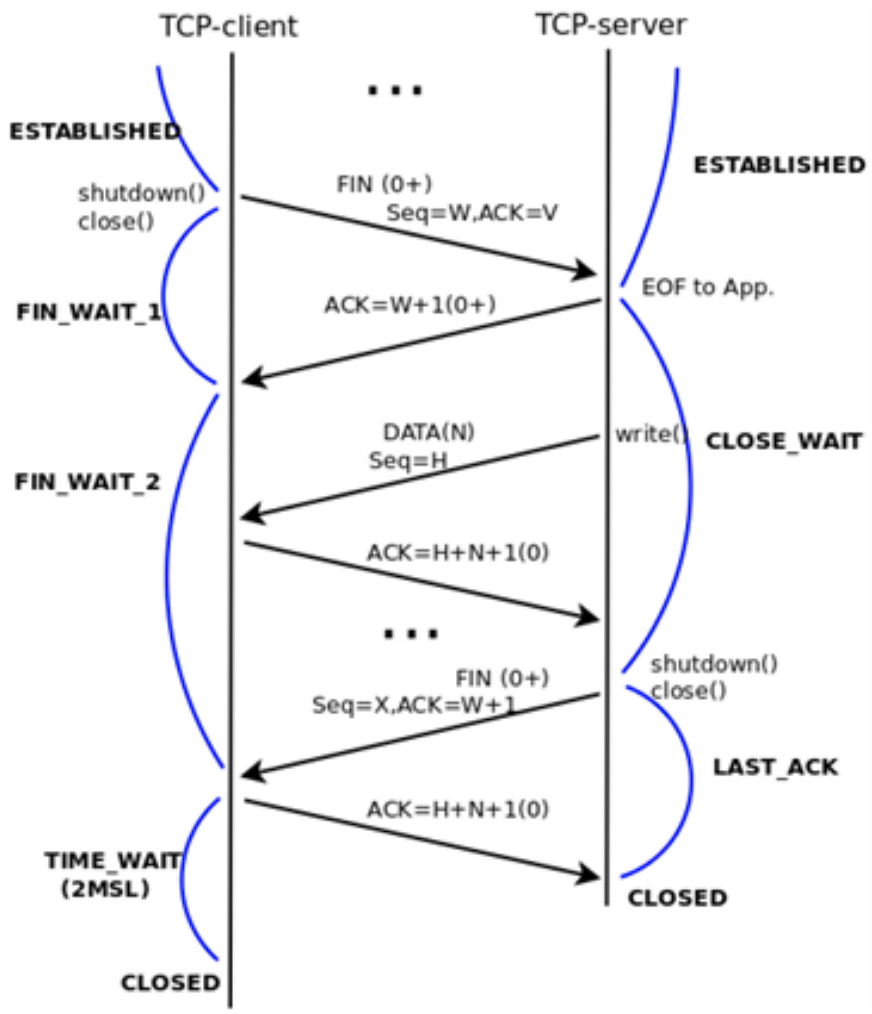
Three Way Handshake con Open Simultáneo

Cierre de Conexión

- El cierre de una conexión TCP se puede dar de varias formas.

Four Way Handshake

- Este es el proceso que usa TCP para cerrar una conexión de manera **ordenada**, permitiendo que **ambos extremos dejen de enviar datos y aseguren que toda la información ha sido transmitida correctamente.**
- Es el estándar.
    1. **Paso 1 (FIN)**
        - Cuando uno de los extremos (por ejemplo el cliente) desea terminar la conexión, envía un segmento con el **bit FIN** activado al otro extremo, indicando de esta forma que no tiene más datos para enviar.
    2. **Paso 2 (ACK)**
        - En este caso, el **servidor** recibe ese segmento y responde con un segmento con el **bit ACK activado**, confirmando al **cliente** que ha recibido la solicitud de cierre de conexión.
        - **En este punto el servidor podría seguir enviando datos, pero el cliente ha terminado su envío.**
    3. **Paso 3 (FIN)**
        - Cuando el servidor termina de enviar datos, envía su propio segmento con el **bit FIN activado** al **cliente**, indicando que también desea cerrar la conexión.
    4. **Paso 4 (ACK)**
        - Cuando el **cliente** recibe el segmento **FIN** del **servidor**, responde con un último segmento con el **bit ACK activado**, confirmando que recibió el mensaje de cierre de parte del servidor.
        - Después de enviar este último segmento el **cliente** entra en un **estado** llamado **TIME-WAIT** durante unos segundos. Si no recibe un segmento duplicado de **FIN** durante este tiempo, considera que la conexión ha terminado y abandona el estado **TIME-WAIT.**
        - **En este momento la conexión TCP se considera Cerrada.**



Four Way Handshake

Cierre abrupto con el bit RST

- En lugar de usar el bit **FIN** para una finalización ordenada, un extremo puede enviar un segmento con el bit **RST** activado. Esto indica que la conexión debe cerrarse **inmediatamente**, sin el proceso de cierre habitual. Cuando se recibe un **RST**, el otro extremo cierra la conexión de inmediato, sin enviar **ACKs** ni esperar confirmaciones.

Cierre por Timeout

- Si un extremo de la conexión TCP no recibe ninguna actividad (datos o ACKs) durante un tiempo prolongado, puede cerrar la conexión por **timeout**. Este mecanismo de cierre es automático y depende de configuraciones como el **TCP Keepalive**.

Cierre Half-Close

- TCP permite que un extremo cierre **solo su lado de la conexión** (usando un mensaje FIN) mientras permite que el otro extremo siga enviando datos. Esto significa que una de las partes ha terminado de enviar datos pero sigue aceptando datos entrantes.

Estados de una conexión TCP

- **CLOSED**: Es la ausencia total de una conexión activa.
- **LISTEN**: Este estado representa la espera de una solicitud de conexión precedente de cualquier TCP remoto y puerto.
- **SYN_SENT**: Representa la espera de una solicitud de conexión coincidente después de haber enviado una solicitud de conexión.
- **SYN-RECEIVED**: Representa la espera de una confirmación de solicitud de conexión después de haber recibido y enviado una solicitud de conexión.
- **ESTABLISHED**: Representa una conexión abierta, lo que permite que los datos recibidos sean entregados al usuario. Es el estado normal durante la fase de transferencia de datos de la conexión.
- **FIN-WAIT-1**: Representa la espera de una solicitud de terminación de conexión procedente del TCP remoto, o la confirmación de la solicitud de terminación de conexión previamente enviada.
- **FIN-WAIT-2**: Representa la espera de una solicitud de terminación de conexión procedente del TCP remoto.
- **CLOSE-WAIT**: Representa la espera de una solicitud de terminación de conexión procedente del usuario local.
- **CLOSING**: Representa la espera de una confirmación de la solicitud de terminación de conexión procedente del TCP remoto.
- **LAST-ACK**: Representa la espera de una confirmación de la solicitud de terminación de conexión previamente enviada al TCP remoto (la cual incluye una confirmación de su solicitud de terminación de conexión).
- **TIME-WAIT**: Representa la espera de suficiente tiempo para asegurar que el TCP remoto ha recibido la confirmación de su solicitud de terminación de conexión.

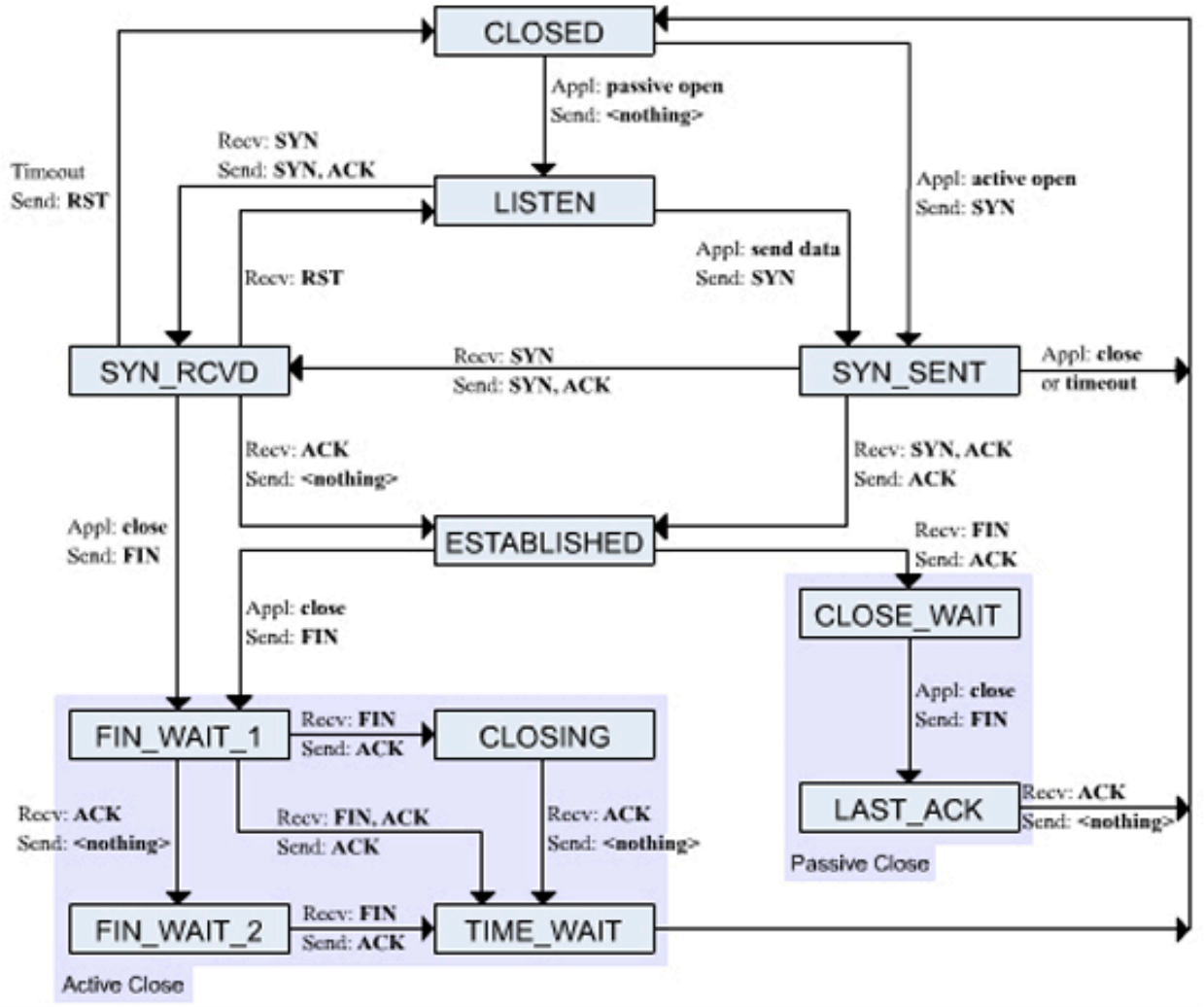
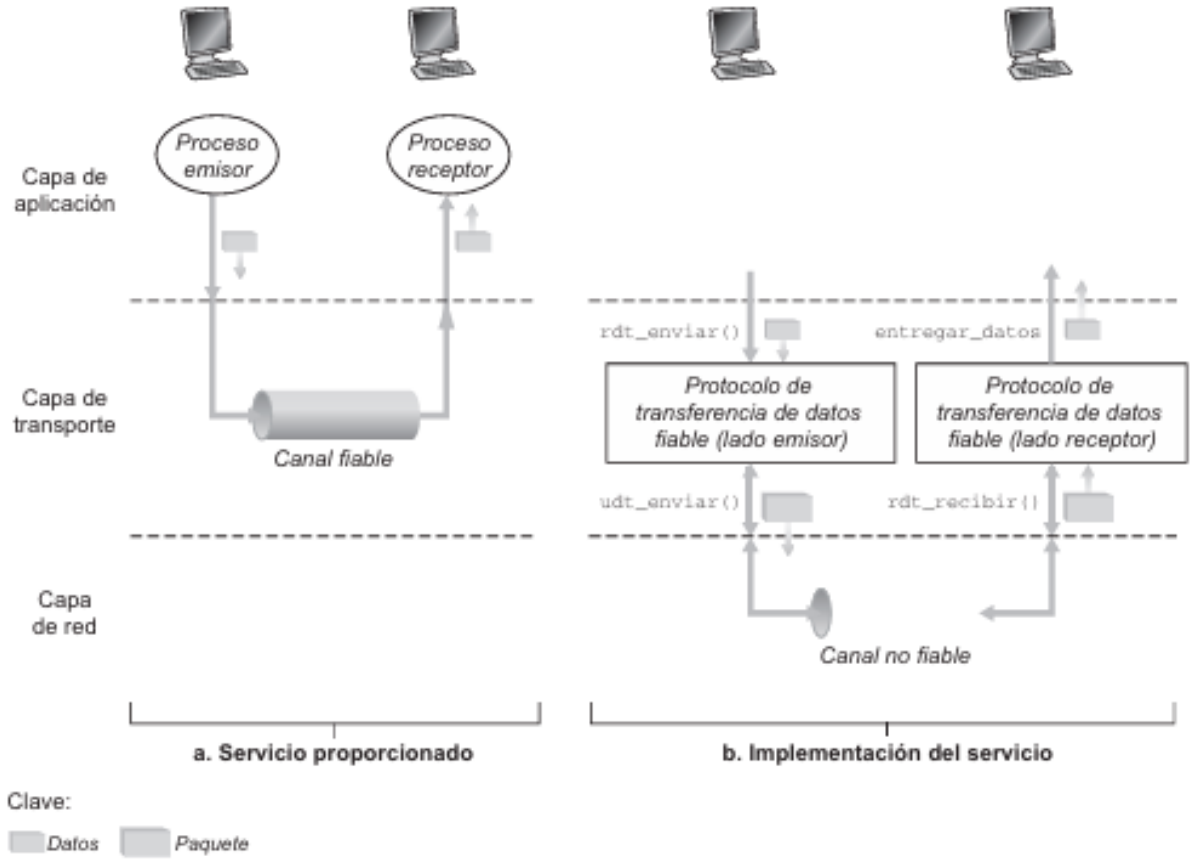


Diagrama de Estados de una Conexión TCP

Control de Errores

- La necesidad de una transferencia de datos fiable es evidente y no solo afecta a la capa de transporte, también afecta a la capa de enlace y a la de aplicación. Si se transportan los datos en un **canal fiable**, no hay riesgo de pérdida o corrupción de la información. Si se posee un **canal no fiable** (por ejemplo IP) se requiere un **mecanismo de control** que brinde una **abstracción** del servicio (en este caso el ofrecido por TCP) proporcionando a las entidades de la capa superior (aplicaciones de Internet que lo invocan) un canal fiable para la transferencia de datos.



Transferencia de datos fiable

- El control de errores se considera un mecanismo protocolar, algoritmo, que permite ordenar los **segmentos que llegan fuera de orden** y recuperarse mediante **solicitudes** y/o **retransmisiones** de aquellos segmentos perdidos o con errores.
- Tiene como **objetivo** recuperarse de los efectos del **reordenamiento**, la **pérdida** o la **corrupción** de los paquetes en la red.
- Se realiza por cada **conexión**: End-to-End, App-to-App.
- Los protocolos de transferencia de datos fiables basados en **retransmisiones** se conocen como **protocolos ARQ (Automatic Repeat reQuest, Solicitud automática de repetición).** Estos protocolos se basan en el uso de:
    - **Números de Secuencia** que identifiquen a los segmentos de datos y permitan al receptor identificar su orden.
    - **Confirmaciones (ACK)** que se envían para validar que los datos se recibieron en orden y sin errores, y además indicar el número de secuencia del siguiente segmento esperado.

Componentes clave del control de errores

RTT (Round Trip Time)

- Representa el tiempo que tarda un paquete de datos en viajar desde el emisor hasta el receptor y volver con la confirmación (ACK). En otras palabras, es la **suma del**

tiempo de transmisión, el tiempo de propagación en la red y el tiempo de procesamiento en el receptor. Se calcula a medida que se envían datos y se reciben ACK.
- Es un factor **fundamental** para **determinar el valor del RTO.**
- **Un RTT alto** puede afectar negativamente el rendimiento del control de errores, especialmente en conexiones de alta velocidad y con un gran ancho de banda.

TimeStamp

- La opción de TimeStamp permite una **estimación más precisa del RTT al incluir marcas de tiempo en los segmentos TCP (Timestamp Value TSval).**
    - Los valores **TSval** se repiten en el lado opuesto de la conexión en el campo **Timestamp Echo Reply TSecr** . Entonces, cuando se confirma un segmento, el remitente de ese segmento puede simplemente restar su marca de tiempo actual del valor **TSecr** para **calcular una medición precisa del tiempo de ida y vuelta (RTT) → RTT = TSecr - TSval.**
- **Beneficios:**
    - Cálculo de RTT sin temporizadores individuales por segmento.
    - Protección contra Wraparounds de números de secuencia (PAWS).

Timer RTO - Retransmission Timeout

- Controla el tiempo de espera para la recepción de una confirmación (ACK). Si el emisor no recibe el ACK dentro del RTO, retransmite el segmento.
- **Un RTO adecuado es crucial para el rendimiento del protocolo de control de errores:**
    - **Si es muy corto** se pueden producir retransmisiones innecesarias, incluso si el segmento original llegó correctamente al receptor, pero el ACK se retrasó. De esta manera hacemos un uso ineficiente del ancho de banda.
    - **Si es muy largo** el emisor esperará demasiado tiempo para retransmitir un segmento perdido, aumentando la latencia y afectando la velocidad de transmisión.

Cálculo del RTO dinámico

- **Para un control de errores eficiente, el RTO debe ser dinámico y adaptarse a las condiciones de la red.** El cálculo del RTO se basa en la estimación del RTT, que puede variar debido a la congestión, la distancia entre los nodos y otros factores. TCP utiliza un algoritmo para calcular el RTO dinámicamente considerando:
    - **SRTT (Smoothed Round Trip Time):** Un promedio ponderado del RTT que suaviza las fluctuaciones y se actualiza con cada nueva medición.
    - **DevRTT (Deviation of RTT):** Una medida de la variabilidad del RTT, que se utiliza para aumentar el margen del RTO en caso de fluctuaciones significativas.
- **La fórmula general para el RTO en TCP es: RTO = SRTT + (4 * DevRTT).**

Buffers de Transmisión (TxBuf) y Recepción (RxBuf)

- Buffers que almacenan los segmentos que están pendientes de transmisión o recepción, respectivamente.

Checksum/CRC

- Mecanismos para detectar errores de bits en los datos recibidos.

Tipos de errores en la capa de red

- En la capa de red se pueden varios errores que buscamos solucionar con los mecanismos de control de errores:
  - **Pérdida de datos o ACK:** Debido a descartes por congestión o errores en la red.
  - **Duplicación de datos o ACK:** Por retransmisiones o errores en los equipos.
  - **Desorden de segmentos:** Causado por el uso de rutas múltiples o errores en el procesamiento.
  - **Corrupción de datos o ACK:** Por errores de transmisión.
  - **Retrasos variables:** Fluctuaciones en el RTT.

Esquemas de control de errores

Stop & Wait (S&W)

- Un esquema **sencillo** aunque **ineficiente** donde el emisor envía **un solo segmento** y espera su confirmación antes de enviar el siguiente.
- No aprovecha el ancho de banda.
- **No es utilizado por TCP.**

Lado del emisor

- El emisor envía un segmento numerado y espera confirmación (ACK).
- Al enviar el segmento arranca un RTO:
  - **Si no recibe** ACK re-envía.
  - **Si recibe confirmación,** envía nuevo segmento, si tiene en el TxBuf , e inicia nuevo RTO (en caso que la capa superior dejó datos para enviar en el buffer).
- Descarta confirmaciones fuera de secuencia, es decir, atrasadas.

Lado del receptor

- Cada vez que recibe un segmento verifica su integridad con Checksum/CRC y confirma indicando el siguiente número de secuencia esperado. Puede confirmar el actual o el que se espera.
- Si el segmento recibido tiene errores, descarta o podría confirmar indicando qué número de secuencia espera. Funciona como **NAK (No Acknowledge → Una confirmación negativa).**

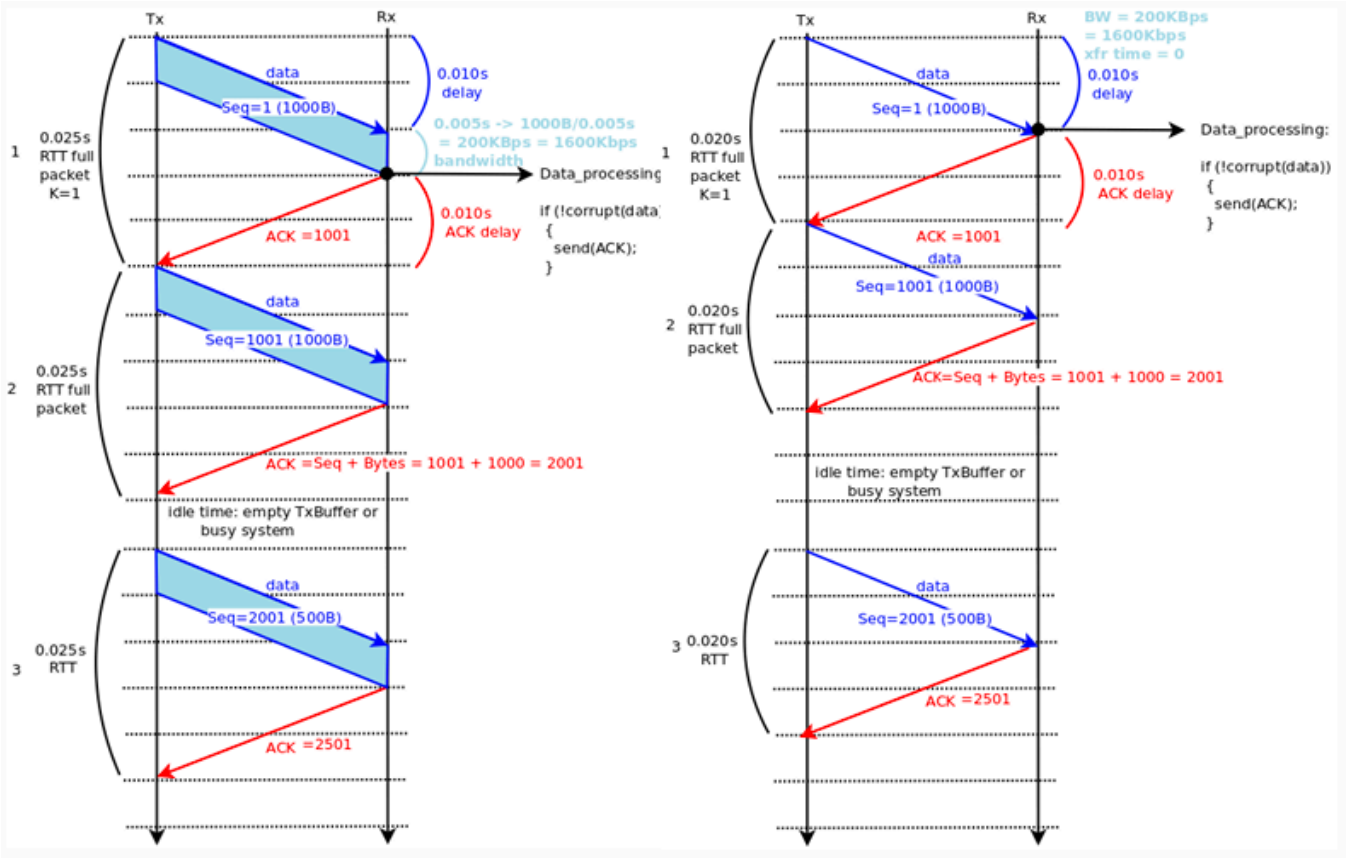


Gráfico de S&W

Pipelining/Sliding Window

- Para mejorar la **eficiencia**, el pipelining permite al emisor enviar **múltiples segmentos** de datos sin esperar la confirmación de cada uno. Se establece una "**ventana**" que define la cantidad de datos que pueden estar "en vuelo" (sin confirmar) al mismo tiempo.
- La **Ventana** se nota como **K** o **W, W =n, donde n >1.**
- Requiere **buffering** de **Tx**, **TxBuf** (lo que deja la capa superior y aún no se confirmó) y de **Rx**, **RxBuf** (lo que se va recibiendo hasta entregar a la capa superior).
- Requiere ampliar los números de segmentos y si se amplían los números de secuencia se brinda una mayor tolerancia a fallas ACK delayed.
- **Tiene 2 alternativas:**
  - **Go-back-N** (confirma en orden).
  - **Selective-Repeat** (confirma fuera de orden).

Lado del emisor

- Mantiene un buffer de transmisión (TxBuf) para los segmentos pendientes de confirmación.
- Envía segmentos de datos hasta llenar la ventana de transmisión e inicia un RTO para el segmento más antiguo sin confirmar.
  - Si el RTO expira o se reciben confirmaciones negativas (NAK) re-transmite.
  - Si se recibe una confirmación se descarta/reinicia el timer RTO.

- Desliza la ventana sobre el TxBuf a medida que se reciben confirmaciones para habilitar nuevos segmentos a transmitir.

Lado del receptor

- Mantiene un buffer de recepción (RxBuf) para los segmentos recibidos.
- A medida que va recibiendo segmentos, envía confirmaciones (ACK) al emisor, indicando el siguiente número de secuencia esperado.
- Si recibe segmentos duplicados, los descarta.
- Si recibe segmentos fuera de orden puede:
  - Descartarlos.
  - Almacenarlos temporalmente.

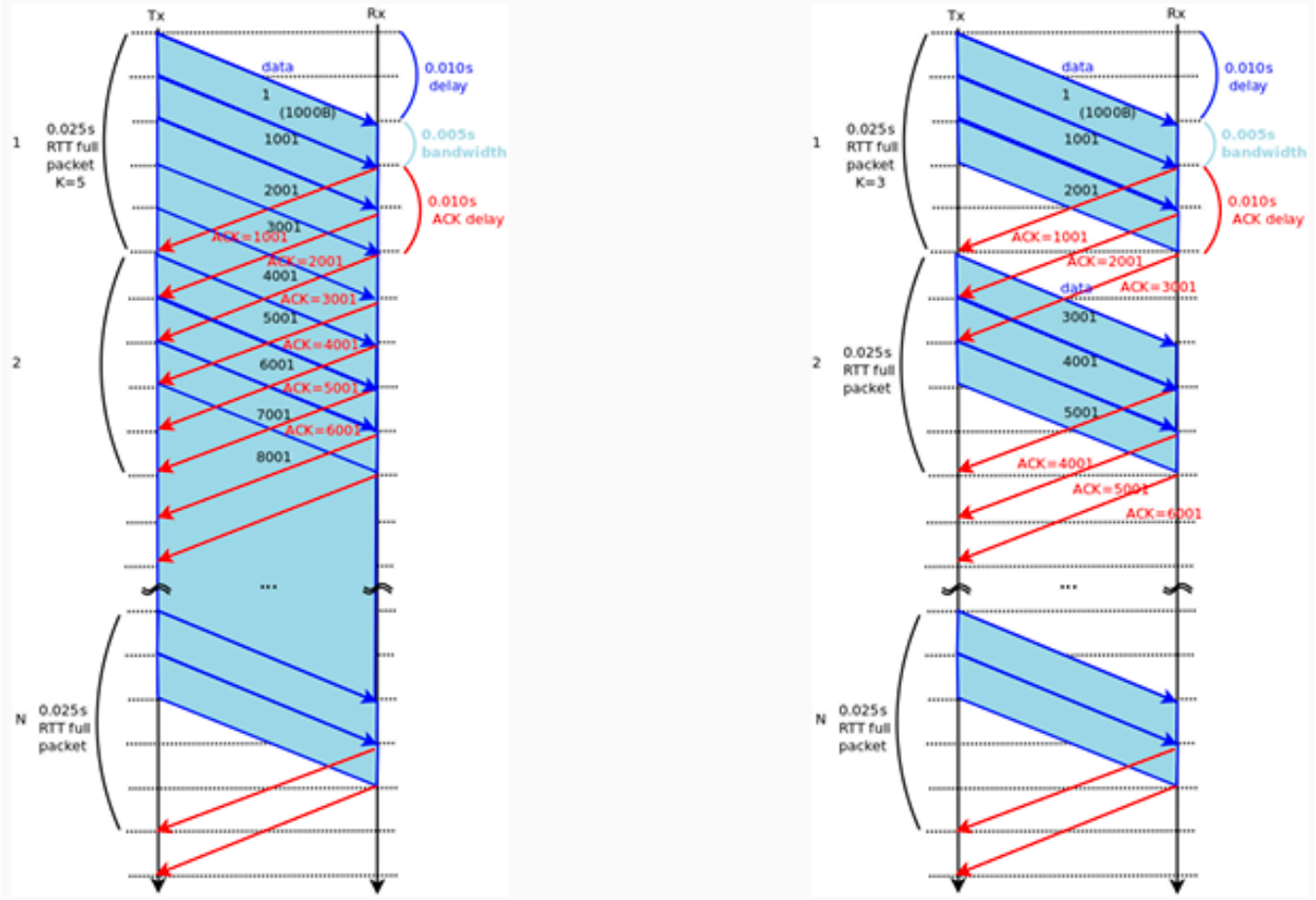


Gráfico de Pipelining

Go-back-N (GBN)

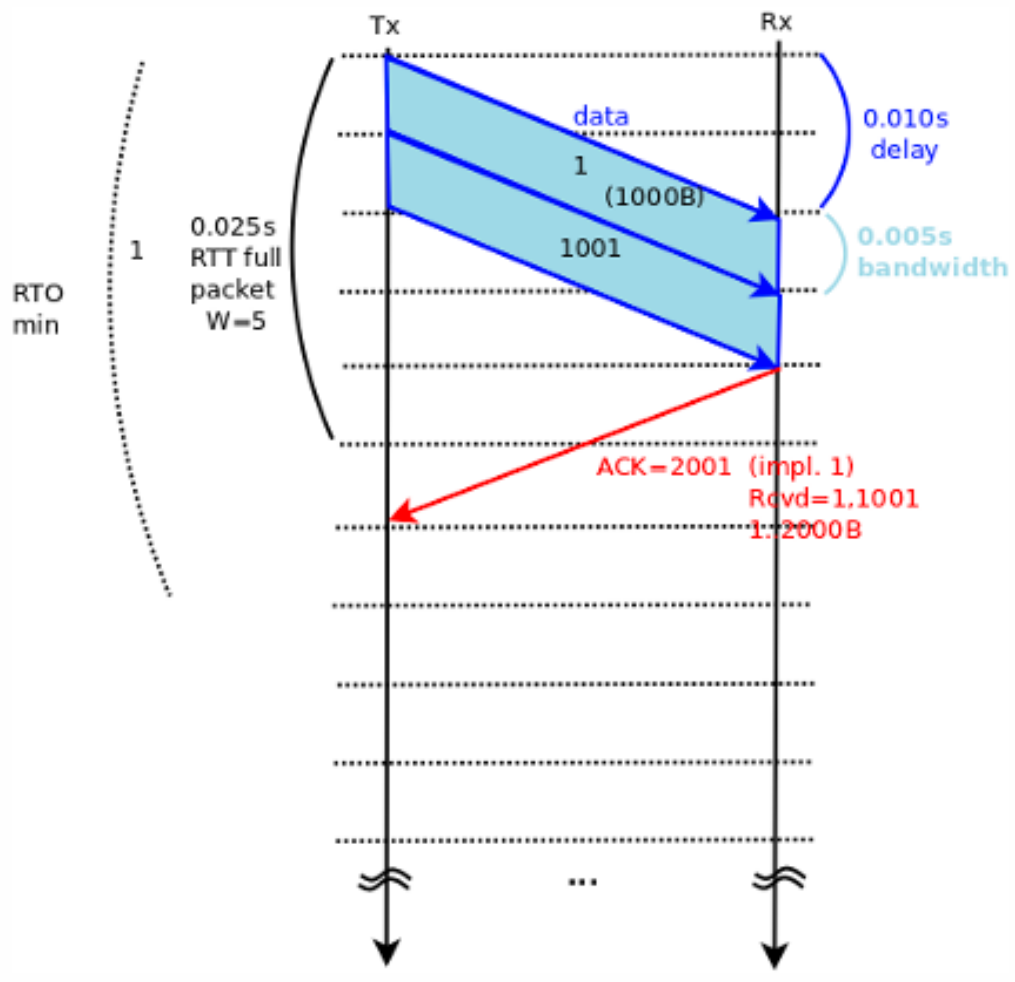
- Esquema de Sliding Window que requiere la **recepción de segmentos en orden**. Si un segmento se pierde, el **receptor** solo confirma hasta el último segmento recibido en orden, y el **emisor** retransmite todos los segmentos a partir del perdido, incluyendo aquellos que llegaron correctamente después.
- Se puede confirmar desde N hacia atrás **(ACK acumulativos)**. No necesariamente se confirma cada segmento individualmente.
- Se tiene una **ventana** "**estática**" de tamaño K = n, n > 1.

Lado del emisor

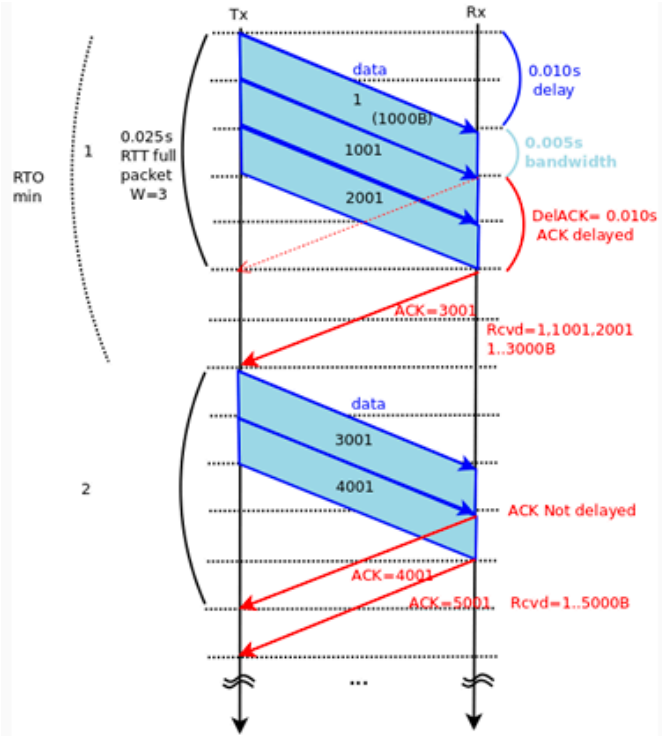
- Transmite segmentos si hay datos en el TxBuf y la ventana lo permite.
- Al enviar un segmento inicia un RTO.
    - Si recibe un ACK en orden desliza la ventana.
    - Si hay segmentos “en vuelo” reinicia el RTO, si no lo descarta.
    - Si no recibe ACK o expira el RTO re-transmite todos los segmentos a partir del más antiguo no confirmado.
- A diferencia de S&W, más de un segmento “in-flight”, más de un ACK “in-flight”.

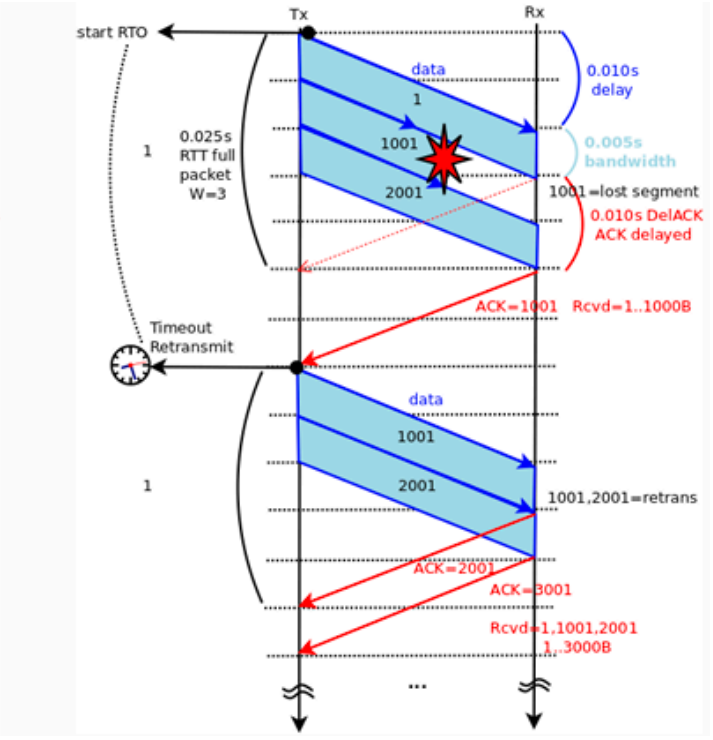
Lado del receptor

- Confirma positivamente (ACK) el **último segmento recibido en orden**, indicando el siguiente esperado.
- Descarta segmentos que estén corruptos o fuera de orden y espera la retransmisión, o, se puede confirmar por la negativa (NAK) indicando que se espera el próximo al último recibido de forma adecuada (Las confirmaciones por la negativa pueden generar ACK duplicados).
- Opcionalmente, puede almacenar temporalmente segmentos fuera de orden y confirmarlos cuando se complete la secuencia (requiere buffering de Rx antes de pasarlos a la capa usuaria).
- Se aprovechan los segmentos de datos para enviar confirmaciones, reduciendo el tráfico de control **(Piggy Backing).**
- Puede confirmar un rango de segmentos con un solo ACK **(Confirmaciones acumulativas).**

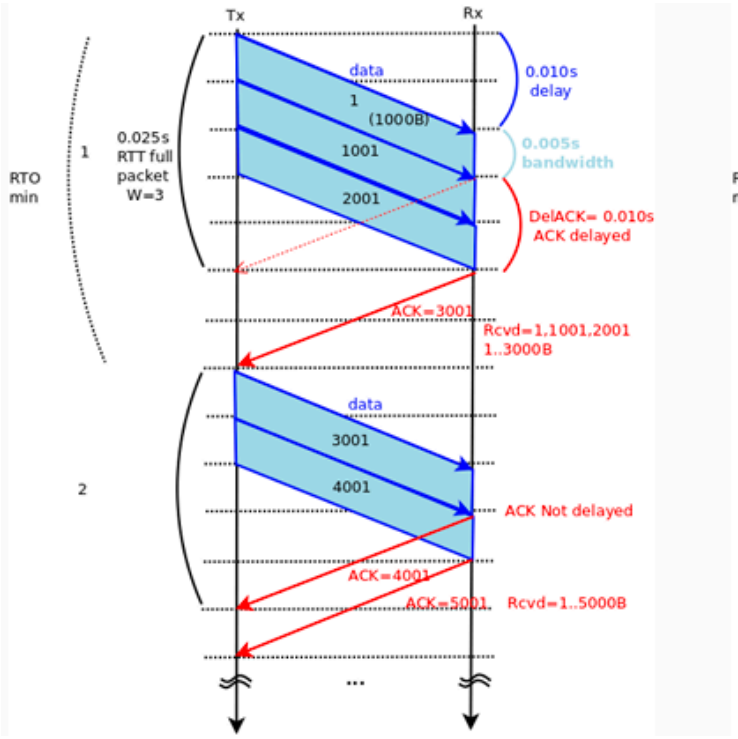


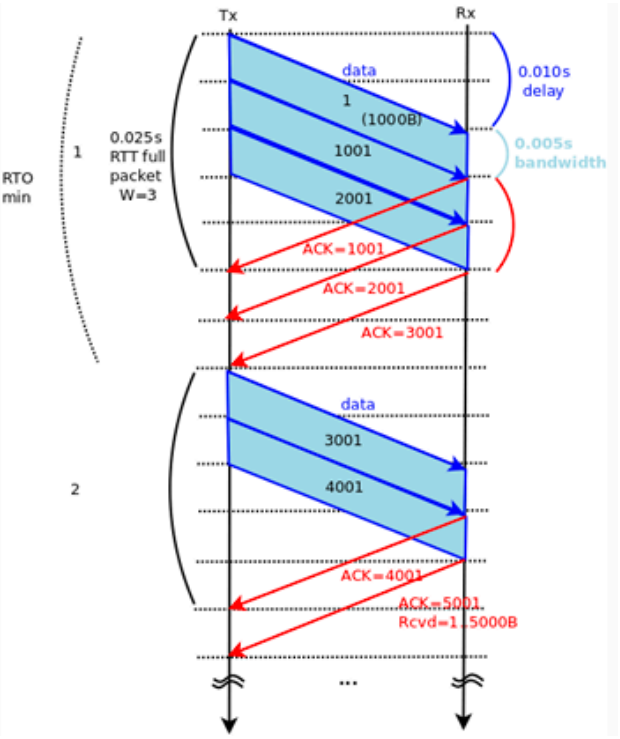
Ventana Envíos



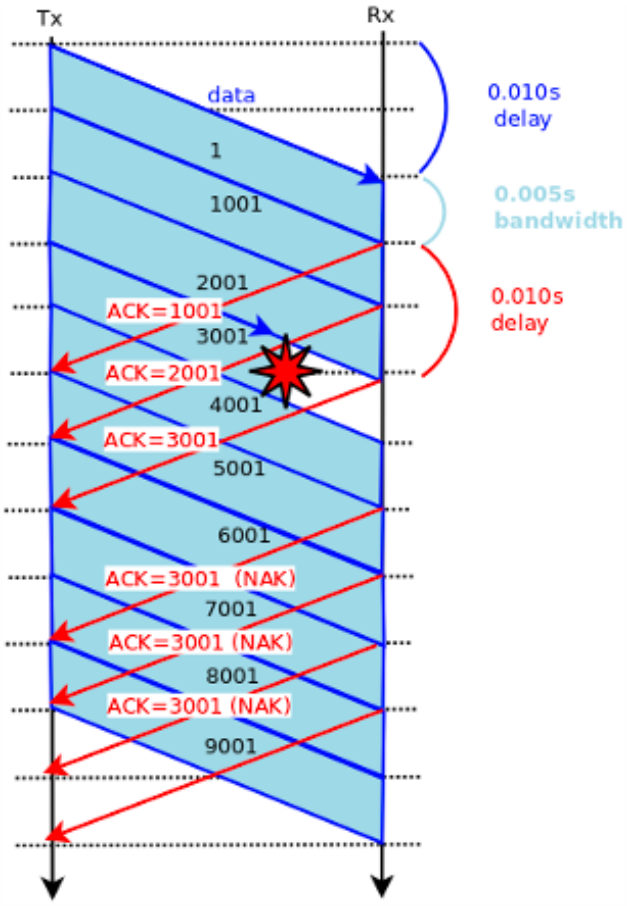


Segmento Perdido





ACK Acumulativos



Selective Repeat (SR)

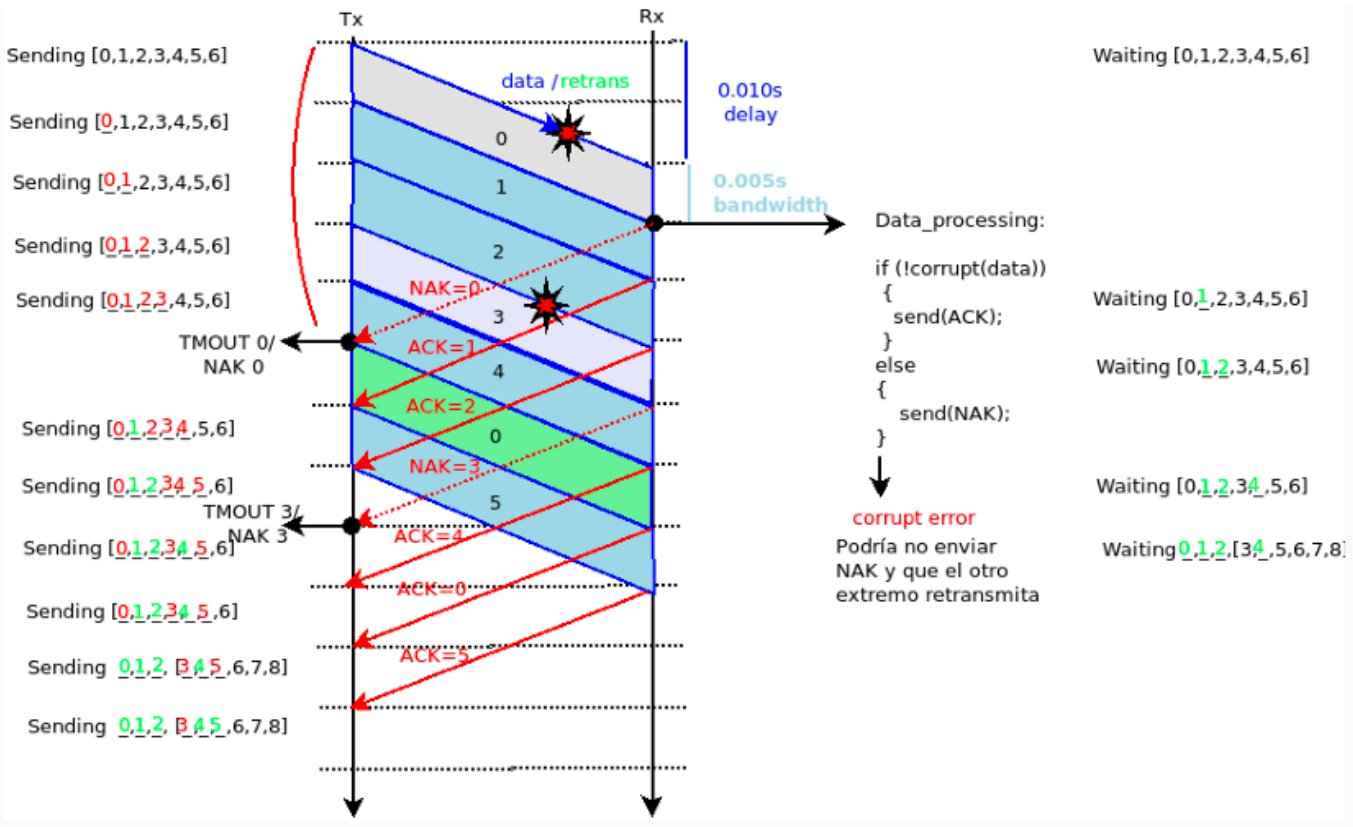
- A diferencia de Go-back-N, SR permite la **recepción de segmentos fuera de orden**. El receptor confirma individualmente cada segmento recibido correctamente, y el emisor sólo retransmite los segmentos que no se confirmaron dentro del RTO.
- No se deben confundir los segmentos de diferentes ráfagas. No se deben reusar #ID/SEQ hasta asegurarse que tiene todos los mensajes previos o estos no están en la red. Para esto el tamaño de la ventana no debe exceder la mitad del tamaño total del espacio de números de secuencia. La razón detrás de esta restricción es evitar la posibilidad de que un número de secuencia se reutilice antes de que el ACK correspondiente haya llegado, ya que la ventana se implementa como un buffer circular, entonces si fuese más grande podría haber paquetes representados por la misma posición en el buffer lo que podría llevar a confusiones en la correcta interpretación de los frames.
- Se realiza en módulo M, W <= (M−1) / 2 ,para evitar confundir los ACK de segmentos.
- La **ventana** se desliza sin dejar huecos, desde los confirmados más viejos.

Lado del emisor

- Transmite segmentos si hay datos en el TxBuf y la ventana lo permite.
- Inicia un RTO para cada segmento individual aunque se puede simular con un solo timer.
- Si recibe un ACK en orden desliza la ventana y detiene el RTO del segmento confirmado.
- Si recibe un ACK fuera de orden, no desliza la ventana, detiene el RTO para el segmento confirmado y lo marca como ya recibido.
- Si vence el RTO, re-envía el segmento asociado.

Lado del receptor

- Confirma individualmente cada segmento recibido dentro de la ventana esperada.
    - Si recibe segmento dentro de la ventana esperada y no lo tiene, lo almacena, si ya lo tiene, lo descarta. Siempre confirma. Requiere bufferear los segmentos RxBuf.
    - Puede confirmar de a uno o usar bit vectors/intervalos de confirmaciones.
    - No puede usar confirmaciones acumulativas.
- Almacena segmentos recibidos fuera de orden hasta que se complete la secuencia.
- Actualiza la ventana a medida que se reciben los segmentos en orden.



Ejemplo

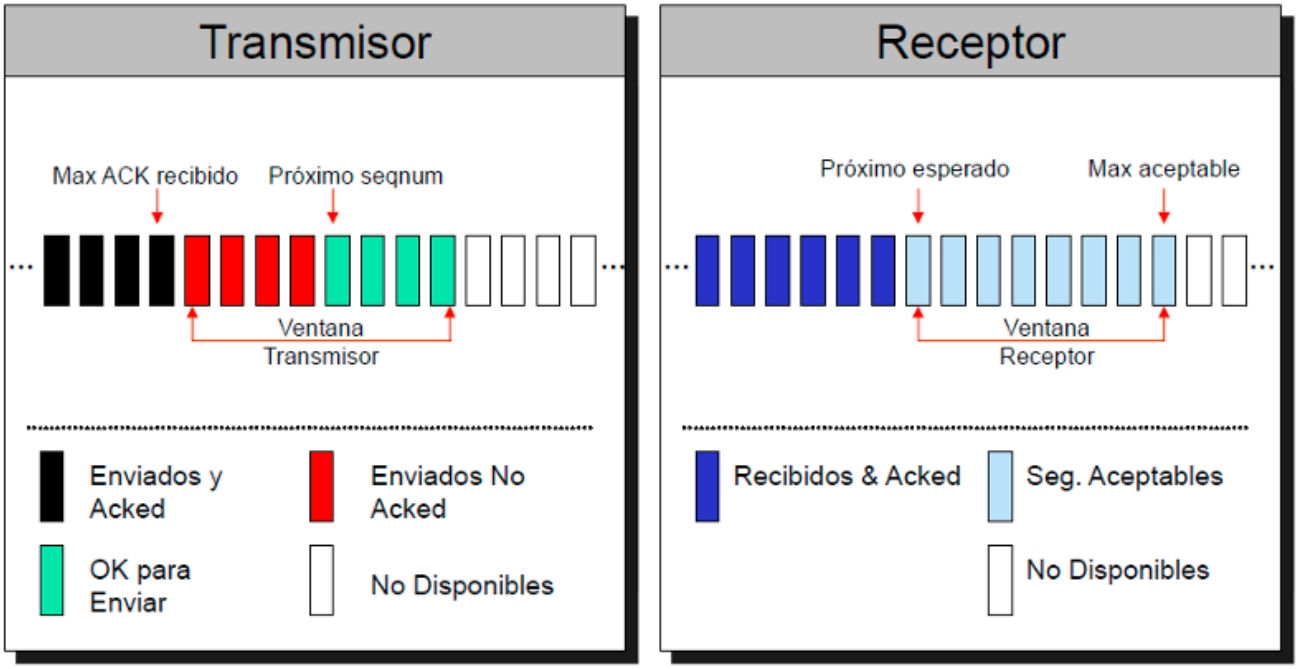
Control de errores en TCP

- Utiliza **Go-back-N (GBN) con ventana dinámica (flow-control)**, utiliza **piggy-backing** y permite **negociar Selective Repeat con opciones SACK (Selective ACK).**
- Mantiene el **control de errores por bytes** (byte oriented), no por segmentos.
- Los **segmentos** se numeran de acuerdo a **bytes enviados (nro. del primer byte) de datos dentro del segmento.**
- Los **números** se **negocian** al establecer la **sesión** y cada implementación los elige libremente (ISN).
- Las confirmaciones son "**anticipativas**", indican el número de byte que esperan.
- Utiliza los campos: **#SEQ, #ACK, flag ACK más timer y algunas opciones.**
- Segmentos ACked no indica leído por aplicación, sí recibido por TCP (ubicado en el **Rx Buffer** del receptor).
- Si el receptor detecta error en el segmento simplemente descarta y espera que expire RTO en el emisor (podría enviar un "NAK", re-enviar ACK para el último recibido en orden, forma de solicitar lo que falta-TCP requiere varios dups. ACKs-.
- Receptor con segmentos fuera de orden descarta directamente y podrá re-enviar ACK (podría dejar en **Rx Buffer** pero no entregar a la aplicación, tiene huecos).
- **Se puede confirmar con ACK acumulativos.**
- TCP NO arrancar un RTO por cada segmento, solo mantiene un por el más viejo enviado y no ACked y arranca uno nuevo solo si no hay RTO activo y hay segmentos en vuelo (in-flight).

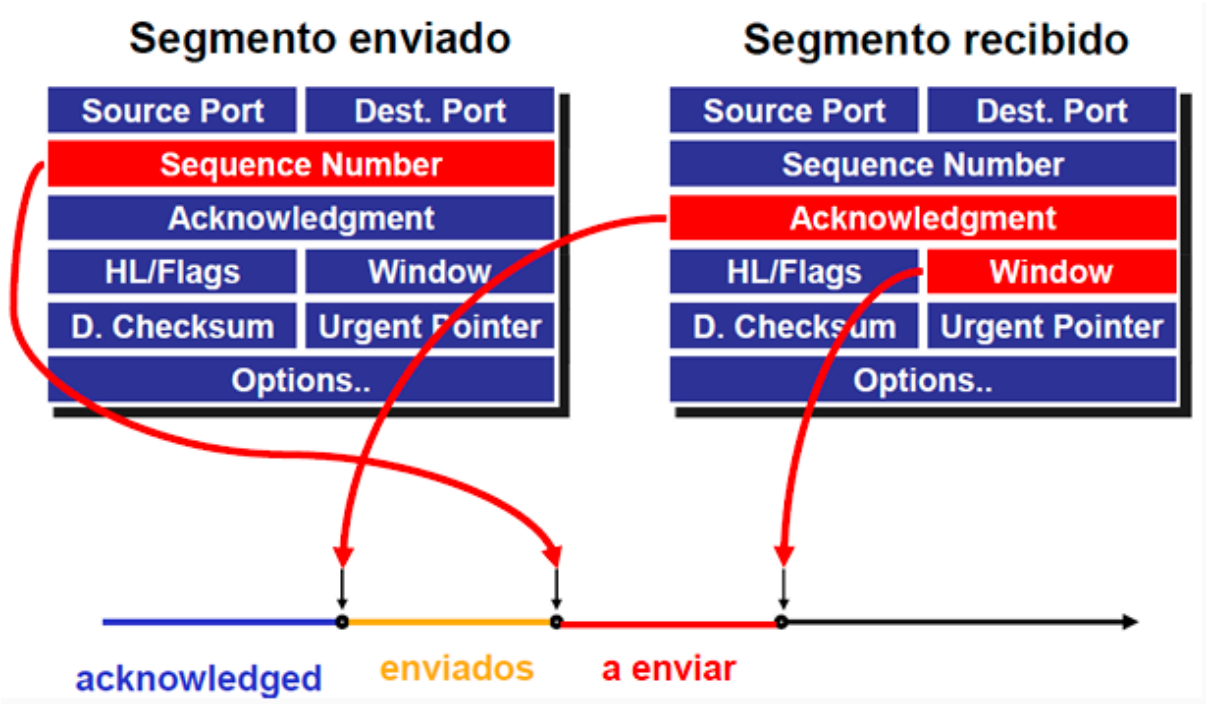
- Si se confirman (ACked) datos, se inicia un nuevo RTO recomendado. El nuevo RTO le está dando más tiempo al segmento más viejo aún no confirmado.
- Si vence un RTO se debe retransmitir el segmento más viejo no ACKed y se debe doblar: Back-off timer RTO = RTO ∗2 RTOMAX = 60s recomendado.
- **TCP calcula el RTO de forma dinámica, ayudado por Timestamp Option.**

Control de Flujo

- Mecanismo protocolar, algoritmo, que **permite al receptor controlar la tasa a la que le envía datos el transmisor.**
- Control cuanto puede enviar una aplicación sabiendo que la receptora tiene capacidad de recibirlo y procesarlo.
- **Objetivo**: prevenir que el emisor sobrecargue al receptor con datos evitando un mal uso de la red.
- Al igual que el **control de errores,** se implementa de **Extremo a Extremo.**
- El receptor (cada extremo puede recibir, es FDX) indica el espacio del buffer de recepción, **Rx Buffer**, en el campo del segmento: **Window** (de datos o ACK) **Advertised Window** (Ventana Anunciada) o **Ventana de Recepción**.
- Por cada segmento que envía indica el tamaño del buffer de recepción **Rx Buffer** (mbufs). **(Cada conexión mantiene su propio buffer)** en espacio del kernel (TCP).
- **Window** (Ventana) indica la cantidad de datos en bytes que el emisor le puede enviar sin esperar confirmación (mejora notablemente contra Stop & Wait).
    - La ventana de recepción de cada extremo es **independiente**.
- Cada vez que llega un segmento nuevo en orden es puesto por TCP en el **Rx Buffer**, TCP lo debe confirmar.
- Cada vez que la aplicación lee se hace espacio en el **Rx Buffer**. **Se va modificando el tamaño de la ventana.** Se comunica con los segmentos de ACK (y de datos).
- Cada vez que llega un ACK en orden se mueve la ventana en el Transmisor, se descartan segmentos confirmados del **Tx Buffer**.
- Se hace uso de una **ventana deslizante y dinámica.**
- **Tiene en cuenta sólo el estado del receptor, no el de la red.**



Ventana Deslizante TCP



Lo que está en "**acknowledged**" son los segmentos ya confirmados
Desde el punto de "**acknowledged**" al punto de "**enviados**" es el tamaño de la ventana de los segmentos "en vuelo"
Desde el punto de "**enviados**" a el punto de "**a enviar**" es el tamaño de la ventana de los segmentos que podría enviar

Ventana Deslizante TCP

- El mecanismo de control de flujo en TCP se basa en una **ventana deslizante**. A medida que el **receptor** procesa los datos y libera espacio en su buffer de recepción, la ventana se desliza para permitir el envío de más datos.
    - Inicialmente, el receptor anuncia una ventana de un tamaño determinado.
    - El emisor envía datos hasta llenar la ventana y espera la confirmación (ACK) del receptor.
    - Al recibir el ACK, la ventana se desliza para permitir el envío de más datos.
    - El tamaño de la ventana puede variar dinámicamente según la capacidad del receptor para procesar los datos.
- El tamaño de la ventana de recepción **puede ser ajustado por el kernel o por la aplicación** mediante la función setsockopt().
- Para redes de alta velocidad y latencia (LFNs), se utiliza el **escalado de ventana** para aumentar el tamaño máximo de la ventana y mejorar el rendimiento.
    - El escalado de ventana permite multiplicar el valor del campo "Ventana" en el segmento TCP.

Control de Congestión

- Se realiza por cada conexión End-to-End, sin la asistencia directa de la red.
- **Permite que las aplicaciones no saturen la capacidad de la red.**
- **Tiene en cuenta el estado de la red** a diferencia del control de flujo que solo ve el receptor.
- **Objetivo**: Controlar el tráfico (cantidad de datos que un emisor inyecta en la red) para evitar congestionar la red y, por ende, la pérdida de paquetes y la necesidad de retransmisiones.
- **Congestión**: Problemas de delay en los routers, problemas de overflow y descarte.

Causas de Congestión en la Red

- **Límite de la capacidad de la red:**
    - Velocidad de los Routers/Switches (CPU).
    - Capacidad de los Buffers de los Routers/Switches (Memoria).
    - Velocidad de los Enlaces (Interfaces).
- **Utilización de la red:**
    - Demasiado tráfico en la red.

Modelo End-to-End

- En este modelo no participa la red, los extremos se autorregulan, se usan nuevos parámetros a los de Control de Flujo que funcionan como variables locales:
    - **cwnd Ventana de congestión**. Tiene en cuenta el estado de la red.
    - **ssthresh Slow Start Threshold** (Umbral).

    - **Se calcula: MaxWin = Min(rwnd, cwnd).** rwnd era la ventana de recepción, usada para el control de flujo.
        - Nunca vamos a poder transmitir más que la **ventana de transmisión (rwnd)** que nos anunció el otro extremo.
- **Hay diferentes fases:**
    - Si la **Ventana de Congestión < Umbral** → Estamos en una **Fase de crecimiento inicial (SS - Slow Start).**
    - Si la **Ventana de Congestión >= Umbral** → Estamos en una **Fase de mantenimiento (CA - Congestion Avoidance).**

Conceptos

Slow Start (SS)

- Fase que se caracteriza por un **crecimiento exponencial** de la **ventana de congestión** al **inicio** de la **conexión TCP**.
- Se llama **Slow Start** porque comienza a probar con pocos paquetes, menos agresivo que el enfoque de enviar tanto como la ventana de recepción permita.
- Comienza con una **ventana de congestión** de **1 * MSS (a veces se usa 2 o 3)** y **duplica** su valor con cada **RTT**, incrementando exponencialmente la **ventana de congestión** en **2 * MSS, 4 * MSS, etc.** por cada **ACK recibido**.
    - Este enfoque permite una rápida evaluación del ancho de banda disponible en la ruta.
- La fase de **Slow Start** termina cuando **ventana de congestión** alcanza un **umbral** predefinido (ssthresh) o se detecta una **pérdida.**

Congestion Avoidance (CA)

- Esta fase se activa cuando la **ventana de congestión alcanza** o **supera** el **umbral** ssthresh.
- En esta fase, el crecimiento de **ventana de congestión** se vuelve **lineal**, incrementándose aproximadamente en **1 MSS por cada RTT** para evitar saturar la red.
- El **objetivo** es encontrar un punto de equilibrio entre el uso óptimo del ancho de banda y la prevención de la congestión.

Fast Retransmit (FRT)

- Este mecanismo busca acelerar la **recuperación de un segmento perdido sin esperar la expiración del RTO.**
- Se activa cuando el emisor recibe **3 ACKs duplicados (4 ACKs para el mismo segmento)**, indicando que el segmento se considera perdido.
    - En este caso, el emisor retransmite el segmento perdido inmediatamente sin esperar el timeout.

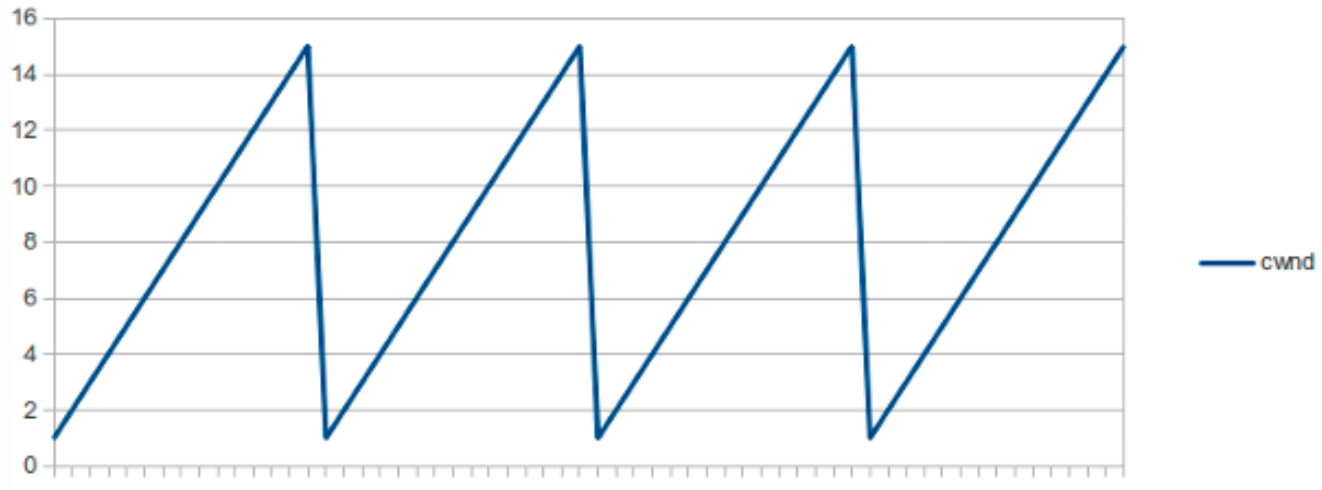
Fast Recovery (FR)

- Después de realizar una **Fast Retransmit**, el **emisor** entra en la fase de **Fast Recovery**.
- En lugar de reiniciar la **ventana de congestión**, Fast Recovery permite un aumento lineal de **ventana de congestión** durante esta fase.
    - Esto se basa en la premisa de que la red aún puede manejar cierto tráfico, ya que se recibieron **ACKs duplicados**.
    - Incrementa de forma lineal la ventana **por cada ACK recibido**, considera que es un espacio nuevo en la red **(incrementa inicialmente en 3, por los 3 ACKs duplicados)**
- Una vez que se recibe el **ACK** del **segmento re-transmitido**, se vuelve a la fase de **Congestion Avoidance.**

Evolución de Control de Congestión TCP

Old Version

- Primer enfoque de Control de Congestión.
- Se divide en capas, TCP/IP, **No considera control de Congestión inteligente**.
- **“Ventana de congestión: cwnd”** (no definida en esta versión), el tráfico crece hasta que se resetea, buffer overflow o error.
- El destino podría limitarla con **rwnd**: valido para LAN.
- **No utiliza Slow Start (SS)**, se supone: incrementa cwnd ++ (en MSS) por cada RTT.
- **ACK perdido o segmento erróneo deriva en arrancar de 0.**
- Esta versión no era adecuada para entornos WAN como Internet.

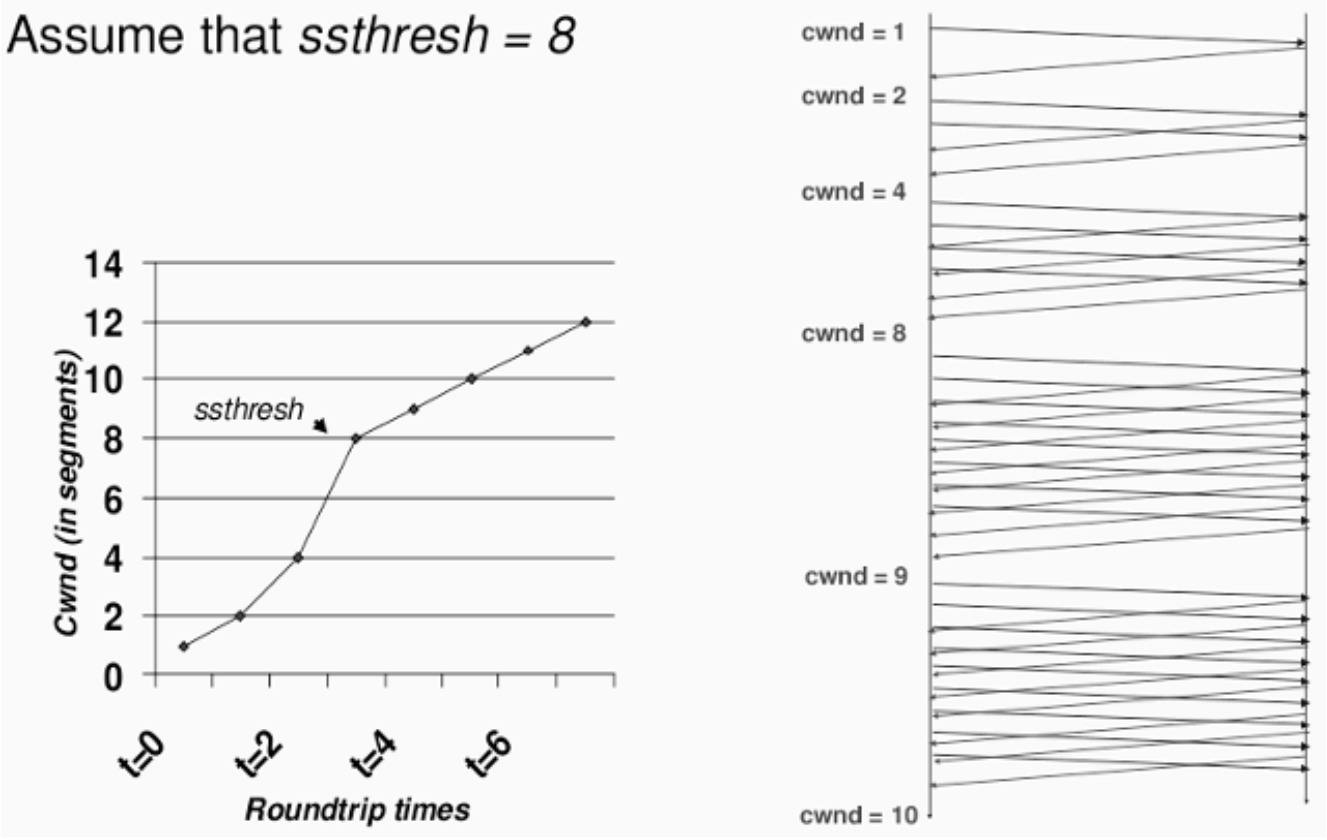


Control de Congestión TCP Old Version - Crecimiento de Ventana

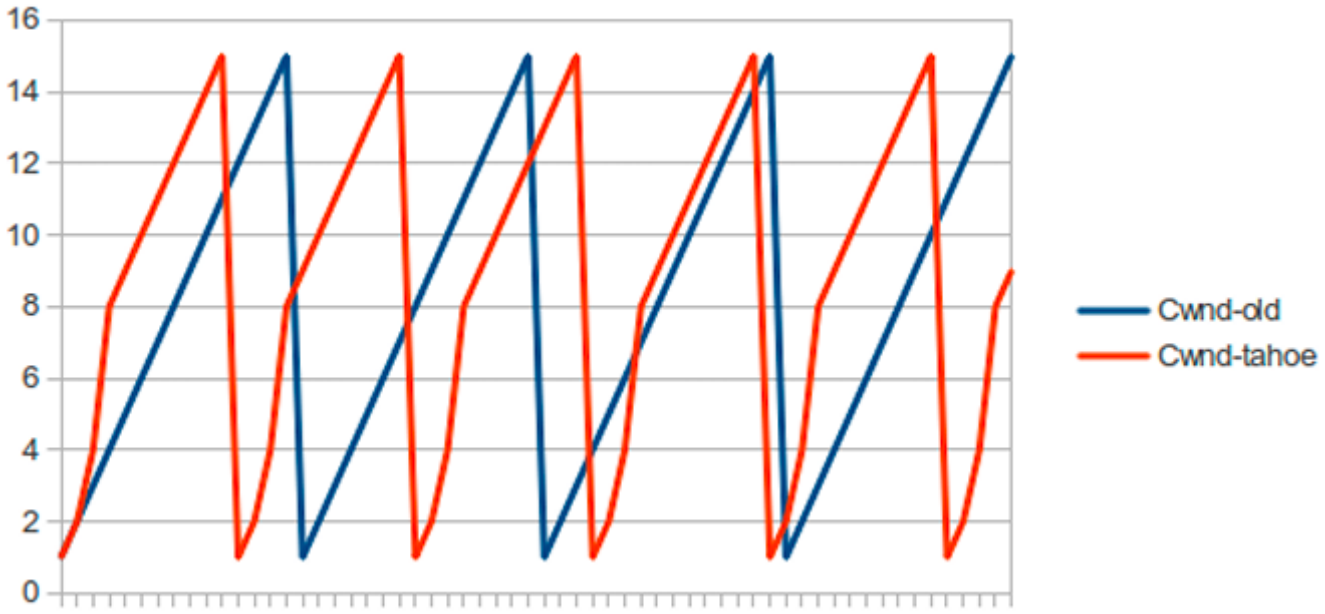
Old Tahoe

- 2do enfoque de Control de Congestión.

- **Utiliza Slow Start (SS)**, la ventana crece exponencialmente, inicio cwnd = IW = 1 ∗ MSS, o similar, IW (Init Window) puede ser 2 o 3 segmentos. **La ventana empieza con un valor pequeño y después crece exponencialmente hasta alcanzar el Umbral.**
- Una vez que se alcanza el **Umbral** (sstresh) se trabaja con **Congestión Avoidance (CA)** → Aumentamos la ventana de forma lineal**.**
    - El valor inicial del Umbral es un valor muy alto.
- **No posee Fast Retransmit.**
    - Tahoe agrega en un punto **Fast Retransmit** pero no está bien implementado.
- La congestión es detectada con **timeout (RTO O 3dup ackS),** derivaba en ambos casos en **Slow Start**.
    - Se establece **umbral** (sstresh): Min(cwnd/2, 2) ∗ MSS → **El umbral pasa a la mitad de la ventana que habíamos alcanzado.**
    - cwnd = LW = 1 ∗ MSS (Loss Window). → **Vuelve a arrancar desde abajo.**
- Puede suceder que **MSS** sea **diferente entre emisor y receptor**, para este caso se considera **SMSS** y **RMSS**. Los **cálculos** se hacen en **base a SMSS**.



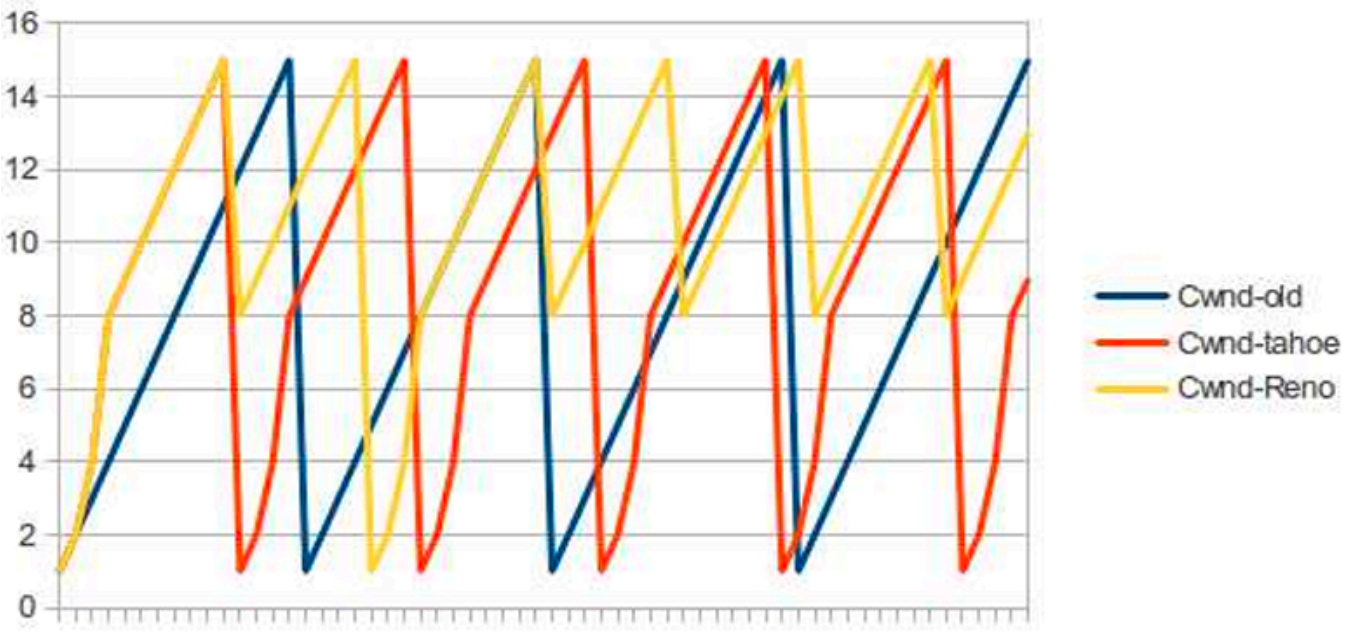
Pasaje de fase SS (izquierda) a CA (derecha)



Comparación entre Old Version y Old Tahoe

Reno

- 3er enfoque de Control de Congestión.
- **Implementa de forma correcta Fast Retransmit y Fast Recovery (además de implementar SS y CA).**
- FlightSize = cwnd si siempre tuvo datos para enviar, aunque en general cwnd > FlightSize.
- **Si hay un timeout (RTO expira):**
    - Hace los mismos cálculos que **Old Tahoe** y arranca con **SS.**
    - Comienza a re-transmitir como **Go-Back-N a partir del segmento que dio timeout** de acuerdo a lo que le permite la **ventana.**
- Utiliza un enfoque **AIMD (Additive Increase/Multiplicative Decrease)**, crece de forma aditiva con ACK positivos y decrece de forma multiplicativa ante 3 ACKs DUP (a la mitad).
    - **Se tratan de autorregular entre flujos.**



Comparación entre Old Version, Old Tahoe y Reno

New Reno

- 4to enfoque de Control de Congestión (una variante utilizada hoy en día).
- Mejoró el rendimiento en casos de **pérdida de múltiples segmentos**. Permite al **transmisor** continuar con **"Fast Recovery"** incluso después de **re-transmitir un segmento perdido**, lo que **mejora la eficiencia en la recuperación de pérdidas.**
    - Una vez que está en **Fast Recovery,** permite enviar **varios segmentos perdidos** y recuperarse **de ACK parciales.**
- **New Reno** obtiene buen rendimiento con pérdida en varios segmentos y sin SACK.

ECN (Explicit Congestion Notification)

- Mecanismo que permite a los routers **señalar la congestión a los hosts transmisores de manera explícita, sin necesidad de descartar paquetes**. Esto se logra **marcando** los **paquetes IP** con una señal de congestión en lugar de descartarlos.
    - Los hosts transmisores que soportan ECN pueden interpretar estas marcas y ajustar su tasa de transmisión en consecuencia, evitando la congestión y mejorando el rendimiento de la red.
- **Al inicio de una conexión TCP**, los hosts que soportan ECN lo indican en los paquetes SYN. **Si ambos extremos aceptan usar ECN, se configura la conexión para utilizar este mecanismo.**
    - Cuando un **router detecta congestión**, en lugar de descartar un paquete, puede marcarlo estableciendo los bits ECN en el encabezado IP. **Existen dos bits ECN: ECT (ECN-Capable Transport) y CE (Congestion Experienced).**
    - El **receptor**, al recibir un paquete con el bit **CE marcado**, sabe que se ha encontrado congestión en la ruta.

- El **receptor** coloca una marca **ECN-Echo en el segmento ACK** que envía al **transmisor**, notificándole la **congestión**.
- El **transmisor**, al recibir el **ACK con ECN-Echo, reduce su tasa de transmisión** de la misma forma que lo haría si hubiera detectado una pérdida de paquetes.