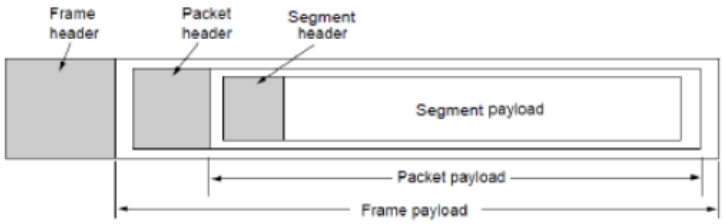
Capa de transporte

- Se ejecuta en los hosts, confía en la capa de Red
- La entidad de transporte es el software de la CT.
- permite que los procesos se ejecuten como si estuvieran directamente conectados.

Intercambia Segmentos, que están contenidos en paquetes (CR) contenidos en tramas (CD)



Se envían paquetes:
- Los datos
- info de control.

debe entregarlos ordenadamente:

- **Solución 1**: Para la entrega ordenada de segmentos al host de destino se puede:
  - Numerar los segmentos a enviar (usando **números de secuencia**) – respetando el orden del flujo de datos recibido de la capa de aplicación.
  - Usar para cada número de segmento enviado un **temporizador de retransmisiones**.
  - Mandar **confirmaciones de recepción (ACK)** para segmentos recibidos correctamente.
  - Si expira el temporizador de un segmento sin recibir el ACK, retransmitir el segmento correspondiente.
  - Los segmentos recibidos son **re-ensamblados en orden** y entregados a la capa de aplicación del receptor.

es responsabilidad de TCP, cuando se terminan dicen temporizadores y reenvían datagramas

Protocolo Seguro que garantiza envío correcto: TCP: recibe

- Retransmisión de paquetes:
  - uso de números de secuencia, confirmaciones de recepción y temporizadores.
- Fijar la duración de temporizadores de retransmisiones (algoritmo complejo)
- Manejo de conexiones entre pares de procesos
- Direccionamiento
- Control de congestión
- Control de flujo

Una ETCP acepta **flujos de datos** a **transmitir** de procesos locales,
- Cada flujo de datos se **divide en fragmentos** llamados segmentos que no exceden los 64 KB,
- y se envía cada segmento dentro de un datagrama IP.

Utiliza sockets este host > redes para la comunicación.

socket: IP + puerto.

- Las conexiones se identifican mediante los identificadores de sockets (S1, S2).

Importante: Cada byte de un flujo de datos a enviar en una conexión TCP tiene su propio **número de secuencia** de 32 bits.

- Esto impone un límite en el tamaño de un flujo de datos.

¿Por qué se necesitan los números de secuencia?

- para confirmaciones de recepción y para otros asuntos según veremos.

La ETCP emisora y la receptora intercambian datos en forma de **segmentos**.

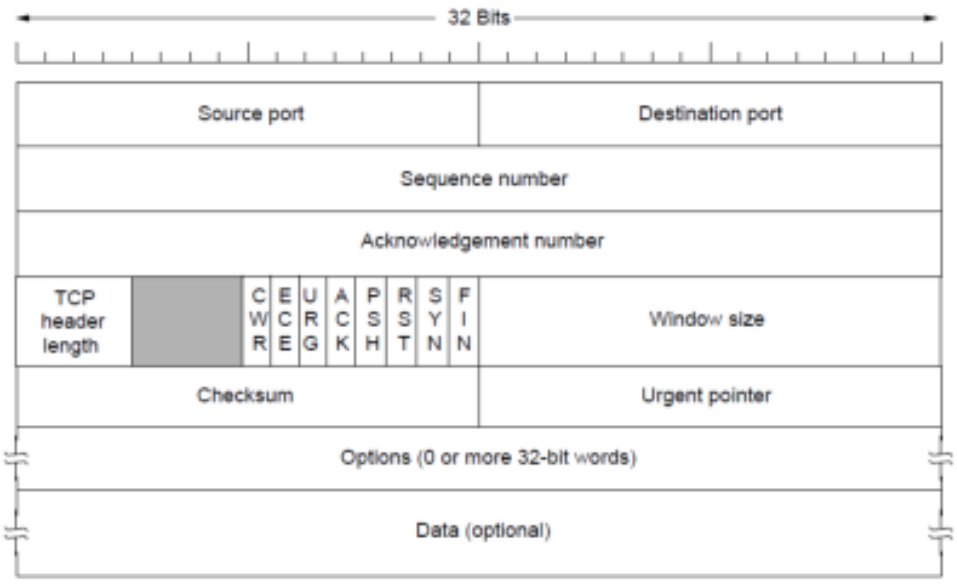
- Segmento = **encabezado TCP** ++ (0 o más bytes) de datos.
- Cada segmento, debe caber en la carga útil de 65.515 bytes del IP.
- Cada red tiene una **unidad máxima de transferencia (MTU)** y cada segmento debe caber en la MTU.
  - En la práctica la MTU es usualmente de 1500 bytes (el tamaño de la carga útil de Ethernet).

TCP reensambla los datagramas según su número de secuencia.

Cuando un transmisor envía un segmento, también inicia un temporizador.

- Cuando llega el segmento a destino, la ETCP receptora devuelve un segmento (con datos si existen, sino sin ellos) que contiene un **número de confirmación de recepción** igual al *siguiente número de secuencia que espera recibir.*
- Si el temporizador expira antes de llegar el ack, el emisor envía de nuevo el segmento.

1. **Encabezado fijo** de 20 bytes
2. **Opciones de encabezado** en palabras de 32 bits
3. **Datos** opcionales



Los segmentos sin datos se usan para acks y mensajes de control.

Puerto de origen y puerto de destino:

- Son de 16 b cada uno
- La direccion de un puerto mas la direccion IP del host forman un punto terminal unico de 48 b
- Los puntos terminales de origen y de destino en conjunto identifican la conexion

El campo **numero de secuencia** de un segmento es un numero de byte en el flujo de bytes transmitido y corresponde al primer byte en el segmento. Tiene 32 b de longitud

El campo **numero de confirmacion de recepcion** indica el siguiente byte esperado del flujo de bytes a transmitir, Tiene 32 b de longitud

Longitud del encabezado TCP: N° de palabras de 32 bits en el encabezado TCP
Longitud del campo de opciones: variable

Direccionamiento:

- Si el cliente no sabe en qué puerto está corriendo:

Solución: Existe un proceso especial llamado **servidor de directorio** que para cada tipo de servicio sabe cuáles son los puertos de los servidores que prestan ese tipo de servicio.

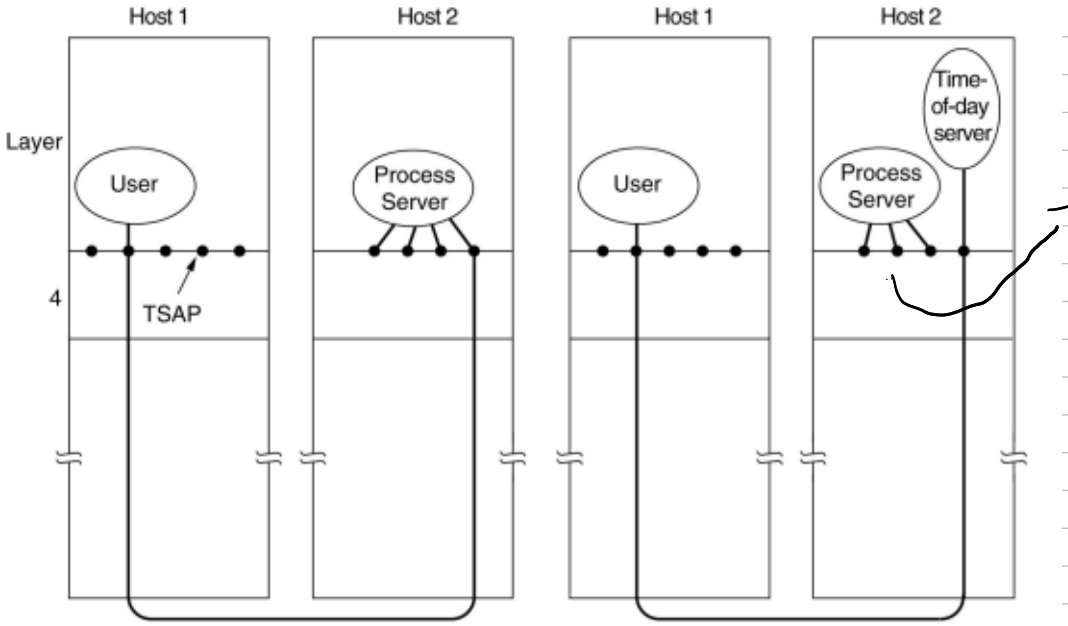
– Pasos seguidos:

1. El usuario establece una conexión con el servidor de directorio (que escucha en un puerto bien conocido).
2. El usuario envía un mensaje especificando el nombre del servicio.
3. El servidor de directorio le devuelve la dirección puerto.
4. El usuario libera la conexión con el servidor de directorio y establece una nueva con el servicio deseado.

¿Cómo se hace cuando se crea un servicio nuevo?

– El servicio nuevo debe registrarse en el servidor de directorio, dando su nombre de servicio como la dirección de su puerto.
– El servidor de directorio registra esta información en su base de datos.

Como no todos los procesos de servicio están siempre activos, entonces el servidor de procesos se encarga de escuchar a los puertos de servicios inactivos y "despertarlos" cuando hace falta.



escucha solicitudes nuevas.

¿Cuál es la diferencia entre dir. de directorio y puerto?

Puertos bien conocidos

- N° puertos bien conocidos, son los números menores a 1024
- Tabla de puertos bien conocidos (ver abajo).
- **Demonios** = procesos servidores que atienden en un puerto
  - P. ej. que el *demonio FTP* se conecte a sí mismo al puerto 21 en el tiempo de arranque.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Problema: Se podría llenar la memoria con demonios que están inactivos la mayor parte del tiempo.

Solución: Un solo demonio llamado **inetd** (**demonio de internet**), escucha un conjunto de puertos al mismo tiempo y espera por un pedido de conexión.

- Usuarios potenciales de un servicio comienzan a hacer **pedido CONNECT** especificando el puerto del servicio que quieren.
- Si no hay ningún servidor esperando por ellos, *inetd* bifurca un nuevo proceso y ejecuta el demonio apropiado en él, y ese demonio maneja la solicitud.

– Inetd aprende qué puertos va a usar de un **archivo de configuración.**
  - Los demonios asociados a los puertos de este archivo **solo** están activos si hay trabajo para hacer.
– Se puede tener **demonios permanentes** en los puertos más ocupados e inetd en los demás.
  - Esto lo fija el administrador de sistema.

Entrega de data confiable.

problema: duplicados.

sol: N° de secuencia, para que número pueda ver si es uno nuevo o duplicado.

Protocolo de Parada y Espera

- **Suposición:** el canal de comunicaciones subyacente puede perder paquetes (de datos, de ACKs)
  - Los paquetes tienen N° de secuencias.
    - Con 1 bit es suficiente.
  - Se trabaja con Acks
    - El receptor debe especificar N° de secuencia del paquete siendo confirmado.
  - Se usan retransmisiones de paquetes.
    - Para esto se requiere de uso de temporizadores.

Comportamiento del emisor:

1. El emisor envía paquete P y **para** de enviar.
2. **Espera**: El emisor espera una cantidad "razonable" de tiempo para el ACK
3. Si llega el ACK a tiempo, se envía siguiente paquete. Goto 2.
4. Sino se retransmite paquete P. Goto 2.

- Si hay paquete o ACK demorado pero no perdido:
  - La retransmisión va a ser un duplicado con igual número de secuencia ; luego se descarta en el receptor.

Protocolo de tubería: se envían multiples paquetes al mismo tiempo.

- emisor tiene un buffer donde guarda los paquetes hasta que recibe la confirmación.

Retroceso-N:

receptor envía ACK acomulativo: mayor N a rec tq todos los seq anteriores se recibieron bien.

emisor tiene timer por el paquete mas viejo no confirmado.

- El receptor descarta todos los paquetes subsequentes al paquete perdido.

Comportamiento del receptor:

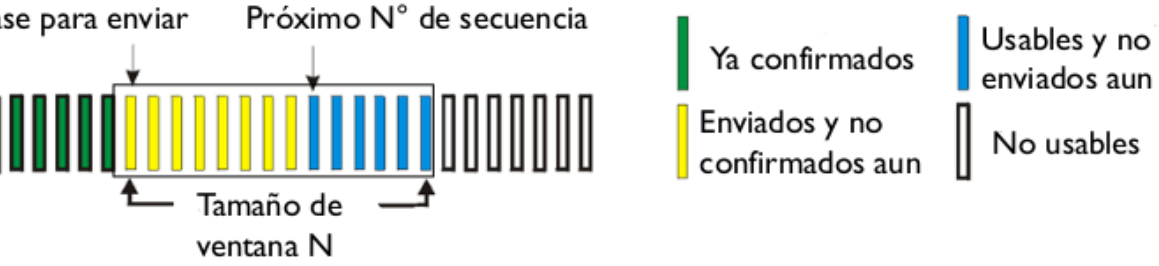
- Receptor envía **ack acumulativo**
  - mayor número de secuencia tal que todos los segmentos anteriores se recibieron bien.
- Asumir que el receptor recibió un paquete n.
- Si *n* está en orden (todos los paquetes anteriores llegaron) y está correcto (sin errores):
  - manda ack para *n* y entrega parte de datos de paquete *n* a capa superior.
- Sino:
  - el receptor descarta el paquete *n* y manda ACK del paquete más reciente recibido en orden.

Comportamiento del emisor:

- El emisor tiene un solo temporizador para el paquete más viejo no confirmado.
- Al expirar el temporizador (del segmento más viejo no confirmado),
  - retransmite todos los segmentos no confirmados.
- Si llega ACK nuevo y hay segmentos enviados no confirmados,
  - el temporizador es reiniciado.
- Si llega ACK nuevo y no hay segmentos sin confirmar,
  - el temporizador es detenido.

problema: hay un límite de lo que puede enviar en el buffer. permite hasta N paquetes sin confirmar.

- **ventana emisora** = tramas enviadas sin ack positivo o tramas listas para ser enviadas.



- *timeout(n):* retransmite paquete *n* y todos los paquetes de mayor N° de secuencia en la ventana.

la ventana emisora no puede superar MAX_SEQ cuando hay MAX_SEQ+1 número de secuencia.

Retroceso-N hace mal uso de los segmentos perdidos o demorados.

Repetición selectiva:

- los paquetes en buen orden llegan a la capa de aplicación. Si se almacenan en buffer, cuando llega E, se mandan todos en orden a capa de app.

- El receptor confirma individualmente todos los paquetes recibidos correctamente.
  - Hay búferes para paquetes según se necesiten para su entrega eventual en orden a la capa de aplicación.
- El emisor solo reenvía paquetes para los cuales el ACK no fue recibido o se recibió un NAK.
  - Hay un temporizador del emisor para cada paquete no confirmado.

- **Ventana del emisor**
  - Contiene *N* N°de secuencias consecutivos
  - Limita N°de sequencias a enviar a paquetes no confirmados.
- **¿Qué tipos de paquetes puede haber en la ventana del emisor? (ayuda considerar que estamos en repetición selectiva)**
- Como se confirman todos los paquetes que llegan y puede haber paquetes perdidos:
  - Paquetes enviados y confirmados porque antes hay paquetes no confirmados
  - Paquetes enviados y no confirmados
  - Paquetes listos para enviarse en búfer

como el receptor no entrega los que no llegan en orden, debe guardar en un buffer los paquetes hasta que lleguen todos los precedentes para enviarlo a capa de aplicación.

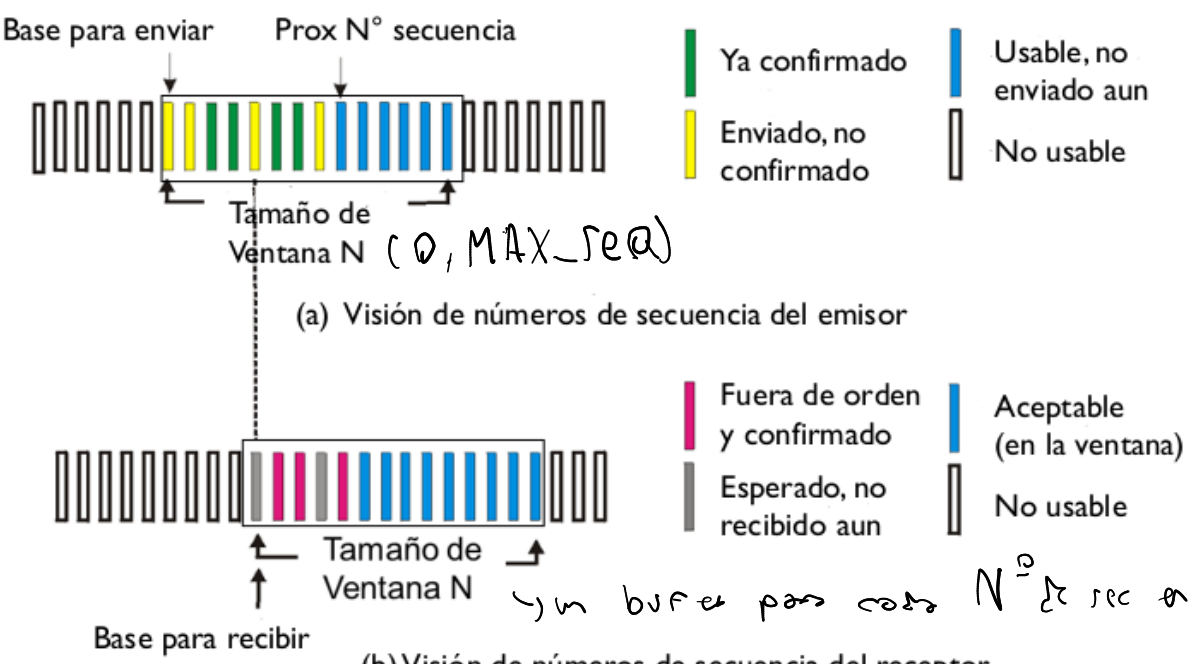
receptor tiene una ventana receptora, es un intervalo centro de todo el espacio de N° de sec.

Tipos de paquetes que puede haber en la ventana del receptor:

- Paquetes esperados y no recibidos
- Paquetes recibidos fuera de orden
- Paquetes aceptables en la ventana que no han llegado aun

Se mantiene en búfer un paquete aceptado por la ventana receptora

- hasta que todos los que le preceden hayan sido pasados a la capa de aplicación.



(0, MAX_SEQ)

un buffer para cada N° de sec en su ventana.

Cuando llega un paquete se chequea si pertenece dentro de la ventana, cuando se envían paquetes a la cap de app, la ventana avanza, por lo que si llega paquete supuestamente antiguos, serán ignorados, pq caen fuera de la ventana.

Emisor

Datos vienen de arriba:

- Si el próximo N° secuencia a enviar de la ventana está disponible, almacenar y enviar paquete

timeout(*n*):

- Reenviar paquete *n*, reiniciar timer

ACK(*n*) en [sendbase,sendbase+N]:

- marcar paquete *n* como recibido
- Si *n* es paquete más pequeño no confirmado, ***avanzar base de ventana*** al siguiente N° secuencia no confirmado.

Receptor

pkt *n* en [base rcv, base rcv +N-1]

- Enviar ACK(n)
- Fuera de orden: almacenarlo
- En orden: entregar (también entregar paquetes en bufer en orden), avanzar ventana al siguiente paquete que no ha sido recibido aun.

pkt *n* en [base rcv-N, base rcv-1]

- Enviar ACK(n)

Sino:

- ignorar

Tamaño ventana receptora:

(MAX_SEQ + 1) // 2

Súposiciones: *N* es tamaño de ventana del receptor. Algoritmo sin NAKs

para hacer más efectiva esta solución, los ACK's se mandan a "cobrarlo" los otros paquetes de datos, para así no mandar muchos paquetes SOLO con ACK's. De esta forma se puede tener un buen intercambio de datos bidireccional.

- la CT para mandar un ACK, debe esperar por un paquete al cual superponer un ACK

- **Solución: método que usa temporizador auxiliar**
  - tras llegar un paquete de datos en secuencia, se arranca un temporizador auxiliar mediante *start_ack_timer*.
  - Si no se ha presentado tráfico de regreso antes de que termine este temporizador, se envía un paquete de ack independiente.

temp_aux <<<< temp_retransmisión.

Desempeño de protocolo de entrega de datos confiable

- D_{envio} = demora en enviar paquete = $\frac{L}{R}$ → tamaño del paquete; R → vel. transmisión.
- U_{emisor} = Utilización de la línea - Fracción del tiempo en que el emisor está ocupado enviando: ⇒ $\frac{D_{envio}}{RTT + D_{envio}}$
- RTT = tiempo ida y vuelta de un bit.

Si se envían muchos paquetes seguidos (pipeline) la utilización se multiplica por la cant de paquetes enviados.

Control de flujo

- evitar que un emisor rápido desborde un receptor lento.

- **Tipo de control de flujo del que se ocupa la capa de enlace de datos:**
  - Control de flujo entre dos máquinas directamente conectadas entre sí (pueden ser enrutador o host).
- **¿Por qué puede necesitarse control de flujo en la capa de transporte si la capa de enlace de datos lo hace?**
- El receptor puede demorarse en procesar mensajes debido a los problemas de la red:
  - pérdida de segmentos,
  - no se pueden procesar segmentos porque faltan anteriores.

- Podemos asumir que el receptor maneja búferes para los mensajes que llegan.
- **Esto es necesario porque:**
  - Si la llegada de segmentos del emisor es mucho más rápido que el receptor para procesar los segmentos recibidos,
    - entonces el receptor necesitará poder almacenar segmentos antes de procesarlos.
  - El receptor puede acumular una cantidad de segmentos suficientes antes de pasarlos a la capa de aplicación para que los procese.
  - Los segmentos pueden llegar desordenados;
    - por lo tanto si llegan un grupo de segmentos y faltan segmentos previos a ellos, habrá que almacenarlos segmentos de ese grupo en buffer.

Problema: ¿Qué hace el receptor con los búferes si tiene varias conexiones?

Solución 1: se usan los búferes a medida que llegan segmentos.

Solución 2: se dedican conjuntos de búferes específicos a conexiones específicas.

Solución 1:

- Cuando entra un segmento el receptor intenta adquirir un búfer nuevo;
- si hay uno disponible, se acepta el segmento; de otro modo se lo descarta.

Suposición: cambia el patrón de tráfico de la red; se abren y cierran varias conexiones en el receptor.

Consecuencias:

- El receptor y el emisor deben ajustar dinámicamente sus alojamientos de búferes.
  - Esto significa ventanas de tamaños variables.
- Ahora el emisor no sabe cuántos datos puede mandar en un momento dado, pero sí sabe cuántos datos le gustaría mandar.

Solución 2:

Solución: El host emisor **solicita espacio en búfer en el otro extremo.**

- Para estar seguro de no enviar de más y sobrecargar al receptor.
- Porque sabe cuánto necesita.

➢ **Cuando el receptor recibe este pedido:**

- Sabe cuál es su situación y cuánto espacio puede otorgar.
- Aquí el receptor reserva una cierta cantidad de búferes al emisor.

➢ Los búferes podrían repartirse por conexión, o no.

➢ **Si los búferes se reparten por conexión y aumenta la cantidad de conexiones abiertas:**

- El receptor necesita ajustar dinámicamente sus reservas de búferes.

1. **Inicialmente el emisor solicita una cierta cantidad de búferes, con base en sus necesidades percibidas.**
2. El receptor otorga entonces tantos búferes como puede.
3. El receptor, sabiendo su capacidad de manejo de búferes podría indicar al emisor ***"te he reservado X búferes"*.**

- **¿Cómo hace el receptor con las confirmaciones de recepción?**
- ***El receptor puede incorporar tanto las ack como las reservas de búfer al en el mismo segmento.***

– El emisor lleva la cuenta de su **asignación de búferes** con el receptor.

– **Cada vez que el emisor envía un segmento:**

- Debe disminuir su asignación de búferes disponibles.

– **Cuando la asignación de búferes (disponibles) llega a 0:**

- El emisor debe detenerse por completo

para evitar situaciones donde la información de reserva se pierde, y se ocasiona un deadlock, los paquetes con ese info se envían constantemente.

Control de flujo en TCP

No se requiere:

- que los emisores envíen datos tan pronto como llegan de la aplicación.
- que los receptores envíen confirmaciones de recepción tan pronto como sea posible.
- que los receptores entreguen datos a la aplicación apenas los reciben.
  - Esta libertad puede explotarse para mejorar el desempeño.

No se puede usar el protocolo de control de flujo anterior para TCP.

- Porque en TCP los números de secuencia no significan número de paquete.
- Antes cada búfer ocupado tenía un número de paquete.
- Ahora los números de secuencia son posiciones en el flujo de datos a enviar.
- El receptor a lo más puede saber qué rangos de números de secuencia de bytes recibidos tiene en búfer.

Algunas mejoras que se pueden hacer en relación al protocolo anterior:

- Los encabezados de los segmentos recibidos ocupan espacio y no hace falta almacenarlos en búfer.
  - En su lugar se pueden almacenar datos recibidos del flujo de datos.
- No es necesario que el emisor solicite espacio de búfer al receptor.
  - El receptor sabe de cuanto espacio dispone y cuanto espacio puede otorgar.

para esto se usa un buffer de recepción circular en el receptor (guarda solo datos)

Como TCP usa un búfer circular único, el receptor no le puede decir al emisor: 'te he reservado x búferes'.

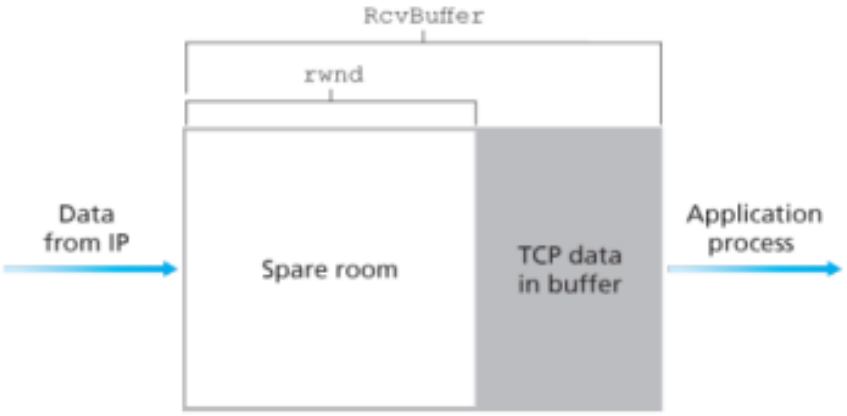
Para anunciar al emisor la reserva de espacio en búfer:

- El receptor puede indicar al emisor la cantidad de bytes consecutivos que se pueden enviar; comenzando por el byte cuya recepción se ha confirmado.
- A esto se le llama en TCP **tamaño de ventana**.
- En el encabezado TCP un **campo de tamaño de ventana** (de 16 bits) se usa para indicar esta información.

- el emisor también tiene un buffer circular para los datos a enviar.
- los byte que pueden enviar dependen de
  - el tamaño del buffer del receptor y la ventana
  - cant. de byte no es mayor que ninguno de los valores anteriores.

La fórmula para calcular el tamaño de ventana el receptor es:

Tamaño de ventana = RcvBuffer−[LastByteRcvd−LastByteRead]



lastByteRcvd es ultimo byte recibido por la capa de aplicación

Figure 3.38 The receive window (rwnd) and the receive buffer (RcvBuffer)

El receptor:

- Cuando la conexión TCP recibe bytes en el orden correcto y en secuencia, coloca los datos en el buffer de recepción.
- El receptor puede confirmar llegada de datos nuevos y anunciar nuevo tamaño de ventana al emisor.
- Si búfer de recepción está lleno, avisar tamaño de ventana de cero.
- Una vez que el receptor entrega a la capa de aplicación X datos de búfer de recepción lleno, puede avisar al emisor de un tamaño de ventana de X.

El emisor:

- Si el tamaño de ventana anunciado es cero el emisor no podrá enviar datos.
- El emisor envía segmentos cumpliendo la siguiente propiedad:

LastByteSent−LastByteAcked ≤ tamaño de ventana.

Pérdida de Segmentos en TCP

Solución 1: el receptor solicita segmento/s específico/s mediante segmento especial llamado NAK.

- Tras recibir segmento/s faltante/s, el receptor puede enviar una confirmación de recepción de todos los datos que tiene en búfer.
- Cuando el receptor nota una brecha entre el número de secuencia esperado y el número de secuencia del paquete recibido, el receptor envía un NAK en un campo de opciones.

Solución 2: (**acks selectivos**) el receptor le dice al emisor que piezas recibió.

- El emisor puede así reenviar los datos no confirmados que ya envió.
- Se usan dos campos de opciones:
  - **Sack permited option**: se envía en segmento SYN para indicar que se usarán acks selectivos.
  - **Sack option**: Con lista de rangos de números de secuencia recibidos.

Cuando la ventana es de 0, el emisor no puede enviar segmentos, salvo en dos situaciones:

1. pueden enviarse **datos urgentes** (p.ej. Para que el usuario elimine el proceso en ejecución en la máquina remota),
2. el emisor puede enviar un segmento de 1 B para hacer que el receptor ***re-anuncie*** el siguiente byte esperado y el tamaño de la ventana.
   - TCP proporciona esta opción para evitar un bloqueo irreversible si llega a perderse un anuncio de ventana.

Solución (opción de escala de ventana): permitir al emisor y al receptor negociar un factor de escala de ventana.

- Ambos lados pueden desplazar el tamaño del campo de ventana hasta 14 bits a la izquierda,
- permitiendo por lo tanto ventanas de hasta 2^30 bytes.
- La mayoría de las implementaciones actuales de TCP manejan esta opción.

Control de Congestion

S: un emisor manda mas de lo que la red puede soportar de congestion. Este control de congestion se aplica al emisor.

Para controlar la congestión:

- En TCP algunos hosts **disminuirán la tasa de datos**.

Para llevar la cuenta de cuántos datos un host puede enviar por la red:

- TCP maneja una **ventana para la congestión (VC) -** cuyo tamaño es el número de bytes que el emisor puede tener en la red en un momento dado.

En TCP el host tiene una forma de **detectar congestión**.

En TCP cuando un host detecta congestión:

- El host **ajusta** el tamaño de la VC.

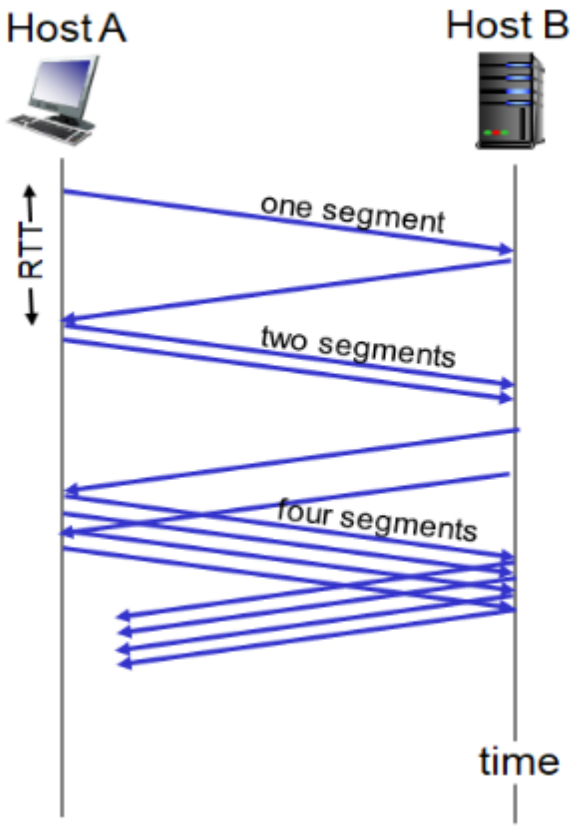
expiracion de temporizadores.

TCP asume que las expiraciones no por congestion.

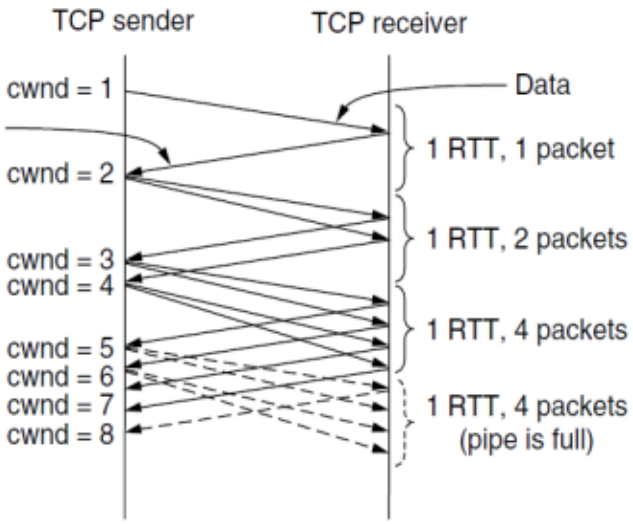
¿Como calcula la ventana de congestion?

Algoritmo de arranque lento (Jacobson 1988).

- El emisor asigna a la VC el segmento de tamaño máximo (STM) usado por la conexión; entonces envía 1 STM.
  - Emisor y receptor se ponen de acuerdo en el tamaño del STM.
- Si se recibe el ack de este segmento antes que expire el temporizador, el emisor agrega el equivalente en bytes de un segmento a la VC para hacerla de 2 STM y envía dos segmentos.



- Cuando la VC es de *n* segmentos, si de todos los *n* se reciben acks a tiempo, se aumenta la VC en la cuenta de bytes correspondiente a *n* segmentos.
- La VC sigue creciendo exponencialmente hasta expiración temporizador (timeout) o **alcanzar el tamaño de la ventana receptora**.
- Si ocurre timeout se recorta la VC a tamaño VC/2, o sea no se enviarán ráfagas de segmentos mayores a VC/2.
- Esta es la solución de la edición ante penúltima del libro de Tanembaum.



Se desperdicia capacidad de la red al hacer esto.

Esperar a ese timeout puede resultar en esperar largo.

¿Como hacemos las pedidas mas rapidamente?