

Transporte fiable

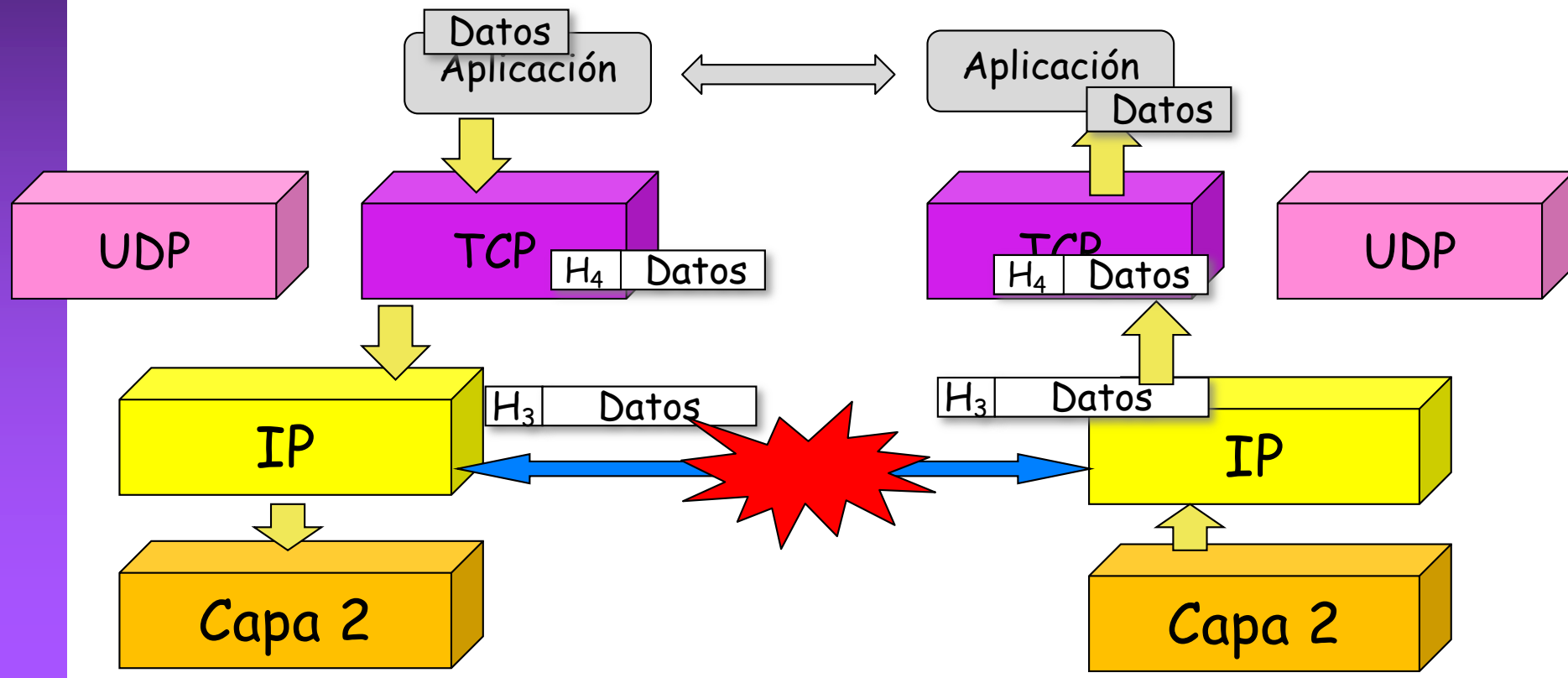
Area de Ingeniería Telemática
<http://www.tlm.unavarra.es>

Arquitectura de Redes, Sistemas y Servicios

Escenario

Transporte fiable en capa 4

- Problema: Ofrecer un servicio de transporte fiable en base a un servicio no fiable
- Nos centramos en ofrecerlo en capa 4 en base a un servicio capa 3 *best-effort* (Ejemplo: TCP sobre IP)



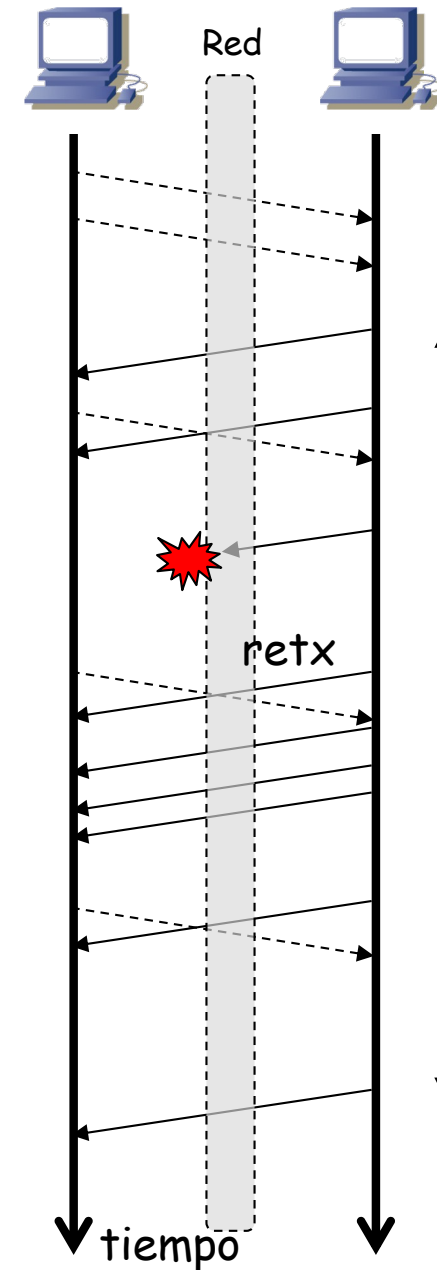
Duplex y orden

- Normalmente la comunicación entre las aplicaciones será dúplex
- Una “conversación” en capa 4 será full-duplex
- Resolveremos el problema de transporte fiable en un sentido
- En el sentido contrario solo hay que replicar el procedimiento
- Supondremos al principio que los paquetes no se desordenan en la red, solo se pierden
- Veremos después cómo podemos añadir entrega en orden



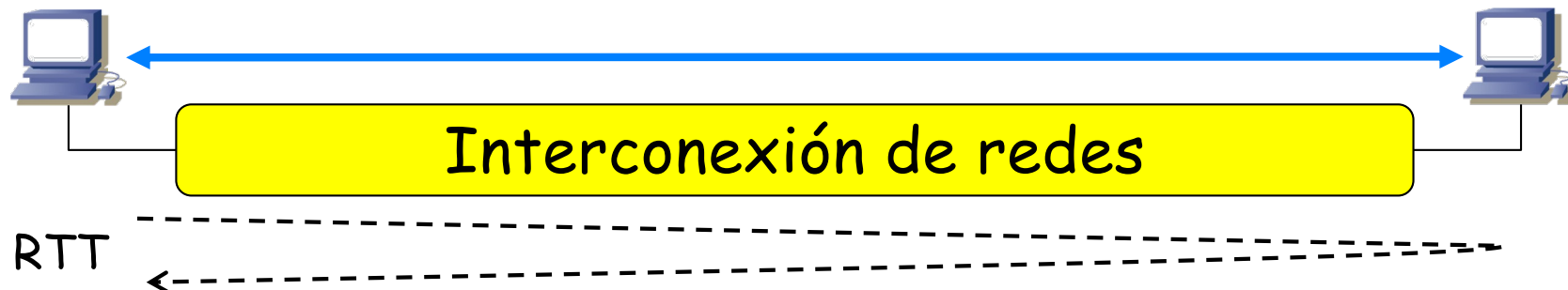
Ejemplo

- Transferencia de datos de una aplicación a otra (requiere múltiples paquetes)
- Por ejemplo, copia de un fichero
- Algunos paquetes se "pierden" en la red
- O se corrompen (llegan pero modificados)
- Habrá que retransmitir los datos perdidos
- ¿Cuánto tiempo tarda en completarse la transferencia?
- ¿Cuál es la velocidad de transferencia que ve la aplicación?
- ¿En qué medida podemos acercarnos a la capacidad del cuello de botella de la red?
- ¿Cómo depende de lo frecuentes que sean las pérdidas?



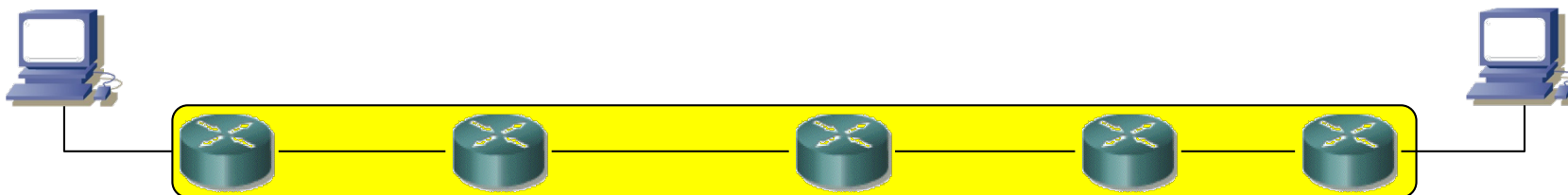
Condiciones de contorno

- Comunicación entre las aplicaciones
- Los paquetes que envía la aplicación llegan (o no) a la aplicación destino
- La red entre ellas da un servicio *best-effort*: no fiable, puede perder, desordenar y duplicar paquetes
- Los paquetes en la red tienen una MTU
- RTT en la red: tiempo de ida y vuelta entre los hosts, variable
- Cómo ofrece la red ese servicio no importa para la capa 4
- Aunque nosotros sabemos que esa red...



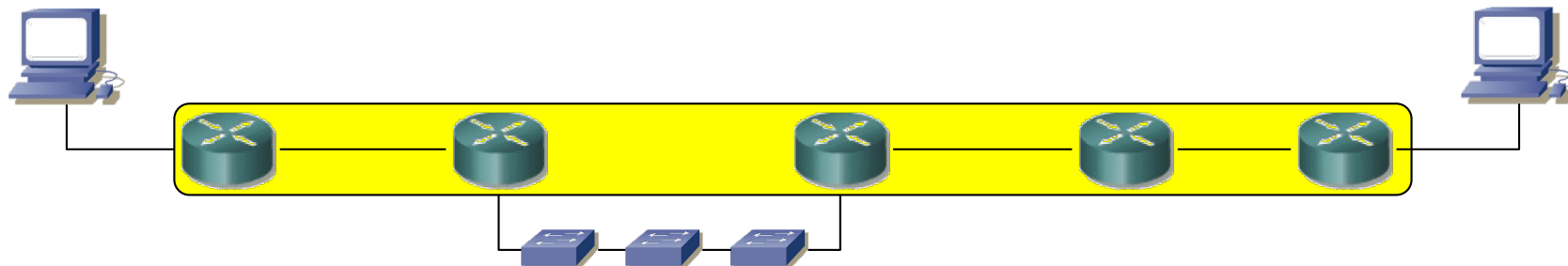
Funcionamiento de esa red

- Conmutadores de paquetes capa 3



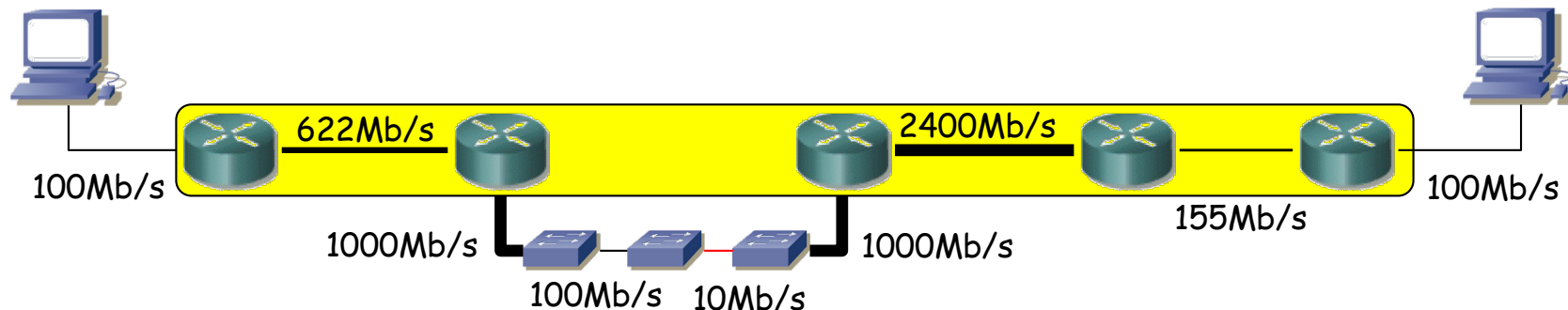
Funcionamiento de esa red

- Conmutadores de paquetes capa 3
- La comunicación entre dos conmutadores capa 3 adyacentes es mediante una tecnología capa 2
- Puede ser un cable directo o toda una tecnología de red



Funcionamiento de esa red

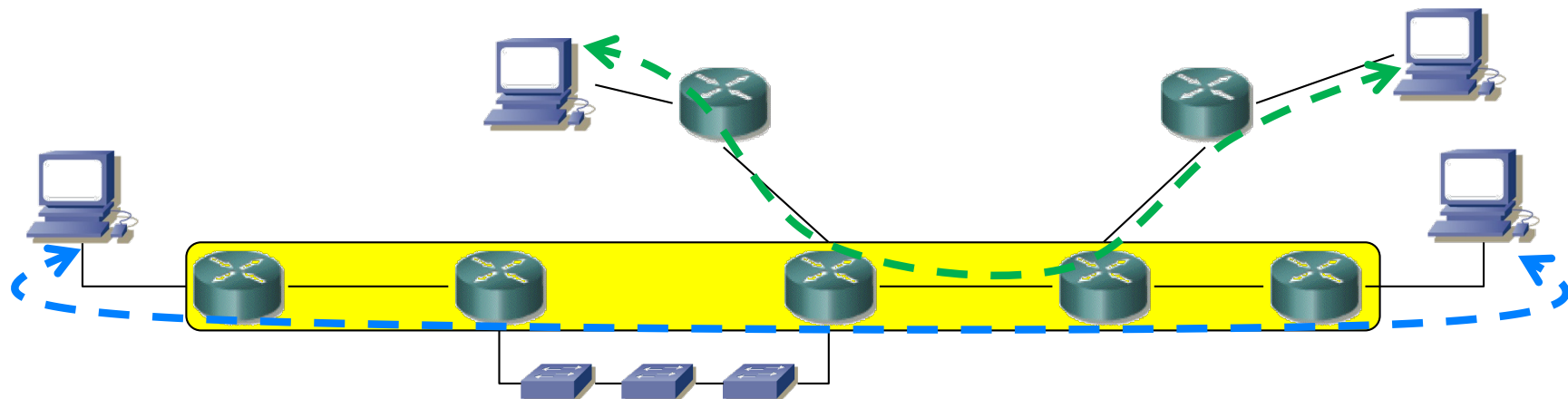
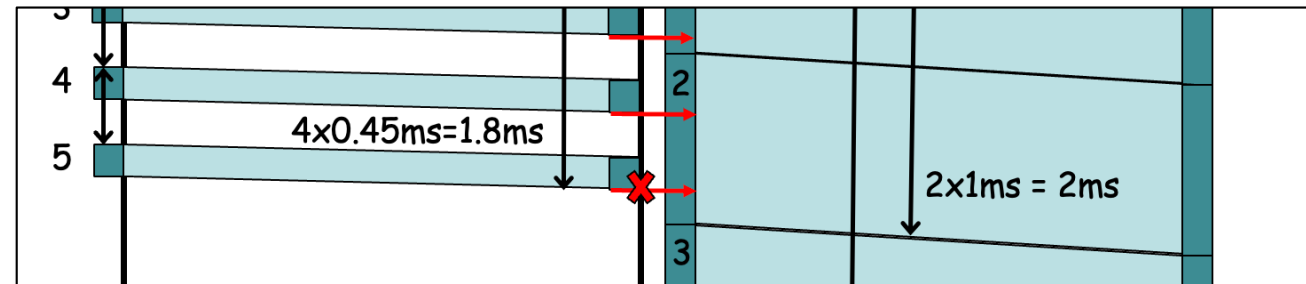
- Los enlaces tienen una capacidad
- Cuello de botella en el camino será “lo mejor” que puedan ver las aplicaciones
- Hay que tener en cuenta que hay más tráfico de otros usuarios



Funcionamiento de esa red

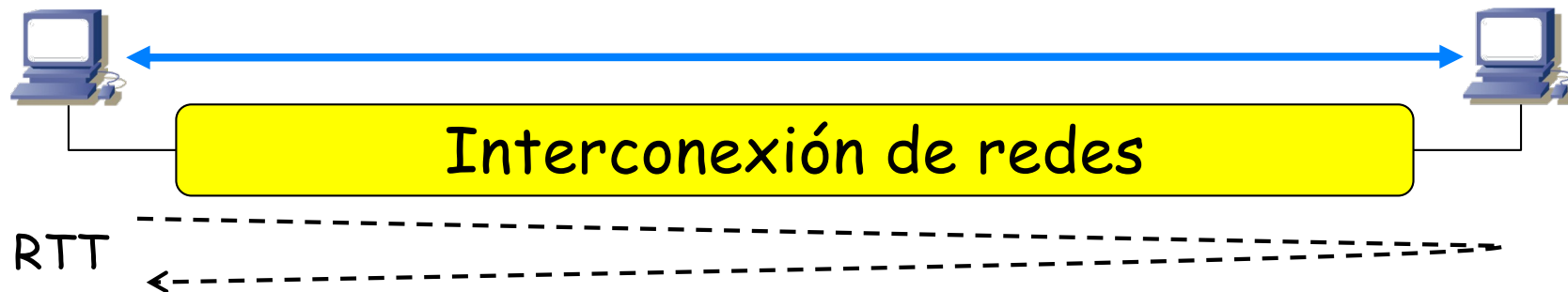
- Los paquetes normalmente se perderán por llegar a un conmutador que tenga el buffer de paquetes lleno

Latency (using 64 byte packets)	
• 10Mbps:	35µs (max)
• 100Mbps:	30µs (max)
• 1000Mbps:	15µs (max)
RAM buffer	
• 256KB	
MAC address database	
• 8K	



Condiciones de contorno

- El nivel de transporte no ve esa complejidad
- Envía paquetes y llegan (tras un tiempo) o no

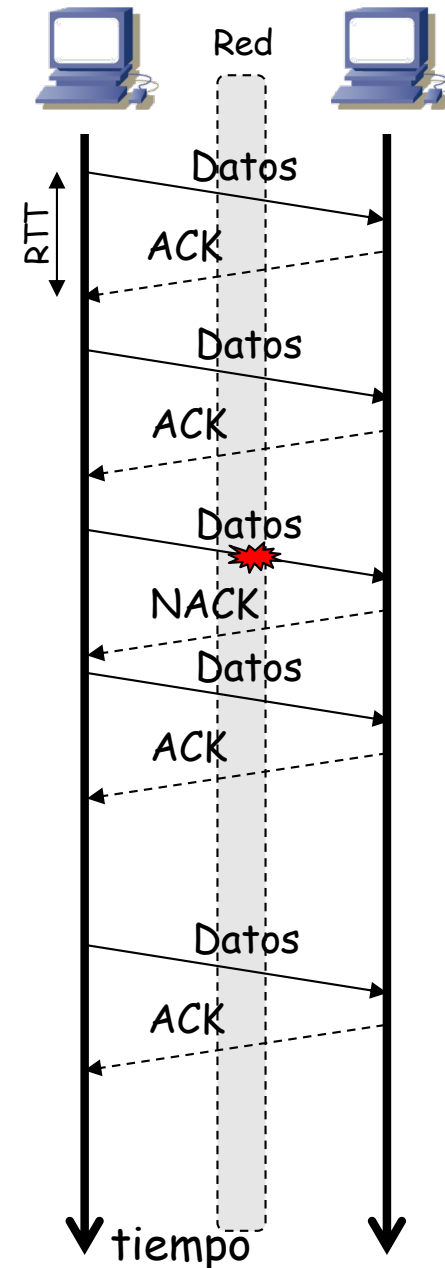


Stop & Wait

Stop & Wait sin pérdidas

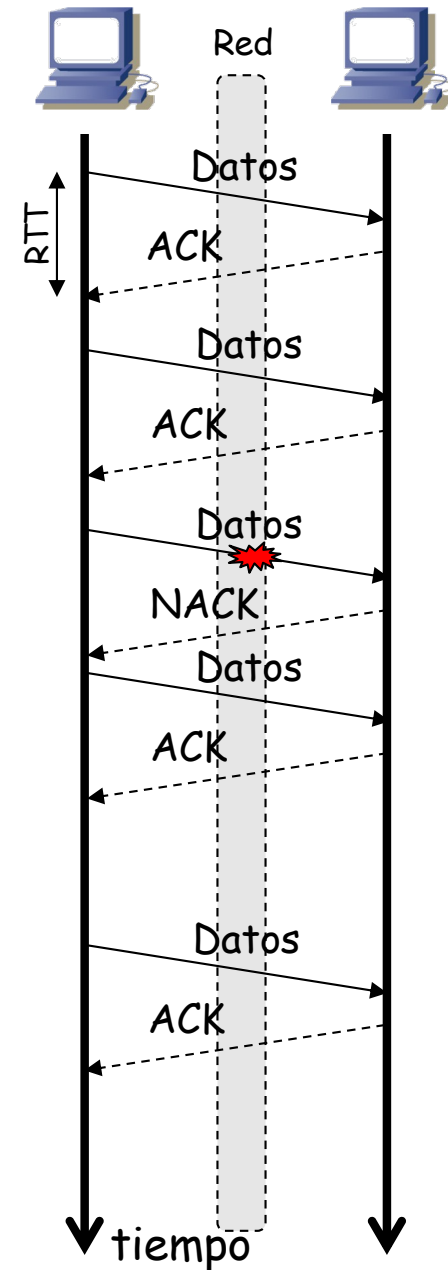
Idea general con errores

- Describimos solo comunicación en un sentido
- Emisor envía paquete y espera confirmación
- Comunicación capa 4 extremo a extremo de hosts
- Suponemos que los mensajes se pueden corromper, pero no perder
- Al llegar un mensaje al receptor puede decidir si es correcto o se ha corrompido
- Para detectar errores en el mensaje: checksum
- Si es correcto: envía al otro extremo un mensaje de confirmación (*acknowledgement*)
- Si no es correcto: envía un mensaje de confirmación negativa (*negative-ack*, *NAK* o *NACK*)



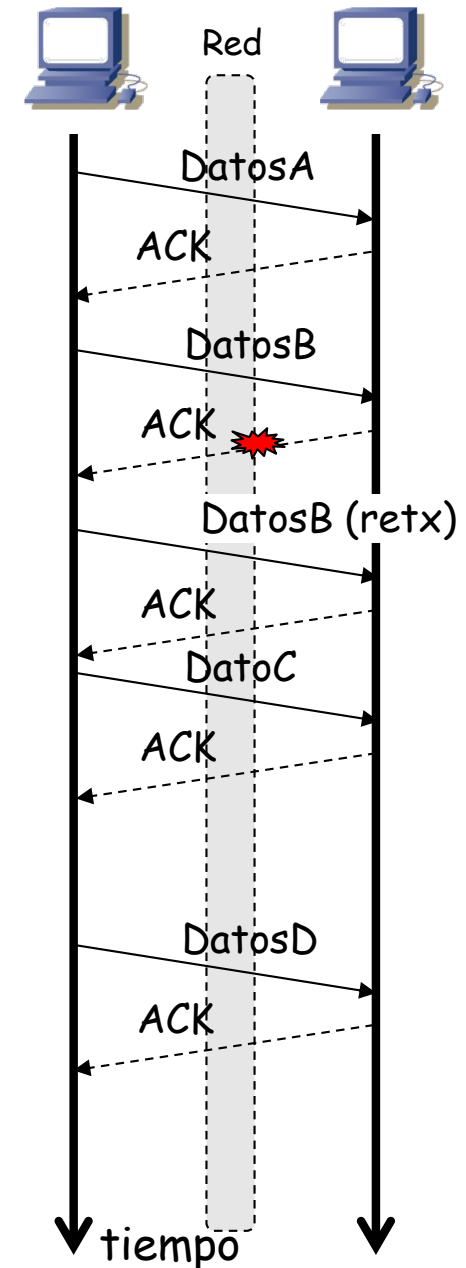
Stop & Wait

- Protocolo de tipo **ARQ** = *Automatic Repeat reQuest*
- Tras enviar datos espera ACK
- ACK/NAK hace referencia al paquete de datos enviado (solo hay uno enviado sin confirmar)
- Si recibe NAK retransmite el paquete de datos y vuelve a esperar
- Si recibe ACK pasa a enviar el siguiente paquete de datos
- Solo puede haber un paquete “en vuelo” (enviado sin confirmar)



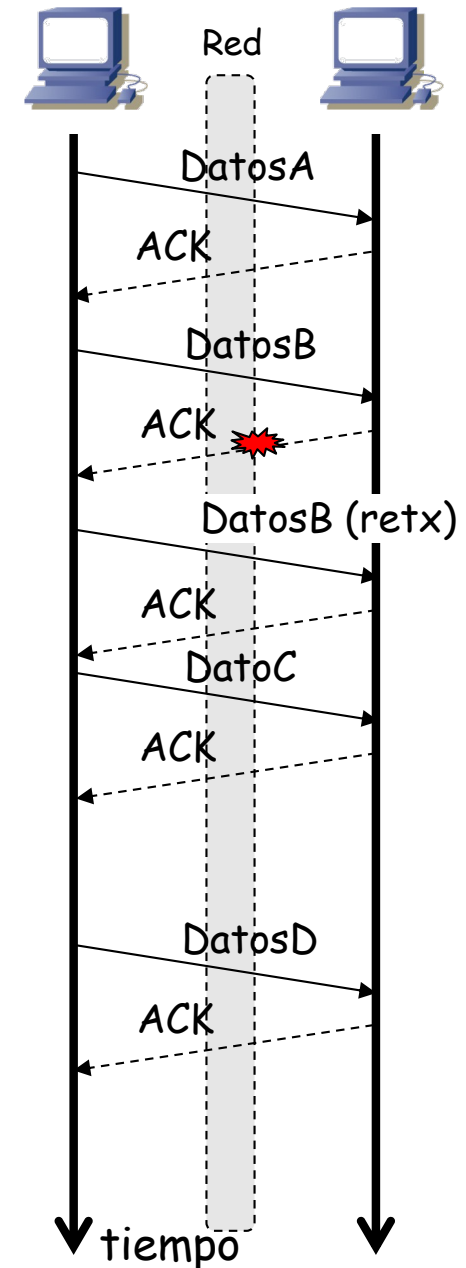
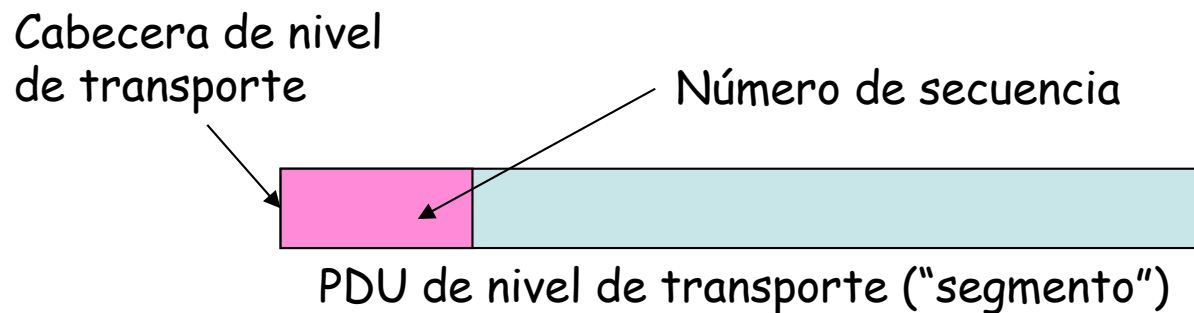
Stop & Wait : Problemas

- ¿Y si se corrompe el ACK/NAK?
- Receptor no sabe si es un ACK o un NAK
- Podríamos introducir un nuevo mensaje tipo un N-ACKoNAK para decir que lo hemos recibido corrompido
- Indicaría que hemos recibido mal un mensaje que podía ser un ACK o un NAK
- Pero se puede de nuevo corromper...
- Otra opción es suponer que es un NAK y retransmitir el paquete de datos
- Pero si era un ACK hemos introducido un duplicado
- El receptor no sabe que su ACK se corrompió y esto le puede parecer el siguiente paquete de datos
- Así pues hace falta reconocer el duplicado



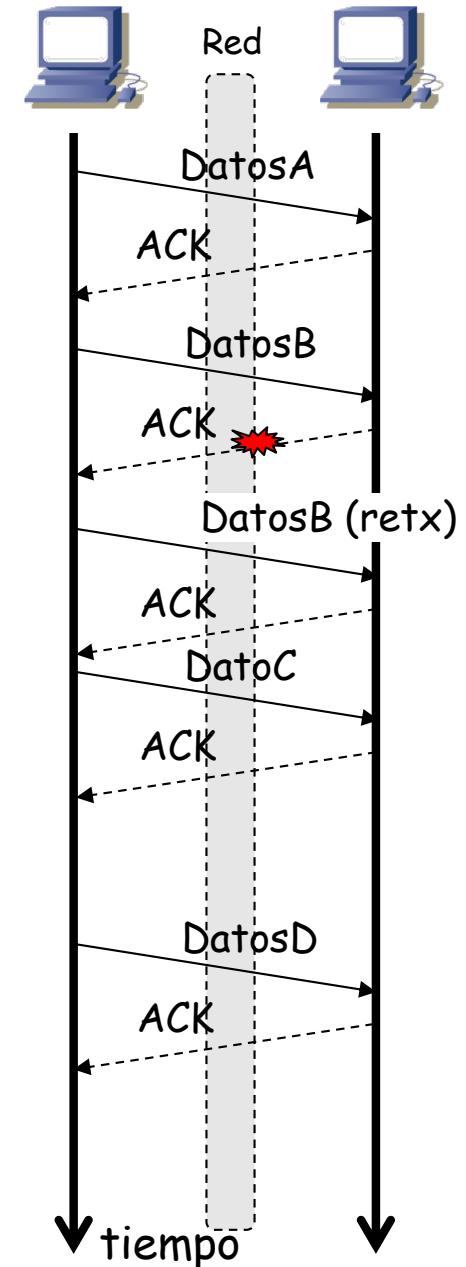
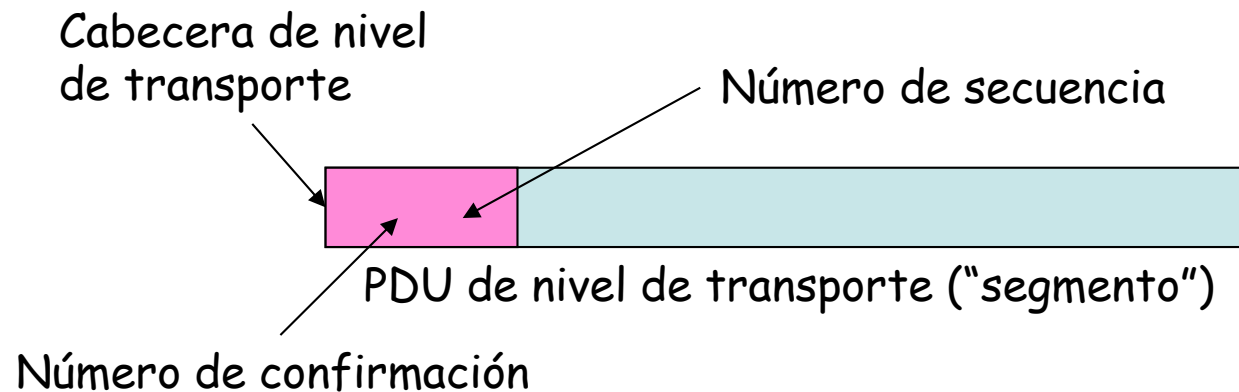
Números de secuencia

- Puede ser simplemente un bit para decir si es o no una retransmisión del paquete anterior
- O podemos numerar los paquetes de datos con un campo en la cabecera de transporte
- Algunos protocolos numeran todos los bytes de datos (el paquete lleva el nº del primer byte)
- El campo tiene un tamaño, así que se puede desbordar: emplea aritmética módulo



Números de confirmación

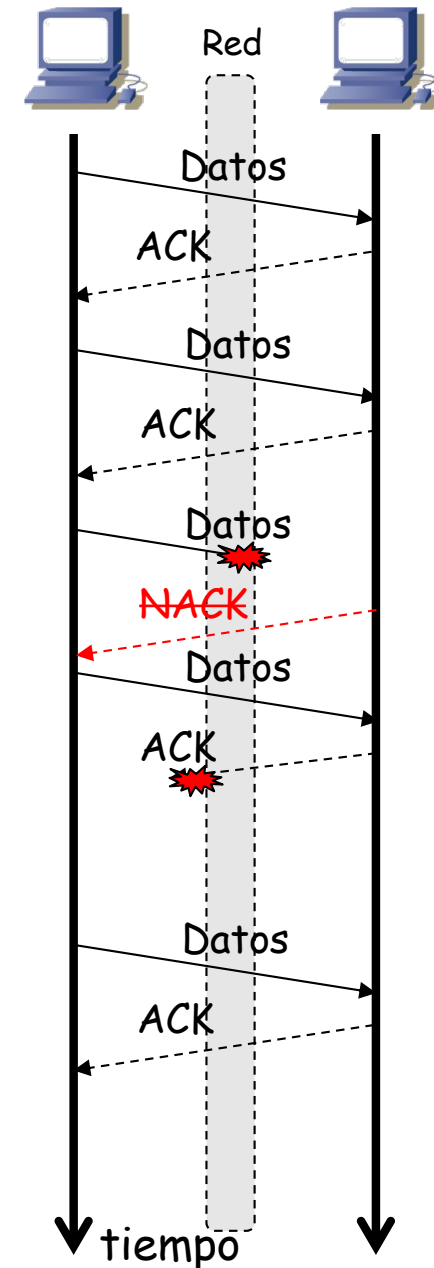
- El ACK/NAK es un mensaje de transporte sin datos, con el número de mensaje que confirma
- Muchos protocolos meten el ACK en mensaje de datos de sentido contrario (*piggybacking*)
- Normalmente el ACK dice el número de mensaje que confirma o el siguiente que espera
- En vez de enviar un NAK se puede enviar un ACK repitiendo el número del mensaje anterior recibido correctamente
- Al recibir ese **ACK duplicado** el emisor reconoce que el último paquete llegó con error



Stop & Wait con pérdidas

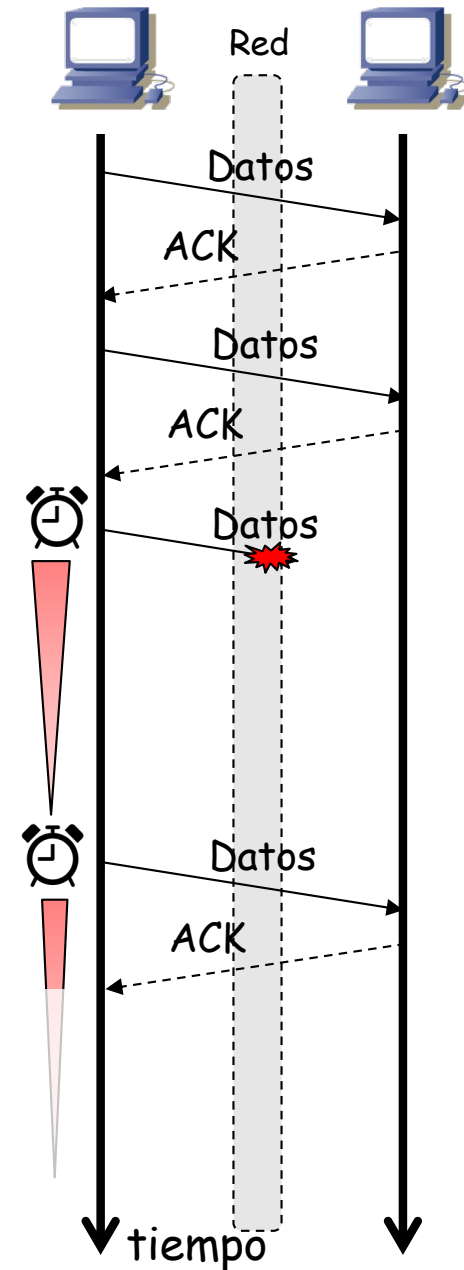
Idea general con pérdidas

- Un mensaje puede no llegar al otro extremo
- No hay NACK pues no llega mensaje
- Afecta también a ACKs/NAKs
- Se suele delegar la responsabilidad en el emisor, que es el único que sabe que hay datos a enviar
- Se pierda el paquete de datos o el ACK, el emisor ve lo mismo: que no le llega ACK
- El resultado es retransmitir
- Una retransmisión cuando se perdió el ACK genera un duplicado en el receptor
- ¿Pero cuándo decide que debe retransmitir?



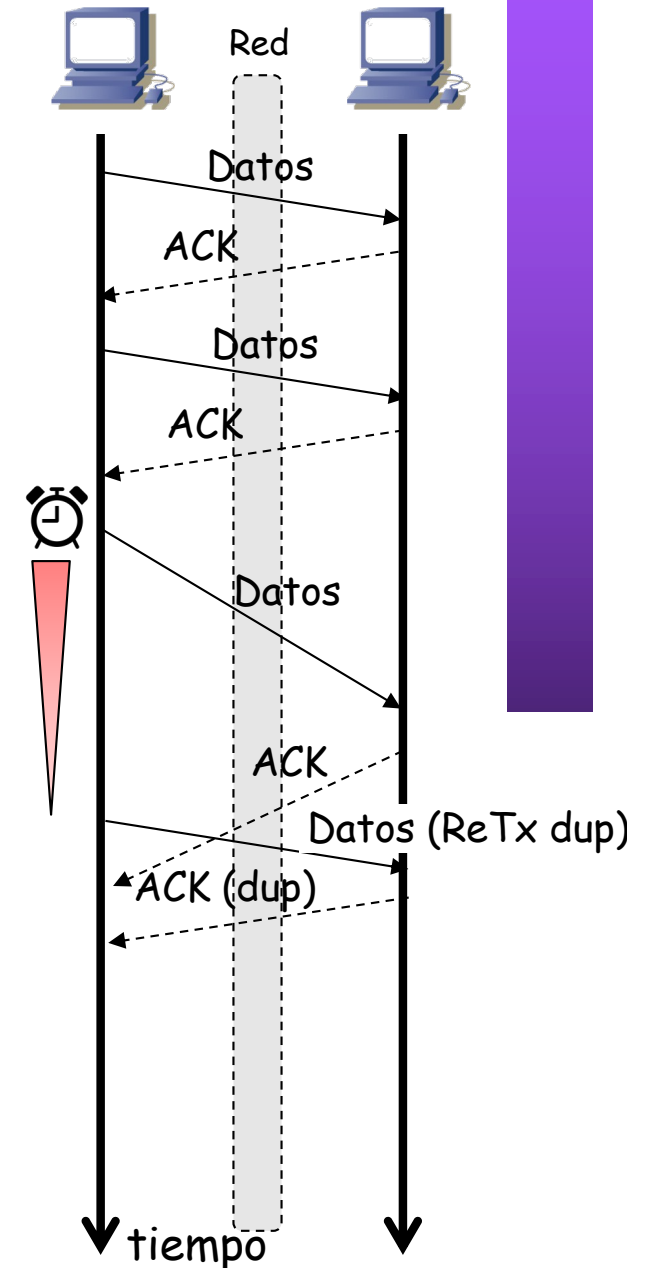
Timer de retransmisión

- Cada vez que envíe un paquete de datos deberá comenzar un contador de tiempo
- El tiempo máximo debe ser mayor que lo que suele tardar en llegar el ACK
- Si caduca sin recibirlo se retransmite
- Si llega durante ese tiempo se cancela el timer
- El tiempo que “normalmente” tarda en llegar el ACK es variable
- Depende del host receptor, pero sobre todo del retardo en la red



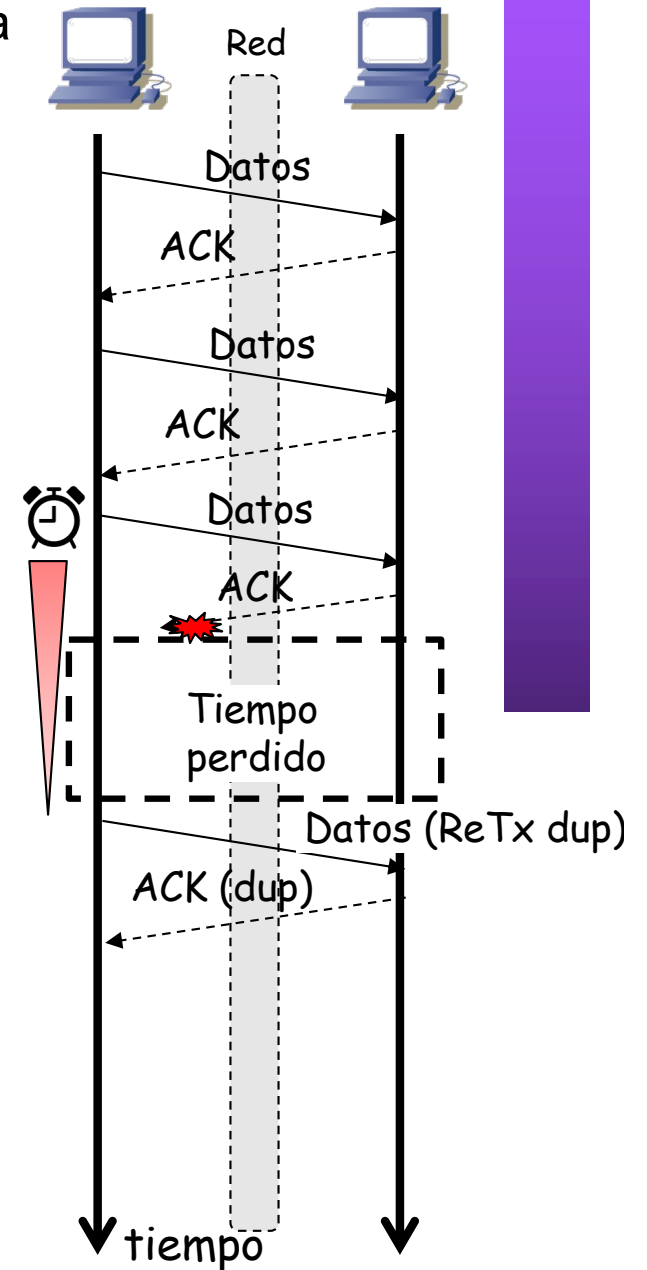
Retransmisión espúrea

- Si el timer caduca antes de que llegue el ACK porque hemos subestimado el RTT: duplicados
- Debemos tener mecanismo para detectar duplicado
- Introducimos tráfico extra innecesario
- Valor ideal del timeout: $RTT + \epsilon$
- Es decir, aproximadamente RTT
- El problema es conocer el valor de RTT
- Que además cambia en función del tráfico (debido al retardo en cola)
- La mayoría de protocolos estiman el valor del RTT con los intercambios Datos+ACK
- Y aplican un multiplicador > 1 al valor que estiman del RTT para usar como timeout



Retransmisión tardía

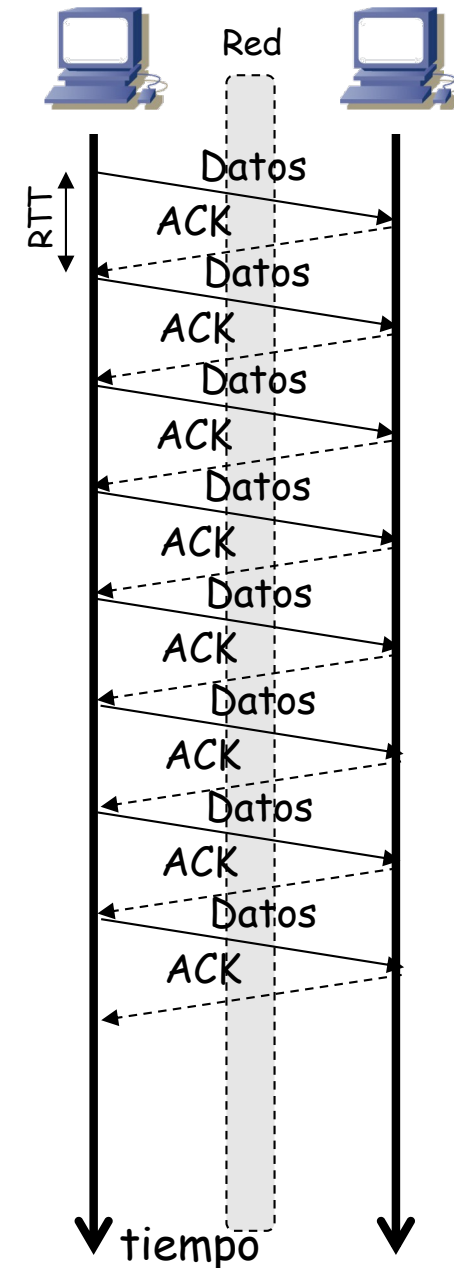
- Si el timer es mucho mayor que el RTT se tarda mucho en recuperar la pérdida
- Mientras tanto la comunicación está detenida



Stop & Wait: rendimiento

Escenario ideal

- Sin pérdidas
- Confirmación inmediata
- Envío del siguiente bloque de datos en cuanto se recibe la confirmación
- Un paquete de datos cada RTT
- Paquete de datos con L bits útiles
- Throughput medio = L / RTT
- Ejemplo:
 - Paquete con 1460 bytes
 - RTT de 40ms
 - Throughput = $1460 \times 8 / 0.04 = 292 \text{ Kb/s}$
 - Da igual que el cuello de botella tenga una capacidad muy superior
- ¿Cuánto tiempo tarda en transferir M pkts?
 - $M \times \text{RTT}$

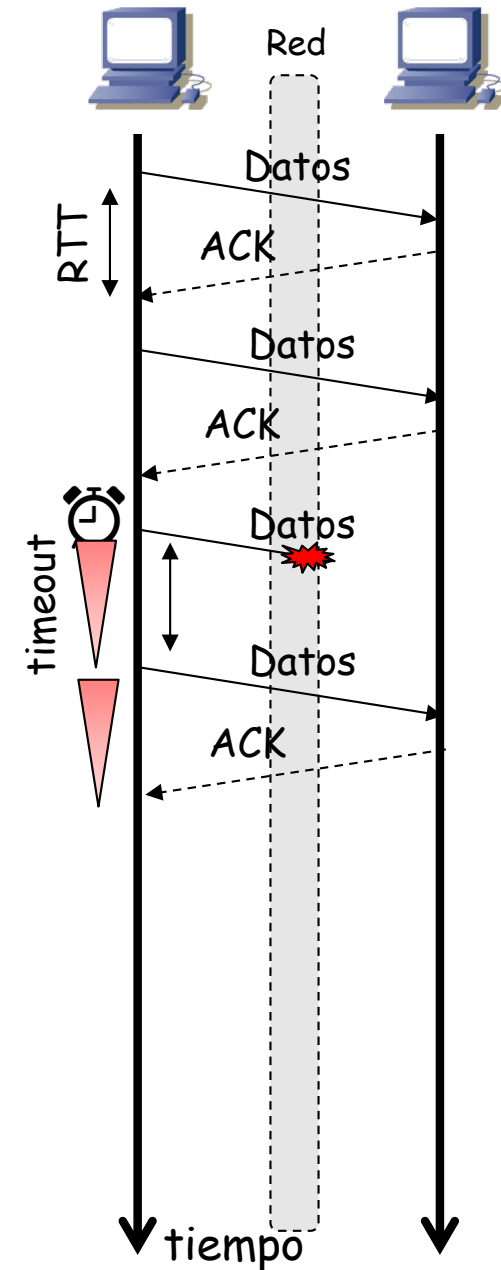


Escenario ideal

- Stop&Wait ofrece baja eficiencia para enlaces de alta capacidad y alto RTT
- Es razonable cuando el tiempo de transmisión del paquete es comparable al RTT
- Ejemplo LAN:
 - Paquete de 1518 bytes @ 10Gb/s son $1.2 \mu\text{s}$ para transmitirse
 - $1.2 \mu\text{s}$ @ 220.000 Km/s recorre 267m
 - En el entorno LAN podríamos saturar 10Gb/s con stop&wait
- Ejemplo WAN:
 - Cruzar el Atlántico pueden ser 6.000Km, o unos 30ms (¡en un sentido!)
 - RTT 60ms, Stop&wait, paquete de 1518bytes máximo de 200Kb/s, un enlace de 1Gb/s tiene una utilización del 0.02%
 - Un flujo de transporte nunca saturaría ese enlace

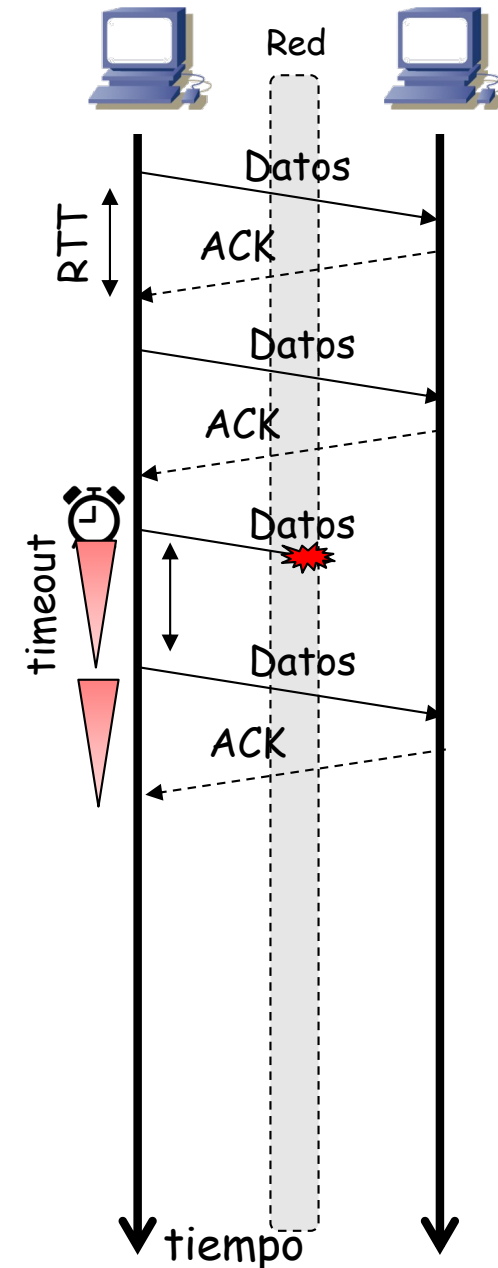
Con pérdidas

- 1 paquete cada RTT si no hay pérdidas
- Si hay pérdida tarda $\text{timeout} + \text{RTT}$
- Timeout ideal sería de un RTT
- Así que ante pérdida (simple) el paquete tarda $2 \times \text{RTT}$
- Ante pérdida doble tardaría $3 \times \text{RTT}$



Tiempo de transferencia

- Supongamos probabilidad de pérdida p
- Independientes
- Con probabilidad $1-p$ llega con éxito
- Con probabilidad p se pierde
- Entonces un RTT después se retransmite
- ¿Cuál es la probabilidad de que llegue tras una retransmisión?
 - $p(1-p)$
 - Se pierde el primero y llega el segundo
- ¿Probabilidad de que llegue tras 2 retx?
 - $p^2(1-p)$
 - Se pierden original y 1ª retx, llega la 2ª
- ¿qué llegue tras n retx?
 - $p^n(1-p)$



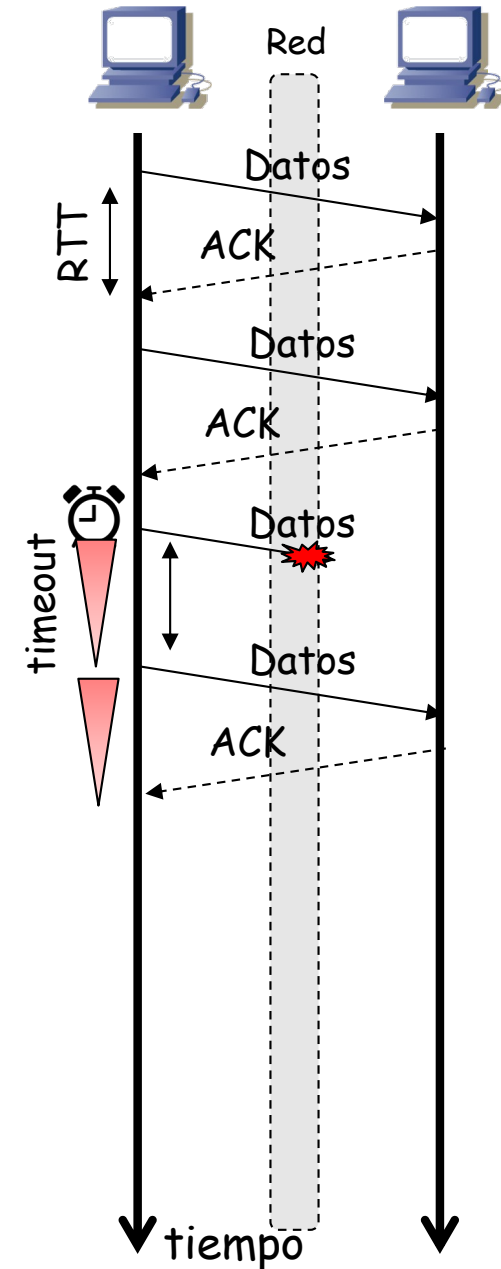
Tiempo de transferencia

Probabilidad

Tiempo

- $1 - p$
- $p(1 - p)$
- $p^2(1 - p)$
- $p^3(1 - p)$
- ...
- $p^{n-1}(1 - p)$
- $nRTT$
- Es una distribución geométrica
- Media: $\frac{RTT}{1-p}$
- Throughput medio para paquetes con L bits:

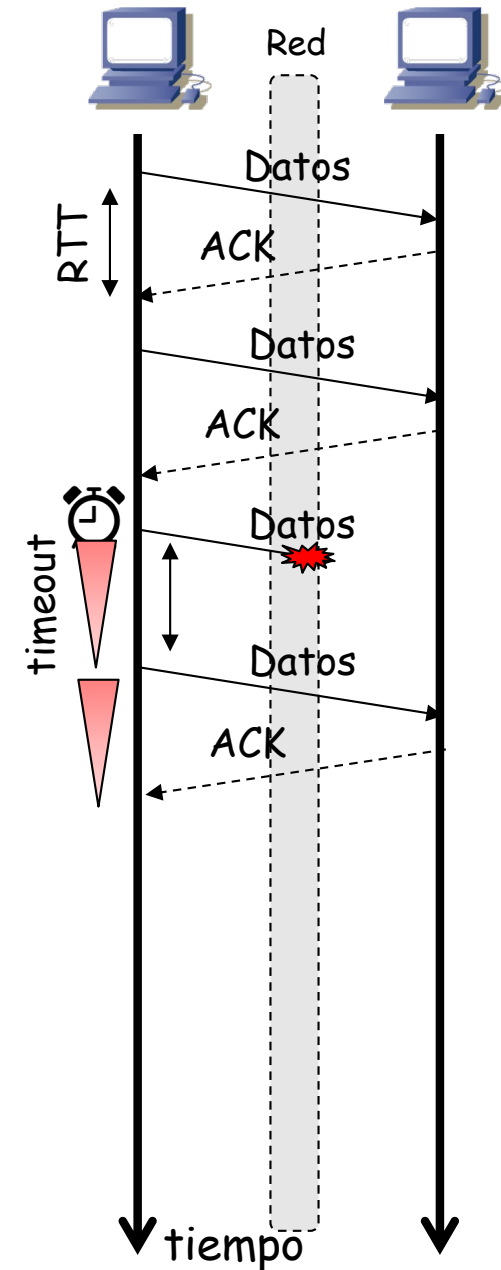
$$v = L \frac{1 - p}{RTT}$$
- Ejemplo:
 - $L = 1460$ bytes, $p=0.01$, $RTT=40ms$
 - $v = 1460 \times 8 \times (1-0.01) / 0.04 = 289$ Kb/s



Tiempo de transferencia

- $\frac{RTT}{1-p}$ es el tiempo medio para que un paquete llegue al destino correctamente
- Transferencia de M paquetes, tiempo medio:

$$M \frac{RTT}{1-p}$$



Para pensar

- ¿Depende la probabilidad de pérdida del tamaño del paquete?
- ¿Es razonable considerarlas independientes?
- Paquetes más pequeños hacen que la misma cantidad de datos requiera más paquetes; ¿cómo afecta al tiempo total?
- ¿Cómo afecta a estos cálculos que las pérdidas sean de las confirmaciones?