TCP y UDP

Álvaro González Sotillo

21 de marzo de 2022

Índice

1. Introducción

- TCP y UDP son de la capa de transporte
- Es la primera que une dos entidades (procesos), en vez de dos hosts
- Suele ser la primera capa visible para los programadores de aplicaciones
  - IP se puede considerar la *frontera* entre los *administradores* y los *programadores*

2. UDP

- *User Datagram Protocol*
- Funciones:
  - Entregar un datagrama entre el emisor y receptor (procesos)
  - Detección de errores

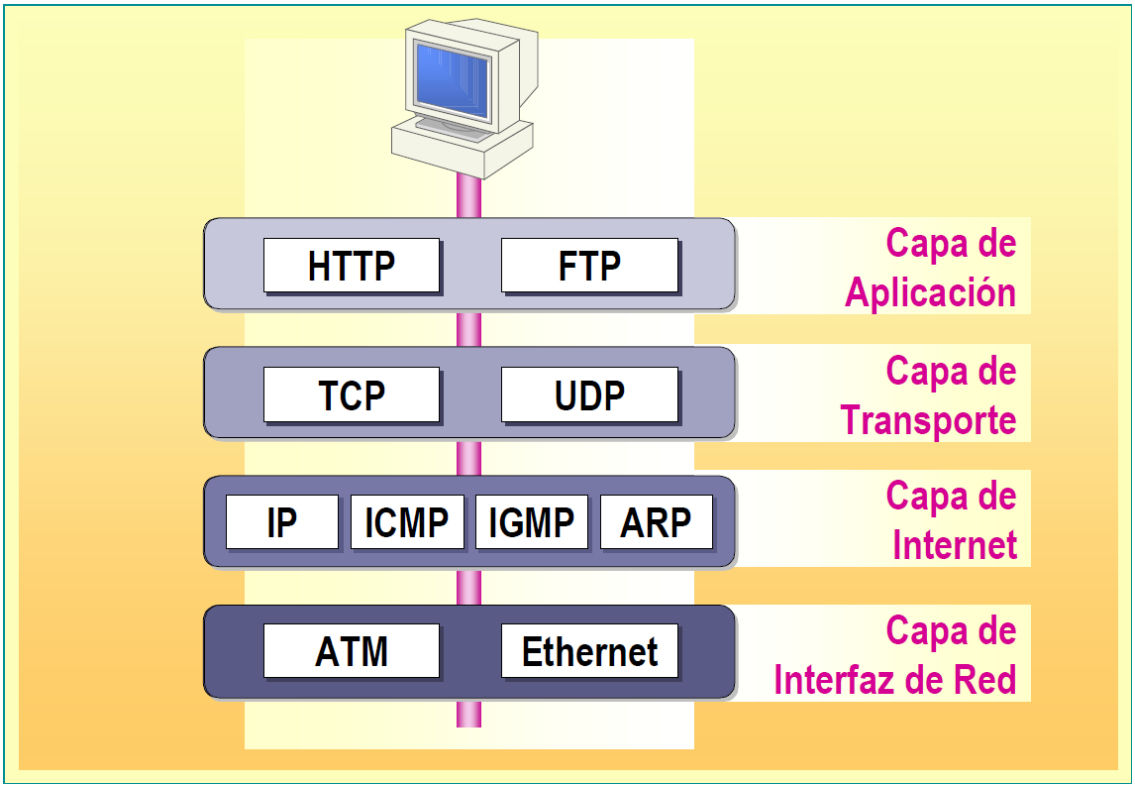


Figura 1: Esquema de niveles de red

2.1. Formato de trama UDP

	Bits 0 - 15	16 - 31
0	Source Port	Destination Port
32	Length	Checksum
64	Data	

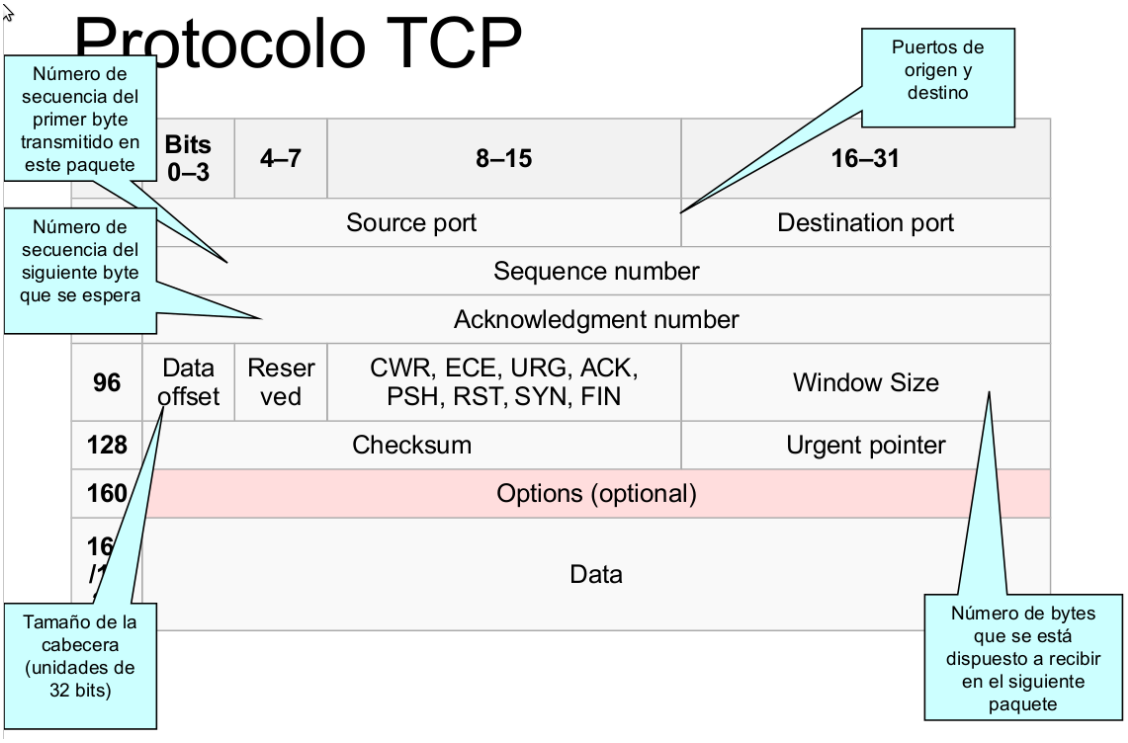
2.2. Características de UDP

- Los datagramas pueden llegar en un orden diferente al enviado (si IP elige rutas distintas para ellos)
- El emisor no tiene la seguridad de que los datagramas llegan al receptor
- Por tanto, no se utilizan **conexiones** ni es **confiable**. Cada datagrama se envía de forma independiente.

- ¿Cuántos puertos hay?
- ¿Cuál es el tamaño máximo de un datagrama UDP?

3. TCP

- *Transmission Control Protocol*
- Asegura que la transmisión se realiza por un medio fiable
- Garantiza la recepción de los mensajes en orden correcto
- Garantiza al emisor que los mensajes llegan correctamente al receptor
- Por tanto, es orientado a **conexión** y **confiable**.



3.1. Ventana y corrección de errores

- TCP necesita confirmación de cada mensaje enviado
  - Para garantizar la confiabilidad
- Opciones:
  - **Parada y espera**: Cada mensaje necesita confirmación
  - **Piggybacking**: La confirmación puede retrasarse algunos mensajes (ventana)
- La parada y espera es más simple, pero desaprovecha ancho de banda

4. *Sequence number* y *Acknowledgment number*

- **TCP** intenta que la comunicación se asemeje a un flujo de bytes
  - Todos los bytes que entran por un extremo
  - ... deben salir por el otro lado
- En **cada paquete** se envía
  - el número de secuencia del primer byte transmitido en el paquete
  - el número de secuencia del siguiente byte que se espera

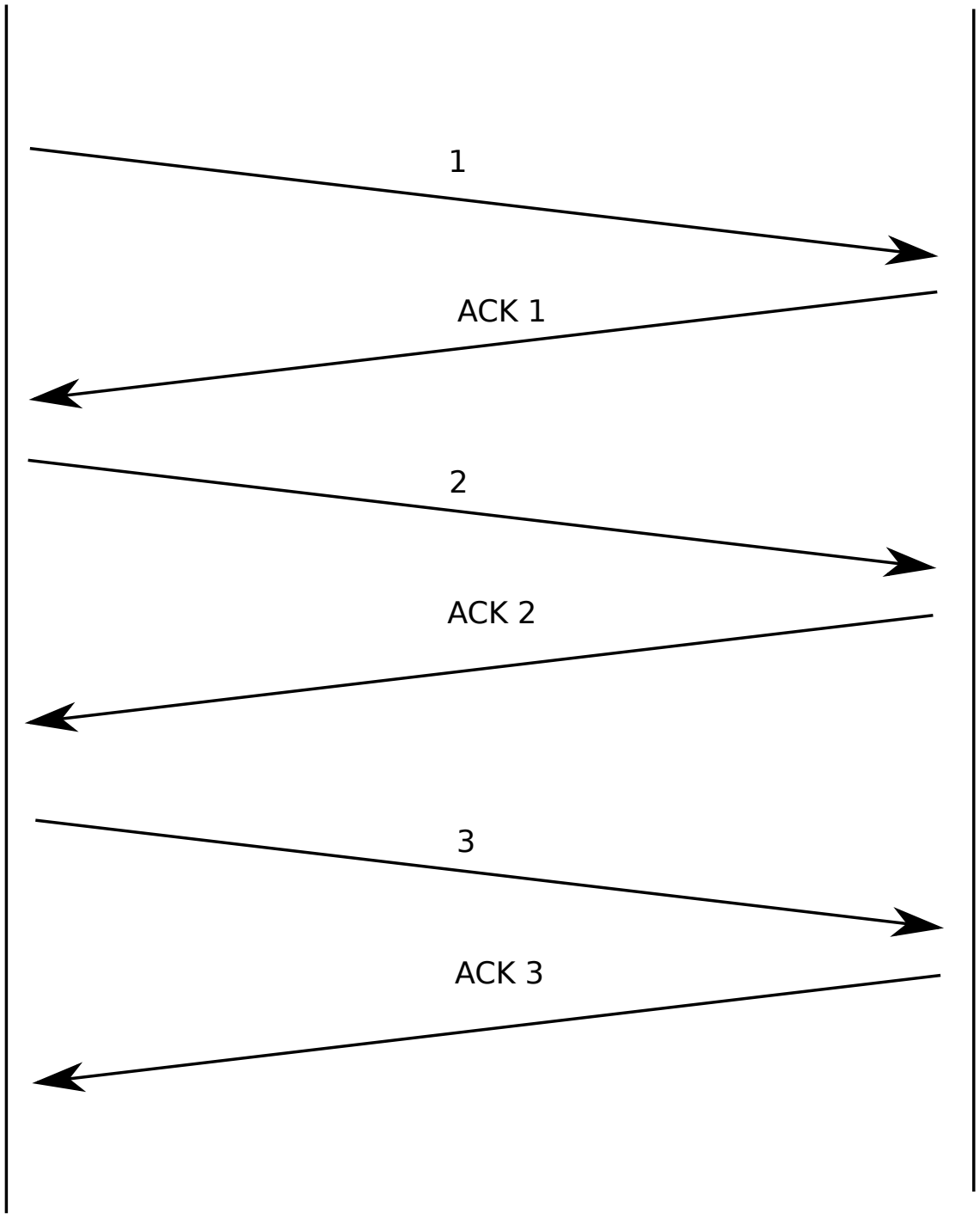
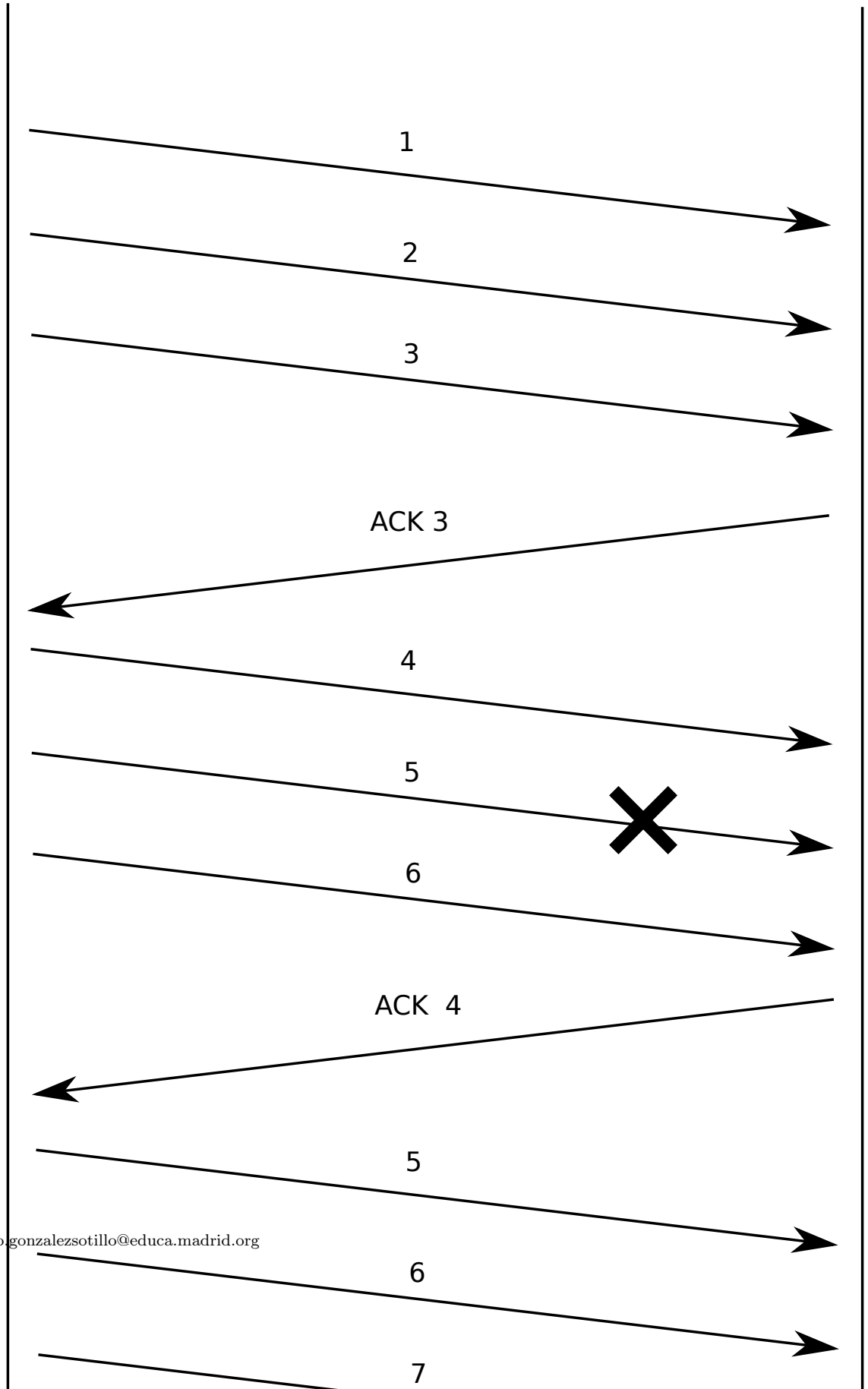


Figura 2: Ejemplo de parada y espera


1
2
3
ACK 3
4
5
6
ACK 4
5
6
7

4.1. Estado de la conexión

- Cada extremo de la comunicación debe saber:
  - Cuál es su siguiente byte a enviar (*sequence number*)
  - Cuál es su siguiente byte a recibir (*acknowledgment number*)
- Además, también lleva la cuenta de su opinión acerca del *sequence number* y del *acknowledgment number* del otro extremo

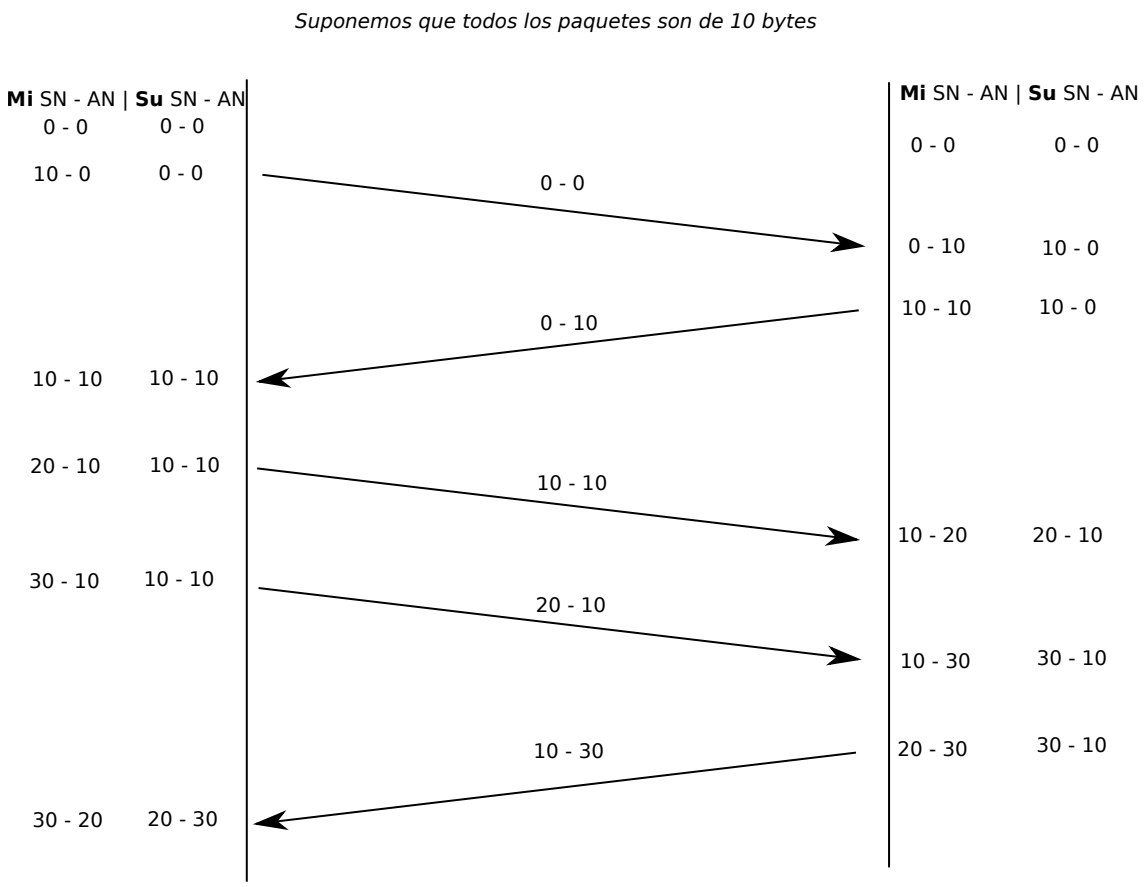


Figura 4: *Sequence number* y *acknowledgment number*

4.2. Corrección de errores

- Si se recibe un *sequence number* posterior a nuestro *acknowledgment number*
  - Es un paquete **posterior** al que esperamos
    - Se puede guardar en la capa **TCP** hasta que lleguen los anteriores
    - O se puede ignorar, y reclamar los paquetes perdidos enviando un *acknowledgment number* menor que el que espera el otro

Suponemos que todos los paquetes son de 10 bytes

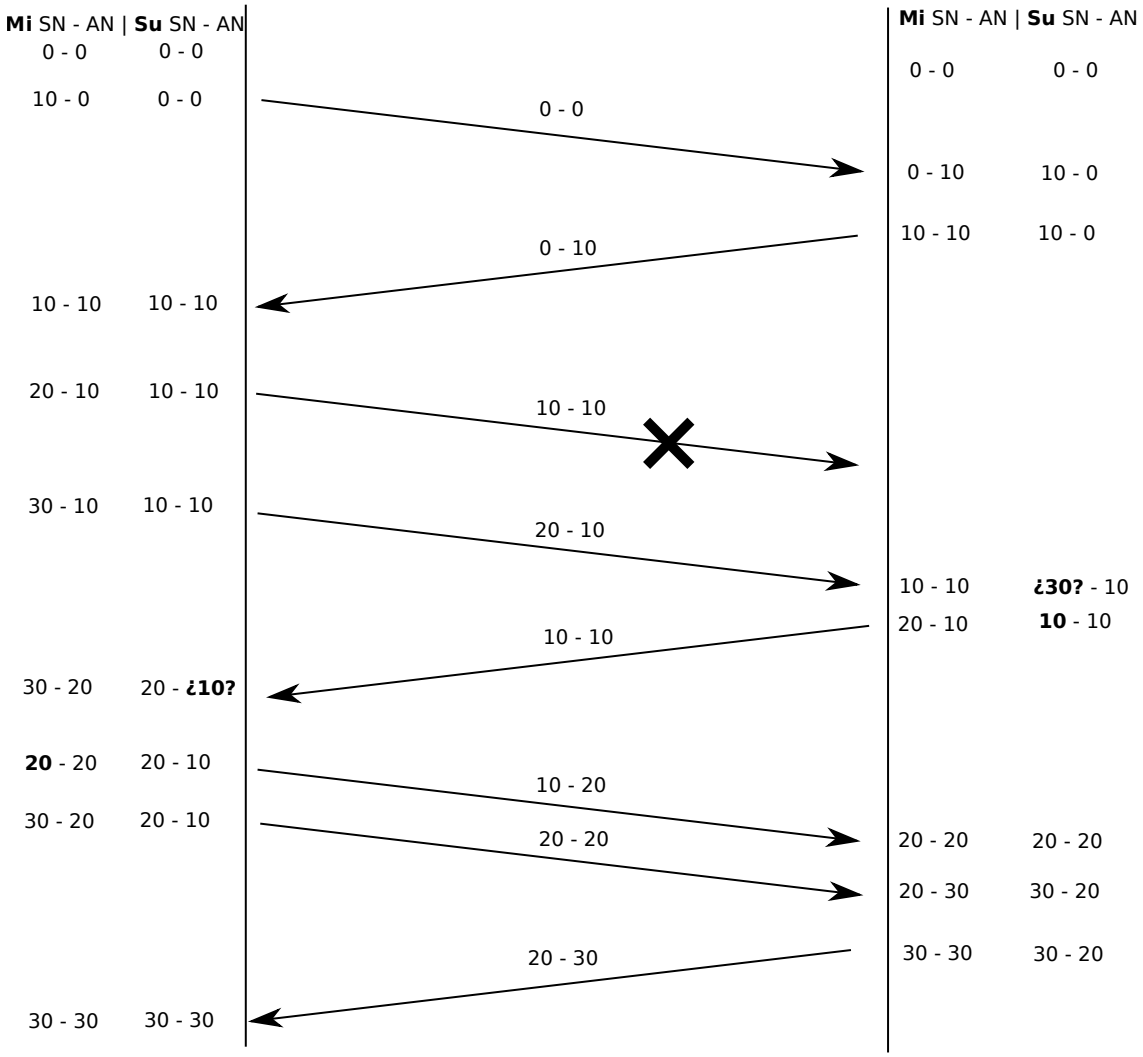


Figura 5: Llegada de un paquete posterior al esperado

- El otro lado reenviará los paquetes necesarios

- Si se recibe un *sequence number* anterior a nuestro *acknowledgment number*
  - Es un paquete ya recibido (se habrá duplicado)
  - Por tanto, se ignora
- Si me llega un *acknowledgement number* menor que los bytes que yo he enviado
  - Reenviaré a partir de dicho *acknowledgement number*

Suponemos que todos los paquetes son de 10 bytes

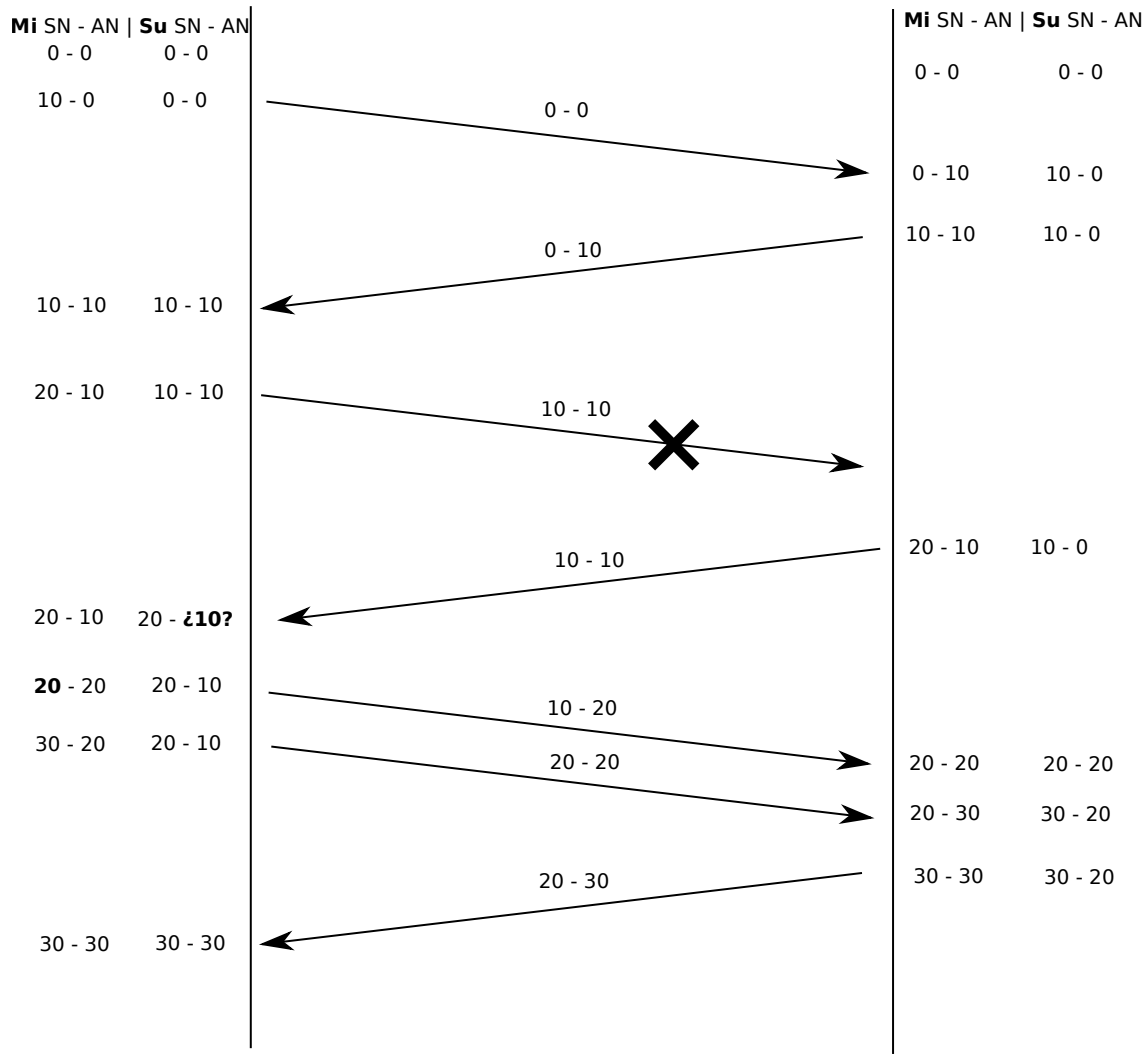


Figura 6: El otro lado informa de la pérdida de algún paquete

- Si no tengo confirmación de un paquete enviado tras un *timeout*
  - Reenviaré el paquete

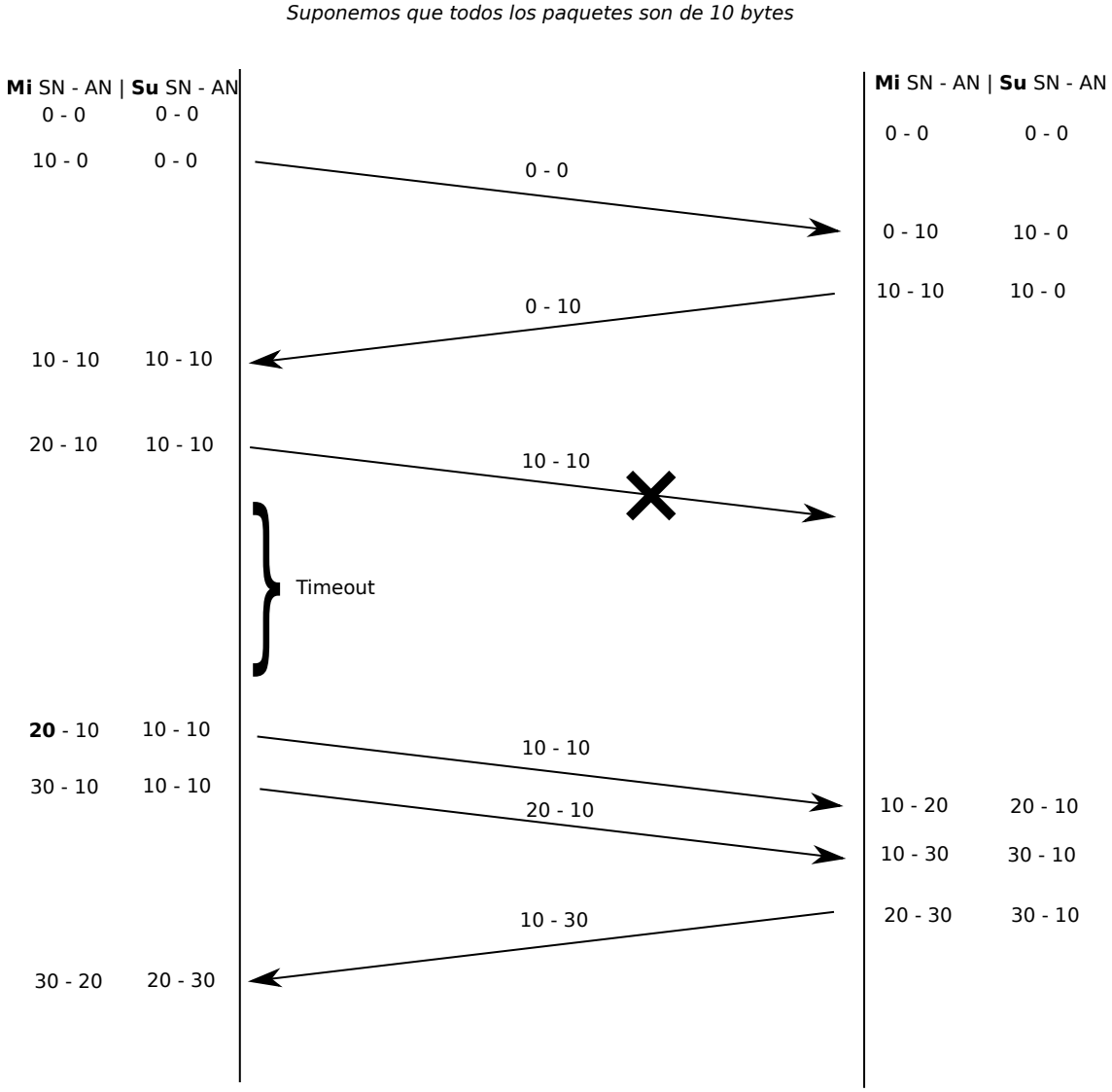


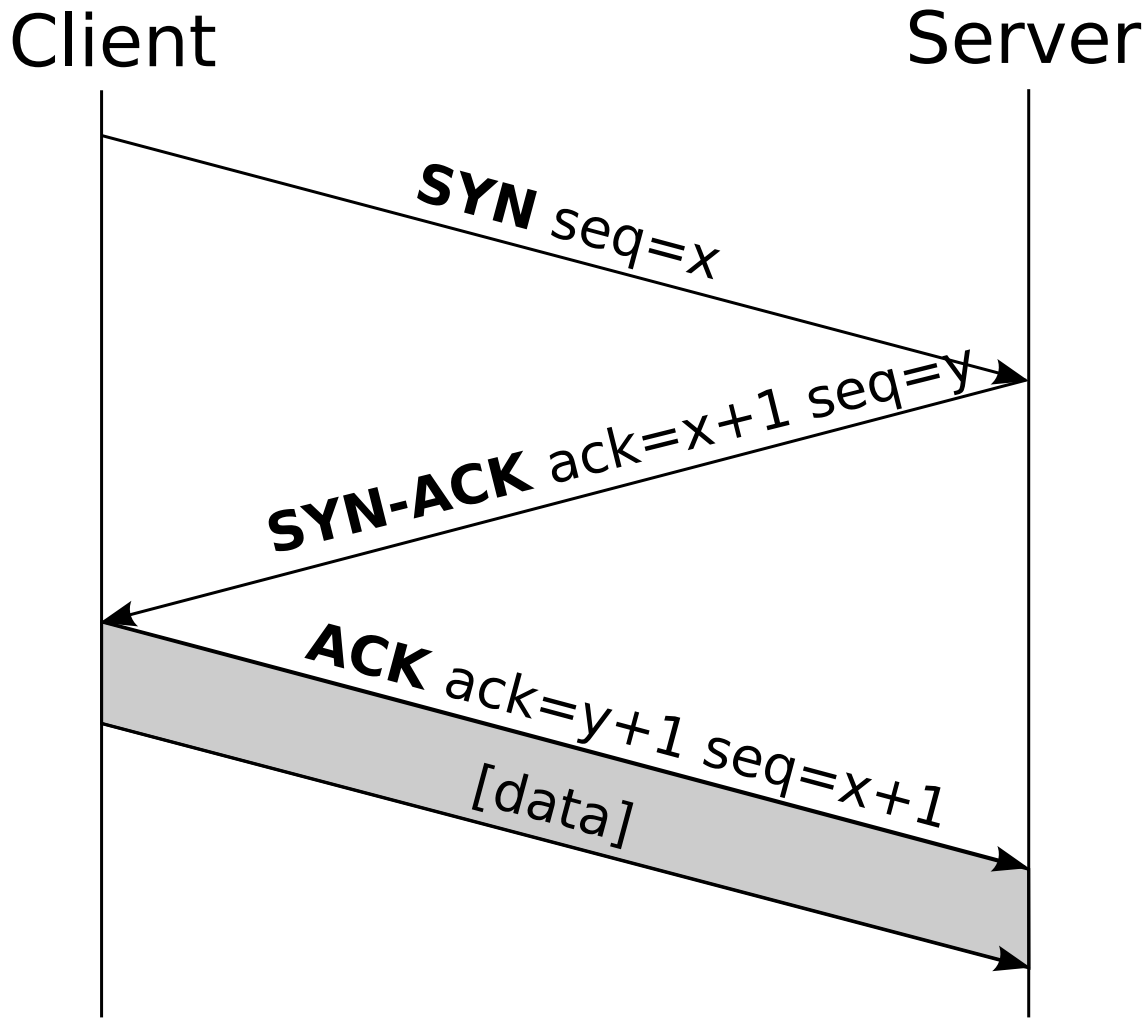
Figura 7: Paquete no confirmado tras un *timeout*

5. Mensajes TCP

5.1. Establecimiento de conexión

1. Un servidor escucha en un puerto

2. Un cliente envía una solicitud de conexión
3. El servidor responde con una aceptación de la conexión
4. El cliente *acepta* la aceptación

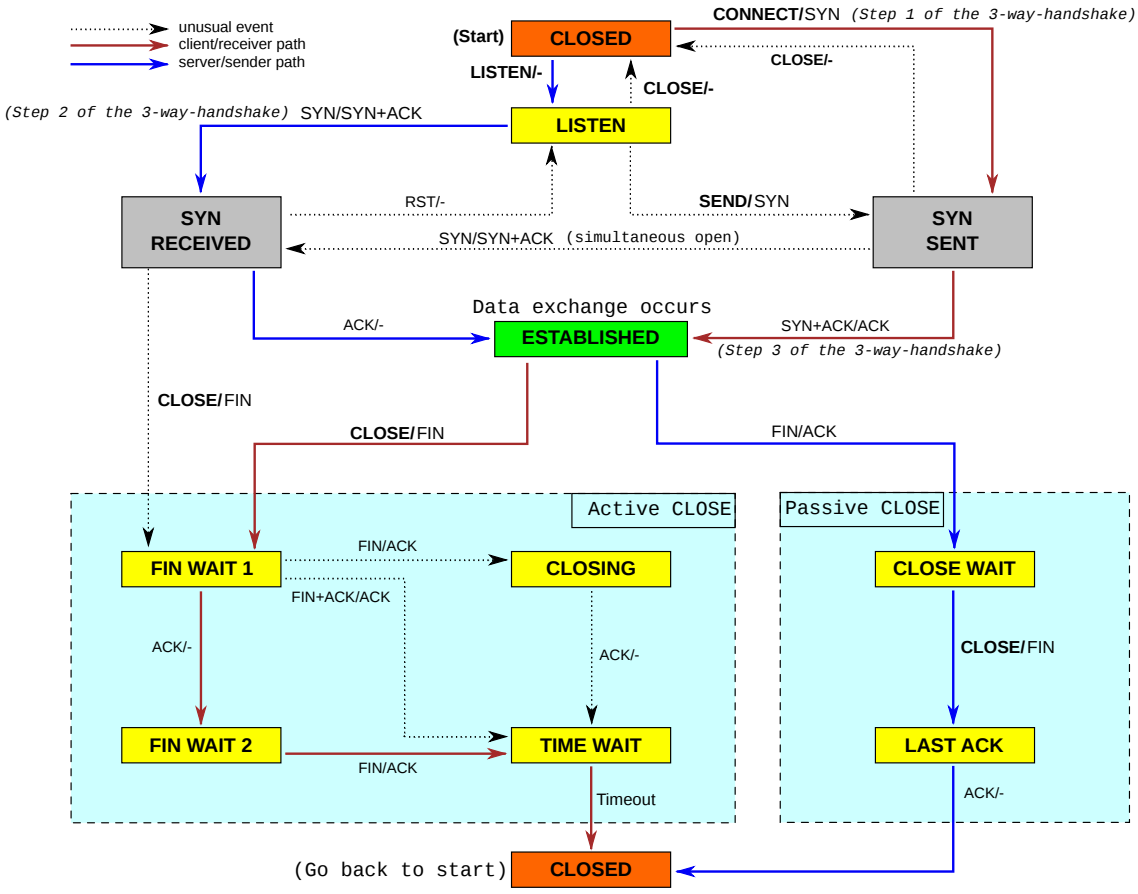


Fuente: By Snubcube

5.2. Estados TCP

- Los principales estados son:
  - **Closed**: Ninguna conexión
  - **Listening**: Un servidor está esperando en un puerto a ser conectado
  - **Established**: Un cliente ha conectado con un servidor
  - **Time wait**: Esperando a que la conexión termine

- Todos los estados en http://www.medianet.kent.edu/techreports/TR2005-07-22-tcp-EFSM.pdf



Fuente:By Scil100

6. Puertos

- Se llama puerto a la dirección de nivel de transporte en
  - TCP
  - UDP
- La asignación de puertos se realiza según el RFC 1700

Well-known ports	0 - 1023	Solo para el administrador
Registered ports	1024 - 49151	Generalmente, servicios menos críticos
Dynamic ports	49152 to 65535	Asignables a los clientes

6.1. Asignación de puertos

- **Servidor**: El proceso escucha en un puerto conocido

- Ejemplos: 80 para HTTP, 25 para SMTP...
- Lista de *well known ports* en el fichero `/etc/services`

- **Cliente**: El cliente inicia una conexión un servidor.
  - El sistema le asigna un puerto no utilizado cualquiera
  - Generalmente, un puerto dinámico
  - El cliente también puede solicitar un puerto, pero es poco frecuente

6.2. Comando `netstat`

- Informa de
  - Las conexiones TCP y UDP activas
  - Los programas escuchando en puertos TCP y UDP

	Linux	Windows
Conexiones TCP	`-t`	`-p tcp`
Conexiones UDP	`-u`	`-p udp`
Proceso	`-p`	`-o`
No traducir direcciones	`-n`	`-n`
Escuchando	`-l`	
Escuchando y establecidas	`-a`	`-a`

- Si no está `netstat`, se puede usar `ss`
  - `ss --process --numeric --tcp --all`

6.2.1. Ejercicio

- Comprueba qué conexiones están establecidas en tu ordenador
- Comprueba a qué direcciones está conectado tu ordenador
- Comprueba qué puertos admiten conexiones
- Comprueba qué procesos admiten conexiones

6.3. Comando `nc`

- Netcat permite realizar conexiones TCP/UDP y redirigir su entrada/salida
- Usos
  - Simular de forma rápida un cliente para probar un servidor
  - Simular un servidor para probar un cliente
  - Comprobar si el *firewall* permite conexiones
  - Transferir información por red
- Versión Windows:

- https://eternallybored.org/misc/netcat/

- Otras opciones para Windows
  - Máquina virtual con Linux
  - powercat
  - Windows Subsystem for Linux

6.3.1. Ejercicio

- Pon a **netcat** a escuchar en un puerto
- Haz que un compañero se conecte a ese puerto
- Utiliza **netcat** como chat.

6.3.2. Youtube, 1990

```
nc towel.blinkenlights.nl 23
```

6.3.3. Gmail, 1990

```
telnet bbs.fozztexx.com
```

7. Direcciones de escucha

- Cuando un proceso escucha en un puerto, también elige en qué dirección de red escucha
- La dirección IP se puede utilizar como un *firewall* rudimentario

Dirección	Efecto
`0.0.0.0`	Escucha en todas las direcciones IP accesibles
`127.X.X.X`	Escucha en una dirección local
`X.X.X.X`	Escucha en una interfaz de red concreta

- No todas las combinaciones son posibles
  - Es posible escuchar en el mismo puerto en `127.X.X.X` y en `X.X.X.X`
  - `0.0.0.0` no es compatible con ningún otro

7.1. Posibilidades

- El servidor está accesible a todo el mundo
- El servidor solo está disponible desde la máquina local (por ejemplo email local)
- El servidor solo está disponible por una de las interfaces de red
- Diferentes servidores en diferentes interfaces de red

7.2. ¿Más de un proceso escuchando en el mismo puerto?

- Dependiendo de la versión de Linux/Windows, más de un proceso puede escuchar en el mismo puerto
- Se hace para repartir mejor múltiples conexiones de clientes entre las CPU del servidor

```
SO_REUSEPORT (since Linux 3.9)

Permits multiple AF_INET or AF_INET6 sockets to be
bound to an identical socket address. This option
must be set on each socket (including the first
socket) prior to calling bind(2) on the socket. To
prevent port hijacking, all of the processes
binding to the same address must have the same
effective UID. This option can be employed with
both TCP and UDP sockets.
```

8. TCP vs UDP

- TCP es un medio de transmisión asegurado
  - Las aplicaciones que usan TCP no envían paquetes, sino bytes.
  - TCP decide cuando enviar un paquete (las aplicaciones pueden *opinar*)
  - Consume más CPU y memoria, por la ventana de emisión y los reenvíos
  - Reduce la *transferencia útil*
- UDP es más eficiente
  - No necesita mantener conexión, ni reordenar paquetes, ni retransmitir paquetes
  - Las aplicaciones son *conscientes* de que se envían paquetes, no bytes.
  - En redes con pocos errores, puede ser más adecuado
  - Interesante cuando se necesita mucha transferencia útil pero no importa perder algún paquete (voz, vídeo)

8.1. TCP Joke

1. Hello, would you like to hear a TCP joke?
2. Yes, I'd like to hear a TCP joke.
3. OK, I'll tell you a TCP joke.
4. OK, I'll hear a TCP joke.
5. Are you ready to hear a TCP joke?
6. Yes, I am ready to hear a TCP joke.
7. OK, I'm about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline.

8. OK, I'm ready to hear the TCP joke that will last 10 seconds, has two characters, does not have a setting and will end with a punchline.
9. I'm sorry, your connection has timed out... ...Hello, would you like to hear a TCP joke?

8.2. UDP Joke

- Lo anterior era una broma de TCP. No me importa si la pillas o no.

9. Referencias

- Formatos:
  - Transparencias
  - PDF
  - EPUB
- Creado con:
  - Emacs
  - org-re-reveal
  - Latex
- Alojado en Github