centre adscrit a:

UNIVERSITAT DE BARCELONA

02 – Comunicación entre ordenadores: Sockets

Programación de juegos en red, 2015/16

Carmen Soler

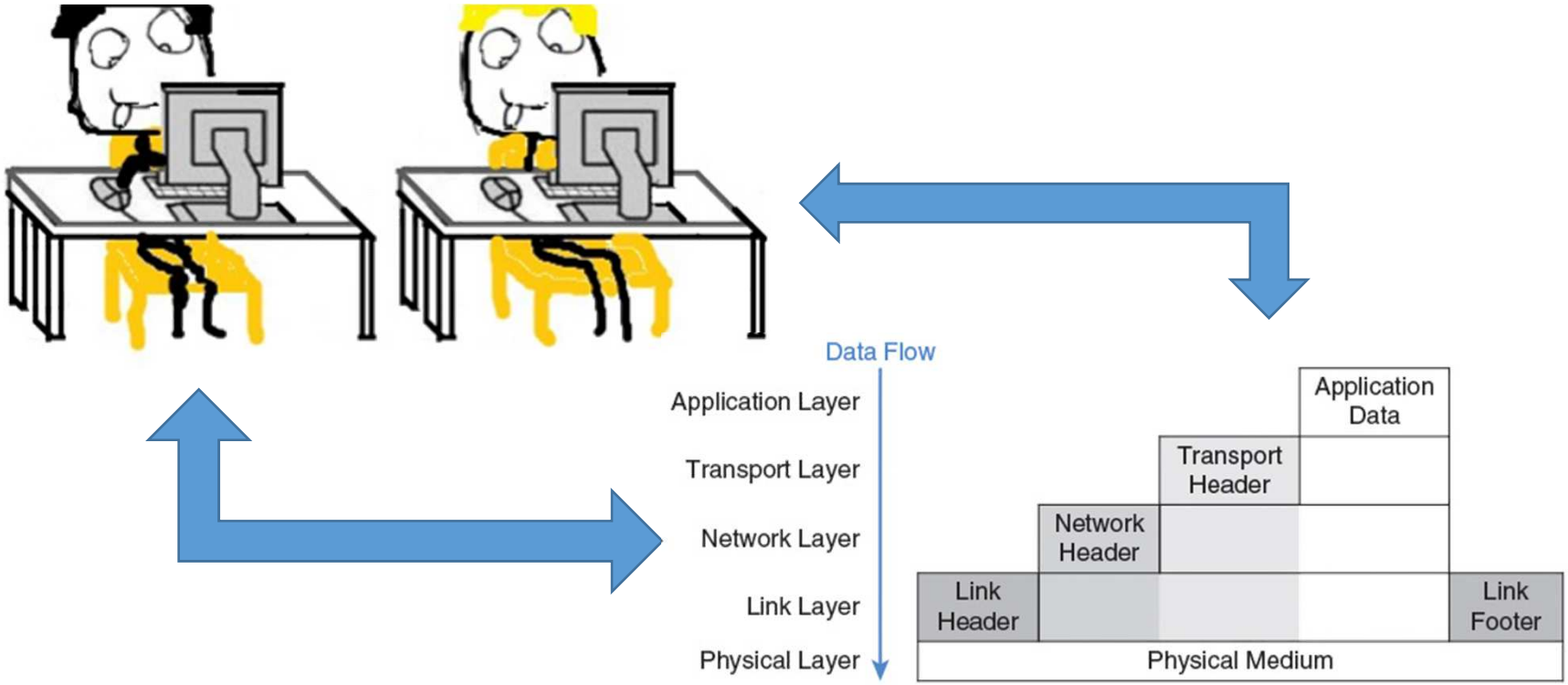
carmensolerchorro@enti.cat

Guion

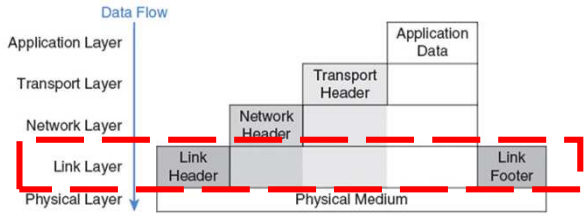
1. Internet
    1. ¿Qué hay detrás de los datos que viajan por internet?
    2. La capa de enlace (link layer)
    3. La capa de red (network layer)
        1. IPv4
        2. Fragmentación
        3. ARP
    4. La capa de transporte (transport layer)
        1. UDP
        2. TCP
2. Sockets
    1. ¿Qué son? ¿Para qué sirven?
    2. Berkeley Sockets - ¿Qué son?
    3. Cargar y descargar librería
    4. Crear sockets
    5. Destruir sockets
    6. SocketAddress
    7. Vincular un socket: BIND
3. Sockets UDP
    1. Características
    2. Enviar
    3. Recibir
4. Sockets TCP
5. Evitar el bloqueo

1.1. ¿Qué hay detrás de lo datos que viajan por internet?

- Como programadores de juegos en red, hemos de entender las particularidades de la transmisión de datos por Internet.
- Los datos se transmiten utilizando varias capas de protocolos y es interesante conocerlos para después programar en consecuencia.



1.2. La capa de enlace (Link Layer)



¿Cómo se identifican los hosts?

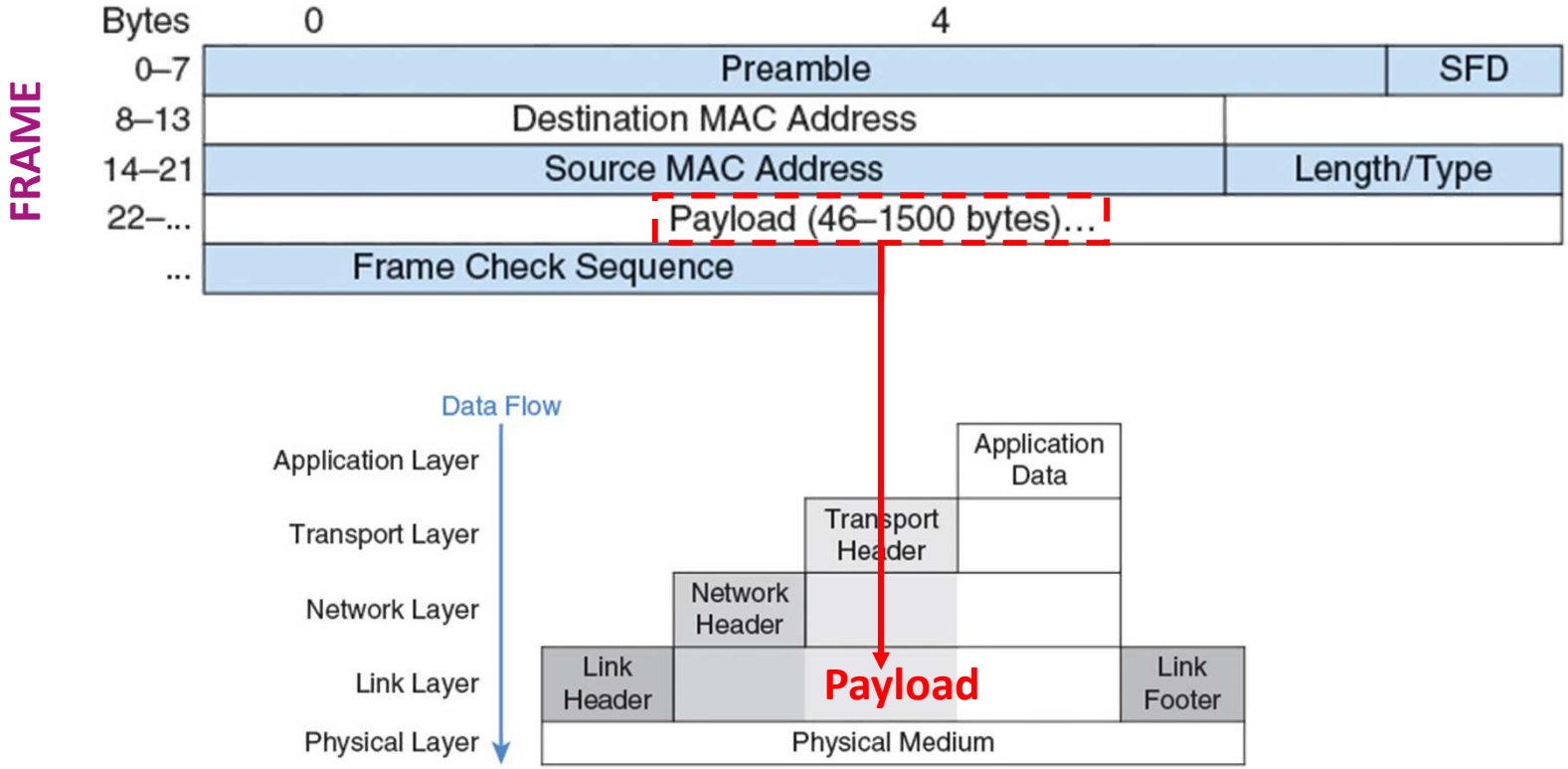
- Utiliza **direcciones MAC**.
- Tienen 48 bits y van asociadas a las tarjetas de red.
- Se graba por hardware en el momento de fabricación.
- Un ordenador no cambia de MAC a no ser que cambie su tarjeta de red.

¿Sabéis cómo consultar la MAC de vuestro ordenador sin tener que desmontarlo?

1.2. La capa de enlace (Link Layer)

¿En qué formato envío la información?

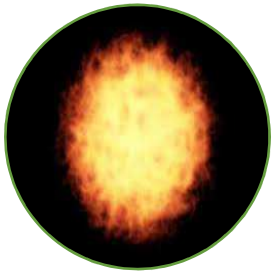
- Eso depende del protocolo de enlace que utilicemos.
- Para el caso concreto de **Ethernet/802.3** sería así:



1.2. La capa de enlace (Link Layer)

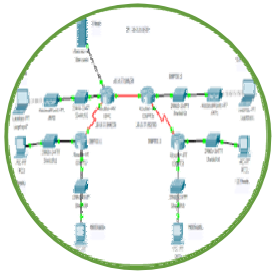
¿Podría ser suficiente una capa de enlace?

- Puedo identificar hosts y hacerles llegar paquetes. Parece que no debería necesitar nada más….
- Veamos **3 situaciones** en que esto no es así.



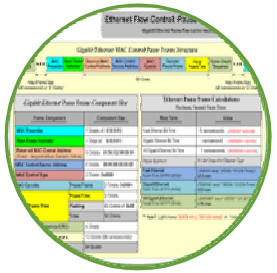
Situación 1: DESTRUCCIÓN

- Un servidor que recibe millones de visitas diarias a través de su MAC.
- Un día su tarjeta de red se destruye y hemos de poner otra.
- ¿Y ahora qué? Ya nadie sabe dónde está el servidor.



Situación 2: ALCANCE DEL FRAME

- Envío un frame a una determinada dirección destino.
- El paquete tratará de buscar al destinatario por toda la red mundial de ordenadores conectados
- Porque a que a nivel de enlace no se puede hacer subnetting.

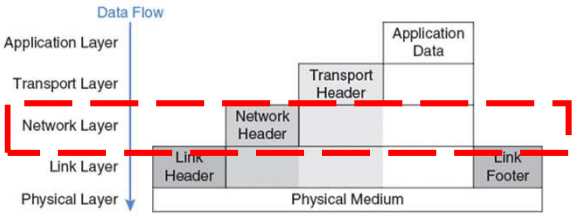


Situación 3: PROTOCOLOS DIFERENTES

- Quiero enviar un frame a un dispositivo que usa un protocolo de enlace diferente al mío.
- Utilizando sólo la capa de enlace no es posible.

Esta claro que la capa de enlace no es suficiente.
Necesitaremos subir de nivel.
CAPA DE RED

1.3. La capa de red (Network Layer)



¿Qué solucionamos con esta capa?

- Tenemos un **direccionamiento lógico** para cada host.
  - La dirección lógica es **independiente del HW**.
- Tenemos **un modo de definir subredes**.
  - Así **delimitamos** también **el espacio de direcciones físicas** a las que tiene alcance la capa de enlace.
  - El frame no se paseará por la red mundial de ordenadores, sólo por la subred definida desde la capa de red.
- En consecuencia, también ofrece un **modo de enviar datos entre subredes**.
- Un ejemplo de protocolo de red es **IPv4**

1.3.1. La capa de red (Network Layer): IPv4

IPv4 Header

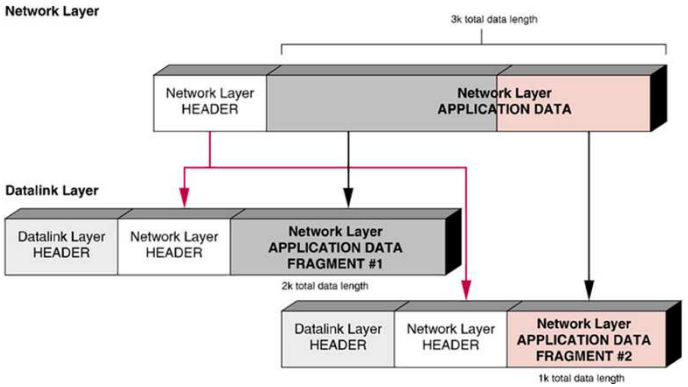
Bits	0			16
0–31	Version	Header Length	Type of Service	Total Length
32–63	Identification		Flags	Fragment Offset
64–95	Time to Live	Protocol	Header Checksum	
96–127	Source Address			
128–159	Destination Address			
160–...	Options			

65515 bytes máximo

Capa de enlace	Capa de red
• MTU de 1500 bytes	• Payload máximo de 65515

¿Cómo gestionamos esto?

FRAGMENTACIÓN



1.3.2. La capa de red (Network Layer): Fragmentación

Entonces, la fragmentación es genial

No importa el tamaño del paquete que enviemos, porque a nivel de enlace se crearán los paquetes necesarios para enviarlo.

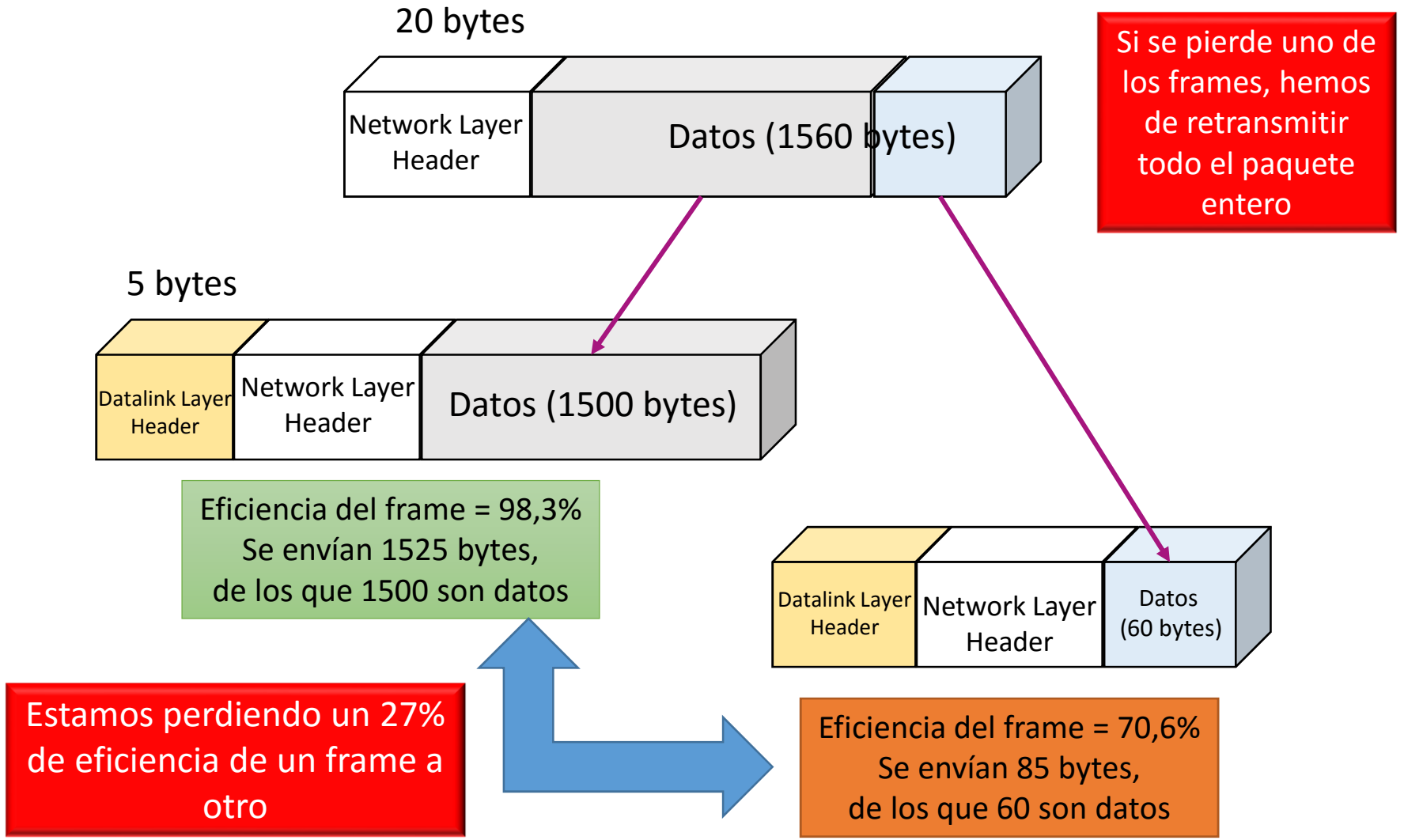
1.3.2. La capa de red (Network Layer): Fragmentación

Pues de eso nada:
La fragmentación tiene problemas de eficiencia

Al tener más frames → Más ancho de banda en cabeceras

Si se pierde sólo un frame de los que forma el paquete → Hemos de volver a retransmitir todo.

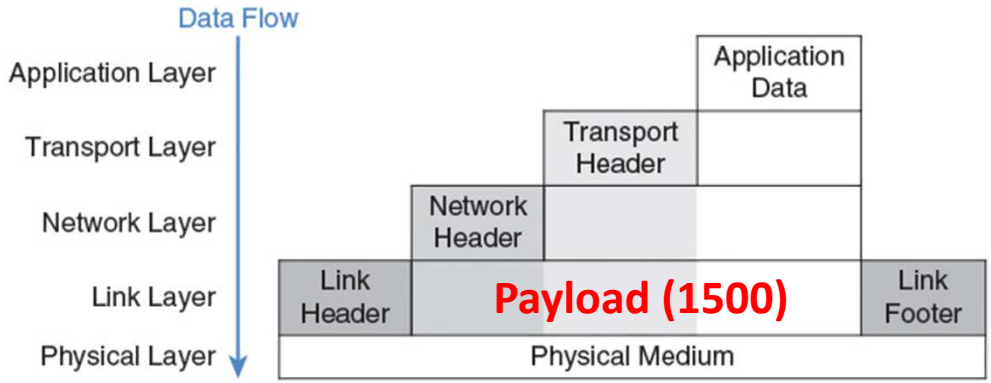
1.3.2. La capa de red (Network Layer): Fragmentación



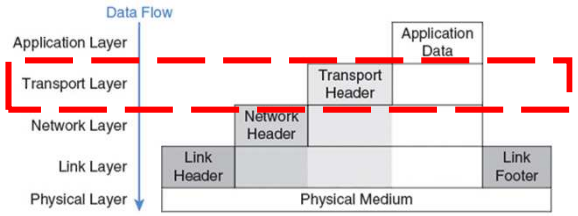
1.3. La capa de red (Network Layer)

Conclusiones sobre el tamaño de los paquetes

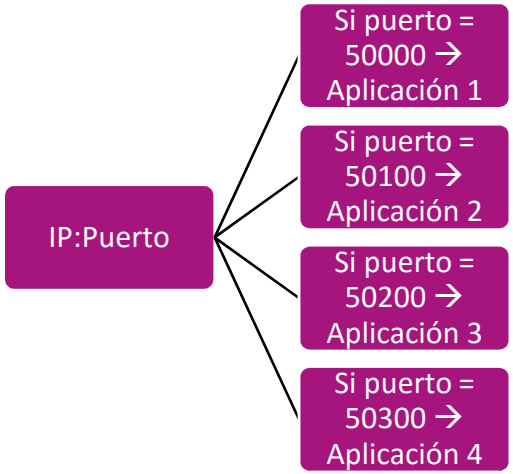
- Tratar de enviar **mensajes de un tamaño inferior al MTU**.
  - De hecho, como no sabemos por qué tipos de capas de enlace pasará nuestro paquete, es bueno poner un tamaño de paquete inferior a 1500.
  - **1300 bytes es un tamaño aconsejable** porque también se pierden bytes con las cabeceras de red y de transporte.
- **Conclusión**: Transmitir datos en **paquetes cercanos a 1300 bytes y sin pasarnos**.



1.4. La capa de transporte (Transport Layer)



¿Por qué una capa de transporte?



- Bien… el paquete llega al host destino… ¿Y **para cuál de las n-mil aplicaciones abiertas es el paquete**?
- Necesitamos usar **PUERTOS**.
- Cada aplicación <-> Un puerto.
- El puerto se suele representar después de la IP. Por ejemplo: 18.19.20.21:80

1.4. La capa de transporte (Transport Layer)

Organización en la numeración de puertos

Del 0 al 1023

- Reservados para sistema.
- No es aconsejable utilizarlos.

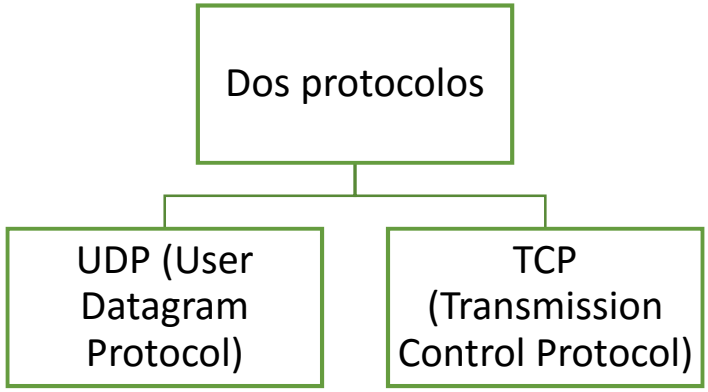
Del 1024 al 49151

- De usuario o reservados.
- Si diseñamos un protocolo/aplicación nueva, podemos pedir a la IANA (Internet Assigned Numbers Authority) que nos reserve un puerto para nosotros.

Del 49152 al 65535

- Puertos dinámicos.
- No pueden asignarse y los podremos utilizar para nuestros juegos.

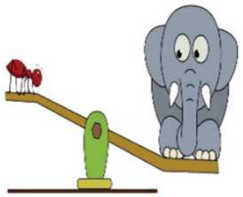
1.4. La capa de transporte (Transport Layer)



Bits	0	16
0–31	Source Port	Destination Port
32–63	Length	Checksum

Bits	0	4	7	16
0–31	Source Port			Destination Port
32–63	Sequence Number			
64–95	Acknowledgment Number			
96–127	Data Offset	Reserved	Control Bits	Receive Window
128–159	Checksum			Urgent Pointer
160–...	Options			

UDP
Protocolo ligero, pero nada fiable



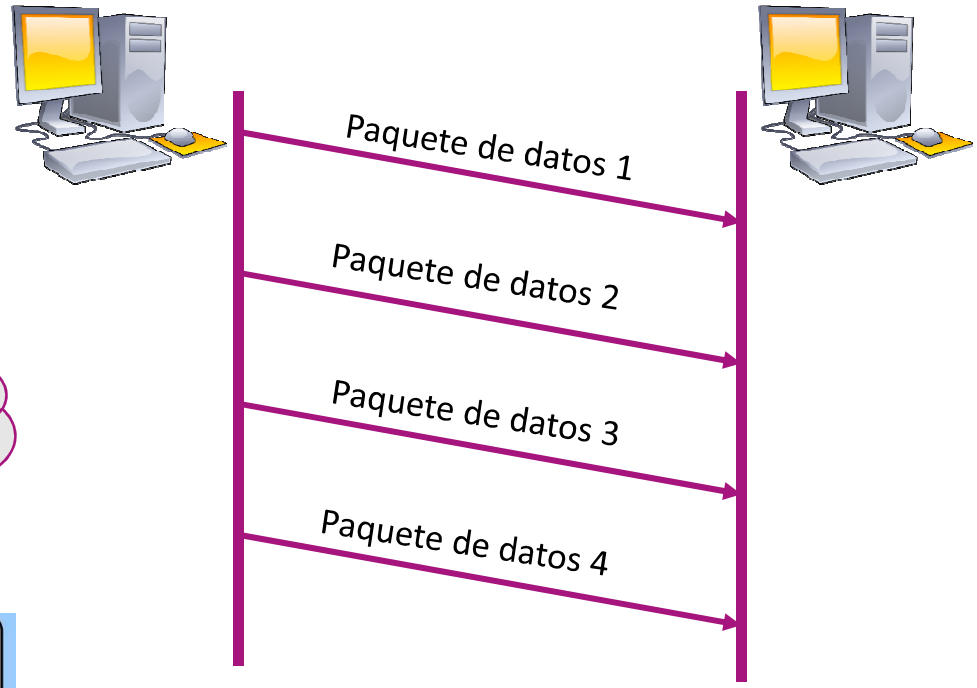
TCP
Protocolo pesado, pero totalmente fiable

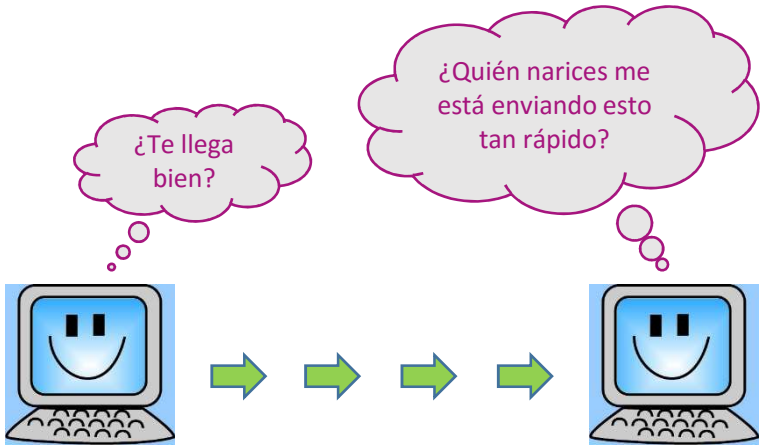
1.4.1. La capa de transporte (Transport Layer): UDP

UDP (User Datagram Protocol)

- Tiene sólo 8 bytes de cabecera.
- No asegura que el mensaje llegue a destino
- Se envía y ya está.

Bits	0	16
0–31	Source Port	Destination Port
32–63	Length	Checksum



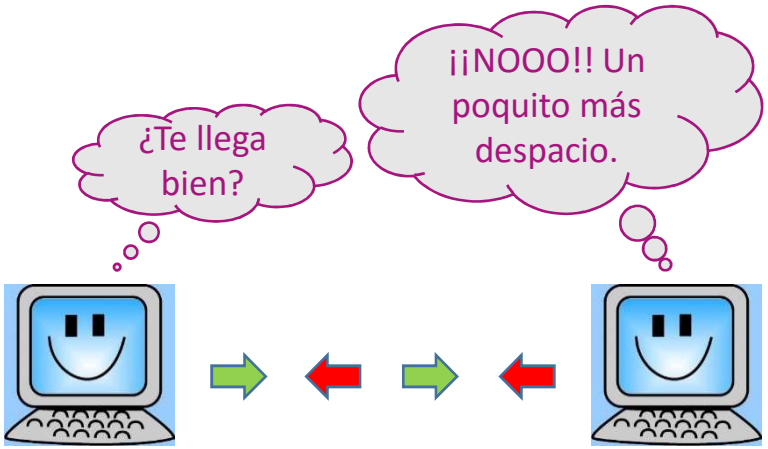


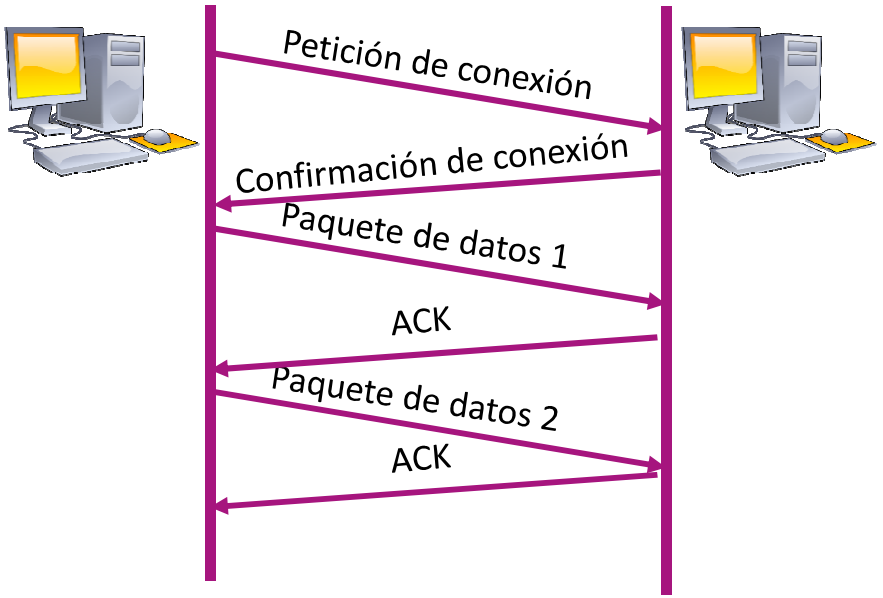
1.4.2. La capa de transporte (Transport Layer): TCP

TCP (Transmission Control Protocol)

- Crea una conexión persistente entre dos hosts.
- Todos los paquetes que se envían llegan y en orden. Como estar escribiendo en un fichero.
- Todo a base de cabeceras mucho más largas y de más tráfico de red.

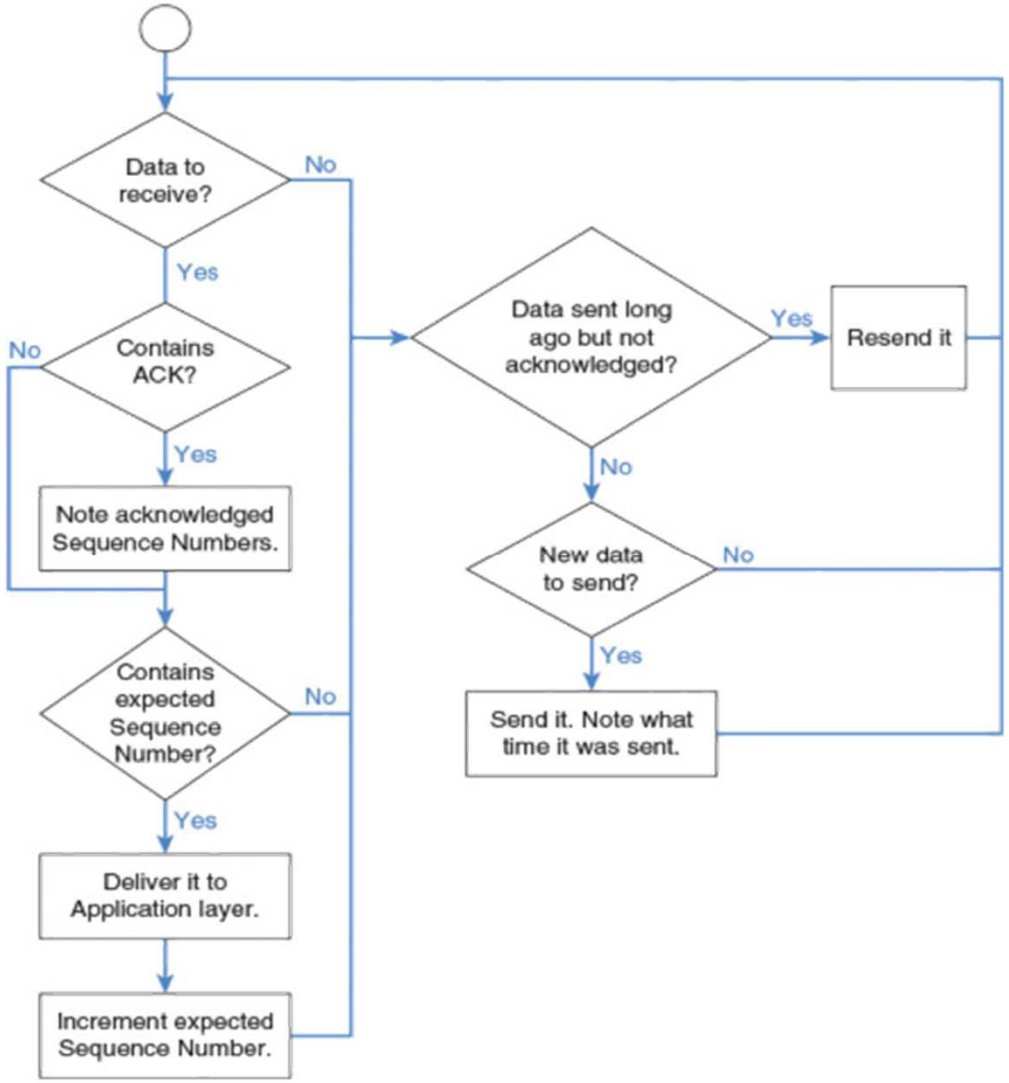
Bits	0	4	7	16
0–31	Source Port			Destination Port
32–63	Sequence Number			
64–95	Acknowledgment Number			
96–127	Data Offset	Reserved	Control Bits	Receive Window
128–159	Checksum			Urgent Pointer
160–...	Options			





1.4.2. La capa de transporte (Transport Layer): TCP

Sólo una pequeña idea de cómo se organiza el envío de datos en TCP



Conclusiones sobre cómo viajan los datos por internet

Enviar mensajes cercanos a 1300 bytes.

Usamos puertos: Vigilar de un usar puertos de otra aplicación.

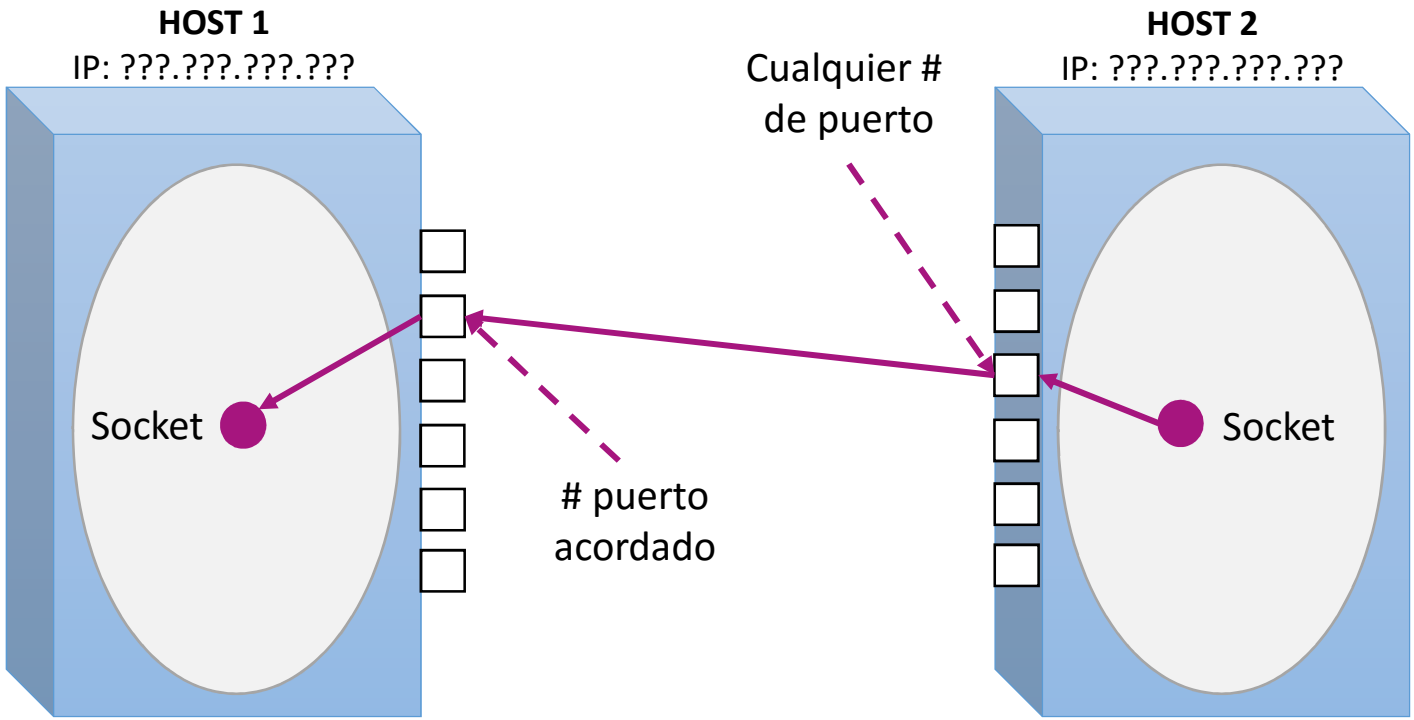
2 protocolos:

- **UDP**: Cabeceras cortas pero poco fiable.
- **TCP**: Cabeceras largas pero totalmente fiable. Se establece una conexión persistente entre hosts.

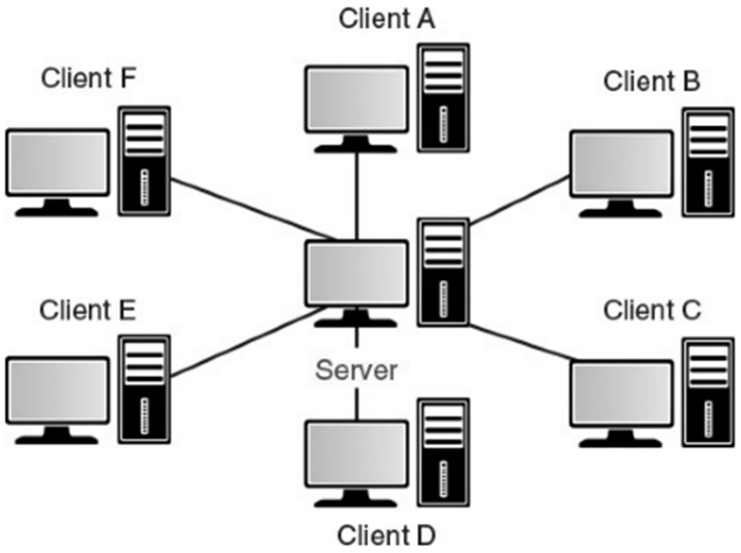
CAMBIO DE TEMA
Empezamos con sockets

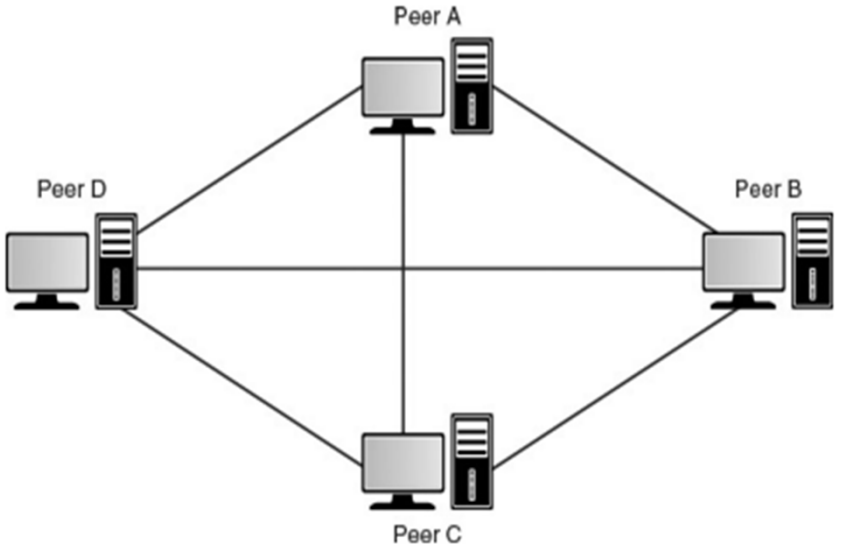
2.1. Sockets: ¿Qué son? ¿Para qué sirven?

- Los sockets son un mecanismo que permite comunicar dos procesos, generalmente ubicados en dos máquinas diferentes.



2.1. Sockets: ¿Qué son? ¿Para qué sirven?





Cliente / Servidor

- Cada cliente mantiene un socket
- El servidor mantiene N sockets o bien, gestiona N comunicaciones a través de socket.

Peer to Peer

- Cada cliente mantiene N-1 sockets o gestiona N-1 comunicaciones a través de socket.

2.2. Berkeley Sockets - ¿Qué son?

- Berkeley Socket **es la API más utilizada para implementar la parte de red** en el desarrollo de juegos multiplayer.
- La versión para Windows de esta librería es **Winsock2.h**
  - Existe una versión anterior: Winsock, que ahora está incluida en Windows.h
  - Para que funcione todo correctamente debemos incluir Winsock2.h y Windows.h
  - Como estas dos librerías juntas generan algún conflicto de funciones declaradas dos veces, antes de incluir Windows.h debemos definir la macro **WIN32_LEAN_AND_MEAN**

```
#define WIN32_LEAN_AND_MEAN
#include <WinSock2.h>
#include <Windows.h>
```

2.3. Cargar y descargar la librería

- **Winsock2** es una librería que **necesita que la carguemos** antes de utilizarla **y que la descarguemos** al acabar.
  - Hay que descargarla tantas veces como la hayamos cargado.

Cargar la librería

```
int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData);
```

- **wVersionRequested**
  - Word de 2 bytes para indicar la versión de Winsock.
  - En este momento vamos por la 2.2, así que pasaremos MAKEWORD(2, 2)
- **lpWSAData**
  - Es una estructura de Windows que almacena información acerca de la activación de la librería.
  - No lo necesitaremos.
- **Retorna** 0 si se ha activado correctamente. En caso contrario, retorna el código de error.

2.3. Cargar y descargar la librería

Descargar la librería

```
int WSACleanup();
```

- **Retorna** 0 si se ha activado correctamente. En caso contrario, retorna el código de error.
- Cuando se llama, se interrumpen todas las operaciones de sockets y se eliminan de memoria todos los sockets.
- Por eso es una buena práctica **cerrar todos los sockets correctamente antes de descargar la librería**

Capturar errores

```
int WSAGetLastError()
```

- Generalmente, cuando hay error en sockets se retorna -1 o bien SOCKET_ERROR (macro)
- Usamos esta función para obtener más información del error.
- **Retorna** sólo el código del último error generado. Por eso es importante usarla justo después de que se produzca un error.

2.4. Crear sockets

- Para crear socket:

SOCKET socket(int af, int type, int protocol)

- af
  - Indica de qué familia es la dirección de red
  - AF_INET o AF_INET6
- type
  - Indica el protocolo de transporte
  - SOCK_STREAM o SOCK_DGRAM
- protocol
  - Indica el protocolo específico para enviar paquetes.
  - Lo mejor es dejar que se coja el por defecto (0) en función del tipo.
- Si algo fue mal, el socket que retorna es = a la macro INVALID_SOCKET.

2.4. Crear sockets

Tipos de familias de dirección de red

Macro	Meaning
AF_UNSPEC	Unspecified
AF_INET	Internet Protocol Version 4
AF_IPX	Internetwork Packet Exchange: An early network layer protocol popularized by Novell and MS-DOS
AF_APPLETALK	Appletalk: An early network suite popularized by apple computer for use with its Apple and Macintosh computers
AF_INET6	Internet Protocol Version 6

Protocolo de transporte utilizado

Macro	Meaning
SOCK_STREAM	Packets represent segments of an ordered, reliable stream of data
SOCK_DGRAM	Packets represent discrete datagrams
SOCK_RAW	Packet headers may be custom crafted by the application layer
SOCK_SEQPACKET	Similar to SOCK_STREAM but packets may need to be read in their entirety upon receipt

Protocolo específico para enviar paquetes

Macro	Required Type	Meaning
IPPROTO_UDP	SOCK_DGRAM	Packets wrap UDP datagrams
IPPROTO_TCP	SOCK_STREAM	Packets wrap TCP segments
IPPROTO_IP / 0	Any	Use the default protocol for the given type

2.5. Destruir sockets

- Si tenemos una conexión persistente (TCP)

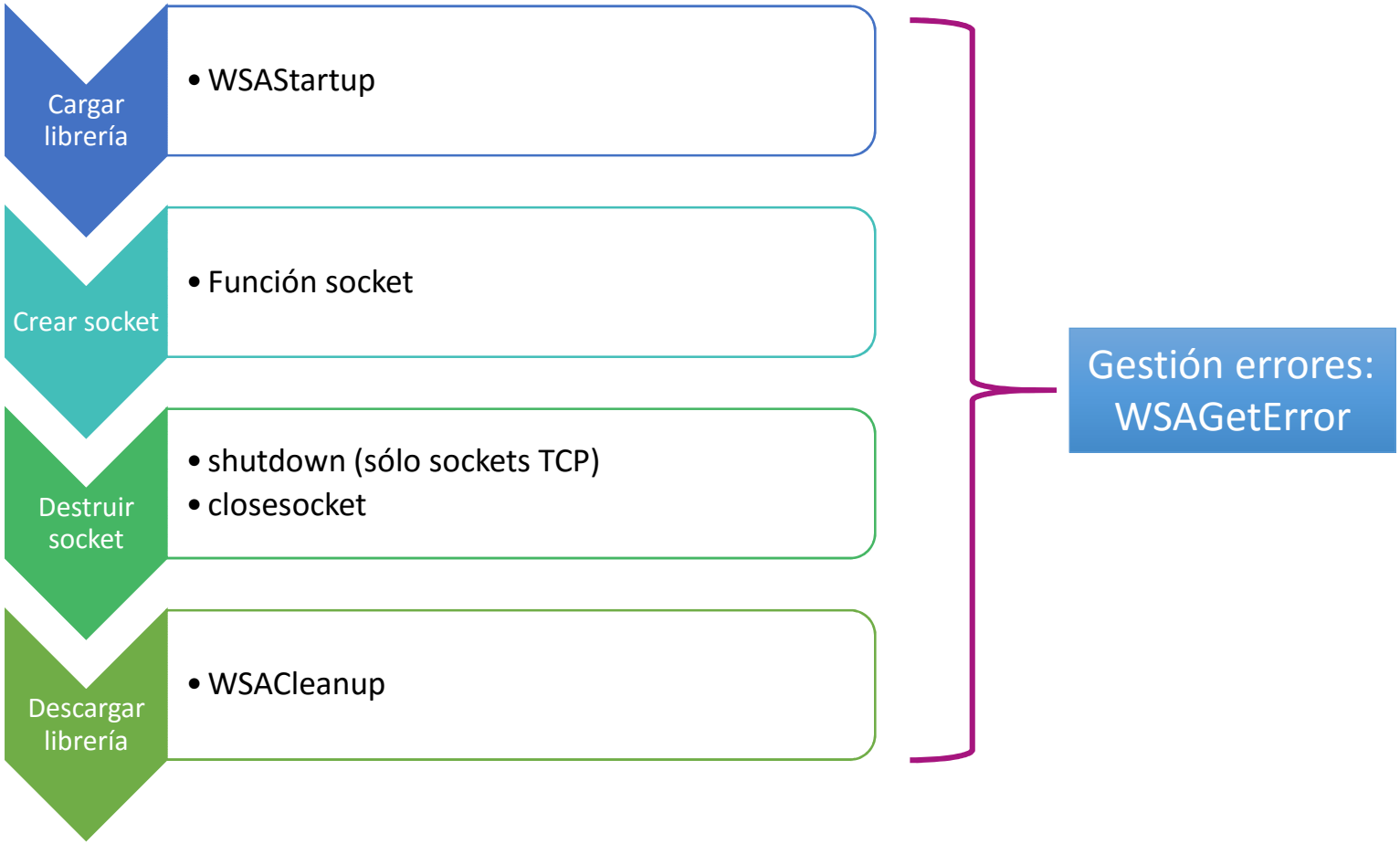
```
int shutdown(SOCKET sock, int how);
```

- how
  - SD_SEND: Para parar de enviar.
  - SD_RECEIVE: Para parar de recibir.
  - SD_BOTH: Para parar de enviar y recibir.
  - Cuando dejo de enviar se envía un paquete de FIN, de forma que en el otro lado saben que he dejado de transmitir y puede también cerrar su socket tranquilamente.

- Para cerrar el socket definitivamente:

```
int closesocket(SOCKET sock);
```

Hasta el momento



Y… EMPEZAMOS CON PROGRAMACIÓN

Vamos a empezar a crear nuestro wrapper para socket

- ¿Recordamos cómo se trabaja con bitbucket y sourcetree?

- Para crear el nuevo repositorio → SourceTree → Clone/New → Create New Repository
- Nos pedirá seleccionar una carpeta. Mejor seleccionemos una carpeta que esté vacía y sobre la que empezaremos a trabajar desde cero.
- Añadamos el .gitignore en la raíz de la carpeta para que no se nos añadan ficheros innecesarios. Este es el contenido que tengo puesto en mi .gitignore:

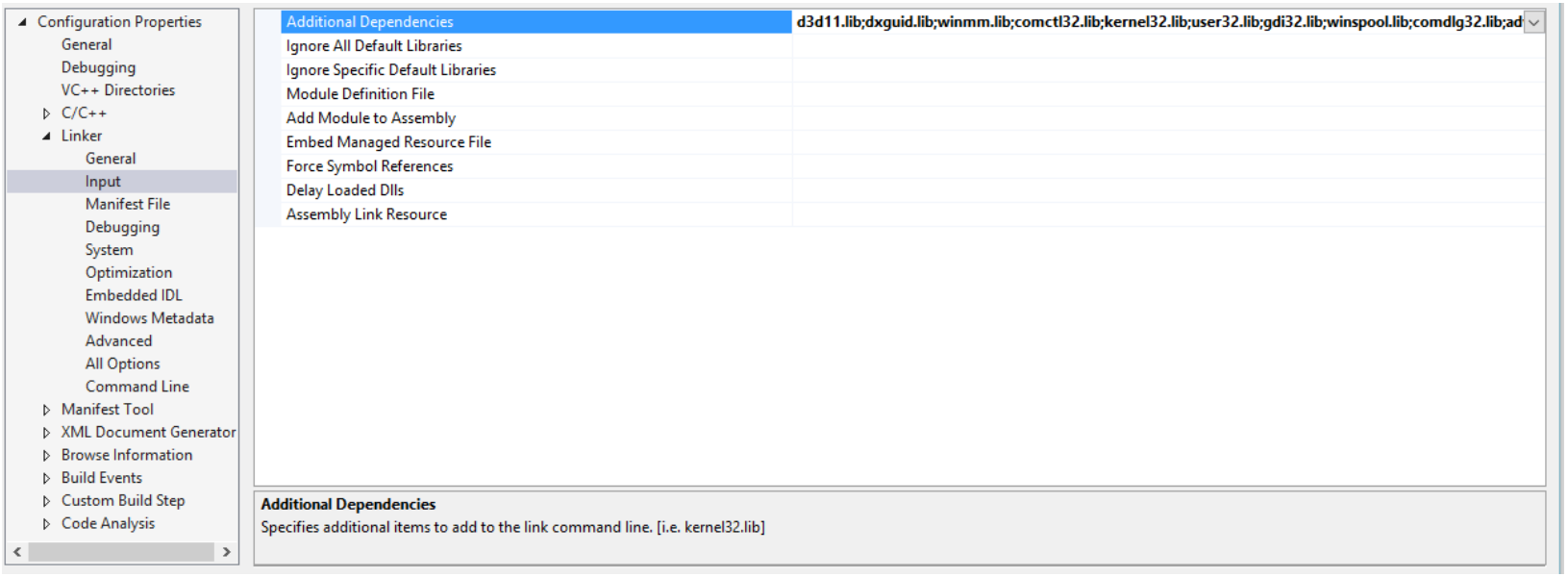
```
1  *.opendb
2  *.sdf
3  *.suo
4  *Debug*
5  *Release*
```

Más información sobre cómo trabajar con Bitbucket:

- Getting started with BitBucket Cloud -- https://goo.gl/1r25k7
- BitBucket Cloud Teams -- https://goo.gl/CzcFEJ

Vamos a empezar a crear nuestro wrapper para socket

- Recordad que es una buena práctica tener todos los pequeños ejemplos que vayamos haciendo en clase en una sola solución y organizada por proyectos.
  - El objetivo es que no perdáis ejemplos por tenerlos en n-mil soluciones diferentes.
- Una vez creado el proyecto, tenemos que configurarlo para que compile sockets:
  - En **Propiedades del proyecto → Linker → Input → Additional Dependencies**, añadir estas librerías: **d3d11.lib; dxguid.lib; winmm.lib; comctl32.lib; Ws2_32.lib**



EJERCICIO 02_01: Empezar a crear nuestro GenericSocket

GenericSocket

- Atributos
  - SOCKET
- GenericSocket(int type)
  - Inicializa el socket como un socket del tipo type.
  - Si se produce un error, llama a MostrarError.
- ~GenericSocket()
  - Cierra el socket
  - Si se produce un error, llama a MostrarError.

SocketTools

- Será más cómodo que sean funciones static
- CargarLibreria
  - Si no se carga correctamente, se llama a MostrarError
- DescargarLibreria
  - Si no se descarga correctamente, se llama a MostrarError
- MostrarError(string mensaje)
  - Muestra el mensaje del error generado y el código de error.

Crea un mini-Main en el que se llame al constructor y destructor de GenericSocket.
Procura controlar todos los posibles errores.

2.6. SocketAddress

- Cada paquete que enviemos necesitará una dirección origen y una dirección destino y un puerto origen y destino.
- Para pasar esta información la API nos da el tipo **sockaddr**:

```
struct sockaddr
{
      uint16_t sa_family;
      char sa_data[14];
};
```

- **sa_family**
  - Tipo de dirección. AF_INET.
- **sa_data**
  - Contiene la dirección actual.
  - Se da en un char[14] porque así permite contener cualquier tipo de dirección de cualquiera de las familias.

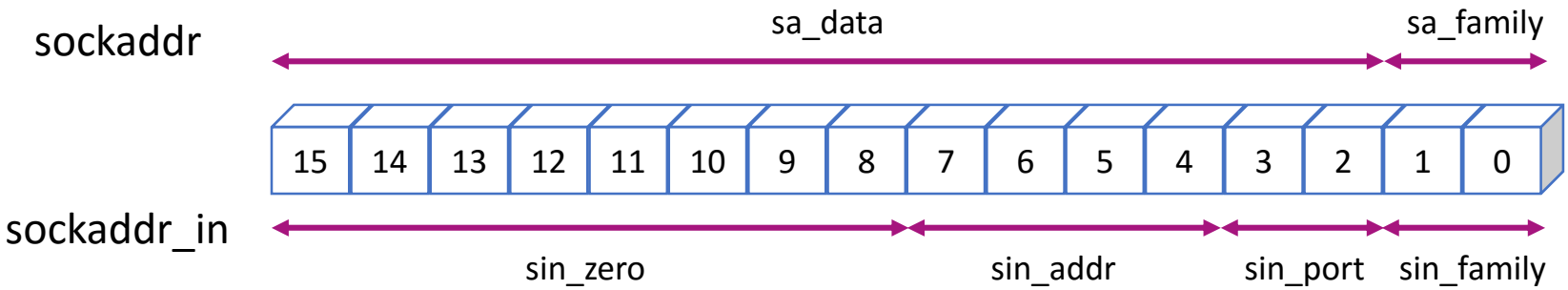
2.6. SocketAddress

- Para crear específicamente la dirección de un paquete IPv4, se usa el tipo **sockaddr_in**.

```
struct sockaddr_in
{
      short sin_family;
      uint16_t sin_port;
      struct in_addr sin_addr;
      char sin_zero[8];
};
```

- **sin_family**
  - Hace match con el campo sa_family de sockaddr. (2 bytes)
- **sin_port**
  - 16 bits para almacenar el puerto de la dirección. (2 bytes)
- **sin_addr**
  - 4 bytes para almacenar la dirección IP.
  - Para mantener la compatibilidad entre plataformas, la API da un **struct unión** que permite almacenar la dirección en varios formatos.
- **sin_zero**
  - No se usa. Sólo es un padding para que este tipo pueda hacer match con sockaddr.

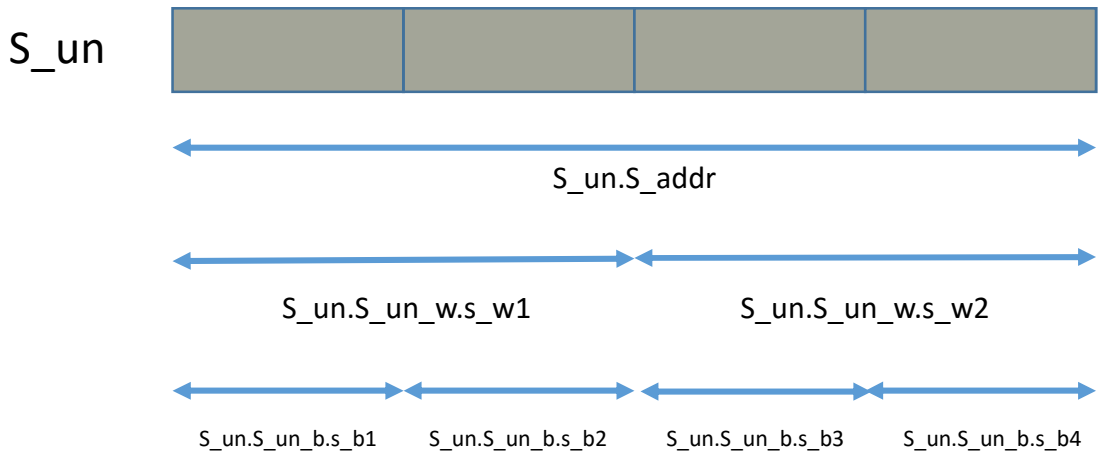
2.6. SocketAddress: sockaddr vs. sockaddr_in



- sockaddr es una estructura genérica para cualquier tipo de dirección.
- sockaddr_in es específica para direcciones IPv4.
- Están preparados para poder hacer un cast sin problemas. A sockaddr_in se le añade un padding de ceros para que sea así.

2.6. SocketAddress

```
struct in_addr {
      union {
            struct {
                  uint8_t s_b1, s_b2, s_b3, s_b4;
            } S_un_b;
            struct {
                  uint16_t s_w1, s_w2;
            } S_un_w;
            uint32_t S_addr;
      } S_un;
};
```



2.6. SocketAddress

- La estructura anterior es una forma muy rápida de hacer conversiones.
- En este caso, podemos pasar una dirección IP dada en 4 bytes a un entero de 32 sin más esfuerzo de consultar otro campo del union.

EJERCICIO:
Comprueba qué vale la dirección 127.0.0.1 en formato de un int32 y en formato de 2 words si la inicializas en formato de 4 bytes, utilizando la estructura in_addr.

2.6. SocketAddress

- Para poder pasar un nombre de dominio a IP, tenemos otras funciones y estructuras:

```
int getaddrinfo(const char* hostname, const char* servname, const addrinfo* hints, addrinfo** res);
```

- **hostname**
  - Nombre del dominio. Ejemplo: live-998.herokuapp.com
- **servname**
  - Número de puerto o nombre del servicio. Ejemplo: “80” o “http”
- **hints**
  - Permite definir un filtro sobre los resultados que queremos que se retornen.
  - Pasamos nullptr si queremos que se retornen todos.
- **res**
  - Puntero a una lista de direcciones posibles.
- Esta función crea un paquete de protocolo DNS y la envía vía UDP y TCP a los servidores DNS que tenemos configurados para averiguar la IP asociada.
- Esto puede hacer que tarde un poco en responder.
- Es una función que bloquea la ejecución. Habría que plantearse ponerla en un thread a parte.

2.6. SocketAddress

```
struct addrinfo {
      int ai_flags;
      int ai_family;
      int ai_socktype;
      int ai_protocol;
      size_t ai_addrlen;
      char* ai_canonname;
      sockaddr* ai_addr;
      addrinfo* *ai_next;
}
```

- **ai_flags, ai_socktype, ai_protocol**
  - Utilizamos estos campos como filtro. En nuestro caso de usar addrinfo como resultado, podemos ignorarlos.
- **ai_family**
  - Indica a que familia pertenece la dirección: AF_INET, AF_INET6.
- **ai_addrlen**
  - Tamaño ai_addr.
- **ai_canonname**
  - Contiene el nombre canónico del hostname. Sólo si hemos puesto AI_CANONNAME en el campo ai_flags.
- **ai_addr**
  - Contiene la dirección del tipo ai_family a la que apunta el hostname y el puerto especificado.
- **ai_next**
  - Puntero a una lista de addrinfo. Un dominio puede tener más de una IP asignada y de diferentes familias. Podemos iterar sobre esta lista para ver qué dirección se adapta a lo que queremos.
  - Si en ai_family he especificado un tipo de dirección concreto, sólo me buscarán direcciones de este tipo.
  - En la última casilla ai_next tiene un nullptr, para indicar que es el final.

EJERCICIO 02_02: Creamos SocketAddress

SocketAddress

- Atributos: sockaddr_in
- Constructores:
  - Por defecto
  - Por parámetros 4 bytes de la ip + puerto
  - Constructor de copia
- Sobrescribir el cout.
- Destructor

Incorpora también la función SetAddress, de la que ya tienes el código.
Trata de entender lo que hace la función.

2.7. Vincular un socket: BIND

¿Qué hace BIND?

Vincula al socket con un puerto concreto

Si el socket trata de enviar:

- Se indica este puerto como puerto origen del paquete.

Si el socket trata de recibir:

- Leeré de el puerto vinculado y los demás sockets me enviarán los paquetes a este puerto.

¿Qué pasa si no está hecho el BIND?

Si el socket trata de enviar:

- Se vincula automáticamente al primer puerto libre.

Si el socket trata de recibir:

- Es imposible, los demás sockets no saben a qué puerto enviarme el paquete.
- No estoy escuchando por ningún puerto concreto.

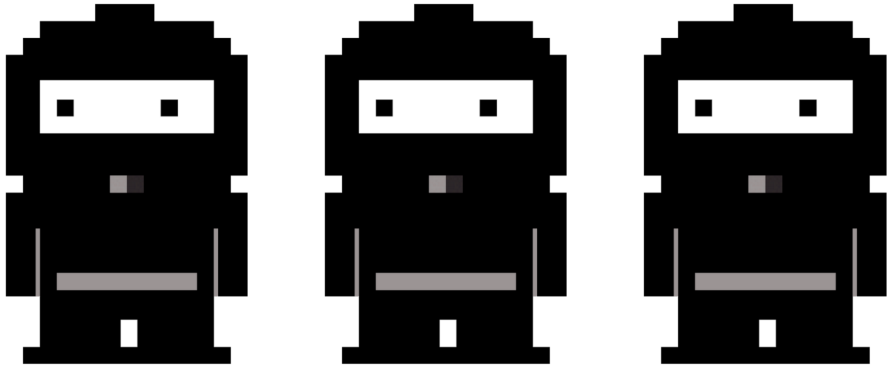
2.7. Vincular un socket: BIND

```
int bind(SOCKET sock, const sockaddr* address, int address_len);
```

- **sock**
  - Socket que vamos a vincular.
- **address**
  - Dirección de mi máquina a la que se vincula el socket.
  - Recordemos que un host puede tener varias IPs.
    - Aunque esto suele ocurrir en máquinas que hacer de routers o bridges entre redes.
  - Para juegos en red podemos decir que se escuche a un puerto determinado a través de todas las interfaces de red que haya disponibles.
  - Esto lo indicamos con la macro INADDR_ANY del campo sin_addr del struct sockaddr_in
- **address_len**
  - Tamaño del sockaddr que paso como parámetro.
- Retorna 0 si todo ha ido bien. -1, si ha habido error.
  - Recordemos que para saber más sobre el error debemos usar **WSAGetLastError()**

EJERCICIO: Incorpora **int Bind(SocketAddress& address)** a GenericSocket.

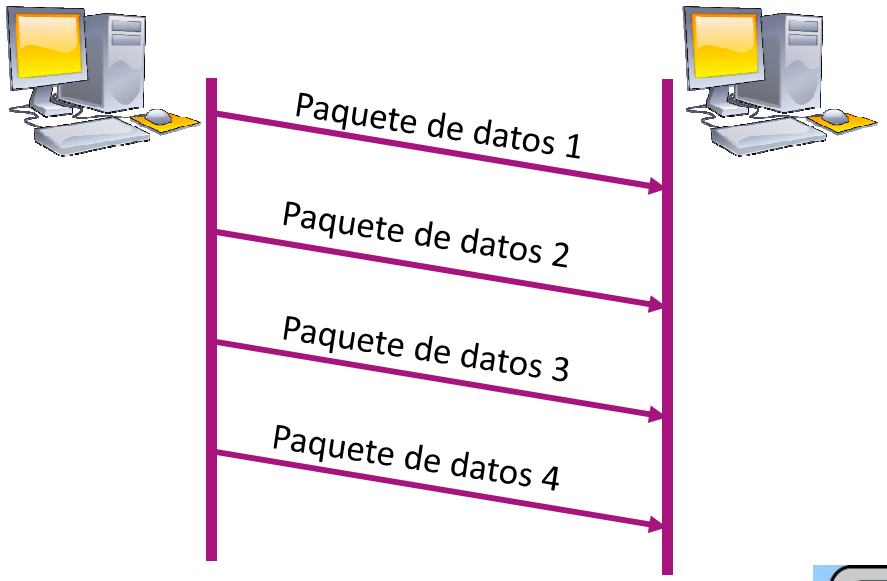
Y AHORA… SOCKETS UDP.
Y PODREMOS EMPEZAR A HACER ALGO!!!!

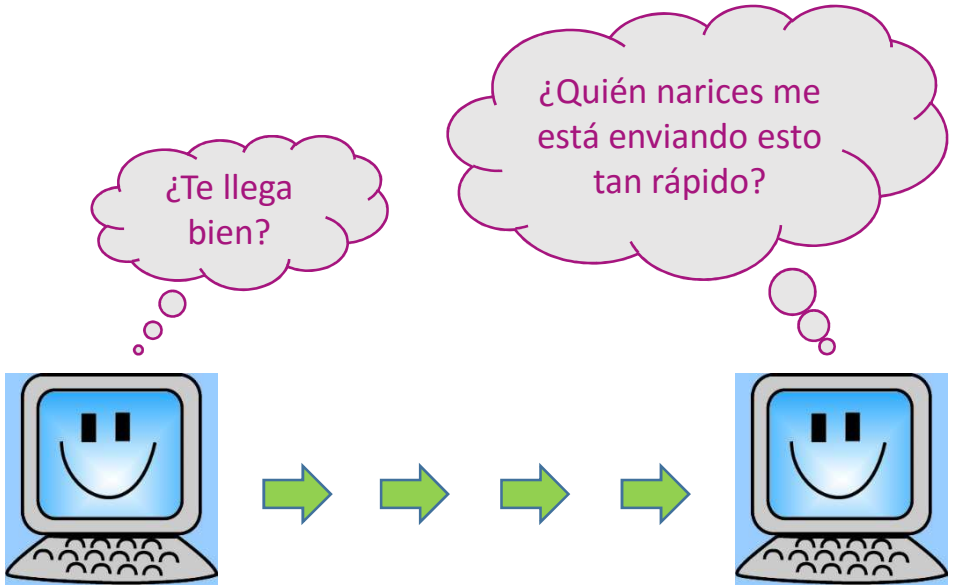


3.1. Sockets UDP: Características

Recordamos

- UDP es un protocolo de transporte ligero.
- Tiene sólo 8 bytes de cabecera.
- Pero es incapaz de asegurar que el paquete llega a destino.





3.1. Sockets UDP: Características

Importante

- En un host, el mismo socket puede ser utilizado para enviar y para recibir.

Cuando envío

Puedo saltarme el paso 2 si supongo un BIND automático

1. Creo el socket.
2. BIND
3. Envío a destino

Cuando recibo

1. Creo el socket.
2. BIND → **OBLIGATORIO: Sino no saben a dónde enviarme los paquetes**
3. Recibo de cualquier destinatario que conozca mi @IP y puerto

3.2. Sockets UDP: Enviar

```
int sendto(SOCKET sock, const char* buf, int len, int flags, const sockaddr* to,
int tolen);
```

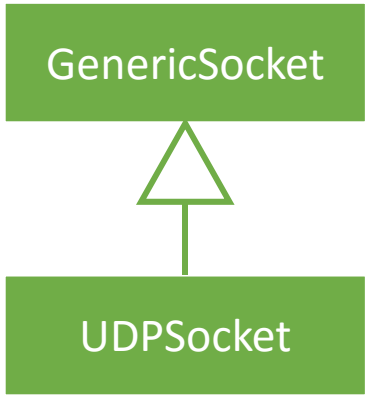
- **sock**
  - Socket a través del que enviamos.
- **buf**
  - Puntero a la primera posición de los datos que vamos a enviar.
  - Puede ser un char* o cualquier tipo que pueda hacer cast a char.
  - void* sería el tipo más apropiado, pero la librería se definió así.
- **len**
  - Longitud de los datos que enviaremos.
  - El tamaño máximo de un datagrama UDP es de 65535 bytes (menos los bytes de cabecera)
  - Pero la MTU para Ethernet es de 1500 bytes. Si descontamos otros tipos de cabecera, nos podemos quedar en 1300 bytes.
  - Para evitar la fragmentación, es aconsejable que evitemos enviar datagramas con más de 1300 bytes de datos.
- **flags**
  - Lo podemos dejar a 0.
- **to**
  - Dirección destino.
  - Debe ser de la misma familia que la origen.
- **tolen**
  - Tamaño de la dirección usada en **to:** sizeof(sockaddr_in)
- Retorna 0 si todo ha ido bien. -1, si ha habido error.
  - Recordemos que para saber más sobre el error debemos usar **WSAGetLastError()**

3.3. Sockets UDP: Recibir

```
int recvfrom(SOCKET sock, const char* buf, int len, int flags, const
sockaddr* from, int* fromlen);
```

- **sock**
  - Socket que recibe los datos.
- **buf**
  - Puntero a la primera posición de memoria en la que se copiarán los datos recibidos.
- **len**
  - Tamaño máximo de datos que podemos leer.
  - Lo usamos para evitar errores de overflow.
  - Los datos que no quepan en este **len**, se perderán.
  - Por eso, nos hemos de asegurar de que **len** sea lo suficientemente grande.
- **flags**
  - Lo podemos dejar a 0.
- **from**
  - Dirección que nos envía
  - No debemos inicializarla. La función nos copia los datos de la dirección que nos envía.
- **fromlen**
  - Tamaño de la dirección. Hemos de indicar sizeof(sockaddr).
- Retorna -1, si ha habido error. Sino, retorna el número de bytes recibidos.
  - Recordemos que para saber más sobre el error debemos usar **WSAGetLastError()**

EJERCICIO 02_03: Creamos UDPSocket



- Crea la clase UDPSocket como clase derivada de GenericSocket.
- Incorpora las funciones:
  - int SendTo(const void* data, int lenData, const SocketAddress& to);
  - int ReceiveFrom(void* data, int lenData, SocketAddress& from);

int SendTo(const void* data, int lenData, SocketAddress& to);

- **data**: Datos a enviar
- **lenData**: Longitud de estos datos
- **to**: Destino a donde se envían estos datos
- -1 si hay error. 0, si todo va bien.

int ReceiveFrom(void* data, int lenData, SocketAddress& from);

- **data**: Guarda los datos que se han recibido
- **lenData**: Cantidad de datos de los que se reciben que estamos dispuestos a almacenar
- **from**: Guarda la dirección del socket que ha enviado los datos
- -1 si hay error. 0, si todo va bien.

Ejercicio 02_04

- Crear un programa de consola que pueda funcionar de estas dos formas, en función de lo que le pasemos como argumento:

ejercicio02_04.exe servidor localhost:5000

- El programa hace de servidor
- Escucha por la IP localhost y el puerto 5000
- Queda a la espera de lo que le envíe el cliente
- Cuando recibe algo lo muestra por pantalla (consola)
- El programa finaliza cuando recibe la palabra “exit”

ejercicio02_04.exe cliente localhost:5000

- El programa hace de cliente
- Envía a la dirección localhost y al puerto 5000
- El programa pregunta por línea de comandos lo que le queremos enviar al servidor
- Al picar enter, se envía el mensaje a destino (debe aceptar mensajes con espacios en medio)
- Cuando escribamos “exit”, enviaremos el mensaje a servidor y finalizaremos el programa.

Ejercicio 02_05

Sobre el ejercicio anterior, hacer las modificaciones que consideres necesarias para que:

- Más de un cliente pueda enviar mensajes al servidor.
- El servidor debe poder recibir mensajes de varios clientes.
- Cuando escribimos exit desde un cliente, enviamos exit al servidor y cerramos el cliente.
- Pero el servidor no se desconecta hasta que no hayamos cerrado todos los clientes.