

# Videojuegos multi-jugador con **SDLNet**

TPV2  
Samir Genaim

# Videojuegos multi-jugador

- ◆ Juego que permita la interacción de dos o más jugadores al mismo tiempo
  - de manera física en una misma consola/PC
  - en 2 o más consolas/PCs mediante conexión red
  - mediante servicios en línea
- ◆ En tiempo real o por turnos
- ◆ En este tema vamos a ver cómo desarrollar un videojuego multi-jugador en red

# Videojuegos multi-jugador en red

- ◆ Para desarrollar un videojuego de este tipo, primero necesitamos una forma de intercambiar información entre los jugadores (es decir entre sus programas)
- ◆ Vamos a usar SOCKETS, un mecanismo que permite la comunicación entre programas (en el mismo PC o en PCs distintos)
- ◆ En particular vamos a usar SDL\_Net: API para sockets que funciona igual en distintos sistemas operativos (Windows, Unix, OSX, etc.)
- ◆ En cada sistema operativo SDL\_Net usa una librería correspondiente para sockets, pero nosotros no tenemos que saberlo ...

# Sockets

- ◆ Los sockets son un mecanismo de comunicación entre procesos.
- ◆ Comunicación bi-direccional entre procesos en una misma máquina o entre procesos lanzados en diferentes máquinas.
- ◆ Hay varios tipos de sockets:
  - Stream sockets: usan el protocolo Transmission Control Protocol (TCP)
  - Datagram sockets: usan el protocolo User Datagram Protocol (UDP)
  - ...

# Sockets y Redes

Los sockets se pueden ver como una interfaz entre la aplicación y la capa de transporte

Protocolos de Transporte

(Logical) Network

Physical

APPLICATION

SOCKETS

TCP

UDP

IP (y otros)

Network

# Direcciones IP y HostName

- ◆ Cada ordenador (o cualquier aparato en general) en la red de Internet tiene una dirección IP, p.ej., **147.81.1.92** – es un identificador único del ordenador en la red.
- ◆ HostName: nombres de dominio tienen la forma **www.ucm.es**, **www.google.com** – están asociados a direcciones IP.
- ◆ Hay servidores que se llaman **DNS Servers** que pueden traducir un HostName a IP – es mas fácil usar nombres (el IP puede cambiar pero el nombre es fijo).
- ◆ Todos conocen las direcciones IP de los **DNS Servers** para poder comunicar con ellos (p.ej., los de google son **8.8.8.8**, **8.8.4.4**). No vamos a trabajar directamente con **DNS Servers**, son parte de la configuración de la red.
- ◆ Cada ordenador tiene una red local para su propio, con dirección IP **127.0.0.1** y hostname **localhost**. Es para el uso local entre programas el mismo ordenador, no se puede conectar a otro ordenador usando esta dirección.

# Ports (Puertos)

- ◆ IPs dan identificadores a ordenadores, pero como en el ordenador hay muchos programas ejecutando queremos permitir comunicar con un programa específico.
- ◆ Para eso existe el concepto de puertos (un número entre 0 y 65535 - 2 bytes). Sólo un programa puede usar un puerto a la vez, así que la combinación de IP y puerto sirve como identificador único para un programa/servicio.
- ◆ Normalmente los puertos desde 0 hasta 1023 están reservados para el sistema operativo o servidores comunes, p.ej., el puerto 22 normalmente está reservado para el “remote shell”, el 80 para el “http server”, etc.
- ◆ Para clientes normalmente el puerto se elige de manera automática al crear el socket, para los servidores es importante especificarlo porque los clientes necesitan saberlo para conectar.

# TCP vs UDP (muy breve de momento)

## ◆ TCP

1. Se puede enviar/recibir **secuencias de bytes** de (al nivel del programa). La capa de transporte los divide en paquetes y los envía al destino. Para intercambiar información hay que establecer una conexión antes.
2. Está garantizado que todos los bytes que enviamos llegaran al destino, y en el mismo orden.

## ◆ UDP

1. Se puede enviar/recibir **datagrams** (paquetes de tamaño fijo). No hace falta establecer conexión, simplemente enviamos a una dirección (IP más puerto)
2. No está garantizado que todos los **datagrams** lleguen a su destino, ni el orden de llegada (pero habitualmente llegan, y en el mismo orden)

◆ TCP es fiable, pero UDP es más rápido por eso se usa en aplicaciones que necesitan comunicación rápida.

# Inicilizer y Finalizar SDL\_Net

Antes de empezar a usar SDINet siempre  
hay que llamar a `SDLNet_Init()`

```
if (SDLNet_Init() < 0) {  
    error();  
}
```

```
SDLNet_Quit();
```

Antes de salir del programa  
llamar a `SDLNet_Quit()`

**UDP con SDL\_Net**

# Direcciones en SDL\_Net

IPAddress es un struct de `SDL_Net` que tiene 2 números: uno de tipo `Uint32` para representar un IP y otro de tipo `Unit16` para representar el puerto

```
IPaddress ip;
```

```
if (SDLNet_ResolveHost(&ip, host, port) > 0) {  
    error();  
}
```

'host' es un `char*` que corresponde a IP o hostname, p.ej., "147.81.1.92" o "www.ucm.es". `SDLNet_ResolveHost` rellena la información correspondiente a `host:port` en 'ip'. Usa los servidores DNS para convertir el hostname 'host' a IP. Hay un caso particular en el que 'host' es `nullptr`, lo usaremos con TCP más adelante.

# Crear un Socket

```
UDPSocket sock = SDLNet_UDP_Open(port);  
if (!sock) {  
    error();  
}
```



Crea un socket para recibir/enviar información en el puerto 'port'. Si port=0 elige un puerto libre automáticamente (lo usamos normalmente en el lado del cliente, en el lado del servidor necesitamos decir a qué puerto queremos escuchar porque se supone que los clientes saben esa información).

Devuelve `nullptr` si ocurre algún error (como el puerto ya está en uso). `UDPSocket` es un puntero de tipo `_UDPSocket*`

```
typedef struct _UDPSocket *UDPSocket;
```

# UDP Packet

```
UDPpacket *p = SDLNet_AllocPacket(MAX_SIZE); // hasta 512  
if (!p) { error(); }
```

```
typedef struct {  
    int channel; // The src/dst channel of the packet  
    Uint8 *data; // Pointer to buffer of size MAX_SIZE  
    int len; // The length of data in the buffer data  
    int maxlen; // The size of the data buffer (MAX_SIZE)  
    int status; // packet status after sending  
    IPaddress address; // The src/dest address of an incoming/outgoing packet  
} UDPpacket;
```

Es la estructura de datos que usamos para recibir y enviar **datagrams** (el contenido está en le buffer **data**). Es suficiente tener 1, porque al recibir copiamos la información a otra parte

# Enviar datos

```
IPaddress ip;  
if (SDLNetResolveHost(&ip, host, port) < 0) {  
    error();  
}  
...  
p->address = ip; // fill in data in p->data  
...  
p->len = ...;  
SDLNet_UDP_Send(sock, -1, p);
```

Rellenar la dirección del destinatario en 'ip'

Copiar la dirección del destinatario a p->address

Rellenar los bytes que quieras enviar en p->data

El número de bytes que hemos puesto en p->data

Envía el paquete al destinatario. El -1 significa no usar canales. Más adelante veremos qué son. Devuelvo el numero de envíos o 0 si hay algún error. En este caso el numero de envíos es 1, pero con canales puede ser mas

# Recibir datos

```
SDLNet_UDP_Recv(sock, p);
```



Recibe un paquete en p (este paquete alguien lo envió al puerto asociado a sock). Hay que crear p antes.

La llamada **No bloquea** si no hay paquetes. Devuelvo 1 si ha recibido un paquete, 0 si no hay nada, -1 si hay algún error.

p->**address** es la dirección del remitente

p->**data** el buffer con el contenido (los bytes que han enviado)

p->**len** la longitud del contenido

p->**channel** si no usamos canales sería -1

# Cerrar un Socket

```
SDLNet_UDP_Close(sock);
```



Cuando acabamos de usar un socket hay que cerrarlo. En el caso de `SDL_Net`, aparte de cerrar el socket también libera la memoria dinámica que usa, etc.

# Comunicación (No) Bloqueante

A veces queremos saber si hay actividad en los sockets antes de intentar a recibirla (por ejemplo en el caso de TCP la llamada a `SDLNet_TCP_Recv` bloquea hasta que haya información, algo que no podemos permitir en algunos contextos).

Por otro lado, hay programas que quieren bloquear si no hay actividad en los sockets hasta que haya algo (en lugar de consultar continuamente si hay actividad o no)

En los dos casos, podemos usar un mecanismo que se llama **SocketSet**, que nos permite preguntar si hay una actividad en un conjunto de sockets antes de intentar a recibir la información, y también permite bloquear el programa (si que consume CPU) hasta que haya actividad ..

# Como Crear un SocketSets

```
SDLNet_SocketSet set = SDLNet_AllocSocketSet(10);
```

```
SDLNet_UDP_AddSocket(set, s1);
```

```
SDLNet_UDP_AddSocket(set, s2);
```

...

Crear SocketSet

Añadir Sockets al SocketSet

- ◆ Creamos un conjunto de (hasta 10) sockets y añadimos los sockets correspondientes.
- ◆ Al final hay que borrar el conjunto de sockets usando `SDLNet_FreeSocketSet(set)`.
- ◆ Se puede quitar un socket del conjunto usando el método `SDLNet_UDP_DelSocket(set, s2)`
- ◆ Se puede añadir/quitar en cualquier momento.

# Como usar un SocketSets

`SDLNet_CheckSockets(set, N)` devuelve el número de sockets que tienen actividad en el conjunto 'set'. Si no hay actividad bloquea hasta que haya o hasta que pasen `N` milisegundos (en el ultimo caso devuelve 0). Si usamos `N=0` y no hay actividad la llamada no bloquea, simplemente devuelve 0.

```
if (SDLNet_CheckSockets(set, N) > 0) {  
    if (SDLNet_SocketReady(s1)) { non-blocking ... }  
    if (SDLNet_SocketReady(s2)) { non-blocking ... }  
    ...  
}
```

Si hay actividad en el conjunto, preguntamos en qué socket hay actividad porque el 'set' puede tener varios sockets. Siempre hay que llamar a `SDLNet_CheckSockets` antes de llamar `SDLNet_SocketReady`.

# Channels (Bind y Unbind)

Es un mecanismo de SDL\_Net (sólo para UDP) que nos permite vincular direcciones a canales para facilitar en envío de mensajes en algunas aplicaciones ...

Vincular la dirección **addr** (de tipo **IPAdress**) al canal **n** del socket **sock** (se pueden vincular varias direcciones a un canal y una dirección a varios canales)

**SDLNet\_UDP\_Bind(sock, n, &addr)**

**SDLNet\_UDP\_Unbind(sock, n)**

**SDLNet\_UDP\_GetPeerAddress(sock, n)**

Desvincular todas las direcciones vinculadas al canal **n** de **sock** (sólo de ese canal)

Devuelve la 1<sup>a</sup> dirección vinculada al canal **n** de **sock** (como **IPAdress\***)

# Channels (Send y Receive)

Si **n** no es -1, envía **p** a todas las direcciones vinculadas con el canal **n** del **sock**. En este caso ignore **p->address**

**SDLNet\_UDP\_Send(sock,n,p)**

**SDLNet\_UDP\_Rec(sock,p)**

**p->channel** se actualiza con el numero de canal al que la dirección del remitente está vinculado – el máximo si hay varios, y -1 si no está vinculado a ningún canal

# Enviar/Recibir varios packets

Crea un array de n (punteros a) packets, cada packet es de tamaño máximo MAX\_SIZE. La posición n+1 del array es **nullptr** para poder identificar su final en send/receive.

```
UDPpacket **p = SDLNet_AllocPacketV(n, MAX_SIZE)  
SDLNet_FreePacketV(p) // para libera el array
```

SDLNet\_UDP\_SendV(sock,p,n)

SDLNet\_UDP\_RecvV(sock,p)

Envia **n** packets del array p. Envia p[i] al canal especificado en p[i]->channel y si es -1 lo envia a p[i]->address

Recibir varios packets a la vez en p. Devuelve el número de packets recibidos o -1 si hay error. El valor de p[i]->channel indica el canal al que la dirección del remitente está vinculada (como en el caso del **SDLNet\_UDP\_Recv**).

# Client/Server TCP: ejemplo I

## Server

1. Espera un mensaje de clientes (hasta 255 chars)
2. Escribe el mensaje del cliente
3. Envía el mensaje “Received!” al cliente
4. goto 1

## Client

1. Pide un mensaje al usuario (hasta 255 chars)
2. Si es “exit” cierra la conexión y sale, si no envía el mensaje al servidor
3. Espera la respuesta del servidor para 3seg
4. goto 2

# Server: inicialización/bucle principal

```
void server(int port) {  
    UDPsocket sd = SDLNet_UDP_Open(port);  
    if (!sd) { error(); }  
  
    UDPpacket *p = SDLNet_AllocPacket(MAX_PACKET_SIZE);  
    if (!p) { error(); }  
  
    char *buffer = reinterpret_cast<char*>(p->data);  
  
    SDLNet_SocketSet socketSet = SDLNet_AllocSocketSet(1);  
    SDLNet_UDP_AddSocket(socketSet, sd);  
  
    while (...) {  
        // TODO I: PROCESS DATA on socket sd  
    }  
  
    SDLNet_FreePacket(p);  
    SDLNet_FreeSocketSet(socketSet);  
    SDLNet_UDP_Close(sd);  
}
```

# Server: actividad en clientes

```
if (SDLNet_CheckSockets(socketSet, SDL_MAX_UINT32) > 0) {  
    if (SDLNet_SocketReady(sd)) {  
        if (SDLNet_UDP_Recv(sd, p) > 0) {  
  
            // print client's message  
            cout << "Client says: " << buffer << endl;  
  
            // send a response  
            memcpy(buffer, "Received!", 10);  
            p->len = 10;  
            SDLNet_UDP_Send(sd, -1, p);  
        }  
    }  
}
```



p ya tiene la dirección de quien ha enviado el mensaje  
(se rellena en la llamada a `SDLNet_UDP_Recv`).

# Client: conectar

```
void client(char* host, int port) {
    UDPsocket sd = SDLNet_UDP_Open(0);
    IPEndPoint srvadd;
    if (SDLNet_ResolveHost(&srvadd, host, port) < 0) {error(); }
    UDPpacket *p = SDLNet_AllocPacket(MAX_PACKET_SIZE);
    char *buffer = reinterpret_cast<char*>(p->data);
    SDLNet_SocketSet socketSet = SDLNet_AllocSocketSet(1);
    SDLNet_UDP_AddSocket(socketSet, sd);

    while (...) {
        // TODO I: PROCESS DATA on socket sd
    }

    SDLNet_FreePacket(p);
    SDLNet_FreeSocketSet(socketSet);
    SDLNet_UDP_Close(sd);
}
```

# Client: intercambiar datos

```
cout << "Enter a message: ";
cin.getline(buffer, 255);
```

```
if (strcmp(buffer, "exit") == 0) { break; }
```

```
p->len = static_cast<int>(strlen(buffer)) + 1;
p->address = srvadd;
```

```
SDLNet_UDP_Send(sd, -1, p);
```

Espera 3seg o hasta recibir la respuesta del servidor

```
if (SDLNet_CheckSockets(socketSet, 3000) > 0) {
    if (SDLNet_SocketReady(sd)) {
        while (SDLNet_UDP_Recv(sd, p) > 0) {
            cout << "Server says: " << buffer << endl;
        }
    }
}
```

Usamos **while** porque si el mensaje tarda en llegar podemos recibir varios a la vez.

# Client/Server UDP: ¡Cuidado!

- ◆ Cuando el cliente recibe un mensaje, no comprueba que el remitente es de verdad del servidor!
- ◆ Dado que cualquiera puede enviar mensajes al cliente en UDP, sin establecer una conexión, esto puede ser un problema.
- ◆ Solución I: se puede usar canales para asegurarnos de que el remitente está vinculado a algún canal, y si el paquete que recibe tiene -1 en `p->channel` lo rechazamos (pero hay que identificar clientes en el primer mensaje de alguna manera).
- ◆ Solución II: incluir una cabecera en cada mensaje con un código y rechazar mensajes que no lo tienen.
- ◆ En nuestros ejemplos, ignoramos este problema pero en aplicaciones reales hay que tenerlo en cuenta ...

# Resumen Client/Server: UDP

- ◆ El servidor es un programa que ejecuta en una maquina con el nombre **bla.domain.com** (p.ej, con IP **145.33.44.66**)
- ◆ El servidor abre un Socket en un puerto, p.ej., **2000**, y lo usa para recibir **datagrams** en ese puerto de cualquiera y enviar paquetes a cualquiera.
- ◆ El cliente abre un socket (asignado a algún puerto local libre) y lo usa para enviar **datagrams** a cualquiera (p.ej., al servidor) y para recibir **datagrams** de cualquiera.
- ◆ No es necesario establecer un conexión con el servidor (es como enviar cartas por correos). La dirección servidor es normalmente conocida, la del cliente está incluida en el paquete que se envía al servidor así que el servidor puede usarla para contestar.
- ◆ UDP no garantiza que los **datagrams** llegan, no garantiza el orden de llegada, no comprueba errores (pero verifica el checksum), etc. Es más rápido que TCP pero menos fiable – se usa en aplicaciones que necesitan comunicación rápida.

**TCP con SDL\_Net**

# TCP y Master Socket

- ◆ En TCP hay que establecer una conexión antes de enviar datos, para esto existe el concepto de “**master socket**” que es un socket que se usa sólo para peticiones de conexión.
- ◆ El servidor abre un Socket (el **master socket**) en un puerto, p.ej., **2000**, para recibir (escuchar) peticiones de conexión.
- ◆ El cliente abre un Socket y pide conectar al servidor **145.33.44.66:2000**. El Socket también está automáticamente asociado a un puerto local para recibir mensajes, pero no necesitamos saber qué puerto es, todo es implícito.
- ◆ El servidor recibe la petición en su **master socket**, y cuando la acepta tendrá automáticamente otro socket (con otro puerto) que se usa para la comunicación con el cliente (el cliente sigue usando el mismo socket) — **el servidor usa el master socket sólo para escuchar peticiones de conexión**.
- ◆ La conexión se mantiene abierta entre el servidor y el cliente hasta que cierren los sockets. El servidor puede tener varias conexiones abiertas a la vez con clientes (y servirlos de forma forma secuencial o paralela)

# Server - Crear el Master Socket

```
IPaddress ip;  
if (SDLNet_ResolveHost(&ip, nullptr, port) < 0) {  
    error();  
}  
TCPsocket masterSocket = SDLNet_TCP_Open(&ip);  
  
if (!masterSocket) {  
    error();  
}
```

Rellena la información del host y port en el 'ip' (en este caso host es `nullptr`)

Es un socket que se puede usar sólo para aceptar peticiones de conexión. `SDLNet_TCP_Open` lo crea así porque el segundo parámetro de `SDLNet_ResolveHost` es `nullptr`.

# Client - Crear un Socket

'host' es un **char\*** que corresponde a IP o hostname, p.ej., "147.81.1.92" o "www.ucm.es". **SDLNet\_ResolveHost** rellena la información correspondiente a **host:port** en 'ip'. Usa los servidores DNS para convertir el hostname 'host' a IP. Recuerda que 'host' no es **nullptr**, si no sería para **masterSocket**

```
IPaddress ip;  
if (SDLNet_ResolveHost(&ip, host, port) < 0) {  
    error();  
}  
  
TCPsocket conn = SDLNet_TCP_Open(&ip);  
  
if (!conn) {  
    error();  
}
```

Es un socket conectado a la dirección en 'ip'. Se no ha conseguido establecer la conexión devuelve **nullptr**.

# Server - Aceptar una Conexión

```
TCPsocket client = SDLNet_TCP_Accept(masterSocket);
```

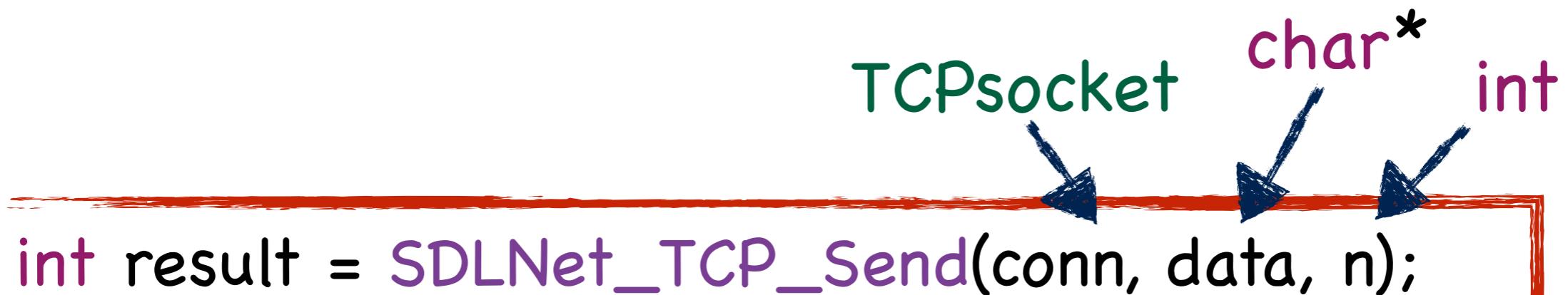


Si hay un cliente intentando a conectar, **SDLNet\_TCP\_Accept** devuelve un socket que usamos para comunicar con el cliente. Devuelve **nullptr** si no hay clientes que están intentando a conectar.

El masterSocket se puede usar para recibir más peticiones de conexión de otros clientes (para cada cliente hay que llamar a ese método).

En las librerías estándar de sockets, la ejecución del método que corresponde al método **SDLNet\_TCP\_Accept** **bloquea** hasta que haya una petición de conexión.

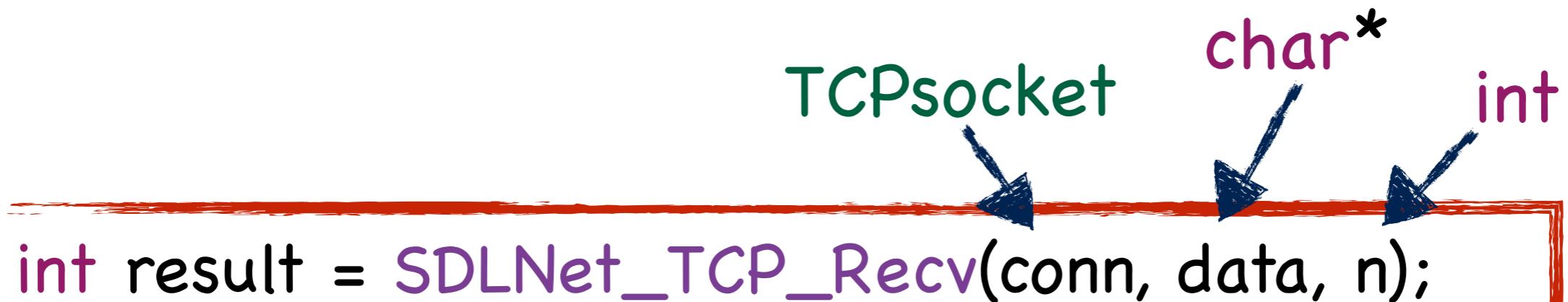
# Enviar datos (secuencias de bytes)



Envía en el socket 'conn' los primeros 'n' bytes a partir de la dirección 'data'.

`SDLNet_TCP_Send` devuelve el número de bytes que ha conseguido enviar. Un valor negativo indica que se ha ocurrido algún error.

# Recibir datos (secuencias de bytes)



Recibe desde el socket 'conn' hasta 'n' bytes y los copia a la dirección 'data'. La ejecución **bloquea** hasta que haya datos en el socket.

`SDLNet_TCP_Recv` devuelve el número de bytes que ha recibido. El valor 0 indica que el otro lado ha cerrado la conexión, un valor negativo indica que se ha ocurrido algún error.

# Cerrar un Socket

```
SDLNet_TCP_Close(socket);
```



Cuando acabamos de usar un socket hay que cerrarlo, tanto en la parte del cliente como en la parte del servidor. En el caso de SDLNet, aparte de cerrar el socket también libera la memoria dinámica que usa, etc.

# Comunicación (No) Bloqueante

Como en el caso de UDP, se puede tener más control sobre el comportamiento de bloqueo usando socket set. En el caso de TCP usamos `SDLNet_TCP_AddSocket` y `SDLNet_TCP_DelSocket`.

En el caso de TCP esto es muy importante porque las llamadas a `SDLNet_TCP_Recv` y `SDLNet_TCP_Accept` son bloqueares.

# Como Crear un SocketSets

```
SDLNet_SocketSet set = SDLNet_AllocSocketSet(10);
```

```
SDLNet_TCP_AddSocket(set, s1);
```

```
SDLNet_TCP_AddSocket(set, s2);
```

...

Crear SocketSet

Añadir Sockets al SocketSet

- ◆ Creamos un conjunto de (hasta 10) sockets y añadimos los sockets correspondientes.
- ◆ Al final hay que borrar el conjunto de sockets usando `SDLNet_FreeSocketSet(set)`.
- ◆ Se puede quitar un socket del conjunto usando el método `SDLNet_TCP_DelSocket(set, s2)`
- ◆ Se puede añadir/quitar en cualquier momento.

# Como usar un SocketSets

`SDLNet_CheckSockets(set, N)` devuelve el número de sockets que tienen actividad en el conjunto 'set'. Si no hay actividad bloquea hasta que haya o hasta que pasen `N` milisegundos (en el ultimo caso devuelve 0). Si usamos `N=0` y no hay actividad la llamada no bloquea, simplemente devuelve 0.

```
if (SDLNet_CheckSockets(set, N) > 0) {  
    if (SDLNet_SocketReady(s1)) { non-blocking ... }  
    if (SDLNet_SocketReady(s2)) { non-blocking ... }  
    ...  
}
```

Si hay actividad en el conjunto, preguntamos en qué socket hay actividad porque el 'set' puede tener varios sockets. Siempre hay que llamar a `SDLNet_CheckSockets` antes de llamar `SDLNet_SocketReady`.

# Client/Server TCP: ejemplo I

## Server

1. Espera una petición de conexión de un cliente
2. Espera un mensaje del cliente (hasta 255 chars)
3. Envía el mensaje “Received!” al cliente
4. Cierra la conexión con el cliente
5. goto 1

## Client

1. Conecta al servidor
2. Pide un mensaje al usuario (hasta 255 chars)
3. Envía el mensaje al servidor
4. Espera la respuesta del servidor
5. Cierra la conexión y sale del programa

# Server: inicialización/bucle principal

```
void server(int port) {  
    IPaddress ip;  
    if (SDLNet_ResolveHost(&ip, nullptr, port) < 0) { error(); }  
    TCPsocket masterSocket = SDLNet_TCP_Open(&ip);  
    if (!masterSocket) { error(); }  
  
    SDLNet_SocketSet socketSet = SDLNet_AllocSocketSet(1);  
    SDLNet_TCP_AddSocket(socketSet, masterSocket);  
  
    while (true) {  
        if (SDLNet_CheckSockets(socketSet, SDL_MAX_UINT32) > 0) {  
            // TODO I: process connection request on masterSocket  
        }  
    }  
  
    SDLNet_FreeSocketSet(socketSet);  
    SDLNet_TCP_Close(masterSocket);  
}
```

# Server: actividad en masterSocket

```
char buffer[256]; ← Un buffer para recibir datos  
int result = 0;  
...  
  
if (SDLNet_SocketReady(masterSocket)) {  
    TCPsocket client = SDLNet_TCP_Accept(masterSocket);  
    result = SDLNet_TCP_Recv(client, buffer, 255);  
  
    if (result > 0) {  
        cout << "Client says: " << buffer << endl;  
        SDLNet_TCP_Send(client, "Received!", 10);  
    }  
    SDLNet_TCP_Close(client);  
}  
  
} ← Cerrar el socket
```

Si hay actividad en el masterSocket

Aceptar la conexión

Recibir el mensaje del cliente en buffer, hasta 255 chars

Enviar "Received" al cliente, 10 chars. El décimo es el código ASCII 0 (el último de una cadena de caracteres)

# Client: Conectar al Server

```
void client(char* host, int port) {  
    char buffer[256];  
    int result = 0;  
  
    IAddress ip; ← Resolver el hostname 'host' y rellenar  
    if (SDLNet_ResolveHost(&ip, host, port) < 0) { error(); }  
  
    TCPsocket conn = SDLNet_TCP_Open(&ip); ← Conectar al  
    if (!conn) { error(); }  
  
    // TODO: SEND MSG AND WAIT FOR RESPONSE  
  
    SDLNet_TCP_Close(conn); ← Cerrar el socket  
}
```

Resolver el hostname 'host' y rellenar  
'ip' con la información correspondiente

Conectar al  
servidor

Cerrar el socket

# Client: intercambiar datos

```
// ...  
cout << "Enter a message: ";  
cin.getline(buffer, 255);
```

Pedir un mensaje al usuario y almacenarlo en buffer (el último carácter el código ascii 0)

```
int size = strlen(buffer)+1;
```

La longitud del mensaje, más 1 para el 0 al final.

```
result = SDLNet_TCP_Send(conn, buffer, size);  
if (result != size) { error(); }
```

Enviar el mensaje

```
result = SDLNet_TCP_Recv(conn, buffer, 255);  
if (result < 0) error();  
else if (result == 0) cout << "server closed ...";  
else cout << buffer << endl;  
// ...
```

Esperar la respuesta del servidor

# Inicilizer y Finalizar SDL\_Net

```
if (SDLNet_Init() < 0) {  
    error();  
}
```

Antes de empezar a usar  
SDINet siempre hay que  
llamar a `SDLNet_Init()`

```
SDLNet_Quit();
```

Antes de salir del programa  
llamar a `SDLNet_Quit()`

# Client/Server: ejemplo TCP II

## Server

Como el servidor anterior pero mantiene las conexiones con los clientes y sigue recibiendo mensajes de ellos ...

## Client

1. Conecta al servidor
2. Pide un mensaje al usuario (hasta 255 chars)
3. Si es “exit” cierra la conexión y sale
4. Envía el mensaje al servidor
5. Espera la respuesta del servidor
6. goto 2

# Server: inicialización/bucle principal

```
void server(int port) {  
    // inicializar el servidor como antes  
    ...  
    // array para mantener los sokcets de los clientes  
    const int MAX_CLIENTS = 10;  
    TCPsocket clients[MAX_CLIENTS];  
    for (int i = 0; i < MAX_CLIENTS; i++) { clients[i] = nullptr; }  
  
    while (true) {  
        if (SDLNet_CheckSockets(socketSet, SDL_MAX_UINT32) > 0) {  
            // TODO I: PROCESS DATA on masterSocket  
            // TODO II: PROCESS DATA on client sockets  
        }  
    }  
  
    SDLNet_FreeSocketSet(socketSet);  
    SDLNet_TCP_Close(masterSocket);  
}
```

# Server: actividad en masterSocket

```
if (SDLNet_SocketReady(masterSocket)) {
    TCPsocket client = SDLNet_TCP_Accept(masterSocket);
    // look for a free slot in the array
    int j = 0;
    while (j < MAX_CLIENTS && clients[j] != nullptr) j++;

    if (j < MAX_CLIENTS) {
        clients[j] = client;
        SDLNet_TCP_AddSocket(socketSet, client);
        buffer[0] = 0;
        SDLNet_TCP_Send(client, buffer, 1);
    } else {
        buffer[0] = 1;
        SDLNet_TCP_Send(client, buffer, 1);
        SDLNet_TCP_Close(client);
    }
}
```

# Server: actividad en clientes

```
for (int i = 0; i < MAX_CLIENTS; i++) {
    if (clients[i] != nullptr && SDLNet_SocketReady(clients[i])) {
        result = SDLNet_TCP_Recv(clients[i], buffer, 255);

        if (result <= 0) {
            SDLNet_TCP_Close(clients[i]);
            SDLNet_TCP_DelSocket(socketSet, clients[i]);
            clients[i] = nullptr;
        } else {
            cout << "Client " << i << " says: " << buffer << endl;
            SDLNet_TCP_Send(clients[i], "Received!", 10);
        }
    }
}
```

# Client: conectar

```
void client(char* host, int port) {  
    char buffer[256];  
    int result = 0;  
  
    IPAddress ip;  
    if (SDLNet_ResolveHost(&ip, host, port) < 0) { error(); }  
  
    TCPsocket conn = SDLNet_TCP_Open(&ip);  
    if (!conn) { error(); }  
  
    // TODO I: WAIT FOR CONFIRMATION MESSAGE  
    // TODO II: SEND/RECEIVE LOOP  
  
    SDLNet_TCP_Close(conn);  
}
```

# Client: esperar confirmación

```
result = SDLNet_TCP_Recv(conn, buffer, 1);
if (result < 0) {
    error(); // something went wrong
} else if (result == 0) {
    cout << "The server closed the connection ..." << endl;
} else {
    if (buffer[0] == 0) {
        cout << "Connected!" << endl;
        done = false;
    } else {
        cout << "Connection refused!" << endl;
        done = true;
    }
}
```

# Client: intercambiar datos

```
while (!done) {  
    cout << "Enter a message: ";  
    cin.getline(buffer, 255);  
    if ( strcmp(buffer,"exit") == 0 ) break;  
  
    int size = strlen(buffer)+1;  
    result = SDLNet_TCP_Send(conn, buffer, size);  
    if (result != size) error();  
  
    result = SDLNet_TCP_Recv(conn, buffer, 255);  
    if (result < 0) error();  
    else if (result == 0) {  
        cout << "server closed ..."; break;  
    } else {  
        cout << buffer << endl;  
    }  
}
```

# Resumen Client/Server: TCP

- ◆ Suponemos que el servidor es un programa en una máquina [bla.domain.com](http://bla.domain.com) con IP [145.33.44.66](http://145.33.44.66)
- ◆ El servidor abre un Socket (el [master socket](#)) en un puerto, p.ej., [2000](#), para recibir (escuchar) peticiones de conexión.
- ◆ El cliente abre un Socket y pide conectar al servidor [145.33.44.66:2000](http://145.33.44.66:2000). El Socket también está automáticamente asociado a un puerto local para recibir mensajes, pero no necesitamos saber qué puerto es, todo es implícito.
- ◆ El servidor recibe la petición en su [master socket](#), y cuando la acepta tendrá automáticamente otro socket que se usa para la comunicación con el cliente (el cliente sigue usando el mismo socket) – [el servidor usa el master socket sólo para escuchar peticiones de conexión](#).
- ◆ La conexión se mantiene abierta entre el servidor y el cliente hasta que cierren los sockets. El servidor puede tener varias conexiones abiertas a la vez con clientes (y servirlos de forma forma secuencial o paralela)

# Resumen Client/Server: TCP

- ◆ El servidor y el cliente usan los correspondientes sockets para enviar/recibir **secuencias de bytes** (al nivel del programa) entre ellos.
- ◆ Al nivel de la capa de transporte, esas secuencias de bytes se dividen/acumulan en paquetes (que a parte de los datos llevan más información sobre la conexión, los paquetes, etc).
- ◆ El protocolo TCP garantiza que los paquetes siempre llegaran al destino correctamente y en el mismo orden en que han sido enviados: si un paquete no llega lo vuelve a enviar, si hay un error en un paquete intenta corregirlo o pide que se lo envía otra vez, etc.
- ◆ Los paquetes siempre van en un mismo camino (en la red) que se elige en el momento de establecer la conexión.
- ◆ TCP es fiable, pero puede ser lento para algunas aplicaciones

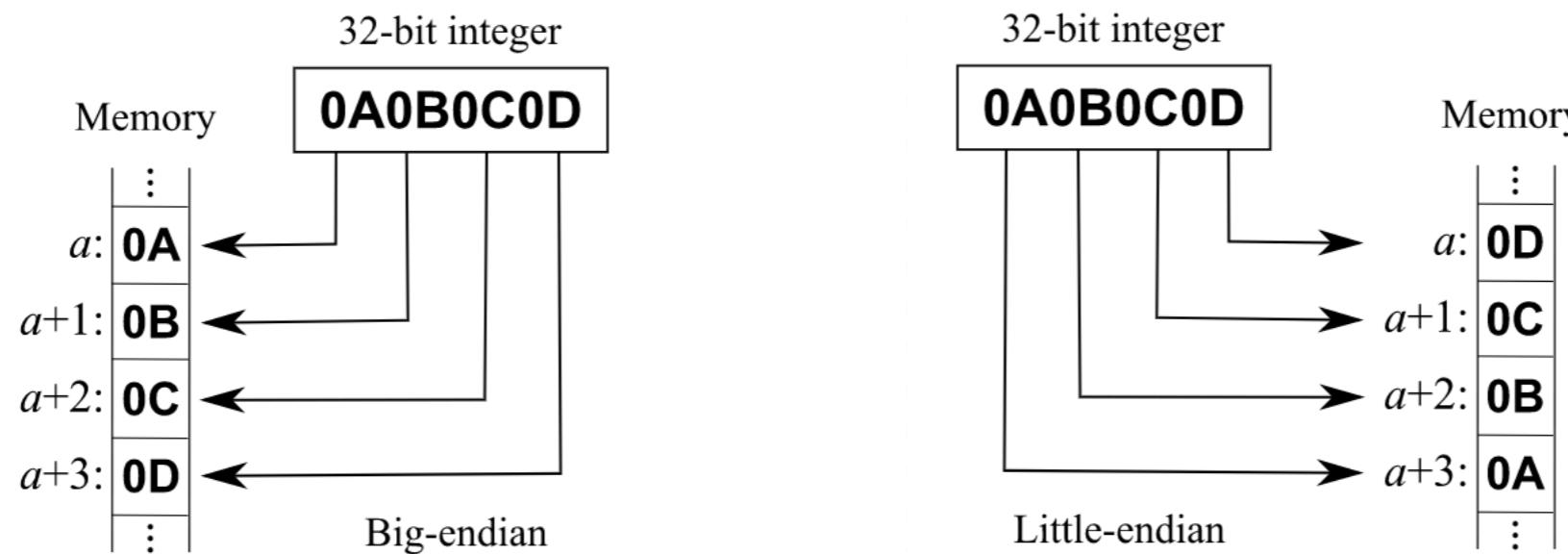
# Serialisation en el contexto de SDLNet

# Structs y intercambio de datos

- ◆ Ya hemos visto en los ejemplos en clase que usar structs para intercambiar datos es cómodo, pero ...
- ◆ No es fiable si no desactivas el padding.
- ◆ No es fiable si no usamos tipos de tamaño fijo (en cualquier compilador). Hay casos que nos pueden sorprender, p.ej., en 64bits el tipo **long** es 8-bytes en g++ pero 4-bytes en Visual Studio (el standard de c++ sólo pide que sea al menos 4).
- ◆ También hay el problema de intercambiar números negativos, porque no todos usan **two's complement** (aunque es muy raro).
- ◆ Ni hablamos de tipos de floating point, como **float**, **double**, ...
- ◆ No es fiable si ignoramos el problema Little/Big-Endian
- ◆ Lo mas fiable es: usar tipos de tamaño fijo y escribir métodos para convertir structs en una secuencia de byte y vice versa. Esto normalmente se llama **Serialization**.

# Little-Endian y Big-Endian

Las arquitecturas pueden representar números (enteros) de manera distinta en la memoria: Least-Significant byte a Most-Significant byte o al revés ...



- ◆ Hay que tener cuidado cuando intercambiamos datos entre programas ejecutando en arquitecturas distintas.
- ◆ La norma (en protocolos de redes) dice: trasmisir los números en Big-Endian y conviértelos al endianness de la maquina local (host) que los recibe.
- ◆ Hay métodos en SDLNet (y en las librerías de sockets) para convertir del “host endianness” a Big-Endian y vice versa.

# Big/Little-Endian en SDLNet

```
inline Uint16 sdlnet_hton(Uint16 v) {  
    Uint16 nv;  
    SDLNet_Write16(v, &nv);  
    return nv;  
}
```

Convierte del endianness de la maquina (host) a big-Endian

```
inline Uint16 sdlnet_ntoh(Uint16 nv) {  
    return SDLNet_Read16(&nv);  
}
```

Convierte de big-Endian al endianness de la maquina (host)

```
inline Uint32 sdlnet_hton(Uint32 v) {  
    Uint32 nv;  
    SDLNet_Write32(v, &nv);  
    return nv;  
}
```

Otras versiones para Uint32

```
inline Uint32 sdlnet_ntoh(Uint32 nv) {  
    return SDLNet_Read32(&nv);  
}
```

# Tipos de tamaño fijo

- ◆ `Uint8`, `Uint16`, `Uint32` (son de SDL, corresponden a `uint8_t`, `uint16_t`, `uint32_t`): todos son **enteros no negativos**.
- ◆ `Sint8`, `Sint16`, `Sint32` (son de SDL, corresponden a `int8_t`, `int16_t`, `int32_t`): todos son **enteros**, mejor transmitir el signo y el valor absoluto y reconstruirlo.
- ◆ `unsigned char` siempre es un byte con valor **entero no negativo** (`uint8_t` es `unsigned char` normalmente).
- ◆ `signed char` siempre es un byte, pero permite valores negativos, tratarlo como `int8_t`.
- ◆ `char` es un bytes, pero es un caso particular, porque algunos compiladores lo implementan como `signed` y otros `unsigned`. Mejor usarlo sólo para representar (cadenas de) caracteres que siempre tienen valor entero no negativo (código ASCII).
- ◆ `float`, se puede usar si cumple la restricción `sizeof(float)==4`, otra posibilidad es enviar la mantissa, el exponente, y el signo cada uno por separado y reconstruirlo ...

# Serialization: serialize y deserialize

```
struct T1 {  
    Uint32 a;  
    Uint8 b;  
    Sint16 c;  
    float d;
```

Escribe una representación de 'a' en la memoria a partir de la dirección buf, y devuelve el puntero buf+n donde n es el tamaño de esa representación

```
inline Uint8* serialize(Uint8 *buf) {  
    buf = _serialize_(a, buf);  
    buf = _serialize_(b, buf);  
    buf = _serialize_(c, buf);  
    buf = _serialize_(d, buf);  
    return buf;  
}
```

```
inline Uint8* deserialize(Uint8 *buf) {  
    buf = _deserialize_(a, buf);  
    buf = _deserialize_(b, buf);  
    buf = _deserialize_(c, buf);  
    buf = _deserialize_(d, buf);  
    return buf;  
};
```

Escribe en 'a' el valor (Uint32) que esta representado en la memoria a partir de la dirección buf, y devuelve buf+n como \_serialize\_

# Serialization: unsigned integer

```
inline Uint8* _serialize_(Uint32 &v, Uint8 *buf) {  
    *reinterpret_cast<Uint32*>(buf) = sdlnet_hton(v);  
    return buf + sizeof(Uint32);  
}
```

↑ serialise lo transmite como Big-Endian

```
inline Uint8* _deserialize_(Uint32 &v, Uint8 *buf) {  
    v = sdlnet_ntoh(*reinterpret_cast<Uint32*>(buf));  
    return buf + sizeof(Uint32);  
}
```

deserialise lo convierte al endianness de la maquina (host)

`reinterpret_cast<Uint32*>(buf)` dice al compilador que trate 'buf' como se fuera de tipo `Uint32*` en tiempo de compilación (al contrario de `static_cast`, ver información en [cppreference.com](http://cppreference.com)).

# Serialization: signed integer (I)

Si las maquinas no usan la misma representación para números negativos, enviamos signo y el valor absoluto y lo reconstruimos

```
inline Uint8* _serialize_(Sint32 &v, Uint8 *buf) {  
    Uint8 sign = v >= 0 ? 0 : 1; ← sign(v)  
    Uint32 abs_v = v >= 0 ? v : -1*v; ← abs(v)  
    Uint32 abs_nv = sdlnet_hton(abs_v);  
    *reinterpret_cast<Uint8*>(buf) = sign;  
    *reinterpret_cast< Uint32*>(buf + 1) = abs_nv;  
    return buf + sizeof(Uint32) + 1;  
}
```

```
inline Uint8* _deserialize_(Sint32 &v, Uint8 *buf) {  
    Uint8 sign = *reinterpret_cast<Uint8*>(buf); ← sign(v)  
    Uint32 abs_nv = *reinterpret_cast<Uint32*>(buf + 1); ← abs(v)  
    Uint32 abs_v = sdlnet_ntoh(abs_nv);  
    sign == 0 ? v = abs_v : v = -1*static_cast<Sint32>(abs_v);  
    return buf + sizeof(Uint32) + 1;  
}
```

# Serialization: signed integer (II)

Si los dos usan la misma representación, por ejemplo **two's complement**, seria suficiente tratar los 4 bytes como se fueran de **Uint32**

```
inline Uint8* _serialize_(Sint32 &v, Uint8 *buf) {  
    return _serialize_(reinterpret_cast<Uint32&>(v),buf);  
}  
  
inline Uint8* _deserialize_(Sint32 &v, Uint8 *buf) {  
    return _deserialize_(reinterpret_cast<Uint32&>(v),buf);  
}
```

# Serialization: float

Tratar los 4 bytes del float como se fueran de Uint32 normalmente es suficiente ...

```
inline Uint8* _serialize_(float &v, Uint8 *buf) {  
    static_assert( sizeof(float) == 4);  
    return _serialize_(reinterpret_cast<Uint32&>(v),buf);  
}  
  
inline Uint8* _deserialize_(float &v, Uint8 *buf) {  
    static_assert( sizeof(float) == 4);  
    return _deserialize_(reinterpret_cast<Uint32&>(v),buf);  
}
```

Otra solución más complicada seria enviar el signo, el exponente y la mantissa, y reconstruirlo.

# Serialization: Ejemplo de uso

Si **T2** es un struct como **T1** pero usa packing, el siguiente código copiaría **x** a **z** de manera correcta (y vice versa):

```
Uint8 buffer[100];
T1 x = {12345, 4, -20, 45.43f};
T2 z;
x.serialize(buffer);
z.deserialize(buffer);
```

# Serialization: \_(de)serialize\_array\_

Simplemente usa los métodos serialize y deserialize para cada elemento

```
template<typename T>
inline Uint8* _serialize_array_(T *v, std::size_t N, Uint8 *buf) {
    for (auto i = 0u; i < N; i++)
        buf = _serialize_(v[i], buf); ←
    return buf;
}
```

```
template<typename T>
inline Uint8* _deserialize_array_(T *v, std::size_t N, Uint8 *buf) {
    for (auto i = 0u; i < N; i++)
        buf = _deserialize_(v[i], buf); ←
    return buf;
}
```

# Serialization: otro ejemplo

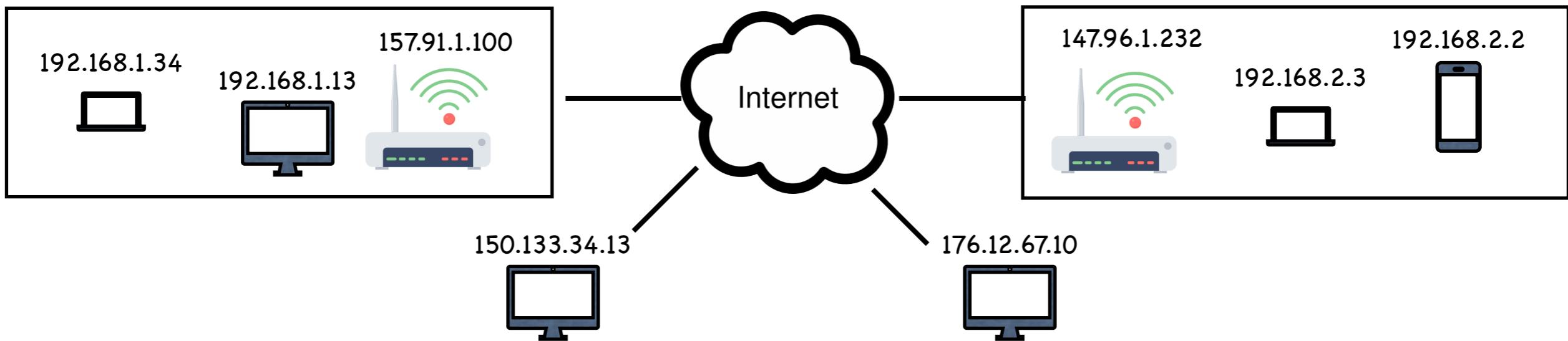
```
struct Data : T1 {  
    Uint32 a;  
    Unit8 b[10]; ←  
    Type c; ←  
    inline Unit8* serialize(Unit8 *buf) {  
        buf = T1::serialize(buf);  
        buf = _serialize_(a, buf);  
        buf = _serialize_array_(b, 10, buf);  
        buf = c.serialize(buf);  
        return buf;  
    }  
};
```

```
inline Unit8* deserialize(Unit8 *buf) {  
    buf = T1::deserialize();  
    buf = _deserialize_(a, buf);  
    buf = _deserialize_array_(b, 10, buf);  
    buf = c.deserialize(buf);  
    return buf;  
}  
};
```

- Hereda de T1 (ejemplo anterior)
- Tiene un array
- Tiene un atributo de tipo Type (suponemos que tiene serialize)

# **Redes locales y acceso desde fuera**

# Redes Locales y Routers



- ◆ En redes locales (como la red wifi en casa) las maquinas normalmente tienen direcciones IPs locales al que no se puede acceder desde fuera (no son válidas globalmente), sólo el router tiene una dirección global ...
- ◆ Para acceder a un servidor local desde fuera, hay que cambiar la configuración NAT del router y hacer **forwarding** de todo el trafico TCP/UDP que llega a un puerto específico a una dirección local específica (donde ejecuta el servidor).
- ◆ Mejor asignar dirección **IP estática** a la maquina del servidor para que siempre tenga la misma dirección (se hace en la configuración del router usando la dirección MAC de la tarjeta wifi/lan de la máquina).

# Redes locales y VPN

- ◆ Otra posibilidad para acceder a ordenadores en una red local es unirlos en la misma red VPN, cada uno tendrá una dirección “local” al que los otros pueden acceder (si los puertos no están bloqueados, como en la VPN de la UCM).
- ◆ VPN (Virtual Private Network) es una red virtual, construida (virtualmente) sobre la red física. Tiene varios objetivos:
  1. Unir a varios ordenadores (locales) en una única red, para que vean uno al otro.
  2. Hacer que la dirección IP de uno parezca de un país distinto.
  3. Tener una conexión más segura con encryption.
  4. ...