UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Programación para videojuegos II

Unidad 5
Introducción al desarrollo de videojuegos en red

Docente
Pablo Abratte

CONTENIDO

En esta unidad abordaremos la programación de aplicaciones que se comunican entre sí a través de redes. Las funcionalidades que SFML nos provee para facilitar esta tarea son similares a las presentes en cualquier biblioteca que permita programar este tipo de aplicaciones, ya que se basan en conceptos básicos que son independientes de cualquier biblioteca o lenguaje de programación. Antes de introducirnos de lleno en los detalles de implementación de este tipo de aplicaciones, presentaremos algunas nociones generales sobre el funcionamiento de las redes de computadoras.

1. Introducción a las redes de computadoras

La comunicación entre dos o más dispositivos es un proceso muy complejo. Para que sea posible superar esta complejidad a la hora de diseñar las redes, éstas se organizan como una pila de capas o niveles, cada una construida a partir de la que está debajo de ella. El propósito de cada capa es ofrecer ciertos servicios a las capas superiores, ocultando a la vez los detalles reales de implementación de los servicios ofrecidos.

El termino *host* es utilizado en informática para referirse a una computadora conectada a una red.

De esta manera, la capa *n* de una máquina o *host* mantiene una conversación con la capa *n* de otra máquina. Las reglas y convenciones utilizadas en esta conversación se conocen de manera colectiva como *protocolo* de capa *n.* Básicamente, un protocolo es un acuerdo entre las partes en comunicación sobre cómo se debe llevar a cabo dicha comunicación. El incumplimiento del protocolo puede dificultar o incluso imposibilitar la comunicación. El esquema de la *Figura 1* ilustra las capas, interfaces entre las mismas y protocolos utilizados en la comunicación de dos hosts.

Un *protocolo* es un acuerdo entre las partes intervinientes sobre cómo debe llevarse a cabo la comunicación. El incumplimiento del protocolo puede dificultar o incluso imposibilitar la comunicación.

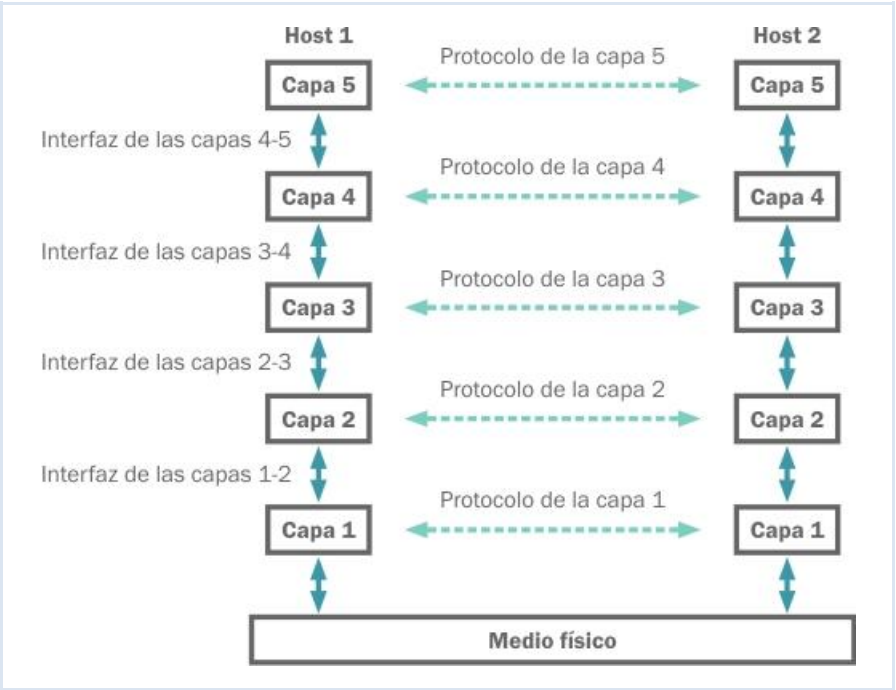

Figura 1. Pila de capas con sus respectivas interfaces y protocolos utilizada en la comunicación de dos hosts.

En realidad, los datos no se transfieren de manera directa desde la capa *n* de una máquina a la capa *n* de la otra, sino que cada capa pasa los datos a la que está inmediatamente debajo, hasta que se alcanza la capa más baja. Debajo de la capa 1 se encuentra el medio físico a través del cual ocurre la transmisión real de los datos. Una vez que la información alcanza la primera capa del segundo host, éste comienza a pasarla a las capas superiores. En la *Figura 1,* la comunicación virtual entre capas se muestra con líneas punteadas, en tanto que la física, con líneas sólidas.

Entre cada par de capas adyacentes hay una interfaz. Ésta define qué operaciones y servicios pone cada capa a disposición de la capa superior inmediata. Las capas más

altas se encuentran más cercanas a las aplicaciones de usuario, mientras que las más bajas guardan mayor relación con el hardware encargado de la transmisión.

Un conjunto de capas y protocolos se conoce como *arquitectura de red.* La especificación de una arquitectura debe contener información suficiente para permitir que un implementador escriba el programa o construya el hardware para cada capa, de modo que se cumpla correctamente con el protocolo apropiado.

Arquitectura de red
Es la especificación de las capas y los protocolos en los cuales se descompone la comunicación. Esta especificación sirve para implementar software y hardware que pueda comunicarse con otros equipos de la red.

1.1. El modelo TCP/IP

Abordaremos nuestro estudio de las redes describiendo de forma general la arquitectura propuesta por el modelo TCP/IP. Los protocolos de dicho modelo son actualmente utilizados en internet. Otro modelo de referencia popular, y muy usado por su generalidad, es el modelo OSI (Open System Interconnection). Sin embargo, los protocolos asociados al mismo se encuentran en desuso.

El modelo TCP/IP surgió en 1970 con la red ARPANET, creada por el Departamento de Defensa de los Estados Unidos. Dicha red fue la antecesora de la actual internet. El modelo propone cuatro capas: la capa de enlace, la capa de red, la capa de transporte y la capa de aplicación.

Modelo TCP/IP
Describe los lineamientos de diseño e implementación de protocolos de una arquitectura de red definida por cuatro capas. Los protocolos propuestos por el modelo son utilizados actualmente en internet.

En la *Figura 2* es posible observar un esquema de capas similar al de la *Figura 1,* pero esta vez con las capas del modelo TCP/IP. El flujo de datos a través de las capas se describe en los párrafos siguientes.

Los programas del usuario se encuentran en la capa de aplicación, en tanto que los datos enviados por los mismos son pasados a la capa inmediatamente inferior: la capa de transporte. Esta capa agrega una cabecera que contiene información para la capa de transporte del host receptor, encapsulando los datos pasados por la capa de aplicación. Los datos de la capa de aplicación con la cabecera de la capa de transporte son pasados nuevamente a la capa inmediatamente inferior. La capa de red encapsulará toda la información recibida de las capas superiores, agregando su propia cabecera con información pertinente a la función que la capa debe realizar, y pasará toda esta información a la capa de enlace. La capa de enlace realizará el mismo trabajo envolviendo todos los datos de las capas superiores con su propia cabecera. Finalmente, toda la información producto de los encapsulamientos sucesivos será enviada a través del medio físico hacia el otro ordenador.

Al recibir la información enviada, la capa de enlace del host de destino quitará la cabecera de capa de enlace, utilizará su información para cumplir sus funciones y pasará los datos a la capa superior. La capa de red quitará también su encabezado y utilizará la información del mismo, pasando luego los datos a la capa superior. Este proceso se repetirá en la capa de transporte para finalmente entregar los datos a la aplicación a la que estaban destinados. De esta forma, cada capa de la entidad receptora utiliza la información que la capa equivalente del emisor colocó en el paquete.

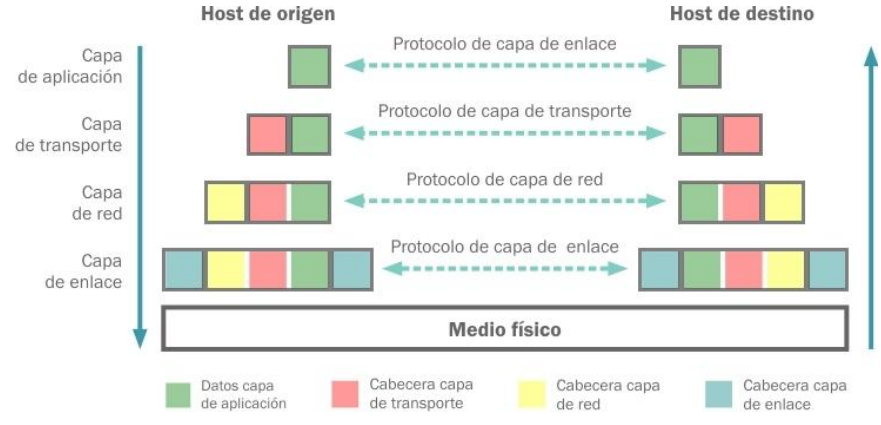

Figura 2. Envío de datos de un host a otro en el modelo TCP/IP

El encapsulamiento de la información a través de las distintas capas es necesario, ya que cada capa realiza una función específica y necesaria para poder llevar a cabo de manera exitosa la comunicación. A continuación, brindaremos una breve explicación de los objetivos y funcionamiento de cada una de las capas de este modelo.

Capa de enlace

La *capa de enlace* tiene como objetivo posibilitar la transmisión de datos entre dispositivos que están directamente conectados por un cable o enlace. Para esto, el protocolo utilizado en esta capa debe proveer mecanismos para sincronizar la transmisión entre los dispositivos, es decir, decidir quién transmite y quiénes escuchan. Otro asunto importante del cual debe encargarse un protocolo de capa de enlace es el direccionamiento, esto es, brindar alguna forma de identificar al dispositivo que transmite y al de destino.

Switch o conmutador
Dispositivo de comunicación que opera en la capa de enlace del modelo TCP/IP. Permite conectar varios segmentos de una única red.

El protocolo de capa de enlace utilizado generalmente en internet es el protocolo *Ethernet*. Como mencionamos anteriormente, el objetivo de este protocolo es gestionar el transporte de datos entre un origen y destino que están dentro de una misma red.

Para identificar emisor y receptor, cada dispositivo posee una dirección MAC. Ésta es un identificador único que consta de seis octetos (bytes) y viene generalmente grabada de fábrica en el firmware del dispositivo, por lo que no puede alterarse. Cada placa de red de una computadora, ya sea inalámbrica o cableada, posee una dirección MAC.

En la cabecera de los paquetes del protocolo, Ethernet contiene –entre otras cosas– la dirección MAC del dispositivo que envió el paquete y la del destino al cual debe ser entregado.

Los aparatos que actúan en esta capa de la comunicación se denominan *conmutadores* o *switches* y hacen posible conectar varios segmentos de una red. Un *switch* tiene varias entradas para cables Ethernet y mantiene, internamente, una tabla con las direcciones MAC de los dispositivos conectados a cada una de sus entradas.

Cuando llega un paquete por una entrada determinada, el conmutador lo almacena por unos instantes hasta que sea su turno de transmitir. Entonces, a partir de la dirección MAC del dispositivo de destino que figura en la cabecera del paquete, lo retransmitirá por la salida correspondiente que figure en la tabla.

En la *Figura 3* podemos observar la parte posterior de un conmutador y el esquema de una red de cinco computadoras conectadas mediante uno.

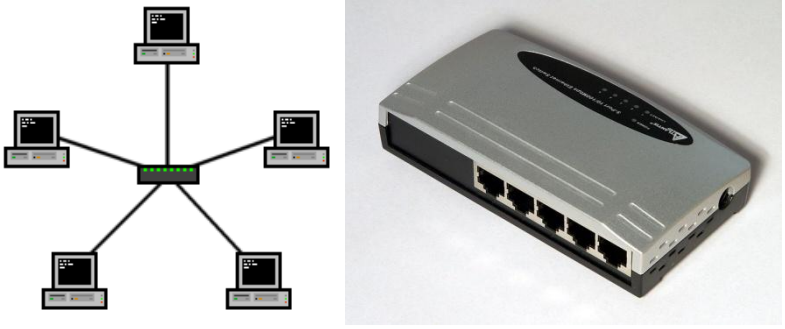

Figura 3. Esquema de una red de cinco computadoras conectadas mediante un conmutador (derecha) y la fotografía de la parte posterior de un switch de cinco bocas (izquierda).

Capa de red

La *capa de red* es la encargada de que la información pueda llegar desde el origen al destino, aún cuando los hosts no estén directamente conectados. Este proceso implica que los paquetes viajen a través de varias redes para llegar al equipo destinatario de los mismos. En la *Figura 4* puede observarse esta situación: el proceso P1 que corre en el host H1 necesita comunicarse con el proceso P2 que está corriendo en la máquina

H2. Los equipos se encuentran en redes diferentes y las mismas están a la vez conectadas a la red de la empresa proveedora del servicio de internet.

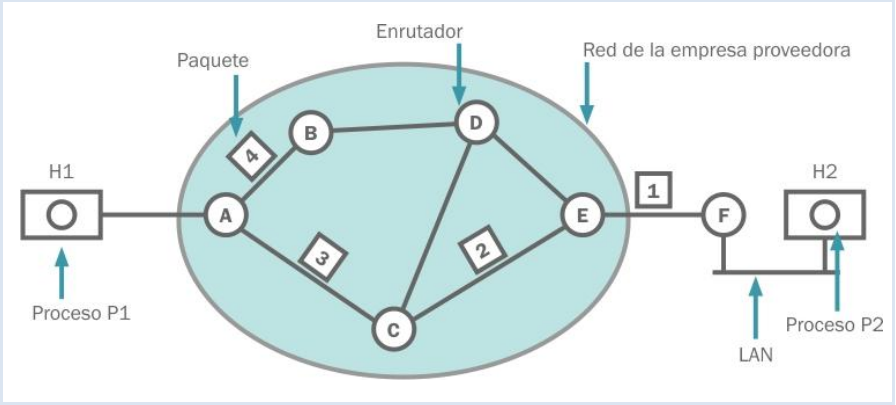

Figura 4. Comunicación de dos computadoras a través de una red.

La capa de red se encarga de dividir la información en fragmentos o paquetes y enviarlos a través de la red para que alcancen su destino. Para llegar al mismo, dichos paquetes pueden requerir pasar por varios equipos, incluso pueden existir múltiples caminos al destinatario, por lo que es preciso contar con alguna forma de encontrar la mejor ruta y evitar que los paquetes circulen indefinidamente por la red.

El principal protocolo de esta capa es el *IP* (Internet Protocol). Este protocolo y el protocolo *TCP* (Transfer Control Protocol), utilizado en la capa de transporte, constituyen el corazón de internet.

En esta capa, al igual que en la de enlace, es necesario un esquema de direccionamiento que permita identificar al emisor y receptor de los paquetes. Para esto, cada dispositivo de la red es identificado con una dirección IP que consiste en cuatro octetos (por ejemplo, 192.168.0.120).

Los dispositivos encargados de llevar los paquetes desde su origen al destino reciben el nombre de *encaminadores* o *routers* (en la *Figura 3* están simbolizados con las letras *A-F*). Estos dispositivos tienen la capacidad de intercambiar información entre ellos para conocer la topología de la red y decidir cuál es el mejor camino para que un paquete llegue a su destino. El proceso realizado por un router, de decidir por dónde reenviar un paquete recibido, se conoce generalmente como *enrutamiento.*

Router o encaminador
Dispositivo que opera en la capa de red del modelo TCP/IP. Decide cuál es el camino que debe tomar un paquete para cruzar internet y llegar a su destino. Para lograr esto, intercambia información con otros routers con los que está conectado mediante un protocolo especial.

Existen diferentes tipos de routers según el desempeño que se necesite. Los routers utilizados en casas u oficinas (denominados routers de acceso) son muy similares al switch mostrado anteriormente, ya que en general poseen el mismo tipo de entradas. Sin embargo, cumplen funciones diferentes. En la *Figura 5* pueden observarse un router de acceso (izquierda) y la parte posterior de un router de mayor tamaño para conectar redes troncales (derecha).

Figura 5. Router de oficina (izquierda) y router de gran tamaño utilizado en la conexión de redes de backbone o troncales (derecha).

En el ejemplo de la *Figura 4,* el host H1 segmenta en cuatro paquetes (1, 2, 3 y 4) la información que desea enviar y luego, los manda al router A (que actúa como gateway o puerta de enlace de la red donde se encuentra H1).

Gateway o puerta de enlace
Es el dispositivo de una red (generalmente un router) al que se envía todo el tráfico que debe salir hacia internet.

El router A conoce, a partir de la información intercambiada con los encaminadores B y C, los caminos posibles a la máquina H2 y envía los paquetes a sus routers vecinos para que éstos, a su vez, los reenvíen a sus propios vecinos hasta que finalmente lleguen al host de destino.

Un hecho muy importante es que el enrutamiento de cada paquete se realiza de manera independiente al resto, es decir, que puede transitar un camino distinto según las decisiones tomadas por los routers, teniendo en cuenta la velocidad y congestión de cada enlace en el momento de reenviar el paquete.

La cabecera de la capa de red contiene, entre muchas otras cosas, la dirección IP de las máquinas de origen y destino y un número entero denominado *TTL* (time to live), que indica la vida del paquete, el cual es decrementado cada vez que dicho paquete pasa por un router. Esto permite a los routers descartar los paquetes cuyo tiempo de vida es menor a cero, en lugar de retransmitirlos, y evitar así que esos paquetes circulen por la red infinitamente sin llegar a destino.

A pesar de proveer mecanismos para que los paquetes atraviesen las redes y alcancen su destino, la capa de red no ofrece ningún tipo de garantías de que lleguen a destino o de que lo hagan en orden. Incluso puede ocurrir que lleguen paquetes duplicados.

Capa de transporte

La *capa de transporte* gestiona la conexión de extremo a extremo entre dos hosts, ocultando la complejidad de las redes a través de las cuales ambos están conectados (lo cual es manejado por la capa de red).

A diferencia del código de capa de red, que se ejecuta dentro de los routers, las funciones de la capa de transporte se ejecutan en las computadoras de los usuarios que establecen la conexión. Esto es muy importante porque implica que los programadores pueden utilizar los servicios brindados por esta capa en sus aplicaciones para que las mismas se comuniquen a través de la red.

Al igual que las capas anteriores, la capa de transporte propone un esquema de direccionamiento. El mismo consiste en identificar las conexiones de capa de transporte mediante un número entre 0 y 65535, denominado *puerto.* Esto permite que un único host (con una única dirección IP) pueda establecer muchas conexiones simultáneas a través de distintos puertos.

Los puertos del 0 al 1023 se denominan *puertos bien conocidos,* ya que están reservados para ser utilizados por aplicaciones estándar. Por ejemplo, los servidores web están a la escucha por conexiones entrantes en el puerto 80 y el cliente puede conectarse al puerto 80 del servidor utilizando cualquier número de puerto local. Esto significa que para establecer la conexión no es necesario que el número de puerto sea el mismo en cliente y servidor, aunque es necesario que el puerto utilizado por el servidor sea conocido por el cliente.

Esta capa ofrece dos tipos de servicios: uno orientado a la conexión y otro no orientado a la conexión. A continuación explicaremos ambos.

Servicio orientado a la conexión

Anteriormente mencionamos que la capa de red se encarga de que los paquetes encuentren su camino desde el origen al destino, pero no garantiza la entrega, el orden o unicidad de los mismos.

Estas deficiencias son salvadas por el protocolo TCP, el cual se encarga de ordenar los paquetes entrantes, pedir al emisor que reenvíe los que no llegaron o lo hicieron con errores y descartar los repetidos, así como de regular la velocidad de transmisión del emisor para que no sobrepase la capacidad del receptor. Se dice que TCP es un protocolo de streaming, ya que se ocupa de reconstruir toda la información enviada en

los paquetes independientes del protocolo IP y presentarle al usuario un único flujo de datos, idéntico al que fue enviado por el emisor.

Al ser un protocolo orientado a la conexión, TCP requiere una fase de establecimiento de la conexión antes de que pueda transmitirse información. Además, una vez finalizada la transmisión y recepción de datos, requiere otra fase de liberación de la conexión. Mientras la conexión esté activa entre dos hosts, ninguno de los dos podrá establecer otra conexión en el mismo puerto.

El diagrama de la *Figura 6* ilustra los estados durante el proceso de comunicación entre dos computadoras, utilizando este protocolo: el primer ordenador se encuentra inicialmente en reposo (desconectado) y puede optar por ponerse a la escucha de conexiones entrantes (conexión pasiva) o intentar conectarse con un equipo remoto especificando su número de IP y el puerto al cual desea conectarse (intento de conexión activa).

El equipo que se encuentra a la escucha de conexiones entrantes se conoce generalmente como equipo s*ervidor,* mientras que los demás equipos que se conectan al mismo reciben la denominación de *clientes.* En ambos casos, el equipo servidor debe aceptar la conexión entrante para que ambos equipos den por establecida la conexión. Luego podrán proceder al intercambio de información en cualquiera de los dos sentidos hasta que alguno de los dos equipos notifique a su contraparte que desea desconectarse. Finalmente, la conexión será liberada y ambos equipos regresarán a su estado inicial, quedando nuevamente preparados para establecer una nueva conexión en el puerto utilizado.

La cabecera del protocolo TCP posee muchos campos, entre ellos, el número de puerto de origen y destino de los hosts emisor y receptor, un número de secuencia para que el destinatario pueda ordenar los paquetes, una suma de verificación o checksum para saber si el paquete llegó correctamente y otros campos para el control de flujo entre emisor y receptor.

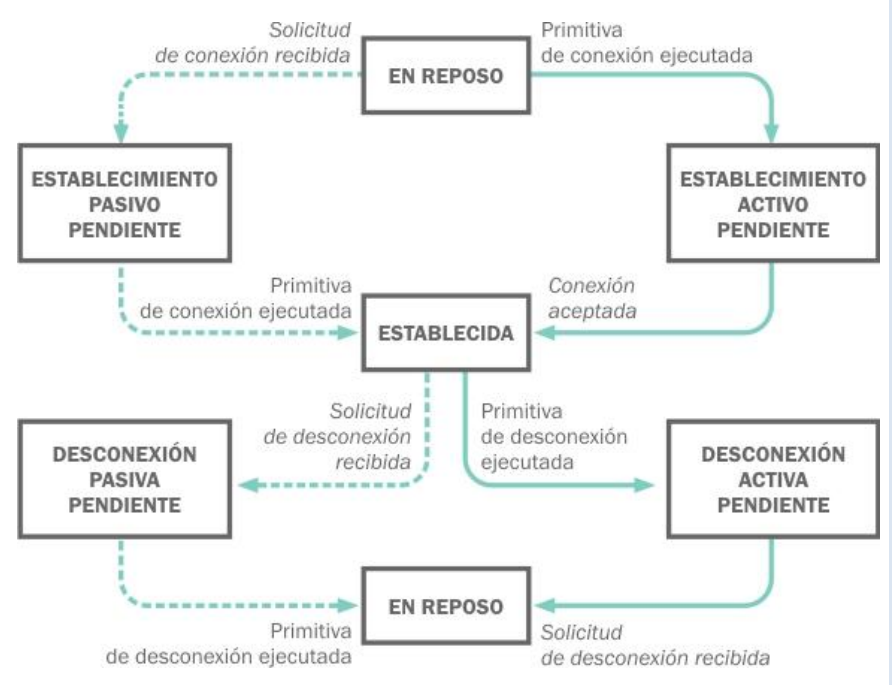

Figura 6. Estados por los que pasa un host durante una comunicación con el protocolo TCP.

Servicio no orientado a la conexión

El protocolo TCP que vimos anteriormente provee una transmisión fiable de extremo a extremo, realizando –entre otras cosas– control de errores y de flujo. Sin embargo, requiere etapas de establecimiento y liberación de la conexión, paquetes con grandes

cabeceras y reenvío de paquetes erróneos. Esto produce una gran degradación en la velocidad de transmisión de los datos, convirtiendo a TCP en un protocolo pesado.

Por esta razón, la capa de transporte también ofrece el protocolo *UDP* (User Datagram Protocol).

A diferencia de TCP, que es un protocolo de streaming, UDP (al igual que IP) es un protocolo de datagramas, lo cual significa que cada paquete enviado es totalmente independiente del resto. UDP no realiza ningún tipo de control sobre los datagramas recibidos, por lo cual nunca se puede tener la seguridad de que los datagramas lleguen o que lo hagan en orden o sin errores. Los paquetes, incluso, pueden llegar repetidos. Además, UDP no requiere ningún tipo de establecimiento o liberación de la conexión; los datagramas simplemente se envían especificando la dirección IP y el número de puerto del destinatario. Por otra parte, el equipo servidor puede ponerse a la escucha de datos entrantes en un determinado puerto sin necesidad de aceptar una conexión.

La cabecera del protocolo UDP es pequeña y sencilla y posee solamente los números de puerto de origen y destino de los hosts emisor y receptor.

En otras palabras, el protocolo UDP funciona de forma muy similar a IP, sólo que en capa de transporte. Asimismo, permite a un solo host (que posee una única dirección IP) establecer múltiples comunicaciones a través de distintos puertos.

Cabe destacar que los puertos utilizados para el protocolo TCP son distintos a los usados para UDP. Es decir, un host puede establecer una conexión en el puerto *n* con el protocolo TCP y, al mismo tiempo, otra conexión en el mismo puerto utilizando UDP.

A pesar de no garantizar una entrega fiable de datos, UDP es un protocolo liviano y resulta muy útil cuando la velocidad de transmisión de los datos es más importante que la confiabilidad. Es el caso, por ejemplo, de la transmisión de video en tiempo real, donde no importa si un cuadro llega repetido o con errores, sino que llegue a destino lo más rápido posible.

Otra característica interesante de UDP es que permite a un emisor transmitir información a múltiples destinos *(multicast).* La comunicación en TCP, en cambio, se produce entre un emisor y un único receptor *(unicast).*

Capa de aplicación

La *capa de aplicación* refiere, en palabras simples, al formato de la información transmitida por las aplicaciones que se comunican. Por ejemplo, si en un juego en red deseáramos transmitir la posición y energía de nuestro personaje, el paquete de información contendría tres valores numéricos: su coordenada en *x,* su coordenada en *y* y su energía (en este orden). La contraparte que reciba dicha información, deberá saber el significado de cada uno de estos valores y su posición en el paquete. Este formato en los datos constituye un protocolo de capa de aplicación que ambos programas deben respetar si desean comunicarse de manera exitosa.

Algunos ejemplos de protocolos de capa de aplicación conocidos son *HTTP* (Hypertext Transfer Protocol) para el acceso a páginas web, *FTP* (File Transfer Protocol) para la transmisión de archivos, *SMPT* (Simple Mail Transfer Protocolo) y *POP* (Post Office Protocol), que permiten el envío y recepción de correo electrónico.

Un protocolo de capa de aplicación que merece una mención especial por su importancia es *DNS* (Domain Name System), el cual permite asociar nombres a direcciones IP. Si no existiese dicho protocolo, sería necesario conocer la dirección IP de cada máquina con la que se quisiera establecer una conexión. La *Figura 7* ilustra el funcionamiento de DNS: el usuario desea conectarse al servidor www.google.com, pero no conoce su dirección IP. Conoce, en cambio, la dirección IP del servidor DNS de su proveedor de internet, con el cual puede conectarse para consultar por la dirección IP del destino deseado. Una vez que el servidor DNS responde con la dirección IP del servidor buscado, el usuario puede intentar una conexión con el mismo.

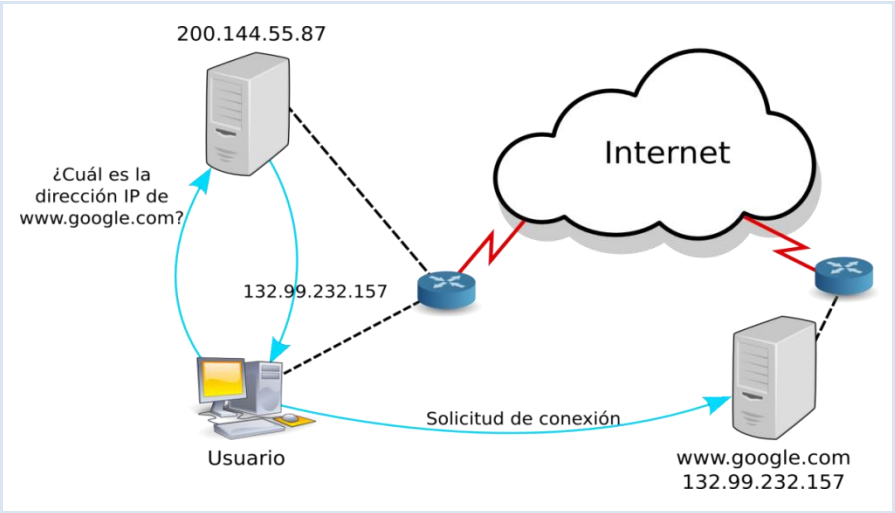

Figura 7. Proceso de conexión a un servidor mediante una consulta a un servidor DNS.

2. Programación de aplicaciones en red con SFML

Para crear programas que puedan comunicarse entre ellos a través de la red, SFML, al igual que la mayoría de las bibliotecas con este propósito, pone a nuestra disposición los denominados *sockets*.

Los sockets son abstracciones que representan los puntos terminales de una conexión bidireccional entre dos procesos hosts que se comunican mediante protocolos de red. Es decir, son como los extremos de una tubería que conecta dos procesos que se ejecutan en máquinas conectadas a una red. Un socket se define a partir de una dirección IP y un número de puerto, de manera que los datos enviados a un socket por un host son recibidos por otro socket en el otro extremo de la conexión.

Los *sockets* son abstracciones que representan los puntos terminales de una conexión bidireccional entre dos procesos hosts que se comunican mediante protocolos de red. Un socket se define a partir de una dirección IP y un número de puerto.

Los sockets son las entidades más básicas con las que se construyen las aplicaciones en red. SFML nos proporciona las clases *sf::SocketTCP* y *sf::SocketUDP* para trabajar con dichos protocolos. En los siguientes parágrafos mostraremos la utilización y algunos ejemplos de ambas.

2.1. Sockets TCP

Como mencionamos anteriormente, el protocolo TCP provee una transmisión fiable de los datos a expensas de ser un poco más lento y pesado que UDP.

Para transferir datos con el protocolo TCP, SFML pone a nuestra disposición la clase *sf::SocketTCP.* Una instancia de esta clase representa un extremo de la comunicación, a partir del cual podremos transmitir y recibir datos del proceso ubicado en el otro extremo de la comunicación.

Debemos tener en cuenta que antes de transferir datos en una comunicación mediante el protocolo TCP, es necesaria una fase de establecimiento de la conexión, así como una fase de liberación de la conexión luego de finalizada la transferencia.

El siguiente código, que explicaremos a continuación, corresponde a un programa simple que se conecta con otro proceso (conexión activa) para intercambiar información:

```
#include <iostream>
#include <SFML/System.hpp>
#include <SFML/Network.hpp>
using namespace std;

int main(int argc, char *argv[]) {
    cout<<"Ejemplo de sockets TCP en SFML (cliente)"<<endl;

    sf::SocketTCP serverSocket;
    const unsigned port=1712;

    // intenta conectarse a 127.0.0.1 en el puerto 1712
    if(serverSocket.Connect(port, "127.0.0.1") != sf::Socket::Done){
        cout<<"ERROR: no pudo establecerse la conexion"<<endl;
        return -1;
    }

    // una vez conectado, podemos recibir datos desde el servidor
    // mediante serverSocket
    char buffer[128];
    size_t Received;
    if(serverSocket.Receive(buffer, sizeof(buffer), Received)
        != sf::Socket::Done){
        cout<<"ERROR: no pudieron recibirse los datos"<<endl;
    }

    // muestra el mensaje del servidor
    cout<<buffer<<endl;

    // lee un mensaje desde consola
    string mensaje;
    cout<<"Ingrese el mensaje que desea enviar: ";
    getline(cin, mensaje);

    // y envia el mensaje al servidor
    if(serverSocket.Send(mensaje.c_str(), mensaje.size()+1)
        != sf::Socket::Done){
        cout<<"ERROR: fallo el envio del mensaje"<<endl;
    }else{
        cout<<"Mensaje enviado con exito."<<endl;
    }

    // finaliza la conexion
    serverSocket.Close();
    return 0;
}
```

Como vemos, en este código se crea una instancia de *sf::TCPSocket,* denominada *serverSocket.*

Mediante este objeto se podrá establecer la conexión y luego enviar y recibir datos del servidor. Para realizar la conexión, se llama a la función miembro *Connect(),* la cual recibe como parámetros la dirección IP y el puerto del servidor con el que se desea conectarse; en este caso, el puerto 1712 del host cuya dirección IP es 127.0.0.1.

La función *Connect(),* al igual que muchas funciones de un objeto de tipo *sf::Socket*, devuelve el estado del mismo mediante un valor del tipo *sf::Socket::Status,* que puede tomar los siguientes valores:

- *sf::Socket::Done:* la operación fue completada exitosamente.
- *sf::Socket::NotReady:* el socket todavía está ocupado realizando la última tarea solicitada (esto sólo es válido al utilizar sockets asíncronos).
- *sf::Socket::Disconnected:* el socket se ha desconectado.
- *sf::Socket::Error:* ha ocurrido algún error inesperado.

En el ejemplo, se compara el valor retornado por las llamadas a las funciones *Connect(), Send()* y *Receive()* con la constante *sf::Socket::Done*. Si son distintos se muestra un mensaje de error, ya que esto significa que la operación no ha sido exitosa.

Una vez establecida la conexión, la llamada a la función *Receive()* permite recibir un mensaje de bienvenida del servidor. La función recibe tres parámetros: el buffer donde se almacenará la información, el tamaño de dicho buffer y un valor de tipo *size_t,* pasado por referencia, donde la función colocará la cantidad real de bytes que ha colocado en el buffer. El tipo de dato utilizado para el buffer es *char,* ya que el tamaño del mismo es un byte. La función *Receive()* devuelve un valor indicando el estado del socket de la misma manera que *Connect().*

Una vez recibido y mostrado el mensaje de bienvenida del servidor, el cliente solicita un mensaje al usuario para luego enviarlo al servidor. El envío de datos se realiza mediante la función miembro *Send().* Ésta, al igual que *Receive(),* recibe como parámetros el buffer y el tamaño del mismo. Para obtenerlos, se utilizan las funciones miembros *c_str()* y *size()* de string que devuelven, respectivamente, el puntero a la memoria con los datos y la cantidad de caracteres en la cadena (sin contar el carácter de finalización). *Send()* devuelve un valor indicando el estado del socket de la misma manera que *Connect().*

Dirección de loopback
Es la dirección de IP 127.0.0.1 que está reservada para referenciar al propio host dentro del cual el proceso está corriendo. Permite que dicho host simule ser servidor y cliente al mismo tiempo para, entre otras cosas, comprobar el funcionamiento de los programas.

Una vez finalizado el intercambio de datos, el socket debe llamar a la función *Close()* para notificar a su contraparte de su intención de finalizar la comunicación.

Cabe mencionar que la elección de la dirección IP 127.0.0.1 para establecer la conexión no es casual, ya que la misma es una dirección reservada conocida como *dirección de loopback* y representa al propio host en el que el proceso está ejecutándose. Tener esta dirección IP especial nos provee de una ventaja muy importante: nos permitirá correr tanto el proceso cliente como el servidor en una única computadora; es decir, posibilitará que la máquina simule una conexión consigo misma. De esta forma, podremos comprobar el funcionamiento de los programas que desarrollemos sin necesidad de contar con varias computadoras conectadas en red.

A continuación, mostraremos el código del programa servidor con el cual el cliente presentado anteriormente se conectará e intercambiará datos:

```
#include <iostream>
#include <SFML/System.hpp>
#include <SFML/Network.hpp>
using namespace std;

int main(int argc, char *argv[]) {
    cout<<"Ejemplo de sockets TCP en SFML (Servidor)"<<endl;

    // sockets para el servidor y cliente
    sf::SocketTCP serverSocket, clientSocket;
    const unsigned port=1712;

    // pone al socket del servidor a escuchar
    // por conexiones entrantes en el puerto 1712
    serverSocket.Listen(port);

    // acepta la conexion utilizando el socket clientSocket para
    // referirse al otro equipo (al cliente)
    serverSocket.Accept(clientSocket);

    // envia un mensaje al cliente
    string saludo="Hola. Soy el servidor.";
    if (clientSocket.Send(saludo.c_str(), saludo.size()+1)
        != sf::Socket::Done){
        cout<<"ERROR: fallo el envio del mensaje"<<endl;
    }

    // recibe un mensaje desde el ciente
    char buffer[128];
    std::size_t Received;
    if(clientSocket.Receive(buffer, sizeof(buffer), Received)
       != sf::Socket::Done){
```

```
        cout<<"ERROR: fallo recepcion del mensaje"<<endl;
    }

    // muestra el mensaje
    cout<<"Mensaje recibido: "<<buffer<<endl;

    // cerramos las conexiones
    clientSocket.Close();
    serverSocket.Close();
    return 0;
}
```

La forma de programar el servidor es levemente diferente a la del cliente.

Utilizaremos dos sockets TCP que denominaremos *serverSocket* y *clientSocket.* El primer socket escuchará por conexiones entrantes y para esto se llamará a la función miembro *Listen()* de *serverSocket,* que recibe como parámetro el número de puerto en el cual debe escuchar para recibir conexiones. En este caso, el programa servidor recibirá pedidos de conexión de cualquier dirección IP, pero en el puerto 1712.

Para que la comunicación se establezca, es necesario llamar a la función *Accept(),* a fin de aceptar la conexión entrante. El parámetro de esta función es el socket que utilizaremos para comunicarnos con el cliente. De esta forma, la conexión del servidor con el cliente se realizará mediante el objeto *clientSocket,* utilizando otro puerto elegido por el sistema operativo, mientras que *serverSocket* quedará a la espera de nuevas conexiones entrantes en el puerto 1712 y podrá, eventualmente, aceptarlas.

La función *Accept()* puede recibir opcionalmente un puntero a un objeto del tipo *sf::IPAdress,* el cual rellenará con la dirección IP del cliente.

El intercambio de datos se realizará mediante el socket *clientSocket,* de la misma forma que mostramos para el caso del cliente. Luego de establecida la conexión, el servidor utilizará la función *Send()* para enviar un mensaje de bienvenida al cliente y luego llamará a *Receive()* para recibir una cadena de texto del mismo.

Por último, es necesario cerrar ambos sockets para notificar a la contraparte de que deseamos finalizar la conexión, así como avisar al sistema operativo que ya hemos terminado de utilizar los puertos y que puede disponer de ellos para nuevas conexiones.

2.2. Sockets UDP

Para utilizar el protocolo UDP, SFML nos ofrece la clase *sf::UDPSocket.* Recordemos que el protocolo UDP es más rápido y liviano que TCP, pero no ofrece ninguna garantía de la entrega de la información enviada. Asimismo, para comunicarse mediante sockets UDP, no son necesarias las fases de establecimiento y liberación de la conexión.

A continuación, mostramos el código del programa cliente, desarrollado anteriormente, pero utilizando el protocolo UDP:

```
#include <iostream>
#include <SFML/System.hpp>
#include <SFML/Network.hpp>
using namespace std;

int main(int argc, char *argv[]) {
    cout<<"Ejemplo de sockets UDP en SFML (cliente)"<<endl;

    sf::SocketUDP serverSocket;
    const unsigned port=1712;

    // lee un mensaje desde consola
```

```
    string mensaje;
    cout<<"Ingrese el mensaje que desea enviar: ";
    getline(cin,mensaje);

    // y envia el mensaje al otro host
    if(serverSocket.Send(mensaje.c_str(), mensaje.size()+1,
"127.0.0.1", port) != sf::Socket::Done){
        cout<<"ERROR: fallo el envio del mensaje"<<endl;
    }else{
        cout<<"Mensaje enviado con exito."<<endl;
    }

    // libera el socket
    serverSocket.Close();
    return 0;
}
```

Como vemos, el código resulta mucho más corto, ya que no es necesario establecer ni liberar la conexión. Tampoco se recibe el mensaje de bienvenida por parte del servidor, dado que la conexión nunca se establece, sino que la información es enviada sin ningún tipo de preámbulo. La función *Send()* de *sf::UDPSocket* opera de la misma manera que la homónima para los sockets TCP. Sin embargo, recibe dos parámetros extras con la dirección IP y el puerto del destinatario, ya que el socket no se encuentra conectado a ningún host en particular.

La llamada al método *Close()* sigue resultando conveniente, pero no para liberar la conexión, pues la misma nunca se establece, sino para notificar al sistema operativo que ya no utilizaremos los recursos empleados por el socket.

Ahora mostramos el código del servidor desarrollado anteriormente, utilizando sockets UDP:

```
#include <iostream>
#include <SFML/System.hpp>
#include <SFML/Network.hpp>
using namespace std;


int main(int argc, char *argv[]) {
    cout<<"Ejemplo de sockets UDP en SFML (Servidor)"<<endl;

    // sockets para el servidor y cliente
    sf::SocketUDP clientSocket;
    short unsigned port=1712;

    // asocia el socket al puerto
    clientSocket.Bind(port);

    // recibe un mensaje desde el ciente
    char buffer[128];
    std::size_t Received;
    sf::IPAddress a("127.0.0.1");
    if(clientSocket.Receive( buffer, sizeof(buffer),
                             Received, a, port)!=sf::Socket::Done){
        cout<<"ERROR: fallo recepcion del mensaje"<<endl;
    }

    // muestra el mensaje
    cout<<"Mensaje recibido: "<<buffer<<endl;

    // cerramos las conexiones
    clientSocket.Close();
    return 0;
}
```

El servidor no envía un saludo al cliente, dado que este nunca se conecta. Para recibir los datos, el programa utiliza la función *Receive()* miembro de *sf::UDPSocket,* la cual

posee los mismos parámetros que la función de recepción de los sockets TCP, como así también dos variables extra que serán rellenadas con el puerto y la dirección IP del equipo remitente.

Antes de utilizar la función *Receive()* es necesario avisarle al sistema operativo que eventualmente estaremos empleando el puerto para recibir datos, pese a no establecer ninguna conexión. Esto se logra llamando a la función miembro *Bind()* con dicho número de puerto. De esta manera, se asocia al socket con el puerto para que el sistema operativo no lo destine a otro uso.

La llamada a *Close()* notifica al sistema operativo de que ya no utilizaremos el puerto y que puede disponer de él.

2.3. Paquetes

El intercambio de datos a través de una red puede resultar más complicado de lo que parece, ya que diversos problemas pueden surgir cuando nuestros programas tratan de comunicarse a través de internet utilizando diferentes plataformas.

El primero de estos problemas es el orden en que el hardware de una plataforma específica almacena los bits. Esto, en inglés, es conocido como *endianess.* Hay dos formas principales de almacenar los bits: *big-endian,* en la cual los bits más significativos se almacenan a la derecha, y *little-endian*, en la que los bits más significativos son almacenados a la izquierda.

Para ejemplificar este problema, supongamos que enviamos un entero de 16 bits con codificación little-endian (los procesadores Intel x86 utilizan esta codificación) y el servidor lo recibe y lo interpreta como big-endian (por ejemplo, los procesadores Apple PowerPC). Si enviamos un 48 (00000000 00110000), el servidor lo interpretará, en realidad, como 3072 (00001100 00000000).

Otro problema es el tamaño de los tipos primitivos de datos. Es decir, diferentes plataformas y compiladores pueden utilizar tamaños distintos para el mismo tipo de dato. Por ejemplo, un puntero ocupa cuatro bytes en una plataforma de 32 bits, pero en una de 64 bits su tamaño será de ocho bytes y los datos enviados no serán interpretados de forma correcta.

Para resolver algunos de estos problemas relacionados con el manejo de bajo nivel de los datos y facilitarnos el envío y recepción de los mismos, SFML nos proporciona la clase *sf::Packet* para representar un paquete de datos. Un objeto de tipo *sf::Packet* puede usarse de la misma manera que un flujo (por ejemplo, *cout*) para agregar u obtener del mismo distintos datos, tal como mostramos en los siguientes fragmentos de código:

```
// insercion de datos en un paquete
int x = 24;
string s = "hola";
float f = 59864.265f;

sf::Packet paqueteParaEnviar;
paqueteParaEnviar << x << s << f;
...
```

```
// obtencion de datos de paquete
int x;
string s;
float f;

sf::Packet paqueteRecibido;
paqueteRecibido >> x >> s >> f;
...
```

Pueden ocurrir errores si se trata de leer más información que la que el paquete contiene. En este caso, la operación de lectura fallará y se encenderá una bandera dentro del paquete indicando su estado.

Para comprobar el estado de error de la lectura, se puede proceder de las siguientes formas:

```
// la primer forma es realizar la comparacion con el objeto
Received >> x >> s >> f;
if (!Received){
    cout<<"Ocurrio un error en la lectura"<<endl;
}

// tambien puede preguntarse el resultado de la operacion
// de lectura de la siguiente manera
if (!(Received >> x >> s >> f)){
    cout<<"Ocurrio un error en la lectura"<<endl;
}
```

Si se desea incluir en un paquete tipos de datos definidos por el usuario, pueden sobrecargarse los operadores de inserción y extracción (<< y >>) de la forma que mostramos a continuación:

```
struct Personaje{
    float Energia;
    string NombreJugador;
    unsigned Puntos;
};

sf::Packet& operator <<(sf::Packet& Packet, const Personaje& C){
    Packet << C.Energia << C.NombreJugador << C.Puntos;
    return Packet;
}

sf::Packet& operator >>(sf::Packet& Packet, Personaje& C){
    Packet >> C.Energia >> C.NombreJugador >> C.Puntos;
    return Packet;
}
```

Las sobrecargas de los operadores insertan o extraen información de un objeto de la clase *Personaje,* realizando la inserción / extracción de sus datos primitivos. Las sobrecargas reciben dos parámetros: el paquete y el personaje, de los cuales se inserta o extrae la información. El objetivo de que las sobrecargas devuelvan el mismo paquete que reciben en la llamada es para permitir inserciones o extracciones anidadas.

Con estos operadores sobrecargados pueden extraerse o insertarse objetos de tipos definidos por el usuario como si se trataran de tipos primitivos. El siguiente código ilustra esto:

```
Personaje miPersonaje;
sf::Packet Packet;

Packet << miPersonaje;
Packet >> miPersonaje;
```

Para enviar o recibir objetos de tipo *sf::Packet,* existen sobrecargas para las funciones *Send()* y *Receive()* que vimos anteriormente:

```
// para sockets TCP
Socket.Send(Packet);
Socket.Receive(Packet);

// para sockets UDP
Socket.Send(Packet, Address, Port);
Socket.Receive(Packet, Address);
```

Como dijimos, los tipos de datos primitivos no son adecuados para enviar información a través de la red, ya el tamaño de los mismos es dependiente de la plataforma.

Lamentablemente, la funcionalidad de los paquetes facilitados por SFML no puede solucionar este problema de manera automática. Para evitar inconvenientes de este tipo, lo aconsejable es utilizar los tipos primitivos de tamaño fijo de SFML: *sf::Int8, sf::Uint16, sf::Int32,* etc. Utilizando estos tipos de datos, tendremos la garantía de que los mismos serán siempre del mismo tamaño sin importar el compilador o la plataforma.

Es importante aclarar que los paquetes de SFML tienen su propio orden de bits (*endianess*) y estructura, por lo cual no pueden ser usados si la contraparte de la comunicación no los utiliza. Para intercambiar información con aplicaciones que no empleen paquetes de SFML, deben usarse arreglos de bytes de la forma en que se mostró en los primeros ejemplos.

2.4. Sockets asíncronos

La mayoría de las funciones de sockets que hemos utilizado hasta ahora son síncronas, es decir, cuando son llamadas, bloquean la ejecución del programa hasta que la operación de entrada / salida se haya completado. Esto implica que no es posible recibir información de más de un socket a la vez.

Una solución para este problema es utilizar threads o hilos de ejecución y colocar las llamadas a funciones de sockets en otro hilo de ejecución, distinto del principal. De esta manera, el programa podrá seguir corriendo mientras los sockets realizan sus funciones. No obstante, la implementación de threads no es sencilla, ya que deben tener una buena sincronización, son difíciles de depurar y pueden ralentizar la ejecución del programa si hay muchos hilos corriendo al mismo tiempo.

En cambio, SFML nos provee de una forma más sencilla de lograr esta funcionalidad, utilizando selectores *(sf::Selector).* Estos objetos nos permiten multiplexar un conjunto de sockets sin necesidad de crear un nuevo hilo de ejecución.

Los selectores funcionan de forma similar a los arreglos: se le puede agregar o quitar sockets y nos avisará cuándo alguno de los sockets bajo su control esté listo para recibir datos.

SFML nos proporciona dos clases de selectores: *sf::SelectorTCP,* para sockets TCP, y *sf::SelectorUDP,* para sockets UDP. Ambas clases tienen exactamente las mismas funciones y sólo difieren en el tipo de socket que gestionan.

El siguiente código ejemplifica la utilización de un selector con sockets TCP:

```
sf::SelectorTCP Selector;
sf::SocketTCP Listener;
Listener.Listen(4567);
Selector.Add(Listener);
...
Selector.Add(Cliente1);
Selector.Add(Cliente2);

...
```

```
while(true){
    ...
    unsigned int nSockets = Selector.Wait();

    for (unsigned int i = 0; i < nSockets; i++){
        sf::SocketTCP Socket = Selector.GetSocketReady(i);
        // hacer algo con el socket
    }
    ...
}
```

En este ejemplo se crea un socket, que estará a la espera de conexiones entrantes, y dos sockets para la comunicación con clientes. Estos sockets son agregados al selector. Luego, se ingresa en un bucle infinito en el cual se le preguntará al selector cuáles sockets están listos para recibir datos.

La función *Wait()* esperará hasta que al menos un socket esté preparado o se haya agotado un tiempo de espera que puede ser pasado como parámetro. Una vez finalizada la llamada, la función devolverá la cantidad de sockets en el selector que están listos para ser leídos. Mediante la función *GetSocketReady(),* se le podrá solicitar al selector el i-ésimo socket preparado para recibir datos, a fin de realizar con él las operaciones necesarias.

En el siguiente parágrafo mostraremos cómo utilizar este mecanismo para implementar un servidor capaz de manejar múltiples clientes.

2.5. Manejo de múltiples clientes

Mostraremos, ahora, cómo aplicar las funcionalidades de SFML –que vimos anteriormente– en el desarrollo de un pequeño programa que, a pesar de ser muy sencillo, nos dará una idea de los problemas que surgen en el desarrollo de videojuegos reales.

La aplicación consiste en un sprite que puede ser arrastrado por la ventana utilizando el ratón. Sin embargo, la aplicación se conectará con un servidor y mostrará no sólo el sprite propio, sino también los sprites de las otras instancias de la aplicación que se encuentren conectadas con el servidor. El usuario podrá visualizar en su ventana todos los sprites, pero solo podrá mover el correspondiente a su cliente propio.

Por otra parte, la aplicación servidor que desarrollaremos será un servidor dedicado. Esto significa que no tendrá interface gráfica, sino que se ocupará específicamente de gestionar la interconexión entre los distintos clientes y distribuir la información entre ellos.

Antes de empezar a programar, debemos diseñar cómo se llevará a cabo la comunicación entre las partes involucradas. La arquitectura propuesta es la siguiente: cuando un cliente se mueve, enviará al servidor un paquete con los datos de su nueva posición y color (los sprites se diferenciarán entre sí por su color).

Al recibir datos de cambio de posición de al menos uno de los clientes, el servidor armará un paquete con las posiciones y los colores de todos los clientes y se lo reenviará a cada uno de ellos. Si bien este protocolo está lejos de ser óptimo, ya que reenvía mucha información redundante, nos servirá para ejemplificar el funcionamiento de un servidor que maneja múltiples clientes.

En el código siguiente se muestra el tipo de dato que utilizamos para modelar un cliente y las sobrecargas de los operadores << y >> para insertarlos en un objeto de tipo *sf::Packet*:

```
struct Cliente: public sf::Sprite{
    unsigned clientId;
};

sf::Packet &operator<<(sf::Packet &p, Cliente &c){
    sf::Vector2f pos=c.GetPosition();
    sf::Uint16 x=pos.x, y=pos.y;
    p<<x<<y;
    sf::Color color=c.GetColor();
    sf::Uint8 r=color.r, g=color.g, b=color.b;
    p<<r<<g<<b;
    sf::Uint8 id=c.clientId;
    p<<id;
    return p;
}

sf::Packet &operator>>(sf::Packet &p, Cliente &c){
    sf::Uint16 x, y;
    p>>x>>y;
    c.SetPosition(x, y);
    sf::Uint8 r, g, b;
    p>>r>>g>>b;
    c.SetColor(sf::Color(r,g,b));
    sf::Uint8 var;
    p>>var;
    c.clientId=var;
    return p;
}
```

Nuestro cliente es simplemente un sprite, pero que agrega un número de identificación. Este número será utilizado por el servidor para saber de qué cliente proviene el paquete.

En un sprite tenemos todos los datos que necesitamos para representar la entidad que queremos mover por la pantalla: posición, color, imagen, etc. Sin embargo, a la hora de enviar los datos para comunicarnos con el servidor, sólo necesitamos una parte de toda esa información: la posición, el color y el número de identificación del cliente que posee el sprite. No será necesario enviar la imagen porque todos los clientes utilizarán la misma y tendrán su propia copia. Las sobrecargas de los operadores de inserción y extracción permiten insertar estos datos primitivos de un *Cliente* en un paquete de SFML.

El código que sigue corresponde al servidor dedicado:

```
int main(int argc, char *argv[]) {
    unsigned nSocketsReady, i, nClientes;
    bool debeActualizar;

    vector<Cliente> clientes;
    vector<sf::SocketTCP> socketsClientes;

    sf::SocketTCP listener;
    listener.Listen(1712);

    sf::SelectorTCP selector;
    selector.Add(listener);


    while(true){
        debeActualizar=false;
        // pregunta si hay sockets listos para leer
        nSocketsReady=selector.Wait(0.0001);
        // recorre los sockets listos para leer
        for(i=0; i<nSocketsReady; i++){
            sf::SocketTCP s;
            s=selector.GetSocketReady(i);
            // si el socket listo es el listener, entonces
```

```
            // tenemos una solicitud de un cliente nuevo
            if(s==listener){
                // aceptamos al nuevo cliente en un nuevo socket
                // enviandole los datos de un cliente solamente
                // con su numero de id
                sf::SocketTCP t;
                s.Accept(t);
                Cliente c;
                c.clientId=clientes.size();
                sf::Packet enviar;
                enviar<<c;
                t.Send(enviar);

                clientes.push_back(c);
                socketsClientes.push_back(t);
                selector.Add(t);
                cout<<"Ciente conectado. Id: "<<c.clientId<<endl;
            }else{
                // si recibimos datos de un cliente, actualizamos
                Cliente c;
                sf::Packet recibir;
                s.Receive(recibir);
                recibir>>c;
                clientes[c.clientId]=c;
                debeActualizar=true;
            }
        }

        // arma un paquete con el estado de todos los clientes
        if(debeActualizar){
            sf::Packet estado;
            nClientes=socketsClientes.size();
            for(unsigned j=0; j<nClientes; j++){
                estado<<clientes[j];
            }

            // envia el paquete a cada cliente
            unsigned nSocketsClientes=socketsClientes.size();
            vector<sf::SocketTCP>::iterator p=socketsClientes.begin();
            while(p!=socketsClientes.end()){
                p->Send(estado);
                p++;
            }
        }
    }

    selector.Clear();

    return 0;
}
```

El servidor guarda dos vectores con los datos (posición, color e id) de todos los clientes y otro con los sockets de dichos clientes para enviarles la información a cada uno.

El socket *listener* estará a la espera de conexiones entrantes en el puerto 1712 y será agregado al selector. El servidor entrará, entonces, en un bucle en el cual pedirá al selector los sockets que estén listos para ser leídos, utilizando la función *Wait(),* la cual esperará a los sockets por un lapso máximo de 0.0001 segundos.

Luego, en caso de que exista al menos algún socket listo, se recibirán datos del mismo. Si el socket listo es *listener,* se generará un nuevo socket y se aceptará la conexión entrante con el mismo. Además, se creará un objeto cliente y se le asignará el próximo id disponible. Este objeto será enviado al cliente para que el mismo conozca su id y lo utilice en los próximos envíos de datos al servidor. El socket utilizado para aceptar la conexión será agregado al selector para conocer cuándo el cliente envía datos.

Si, en cambio, el socket listo para leer corresponde a un cliente, se leerán del socket los datos enviados por el mismo y se actualizará el vector que contiene todos los clientes. También se encenderá una bandera para indicar que deben reenviarse los

datos actualizados a los clientes. Para esto, se construirá un paquete con los datos de todos ellos y se enviará por cada socket, a excepción de *listener.*

Debajo es posible observar un fragmento de código del cliente:

```
...
// si estamos arrastrando
if(draggingBicho){
    // movemos el bicho
    if(mouse_x!=mouse_x_anterior && mouse_y!=mouse_y_anterior){
        miBicho.SetPosition(mouse_x, mouse_y);
        // enviamos la nueva posicion al servidor
        mouse_x_anterior=mouse_x;
        mouse_y_anterior=mouse_y;
        paquete.Clear();
        paquete<<miBicho;
        socket.Send(paquete);
    }
}

// recibimos datos del servidor
nSocketsReady=selector.Wait(0.0001);
// solo tenemos un socket en el selector
if(nSocketsReady>0){
    clientes.clear();
    paquete.Clear();
    Cliente temp;
   selector.GetSocketReady(0).Receive(paquete);

    while(paquete>>temp){
        temp.SetImage(i);
        temp.SetCenter(i.GetWidth()/2,i.GetHeight()/2);
        clientes.push_back(temp);
    }
}
...
```

Para evitar el envío innecesario de información, el cliente sólo manda datos al servidor cuando está arrastrando y hay, efectivamente, un cambio en la posición.

El cliente posee un único socket: el que utiliza para comunicarse con el servidor. Este socket también se encuentra dentro de un selector para poder consultarlo de la misma forma que en el servidor, sin bloquear la ejecución del programa. Cuando llega la información, ésta contiene los datos de todos los clientes conectados (incluso los propios). Estos datos son volcados en un vector que se utilizará luego para dibujar los clientes.

Como mencionamos anteriormente, la forma de diálogo entre servidor y cliente posee muchos aspectos que deberían optimizarse. Por ejemplo, el servidor reenvía los datos de todos los clientes aunque sólo uno de ellos haya cambiado. Además, el color se envía siempre a pesar de no cambiar nunca.

Como puede apreciarse, el desarrollo de aplicaciones en red requiere mucha atención a la hora de tomar decisiones sobre el diseño de la comunicación, ya que se trata de un proceso muy complejo. Algunos factores a tener en cuenta se refieren a qué recursos estarán en el servidor y cuáles en los clientes; cuándo es necesario enviar un dato y cuándo no; qué formato usar para los mensajes, o si la comunicación se realiza con un elemento central (servidor) o si es distribuida (los clientes se comunican con otros clientes).

Bibliografía

Tanenbaum, Andrew S. *Redes de computadoras, Pearson Educación,* 2003, 4° ed.

Tutoriales SFML [en línea] http://www.sfml-dev.org/tutorials/1.6/

Wikipedia [en línea]

http://en.wikipedia.org/wiki/TCP/IP_model

http://en.wikipedia.org/wiki/Internet_socket