COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación de servicios y procesos			CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD COMPETENCIA					

Tema 3 – Networking

Conceptos básicos: lectura de repaso

```
- Please, give me your address
- maxpower@springfield.org
- Not this, the other...
- 80.58.0.33
- No, the private, please
- 192.168.1.11
- I mean the physic one!
- 00:1d:7d:e5:91:a0
- OMG, home address!
- 127.0.0.1
```

Si no entiendes lo anterior hay muuucho que repasar…

Para poder realizar programación en redes, lo primero es tener claro las distintas arquitecturas y conceptos usados en el mundo del networking tal cual se vio en la asignatura de primero de Sistemas Informáticos. En cualquier caso haremos un rápido repaso a los conceptos esenciales.

En una red tenemos una serie de equipos (**hosts**) interconectados entre sí formando a su vez subredes y a la vez enrutados e interconectados mediante distintos dispositivos como **routers** y **switches.**

Estos equipos se comunican entre sí enviándose **paquetes**, que son ristras de bytes con la información que se quiere transmitir además de información de control que indica a la red qué hacer con el paquete (por ejemplo su destino).

Por tanto en dicho paquete va la información propiamente dicha y bytes adicionales que introducen las distintas **capas de red** para que la comunicación sea efectiva.

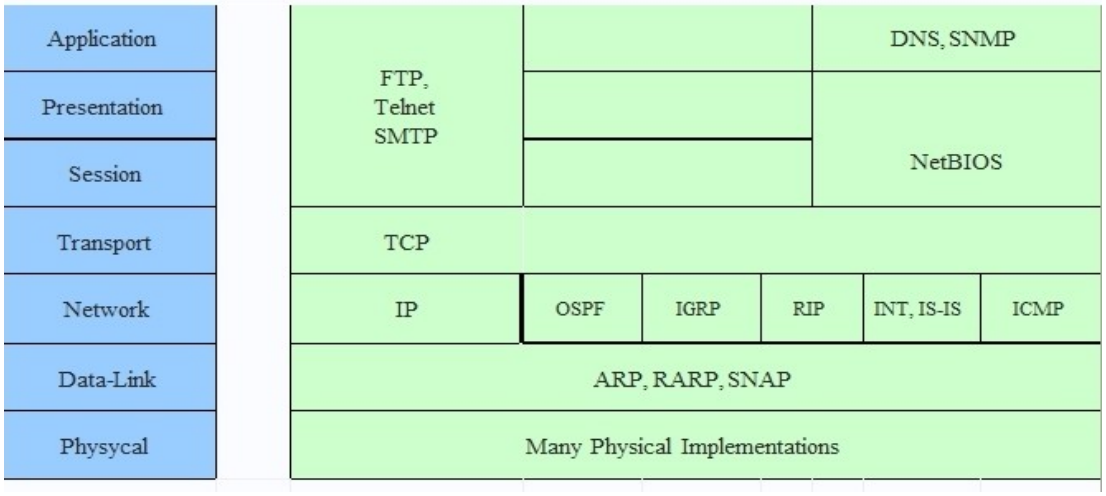
Protocolos de comunicación

Un **protocolo de comunicación** es un convenio mediante el cual dos máquinas pueden iniciar y llevar a cabo una comunicación. Para ello el protocolo define una secuencia determinada de paquetes, los tipo de paquetes, su tamaño, que comandos de control incluyen, qué significan los bytes de control y como están construidos, etc...

Existen distintos protocolos para optimizar distintos tipos de comunicaciones. Por ejemplo http para intercambio de páginas web, ftp para intercambio de archivos, etc...

Una de las tareas principales cuando se programa una aplicación con conexión es establecer un protocolo de comunicación: o se coge uno existente o hay que crear uno nuevo.

TCP/IP es un conjunto de protocolos. Los principales son **TCP** (Transmission Control Protocol), **UDP** (User Datagram Protocol) e **IP** (Internet Protocol) que son la base de la comunicación en internet hoy en día.



Los protocolos se pueden montar por capas, así protocolos como FTP o HTTP están una capa por encima de TCP y esta a su vez por encima de IP.

IP es la **capa de red**, encargada de crear una conexión virtual entre dos equipos enviando paquetes entre ellos ya que cada paquete IP contiene la dirección destino.

Por encima de la capa de red tenemos la **capa de transporte** donde conviven los protocolos TCP y UDP. Esta capa se encarga de refinar el direccionamiento, ya que IP sólo conecta entre equipos. TCP y UDP se encarga de conectar entre aplicaciones en una dirección dada por IP usando un puerto determinado.

TCP es un protocolo orientado a conexión lo que permite que sea fiable ante errores de forma que si la información no llega a su destino el remitente al final lo sabría. Para que haya una comunicación TCP los dos equipos deben llegar a un acuerdo de comunicación para que se establezca la conexión.

UDP es más ligero pero una vez enviado un paquete se olvida de él y no se confirma su llegada. Es no orientado a conexión. Son las aplicaciones que usan este protocolo las encargadas de lidiar con todo lo que UDP no hace: confirmaciones, reenvíos, …

El uso de UDP puede parecer más limitado, pero sin embargo gracias a su ligereza está ampliamente usado hoy en día en streaming de audio y video en tiempo real donde si llega un paquete mal, no tiene sentido parar la secuencia a la espera de que llegue de nuevo el paquete. Otros usos son tareas de control en red.

Una dirección completa apunta a un equipo y una aplicación de ese equipo. El equipo lo identificamos por su **dirección IP** (4 octetos en IPv4). La aplicación por el **puerto** que es un número entre 0 y 65535. Por debajo de 1024 los puertos están reservados para servicios estándar como FTP, POP o HTTP. Un ejemplo de dirección completa sería 192.168.1.10:3240.

Como las direcciones son complicadas de recordar para los humanos a los equipos se les puede dar un **nombre de dominio** dentro de una red. De esta forma un ordenador puede comunicarse con otro usando dicho nombre "humanizado" a través de un servidor **DNS** (Domain Name System) que se encarga de traducir su nombre a una IP. En local puede haber un emulador de este servicio mediante el archivo *hosts*.

Arquitectura Cliente/Servidor

Un **cliente** es una aplicación en un equipo que necesita algo y por tanto debe iniciar la comunicación con un **servidor** que es otra aplicación en el mismo u otro equipo que puede darle lo que necesita el cliente. El servidor no inicia la comunicación pero está permanentemente a la escucha por si algún cliente le requiere.

Si un cliente inicia una petición, el servidor debe crear un hilo para atender a dicho cliente y seguir a la escucha de otros posibles clientes en el puerto de escucha (el servidor pasa el cliente a otro puerto para poder comunicarse, de esta forma hay un puerto de escucha y luego uno de comunicación para cada cliente). **Cada par IP_cliente:puerto_cliente+IP_servidor:puerto_servidor establece un canal de conexión único.**

El cliente por tanto debe conocer la IP del servidor y el puerto de la aplicación que le va a dar el servicio. Es por esto que hay una serie de puertos estándar (ftp 21, http 80, ...) para facilitar la comunicación, pero un servidor se puede colocar en el puerto que desee.

En este enlace se pueden ver la numeración de **puertos estándar**:

http://es.wikipedia.org/wiki/Anexo:Números_de_puerto

Puedes ver los puertos abiertos en tu equipo y el proceso asociado (su PID)

mediante el siguiente comando:

```
netstat -aob
```

a: muestra todas las conexiones y puertos.

o: Indica el PID del proceso asociado a la conexión

b: Indica el nombre del proceso (requiere permisos de administrador, puedes no usarla).

Clases genéricas para networking en .Net

Obtener información de equipos

Muchas veces es necesario conocer las direcciones IPs de un equipo propio o remoto a partir de un nombre del equipo. O también puede ser necesario lo inverso: a partir de una IP obtener nombre de equipo u otras IPs asociadas.

Para obtener en .Net esta información se pueden usar métodos de las siguientes **clases**:

Dns: Clase con métodos para la resolución de nombres.

IPAddress: Clase que encapsula y gestiona direcciones IP.

IPHostEntry: Clase que mantiene información de un equipo conectado en red.

Veamos mediante un ejemplo sencillo y autoexplicativo estas clases en funcionamiento. Es necesario tener los espacios de nombres **System.Net** y **System.Net.Sockets** incluidos.

```
using System;
using System.Net;
using System.Net.Sockets;

namespace Networking
{
  class Program
  {
    static void ShowNetInformation(string name)
    {

      IPHostEntry hostInfo;

      //Tratamos de resolver el DNS
      hostInfo = Dns.GetHostEntry(name);

      // Mostramos el nombre del equipo
      Console.WriteLine("Name: {0}", hostInfo.HostName);
```

```
      //Lista de IPs del equipo
      Console.WriteLine("IP list: ");
      foreach (IPAddress ip in hostInfo.AddressList)
      {
        //Para ver sólo las direcciones IPv4 se compara con
AddressFamily.InterNetwork
        //Para IPv6 se usaría AddressFamily.InterNetworkV6
        if (ip.AddressFamily == AddressFamily.InterNetwork)
        {
          Console.WriteLine("\t{0,16}", ip);
        }
      }
      Console.WriteLine("\n");

    }

    // Sobrecarga con parámetro IPAddress
    static void ShowNetInformation(IPAddress ipAddress)
    {
        //Llama a la otra sobrecarga obteniendo previamente el nombre del host
      IPHostEntry hostInfo = Dns.GetHostEntry(ipAddress);
      ShowNetInformation(hostInfo.HostName);
    }


    static void Main(string[] args)
    {
      // Obtenemos el nombre del equipo local y lo mostramos
      String localHost = Dns.GetHostName();
      Console.WriteLine("Localhost name: {0} \n", localHost);

      //Mostramos información del equipo local y de uno remoto
      ShowNetInformation(localHost);
      ShowNetInformation("www.google.es");

      // ShowNetInformation(new IPAddress(new byte[] { 82, 98, 160, 124 }));
      ShowNetInformation(IPAddress.Parse("82.98.160.124"));

      Console.ReadKey();
    }
  }
}
```

La clase **IPAddress** es una de las que dispone .Net para manejar direcciones IP de forma encapsulada. Como se ve en el ejemplo anterior dispone de propiedades y métodos interesantes. A continuación vemos algunos de ellos:

Parse: Devuelve a partir de la forma típica de escribir una IP como string (4 octetos separados por puntos) un IPAddress válido. Se usa más a menudo que el constructor que solo se puede crear a partir de un tipo long o de un array de 4 bytes.

Otras formas de inicializar a Ips típicas en IPv4 del sistema son:

IPaddress.Loopback : normalmente 127.0.0.1

IPAddress.Broadcast*:* dirección de broadcast para la red local.

IPAddress.Any: Referido a una IP cualquiera que escucha en todos los interfaces de red. La usaremos habitualmente en servidores.

Equals: Compara dos direcciones IP contenidas en los objetos correspondientes.

AddressFamily: Para indicar si es IPv4 (InternetWork) o IPv6 (InternetWorkV6).

TryParse: Devuelve true o false según sea una IP válida. Hay que tener cuidado al usarla pues realiza ciertas interpretaciones al pasarle ciertos valores que en principio no son IP's válidas. Para tener más información revisa el apartado Remarks del siguiente enlace:

https://learn.microsoft.com/en-us/dotnet/api/system.net.ipaddress.tryparse?view=net-7.0

La clase IPEndPoint

Como se comentó previamente, para realizar una conexión no llega una dirección IP si no que se necesita un par IP+puerto (de hecho dos pares, local y remoto). Para ello .Net dispone de la clase **IPEndPoint** que encapsula estos dos datos en un único objeto. El constructor más habitual que usaremos es:

```
IPEndPoint(IPAddress direccionIP, int puerto)
```

Si más tarde necesitamos estos datos u otros sobre la conexión los podemos obtener en diversas propiedades. También se pueden modificar los datos de la conexión como se ve en el siguiente ejemplo.

```
IPAddress ip = IPAddress.Loopback;
IPEndPoint ie = new IPEndPoint(ip, 1200);

Console.WriteLine("IPEndPoint: {0}", ie.ToString());
Console.WriteLine("AddressFamily: {0}", ie.AddressFamily);
Console.WriteLine("Address: {0}, Puerto: {1}", ie.Address, ie.Port);
Console.WriteLine("Ports range: {0}-{1}", IPEndPoint.MinPort,
IPEndPoint.MaxPort);

ie.Port = 80;
ie.Address = IPAddress.Parse("80.1.12.128");
Console.WriteLine("New End Point: {0}", ie.ToString());
```

Sockets

Un **socket** es una abstracción que nos permite unificar en un único elemento de programación (objeto o estructura, dependiendo del lenguaje de programación usado) un canal de comunicación con la red. Se puede ver (y se usa de forma similar) a un objeto o variable tipo archivo en un lenguaje de programación.

De esta forma cuando se inicia un socket hay que indicar datos como la dirección IP, el puerto, protocolos usados, etc. ya que es la vía de comunicación de una aplicación con la red.

Los sockets más usados son los Stream sockets que usan protocolo TCP (y son los que más vamos a usar) y los Datagram sockets que usan el protocolo UPD.

Una vez establecida la comunicación el **socket** identifica un par único

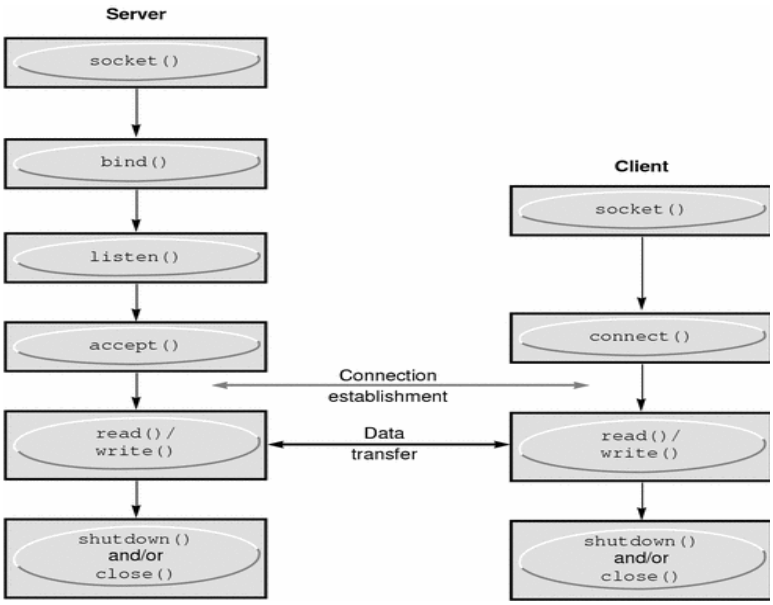
ip:puerto local + ip:puerto remoto.

Programación con sockets

Como ya se comentó, un socket es una vía de conexión a la red al cual hay que indicarle varios parámetros según la forma y el destino al que nos conectaremos.

Se puede ver como una variable o clase tipo fichero (file descriptor) un tanto especial ya que en UNIX, padre del tratamiento de sockets, todos los dispositivos se tratan como archivos.

Si estamos usando un protocolo **TCP orientado a conexión**, la forma de trabajo en una aplicación cliente servidor es la siguiente:



En el **servidor** se **abre el socket**, que es la vía de comunicación, y se indica con qué protocolo se va a trabajar (en este caso sería TCP).

Mediante **bind**() se **enlaza el socket a un puerto** y una (o varias) IP de escucha.

A continuación el servidor está preparado y se pone a la **escucha con listen**(). Esta se queda a la espera de que un cliente quiera realizar una conexión.

Este es el denominado socket de escucha del servidor que siempre está en el mismo puerto y es conocido por los clientes que deseen conectarse a él.

En el **cliente** se ha **creado un socket** y a continuación lanza un ***connect***() que realiza una petición a la IP y al puerto donde está el servidor.

El **servidor** si lo considera apropiado (permisos, recursos, etc...) acepta la petición mediante **accept**() y la **conexión está establecida**.

En este punto si se desea que el servidor pueda atender a muchos clientes se podría crear un hilo por cada uno de los clientes tratados.

En este caso, el cliente puede usar siempre la misma dirección IP+puerto, aunque por cada una de las conexiones el sistema operativo la tratará con un socket

distinto ya que consta de un par ip+puerto local e ip+puerto remoto y un protocolo determinado.

De esta forma puede usarse múltiples veces el puerto de escucha que está asociado a distintos sockets, que son los verdaderos canales de comunicación.

Si se quiere profundizar en esto se pueden ver las respuestas en este enlace. Sobre todo son interesantes la segunda y tercera:

https://stackoverflow.com/questions/3329641/how-do-multiple-clients-connect-simultaneously-to-one-port-say-80-on-a-server

La comunicación se establece con comandos de envío write() o send() y de recepción de datos read() o receive() en el hilo de atención al cliente.

Usaremos funciones de más alto nivel que **send** y **receive** por comodidad: usaremos **StreamReaders** y **StreamWriters** que ya los conocemos de Archivos.

Finalmente se cierra el socket cuando la comunicación ha finalizado.

En el caso de usar un protocolo UDP no orientado a conexión, no existe paso de escucha ni aceptación en el servidor, simplemente se envían y reciben los mensajes. Por el momento no lo veremos ya que la mayoría de nuestras conexiones serán TCP.

Creación de una aplicación cliente-servidor

En este caso realizaremos como ejemplo explicativo un servidor de Echo, es decir, un servidor que está a la espera de recibir un mensaje de texto de algún cliente y cuando lo recibe se lo devuelve al cliente.

La clase Socket

Será la clase que representa la conexión. Permite muchas combinaciones de parámetros pero nosotros no profundizaremos demasiado en este primer acercamiento.

Nos quedaremos sólo con los elementos más estándar e imprescindibles para realizar una conexión TCP en una red IPv4 y a medida que necesitemos otras posibilidades las iremos viendo. Usaremos el siguiente constructor:

```
Socket(AddresFamily familia, SocketType tipo, ProtocolType protocolo)
```

familia: Enumerado con el que especificamos el tipo de direccionamiento usado. No se limita a IPv4 o IPv6 si no que admite otros estándares como AppleTalk, Netbios, OSI, etc...). Usaremos **InternetWork** para IPv4. Digamos que describe la red con la que vamos a trabajar.

tipo: El tipo de socket que vamos a usar. En principio usaremos sólo **Stream** que sirve para conexiones bidireccionales basadas en conexión (como TCP, pero no es la única). Pero admite otros tipos como sin conexión para UDP (Dgram) o incluso bajar a la capa de transporte (RAW) para usar otros protocolos como ICMP (El que usa el ping).

protocolo: El protocolo de comunicación usado . Usaremos normalmente **Tcp.**

Por lo tanto el trabajo habitual con sockets para nosotros en general comenzará con este constructor:

```
Socket s = new Socket(AddressFamily.InterNetwork,
                      SocketType.Stream,
                      ProtocolType.Tcp);
```

Trabajaremos en una red TCP/IP orientada a conexión y protocolo TCP.

Una vez finalizado el uso del socket, al igual que sucede con archivos hay que **cerrarlos** mediante un **Close** para no ocupar recursos. También se puede usar **using**.

Las excepciones del tipo **SocketException** son muy genéricas, pero vienen acompañadas de la propiedad **ErrorCode** que nos da más información yendo a las tablas de MSDN:

http://msdn.microsoft.com/en-us/library/windows/desktop/ms740668(v=vs.85).aspx

Comprobación de puerto de escucha libre/ocupado

Hay varias formas de saber si un puerto está ocupado. Una de ellas es intentar conectarnos al mismo. Es decir, si puedo conectarme es que hay algún servidor escuchando y por tanto está en uso. Otra es intentar ponerme a la escucha en dicho puerto. Si puedo, está libre, si no, ocupado por otra aplicación que lo está usando.

Veamos un ejemplo de esta última:

```
int port = 135;
IPEndPoint ie = new IPEndPoint(IPAddress.Any, port);

//Creacion del Socket
using (Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp))
{
      try
      {
          //Enlace de socket al puerto (y en cualquier interfaz de red)
          //Salta excepción si el puerto está ocupado
          s.Bind(ie);
          Console.WriteLine($"Port {port} free");
      }
      catch (SocketException e) when (e.ErrorCode ==
(int)SocketError.AddressAlreadyInUse)
      {
          // ó 10048
          Console.WriteLine($"Port {port} in use");
      }
}
```

Realización de la aplicación: Lado del Servidor

Empezaremos en este apartado a realizar la parte del servidor paso a paso estableciendo el protocolo de comunicación.

En principio **lo probaremos con Telnet** y posteriormente haremos un cliente que cumpla el protocolo del servidor.

Servidor: Conexión

Los pasos son parte de los vistos en el gráfico de conexión mediante protocolo TCP:

1. Creación del socket
2. Apertura del puerto
3. Escucha
4. Aceptación de la petición del cliente en cuanto exista.

```
IPEndPoint ie = new IPEndPoint (IPAddress.Any, 31416);

//Creacion del Socket
Socket s = new Socket (AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

//Enlace de socket al puerto (y en cualquier interfaz de red)
//Salta excepción si el puerto está ocupado
s.Bind (ie);

//Esperando una conexión y estableciendo cola de clientes pendientes
s.Listen (10);

//Esperamos y aceptamos la conexion del cliente (socket bloqueante)
Socket sClient = s.Accept();

//Obtenemos la info del cliente
//El casting es necesario ya que RemoteEndPoint es del tipo EndPoint
//mas genérico
IPEndPoint ieClient = (IPEndPoint) sClient.RemoteEndPoint;

Console.WriteLine ("Client connected:{0} at port {1}", ieClient.Address,
ieClient.Port);

sClient.Close (); // Se puede usar using con Socket y nos ahorramos los
```

```
close.
    s.Close ();
```

Notas sobre el código anterior:

- En la llamada a la función **Listen(int backlog)**, el backlog es el número de conexiones que puede recibir. Por ejemplo si es 3 y tiene 3 clientes, acepta el primero y pone a los otros dos en espera, pero si llega un cuarto este es rechazado.

- Se debería comprobar si el puerto que vamos a usar está en uso mediante un control de excepciones.

- Se trabaja con dos sockets, el del servidor que es el primero que se crea (*s: socket de escucha*) y el del cliente en cuanto llega uno (*sCliente*). Más adelante veremos cómo tratar varios clientes en varios hilos.

- Puedes probar el servidor anterior ejecutando desde consola:

  ```
  telnet 127.0.0.1 31416
  ```

  En versiones de Windows Vista en adelante no viene instalado el cliente telnet. Para instalarlo ve a panel de control → Programas y características → Activar y desactivas características de Windows.

  Ahí puedes seleccionar cliente Telnet.

  Otra opción es usar otro programa tipo telnet como putty o el Ubuntu integrado que se puede descargar de la MS Store y activando el subsistema para Linux (en el mismo sitio donde activas el telnet).

Si ejecutas un netstat antes del cierre de los sockets en el servidor puedes ver algo similar a esto:

```
C:\>netstat -ao

Conexiones activas
  Proto  Dirección local          Dirección remota        Estado
  ...
  TCP    0.0.0.0:31416            A22-PC32:0                LISTENING
  ...
  TCP    127.0.0.1:31416        A22-PC32:50318         ESTABLISHED
```

La primera linea es el socket de escucha y la segunda el de cliente.

Servidor: transmisión

Una vez realizada la conexión, el servidor le envía al cliente un mensaje de bienvenida (estamos estableciendo un protocolo de comunicación) que el cliente tendrá que recibir.

Esto se puede hacer a bajo nivel enviando byte a byte como se puede ver en el ejemplo del Apéndice I. Pero nosotros por simplicidad vamos a usar una capa intermedia de Streams que ya los conocemos de usarlos con archivos. Es decir, los solucionaremos usando **StreamReader** y **StreamWriter**.

Además esta es una solución existente **en** otros lenguajes como **Java**, no siendo exclusivo de .Net y por tanto no perderemos generalidad:

https://poliformat.upv.es/access/content/group/OCW_6069_2008/T2.-Comunicación%20I%3A%20del%20C_S%20al%20modelo%20de%20objetos/Tecnolog%C3%ADa%20JAVA/Java-sockets.pdf

Estas dos clases sabemos como usarlas para manejar archivos de texto. Pero de forma genérica se le puede pasar un stream de bytes cualesquiera. De esta forma usaremos un stream de red (**NetworkStream**) para inicializar estas conocidas clases en lugar de un archivo del disco duro o un file stream y luego usaremos WriteLine y ReadLine para enviar y recibir mensajes de texto respectivamente.

La clase NetworkStream no es más que una interfaz que traduce los sockets a streams para que puedan ser usados por manejadores de streams como StreamReader y StreamWriter.

Sustituyendo a los dos Close() de los socket escribe el siguiente código:

```
 //Preparando End Point del servidor
//Creación del Stream de Red. Nuevamente puede hacerse con using.
NetworkStream ns = new NetworkStream (sClient);


//StreamReader y StreamWriter aceptan un Stream
//como parámetro en el constructor
StreamReader sr = new StreamReader (ns);
StreamWriter sw = new StreamWriter (ns);
```

```
string welcome = "Welcome to The Echo-Logic, Odd, Desiderable, " +
  "Incredible, and Javaless Echo Server (T.E.L.O.D.I.J.E Server)";



//El envío por red se convierte en un WriteLine
sw.WriteLine (welcome);

//Con flush se fuerza el envío de los datos sin esperar al cierre
sw.Flush ();

//El código del protocolo debe ir antes de esta línea
//Siempre cerramos los Streams y sockets si no lo hemos hecho con using.
sw.Close ();
sr.Close ();
ns.Close ();
sClient.Close ();
s.Close ();
```

Primero se crea un NetworkStream que haga de puente entre el socket y los StreamWriter y StreamReader que ya conocemos.

Para enviar un mensaje por red el servidor realiza una operación de escritura sobre el stream correspondiente.

Como normalmente las escrituras sobre Streams son en memoria, como ocurría con archivos, y los datos no se vuelcan al destino final hasta que se cierran dichos streams, es necesario hacer un **flush** para realizar el envío de datos cada vez que se hace un Write.

Servidor: intercambio de datos

Finalmente el intercambio de mensajes en este caso se transforma en un bucle en el que el servidor espera a recibir un mensaje con un ReadLine y cuando lo recibe lo devuelve con un WriteLine sobre los streams correspondientes.

```
 string msg = "";

while (msg != null) {
   try {
    // Leemos el mensaje del cliente
    msg = sr.ReadLine ();
   // Si el cliente manda mensaje (si manda null es que ha desconectado)
    // se le envía de vuelta
   if (msg != null) {
     Console.WriteLine (msg);
     sw.WriteLine (msg);
     sw.Flush ();
   }
   }
  // Si se cierra el cliente, salta excepción
  // Al siguiente readline que será null
  catch (IOException e) {
   msg = null;
   }

}
Console.WriteLine ("Client disconnected.\nConnection closed");
```

La forma en que se cierra la conexión en este caso es exclusivamente que el cliente se cierre abruptamente (cierra la consola del telnet para probarlo con el icono X). En este caso, si el cliente no empezó a escribir nada el ReadLine devuelve null y a la siguiente vuelta del bucle, termina.

Puedes probar a quitar el try/catch y en principio parece que funciona igual, sin embargo, puede darse el caso que el cliente empieza a escribir algo pero le da al icono X (cierre) antes de que se envíe ese mensaje (retorno de carro). En ese caso el mensaje no es null y al hacer el luego el *WriteLine* saltaría excepción, de ahí la necesidad del try.

De esta forma tenemos nuestro servidor finalizado.

Es importante indicar que **estamos usando sockets con bloqueo** de forma que hay que alternar entre los WriteLine y ReadLine de cliente y servidor para que no entren ninguno de los programas (ni cliente ni servidor) en deadlock. Existe la posibilidad de usar sockets de forma asíncrona de forma que el programa no se bloquee. Existen varias posibilidades; nosotros usaremos hilos para solventarlo más adelante.

Si quieres poner un tiempo de espera limitado para el cliente, se puede establecer un timeout. Por ejemplo si añades la siguiente línea:

```
sr.BaseStream.ReadTimeout = 2000;  //milliseconds, so 2 seconds
```

tras la creación del StreamReader (por ejemplo antes del mensaje de bienvenida), se provoca que si el cliente tarda más de 2 segundos en enviar algún mensaje entonces se produce la desconexión (también salta excepción).

El código completo (usando using)

```
IPEndPoint ie = new IPEndPoint(IPAddress.Any, 31416);

using (Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp))
{
    s.Bind(ie);
    s.Listen(10);
    Console.WriteLine($"Server listening at port:{ie.Port}");
    Socket sClient = s.Accept();

    IPEndPoint ieClient = (IPEndPoint)sClient.RemoteEndPoint;
    Console.WriteLine("Client connected:{0} at port {1}", ieClient.Address,
ieClient.Port);

    using (NetworkStream ns = new NetworkStream(sClient))
    using (StreamReader sr = new StreamReader(ns))
    using (StreamWriter sw = new StreamWriter(ns))
    {
        string welcome = "Welcome to The Echo-Logic, Odd, Desiderable,"+
        "Incredible, and Javaless Echo Server(T.E.L.O.D.I.J.E. Server)";
        sw.WriteLine(welcome);
        sw.Flush();
        string msg = "";
        while (msg != null)
        {
            try
            {
                msg = sr.ReadLine();
                if (msg != null)
                {
                    Console.WriteLine(msg);
                    sw.WriteLine(msg);
                    sw.Flush();
                }
            }
            catch (IOException e)
            {
                msg = null;
            }
        }
        Console.WriteLine("Client disconnected.\nConnection closed");
     }
     sClient.Close();  // Este no se abre con using, pues lo devuelve el accept.
}
```

Realización de la aplicación: Lado del Cliente

Nota: Si quieres ejecutar dos proyectos a la vez en la misma solución del Visual Studio, vete a propiedades de la solución (botón derecho sobre la solución) y ahí elige proyectos múltiples y en ambos indica iniciar.

En otro proyecto (o solución) crearemos el cliente de nuestra aplicación.

Lo que es fundamental es que haya correspondencia en el protocolo entre el cliente y el servidor. Es decir, no se crea el cliente en función del servidor ni viceversa. Lo que debe hacerse es primero definir bien el protocolo y luego hacer tanto el cliente como el servidor en función de dicho protocolo.

A diferencia del servidor el cliente es activo no pasivo. Es decir, es el que inicia la conexión por eso lo primero que encontramos es el connect.

```
            const string IP_SERVER = "127.0.0.1";
            string msg;
            string userMsg;

            // Indicamos servidor al que nos queremos conectar y puerto
            IPEndPoint ie = new IPEndPoint(IPAddress.Parse(IP_SERVER), 31416);
            Console.WriteLine("Starting client. Press a key to init
connection");

            Console.ReadKey();
            Socket server = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);
            try
            {
                // El cliente inicia la conexión haciendo petición con Connect
                server.Connect(ie);
            }
            catch (SocketException e)
            {
                Console.WriteLine("Error connection: {0}\nError code: {1}({2})",
                    e.Message, (SocketError)e.ErrorCode, e.ErrorCode);
                Console.ReadKey();
                return;
            }
```

```
            // Si la conexión se ha establecido se crean los Streams
            // y se inicial la comunicación siguiendo el protocolo
            // establecido en el servidor
            using (NetworkStream ns = new NetworkStream(server))
            using (StreamReader sr = new StreamReader(ns))
            using (StreamWriter sw = new StreamWriter(ns))
            {
             // Leemos mensaje de bienvenida ya que es lo primero que envía el
servidor
                msg = sr.ReadLine();
                Console.WriteLine(msg);
                while (true)
                {
                    // Lo siguiente es pedir un mensaje al usuario
                    userMsg = Console.ReadLine();

                    //Enviamos el mensaje de usuario al servidor
                    // que que el servidor está esperando que le envíen algo
                    sw.WriteLine(userMsg);
                    sw.Flush();

                    //Recibimos el mensaje del servidor
                    msg = sr.ReadLine();
                    Console.WriteLine(msg);
                }
            }
            Console.WriteLine("Ending connection");

            //Indicamos fin de transmisión.
            server.Close();
```

Ahora ejecuta primero el servidor y a continuación el cliente. Si los tienes en la misma solución usa **ReadKey**() para marcar el inicio correcto.

Otra posibilidad es que lo probéis contra el servidor de algún compañero.

El código está autoexplicado y no es difícil de entender si se ha entendido el funcionamiento del servidor, el protocolo y la arquitectura cliente/servidor.

Por supuesto cualquier otra clase que trabaje con streams (BinaryReader, BinaryWriter, etc...) es capaz de trabajar también con NetworkStream por lo que se pueden simplificar muchas tareas.

Sockets e Hilos

Existen diversas formas de hacer que un socket no bloquee el programa en ejecución a la espera de realizar alguna acción. En este apartado veremos una de ellas que consiste en lanzar las operaciones que pueden bloquear el programa en un hilo aparte.

De esta forma al no estar el programa bloqueado, se pueden atender a varios clientes simultáneamente, cosa que no hemos hecho por el momento.

En el servidor el socket es común. Sólo hay un puerto de espera. Una vez aceptada la conexión con el cliente es cuando se lanza el hilo.

El algoritmo de esto se presenta a continuación.

El **bucle principal** de tratamiento de clientes:

```
creación del socket
bind
listen
while (servidor activo)
     acepta conexión
     lanza hilo con ese cliente
fin while
```

En el **hilo de cada cliente** el resto de las tareas:

```
envío de mensaje de bienvenida
bucle de envío y recepción de datos (según protocolo)
cierre del socket
```

Cuando el servidor acepta la conexión, obtiene un socket de cliente que le devuelve el método Accept. Este socket es el objeto parámetro que hay que pasar al thread.

Tomando como referencia el cliente y el servidor vistos anteriormente en el apartado de Streams, el cliente que usaremos será el mismo. Es el servidor el que cambiaremos.

Por un lado tenemos el programa principal y por otro la función que será la que se

ejecute como thread.

El programa principal queda de la siguiente forma según vimos en el esquema anterior:

```
static void Main(string[] args)
{
    IPEndPoint ie = new IPEndPoint(IPAddress.Any, 31416);
    Socket s = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);
    s.Bind(ie);
    s.Listen(10);
    Console.WriteLine("Server waiting at port {0}", ie.Port);
    while (true)
    {
        Socket cliente = s.Accept();
        Thread hilo = new Thread(hiloCliente);
        hilo.Start(cliente);
    }
}
```

Tras la aceptación de un cliente y la obtención del socket de conexión con el mismo, se lanza el hilo que se encargará del intercambio de información con ese cliente.

En este caso se llama a la función hiloCliente que vemos más abajo y se le pasa como parámetro el socket. Se podría hacer una clase en la que esté dicha función más el socket y crear un nuevo objeto para luego lanzar como hilo y sin parámetro la función de dicho objeto.

Aquí se ha planteado un bucle infinito, pero en un caso real debe hacerse dependiente de alguna bandera para poder finalizar el servidor desde algún otro hilo.

La función hiloCliente se ve entera en la siguiente página.

```
static void hiloCliente(object socket)
{
    string mensaje;
    Socket cliente = (Socket)socket;
    IPEndPoint ieCliente = (IPEndPoint)cliente.RemoteEndPoint;
    Console.WriteLine("Connected with client {0} at port {1}",
                    ieCliente.Address, ieCliente.Port);

    using (NetworkStream ns = new NetworkStream(cliente))
    using (StreamReader sr = new StreamReader(ns))
    using (StreamWriter sw = new StreamWriter(ns))
    {
        string welcome = "Wellcome to this great Server";
        sw.WriteLine(welcome);
        sw.Flush();

        while (true)
        {
            try
            {
                mensaje = sr.ReadLine();
                sw.WriteLine(mensaje);
                sw.Flush();

                //El  mensaje es null al cerrar
                if (mensaje != null)
                {
                    Console.WriteLine("{0} says: {1}",
                        ieCliente.Address, mensaje);
                }
            }
            catch (IOException)
            {
                //Salta al acceder al socket
                //y no estar permitido
                break;
            }
        }
        Console.WriteLine("Finished connection with {0}:{1}",
                ieCliente.Address, ieCliente.Port);
    }
    cliente.Close();
 }
```

Se puede pensar que al haber varios clientes conectados al mismo puerto del servidor puede haber algún conflicto, pero no es así, recordemos que cada conexión se identifica por los dos pares de IP+puerto tanto del cliente como del servidor (Además del protocolo).

Existen otras formas de tratar el bloqueo de los sockets pero no los veremos a menos que nos haga falta más adelante.

Otras clases para trabajo en Red

Hemos visto en este tema la base de las conexiones en red mediante sockets. Es cierto que cualquier tipo de intercambio de datos o protocolo se puede hacer con las herramientas vistas, sin embargo existen en .Net multitud de clases que nos facilitan en gran medida el trabajo en red dependiendo de los protocolos con los que queramos trabajar. Veamos algunos ejemplos:

TcpClient, TcpListener, UdpClient: Clases que simplifican la conexión por sockets. No son tan potentes, pero la inicialización es mucho más sencilla.

WebClient: Clase para comunicaciones HTTP.

System.Web.Mail: Espacio de nombres para la gestión de correo electrónico.

Remoting: posibilidad de tener programas distribuidos en varios equipos y que desde unos se puedan ejecutar funciones sitas en otros.

System.Security.Cryptography: Espacio de nombres para la gestión de seguridad.

NetworkInterface: Clase que nos da información sobre los distintos interfaces de red y algunos estadísticos de los mismos.

HttpListener: Clase para incluir en una aplicación la posibilidad de aceptar y responder peticiones HTTP.

En general si vas a realizar algún tipo de intercambio de información por red comprueba primero la existencia de clases para dicho intercambio lo cual te facilitará la labor en gran medida.

Ejercicios

Todos los clientes de los siguientes ejercicios deben tener la posibilidad de configurar IP del servidor y Puerto.

Ejercicio 1

a) Realiza un **servidor** de fecha y hora. Aceptará los siguientes comandos desde cualquier cliente (Protocolo):

time: Envía hora, minutos y segundos,

date: Envía día, mes y año.

all: Envía tanto la fecha como la hora.

close password: Junto con el comando close se debe verificar que viene una contraseña. Si esta es correcta el servidor ha de finalizar y se lo indica al cliente. Si no devuelve un mensaje de error al cliente (Debe diferenciarse el error de contraseña no válida o que no se haya enviado la contraseña).

La contraseña estará guardad en un archivo de texto en %PROGRAMDATA%.

En cuanto se envíe el dato pedido, se cerrará la conexión (No hay bucle de atención al cliente en el hilo del cliente).

b) Haz un cliente simple en modo gráfico con cuatro botones (uno por comando) y un TextBox para meter una contraseña y una Label para mostrar el resultado y probarlo.

El cliente no se mantiene conectado, es decir, cada vez que se pulsa un botón, se conecta, se envían y reciben datos y se desconecta. Fíjate que todos los botones van a hacer casi lo mismo por lo que debes minimizar el código repetido.

También tendrá la posibilidad de indicarle la IP y puerto de conexión (aunque venga con unos predefinidos) en un formulario de diálogo.

Ejercicio 2

Realiza un cliente gráfico para el servidor de turnos de clase. Debe tener posibilidades de configurar IP y puerto del servidor, así como el usuario de conexión y aceptar los comandos ADD y LIST mediante sendos botones.

La idea es similar al cliente creado en el ejercicio 1. Ambos botones realizan conexión, mandan y reciben mensajes y finalmente se desconecta.

La última configuración usada (ip, puerto, usuario) se guardará en un archivo para la siguiente vez que se ejecute el programa.

Ejercicio 3

Haz un chatroom. El servidor permite conectarse a cuantos clientes deseen y lo que recibe de un cliente lo reenvía al resto de los clientes indicando la IP.

Para realizarlo debes crear una colección de clientes conectados que sea común a todos los hilos. La colección, para este caso, llega con que sea de StreamWriters, pero si deseas hacerla de otro tipo (Socket, Clase definida por ti,…) puedes hacerlo.

Cuando un cliente se conecte debe indicar un nombre de usuario de forma que cuando envíe un mensaje al resto de los usuarios, se indicará en el formato **usuario@IP** quién lo envía antes del mensaje en sí mismo.

Además el servidor informará al resto de clientes de quién se conecta y quién se desconecta cuando esto suceda.

Un usuario se podrá desconectar cuando lo desee escribiendo el comando **#exit**. También podrá ver la lista de usuarios conectados escribiendo **#lista**. Por supuesto debe ser robusto ante salidas abruptas de un cliente. Como cliente usa directamente telnet.

(Opcional) Realiza un cliente gráfico. Ten en cuenta que necesitas trabajar con hilos distintos para poder enviar y recibir en cualquier momento desde el

cliente.

También de forma opcional puedes ampliar el servidor de forma que admita varias salas de chat, usuarios registrados, etc...

Ejercicio 4

Se desea realizar un servidor de turnos similar al que usamos en clase. Crea en un nuevo proyecto de consola una clase **ShiftServer** en la cual estarán definidos los siguientes miembros:

- Dos listas de usuarios: por un lado un vector de usuarios posibles denominado **users** (vector de strings) y una colección de espera (**waitQueue**) donde se añaden los usuarios que están en lista de espera (colección de strings).

- Función **ReadNames** la cual lee de un archivo de texto, cuya ruta está indicada en el parámetro, nombres de usuarios separados por punto y coma (;). Rellena el vector users con dicha lista de nombres.

- Función **ReadPin** la cual lee de un archivo binario, cuya ruta está indicada en el parámetro, un integer y lo devuelve. Si hay un problema con el archivo o el pin de 4 dígitos devuelve -1.

- Habrá una función **Init** la cual inicia el servidor multihilo con las especificaciones siguientes:

  - El puerto de espera será el 31416. Si este puerto estuviera ocupado busca el primero libre a partir del 1024. Si llegara al último válido la aplicación finaliza. Muestra en pantalla el puerto de conexión.

  - Luego cargará la lista de usuarios válidos mediante la función ReadNames del archivo usuarios.txt que se encuentra en el directorio indicado por %userprofile%.

  - Una vez que el cliente se conecta se lanza el hilo que controla a dicho cliente. Lo primero es que este recibe un mensaje de bienvenida y se le pide nombre de usuario. Comprueba si dicho nombre está en la

lista o si es la palabra **admin**. Si no cumple lo anterior simplemente se desconecta informando de "Usuario desconocido".

- Si es **admin** se le pide un pin. El servidor obtiene el pin mediante la función ReadPin pasándole un archivo denominado pin.bin que se encuentra también en %userprofile%. Si hay algún problema al leer el archivo la contraseña será 1234. Si la contraseña no es correcta, lo desconecta directamente.

- Una vez conectado, para un usuario normal los comandos posibles son:

  - **list**: Envía al cliente el listado de alumnos en espera (waitQueue).

  - **add**: Añade el usuario actual al final de la lista. Debes concatenarle al nombre la fecha y hora de conexión. Envía al cliente un mensaje de OK.

  En cuanto el usuario mete uno de estos dos comandos, ya se finaliza la conexión (no se queda en bucle a la espera de otro comando).

- En el caso de ser el **usuario admin**, además de las anteriores dispone de los siguiente comandos:

  - **del *pos***: borra el usuario indicado por la posición pos. Si hay alguna incorrección envía el mensaje **delete error** al cliente.

  - **chpin *pin***: guarda pin en el archivo binario pin.bin en caso de que sea un pin válido (integer de al menos 4 cifras). Indica al cliente si ha sido guardado o si ha habido un error.

  - **exit**: Sale del servidor. Al revés que un cliente "normal", el administrador no es desconectado tras meter un comando si no que solo finaliza la conexión mediante este comando (o shutdown).

  - **shutdown**: Finaliza el servidor de forma natural.

El Main estará en otra clase y simplemente creará un objeto de la clase Servidor y ejecutará su función principal.

Para la parte de usuario normal debería funcionar con este servidor el cliente creado en el ejercicio 2.

Ejercicio 5

Ahorcado en red. Se trata de realizar un servidor de palabras y récords y un cliente de juego.

Servidor

Dispone de acceso a un archivo de palabras (archivo de texto con palabras separadas por comas). Cuando lea el archivo pasa las palabras a mayúsculas.

También mantiene un archivo de récords, que será un archivo binario donde se guarda una lista con 3 objetos tipo Record (clase sencilla que almacena nombre de 3 caracteres y cantidad de segundos)

Se dispone del siguiente protocolo de comandos que puede recibir el servidor:

getword: El servidor envía una palabra aleatoria al cliente de la lista que tiene.

sendword palabra: El cliente le envía una palabra nueva al servidor. Este la guarda en el archivo y la añade a la colección de palabras actual. El servidor envía un OK o ERROR si se ha hecho todo bien o hubo un problema respectivamente.

getrecords: El servidor le envía la lista de récords del archivo al cliente.

sendrecord record: El cliente le envía un nuevo récord al servidor. Si el tiempo es menor que alguno que los que tiene almacenado lo guarda y, si fuera necesario porque ya hay tres, deshecha el peor. Si no fuera menor que ninguno lo deshecha. El servidor responde con un ACCEPT o REJECT dependiendo si ha guardado o no el récord.

closeserver clave: cierra el servidor si se dispone de la clave adecuada.

Cliente

El cliente al ejecutarse pedirá una palabra al servidor. El cliente cuando acaba la partida envía el tiempo en segundos que le ha llevado adivinar la palabra y un nombre de 3 caracteres.

El resto del juego será como de costumbre: en el cliente, tras comenzar un juego nuevo pedirá una palabra al servidor y usará **varias labels** para colocar los subrayados (o letras cuando la acierte). No se puede hacer en una única label. Aparecerá un fragmento de dibujo del ahorcado usando **el componente a medida del tema de DI**.

Acuérdate de medir el tiempo (hazlo mediante un Timer) y pedir el nombre para el récord solo si va a entrar en la tabla de récords. También debe permitir ver la tabla de récords que se le pedirá al servidor.

Habrá también un formulario específico de envío de palabras. Se permitirán dos posibilidades, o enviar una única palabra o enviar un archivo de palabras. El archivo serán palabras separadas por comas.

Crea un interfaz adecuado con todas las opciones en un menú.

(Opcional): Busca la manera de disponer en el servidor de un archivo con una lista de contraseñas válidas. Estás contraseñas estarán encriptadas (de estilo de archivo passwd de linux). También el envío de la contraseña desde el cliente se encriptará. Busca alguna librería o método para realizar esta parte.

Ejercicio 6

Juego número más alto en red. Al servidor se conectan al menos dos clientes (puede haber más) y les da un número aleatorio entre 1 y 20. Cuando todos los clientes estén conectados (Hazlo por tiempo. Que los clientes vean el tiempo que queda para empezar) el servidor comprueba quién es el ganador e informa de ello a los clientes. Debe indicar a los que pierde el número ganador además del número que sacaron.

Ejercicios Opcionales

Op 1

a) Realiza un servidor de archivos con las siguientes características:

- Existe un directorio por defecto que será donde estén los posibles archivos a transferir.

- Admite varios clientes a la vez.

- Debe haber cierta sincronización entre hilos de forma que si un cliente está descargándose un archivo otros no pueden empezar a descargarlo hasta que el primero acabe.

- Los comandos que habrá son:

get nombrearchivo: Puede ser cualquier tipo de archivo. Busca una forma de codificarlos como testo para enviarlos. Cada byte un carácter, hexadecimal como string en bloques, etc...

list: Manda un alista de los archivos disponibles

upload: El servidor se queda esperando a que se le envíe un archivo (usa el mismo sistema de codificación que en get).

Exit: El cliente se desconecta

close: cierra el servidor una vez que terminen las tareas pendientes.

b) Haz un cliente en modo gráfico que permita trabajar con el servidor previo

Op 2

Haz un programa de chat punto a punto. El programa será tanto cliente como servidor. El formulario tendrá un botón de Escucha el cual activa el modo

servidor y se prepara para aceptar un cliente (puede aceptar varios en múltiples ventanas o tabs). Otro botón de conexión mediante el cual se conectará a otro servidor.

Si se establece la conexión entre ambos, se podrán enviar mensajes de texto entre las aplicaciones como si de un chat se tratase.

Op 3

Realiza un servidor Web simple. Lo único que hará es servir páginas html/htm si las hay y dar el mensaje de error si no existen. Observa en: http://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol el Ejemplo de diálogo. Solo tiene que responder a peticiones GET.

Op 4

Realiza un programa que realice un ping por programación a una IP que se le indique. Puedes hacerlo en modo consola para que funcione como comando. Debes informarte sobre el protocolo ICMP y los sockets en modo RAW para poder hacerlo.

Apéndice I: Cliente-servidor con send/receive

En este apéndice se verá la realización de un programa con su cliente y su servidor pero trabajando a bajo nivel, con envío de bytes usando send/receive en lugar de Streams como se vio durante el tema.

Servidor: partimos del mensaje de bienvenida, lo anterior es igual.

Una vez realizada la conexión, el servidor le envía al cliente un mensaje de bienvenida (estamos estableciendo un protocolo de comunicación) que el cliente tendrá que recibir.

El código del servidor sería:

```
string bienvenida = "Bienvenido al servidor de Echo, tu di que yo repito";
byte[] datos = Encoding.UTF8.GetBytes(bienvenida);
sCliente.Send(datos);
```

La transmisión de datos a través de sockets se hace como un stream de bytes, concretamente como un vector de bytes. Por tanto convertimos lo que queramos mandar a un vector de bytes mediante la clase Encoding (se encuentra en el namespace *System.Text*).

Esta clase tiene varias posibilidades. Usamos UTF8 para poder enviar distintos juegos de caracteres.

Al método Send se le pasa el vector de bytes que lo envía al cliente, este último debe tener preparado un Receive para obtener estos datos.

Nuevamente puedes probarlo con el telnet.

Cliente: recepción

A continuación el código del cliente que cumple el protocolo anterior y recibe el mensaje de bienvenida:

```
byte[] datos=new byte[512];
int nDatos = servidor.Receive(datos);
string mensaje = Encoding.UTF8.GetString(datos, 0, nDatos);
Console.WriteLine(mensaje);
```

Se debe establecer el tamaño máximo de mensaje que se va a enviar (esto debe ser parte del protocolo).

El método Receive devuelve el número de bytes recibidos y en el parámetro coloca los datos recibidos.

Cuando el cliente remoto cierra una sesión el método Receive devuelve 0. Si no devuelve la cantidad de bytes recibidos.

Encodig.UTF8.GetString vuelve a convertir a cadena los bytes que previamente convertimos a UTF8.

Servidor: intercambio de datos

Finalmente mediante un bucle hacemos funcionar el servidor de Echo:

```
while(true)
{
        datos = new byte[512];
        int nDatos = sCliente.Receive(datos);
        if (nDatos == 0)
                break;
        string mensaje=Encoding.UTF8.GetString(datos, 0, nDatos);
        Console.WriteLine(mensaje);
        sCliente.Send(datos, nDatos, SocketFlags.None);
}
```

Puedes probarlo mediante telnet. Verás que lo que se escribe desde la consola lo devuelve repetido. Puedes tener problemas con la codificación: prueba ASCII en lugar de UTF8.

En cada vuelta del bucle se inicializa el vector de datos. Luego se espera que el cliente envíe datos. Si hay datos recibidos (Receive devuelve una cantidad mayor que 0) se transforman en string y se devuelven al cliente mediante Send. Previamente se muestran por pantalla.

Es necesario usar una sobrecarga distinta del Send en la cual se indica la cantidad de datos real que se quiere enviar porque si no se envían los 512 bytes y sólo nos

interesa la parte del mensaje. Del enumerado SocketFlag no es necesario preocuparse por el momento, simplemente se indica que no hace falta ningún flag (no es necesario cambiar el comportamiento del socket) mediante None.

Cliente: Intercambio de datos

En este caso el bucle de intercambio para cumplir el protocolo es:

```
while (true)
{
    mensaje = Console.ReadLine();
    if (mensaje == "salir")
        break;
    servidor.Send(Encoding.UTF8.GetBytes(mensaje));
    datos = new byte[1024];
    nDatos = servidor.Receive(datos);
    mensaje = Encoding.UTF8.GetString(datos, 0, nDatos);
    Console.WriteLine(mensaje);
}
//Finalizamos la conexión.
//Esto provoca que Receive en el servidor devuelve 0
Console.WriteLine("Terminando la conexión...");
servidor.Shutdown(SocketShutdown.Both);
```

En este bucle se pide una cadena al usuario. Con la palabra *salir* se finaliza la conexión. En caso contrario se le envía dicho mensaje al servidor y se espera que nos devuelva dicho mensaje si todo funciona correctamente. Finalmente se muestra.

El método Shutdown es necesario para indicar que se finalizan las transmisiones y es el que provoca que el Receive en el servidor devuelva 0 para detectar dicho fin de transmisión.

Otras consideraciones:

- Si se usa un buffer de recepción muy pequeño, en sucesivos envíos nos irá llegando los datos anteriores que no se han metido en el buffer (realmente se encuentran en el buffer TCP interno del sistema). Prueba metiendo en el cliente un buffer (vector *datos*) de tamaño 10 por ejemplo.

- Nada asegura que un Receive vaya a recibir datos de un único Send, puede darse el caso de que si se dan varios Send seguidos, al hacer un Receive se

cojan los datos de dos o más Sends si ya están en el buffer del sistema (en el siguiente apartado vemos el problema y una solución).

- Como solución a los puntos anteriores hay varias posibilidades: mensajes siempre del mismo tamaño, enviar el tamaño con el mensaje (se ve más abajo) o usar una marca para separar mensajes. También podemos usarlo cómo veremos más adelante con Streams.

- Si se usa la propiedad ReceiveTimeout de un socket a cierta cantidad de ms, una vez ejecutado el Receive si pasan los ms indicados salta una excepción.

- Estamos usando sockets con bloqueo de forma que hay que alternar entre los send y receive para que no entren ninguno de los programas (ni cliente ni servidor) en deadlock. Existe la posibilidad de usar sockets de forma asíncrona de forma que el programa no se bloquee. Existen varias posibilidades; nosotros usaremos hilos para solventarlo.

- No veremos UDP en este tema, pero las diferencia básicas son la creación del socket y el envío y recepción de los datos. El socket se crearía de la siguiente forma:

  ```
  Socket s = Socket(AddressFamily.InterNetwork, SocketType.Dgram,
                    ProtocolType.Udp);
  ```

  Y para enviar y recibir se usarían respectivamente los métodos *SendTo*() y *ReceiveFrom*().

Cómo se comentó anteriormente, puede haber un problema a la hora de enviar y recibir mensajes ya que un Receive no tiene por que se equivalente a un Send, si no que recoge lo que tiene disponible en el buffer TCP del sistema.

Veamos un ejemplo. Usamos el siguiente cliente:

```
#if LOCAL
            const string IP_SERVIDOR = "127.0.0.1";
#else
            const string IP_SERVIDOR = "10.211.55.2";
#endif

            IPEndPoint ie=new IPEndPoint(IPAddress.Parse(IP_SERVIDOR), 31416);
            Console.WriteLine(
                   "Iniciando cliente. Pulsa una tecla para iniciar la conexión");
            Console.ReadKey();
            Socket servidor = new Socket(AddressFamily.InterNetwork,
```

```
                                SocketType.Stream, ProtocolType.Tcp);
try
{
    servidor.Connect(ie);
}
catch (SocketException e)
{
    Console.WriteLine(
       "Error de conexión: {0}\nCódigo de error: {1}({2})" ,
       e.Message, (SocketError)e.ErrorCode, e.ErrorCode);
    Console.ReadKey();
    return;
}

byte[] datos=new byte[1024];
int nDatos = servidor.Receive(datos);
string mensaje = Encoding.UTF8.GetString(datos, 0, nDatos);
Console.WriteLine(mensaje);

string[] dias = new string[] { "Lunes", "Martes", "Miércoles", "Jueves",
                               "Viernes", "Sábado", "Domingo" };
foreach (string dia in dias)
{
    servidor.Send(Encoding.UTF8.GetBytes(dia));
    //System.Threading.Thread.Sleep(100);

}

Console.WriteLine("Terminando la conexión...");
servidor.Shutdown(SocketShutdown.Both);
servidor.Close();
Console.ReadKey();
```

Básicamente es el mismo cliente usado anteriormente. El cambio estriba a partir del mensaje de bienvenida. En este caso enviamos una serie de mensajes todos seguidos.

El servidor (receptor) de los mensajes puede ser el mismo. Puedes comentar la línea

```
sCliente.Send(datos, nDatos, SocketFlags.None);
```

que provoca una excepción.

Observarás que al recibir la información pone todos (o algunos, dependiendo del estado de la red) los mensajes en la misma línea. Si quitas el comentario a la linea del Sleep sin embargo se recibe cada mensaje en una línea que es como debe ser.

Esto es porque llegan varios mensajes al buffer TCP del sistema antes de ejecutar el Receive, y cuando este se ejecuta se leen todos a la vez.

Para solucionar esto existen varias posibilidades, algunas ya comentadas:

- Mandar mensajes del mismo tamaño
- Mandar como integer antes del mensaje el tamaño en bytes del mismo
- Usar un carácter especial como marca divisoria de mensajes
- Usar Streams similares a los de archivos aplicados a la red.

A continuación se puede ver la solución mediante el envío del tamaño del mensaje. Es una solución simple a partir de lo que ya conocemos de sockets por lo que no debería tener mayor dificultad su comprensión.

Envío de mensajes con su tamaño

Esta solución se implementa con una función especialmente creada para enviar y otra para recibir que ejecutan de forma más completa el Send y el Receive respectivamente teniendo en cuenta el problema visto anteriormente.

Otro problema que tiene en cuenta es que el método Send no tiene por que enviar todos los datos de un mensaje. De hecho este método devuelve la cantidad de datos que ha enviado de forma que podemos saber los que quedan pendientes por enviar.

```
public static int envioMensaje(Socket s, byte[] mensaje)
{
    int enviados;
    byte[] tam = BitConverter.GetBytes(mensaje.Length);
    int total = 0;
    int pendientes = mensaje.Length;

    //Enviamos el tamaño del mensaje
    enviados = s.Send(tam);
```

```
            //Y luego los datos
            while (total < mensaje.Length)
            {
                //Si se envía el mensaje en la primera vuelta, todo OK.
                //Si no en sucesivas vueltas se envía el resto del mensaje que no se
envió
                enviados = s.Send(mensaje, total, pendientes, SocketFlags.None);
                total += enviados;
                pendientes -= enviados;
            }
            return total;

        }
        public static byte[] recibeMensaje(Socket s)
        {
            int total = 0;
            int recv;
            byte[] tam = new byte[4];

            //Recibimos el tamaño del mensaje
            recv = s.Receive(tam, 0, 4, 0);

            //Finalizamos si Receive nos devuelve 0
            if (recv == 0)
                return Encoding.UTF8.GetBytes("salir");

            int tamInt = BitConverter.ToInt32(tam, 0);
            int pendientes = tamInt;
            byte[] mensaje = new byte[tamInt];

            //Recibimos datos parando cuando se haya completado el tamaño
            while (total < tamInt)
            {
                recv = s.Receive(mensaje, total, pendientes, 0);
                total += recv;
                pendientes -= recv;
            }
            return mensaje;
        }
```

Parte del cliente que cambia (Las funciones anteriores se supone que están en el Servidor, por lo que hay que añadir la referencia al Cliente para poder usarlo):

(…)

```
string mensaje = Encoding.UTF8.GetString(Servidor.Program.recibeMensaje(servidor));
Console.WriteLine(mensaje);

string[] dias = new string[] { "Lunes", "Martes", "Miércoles", "Jueves", "Viernes",
```

```
                                "Sábado", "Domingo" };
foreach (string dia in dias)
{
      Servidor.Program.envioMensaje(servidor, Encoding.UTF8.GetBytes(dia));
}
(…)
```

Parte del servidor que cambia:

```
(…)

string bienvenida = "Bienvenido al servidor\n";
byte[] datos = Encoding.UTF8.GetBytes(bienvenida);
envioMensaje(sCliente, datos);

for (; ; )
{
      string mensaje = Encoding.UTF8.GetString(recibeMensaje(sCliente));
      if (mensaje =="salir")
            break;
      Console.WriteLine(mensaje);
      System.Threading.Thread.Sleep(500);
}
(…)
```

Apéndice II: Información de los interfaces de red mediante el registro de Windows

Además de obtener información lógica sobre la red, en ocasiones es necesario disponer de más información sobre el hardware utilizado. Esto se puede hacer mediante la clase ***NetworkInterface*** pero también accediendo directamente al registro de Windows a partir del WMI (Windows Management Instrumentation) que es una forma estándar de acceder a la información del sistema y que vamos a ver aquí como ejemplo. Se puede ver como una manera de acceder al registro de Windows viéndolo como una base de datos a través de sentencias SQL.

Al ser información propia de Windows no son clases que encontremos en el framework de Mono ya que este último es independiente de la arquitectura.

A continuación un ejemplo de este segundo caso. **Para que funcione** es necesario agregar System.Management a las referencias justo debajo del proyecto en el explorador de soluciones:

```
using System;
using System.Management;

class WMICardGrab
{
    public static void Main()
    {

        ManagementObjectSearcher query = new
        ManagementObjectSearcher("SELECT * FROM Win32_NetworkAdapterConfiguration
WHERE IPEnabled = 'TRUE'");
        ManagementObjectCollection queryCollection = query.Get();
        foreach (ManagementObject mo in queryCollection)
        {
            string[] addresses = (string[])mo["IPAddress"];
            string[] subnets = (string[])mo["IPSubnet"];
            string[] defaultgateways =
                 (string[])mo["DefaultIPGateway"];
            Console.WriteLine("Network Card: {0}", mo["Description"]);
            Console.WriteLine("  MAC Address: {0}", mo["MACAddress"]);
            foreach (string ipaddress in addresses)
            {
                Console.WriteLine("  IP Address: {0}", ipaddress);
            }
            foreach (string subnet in subnets)
            {
                Console.WriteLine("  Subnet Mask: {0}", subnet);
            }
```

```
            foreach (string defaultgateway in defaultgateways)
            {
                Console.WriteLine("  Gateway: {0}", defaultgateway);
            }

        }
        Console.ReadKey();
    }
}
```

Fuentes

C# Network Programming by Richard Blum. ISBN:0782141765. Sybex © 2003 (647 pages). Este libro a pesar de ser antiguo realmente viene muy completo y con muchos ejemplos del trabajo en red teniendo en cuenta que este no ha cambiado demasiado. Gran parte de ejemplos de este documento están basados en este libro.

TCP/IP Sockets in C#: Practical Guide for Programmers. David Makofske, Michael J. Donahoo, and Kenneth L. Calvert. ©2004 by Elsevier Inc. (Morgan Kauffman)

Visual C# 2010 Recipes: A Problem-Solution Approach © 2010 by Allen Jones and Adam Freeman. Apress.

Wikipedia: http://es.wikipedia.org y http://en.wikipedia.org