PSP04.- PROGRAMACIÓN DE COMUNICACIONES EN RED

1.- INTRODUCCIÓN.

Antiguamente la programación de aplicaciones entre diferentes máquinas era difícil, compleja y fuente de muchos errores; el programador tenía que conocer detalles sobre las capas del protocolo de red, incluso sobre el hardware de la máquina. Java ha trabajado en el desarrollo de diferentes librerías con las que ha conseguido que desarrollar aplicaciones en red no sea una tarea tan compleja.

> Actualmente se dispone de clases para establecer conexiones, crear servidores, enviar y recibir datos y para el resto de operaciones utilizadas en las comunicaciones a través de redes de ordenadores. Además, el uso de hilos, que se trataron en la unidad didáctica anterior, nos va a permitir la manipulación simultánea de múltiples conexiones.

2.- CONCEPTOS BÁSICOS DE REDES.

PROTOCOLOS TCP/IP

En Internet se utiliza la pila de protocolos TCP/IP para el establecimiento y realización de conexiones, la cual está basada en un conjunto de protocolos organizados en diferentes niveles o capas.

Capa de aplicación	(HTTP, ftp, telnet,...)
Capa de transporte	(TCP, UDP)
Capa de Internet	(IP)
Capa de acceso a la red	(Ethernet, ...)

La **capa de aplicación:** en este nivel se encuentran las aplicaciones disponibles para los usuarios. Por ejemplo: FTP, SMTP, Telnet, HTTP, etc.

La **capa de transporte:** suministra a las aplicaciones servicio de comunicaciones de extremo a extremo, utilizando dos tipos de protocolos: TCP (Transmision Control Protocol) y UDP (User Datagram Protocol).

La **capa de Internet:** tiene como propósito seleccionar la mejor ruta para enviar los paquetes por la red. El protocolo principal que funciona en esta capa es el Protocolo de Internet (IP).

La **capa de acceso al medio:** es la interfaz con la red real; recibe los datagramas y los transmite al hardware de red.

Normalmente, cuando se escriben aplicaciones Java en Red se programa a nivel de aplicación. Como veremos en esta unidad didáctica es posible realizar programas a más bajo nivel utilizando el paquete java.net de la API de Java, el cual está compuesto por una serie de clases de bajo nivel y otro por clases de alto nivel.

DIRECCIONES IP Y PUERTOS

Un ordenador tiene una o varias conexiones físicas a la red. A través de esas conexiones recibe los datos dirigidos a la máquina. Las **direcciones IP** nos permiten identificar el interface de la máquina a la que queremos enviar los datos mientras que los **puertos** nos indican la aplicación a la que queremos enviar los datos. TCP y UDP utilizan los puertos para dirigir los datos a la aplicación correcta de entre todas las que se estén ejecutando en la máquina.

Por lo tanto, los datos transmitidos a través de Internet (o de una red local), contienen información de direccionamiento que identifica a la máquina y al puerto a los que van dirigidos:

- La máquina se identifica a través de una dirección IP de 32 bits.
- Los puertos se identifican por un número de 16 bits.

Los puertos utilizan 16 bits (0 a 65535) para identificarse. Estos se utilizan tanto para el protocolo TCP (orientado a conexión) como para el protocolo UDP (no orientado a conexión) y del 0 al 1023 están reservados para aplicaciones concretas.

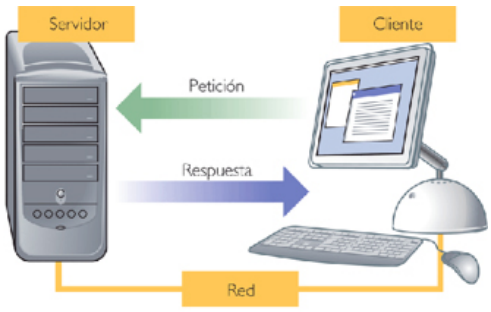
Información de detalle sobre el uso que se les da a los puertos está disponible en el fichero /etc/services (en sistemas operativos Linux) o en el fichero C:\Windows\System32\drivers\etc (en sistemas operativos Windows).

CLIENTES Y SERVIDORES

Muchas de las aplicaciones existentes hoy en día utilizan el modelo Cliente-Servidor, donde:

- El **Cliente**: programa que ejecuta el usuario y que solicita servicio al Servidor.
- El **Servidor**: ofrece servicio a múltiples Clientes.

Modelo cliente-servidor



Algunos ejemplos de aplicaciones que utilizan este modelo son: telnet, ftp, web, ssh. Un cliente ejecuta el comando telnet contra un servidor en el que está corriendo el demonio telnetd. O un cliente utiliza un programa cliente (navegador) y accede a un servidor que tiene corriendo un servidor web (Apache).

CAPA DE TRANSPORTE: TCP vs UDP

Los equipos conectados a Internet se comunican entre sí utilizando los protocolos TCP o UDP. Cuando se escriben programas Java que se comunican a través de la red, se está programando en la capa de aplicación. Normalmente, no es necesario preocuparse por las capas TCP, para lo que se pueden utilizar las clases del paquete **System.Net.** Sin embargo existen diferencias entre una y otra para decidir qué clases usar en los programas:

- **TCP: Transmission Control Protocol**
  - Protocolo orientado a conexión.
  - Provee un flujo de bytes fiable entre dos ordenadores.
  - Llegada en orden, correcta, sin perder nada. Si no fuese así se notificaría un error.
  - Protocolos del nivel de aplicación que usan TCP: telnet, http, ftp.

- **UDP: User Datagram Protocol**
  - No orientado a conexión.
  - Envía paquetes de datos (datagramas) independientes, sin garantías de que vayan a llegar al destino.
  - Permite broadcast.
  - Protocolos del nivel de aplicación que usan UDP: tftp, ping.

3.- CLASES C# PARA COMUNICACIONES EN RED.

El paquete **System.Net** proporciona las clases para la implementación de aplicaciones de red. Se pueden dividir en dos secciones:

- Una **API de bajo nivel**, que se ocupa de las siguientes abstracciones:
  - Direcciones: son los identificadores de red, como por ejemplo las direcciones IP.
  - Sockets: son los mecanismos básicos de comunicación bidireccional de datos.
  - Interfaces: describen las interfaces de red.

- Una **API de alto nivel**, que se ocupa de las siguientes abstracciones:

- URIs: representan identificadores de recursos universales.
- URLs: representan los localizadores de recursos universales.
- Conexiones: representan las conexiones al recurso apuntado por la URL.

Mientras que la API de bajo nivel se usa por las aplicaciones cliente/servidor basadas en protocolos, la API de alto nivel es utilizada en el acceso a los recursos de la red.

3.1.- CLASE SYSTEM.NET.DNS.

La Dns clase es una clase estática que recupera información acerca de un host específico de sistema de nombres de dominio (DNS) de Internet. Se devuelve la información del host de la consulta DNS en una instancia de la **IPHostEntryclase**.

```
IPHostEntry hostInfo = Dns.GetHostEntry("www.contoso.com");
```

3.2.- CLASE SYSTEM.NET.IPHOSTENTRY.

Proporciona una clase contenedora para la información de dirección de host de Internet. La clase únicamente tiene un constructor sin paramatros. Las propiedades sirven para establecer las direcciones IP, alias y el nombre del host.

Constructores	
IPHostEntry()	Inicializa una nueva instancia de la clase IPHostEntry.
Propiedades	
AddressList	Obtiene o establece una lista de direcciones IP asociadas a un host.
Aliases	Obtiene o establece una lista de alias asociados a un host.
HostName	Obtiene o establece el nombre DNS del host.

```
  using System;
using System.Net;
namespace ComunicacionPrimerosPaso
{
    class Program
    {
        static void Main(string[] args)
        {
            IPHostEntry ipHostInfo = Dns.GetHostEntry("www.google.es");
            IPAddress ipAddress = ipHostInfo.AddressList[0];
            Console.WriteLine("La direcion de google es: {0}",ipAddress.ToString());
            ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
            ipAddress = ipHostInfo.AddressList[0];
            Console.WriteLine("La direcion de esta maquina es: {0}", ipAddress);

        }
    }
}
```

3.3.- CLASE URI Y HTTPCLIENT.

Proporciona una representación de objeto de un identificador de recursos uniforme (URI) y un acceso sencillo a las partes del identificador URI.

Un URI es una representación compacta de un recurso disponible para la aplicación en intranet o Internet. La clase URI define las propiedades y los métodos para administrar las URI, incluido el análisis, la comparación y la combinación. Las propiedades de la clase son de solo lectura; para crear un objeto modificable al igual que con los string, se debe utilizar UriBuilder.

Los URIs relativos (por ejemplo, "/new/index.htm") deben expandirse con respecto a un URI base para que sean absolutos. El método MakeRelative sirve para convertir los identificadores URI absolutos en URI relativos cuando sea necesario.

La clase HttpCliente permite realizar peticiones Http de recursos especificados por URIs. A continución se muestra un ejemplo.

Ejemplo

```
  using System;
using System.Net.Http;
using System.Threading.Tasks;
namespace HttpClientYUri
{
    class Program
    {
           // HttpClient is intended to be instantiated once per application, rather than per-use. See Remarks.
            static readonly HttpClient client = new HttpClient();
            static async Task Main()
            {
                // Call asynchronous network methods in a try/catch block to handle exceptions.
                try
                {
                    HttpResponseMessage response = await client.GetAsync("http://www.contoso.com/");
                    response.EnsureSuccessStatusCode();
                    string responseBody = await response.Content.ReadAsStringAsync();
                    // Above three lines can be replaced with new helper method below
                    // string responseBody = await client.GetStringAsync(uri);

                    Console.WriteLine(responseBody);
                }
                catch (HttpRequestException e)
                {
                    Console.WriteLine("\nException Caught!");
                    Console.WriteLine("Message :{0} ", e.Message);
                }
            }

    }
}
```

4.- SOCKETS.

Un socket es un conector que **permite la comunicación bidireccional** entre dos procesos que se encuentran distribuidos. Por lo tanto, los sockets permiten una comunicación bidireccional entre dos programas que se comunican por la red.

Los procesos receptores de mensajes (los Servidores) deben de tener:

- La **dirección IP** del host en el que la aplicación está corriendo.
- Un **puerto local** a través del cual la aplicación se comunica y que identifica al proceso.

De esta forma todas las conexiones realizadas con esa dirección IP y ese puerto llegarán a un proceso receptor.

Cliente	Servidor
Cualquier puerto ->	Puerto acordado

Los procesos pueden utilizar un mismo conector (socket) tanto para enviar como para recibir mensajes. Cada conector se asocia a un protocolo concreto que puede ser TCP o UDP.

4.1.- FUNCIONAMIENTO DE UN SOCKET.

Un puerto es un punto de destino que identifica hacia qué aplicación o proceso deben dirigirse los datos. Normalmente en una aplicación cliente-servidor, el programa servidor se ejecuta en una máquina específica y tiene un socket que está unido a un número de puerto específico. El servidor queda a la espera, "escuchando" las solicitudes de conexión de los clientes por ese puerto.

El programa cliente conoce el nombre de la máquina en la que se ejecuta el servidor así como el número de puerto por el que escucha las peticiones. Para realizar una solicitud de conexión, el cliente realiza la petición a la máquina servidor a través del puerto acordado. El cliente también debe identificarse ante el servidor, por lo que durante la conexión utilizará un puerto local asignado por el sistema:

Petición de conexión

Cliente	Servidor
PCA ->	PSA

Si todo va bien, el servidor acepta la conexión. Una vez aceptada, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que por un lado debe seguir atendiendo las peticiones de conexión mediante el socket original y, por otro, debe atender las necesidades del cliente que se conectó:

Cliente	Servidor
PCB ->	PSB

En el lado del cliente, si se acepta la conexión, se crea un socket y el cliente puede utilizarlo para comunicarse con el servidor. Este socket utiliza un número de puerto diferente al usado para conectarse con el servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo respectivamente por sus respectivos sockets.

4.2.- TIPOS DE SOCKETS.

Hay dos tipos básicos de sockets en redes IP: los que utilizan el protocolo TCP (orientados a conexión) y los que utilizan el protocolo UDP (no orientados a conexión).

4.3.- SOCKETS TCP U ORIENTADOS A CONEXIÓN.

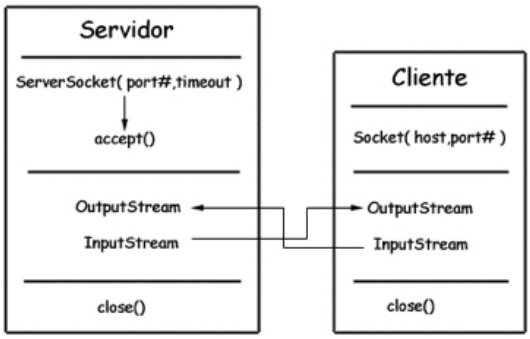
> La comunicación entre las aplicaciones se realiza por medio del protocolo TCP. Por tanto, es una conexión fiable en la que se garantiza la entrega de los paquetes de datos y el orden en que fueron enviados. TCP utiliza un esquema de acuse de recibo de los mensajes de tal forma que, si el emisor no recibe dicho acuse dentro de un tiempo determinado, vuelve a transmitir el mensaje.

La gran mayoría de aplicaciones IP utilizan sockets TCP. Ejemplos de algunos servicios son: FTP (puerto 21), Telnet (puerto 23), HTTP (puerto 80), SMTP (puerto 25).

4.3.1.- MODELO SOCKET TCP.

El **modelo de socket TCP** más simple es el siguiente:

modelo de socket TCP



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

4.4.- SOCKETS UDP O NO ORIENTADOS A CONEXIÓN.

Los sockets UPD son más simples y eficientes que los TCP, pero no está garantizada la entrega de paquetes. No es necesario establecer una "conexión" entre cliente y servidor, como en el caso de TCP. Para ello, cada vez que se envíen datagramas, el emisor debe indicar explícitamente la dirección IP y el puerto del destino de cada paquete, y el receptor debe extraer la dirección IP y el puerto del emisor del paquete.

Este tipo de conexión no es fiable, no se garantiza que un mensaje que se ha enviado llegue a su destino. No existe acuse de recibo y el orden de llegada puede ser diferente al de envío. La aplicación que haga uso de los sockets UDP deberán asegurar la llegada de los datagramas, el orden de los mismos en caso de que fuera necesario.

Un **Datagrama** es un mensaje independiente, enviado a través de una red cuya llegada, tiempo de llegada y contenido no están garantizados. El paquete datagrama está formado por los siguientes campos:

- Cadena de bytes conteniendo el mensaje.
- Longitud del mensaje.
- Dirección IP destino
- Número de puerto destino.

Los sockets UDP se usan cuando es más importante una entrega rápida que una entrega garantizada, o en los casos en que se desee enviar tan poca información que cabe en un único datagrama. Se usan en:

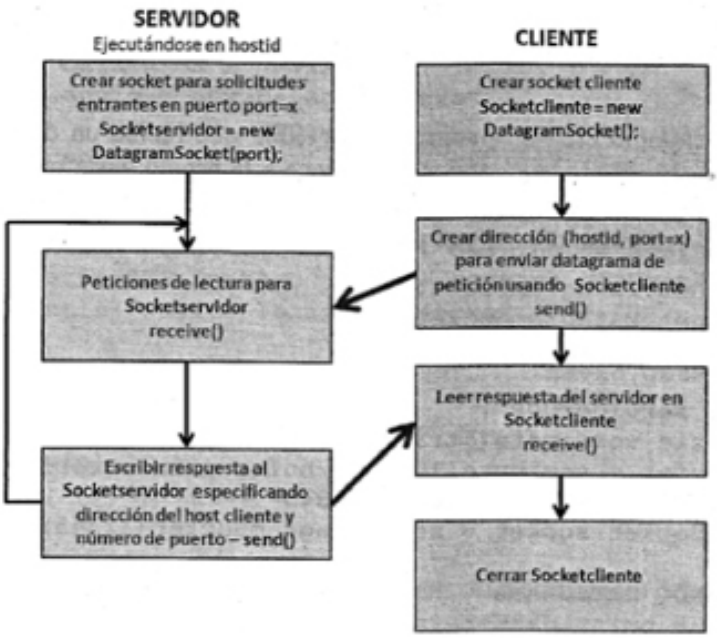
- Las transmisiones de audio y vídeo en tiempo real donde no es posible el reenvío de paquetes retrasados.
- La aplicación NFS (Network File System).
- Los protocolos DNS (Domain Name Server) o SNMP (Simple Network Management Protocol).
- El comando RCP que permite copiar archivos o directorios entre un sistema local y un sistema remoto, o entre dos sistemas remotos.
- Los protocolos de Información de Enrutamiento (RIP) y de configuración dinámica de host (DHCP).

4.4.1.- MODELO SOCKET UDP.

En los sockets UPD no se establece conexión. Los roles cliente-servidor están un poco más difusos que en el caso de los sockets TCP. Se puede considerar al servidor como el que espera un mensaje y responde, y al cliente como el que inicia la comunicación. Tanto uno como otro, si desean ponerse en contacto entre ellos, necesitan saber en qué ordenador y en qué puerto está escuchando el otro.

En la siguiente figura se muestra el flujo de la comunicación entre cliente y servidor utilizando el protocolo UDP. Ambos necesitan crear un socket DatagramSocket:

flujo de la comunicación entre cliente



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

4.5.- CLASE SYSTEM.NET.SOCKET.

La clase proporciona un amplio conjunto de métodos y propiedades para las comunicaciones de red.

La clase Socket permite realizar transferencias de datos sincrónos y asincrónas utilizando cualquiera de los protocolos de comunicación que se enumeran en ProtocolType. Dentro de la enumeración se incluyen TCP y UDP. Por lo tanto, la clase Socket permite manejar socket de ambos protocolos.

Por lo que se refiere a la posibilidad del uso del uso de método asíncrono, esto hace que no se bloquee el hilo que realiza la llamada de lectura o escritura en el socket. Gracias a que el hilo no se bloquea, en caso de que el hilo que haya realizado la llamada sea el que gestiona el interfaz de usario, el interfaz no se bloquea.

Constructores

Socket(AddressFamily, SocketType, ProtocolType)	Inicializa una instancia nueva de la clase Socket con la familia de direcciones, el tipo de socket y el protocolo que se especifiquen.
Socket(SocketInformation)	Inicializa una nueva instancia de la clase Socket utilizando el valor devuelto por DuplicateAndClose(Int32).
Socket(SocketType, ProtocolType)	Inicializa una instancia nueva de la clase Socket usando el tipo de socket y el protocolo que se especifiquen.

Propiedades

AddressFamily	Obtiene la familia de direcciones de Socket.
Available	Obtiene la cantidad de datos que se han recibido de la red y están disponibles para leer.
Blocking	Obtiene o establece un valor que indica si Socket está en modo de bloqueo.
Connected	Obtiene un valor que indica si Socket se conecta con un host remoto a partir de la última operación Send u Receive.
DontFragment	Obtiene o establece un valor de Boolean que especifica si Socket permite fragmentar los datagramas de protocolo Internet (IP).
DualMode	Obtiene o establece un valor Boolean que especifica si Socket es un socket de

	modo dual que se usa tanto para IPv4 como para IPv6.
EnableBroadcast	Obtiene o establece un valor de Boolean que especifica si Socket puede enviar o recibir paquetes de difusión.
ExclusiveAddressUse	Obtiene o establece un valor de Boolean que especifica si Socket permite sólo a un proceso el enlace a un puerto.
Handle	Obtiene el identificador del sistema operativo de Socket.
IsBound	Obtiene un valor que indica si Socket se enlaza a un puerto local concreto.
LingerState	Obtiene o establece un valor que especifica si Socket retrasará el cierre de un socket en un intento de envío de todos los datos pendientes.
LocalEndPoint	Obtiene el extremo local.
MulticastLoopback	Obtiene o establece un valor que especifica si se entregan paquetes de multidifusión saliente a la aplicación emisora.
NoDelay	Obtiene o establece un valor de Boolean que especifica si la secuencia Socket está utilizando el algoritmo de Nagle.
OSSupportsIPv4	Indica si el sistema operativo subyacente y los adaptadores de red admiten la versión 4 del protocolo de Internet (IPv4).
OSSupportsIPv6	Indica si el sistema operativo subyacente y los adaptadores de red admiten la versión 6 del protocolo Internet (IPv6).
ProtocolType	Obtiene el tipo de protocolo de Socket.
ReceiveBufferSize	Obtiene o establece un valor que especifica el tamaño del búfer de recepción de Socket.
ReceiveTimeout	Obtiene o establece un valor que especifica el tiempo tras el que una llamada Receive sincrónica pasará a tiempo de espera.
RemoteEndPoint	Obtiene el extremo remoto.
SendBufferSize	Obtiene o establece un valor que especifica el tamaño del búfer de envío de Socket.
SendTimeout	Obtiene o establece un valor que especifica el tiempo tras el que una llamada Send sincrónica pasará a tiempo de espera.
SocketType	Obtiene el tipo de Socket.
SupportsIPv4	Obtiene un valor que indica si la compatibilidad con IPv4 está disponible y habilitada en el host actual.
SupportsIPv6	Obtiene un valor que indica si el marco de trabajo admite IPv6 para ciertos miembros Dns obsoletos.
Ttl	Obtiene o establece un valor que especifica el valor de período de vida (TTL) de los paquetes de protocolo Internet (IP) enviados por Socket.
UseOnlyOverlappedIO	Especifica si el socket debe utilizar únicamente el modo de E/S superpuesta.

Métodos

Accept()	Crea un nuevo objeto Socket para una conexión recién establecida.
AcceptAsync(SocketAsyncEventArgs)	Comienza una operación asincrónica para aceptar un intento de conexión entrante.
BeginAccept(AsyncCallback, Object)	Comienza una operación asincrónica para aceptar un intento de conexión entrante.
BeginAccept(Int32, AsyncCallback, Object)	Comienza una operación asincrónica para aceptar un intento de conexión entrante y recibe el primer bloque de datos enviado por la aplicación cliente.

BeginAccept(Socket, Int32, AsyncCallback, Object)	Comienza una operación asincrónica para aceptar un intento de conexión entrante desde un socket específico y recibe el primer bloque de datos enviado por la aplicación cliente.
BeginConnect(EndPoint, AsyncCallback, Object)	Inicia una solicitud asincrónica para una conexión a host remoto.
BeginConnect(IPAddress, Int32, AsyncCallback, Object)	Inicia una solicitud asincrónica para una conexión a host remoto. El host se especifica mediante una IPAddress y un número de puerto.
BeginConnect(IPAddress[], Int32, AsyncCallback, Object)	Inicia una solicitud asincrónica para una conexión a host remoto. El host se especifica mediante una matriz IPAddress y un número de puerto.
BeginConnect(String, Int32, AsyncCallback, Object)	Inicia una solicitud asincrónica para una conexión a host remoto. El host se especifica mediante un nombre de host y un número de puerto.
BeginDisconnect(Boolean, AsyncCallback, Object)	Comienza una solicitud asincrónica para la desconexión de un extremo remoto.
BeginReceive(Byte[], Int32, Int32, SocketFlags, AsyncCallback, Object)	Comienza a recibir asincrónicamente los datos de un objeto Socketconectado.
BeginReceive(Byte[], Int32, Int32, SocketFlags, SocketError, AsyncCallback, Object)	Comienza a recibir asincrónicamente los datos de un objeto Socketconectado.
BeginReceive(IList<ArraySegment<Byte>>, SocketFlags, AsyncCallback, Object)	Comienza a recibir asincrónicamente los datos de un objeto Socketconectado.
BeginReceive(IList<ArraySegment<Byte>>, SocketFlags, SocketError, AsyncCallback, Object)	Comienza a recibir asincrónicamente los datos de un objeto Socketconectado.
BeginReceiveFrom(Byte[], Int32, Int32, SocketFlags, EndPoint, AsyncCallback, Object)	Comienza a recibir asincrónicamente los datos de un dispositivo de red especificado.
BeginReceiveMessageFrom(Byte[], Int32, Int32, SocketFlags, EndPoint, AsyncCallback, Object)	Empieza a recibir de forma asincrónica el número especificado de bytes de datos en la ubicación indicada del búfer de datos, mediante el marcador SocketFlags especificado, y almacena el punto de conexión y la información sobre paquetes.
BeginSend(Byte[], Int32, Int32, SocketFlags, AsyncCallback, Object)	Envía datos asincrónicamente a un objeto Socket conectado.
BeginSend(Byte[], Int32, Int32, SocketFlags, SocketError, AsyncCallback, Object)	Envía datos asincrónicamente a un objeto Socket conectado.
BeginSend(IList<ArraySegment<Byte>>, SocketFlags, AsyncCallback, Object)	Envía datos asincrónicamente a un objeto Socket conectado.
BeginSend(IList<ArraySegment<Byte>>, SocketFlags, SocketError, AsyncCallback, Object)	Envía datos asincrónicamente a un objeto Socket conectado.
BeginSendFile(String, AsyncCallback, Object)	Envía el archivo fileName a un objeto Socket conectado mediante el marcador UseDefaultWorkerThread.
BeginSendFile(String, Byte[], Byte[], TransmitFileOptions, AsyncCallback, Object)	Envía, de forma asincrónica, un archivo y búferes de datos a un objeto Socket conectado.
BeginSendTo(Byte[], Int32, Int32, SocketFlags, EndPoint, AsyncCallback, Object)	Envía datos de forma asincrónica a un host remoto concreto.
Bind(EndPoint)	Asocia un objeto Socket a un extremo local.
CancelConnectAsync(SocketAsyncEventArgs)	Cancela una solicitud asincrónica de una conexión a

	un host remoto.
Close()	Cierra la conexión Socket y libera todos los recursos asociados.
Close(Int32)	Cierra la conexión Socket y libera todos los recursos asociados con un tiempo de espera especificado para permitir el envío de los datos en cola.
Connect(EndPoint)	Establece una conexión a un host remoto.
Connect(IPAddress, Int32)	Establece una conexión a un host remoto. El host se especifica mediante una dirección IP y un número de puerto.
Connect(IPAddress[], Int32)	Establece una conexión a un host remoto. El host se especifica mediante una matriz de direcciones IP y un número de puerto.
Connect(String, Int32)	Establece una conexión a un host remoto. El host se especifica mediante un nombre de host y un número de puerto.
ConnectAsync(SocketAsyncEventArgs)	Comienza una solicitud asincrónica para una conexión a host remoto.
ConnectAsync(SocketType, ProtocolType, SocketAsyncEventArgs)	Comienza una solicitud asincrónica para una conexión a host remoto.
Disconnect(Boolean)	Cierra la conexión del socket y permite reutilizarlo.
DisconnectAsync(SocketAsyncEventArgs)	Comienza una solicitud asincrónica para la desconexión de un extremo remoto.
Dispose()	Libera todos los recursos usados por la instancia actual de la clase Socket.
Dispose(Boolean)	Libera los recursos no administrados que usa Socket y, de forma opcional, desecha los recursos administrados.
DuplicateAndClose(Int32)	Duplica la referencia del socket para el proceso de destino y cierra el socket para este proceso.
EndAccept(Byte[], IAsyncResult)	Acepta de forma asincrónica un intento de conexión entrante y crea un objeto Socket nuevo para controlar la comunicación con el host remoto.Este método devuelve un búfer que contiene los datos iniciales transferidos.
EndAccept(Byte[], Int32, IAsyncResult)	Acepta de forma asincrónica un intento de conexión entrante y crea un objeto Socket nuevo para controlar la comunicación con el host remoto.Este método devuelve un búfer que contiene los datos iniciales y el número de bytes transferidos.
EndAccept(IAsyncResult)	Acepta asincrónicamente un intento de conexión entrante y crea un nuevo objeto Socket para controlar la comunicación con el host remoto.
EndConnect(IAsyncResult)	Finaliza una solicitud de conexión asincrónica pendiente.
EndDisconnect(IAsyncResult)	Finaliza una solicitud de desconexión asincrónica pendiente.
EndReceive(IAsyncResult)	Finaliza una lectura asincrónica pendiente.
EndReceive(IAsyncResult, SocketError)	Finaliza una lectura asincrónica pendiente.

EndReceiveFrom(IAsyncResult, EndPoint)	Finaliza una lectura asincrónica pendiente desde un extremo específico.
EndReceiveMessageFrom(IAsyncResult, SocketFlags, EndPoint, IPPacketInformation)	Finaliza una lectura asincrónica pendiente desde un extremo específico.Este método también desvela más información sobre el paquete que EndReceiveFrom(IAsyncResult, EndPoint).
EndSend(IAsyncResult)	Finaliza un envío asincrónico pendiente.
EndSend(IAsyncResult, SocketError)	Finaliza un envío asincrónico pendiente.
EndSendFile(IAsyncResult)	Finaliza un envío asincrónico de archivo pendiente.
EndSendTo(IAsyncResult)	Finaliza un envío asincrónico pendiente en una ubicación específica.
Equals(Object)	Determina si el objeto especificado es igual al objeto actual. (Inherited from Object)
Finalize()	Libera los recursos utilizados por la clase Socket.
GetSocketOption(SocketOptionLevel, SocketOptionName)	Devuelve el valor de una opción de Socket especificada en forma de objeto.
GetSocketOption(SocketOptionLevel, SocketOptionName, Byte[])	Devuelve el valor de la opción de Socket especificada, representado como una matriz de bytes.
GetSocketOption(SocketOptionLevel, SocketOptionName, Int32)	Devuelve el valor de la opción de Socket especificada en una matriz.
GetType()	Obtiene el Type de la instancia actual. (Inherited from Object)
IOControl(Int32, Byte[], Byte[])	Establece modos operativos de bajo nivel para el Socket que utiliza códigos de control numéricos.
IOControl(IOControlCode, Byte[], Byte[])	Establece modos operativos de bajo nivel para el Socket que utiliza la enumeración IOControlCode para especificar códigos de control.
Listen(Int32)	Coloca un objeto Socket en un estado de escucha.
MemberwiseClone()	Crea una copia superficial del objeto Object actual. (Inherited from Object)
Poll(Int32, SelectMode)	Determina el estado de Socket.
Receive(Byte[])	Recibe datos de un Socket enlazado en un búfer de recepción.
Receive(Byte[], Int32, Int32, SocketFlags)	Recibe el número especificado de bytes de un objeto Socket enlazado en la posición de desplazamiento especificada del búfer de recepción, usando el valor de SocketFlags especificado.
Receive(Byte[], Int32, Int32, SocketFlags, SocketError)	Recibe datos de un Socket enlazado en un búfer de recepción, usando el valor de SocketFlags especificado.
Receive(Byte[], Int32, SocketFlags)	Recibe el número especificado de bytes de datos de un objeto Socketenlazado en un búfer de recepción, usando el valor de SocketFlagsespecificado.
Receive(Byte[], SocketFlags)	Recibe datos de un Socket enlazado en un búfer de recepción, usando el valor de SocketFlags especificado.
Receive(IList<ArraySegment<Byte>>)	Recibe, en la lista de búferes de recepción, datos de

	un Socket enlazado.
Receive(IList<ArraySegment<Byte>>, SocketFlags)	Recibe, en la lista de búferes de recepción, datos de un Socket enlazado, utilizando el valor de SocketFlags especificado.
Receive(IList<ArraySegment<Byte>>, SocketFlags, SocketError)	Recibe, en la lista de búferes de recepción, datos de un Socket enlazado, utilizando el valor de SocketFlags especificado.
ReceiveAsync(SocketAsyncEventArgs)	Comienza una solicitud asincrónica para recibir los datos de un objeto Socket conectado.
ReceiveFrom(Byte[], EndPoint)	Recibe un datagrama en el búfer de datos y almacena el extremo.
ReceiveFrom(Byte[], Int32, Int32, SocketFlags, EndPoint)	Recibe el número especificado de bytes de datos en la ubicación especificada del búfer de datos, mediante el SocketFlags especificado y almacena el punto de conexión.
ReceiveFrom(Byte[], Int32, SocketFlags, EndPoint)	Recibe el número especificado de bytes en el búfer de datos mediante el marcador SocketFlags especificado y almacena el extremo.
ReceiveFrom(Byte[], SocketFlags, EndPoint)	Recibe un datagrama en el búfer de datos usando el objeto SocketFlagsespecificado y almacena el extremo.
ReceiveFromAsync(SocketAsyncEventArgs)	Comienza a recibir asincrónicamente los datos de un dispositivo de red especificado.
ReceiveMessageFrom(Byte[], Int32, Int32, SocketFlags, EndPoint, IPPacketInformation)	Recibe el número especificado de bytes de datos en la ubicación especificada del búfer de datos, mediante el elemento SocketFlagsespecificado y almacena el punto de conexión y la información del paquete.
ReceiveMessageFromAsync(SocketAsyncEventArgs)	Comienza a recibir de forma asincrónica el número especificado de bytes de datos en la ubicación indicada del búfer de datos, mediante la propiedad SocketFlags especificada, y almacena la información sobre el extremo y el paquete.
Select(IList, IList, IList, Int32)	Determina el estado de uno o varios sockets.
Send(Byte[])	Envía datos a un objeto Socket conectado.
Send(Byte[], Int32, Int32, SocketFlags)	Envía el número especificado de bytes de datos a un Socket conectado, comenzando en el desplazamiento especificado y usando el SocketFlagsespecificado.
Send(Byte[], Int32, Int32, SocketFlags, SocketError)	Envía el número especificado de bytes de datos a un objeto Socketconectado, a partir de la posición de desplazamiento especificada y usando el valor de SocketFlags especificado.
Send(Byte[], Int32, SocketFlags)	Envía el número especificado de bytes de datos a un objeto Socketconectado, usando el marcador SocketFlags especificado.
Send(Byte[], SocketFlags)	Envía datos a un objeto Socket conectado mediante el marcador SocketFlags especificado.
Send(IList<ArraySegment<Byte>>)	Envía el conjunto de búferes de la lista a un Socket conectado.
Send(IList<ArraySegment<Byte>>, SocketFlags)	Envía el conjunto de búferes de la lista a un Socket

	conectado, utilizando el SocketFlags especificado.
Send(IList<ArraySegment<Byte>>, SocketFlags, SocketError)	Envía el conjunto de búferes de la lista a un Socket conectado, utilizando el SocketFlags especificado.
SendAsync(SocketAsyncEventArgs)	Envía datos de forma asincrónica a un objeto Socket conectado.
SendFile(String)	Envía el archivo fileName a un objeto Socket conectado con el marcador de transmisión UseDefaultWorkerThread.
SendFile(String, Byte[], Byte[], TransmitFileOptions)	Envía el archivo fileName y búferes de datos a un objeto Socket conectado mediante el valor TransmitFileOptions especificado.
SendPacketsAsync(SocketAsyncEventArgs)	Envía de forma asincrónica una colección de archivos o búferes de datos en memoria a un objeto Socket conectado.
SendTo(Byte[], EndPoint)	Envía los datos al extremo especificado.
SendTo(Byte[], Int32, Int32, SocketFlags, EndPoint)	Envía el número especificado de bytes de datos al extremo especificado, comenzando en la ubicación especificada del búfer y usando los SocketFlags especificados.
SendTo(Byte[], Int32, SocketFlags, EndPoint)	Envía el número especificado de bytes de datos al extremo especificado usando los SocketFlags especificados.
SendTo(Byte[], SocketFlags, EndPoint)	Envía datos a un extremo específico mediante el marcador SocketFlagsespecificado.
SendToAsync(SocketAsyncEventArgs)	Envía datos de forma asincrónica a un host remoto concreto.
SetIPProtectionLevel(IPProtectionLevel)	Establece el nivel de protección IP en un socket.
SetSocketOption(SocketOptionLevel, SocketOptionName, Boolean)	Establece la opción de Socket especificada en el valor de Boolean indicado.
SetSocketOption(SocketOptionLevel, SocketOptionName, Byte[])	Establece la opción de Socket indicada en el valor especificado, representado como una matriz de bytes.
SetSocketOption(SocketOptionLevel, SocketOptionName, Int32)	Establece la opción de Socket especificada en el valor entero indicado.
SetSocketOption(SocketOptionLevel, SocketOptionName, Object)	Establece la opción de Socket indicada en el valor especificado, representado como un objeto.
Shutdown(SocketShutdown)	Deshabilita los envíos y recepciones en un objeto Socket.
ToString()	Devuelve una cadena que representa el objeto actual. (Inherited from Object)

Métodos de extensión

AcceptAsync(Socket)	Realiza una operación asincrónica para aceptar un intento de conexión entrante en el socket.
AcceptAsync(Socket, Socket)	Realiza una operación asincrónica para aceptar un intento de conexión entrante en el socket.
ConnectAsync(Socket, EndPoint)	Establece una conexión a un host remoto.
ConnectAsync(Socket, IPAddress, Int32)	Establece una conexión a un host remoto. El host se especifica mediante una dirección IP y un número de puerto.

ConnectAsync(Socket, IPAddress[], Int32)	Establece una conexión a un host remoto. El host se especifica mediante una matriz de direcciones IP y un número de puerto.
ConnectAsync(Socket, String, Int32)	Establece una conexión a un host remoto. El host se especifica mediante un nombre de host y un número de puerto.
ReceiveAsync(Socket, ArraySegment<Byte>, SocketFlags)	Recibe datos de un socket conectado.
ReceiveAsync(Socket, IList<ArraySegment<Byte>>, SocketFlags)	Recibe datos de un socket conectado.
ReceiveFromAsync(Socket, ArraySegment<Byte>, SocketFlags, EndPoint)	Recibe datos de un dispositivo de red especificado.
ReceiveMessageFromAsync(Socket, ArraySegment<Byte>, SocketFlags, EndPoint)	Recibe el número especificado de bytes de datos en la ubicación especificada del búfer de datos, mediante el elemento SocketFlagsespecificado y almacena el punto de conexión y la información del paquete.
SendAsync(Socket, ArraySegment<Byte>, SocketFlags)	Envía datos a un socket conectado.
SendAsync(Socket, IList<ArraySegment<Byte>>, SocketFlags)	Envía datos a un socket conectado.
SendToAsync(Socket, ArraySegment<Byte>, SocketFlags, EndPoint)	Envía datos de forma asincrónica a un host remoto concreto.

4.6.- CLIENTE/SERVIDOR TCP SÍNCRONO.

Un socket TCP se instancia pasando como parámetro el protocolo TCP al contructor. A la hora de instanciar el socket no existen diferencias entre un socket síncrono y asíncrono. La diferencia será que para una comunicación síncrona se va utilizan métodos síncronos que bloquean el hilo hasta que se finaliza la lectura o escritura a través del socket.

Ejemplo Cliente

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

public class SynchronousSocketClient {

    public static void StartClient() {
        // Data buffer for incoming data.
        byte[] bytes = new byte[1024];
  Este array se utiliza como buffer de lectura. Se debe tener en cuenta que en función del tamaño de los datos que se
        // Connect to a remote device.
        try {
            // Establish the remote endpoint for the socket.
            // This example uses port 11000 on the local computer.
            IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
            IPAddress ipAddress = ipHostInfo.AddressList[0];
            IPEndPoint remoteEP = new IPEndPoint(ipAddress,11000);
  En el ejemplo el servidor esta en la primera de la direcciones IP de la maquina, es decir, el servidor y el cliente
            // Create a TCP/IP  socket.
            Socket sender = new Socket(ipAddress.AddressFamily,
                SocketType.Stream, ProtocolType.Tcp );
  Se crea el socket especificando el tipo de dirección IP, el tipo de socket y el protocolo.
            // Connect the socket to the remote endpoint. Catch any errors.
            try {
                sender.Connect(remoteEP);
```

```
 Se establece la conexión con el servidor. El parámetro de entrada a la función son lo datos que se han definido ant
                Console.WriteLine("Socket connected to {0}",
                    sender.RemoteEndPoint.ToString());

                // Encode the data string into a byte array.
                byte[] msg = Encoding.ASCII.GetBytes("This is a test");
 Codificación de los datos que se quieren enviar.
                // Send the data through the socket.
                int bytesSent = sender.Send(msg);
 Envio de los datos codificados al servidor
                // Receive the response from the remote device.
                int bytesRec = sender.Receive(bytes);
                Console.WriteLine("Echoed test = {0}",
                    Encoding.ASCII.GetString(bytes,0,bytesRec));
  Lectura de los datos recibidos desde es servidor a buffer de lectura creado y a continuación se escriben por consol
                // Release the socket.
                sender.Shutdown(SocketShutdown.Both);
                sender.Close();
 Liberación de los recursos del socket.
            } catch (ArgumentNullException ane) {
                Console.WriteLine("ArgumentNullException : {0}",ane.ToString());
            } catch (SocketException se) {
                Console.WriteLine("SocketException : {0}",se.ToString());
            } catch (Exception e) {
                Console.WriteLine("Unexpected exception : {0}", e.ToString());
            }

        } catch (Exception e) {
            Console.WriteLine( e.ToString());
        }
    }

    public static int Main(String[] args) {
        StartClient();
        return 0;
    }
}
```

Todas las llamadas a funciones realizadas sobre el socket son síncronas, por lo que el hilo no avanza hasta que se establece la conexión, se realiza el envío o se termina la lectura de datos. Aunque C# no exige el uso de bloques try/catch su uso es imprescindible para que cualquier excepción en tiempo de ejecución pueda ser tratada de forma adecuada.

Ejemplo Servidor

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

public class SynchronousSocketListener {

    // Incoming data from the client.
    public static string data = null;

    public static void StartListening() {
        // Data buffer for incoming data.
        byte[] bytes = new Byte[1024];
  Buffer para la lectura de datos desde recibidos desde el cliente
        // Establish the local endpoint for the socket.
        // Dns.GetHostName returns the name of the
        // host running the application.
        IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
        IPAddress ipAddress = ipHostInfo.AddressList[0];
        IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);
  Datos para EPEndPoint, primera dirección de la lista ipHostInfo y puerto 11000
        // Create a TCP/IP socket.
        Socket listener = new Socket(ipAddress.AddressFamily,
            SocketType.Stream, ProtocolType.Tcp );
  El socket se instancia de la misma manera que en el cliente, pasando como parámetro el tipo de dirección IP, el tip
        // Bind the socket to the local endpoint and
        // listen for incoming connections.
        try {
            listener.Bind(localEndPoint);
```

```
            listener.Listen(10);
  Asocia al socket el EPEndPoint definido e indica al socket que comience a escuchar peticiones estableciendo la long
            // Start listening for connections.
            while (true) {
                Console.WriteLine("Waiting for a connection...");
                // Program is suspended while waiting for an incoming connection.
                Socket handler = listener.Accept();
                data = null;
  Se crea un nuevo socket a partir del socket que se creo con anterioridad. El socket se creó con una cola de peticio
                // An incoming connection needs to be processed.
                while (true) {
                    int bytesRec = handler.Receive(bytes);
                    data += Encoding.ASCII.GetString(bytes,0,bytesRec);
                    if (data.IndexOf("<EOF>") > -1) {
                        break;
                    }
                }
 Se leen los datos del buffer hasta que se encuenta el token que marca el fin del fichero y en ese momento se sale de
                // Show the data on the console.
                Console.WriteLine( "Text received : {0}", data);

                // Echo the data back to the client.
                byte[] msg = Encoding.ASCII.GetBytes(data);
                handler.Send(msg);
                handler.Shutdown(SocketShutdown.Both);
                handler.Close();
  Se devuelve el mensaje recibido al cliente y se libera el socket que se ha utilizado para leer y enviar datos al cl
            }

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }

        Console.WriteLine("\nPress ENTER to continue...");
        Console.Read();

    }

    public static int Main(String[] args) {
        StartListening();
        return 0;
    }
}
```

El código propuesto para el servidor el capaz de entener multiples llamadas, pero de una en una. Es decir, hasta que no ha terminado de atender la una petición de un cliente no comienza con una nueva. Más adelante se vera como hacer que un servidor atienda a más de un cliente a la vez.

4.7.- CLIENTE/SERVIDOR TCP ASÍNCRONO.

Tal y como se ha expuesto anteriormente, si se utilizan métodos síncronos el hilo en le cual se realizan las llamadas se queda bloqueado a la espera de que los métodos finalizen su ejecución. Hay casos en los que el hilo que realiza las llamadas no puede quedarse bloqueado: servicios Windows, aplicaciones moviles, hilos que manejan interfaces gráficas de usuario... A continuación se muestra un ejemplo de una comunición TCP haciendo uso de métodos asíncronos:

Ejemplo cliente

```
  using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;

// State object for receiving data from remote device.
public class StateObject {
    // Client socket.
    public Socket workSocket = null;
    // Size of receive buffer.
    public const int BufferSize = 256;
    // Receive buffer.
```

```
    public byte[] buffer = new byte[BufferSize];
    // Received data string.
    public StringBuilder sb = new StringBuilder();
}

public class AsynchronousClient {
    // The port number for the remote device.
    private const int port = 11000;

    // ManualResetEvent instances signal completion.
    private static ManualResetEvent connectDone =
        new ManualResetEvent(false);
    private static ManualResetEvent sendDone =
        new ManualResetEvent(false);
    private static ManualResetEvent receiveDone =
        new ManualResetEvent(false);

    // The response from the remote device.
    private static String response = String.Empty;

    private static void StartClient() {
        // Connect to a remote device.
        try {
            // Establish the remote endpoint for the socket.
            // The name of the
            // remote device is "host.contoso.com".
            IPHostEntry ipHostInfo = Dns.GetHostEntry("host.contoso.com");
            IPAddress ipAddress = ipHostInfo.AddressList[0];
            IPEndPoint remoteEP = new IPEndPoint(ipAddress, port);

            // Create a TCP/IP socket.
            Socket client = new Socket(ipAddress.AddressFamily,
                SocketType.Stream, ProtocolType.Tcp);

            // Connect to the remote endpoint.
            client.BeginConnect( remoteEP,
                new AsyncCallback(ConnectCallback), client);
            connectDone.WaitOne();

            // Send test data to the remote device.
            Send(client,"This is a test<EOF>");
            sendDone.WaitOne();

            // Receive the response from the remote device.
            Receive(client);
            receiveDone.WaitOne();

            // Write the response to the console.
            Console.WriteLine("Response received : {0}", response);

            // Release the socket.
            client.Shutdown(SocketShutdown.Both);
            client.Close();

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    private static void ConnectCallback(IAsyncResult ar) {
        try {
            // Retrieve the socket from the state object.
            Socket client = (Socket) ar.AsyncState;

            // Complete the connection.
            client.EndConnect(ar);

            Console.WriteLine("Socket connected to {0}",
                client.RemoteEndPoint.ToString());

            // Signal that the connection has been made.
            connectDone.Set();
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }
```

```
    private static void Receive(Socket client) {
        try {
            // Create the state object.
            StateObject state = new StateObject();
            state.workSocket = client;

            // Begin receiving the data from the remote device.
            client.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
                new AsyncCallback(ReceiveCallback), state);
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    private static void ReceiveCallback( IAsyncResult ar ) {
        try {
            // Retrieve the state object and the client socket
            // from the asynchronous state object.
            StateObject state = (StateObject) ar.AsyncState;
            Socket client = state.workSocket;

            // Read data from the remote device.
            int bytesRead = client.EndReceive(ar);

            if (bytesRead > 0) {
                // There might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(state.buffer,0,bytesRead));

                // Get the rest of the data.
                client.BeginReceive(state.buffer,0,StateObject.BufferSize,0,
                    new AsyncCallback(ReceiveCallback), state);
            } else {
                // All the data has arrived; put it in response.
                if (state.sb.Length > 1) {
                    response = state.sb.ToString();
                }
                // Signal that all bytes have been received.
                receiveDone.Set();
            }
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    private static void Send(Socket client, String data) {
        // Convert the string data to byte data using ASCII encoding.
        byte[] byteData = Encoding.ASCII.GetBytes(data);

        // Begin sending the data to the remote device.
        client.BeginSend(byteData, 0, byteData.Length, 0,
            new AsyncCallback(SendCallback), client);
    }

    private static void SendCallback(IAsyncResult ar) {
        try {
            // Retrieve the socket from the state object.
            Socket client = (Socket) ar.AsyncState;

            // Complete sending the data to the remote device.
            int bytesSent = client.EndSend(ar);
            Console.WriteLine("Sent {0} bytes to server.", bytesSent);

            // Signal that all bytes have been sent.
            sendDone.Set();
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    public static int Main(String[] args) {
        StartClient();
        return 0;
    }
}
```

Ejemplo servidor

```
  using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

// State object for reading client data asynchronously
public class StateObject {
    // Client  socket.
    public Socket workSocket = null;
    // Size of receive buffer.
    public const int BufferSize = 1024;
    // Receive buffer.
    public byte[] buffer = new byte[BufferSize];
// Received data string.
    public StringBuilder sb = new StringBuilder();
}

public class AsynchronousSocketListener {
    // Thread signal.
    public static ManualResetEvent allDone = new ManualResetEvent(false);

    public AsynchronousSocketListener() {
    }

    public static void StartListening() {
        // Establish the local endpoint for the socket.
        // The DNS name of the computer
        // running the listener is "host.contoso.com".
        IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
        IPAddress ipAddress = ipHostInfo.AddressList[0];
        IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

        // Create a TCP/IP socket.
        Socket listener = new Socket(ipAddress.AddressFamily,
            SocketType.Stream, ProtocolType.Tcp );

        // Bind the socket to the local endpoint and listen for incoming connections.
        try {
            listener.Bind(localEndPoint);
            listener.Listen(100);

            while (true) {
                // Set the event to nonsignaled state.
                allDone.Reset();

                // Start an asynchronous socket to listen for connections.
                Console.WriteLine("Waiting for a connection...");
                listener.BeginAccept(
                    new AsyncCallback(AcceptCallback),
                    listener );

                // Wait until a connection is made before continuing.
                allDone.WaitOne();
            }

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }

        Console.WriteLine("\nPress ENTER to continue...");
        Console.Read();

    }

    public static void AcceptCallback(IAsyncResult ar) {
        // Signal the main thread to continue.
        allDone.Set();

        // Get the socket that handles the client request.
        Socket listener = (Socket) ar.AsyncState;
        Socket handler = listener.EndAccept(ar);
```

```
        // Create the state object.
        StateObject state = new StateObject();
        state.workSocket = handler;
        handler.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
            new AsyncCallback(ReadCallback), state);
    }

    public static void ReadCallback(IAsyncResult ar) {
        String content = String.Empty;

        // Retrieve the state object and the handler socket
        // from the asynchronous state object.
        StateObject state = (StateObject) ar.AsyncState;
        Socket handler = state.workSocket;

        // Read data from the client socket.
        int bytesRead = handler.EndReceive(ar);

        if (bytesRead > 0) {
            // There  might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(
                state.buffer, 0, bytesRead));

            // Check for end-of-file tag. If it is not there, read
            // more data.
            content = state.sb.ToString();
            if (content.IndexOf("<EOF>") > -1) {
                // All the data has been read from the
                // client. Display it on the console.
                Console.WriteLine("Read {0} bytes from socket. \n Data : {1}",
                    content.Length, content );
                // Echo the data back to the client.
                Send(handler, content);
            } else {
                // Not all data received. Get more.
                handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
                new AsyncCallback(ReadCallback), state);
            }
        }
    }

    private static void Send(Socket handler, String data) {
        // Convert the string data to byte data using ASCII encoding.
        byte[] byteData = Encoding.ASCII.GetBytes(data);

        // Begin sending the data to the remote device.
        handler.BeginSend(byteData, 0, byteData.Length, 0,
            new AsyncCallback(SendCallback), handler);
    }

    private static void SendCallback(IAsyncResult ar) {
        try {
            // Retrieve the socket from the state object.
            Socket handler = (Socket) ar.AsyncState;

            // Complete sending the data to the remote device.
            int bytesSent = handler.EndSend(ar);
            Console.WriteLine("Sent {0} bytes to client.", bytesSent);

            handler.Shutdown(SocketShutdown.Both);
            handler.Close();

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    public static int Main(String[] args) {
        StartListening();
        return 0;
    }
}
```

4.8.- CLIENTE/SERVIDOR UDP.

Para los escenarios en los que una latencia baja es más importante que la integridad de los datos transmitidos se debería utilizar sockets UDP para la transmisión de los datos. Otra ventaja de UDP es el menor ancho de banda consumido, aspecto fundamental para aplicaciones que utilizan redes de datos móviles como las aplicaciones IOT. Por último, UDP permite el uso de la dirección de broadcast, por lo que si se quiere descubrir máquinas dentro de la red o se quiere enviar un mensaje a todos los nodos de la red se deberá utilizar UDP y no TCP.

A la hora de instanciar el socket hay que especificar el protocolo, UDP y el tipo de socket, socket de datagramas.

Ejemplo cliente broadcast

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

class Program
{
    static void Main(string[] args)
    {
        Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

        IPAddress broadcast = IPAddress.Parse("192.168.1.255");

        byte[] sendbuf = Encoding.ASCII.GetBytes(args[0]);
        IPEndPoint ep = new IPEndPoint(broadcast, 11000);

        s.SendTo(sendbuf, ep);

        Console.WriteLine("Message sent to the broadcast address");
    }
}
```

Ejemplo Servidor broadcast

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

public class UDPListener
{
    private const int listenPort = 11000;

    private static void StartListener()
    {
        UdpClient listener = new UdpClient(listenPort);
        IPEndPoint groupEP = new IPEndPoint(IPAddress.Any, listenPort);

        try
        {
            while (true)
            {
                Console.WriteLine("Waiting for broadcast");
                byte[] bytes = listener.Receive(ref groupEP);

                Console.WriteLine($"Received broadcast from {groupEP} :");
                Console.WriteLine($" {Encoding.ASCII.GetString(bytes, 0, bytes.Length)}");
            }
        }
        catch (SocketException e)
        {
            Console.WriteLine(e);
```

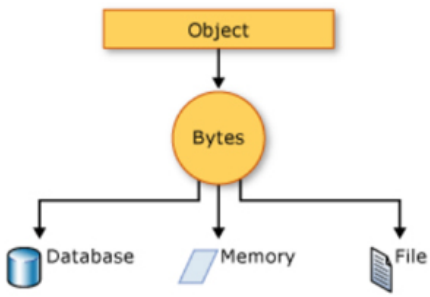
```
        }
        finally
        {
            listener.Close();
        }
    }

    public static void Main()
    {
        StartListener();
    }
}
```

4.9.- ENVÍO DE OBJETOS A TRAVÉS DE SOCKETS: SERIALIZACIÓN.

Hasta ahora se han estado intercambiando cadenas de caracteres entre los programas cliente y servidor. Pero los sockets permiten el envio de datos serializados. En C# los datos se pueden serializar en binario o XML. De la misma manera que se serializan los objetos para guardar su estado en disco, los objetos de pueden serializar y usar streams para su envio.

serialización



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

En la serialización binaria se serializan todos los miembros, incluso aquellos que son de solo lectura, y mejora el rendimiento. La serialización XML proporciona código más legible, así como mayor flexibilidad para compartir objetos y usarlos para fines de interoperabilidad.

Para serializar un objeto, su clase debe ser serializable. Para ello, se debe indicar que la clase es serializable utilizando el atributo [Serializable], si alguno de atributos no se quiere serializar se puede indicar haciendo uso del atributo [NonSerialized()].

Si se quiere especificar como se debe serializar y deserializar los objetos se debe implementar la interfaz ISerializable.

Ejemplo

```
  [Serializable]
public class Mensaje : ISerializable
    {
        public string Sms { get; set; }
        public string Resumen { get; set;}
        public DateTime Stamp { get; set; }
        public Mensaje() { }
        public Mensaje(string sms,string resumen)
        {
            this.Sms = sms;
            this.Resumen = resumen;
            Stamp = DateTime.Now;
        }
        public Mensaje(SerializationInfo info,StreamingContext context)
        {
            Sms = info.GetString("mensaje");
            Resumen = (string)info.GetValue("resumen", typeof(string));
            Stamp = (DateTime)info.GetValue("fecha", typeof(DateTime));
        }

        public void GetObjectData(SerializationInfo info, StreamingContext context)
        {
```

```
            info.AddValue("mensaje", Sms);
            info.AddValue("resumen", Resumen);
            info.AddValue("fecha", Stamp);
        }
    }
```

Serialización binaria

Para realizar la serialización binaria es necesario utilizar un formaterador binario y un stream, en este caso el stream es un MemoryStream ya que el flujo no se va a escribir en disco. El formateador se encarga de serializar el objeto en el stream. Por ultimo, hay que convertir los datos que contiene el stream a un array de bytes.

Ejemplo

```
  Mensaje sms = new Mensaje("Este es mi primer mensaje serializado","7777");
IFormatter formatter = new BinaryFormatter();
Stream stream = new MemoryStream();
formatter.Serialize(stream, sms);
byte[] byteData = ((MemoryStream)stream).ToArray();
client.BeginSend(byteData, 0, byteData.Length, 0,
                new AsyncCallback(SendCallback), client);
```

Deserialización binaria

Para realizar la deserialización binaria se utilizan los mismos elementos que para realizar la serialización. El buffer de datos recibidos a través de socket se añade a un MemoryStream y se llama al método Deserialize() del formateador.

Ejemplo

```
  XmlSerializer serializer = new XmlSerializer(typeof(Mensaje));
Stream stream = new MemoryStream();
serializer.Serialize(stream, data);
byte[] byteData = ((MemoryStream)stream).ToArray();
handler.BeginSend(byteData, 0, byteData.Length, 0,
            new AsyncCallback(SendCallback), handler);
```

Deserialización XML

Para realizar la deserialización XML es necesario que la clase tenga el constructor por defecto. En caso de no tenerlo se lanza una excepción en tiempo de ejecución. El serializador XML también necesita conocer el tipo de datos del objeto que se quiere obtener.

Ejemplo

```
  byte[] byteArray = Encoding.ASCII.GetBytes(response);
MemoryStream stream = new MemoryStream(byteArray);
Mensaje recibido = (Mensaje)new XmlSerializer(typeof(Mensaje)).Deserialize(stream);
```

5.- CONEXIONES MÚLTIPLES CLIENTES. HILOS.

Los programas servidores síncronos vistos hasta ahora solo pueden atender a un cliente en cada momento, pero lo más habitual es que un servidor atienda a muchos clientes de forma simultánea. La solución para poder atender a múltiples clientes está en el uso de hilos o tareas para atender a cada una de las peticiones en a la vez. Tal y como se explico anteriormente el socket que escucha las peticiones de los clientes tiene una cola con las conexiones pendientes, pero este socket no es el que se utiliza para enviar o recibir datos.

```
  Socket listener = new Socket(ipAddress.AddressFamily,
try {
listener.Bind(localEndPoint);
listener.Listen(10);
```

Por lo tanto, una vez se acepta cada una de las conexiones de los clientes y se retorna el socket que se va a utilizar para recibir y enviar datos al cliente, se debe utilizar un hilo o tarea para atender a cada cliente.

```
 Socket handler = listener.Accept();
//iniciar tarea o hilo con el handler
```