I Socket in C#

Rev. Digitale 1.0 del 01/09/2016

La classe System.Net.Socket.Sockets

```
Imports System.Net    -    Imports System.Net.Sockets
```

La classe System.Net.ipHostEntry

Memorizza tutte le informazioni relative ad un host, in particolare: indirizzo **IPv4**, indirizzo **IPv6**, **hostName**, etc.

La classe System.Net.DNS

```
Dns.GetHostName() senza parametri restitisce il nome dell'host corrente
Dns.GetHostEntry("hostName")     oppure
Dns.GetHostEntry("hostIP")
```

ricevuto come parametro il **Nome** o l'**IP Address** (String) di un qualunque host della rete restituiscono il corrispondente oggetto **ipHostEntry** contenente tutte le informazioni relative all'host.

La classe clsAddress

```
Public Class clsAddress {
    public static IPAddress[] localAddress;    // Elenco degli IP della macchina locale
    static Address()  {
        localAddress = new IPAddress[6];
        for (int i = 0; i < 5; i++)   localAddress [i] = null;
    }
```

// Carica nel vettore gli oggetti IPv4 relativi alla macchina locale.
Gli IPv4 possono essere molteplici e l'ordine con cui vengono restituiti è totalmente arbitrario

```
public static void cercaIP() {
     IPHostEntry hostInfo = Dns.GetHostEntry(Dns.GetHostName());
     int cnt = 0;
     foreach (IPAddress ip in hostInfo.AddressList)  {
          if (ip.AddressFamily == AddressFamily.InterNetwork)
                  localAddress[cnt++] = ip;
     }
     ipVett[cnt++] = cercaIP("127.0.0.1");
}
```

// Riceve un HostName di rete (oppure il suo indirizzo IP sotto forma di stringa) e restituisce l'oggetto IP corrispondente. In questo caso **l'indirizzo IP restituito è sempre solo uno** (quello su cui l'host ha risposto), per cui non serve mantenere un vettore, ma l'IP Object viene restituito dalla funzione:

```
public static IPAddress cercaIP(string host)  {
    IPAddress ip = null;
    if (!(IPAddress.TryParse(host, out ip))) {
          IPHostEntry hostInfo = Dns.GetHostEntry(host);
          foreach (IPAddress aus in hostInfo.AddressList)  {
                if (aus.AddressFamily == AddressFamily.InterNetwork) {
                      ip=aus;
                      break;
                }
          }
    } return ip;
```

La classe System.Text.Encoding

La comunicazione tra socket consiste nella trasmissione di un vettore di bytes, detto **STREAM**. Il fatto di trasmettere degli stream anziché dei dati tipizzati, garantisce la comunicazione anche fra ambienti differenti. Per i socket si usa normalmente uno stream ASCII ma non c'è obbligo sul formato. Potrebbe essere Unicode.

Questa classe converte una variabile **da string a sequenza di bytes e viceversa**. Chi trasmette deve convertire i dati da trasmettere in un vettore di bytes ASCII, mentre chi riceve provvede a fare la conversione inversa . Variabili di altro tipo devono prima essere convertite in stringa.

- **Encoding.ASCII.GetBytes** converte una stringa in un vettore di bytes ASCII
- **Encoding.ASCII.GetString** converte un vettore di bytes in stringa.

```
Byte [] buffer = Encoding.ASCII.GetBytes("salve mondo");
string s = Encoding.ASCII.GetString(buffer, 0, buffer.Length);
```

Istanza di un Client UDP

```
Socket sockId = New Socket(AddressFamily.INet, SocketType.Dgram, ProtocolType.Udp);
IPAddress ip = cercaIP(txtIpServer.Text);
IPEndPoint binary = New IPEndPoint(ipServer, txtPortaServer.Text);
```

Istanza di un Server UDP

```
Socket sockId = New Socket (AddressFamily.INet, SocketType.Dgram, ProtocolType.Udp);
IPAddress ip = cercaIP();
IPEndPoint binary = New IPEndPoint (ip, txtPorta.Text);
sock.Bind((EndPoint) binary);
```

Funzionalità di un Server

Un server, normalmente, **non** prevede intrazioni con l'utente, ma consiste tipicamente in un servizio in background che riceve richeste da un client e invia le relative risposte (ad es la pagina HTML richiesta). Gli unici tasti presenti su un server sono di solito **Avvia** e **Arresta**. Per rimanere in ascolto sulla porta indicata, il server deve eseguire un loop infinito, dunque **bloccante** nei confronti dell'interfaccia grafica (dove ci sono almeno i pulsanti Avvia Arresta). A tal fine è consigliato eseguire il ciclo di ascolto in un **thread separato** rispetto all'interfaccia grafica.

Nel caso invece in cui si voglia eseguire una **chat** oppure un **gioco di rete** in cui entrambi i terminali hanno esigenza di inserire delle informazioni in modo interattivo, esitono due possibile approcci.:

- Approccio **Client – Server** in cui i due client che devono comunicare inviano entrambi i dati al server indicando le coordinate del vero destinario del messaggio (es username). Il server rintracciato il destinatario all'interno di un elenco, provvede ad inoltrare il messaggio. Approccio oneroso. Numero arbitrario di client
- Approccio **Peer to Peer** in cui si elimina la presenza di un server ma in cui entrambi i terminali sono dotati contemporaneamente di una componente client e di una componente server ed ognuno può inviare dati all'altro quando vuole. **Soluzione più semplice** da realizzare nel caso in cui si abbiano due soli terminali.

La classe clsSockets

La classe seguente, pur essendo prevista per operare in ambiente **client – server**, implementa sia le funzionalità del server sia quelle del client, in modo da semplificare al massimo la scrittura dell'interfaccia utente.

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

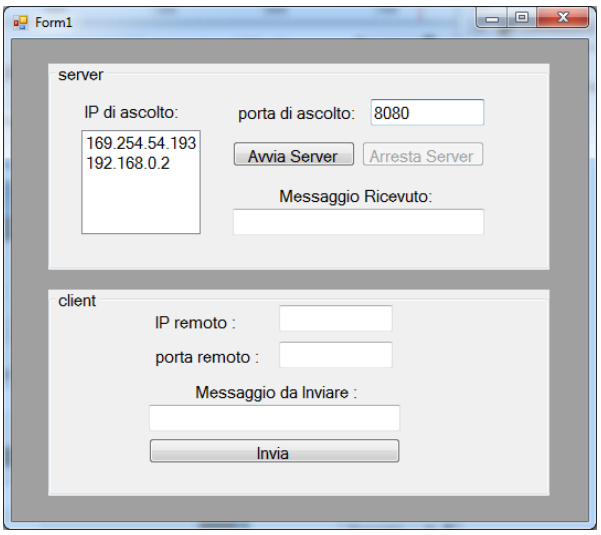
```
public class clsMessaggio  {
     public string ip;
     public int port;
     public string messaggio;
     public override string ToString () {
          return this.ip + ":" + this.porta + " - " + this.msg; }
}

public delegate void datiRicevutiEventHandler(clsMessaggio msg);

public class clsSocket : IDisposable  {
  public static int MAX_ETH = 1460;
  private bool server;        // true=server, false = client
  private Socket sockId;
  private EndPoint binary;  // EndPoint e IPEndPoint = sockaddr e sockaddr_in
  private Thread threadAscolta;
  public event datiRicevutiEventHandler datiRicevutiEvent;

  public clsSocket(IPAddress ip, int porta, bool server)  {
       this.server=server;
       sockId = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
                                                    ProtocolType.Unspecified);
       // Nel caso SERVER binary rappresenta IP e PORTA di ascolto
       // Nel caso CLIENT binary rappresenta IP e PORTA del server a cui connettersi
       binary = new IPEndPoint(ip, porta); // upcasting: binary è un semplice EndPoint
       if(server)
            sockId.Bind(binary);
  }

  public void serverRicevi()  {     // Salva dentro binary le coordinate del client
       int nBytesRicevuti;
       byte[] bufferRX = new byte [MAX_ETH];
       while (true)   {
            nBytesRicevuti = sockId.ReceiveFrom(bufferRX, MAX_ETH, 0, ref binary);
            string messaggio = Encoding.ASCII.GetString(bufferRX, 0, nBytesRicevuti);
            clsMessaggio msg = new clsMessaggio();
            msg.messaggio = messaggio;
            msg.ip = ((IPEndPoint)binary).Address.ToString();
            msg.porta = Convert.ToUInt16(((IPEndPoint)binary).Port);
            datiRicevutiEvent(msg);   // Genera Evento
       }
  }

  public void avviaServer()  {
     if (threadAscolta == null)   {
          threadAscolta = new Thread(serverRicevi);
          threadAscolta.Start();
          // aspetta finchè il thread non è partito
          while (!threadAscolta.IsAlive);
     }
     // il thread non riesce ad andare in 'suspended' perchè è bloccato su 'ricevi'
     else if (threadAscolta.ThreadState == ThreadState.SuspendRequested )
          threadAscolta.Resume();
  }
```

Attenzione che **Socket** è una classe di sistema !!

```
public void arrestaServer()  {
      if (threadAscolta.ThreadState == ThreadState.Running)
          threadAscolta.Suspend();
}

public clsMessaggio clientRicevi() {
      int nBytesRicevuti;
      byte[] bufferRX = new byte[MAX_ETH];
      nBytesRicevuti = sockId.ReceiveFrom(bufferRX, MAX_ETH, 0, ref binary);
      string s = Encoding.ASCII.GetString(bufferRX, 0, nBytesRicevuti);
      clsMessaggio msg = new clsMessaggio();
      msg.messaggio = s;
      msg.ip = ((IPEndPoint)binary).Address.ToString();
      msg.porta = Convert.ToUInt16(((IPEndPoint)binary).Port);
      return msg;
}

public void invia(string messaggio)  {
     byte[] bufferTX;
     bufferTX = Encoding.ASCII.GetBytes(messaggio);
     sockId.SendTo(bufferTX, bufferTX.Length, 0, binary);
}

public void Dispose()  {
    // 1) Ripulisco i buffer
     sockID.Shutdown(SocketShutdown.Both);
    // 2) Chiudo il socket
     sockID.Close();
    // 3) Termino il thread
     if (server && threadAscolta != null)   {
         if (threadAscolta.ThreadState == ThreadState.SuspendRequested)
              threadAscolta.Resume();
         threadAscolta.Abort();
     }
```

L'ordine **DEVE** essere quello indicato. Facendo prima abort e dopo close, UDP funziona lo stesso, ma TCP **non** riesce più a chiudere.

L'interfaccia Grafica



NOTA: Un thread può accedere ai Controlli della Form, ma in **SOLA LETTURA.** Per la scritura bisogna far ricorso a **`this.Invoke`**.

Questo in realtà è soltanto un problema del debugger, che da un thread non riesce a scriere i dati di un altro thread. Lanciando l'eseguibile da bin/debug il problema non si pone.
Resta però il fatto che per poter debuggare occorre utilizzare this.inoke().

```
public delegate void aggiornaGraficaEventHandler(clsMessaggio msg)

  private void Form1_Load(object sender, EventArgs e)  {
       clsAddress.cercaIP();
       lstIpServer.DataSource = clsAddress.ipVett;
       abilitaPulsantiAvvio(true);
  }
  void abilitaPulsantiAvvio(bool ok) {
        btnAvviaServer.Enabled = ok;
        btnArrestaServer.Enabled = !ok;
  }
  private void frmPrincipale_FormClosing( )  {
       if (serverSock != null)
              serverSock.Dispose();
  }

   /* ************************     CODICE LATO SERVER    ********************** */
   /* Il server aspetta in background una richiesta dal client e gli risponde      */

  private clsSocket serverSock;
  private void btnAvviaServer_Click(object sender, EventArgs e)  {
    try  {
       if (serverSock == null)  {
         IPAddress ip = clsAddress.ipVett[lstIpServer.SelectedIndex];
         serverSock = new clsSocket(true, Convert.ToInt(txtPortaServer.Text), ip);
         serverSock.datiRicevutiEvent += new datiRicevutiEventHandler(datiRicevuti);
       }
       serverSock.avviaServer();
       abilitaPulsantiAvvio(false);
       lblStato.Text = "RUNNING...";
    }
    catch (Exception ex)  {
              MessageBox.Show("Errore avvio server \n " + ex.Message);
    }
  }

  private void btnArrestaServer_Click(object sender, EventArgs e)  {
      serverSock.arrestaServer();
      abilitaPulsantiAvvio(true);
      lblStato.Text = "STOPPED...";
  }

  public void datiRicevuti(clsMessaggio msg)  {
```

Nota this.invoke non è asincrona ma bloccante, nel senso che non termina fino a quando la procedura chiamata non viene completamente eseguita. Se però clientSock è in attesa di una risposta dal server, l'interfaccia grafica è bloccata fino a quando tale risposta non arriva, per cui aggiornaGrafica non può essere eseguita fino a quando tale risposta non arriva.

IN PRATICA, quando si simula tramite un solo PC bisogna fare PRIMA INVIA e DOPO INVOKE

```
    serverSock.invia("ACK");
    aggiornaGraficaEventHandler fn = new aggiornaGraficaEventHandler(aggiornaGrafica);
    this.Invoke(fn, msg );  }

  public void aggiornaGrafica(clsMessaggio msg)  {   // solo per debugger
       txtMsgServer.Text = msg.ToString();   }
```

```
/* *************************     CODICE LATO CLIENT    ************************* */
/* Il client invia un msg al server e, nella stessa procedura, attende una risposta*/
private void btnInviaMess_Click(object sender, EventArgs e)  {
     clsSocket clientSock;
     IPAddress ip = clsAddress.cercaIP(txtIpRemoto.Text);
     clientSock = new clsSocket(false, Convert.ToInt32(txtPortaRemoto.Text), ip);
     clientSock.invia(txtInvia.Text);
     clsMessaggio msg = clientSock.clientRicevi();
     MessageBox.Show(msg.msg);
     clientSock.Dispose();
}
```

Interfacciamento con una applicazione ANSI C

- Concatenare alla stringa trasmessa mediante SendTo un carattere **NullChar** finale
  `bufferTx = Encoding.ASCII.GetBytes(txtMsg.Text& ControlChars.NullChar)`
- Tralasciare l'ultimo carattere all'interno del buffer ricevuto con ReceiveFrom
  `risposta = Encoding.ASCII.GetString(bufferRx, 0, bytesRicevuti-1)`

TCP Socket

Modifiche da apportare alla classe clsSocket

- Bisogna istanziare un socket di tipo **stream**
- Dopo il Bind occorre eseguire il metodo `.Listen(20)` per il dimensionamento della coda di richieste
- All'inizio del ciclo while della procedura **Ascolta** bisogna eseguire il metodo **Accept** che restituisce una Connection ID dedicata alla comunicazione con il client, mentre il processo principale rimane in ascolto. **`Socket connID`** deve essere dichiarata **esternamente** in modo da essere visibile ai vari metodi

```
 connID = sockID.Accept();
nBytesRicevuti = connID.Receive(bufferRX, maxETH, 0);
Messaggio.ip = ((IPEndPoint)connID.RemoteEndPoint).Address.ToString();
Messaggio.porta = Convert.ToUInt(((IPEndPoint)connID.RemoteEndPoint).Port);
```

  La **chiusura** della connessione connID viene fatta dal server stesso dopo aver inviato la risposta al client (procedura **`serverInvia`**). Anche **`clientRicevi`** al termine potrebbe chiudere la propria connessione, (operazione fatta da btnInvia del main dove dovrebbe quindi essere eliminata).

- Riguardo alla procedura **Invia** ocorre scrivere due diverse procedure :

```
public void serverInvia(string str)   {
     byte[]  bufferTX = Encoding.ASCII.GetBytes(str);
     connID.Send(bufferTX, bufferTX.Length, 0);
     connID.Shutdown(SocketShutdown.Both)
     connID.Close()
}

public void clientInvia(string str)   {
     byte[] bufferTX = Encoding.ASCII.GetBytes(str);
     sockID.Connect(binary);
     sockID.Send(bufferTX, bufferTX.Length, 0);
}
```

- Nel metodo .**dispose(),** il server è bloccato sull'Accept, per cui lo shutdown **NON** può essere eseguito.

```
if(!server)    sockID.Shutdown(SocketShutdown.Both);
```