Entendiendo la programación de Sockets con GTK# y .NET

por Martín A. O Márquez <xomalli@gmail.com>

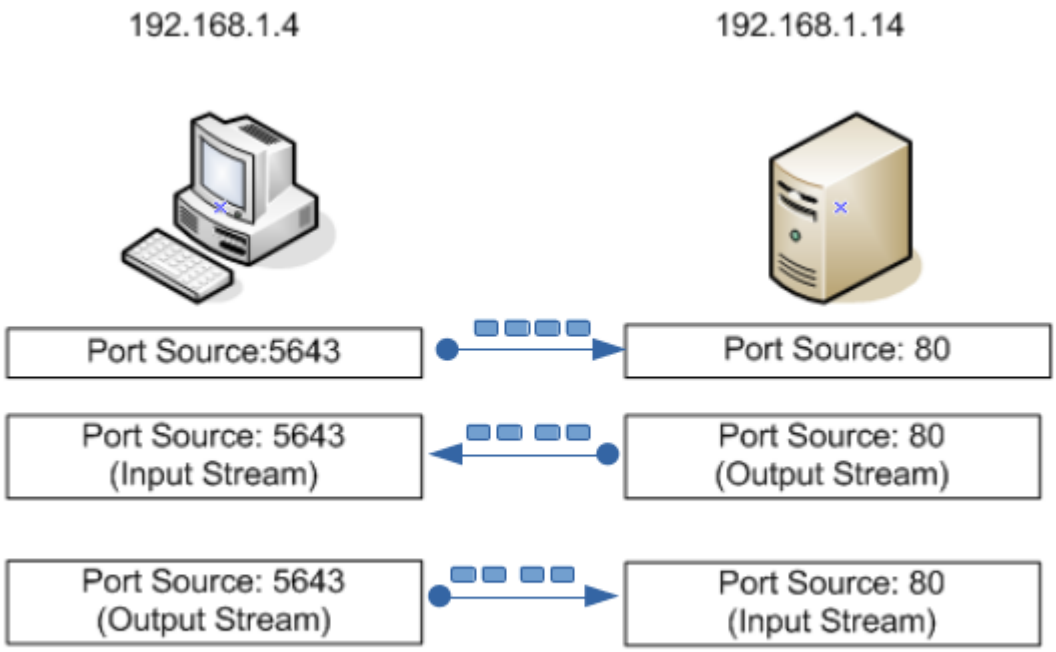
¿Qué son los sockets?

Los **Sockets** o mejor dicho los **Berkeley Sockets (BSD IPC)** son la combinación exclusiva de una dirección IP y un numero de puerto TCP que habilita a los servicios (procesos en ejecución) de una computadora el poder intercambiar datos a través de la red. Los servicios de red utilizan los sockets para comunicarse entre los equipos remotos. Por ejemplo el siguiente comando:

```
$ telnet 192.168.1.14 80
```

Este comando solicita una conexión desde un puerto aleatorio en el cliente (por ejemplo: 5643) al puerto 80 en el servidor (que es el puerto asignado para el servicio HTTP). Con la siguiente figura (fig 1) se ilustra a detalle este esquema denominado cliente-servidor.

Fig 1 Una comunicación utilizando números de puerto para transmitir datos.



Históricamente los **sockets** son una API (Application Programming Interface) de programación estándar utilizada para construir aplicaciones de red que vienen desde el sistema UNIX BSD. Esta interface de programación para la capa 4 del modelo OSI (OSI Model Layer 4) permite a un programador tratar una conexión de red como un flujo de bytes que puede escribirse o leerse sin demasiada complejidad. Con un **socket** se pueden realizar siete operaciones básicas:

1. Conectarse a una máquina remota.
2. Enviar datos.
3. Recibir datos.
4. Cerrar una conexión.
5. Escuchar para los datos entrantes.
6. Aceptar conexiones desde maquinas remotas en el puerto enlazado.
7. Enlazarse a un puerto.

Los Sockets pueden ser orientados a la conexión (Stream Socket) o no (Message-based Socket).

Los **Stream Sockets** son ideales para transmitir grandes volúmenes de información de manera confiable. Una conexión **Stream Socket** se establece mediante el mecanismo three-hand shake de TCP, los datos se transmiten y cada paquete se revisa para asegurarse de la exactitud en la transmisión.

Los **Datagram Sockets** son apropiados para transferencias de datos cortas, rápidas y sin necesidad de un chequeo de errores. Los desarrolladores de aplicaciones los prefieren por ser rápidos y muy fáciles de programar.

Fig 2 Tipos de Socket

TIPO	PROTOCOLO	DESCRIPCIÓN
Stream	TCP	Se utiliza para transferir grandes volúmenes de datos de manera confiable.
Message Based	UDP	Se utiliza para transferir pequeñas cantidades de datos de forma rápida sin importar la confiabilidad.

El Framework .NET posee las clases de alto y bajo nivel que encapsulan la funcionalidad de un Socket (tanto TCP como UDP) para construir aplicaciones de red con relativa facilidad y sin preocuparse por

todo el intricado mecanismo de comunicación que necesitaría muchas líneas de código. La siguiente lista describe las clases principales:

- **NetworkStream:** Una clase derivada de la clase Stream representa el flujo de datos de entrada o de salida desde la red.
- **TcpClient:** Crea conexiones TCP de red para conectarse a un socket de servidor.
- **TcpListener:** Se utiliza para escuchar peticiones de red TCP.
- **UdpClient:** Crea conexiones UDP de red con posibilidad de multicasting.
- **Socket:** Es una clase de bajo nivel que envuelve a la implementación winsock, las clases TcpClient, TcpListener y UDPClient utilizan esta clase para sus operaciones, se puede afirmar que la clase Socket tiene las operaciones de estas clases más otras funcionalidades mucho más avanzadas y de más bajo nivel.

Pasos para la construcción de un Servidor TCP GTK#

Un servidor TCP siempre está ejecutándose de forma continua hasta que recibe una solicitud de conexión por parte de un cliente, cuando se recibe esta solicitud, el servidor establece una conexión con el cliente y utiliza dicha conexión para el intercambio de datos. Si los programas se comunican a través de TCP los datos que se procesan se envían y se reciben como flujo de bytes.

Para demostrar estos conceptos escribí dos programas: el de un servidor y el de un cliente TCP. Ambos utilizan una interfaz de usuario (GUI) en GTK# para comunicarse entre ellos mediante mensajes de texto.

Fig 3 Ejemplo de un servidor TCP con una GUI GTK#.

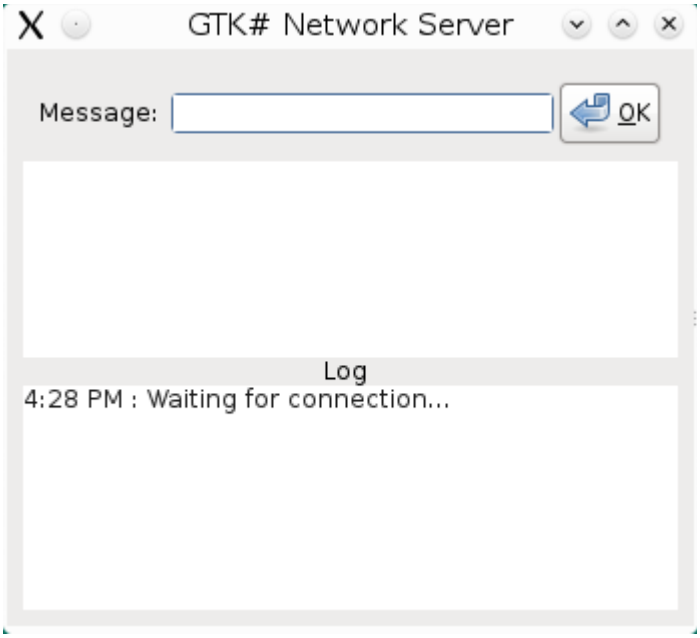
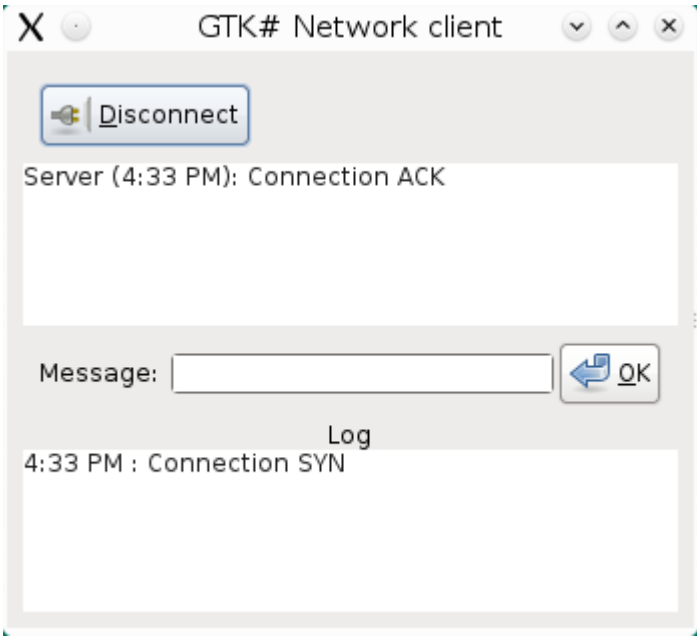


Fig 4 Ejemplo de un cliente TCP con una GUI GTK#.



Pasos para la construcción de un Servidor TCP GTK#

El proyecto del servidor TCP GTK# se compone de 2 clases:

1. La clase **MainWindowServer.cs** es la clase que construye la GUI del programa, maneja los eventos para enviar los mensajes al cliente y las excepciones o mensajes que el programa notifique.
2. La clase **Program.cs** es la clase principal que donde se ejecuta el servidor.

Para construir el servidor TCP se requieren de los siguientes pasos:

1) Crear un objeto **IPEndpoint** que asocia una dirección IP y un número de puerto.

```
IPEndPoint ipEndPoint = new IPEndPoint(IPAddress.Any,6000);
```

2) Crear un objeto **TcpListener** que reciba como argumento un objeto IPEndpoint. (Aquí el objeto TcpListener oculta la intricada programación de un **Server Socket** para una facilidad en la programación)

```
listener = new System.Net.Sockets.TcpListener(ipEndPoint);
```

3) Iniciar el objeto TcpListener para que escuche las peticiones.

```
listener.Start();
```

4) Se utilizar un ciclo para que el **Server Socket** escuche o espere indefinidamente hasta recibir una petición, cuando el servidor recibe la petición crea una conexión hacia el cliente y regresa un objeto **Socket** del ensamblado System.Net.Sockets.Socket.

```
connection = listener.AcceptSocket();
```

5) Se obtiene el flujo de comunicación del Socket.

```
System.Net.Sockets.NetworkStream socketStream =
new System.Net.Sockets.NetworkStream(connection);
```

6) Finalmente se asocia el flujo de comunicación del Server Socket con un escritor y un lector binario para transferir y recibir datos a través del flujo de comunicación.

```
using(writer = new BinaryWriter(socketStream))
{
using(BinaryReader reader = new BinaryReader(socketStream))
{
//the stream goes here
}
}
```

Es muy importante que una vez que se finaliza la comunicación con el cliente cerrar el flujo y la conexión mediante con el método Close de cada uno de los objetos.

```
socketStream.Close();
connection.Close();
```

El código fuente completo de la clase **MainWindowServer.cs**.

```
using System;
using Gtk;
using System.IO;
using System.Threading;
using System.Net;

namespace Samples.GtkNetworking
{
    public class MainWindowServer : Gtk.Window
    {
        VBox mainLayout = new VBox();
        HBox controlLayout = new HBox(false,2);
        Entry txtMsg = new Entry();
        Button btnSend = new Button(Stock.Ok);
        TextView txtChat = new TextView();
        TextView txtLog = new TextView();
        Label msgLabel = new Label("Message: ");
        System.Net.Sockets.Socket connection = null;
        BinaryWriter writer = null;
        System.Net.Sockets.TcpListener listener = null;
        Thread readThread = null;

        public MainWindowServer() : base(WindowType.Toplevel)
        {
            this.Title = "GTK# Network Server";
            this.SetDefaultSize(343, 288);
```

```
 this.DeleteEvent += new DeleteEventHandler(OnWindowDelete);
 this.btnSend.Clicked += new EventHandler(SendMessage);
 mainLayout.BorderWidth = 8;
 controlLayout.BorderWidth = 8;
          controlLayout.PackStart(msgLabel,false,true,0);
          controlLayout.PackStart(txtMsg,true,true,0);
          controlLayout.PackStart(btnSend,false,false,0);
          mainLayout.PackStart(controlLayout,false,true,0);
          mainLayout.PackStart(txtChat,true,true,0);
          mainLayout.PackStart(new Label("Log "),false,true,0);
          mainLayout.PackStart(txtLog,true,true,0);
          this.Add(mainLayout);
          this.ShowAll();
          readThread = new Thread(RunServer);
          readThread.Start();
      }

      protected void OnWindowDelete(object o, DeleteEventArgs args)
{
          //Terminate all
          System.Environment.Exit(System.Environment.ExitCode);
}

      protected void SendMessage(object o,EventArgs args)
      {
          try
          {
              writer.Write(ChatMessage(txtMsg.Text));
              txtChat.Buffer.Text += ChatMessage(txtMsg.Text);
              txtMsg.Text = string.Empty;
          }
          catch (System.Net.Sockets.SocketException error)
          {
              LogMessage("Error: " + error.Message);
          }
      }

      string ChatMessage(string msg)
      {
          return string.Format("Server ({0}): {1} {2}",
          DateTime.Now.ToShortTimeString(),
          msg,
          Environment.NewLine);
      }

      void LogMessage(string msg)
      {
          txtLog.Buffer.Text += string.Format("{0} : {1}{2}",
          DateTime.Now.ToShortTimeString(),
          msg,
          Environment.NewLine);
      }

      void RunServer()
      {
          int counter = 1;
          try
          {
              //step 1: create endpoint
```

```
                IPEndPoint ipEndPoint = new IPEndPoint(IPAddress.Any,6000);
                //Step 2: create TcpListener
                listener = new System.Net.Sockets.TcpListener(ipEndPoint);
                //Step 3: waits for connection request
                listener.Start();
                //Step 4: establish connection upon client request
                while(true)
                {
                    LogMessage("Waiting for connection...");
                    //step 5: accept an incoming connection
                    connection = listener.AcceptSocket();
                    //step 6: create the network stream object associated with
socket
                    //NOTE: Use the namespace to avoid conflicts with GTK.Socket
                    System.Net.Sockets.NetworkStream socketStream = new
System.Net.Sockets.NetworkStream(connection);
                    //step7: create the objects for transferring data across stream
                    using(writer = new BinaryWriter(socketStream))
                    {
                        using(BinaryReader reader = new BinaryReader(socketStream))
                        {
                            LogMessage(TcpFlags.SYN + " : " + counter);
                            //inform client that connection was ACK
                            writer.Write(ChatMessage(TcpFlags.ACK));
                            string theReply = "";
                            //read string data sent from client until receive the
FIN signal
                            do
                            {
                                try
                                {
                                //read the string sent to the server
                                theReply = reader.ReadString();
                                //display the message except the FIN
                                if(!theReply.Equals(TcpFlags.FIN))
                                    txtChat.Buffer.Text += theReply;
                                    else
                                    {
                                        LogMessage("Received " + TcpFlags.FIN);
                                        LogMessage("Send " + TcpFlags.FIN + " to
Client");
                                        writer.Write(TcpFlags.FIN);
                                    }
                                }
                                catch (System.Exception)
                                {
                                    break;
                                }
                            } while (theReply != TcpFlags.FIN
                    && connection.Connected);
                    //Close connection
                    LogMessage("Close connection");
                        }
                    }
                    socketStream.Close();
                    connection.Close();
                    ++counter;
                }
            }
```

```
            catch (System.Exception ex)
            {
                LogMessage("Ex " + ex.ToString());
            }
        }

    }
}
```

El código fuente de la clase **Program** (Server).

```
using System;
using Gtk;

namespace Samples.GtkNetworking
{
 class MainClass
 {
  public static void Main(string[] args)
  {
   Application.Init();
   MainWindowServer win = new MainWindowServer();
   win.Show();
   Application.Run();
  }
 }
}
```

Pasos para la construcción de un cliente TCP GTK#

El proyecto del cliente TCP GTK# se compone de 2 clases:

1. La clase **MainWindow.cs** es la clase que construye la GUI del cliente, maneja los eventos para recibir y enviar los mensajes al servidor, y mostrar las excepciones o mensajes que ocurran.
2. La clase **Program.cs** es la clase principal del cliente

Para construir el cliente TCP se requieren de los siguientes pasos, algunos son idénticos a los que se escribieron para el servidor:

1) Crear un objeto **IPEndpoint** que asocia la dirección IP y el número de puerto del servidor, generalmente estos datos son fijos ya que los servidores se configuran para tenerlos de manera estática. (en este ejemplo utilice una sola máquina como servidor y como cliente)

```
IPEndPoint localEndPoint =
new IPEndPoint(IPAddress.Loopback,6000);
```

2) Se crea un Socket de cliente y se conecta al puerto del servidor.

```
client.Connect(localEndPoint);
```

3) Se obtiene el flujo de comunicación del Socket

```
output = client.GetStream();
```

4) Se crean los objetos lector y escritor para trabajar con el flujo de comunicación.

```
   using(writer = new BinaryWriter(output))
   {
    using(reader = new BinaryReader(output))
    {
//the stream goes here
    }
  }
```

Finalmente cuando se termina la comunicación con el servidor, se cierra el flujo de datos y la conexión con el método *Close* de cada objeto.

```
    output.Close();
    client.Close();
```

El código fuente de la clase **MainWindow (cliente)**

```
using System;
using Gtk;
using System.Net.Sockets;
using System.Net;
using System.IO;
using System.Threading;

namespace Samples.GtkNetworking
{
 public class MainWindow : Gtk.Window
 {
  VBox mainLayout = new VBox();
  HBox controlLayout = new HBox(false, 2);
  HBox connectedLayout = new HBox(false,2);
  Entry txtMsg = new Entry();
  Button btnSend = new Button(Stock.Ok);
  Button btnDisconnect = new Button(Stock.Disconnect);
  TextView txtChat = new TextView();
  TextView txtLog = new TextView();
  Label msgLabel = new Label("Message: ");
  TcpClient client = new TcpClient();
  NetworkStream output = null;
  BinaryWriter writer = null;
  BinaryReader reader = null;
  string message = "";
  Thread readThread = null;


  public MainWindow() : base(WindowType.Toplevel)
  {
   this.Title = "GTK# Network client";
   this.SetDefaultSize(343, 288);
   this.DeleteEvent += new DeleteEventHandler(OnWindowDelete);
   this.btnSend.Clicked += new EventHandler(SendMessage);
   this.btnDisconnect.Clicked += new EventHandler(SendDisconnect);
   mainLayout.BorderWidth = 8;
```

```
 connectedLayout.BorderWidth = 8;
 connectedLayout.PackStart(btnDisconnect,false,false,0);

 controlLayout.BorderWidth = 8;
 controlLayout.PackStart(msgLabel,false,true,0);
 controlLayout.PackStart(txtMsg,true,true,0);
 controlLayout.PackStart(btnSend,false,false,0);
 mainLayout.PackStart(connectedLayout,false,true,0);
 mainLayout.PackStart(txtChat,true,true,0);
 mainLayout.PackStart(controlLayout,false,true,0);
 mainLayout.PackStart(new Label("Log"),false,true,0);
 mainLayout.PackStart(txtLog,true,true,0);
 this.Add(mainLayout);
 this.ShowAll();
 readThread = new Thread(new ThreadStart(RunClient));
 readThread.Start();
}

protected void OnWindowDelete(object o, DeleteEventArgs args)
{
 System.Environment.Exit(System.Environment.ExitCode);
}

protected void SendDisconnect(object o,EventArgs args)
{
 if(client.Connected)
 {
  try
  {
  //Send disconnected signal
  writer.Write(TcpFlags.FIN);
  }
  catch(SocketException error)
  {
  LogMessage("Error " + error.Message);
  }
 }
 else
  LogMessage("You are not connected");
}

protected void SendMessage(object o,EventArgs args)
{
 try
 {
  writer.Write(ChatMessage(txtMsg.Text));
     txtChat.Buffer.Text += ChatMessage(txtMsg.Text);
  txtMsg.Text = string.Empty;
 }
 catch (SocketException error)
 {
  LogMessage("Error " + error.Message);
 }
}

String ChatMessage(string msg)
{
 return string.Format("Client ({0}): {1} {2}",
 DateTime.Now.ToLongTimeString(),
```

```
    msg,
    Environment.NewLine);
   }

   void LogMessage(string msg)
         {
             txtLog.Buffer.Text += string.Format("{0} : {1}{2}",
             DateTime.Now.ToShortTimeString(),
             msg,
             Environment.NewLine);
         }

   protected void RunClient()
   {
    try
    {
     //step 1 create a local endpoint
     IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Loopback,6000);
     //step 2 create a socket client and connect
     client.Connect(localEndPoint);
     LogMessage(TcpFlags.SYN);
     //step 3 get the network stream associated with tcpclient
     output = client.GetStream();
     //step 4 create the objects for writing and reading across the stream
     using(writer = new BinaryWriter(output))
     {
     using(reader = new BinaryReader(output))
     {
     //Receive message until receive the FIN signal
     do
     {
      try
      {
       message = reader.ReadString();
       if(!message.Equals(TcpFlags.FIN))
        txtChat.Buffer.Text += message;
       else
        LogMessage("Received " + TcpFlags.FIN + " from server");
      }
      catch (System.Exception error)
      {
       LogMessage("Error: " + error.Message);
      }
     } while (message != TcpFlags.FIN);
     LogMessage("Closing connection...");
     }
     }
     output.Close();
     client.Close();
    }
    catch (System.Exception ex)
    {
     LogMessage(" Ex " + ex.Message);
    }
   }
  }
}
```

El código fuente de la clase Program (cliente)

```
using System;
using Gtk;

namespace Samples.GtkNetworking
{
 class MainClass
 {
  public static void Main(string[] args)
  {
   Application.Init();
   MainWindow win = new MainWindow();
   win.Show();
   Application.Run();
  }
 }
}
```

La clase TcpFlags

Ambos proyectos utilizan la clase TcpFlags, la cual pretende ilustrar básicamente como son las banderas TCP (aquí más detalles de las TCP Flags) que utiliza la capa de transporte (layer 4) para manejar la comunicación entre dos máquinas.

El código fuente de la clase TcpFlags

```
using System;

namespace Samples.GtkNetworking
{
    public class TcpFlags
    {
        public const string FIN = "Connection FIN";
        public const string ACK = "Connection ACK";
        public const string SYN = "Connection SYN";
    }
}
```

A continuación unas imágenes del cliente y servidor comunicándose entre si.

Fig 5 Enviando un mensaje desde el cliente al servidor.

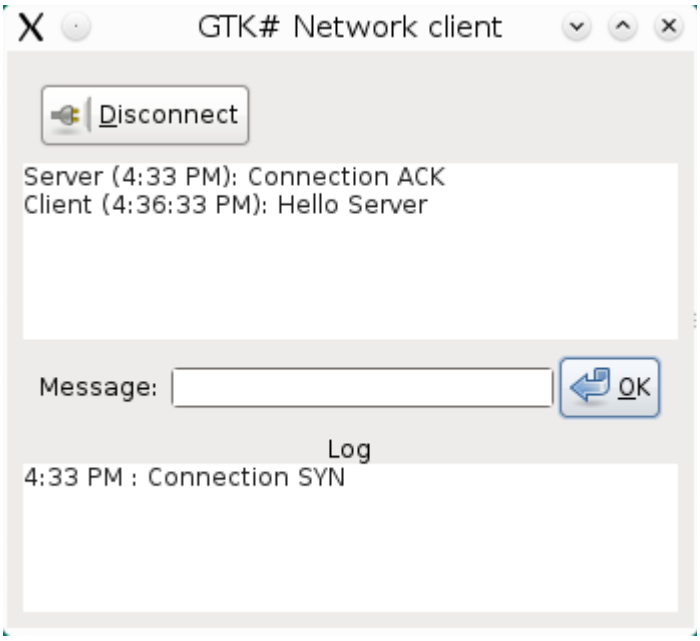


Fig 6 Recibiendo el mensaje del cliente.

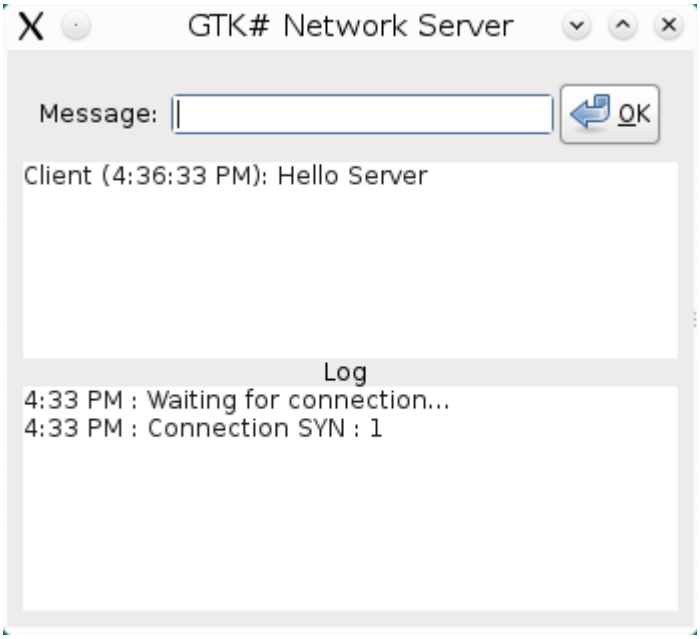


Fig 7 Enviándole un mensaje al cliente desde el servidor.

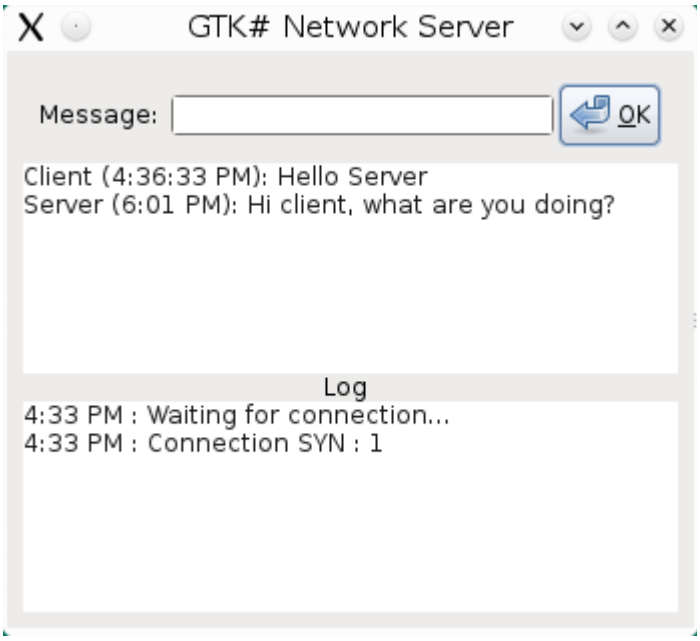


Fig 8 Desconectándose del servidor.

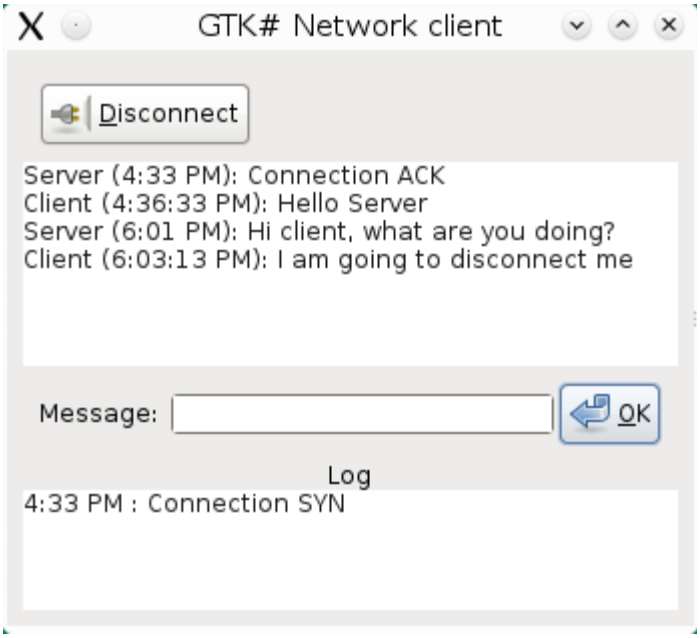
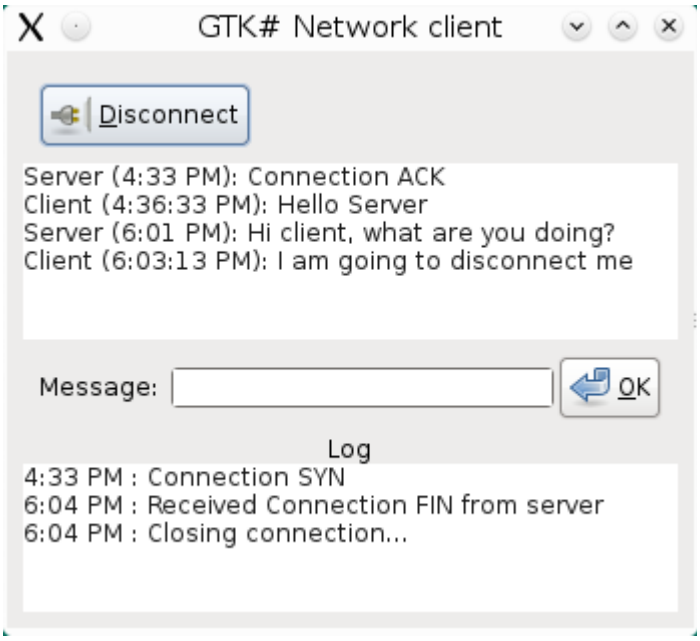


Fig 9 Recibiendo la señal de desconexion del Server.



Descarga el código fuente de la clase TcpFlags.cs

Descarga la solución del servidor TCP GTK#.

Descarga la solución del cliente TCP GTK#.