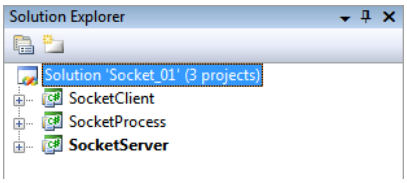
ALGORITMO DE SEGURIDAD DIFFIE-HELLMAN Y PROGRAMACIÓN CON SOCKET

POR CHRISTIAN CHAMORRO

Para esto, he desarrollado una aplicación en Visual Studio con tres proyectos, como se presenta en la siguiente ilustración:



`SocketServer` es un proyecto ejecutable (EXE) que se ejecutará del lado del servidor, en el cual se definirán los números primos y serán compartidos por demanda usando sockets.

`SocketClient` es un proyecto ejecutable (EXE) que se ejecutará del lado del cliente. Este recibirá las claves e intentará validar una contraseña usando el algoritmo de seguridad.

`SocketProcess` es un proyecto de biblioteca de vínculo dinámico que encierra métodos y procedimientos comunes a los proyectos anteriores.

Para que la aplicación funcione he de definir un IP para cada entidad y un puerto. Para que esto sea variable, cada proyecto ejecutable tiene un archivo llamado `app.config` en el cual se definen los parámetros generales que han de usarse en el programa.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" type="System.Configuration.UserSettingsGroup, System, Version=2.0.0.0, Culture
=neutral, PublicKeyToken=b77a5c561934e089" >
            <section name="Socket_01.Properties.Settings" type="System.Configuration.ClientSettingsSection, System,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" allowExeDefinition="MachineToLocalUser"
requirePermission="false" />
        </sectionGroup>
    </configSections>
    <userSettings>
        <Socket_01.Properties.Settings>
            <setting name="privateKey" serializeAs="String">
                <value>202</value>
            </setting>
            <setting name="port" serializeAs="String">
                <value>31999</value>
            </setting>
            <setting name="ipLocal" serializeAs="String">
                <value>192.168.1.6</value>
            </setting>
        </Socket_01.Properties.Settings>
    </userSettings>
</configuration>
```

SocketServer – `app.config`

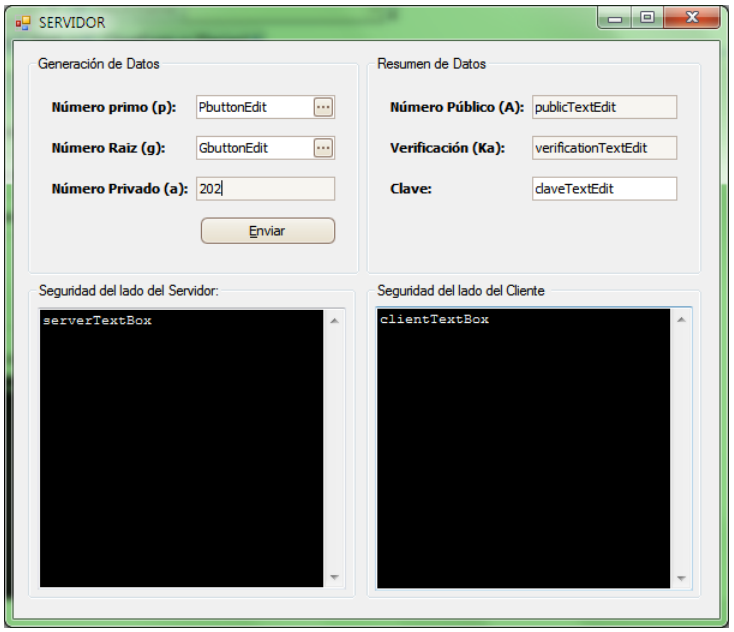
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" type="System.Configuration.UserSettingsGroup, System, Version=2.0.0.0, Culture
=neutral, PublicKeyToken=b77a5c561934e089" >
            <section name="SocketClient.Properties.Settings" type="System.Configuration.ClientSettingsSection, System,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" allowExeDefinition="MachineToLocalUser"
requirePermission="false" />
        </sectionGroup>
    </configSections>
    <userSettings>
        <SocketClient.Properties.Settings>
            <setting name="port" serializeAs="String">
                <value>31999</value>
            </setting>
            <setting name="ipServer" serializeAs="String">
                <value>192.168.1.6</value>
            </setting>
            <setting name="ipLocal" serializeAs="String">
                <value>192.168.1.7</value>
            </setting>
        </SocketClient.Properties.Settings>
    </userSettings>
</configuration>
```

SocketClient - `app.config`

Como pueden ver en las imágenes anteriores. Se configura una dirección IP a cada equipo, y se define un puerto.

En este caso, y dado que el puerto es el mismo, esta aplicación sólo puede funcionar en terminales diferentes. Si se desea que funcione en la misma terminal, se deberán hacer ajustes para que el servidor envíe datos a través de un puerto y reciba datos de las terminales a través de otro puerto. Esto puede ser una gran idea para que ustedes se lo propongan como práctica a partir de este artículo.

Aplicación SocketServer



Como pueden observar, en la siguiente pantalla se definen los datos que corresponden al lado del servidor en el intercambio de las llaves. Además, he colocado una pantalla de verificación del lado del cliente para saber en qué momento se reciben los paquetes.

Voy a dar una breve explicación de cada control utilizado en este formulario:

`PbuttonEdit`: Este campo es un *button edit*, el cual lo pueden encontrar en *suites* de controles para Visual Studio, como DevExpress, ComponentOne o Infragistic. La única funcionalidad extra de un campo de texto, es el botón incrustado que tiene al lado derecho. Este control lo he utilizado para generar el número primo de referencia (p).

`GbuttonEdit`: Esta campo también es un *button edit*, y lo he utilizado para generar el número primo denominado generador (g).

`privateTextEdit`: Este campo es de tipo *textbox* y refleja el valor de la clave privada del lado del servidor (a).

`publicTextEdit`: Este campo también es de tipo *textbox* y refleja el valor de la clave pública del lado del servidor (A).

`verificacionTextEdit`: Este campo también es tipo *textbox*, y refleja el valor de la clave de comparación luego de recibida la clave pública del cliente.

`claveTextEdit`: Este campo también es tipo *textbox*, y se utiliza para escribir la clave/contraseña que luego se va a verificar con el cliente.

`serverTextBox` y `clientTextBox`: He colocado estas cajas de texto para monitorear las actividades de las aplicaciones del lado del cliente y del servidor.

A continuación, explicaré rápidamente cada método del formulario:

El primer método es el constructor de la clase, en el cual únicamente agregué la modificación de la clave privada por el valor contenido en el archivo de configuración de la página anterior, denominado `privateKey`.

```
public ServerForm()
{
        InitializeComponent();
        aTextEdit.Text = global::Socket_01.Properties.Settings.Default.privateKey;
}
```

Luego, he programado el método para actualizar las cajas de texto de seguimiento. Lo hacemos con argumentos para que pueda ser aplicado a cualquier control tipo *TextBox* en toda el programa.

```
protected void CargarLog(TextBox tb, string s)
{
        tb.Text += "" + s + Environment.NewLine;
        tb.SelectionStart = tb.Text.Length;
        tb.ScrollToCaret();
}
```

Luego he programado, en un método sencillo, el cálculo del número primo de referencia, y la generación de la clave pública en caso de estar ambos números primos.

Si se han fijado, he creado un método llamado `NumeroPrimo` en el proyecto `SocketProcess` para realizar este cálculo de manera ordenada y con funciones divididas, tal a como lo manda la programación no estructurada.

A este método le paso como argumentos dos números que representan el mínimo (10,000,000) y el máximo (99,999,999) de referencia para la generación del número primo.

```
private void PbuttonEdit_ButtonPressed(object sender, DevExpress.XtraEditors.Controls.ButtonPressedEventArgs e)
{
        PbuttonEdit.EditValue = SocketProcess.primos.NumeroPrimo(10000000, 99999999);
        CargarLog(serverTextBox, "Número primo 'p' generado!");

        if (GbuttonEdit.Text != "")
                GenerarClavePublica();
}
```

Luego, he realizado la programación en un método similar al botón del otro número primo (generador – g), considerando números primos más pequeños. Uso esta vez 1,000 como mínimo y 9,999,999 como máximo.

```
private void GbuttonEdit_ButtonPressed(object sender, DevExpress.XtraEditors.Controls.ButtonPressedEventArgs e)
{
        GbuttonEdit.EditValue = SocketProcess.primos.NumeroPrimo(1000, 9999999);
        CargarLog(serverTextBox, "Número raiz 'g' generado!");

        if (PbuttonEdit.Text != "")
                GenerarClavePublica();
}
```

Luego, he programado el método para la aplicación de la fórmula para obtener la clave pública (la cual se puede encontrar en la primera parte de este artículo).

En este método, como pueden observar, existe un método en una clase del proyecto `SocketProcess` llamado `Residuo`, el cual extrae el residuo o módulo de la división.

Se preguntarán, ¿por qué no usé el método implícito del .NET Framework – `System.Math.DivRem` – o la instrucción heredada del lenguaje C – a % b – para obtener este resultado? La respuesta es muy sencilla, ya que este método o instrucción no fue diseñada para manejar números tan grandes como los que exige el método para proporcionar alta seguridad, fueron diseñados para cómputos simples y rápidos.

```
private void GenerarClavePublica()
{
        ulong a = Convert.ToUInt64(aTextEdit.Text);
        ulong p = Convert.ToUInt64(PbuttonEdit.EditValue);
        ulong g = Convert.ToUInt64(GbuttonEdit.EditValue);

         publicTextEdit.Text = SocketProcess.matematicas.residuo(g, a, p).ToString();
        CargarLog(serverTextBox, "Clave pública 'A' generada!");
}
```

Al final del artículo incluiré un enlace a la web de la revista para que puedan descargar los códigos fuentes del proyecto completo, incluyendo los métodos no explicados en este documento, de esta forma podrán realizar las pruebas en los escenarios que Ustedes mismos puedan idear y crear.

El siguiente método es el envío de datos a través del socket, el cual se explica por sí solo, con la excepción del encapsulamiento de los métodos `Preparar` y `EnviarUDP`, los cuales explicaré y detallaré luego de la siguiente aclaración.

Según el modelo de red TCP/IP, existen dos tipos de conexiones: las conexiones seguras, basadas en el protocolo de transporte TCP, y las conexiones no seguras, basadas en el protocolo UDP. Las conexiones definen un esquema de seguridad en dependencia de la orientación que tengan de mantener una conexión fija y exclusiva entre los nodos, de forma que las redes basadas sobre TCP son orientadas a conexión, y las UDP son no orientadas a conexión.

NOTA: Si desean conocer más al respecto, les recomiendo buscar los artículos de redes en este ejemplar, o dirigirse a alguna fuente online para corroborar los conceptos.

Dicho esto, ya Ustedes habrán notado que usaremos, para efectos académicos y sólo de estudio, un entorno no orientado a conexión, pensando en aplicar difusión de claves públicas en un esquema de múltiples terminales.

Bien, el método `Preparar`, cuyo código muestro a continuación, define una instancia de la clase `IPEndPoint`, la cual contiene información de la dirección IP y del puerto que se utilizará en la comunicación entre aplicaciones.

```
public static void Preparar(IPAddress ip, Int32 port, String LocalIP)
{
        if (_localEndPoint == null)
        {
                IPAddress _ip = IPAddress.Parse(LocalIP);
                _localEndPoint = new IPEndPoint(_ip, port);
                _Socket.EnableBroadcast = true;
                _Socket.Bind(_localEndPoint);
        }

        _IPEndPoint = new IPEndPoint(ip, port);
}
```

Adicionalmente, se habilita el broadcast de paquete, lo que permite que el socket esté capacitado para enviar paquetes a diferentes direcciones multicast o a una dirección de broadcast. De igual forma, permite recibir paquetes de diferentes direcciones, siempre y cuando, los paquetes sean dirigidos al equipo que administra el socket, y al puerto en el cual éste escucha.

```
private void enviarButton_Click(object sender, EventArgs e)
{
        ulong p = Convert.ToUInt64(PbuttonEdit.Text);
        ulong g = Convert.ToUInt64(GbuttonEdit.Text);
        ulong A = Convert.ToUInt64(publicTextEdit.Text);

        if (p == 0 || g == 0 || A == 0)
        {
                CargarLog(serverTextBox, "Debe generar los datos!");
                return;
        }

        CargarLog(serverTextBox, "Enviando datos (p, g, A) a clientes...");

        try
        {
                String datos = p.ToString() + "|" + g.ToString() + "|" + A.ToString();
                Byte[] Datos = System.Text.Encoding.ASCII.GetBytes(datos);

                Int32 port = global::Socket_01.Properties.Settings.Default.port;
                //IPAddress ip = IPAddress.Parse("192.168.43.48");
                IPAddress ip = IPAddress.Broadcast;

                String LocalIP = global::Socket_01.Properties.Settings.Default.ipLocal;
                SocketProcess.socket.Preparar(ip, port, LocalIP);
```

```
                SocketProcess.socket.EnviarUDP(Datos);

                CargarLog(serverTextBox, "Envío de datos satisfactorio!");
        }
        catch (Exception exc)
        {
                CargarLog(serverTextBox, "ERROR: " + exc.Message);
        }
}
```

El otro llamado al cual hice referencia en el método anterior, es `EnviarUDP`, el cual tiene como única función enviar datos a través de la instancia de `IPEndPoint` definida en la preparación del socket, explicada anteriormente. A este método le paso como argumento un arreglo de bytes con los datos a enviar (puede ser texto, imágenes o archivos) y éste los canaliza a través de la terminal indicada en `IPEndPoint`. Véase a continuación su código fuente:

```
public static bool EnviarUDP(Byte[] datos)
{
        try
        {
                _Socket.SendTo(datos, _IPEndPoint);
        }
        catch(Exception exc)
        {
                return false;
        }

        return true;
}
```

Para controlar los paquetes entrantes, he agregado al formulario un sincronizador, *Timer*, de forma tal que cada 500 microsegundos (0.5 segundos) está escuchando el buffer del socket en busca de paquetes. Luego de las configuraciones del socket, se pueden ver dos métodos que llaman la atención, el primero, `Poll`, definido por la clase `System.Net.Sockets`, determinará el estado del socket, y esperará la cantidad de tiempo proveída en el primer parámetro (en milisegundos) por una respuesta.

```
private void timerReceiver_Tick(object sender, EventArgs e)
{
        Boolean canRead;
        Byte[] buf = newbyte[50];

        try
        {
                Int32 port = global::Socket_01.Properties.Settings.Default.port;
                IPAddress ip = System.Net.IPAddress.Any;
                String LocalIP = global::Socket_01.Properties.Settings.Default.ipLocal;
                SocketProcess.socket.Preparar(ip, port, LocalIP);

                canRead = SocketProcess.socket.Socket.Poll(1000000, SelectMode.SelectRead);
                if (canRead)
                        buf = SocketProcess.socket.RecibirUDP(buf);

                if (buf[0] > 0)
                {
                        if (System.Text.Encoding.ASCII.GetString(buf).IndexOf('|') == -1)
                        {
                                CargarLog(clientTextBox, "Recibiendo datos de cliente...");

                                ClaveCifrada = System.Text.Encoding.ASCII.GetString(buf);
                                CargarLog(clientTextBox, "Clave recibida: " + ClaveCifrada);
                                CargarLog(clientTextBox, "Iniciando validación...");
                        }
                }
        }
        catch
        {
        }
}
```

El segundo método, llamado `RecibirUDP`, el cual tiene como única función recibir datos a través de la instancia de `IPEndPoint` definida previamente a nivel interno de método. En este caso, no es necesario que se haga una verificación de la existencia de datos en el buffer, dado que el método `Poll` tiene esa función

particular. Lo que quiero decir es que esta instrucción de recepción de datos únicamente se ejecutará si se ha asegurado que existen datos en el buffer del socket. El código de este método se puede a continuación:

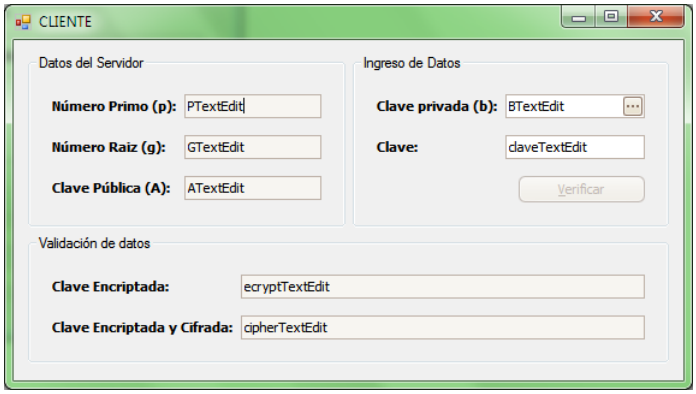
```
public static byte[] RecibirUDP(Byte[] buffer)
{
        _EndPoint = _IPEndPoint;

        _Socket.ReceiveFrom(buffer, ref _EndPoint);
        return buffer;
}
```

Con este método termino con la aplicación del lado del servidor. Veamos ahora la aplicación para el cliente.

Aplicación SocketClient

Como pueden observar, en la siguiente pantalla se definen los controles de formulario que corresponden a la aplicación de lado del cliente para el intercambio de las llaves. En este formulario no hay control de verificación, por lo que mejor conozcamos su funcionamiento interno.

Voy a dar una breve explicación de cada control utilizado en este formulario:



`PTextEdit`: Este control es tipo *textbox* y su función es mostrar el dato del número primo de referencia recibido del servidor cada vez que aquel lo genera y lo envía.
`GTextEdit`: Este control, al igual que el anterior, es tipo *textbox* y muestra el número primo generador recibido del servidor junto con el anterior.
`ATextEdit`: Este control, tipo *textbox*, muestra la clave pública del servidor – sólo para fines de verificación.
`BTextEdit`: Este control, también tipo *textbox*, muestra la clave privada del cliente, generada como número primo de manera aleatoria cada cierto tiempo. Es necesario que haga la aclaración que las claves privadas pueden ser cualquier número natural, no necesariamente un número primo.
`claveTextEdit`: Este control, tipo *textbox*, recibe de usuario la clave que se va a encriptar, cifrar y enviar al servidor para su validación. Es, precisamente, esta función la que hace útil este método: el intercambio de información cifrada con claves con vigencia temporal en el sistema de intercambio.
`encryptTextEdit`: Este control muestra la clave encriptada, antes de cifrar. Para encriptar se utiliza el método MD5, aunque Ustedes pueden usar el que consideren más conveniente.
`cipherTextEdit`: Este control muestra la clave cifrada, luego de encriptar. Para cifrar utilizo métodos de creación hash, ya que son los más comunes.

```
private void timerReceiver_Tick(object sender, EventArgs e)
{
        Boolean canRead;
        Byte[] buf = newbyte[50];

        try
        {
                Int32 port = global::SocketClient.Properties.Settings.Default.port;
                IPAddress ip = IPAddress.Parse(global::SocketClient.Properties.Settings.Default.ipServer);
                String LocalIP = global::SocketClient.Properties.Settings.Default.ipLocal;
                SocketProcess.socket.Preparar(ip, port, LocalIP); ;

                canRead = SocketProcess.socket.Socket.Poll(1000000, SelectMode.SelectRead);
                if (canRead)
                        buf = SocketProcess.socket.RecibirUDP(buf);

                if (buf.Length > 0)
                {
                        if (System.Text.Encoding.ASCII.GetString(buf).IndexOf('|') != -1)
```

```
                        {
                                string[] s = System.Text.Encoding.ASCII.GetString(buf).Split('|');

                                p = Convert.ToUInt64(s[0].Trim());
                                g = Convert.ToUInt64(s[1].Trim());
                                A = Convert.ToUInt64(s[2].Trim());

                                PTextEdit.Text = p.ToString();
                                GTextEdit.Text = g.ToString();
                                ATextEdit.Text = A.ToString();
                        }
                }
        }
        catch(Exception exc)
        {
                MessageBox.Show(exc.ToString());
        }
}
```

El método anterior define el funcionamiento de un temporizador, Timer, el cual tiene un intervalo de tiempo de funcionamiento de 500 milisegundos (0.5 segundos) para leer el contenido del buffer del socket configurado con la instancia de la clase `IPEndPoint` que apunta al servidor, el cual está indicado en el archivo de configuración de la aplicación cliente.

Una vez que los datos del servidor han sido recibidos, se habilita el botón para poder enviar los datos de validación de regreso. Pido disculpas por el nombre técnico con el que fue creado el método, pero sé que Ustedes no se molestarán por no haberle puesto nombre de entrada. El envío de los datos al servidor, luego de haber generado la clave pública y la contraseña de verificación se envían al invocarse el siguiente método:

```
private void simpleButton1_Click(object sender, EventArgs e)
{
        if (BTextEdit.Text == "")
        {
        MessageBox.Show("Ingrese un número privado válido!");
        BTextEdit.Focus();
        return;
         }

        if (claveTextEdit.Text == "")
        {
                MessageBox.Show("Ingrese un número privado válido!");
                claveTextEdit.Focus();
                return;
        }

        try
        {
                B = SocketProcess.matematicas.residuo(g, b, p);
                Kb = SocketProcess.matematicas.residuo(A, b, p);

                string ecrypt = SocketProcess.hash.hashMD5(claveTextEdit.Text);
                string cipher = SocketProcess.hash.hashCipherMD5(claveTextEdit.Text, Kb.ToString());

                ecryptTextEdit.Text = ecrypt;
                cipherTextEdit.Text = cipher;

                Enviar(cipher);
        }
        catch (Exception exc)
        {
                MessageBox.Show("Error: " + exc.Message);
        }
}
```

El método anterior se complementa con el método `Enviar`, muy similar al método que ya vimos en la aplicación de servidor, con la diferencia de que este recibe una cadena de caracteres y no un arreglo de bytes, aunque en utilización de memoria sean prácticamente lo mismo.

```
private void Enviar(String s)
{
        Byte[] Datos = System.Text.Encoding.ASCII.GetBytes(s);

        Int32 port = global::SocketClient.Properties.Settings.Default.port;
        IPAddress ip = IPAddress.Parse(global::SocketClient.Properties.Settings.Default.ipServer);
        String LocalIP = global::SocketClient.Properties.Settings.Default.ipLocal;
        SocketProcess.socket.Preparar(ip, port, LocalIP);
```

```
        SocketProcess.socket.EnviarUDP(Datos);
}
```

Luego, he realizado la programación en un método similar a los de la aplicación de servidor para generar un número primo aleatorio que corresponda a la clave privada del lado del cliente. Uso esta vez 100,000 como mínimo y 999,999 como máximo.

```
private void BTextEdit_ButtonPressed(object sender, DevExpress.XtraEditors.Controls.ButtonPressedEventArgs e)
{
        b = SocketProcess.primos.NumeroPrimo(100000, 999999);
        BTextEdit.Text = b.ToString();
}
```

Bien, con esto he terminado... buena suerte!