SizingServers.IPC

Intro

This library aims to make inter process communication easier.

The main idea behind it is eventing like it is implemented in .Net, but inter process instead of just in the same application. To achieve this it makes use of TCP sockets and does all the otherwise serialization and other boilerplate code + a few extras.

Locally you could just as easy use named pipes on Windows. However, this IPC implementation can work distributed.

It is written in C# (**.Net 4.6**) but could with a bit of effort be ported to other languages, for instance Java.
Maybe I will do this myself in the future.

Installation and examples

All binaries (64 bit) are located in the Build folder.
Reference SizingServers.IPC.dll, from your Visual Studio Project. The library is well commented. XML documentation is available.

There you will also find an installer for the end point manager service.
You will *not* need this if you want to do IPC for the local user. For system wide IPC or IPC over the network this service needs to be installed somewhere where all IPC senders and receivers can connect to over TCP (more on that later).

In the provided Visual Studio (2015) Solution are two console application projects present: a receiver and a sender example. Uncomment or comment the sections in the program.cs files to test the different functionalities of the library. It is possible that your virus scanner will detect a trojan horse. This is a false negative. I don't do malicious stuff.

In order to succesfully open the solution, your Visual Studio needs to have the Installer project templates: https://visualstudiogallery.msdn.microsoft.com/f1cc3f3e-c300-40a7-8797-c509fb8933b9

Local user IPC

This is the most simple use case.

To set this up you need one or multiple IPC receivers and one IPC sender. A sender and receivers are binded using a given handle (string) in their constructor.

A receiver will register itself in the Windows registry. This registration is volatile, meaning that the registry key disappears when the user logs off. A receiver will choose and listen on an available TCP port.

A sender will autodiscover all available receivers with the same handle.

When sending a message you can choose to let the lib do the serialization for you (using a binary formatter) or you can do your own serialization and send the bytes.

From the examples:

Sender

```
using System;
using System.Net;
using System.Timers;

namespace SizingServers.IPC.TestSender {
  class Program {
    static Sender _sender;
    static int _count;
    static void Main(string[] args) {
      _sender = new Sender("SizingServers.IPC.Test");
      _sender.OnSendFailed += _sender_OnSendFailed;

      var tmr = new Timer(1000);
      tmr.Elapsed += Tmr_Elapsed;
      tmr.Start();

      Console.ReadLine();
      tmr.Stop();
      _sender.Dispose();
    }

    private static void _sender_AfterMessageSent(object sen
der, MessageEventArgs e) {
      object message = e.Message;
      if (message is byte[]) message = System.Text.Encoding
.UTF8.GetString(message as byte[]);
      string s = "'" + message + "' sent";
      if (e.RemoteEndPoints.Length == 0) s += ", but no rec
eivers found";
      Console.WriteLine(s);
    }
    private static void _sender_OnSendFailed(object sender,
 System.IO.ErrorEventArgs e) {
      Console.WriteLine(e.GetException());
    }

    private static void Tmr_Elapsed(object sender, ElapsedE
ventArgs e) {
      _sender.Send("Foo" + (++_count));
      //_sender.Send(System.Text.Encoding.UTF8.GetBytes("Fo
oBytes"));
    }
  }
}
```

Receiver

```
using System;
using System.Net;

namespace SizingServers.IPC.TestReceiver {
  class Program {
    static Receiver _receiver;
    static void Main(string[] args) {
      _receiver = new Receiver("SizingServers.IPC.Test");
      _receiver.MessageReceived += _receiver_MessageReceive
d;

      Console.ReadLine();
      _receiver.Dispose();
    }

    private static void _receiver_MessageReceived(object se
nder, MessageEventArgs e) {
      object message = e.Message;
      if (message is byte[]) message = System.Text.Encoding
.UTF8.GetString(message as byte[]);
      Console.WriteLine("'" + message + "' received");
    }
  }
}
```

Systemwide IPC and IPC over the network

To enable this, a Windows Service was written.

Receivers will register on this end point manager service instead of the Windows Registry.
Senders will fetch available receivers from this service.
Suffice to say that the epm service must run somewhere where all parties can connect to.

The Service Installer is a pain:

- The description cannot be formatted, so the installer's welcome page

looks very messy when installing the service
- You cannot provide service start parameters when installing, you must do this yourself via Services.msc or the command line
- When uninstalling, the service cannot be automatically stopped; batch scripts are provided to automate starting and stopping the service a bit

Feel free to write a better installer and don't forget to make a pull request.

The service will listen by default on port 4455. You can give your own port using the service start parameters.
You can also let the traffic from the senders and receivers to the service and vice versa be encrypted (Rijndael) via these start parameters.

This encryption needs a passworrd and a salt. The given salt is a string representation of a byte array.

Alternatively, you can channel the traffic through a SSH tunnel. This will probably be safer and faster.

Btw, this sender & receiver <-> service traffic is gzipped.

Examples:

```
4456
```

or

```
4455 password {0x01,0x02,0x03}
```

If you make a mistake or an error occurs in the service this will be logged to the Windows Event Log (Event Viewer). Status messages will appear there also.

Below a sender and receiver example from the examples in the solution. The sender and the receiver need both a EndPointManagerServiceConnection object to be able to communicate to the service.

To make Firewall settings easier and registered TCP connections more manageable you can give two more things to the constructor of a receiver:

- ipAddressToRegister: the IP address that will be registered at the epm service; this helps enabling sender <-> receiver communication on computers that are connected to multiple networks; if this is null the receiver will choose which IP address will be registered
- allowedPorts: when using this, a TCP port to bind to will be chosen from the given array; if all ports are already in use, an exception will be thrown

Obviously these extra parameters are only handy (needed) when you are using the epm service. In "local user mode" IPC is much more simple.

If you want to apply string encryption and gzipping to the messages you send using this library: that functionality is made available in the static class Shared.

Sender

```
using System;
using System.Net;
using System.Timers;

namespace SizingServers.IPC.TestSender {
  class Program {
    static Sender _sender;
    static int _count;
    static void Main(string[] args) {
      //var epmsCon = new EndPointManagerServiceConnection(
new IPEndPoint(IPAddress.Loopback, Shared.EPMS_DEFAULT_TCP_
PORT));
      var epmsCon = new EndPointManagerServiceConnection(ne
w IPEndPoint(IPAddress.Loopback, Shared.EPMS_DEFAULT_TCP_PO
RT), "password", new byte[] { 0x01, 0x02, 0x03 });

      _sender = new Sender("SizingServers.IPC.Test", epmsCo
n);
      _sender.AfterMessageSent += _sender_AfterMessageSent;
      _sender.OnSendFailed += _sender_OnSendFailed;

      var tmr = new Timer(1000);
      tmr.Elapsed += Tmr_Elapsed;
      tmr.Start();

      Console.ReadLine();
      tmr.Stop();
      _sender.Dispose();
    }

    private static void _sender_AfterMessageSent(object sen
der, MessageEventArgs e) {
      object message = e.Message;
      if (message is byte[]) message = System.Text.Encoding
.UTF8.GetString(message as byte[]);
      string s = "'" + message + "' sent";
      if (e.RemoteEndPoints.Length == 0) s += ", but no rec
eivers found";
      Console.WriteLine(s);
    }
    private static void _sender_OnSendFailed(object sender,
 System.IO.ErrorEventArgs e) {
      Console.WriteLine(e.GetException());
    }

    private static void Tmr_Elapsed(object sender, ElapsedE
ventArgs e) {
      //(sender as Timer).Stop();
      _sender.Send("Foo" + (++_count));
      //_sender.Send(System.Text.Encoding.UTF8.GetBytes("Fo
oBytes"));
    }
  }
}
```

Receiver

```
using System;
using System.Net;

namespace SizingServers.IPC.TestReceiver {
  class Program {
    static Receiver _receiver;
    static void Main(string[] args) {
      //var epmsCon = new EndPointManagerServiceConnection(
new IPEndPoint(IPAddress.Loopback, Shared.EPMS_DEFAULT_TCP_
PORT));
      var epmsCon = new EndPointManagerServiceConnection(ne
w IPEndPoint(IPAddress.Loopback, Shared.EPMS_DEFAULT_TCP_PO
RT), "password", new byte[] { 0x01, 0x02, 0x03 });

      _receiver = new Receiver("SizingServers.IPC.Test", nu
ll, null, epmsCon);
      _receiver.MessageReceived += _receiver_MessageReceive
d;

      Console.ReadLine();
      _receiver.Dispose();
    }

    private static void _receiver_MessageReceived(object se
nder, MessageEventArgs e) {
      object message = e.Message;
      if (message is byte[]) message = System.Text.Encoding
.UTF8.GetString(message as byte[]);
      Console.WriteLine("'" + message + "' received");
    }
  }
}
```

Extra

There is a lot of stuff made available in the static class Shared.

It holds serialization-, gzip-, string encryption (Rijndael)- and stream handeling functionality.

Known issues

When there is a sender- and a receiver app and a new receiver app is launched it can get the same sent message twice just thereafter.
The end point manager service installer could be better.
When opening the Visual Studio solution, it is possible that your virus scanner will detect a trojan horse. This is a false negative. I don't do malicious stuff.