

AMPLEFILE

Network Communication Software

DotNetRemoting Framework

AMPLEFILE

Programming Guide

© Amplefile www.dotnetremoting.com

Table of Contents

CORE FEATURES

PROGRAMMING MODEL BASICS

FRAMEWORK STRUCTURE

CONNECTION

SERIALIZATION AND ESTABLISHING THE CONNECTION

STATUS MESSAGE

CREATION OF ASYNCHRONOUS COMMUNICATION CHANNEL

ESTABLISHING PHYSICAL CONNECTION

SYNCHRONOUS CHANNEL CREATION

THREADS AND DOTNETREMOTINGPLUS

ERROR AND EXCEPTION HANDLING

CLIENT AUTORECONNECTION

CLIENT DISCONNECT AND SERVER START/STOP

UDP BROADCAST

BEHIND PROXY OR NAT

DOTNETREMOTINGPLUS CLASS DIAGRAM

HTTP REMOTING CLASS DIAGRAM

HTTPCLIENT, HTTPSERVER

SENDING OBJECTS VIA HTTP

COMMUNICATION VIA GPRS (POCKETPC, SMARTPHONE)

RUNNING SAMPLES

INTERPROCESS COMMUNICATION COMPONENTS

DotNetRemotingPlus

Cross – Platform Bidirectional Communication Framework

CORE FEATURES

- Bidirectional synchronous and [or] asynchronous communication
- Built for .Net full and Compact Frameworks
- True remoting on PC and PDA
- DotNetRemotingPlus is the only available bidirectional framework for the Handhelds [PDA, PocketPC, Smartphones]
- Outstanding performance (20 times faster than Webservices)
- Accessibility for Clients behind NAT, Firewall or Proxy
- Supported protocols TCP, HTTP, UDP
- Client Autoreconnection
- Encryption
- 100% Managed code
- C# and VBNet sample code included
- Same server is used for PC and PDAs
- GPRS support(for PDAs, Smartphones)
- Connection Autorecovery on PDA (reconnection on wake up)
- Interprocess Communication Components

PROGRAMMING MODEL BASICS

DotNetRemotingPlus supports 3 communication models:

Remoting – execution methods remotely (classic remoting)

Symmetrical Remoting – execution methods remotely on the client and/or on the server (PC only)

Asynchronous – Send/Receive

Synchronous – Send/Wait/Receive

Asynchronous model : is the native model for any bidirectional communication.

The abstract interface for a asynchronous model is

A.Send(object MyAnyObject);

And the reception side may look like

```
void Rx(object MyAnyObject)
{
    // Rx code
}
```

Synchronous model :

Synchronous communication can have two flavors :

1. Remoting (remote execution)
2. Synchronous send.

Remoting

DotNetRemotingPlus remoting is the same as a standard remoting. The method `CreateRemoteObject()` is called on the client. It creates a proxy. (*Remoting object is a proxy in standard remoting*)

Then the method can be executed directly on this object. (proxy)

Synchronous send

Synchronous send is another model supported by DotNetRemotingPlus. The main benefit of using synchronous methods (including traditional remoting) is not only the convenient way of the passing the arguments to the method. The main purpose of the synchronous communication is to return to **the point where the execution was interrupted**. Such model can be easily (and natively) be implemented on the client.

On the server it is more complicated conceptually. Technically it is not too hard to execute something on the client. The client has almost the same structure as the server. But the question is how to select the client.

The proxy that represents the remoting object has no notion of the Client ID at all. So, what particular client the methods have to be executed on? Well, we can implement this method on the proxy (the method that sets the destination client). But that will make the proxy non generic.

This proxy now has to be derived from some kind of interface that has ClientID property.

This condition limits considerably the versatility and simplicity of the system. What about setting this property externally (or statically) without involving the proxy? In a single threading environment this will work, though in multithreading it poses a substantial risk. Without proper and sophisticated synchronization there will be the risk of calling wrong client (at best) or crashing the system. Implementing traditional remoting from the server to the client using delegates (or callbacks) has same problems and difficult to understand for the inexperienced programmer.

Also the execution of the methods on the client is not frequently required. The most important is to deliver the information asynchronously from the server (or sometimes [very infrequently] synchronously).

In the view of the above the most appropriate solution for the server is a **synchronous send** – sending any object and waiting for the response object. It keeps the system ultimately simple and robust.

Synchronous send abstract model:

Object = A.send(MyAnyObject)

where Object is the result returned from the server (or the client). The sending side (client or server) sends the object to the reception side and waits for the response. Such calls are handled by the synchronous method (sync Rx handler).

```
object Serv_RxSync(int ClientID, object InObject)
{
    // do something with InObject and send back the response – also an object
    return MyRetObject;
}
```

The sample code below is from the sample program “Client”

```
object _Sync_Async_Client_RxSync(object Data)
{
    listBox_data.Items.Add(Data); // “in” object is a string
    // let's return something back
    return new ComplexObject(); // out object is Complex object
}
```

Synchronous “send” and asynchronous “send” are 100% symmetrical (can be called from either side)

Remoting on the contrary is unidirectional. However the usage of all 3 models at the same time is not prohibited.

FRAMEWORK STRUCTURE

The client and the server are derived from the base class (not directly) from **BaseCommunication** object. **BaseCommunication** provides the internal data exchange mechanism (that includes status and data) The complete structure can be found in the document **DnrClasses.chm** This document was automatically built from the framework itself using Sendcastle software. Having all the objects derived from the root object, the derived objects are interchangeable. For instance SyncClient can be used as AsyncClient because SyncClient is built on the top of AsyncClient, same with the server.

Apart from base components there are specialized components such as HTTPProxy, NoFirewall tunnel , Peer-To-Peer , ProxyClient and other.

On the top of each communication class the control wrapper is built to support Windows Form Applications. The wrapper handles all the threading and graphical issues. It also facilitates the creation of the object using Visual Studio designer.

The communication objects are integrated with Visual Studio. These controls (wrappers) are

included into Visual Studio Toolbox when the framework is installed.

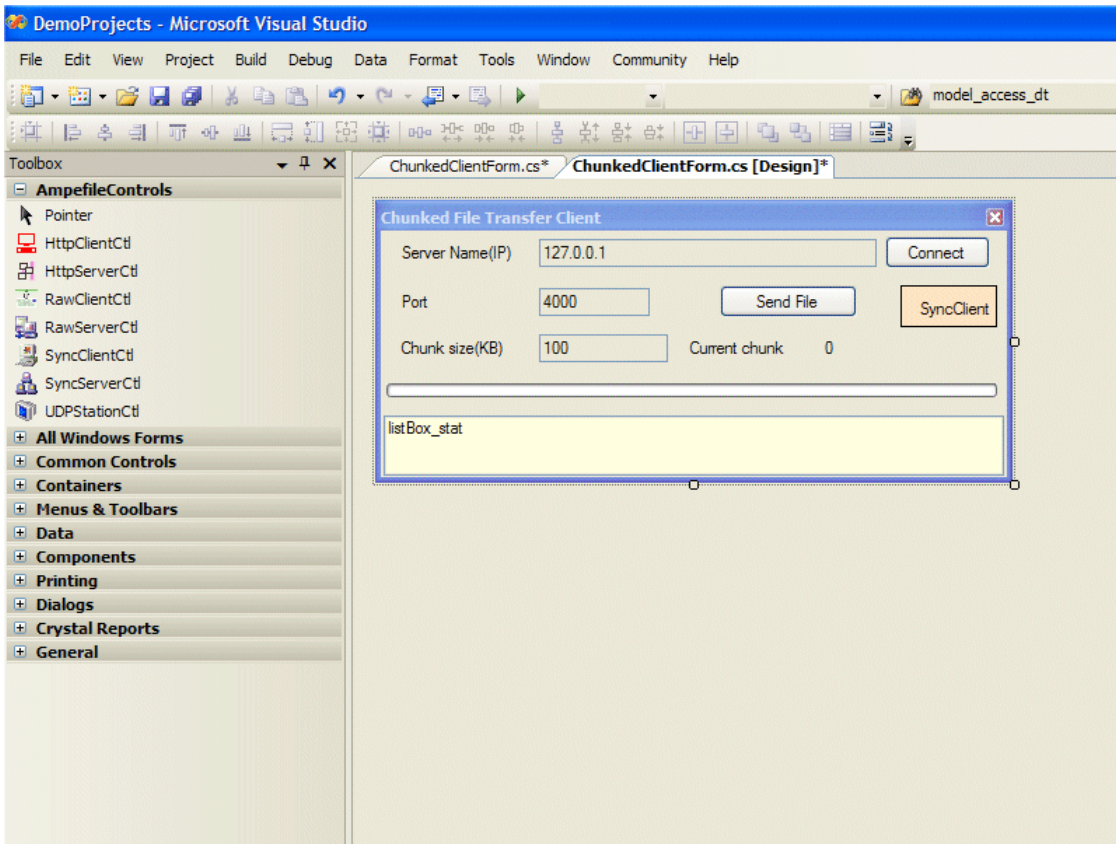


Figure 1. First step : create the client (drag from the toolbox)

To create a client or the server, the control has to be dragged from the toolbox to the form. The next step is the connection to the events. The connection to the events like `DataReceived` or `StatusEvent` can also be done from the designer. Selecting the object (`SyncClient` for example) and double clicking on the `DataReceivedHandler` in the event section will create the handler.

```
namespace DotNetRemoting
{
    public partial class SimpleClientForm : Form
    {
        public SimpleClientForm()
        {
            InitializeComponent();
        }

        private void button_connect_Click(object sender, EventArgs e)
        {
            Client.Connect(textBox_ip.Text, Port);
        }
    }
}
```

```

    }

    private void button_send_Click(object sender, EventArgs e)
    {
        Client.Send("The string");
    }

    private void Client_DataReceivedHandler(object Data)
    {
        ItemsData.Add(Data.ToString());
    }

    private void Client_StatusHandler(StatusMessage sm)
    {
        ItemsStat.Add(sm.ToString());
    }
}

```

The code above presents a fully functional bi-directional client. Total 5 lines of the code. Also this client handles all the threading and GUI issues. **All DotNetRemoting methods can interact with GUI methods directly without creating complex threading adapters.**

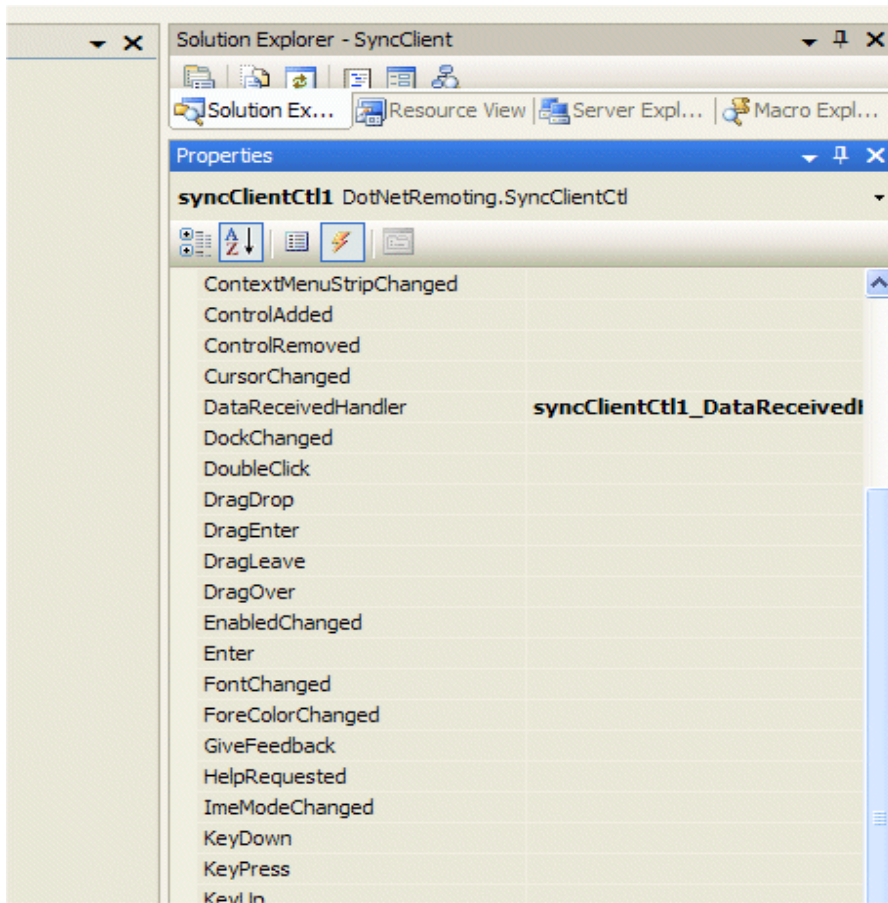


Figure 2. Visual Studio DataHandler creation

CONNECTION

SERIALIZATION AND ESTABLISHING THE CONNECTION

After the socket connection is established, the client sends the information to the server about the serializer that should be used. The serialization in DotNetRemotingPlus has two levels: DotNetRemotingPlus uses generic serializer at the first stage. If the user prefers standard BinaryFormatter, **BinarySerializer.dll** has to be put to the client's working directory and the serializer will be switched to standard BinaryFormatter. However that may not be necessary – generic serializer can handle almost anything (including full serialization of DataSet)

Note : Compact Framework does not have Binary serialization implemented. The only choice is the built-in generic serializer.

STATUS MESSAGE

Every significant event is logged and sent with the status message to the subscriber. The subscriber may be anything. To subscribe to the status event:

```
_AsyncClient.StatsHandler += new StatDelegate(_AsyncClient_StatsHandler);
```

And the event handler generated by the Visual Studio may contain the code below (example)

```
void _AsyncClient_StatsHandler(StatusMessage sm)
{
    listBox_status.Items.Add(sm.ToString());

    if (sm.Status == status.connect)
        button_connect.Text = "Disconnect";

    if (sm.Status == status.disconnect)
        button_connect.Text = "Connect";
}
```

The snippet above is from the sample code that is shipped with the framework. It controls (changes the text to Connect/Disconnect) the button.

Note that the event is fired in the main thread and therefore can control GUI methods directly without Invoke adaptors.

StatusMessage is a class with fields for status enum, message and CodeMessage.

```
public class StatusMessage
{
    public status Status;
    public int ID;
    public DateTime TimeStamp;
    public string Mess;
    public string CodeMessage;

    public StatusMessage(int ID, status Stat, string Message, string CodeMessage)
    {
        this.Status = Stat;
        this.TimeStamp = DateTime.Now;
        this.ID = ID;
        this.Mess = Message;
        his.CodeMessage = CodeMessage;
    }

    public StatusMessage(int ID, string Message) : this(ID, status.none, Message, null) { }
```

```

public StatusMessage(int ID, string Message, string CodeMessage) : this(ID, status.none,
    Message, CodeMessage) { }

public StatusMessage(string Message) : this(-1, Message, null) { }

public override string ToString()
{
    return "stat=" + Status.ToString() + ",codeM=" + CodeMessage + ",mess=" +
        Mess + ",id=" + this.ID.ToString();
}
}

```

Status message is accessible from any point within the object derived from BaseCommunication class. During debugging session it is convenient to use **StatusMessage.ToString()** method to view the status.

ESTABLISHING PHYSICAL CONNECTION

Physical connection with the server can be established via the call of **Connect()**.

1.Connecting directly:

```

AsyncClient["PORT"] = 15000;
AsyncClient["HOSTNAME"] = "MyRemoteServer.com"
_AsyncClient.Connect();

```

2.Connecting via ProxyConnector

```

ProxyConnector pc = new ProxyConnector(); // then set additional ProxyConnector parameters
_AsyncClient.Connect(pc);

```

If the connection fails, the status message will indicate this condition.

REMOTE EXECUTION

There are two ways of creating of an “exec” channel:

1. Using the proxy generated by **Proxy Studio** program

The proxy generated by the **Proxy Studio** can be **used with full and Compact Framework**. The benefit of using this proxy is that proxy is generated from any class, not necessarily from **MarshalByRefObject** class.

For the Compact framework it is the only way to invoke the methods synchronously because

Compact framework does not have [MarshalByRefObject](#) implementation. To invoke the method on the proxy, SyncClient should be used as a client. SyncClient can invoke methods only on the server. The server can not invoke methods synchronously on the client.

Remote execution from the server to the client is not implemented because it is not really required. Synchronous communication from the server to the client can take full advantage only if the relation between the connection points is peer-to-peer, not client- server. The server must serve all the clients, and waiting for the call to complete will delay the communication with other clients.

2. Using the proxy generated by the framework.

Such proxy is available only with the full framework. (not implemented for CF)
The proxy (remoting object) can be created by calling CreateRemoteObject() method.

CLIENT SIDE

```
private RemoteClass _ RemoteClass; // proxy to execute methods  
//remotely on the server
```

```
Client = new SyncClient ();
```

```
// create a proxy  
_ RemoteClass = (RemoteClass)Client.CreateRemoteObject(typeof(RemoteClass));
```

RemoteClass is a proxy derived from [MarshalByRefObject](#) (it is done same way as in standard remoting).

SERVER SIDE

```
Server = new SyncServer(); [or SyncServerCtl if it is Windows Forms Application]
```

If synchronous methods (do not confuse for remote execution) are called , synchronous handler should be provided

```
Server. RxSync += new ObjDelegateSyncClient(_Sync_Async_Client_RxSync);
```

THREADS AND DOTNETREMOTINGPLUS

The communication framework internally is a very complex object with multiple threads running. Each individual thread has its functionality (timers, queues stacks etc). However the limitation of any GUI program (due to the windows messaging model) is that GUI methods can only be called from the main thread. That means that some kind of adapter should call GUI methods from the system that is multithreaded. Generally the thread synchronization is the most complex, sophisticated and hard to debug subject in programming. **DotNetRemotingPlus handles all the threading issues internally. The programmer does not have to think about the threads any more.**

ERROR AND EXCEPTION HANDLING

All the errors and exceptions are logged and passed to the status subsystem. This subsystem fires the event when the status has changed in any way (connection/disconnection/error/info)

Status event delivers **StatusMessage** object with the code and message.
Using status event you can implement your own reconnection system for example.

```
if (sm == status.disconnected)
{
    Client.Reconnect();
}
```

It is just a simplified example because reconnection is more complicated than detecting disconnected status.

REMOVING AND EXCEPTION HANDLING

Remoting exception handling is built on try/catch pattern.

```
try
{
    Myobject.ExecuteMyMethod(params)
}
catch(RPCException rex)
{
    // do something
}
```

Apart from catching the exception, **StatusMessage** has also to be examined, if more detailed information is required.

CLIENT AUTORECONNECTION

This feature allows for the automatic reconnection of the client when the connection is lost. The client and the server, if **KeepConnectionAliveTime** is set to none zero, exchange with the **KeepConnectionAlive** objects. If during (**KeepConnectionAliveTime** * 2) no object is received, the client issues the status message with reconnection request. If **Autoreconnect** is set to **true**, the framework will attempt to reconnect the client to the server.

```
// Autoreconnect ON
_AsyncClient.KeepConnectionAliveTime = 5000;// 5s
_AsyncClient.MaxReconnectionAttempts = 5;
_AsyncClient.TimeBetweenReconnectAttemptsMs = 10000;// 10s
_AsyncClient.AutoReconnect = true;
```

CLIENT DISCONNECT AND SERVER START/STOP

If the client connects to the server, both sides receive the notification, same with the disconnection. The socket notifies the other side if it is shut down. However it may not be always the case. If the connection is lost (somewhere in the network), there is no way to notify the other side that the channel is not up any more. That is why the periodic `KeepConnectionAlive` object exchange is required. If the object is not received on time, the socket is disconnected forcefully. If the server is stopped, all the clients are notified either.

UDP BROADCAST

UDP broadcast is the best way to send some object to all the listeners at once because UDP protocol allows for such multicast. However the protocol does not grantee the object delivery. UDP broadcast makes sense only with a local network and it is the fastest in such operation. The size of the object is limited to 65k, which is more than enough to notify the clients. In order to simplify the usage of such a broadcast, **UDPStation** object was implemented. You have to specify only the port and send the object. The object will be received by all such **UDPStations** on the network.

BEHIND PROXY OR NAT

Since only one connection is open for both directions, the communication behind the NAT is not a problem any more. For the ultimate firewall transparency special components have been designed from `DotNetRemotinPlus` base classes.

NoFirewall tunnel,
Peer-to-Peer,
and Skyping

CONNECTION VIA PROXY (SOCKS OR HTTP PROXY)

Connection via Socks Proxy or `HttpProxy` can be established using `ProxyConnector` Cmpnent.

```
public ProxyConnector(  
    ProxyType proxyType,  
    string HostName,  
    int HostPort,  
    string proxyServer,  
    int proxyPort,  
    string User,  
    string Password)
```

The `ProxyConnector` constructor has a few parameters
Where `ProxyType` – type of the proxy

```
public enum ProxyType  
{
```

```
    None,  
    Socks4,  
    Socks4a,  
    Socks5,  
    HttpConnect  
}
```

```
Client.Connect(pc);
```

The actual code sample of using ProxyConnector is in ClientTest application.

HTTPCLIENT, HTTPSERVER

Instead of TPC/IP HTTP can be used.

The only advantage of using HTTP is that HTTP requests may be relayed by the HTTP proxy.

However this assumption is not always true (if we are talking about bidirectional communication)

The proxy for the bidirectional communication must support HTTP 1.1 (to keep connection alive) and that is not always the case. Therefore the channel has to be tested before doing serious development.

The usage of HTTP protocol can be demonstrated in **HttpClient** and **HttpServer** applications.

COMMUNICATION VIA GPRS (POCKETPC, SMARTPHONE)

General Packet Radio Service (GPRS) is a Mobile Data Service available to users of Global System for Mobile Communications (GSM) and IS-136 mobile phones. GPRS data transfer is typically charged per megabyte of transferred data, while data communication via traditional circuit switching is billed per minute of connection time, independent of whether the user has actually transferred data or has been in an idle state. GPRS can be used for services such as Wireless Application Protocol (WAP) access, Short Message Service (SMS), Multimedia Messaging Service (MMS), and for Internet communication services such as email and World Wide Web access.

Physically communication via GPRS makes no difference from DotNetRemotingPlus perspective. The only condition is that the device should be already connected to the GPRS service provider. GPRS connector is a component for connecting to GPRS.

It uses URL or IP Address as a parameter. The status of the connection is also accessible.

If the connection is no longer needed, it may be dropped.

DOTNETREMOTINGPLUS AND LINUX OPERATING SYSTEM

The library can run on Linux without recompilation. However there are some limitations:

The serialization of the objects in .NET and Mono are not compatible.

Simple objects will run well on both platforms because the basic objects (and some other objects) are serialized by the built-in serializer. Other objects will go through the generic serializer. That

means that on this level the incompatibility of BinaryFormatter from .Net and Mono will not allow the commutation. **The conclusion is that BinaryFormatter should not be involved.**

DOTNETREMOTINGPLUS AND INTERPROCESS COMMUNICATION

DNRPlus supports interprocess communication using Named Pipes binding.

Connecting anything with everything

Basically there are two types of the communication available:

- Communication between the applications (P2P)
- Client-server (IPC) type of the communication, when all the applications talk to the server.

IPC components

IPC client component has just one method **Send** (`int Command`, `object Obj`) and two events (callbacks) **Status** and **RxData**.

The parameter `Obj` is any serializable object, if the sender and receiver are .Net applications, or the byte array, if the receiver and the sender are different.

The parameter **Command** is a short integer. What it is entirely up to the programmer.

The only limitation on this parameter, it must be greater than 100. (these values are reserved for the internal use)

The client works in conjunction with the server. The server has same methods :

Send and 2 Events.

The connection of the client is automatic. When it is created, it starts looking for the server.

VB6 clients are almost same, but first the callbacks have to be set.

VB6 and C++ clients have one extra parameter : `hWnd` – it is the handle to the parent window. It is required for sending data from the worker thread to the GUI thread.

It is the windows convention – the GUI thread and the worker threads can not communicate directly.

If the component is used in the windowless application, this parameter must be set to 0.

The components are roughly the same for C++, VB6 or .Net, however the implementation is slightly different.

The events in VB6 and C++ modules are implemented as callbacks.

(The .Net Event is a callback, only it is presented in .Net manner)

P2P components

Another group of components is P2P.

These components communicate to each other. They require P2P server running.

In order to select the target, another method has been added – **SetTarget**(`string Target`).

All the clients must have the unique names and they can be accessed only by the client's name.

Samples

All the servers (IPC and P2P) are implemented as .Net assemblies.
The clients are C++, .Net and VB6 applications.

Buffer size

One of the parameters when VB or C++ client get initialized is [MaxBuffSize](#).
This parameter sets the maximum buffer for reception and transmission.
.Net version does not have this parameter.

Server as Windows Service

In some cases it beneficial to set the server as Windows Service. The service can start automatically and it can be controlled in a unified manner.
However the service can not interact with GUI based application when it is required.
To overcome this disadvantage, the Server monitor was included in the installation.
Server monitoring system is a parallel server (control server) that runs in the same process with the main server (data server). The monitoring program is a client (with GUI or without) that receives the status from this parallel server. It is also capable of sending the commands to the control server.
Both servers (IPC and P2P) servers have sample monitoring systems.

Installing the service

Run the batch file in [ServiceInstall](#) in the <Service>/bin/debug directory

Starting the client

The client connects automatically and does not require explicit starting.
Once disconnected (for any reason), it will be attempting to connect to the server again.

How many servers allowed?

As many as necessary, as long as they have unique ID (as integer)