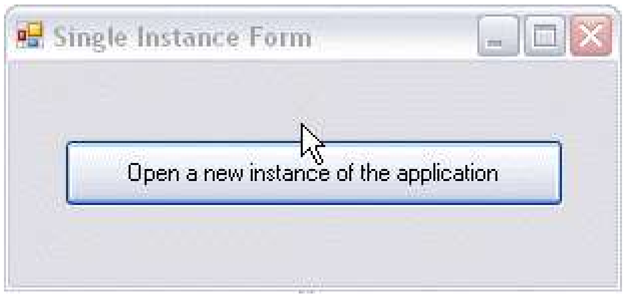
A Pure .NET Single Instance Application Solution

Shy Agam
17 Nov 2007 CPOL

This article shows how to enforce single application instancing, and perform interprocess communication (IPC) using pure .NET code.

Download demo - 7.84 KB

Download source - 18.8 KB



Introduction

Applications sometimes need to make sure that only one instance of them is running at a time. This sometimes brings up another problem, if the application accepts command line arguments which should be passed to the already running instance of the application.

One of my applications had such a need, and I realized it would be more useful if the solution came in a DLL, which could be used by any future application that requires the same ability.

This article describes how to use pure managed code, which creates an infrastructure used to achieve both goals:

- Detecting whether an instance of the application is already running.
- Sending data to the already running instance of the application.

Note: There are some points in the article at which it is assumed that you are somewhat familiar, even to a small extent, with the basics of concepts like .NET Remoting, Serialization, Exceptions Handling, Delegates, Threads Synchronization, Dispose pattern, etc. Needless to say, these topics will not be covered here, as these are not the topic of the article. However, I will override the above statement whenever appropriate.

This is my first article here at CodeProject, so I hope you'll find it useful. Happy reading... :)

Background

As always, before coding, I did some research. Of course, there are more solutions to the above problems, but most of the suggested solutions I've found implemented different things I chose to avoid, if possible, either because the solution had used an out-dated technique (e.g., DDE, etc.), or the solution had used some kind of wrapping classes for unmanaged, native API functions, which had to be manually written (i.e., were not implemented by the .NET Framework). I actually found one, or maybe two, pure managed solutions, but they used a `TcpChannel`, which is pretty wasteful, in terms of performance speed, and maybe resources consumption as well.

I *will* mention other possible solution strategies, just in case you would like to further expand your knowledge, but a comprehensive explanation will only be given to the solution I chose to implement.

Describing yhe Problem

Every application loaded to memory has its boundaries. These boundaries define rules as for what the application can access, and they are called Application Domains. Because such boundaries exist, an application cannot simply "talk" to another application. There's no straightforward way to do it. If an application cannot normally "talk" to another application, it can neither detect a previous instance easily, nor can it send data to it (e.g., command line arguments).

The Solution

Is the application already running?

To detect if a previous instance of the application has been launched, I used a Named Mutex object. A mutex is an object providing a synchronization mechanism for threads, right? Correct, but that's not all. Mutex objects can also be used to synchronize code across application boundaries. What makes it possible to identify an existing mutex, is its unique name. When a mutex with a certain name is created, no other mutex can be created with the same name, hence the term "Named Mutex". However, *it is* possible to open an existing mutex, and use it to synchronize code.

The main idea

Each application instance attempts to create a mutex with a specific name and own it. If the mutex has been successfully owned, it means this is the first instance of the application. Every attempt, to create and own the mutex, which follows will fail, meaning this is not the first instance of the application.

The main idea - Put to code

We need "someone" to provide us with the info regarding the application instance in a simple way, without us having to write any mutex-related code every time. So, let's create a class which would implement the idea explained above (we will expand the functionality of the class later on):

Note: The `System.Threading.Mutex` class inherits from `System.Threading.WaitHandle`, thus having a `Close()` method. Calling this method when we're done using it is very important for releasing resources, and also freeing the mutex. This is why the following class implements the `IDisposable` interface.

C#

```csharp
public class SingleInstanceTracker : IDisposable
{
    private bool disposed;
    private Mutex singleInstanceMutex;
    private bool isFirstInstance;

    public SingleInstanceTracker(string name)
    {
        if (string.IsNullOrEmpty(name))
            throw new ArgumentNullException("name",
                                            "name cannot be null or empty.");

        try
        {
            singleInstanceMutex = new Mutex(true, name, out isFirstInstance);
        }
        catch (Exception ex)
        {
            throw new SingleInstancingException("Failed to instantiate a new
                                                 SingleInstanceTracker object.
                                                 See InnerException for more
                                                 details.", ex);
        }
    }

    ~SingleInstanceTracker()
    {
        Dispose(false);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                if (singleInstanceMutex != null)
                {
                    singleInstanceMutex.Close();
                    singleInstanceMutex = null;
                }
            }

            disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    public bool IsFirstInstance
    {
        get
        {
            return isFirstInstance;
        }
    }
}
```

Let's overview the constructor's code... First, we attempt to create a `Mutex` object. The arguments for the `Mutex`'s constructor are as follows:

- `bool initiallyOwned` - This value decides whether the calling code would own the mutex, in case it is created.
- `string name` - This is the unique name which identifies the mutex.
- `out bool createNew` - When the mutex's constructor returns, this argument contains a value indicating whether the mutex was created or opened.

So now, we know how to check if a previous instance exists. We simply read the value of the last argument... If its value is `true`, it means the mutax was created and this is the first application instance. If its value is `false`, it means the mutex already exists, thus it was opened (and not created), and this is not the first application instance. Hooray!!! First problem solved!

Note: I've seen different detection strategies using `Mutex`, which involve using the `WaitOne()` method, but I chose to go with the constructor, as it results in a one-lined detection code.

Passing data between instances

What's left to do is to make our class be able to "talk" with the first application instance, and send data to it. The act of conversing between applications, across application domains, is called Interprocess Communication (a.k.a. IPC). IPC can be implemented in many different ways, and if you're interested, here is a place for you to start: Microsoft's IPC Page.

After a long journey on the net, I decided to write a wrapper class which would use named pipes to deal with communications. But then, I had a brilliant idea! What if I write "IPC" in the Object Browser's search box?? So I did... and voila!! There's a namespace dedicated just for IPC, which is called `System.Runtime.Remoting.Channels.Ipc`. This entire namespace is new in .NET Framework 2.0, and it contains three classes: `IpcServerChannel`, `IpcClientChannel`, and `IpcChannel` which is a combination of both. These classes are implemented using named pipes. To use the namespace, a reference to *System.Runtime.Remoting.dll* must be added to the project.

Note: The `System.Runtime.Remoting.Channels.Ipc` is useful only when communicating between processes running on the same machine. But, this is not a disadvantage in the situation we're currently dealing with. On the contrary, this is a great advantage, because as opposed to using a `TcpChannel`/`HttpChannel`, an `IpcChannel` does not rely on a network connection, which means there will be a lot less overhead, and a great improvement in communication speed. That is exactly the reason I said I chose to avoid a `TcpChannel` solution, at the beginning of the article.

So, now that we know what tools we're going to use, let's continue to the solution...

The main idea

First, a few words on remoting... Remoting allows an object residing on one application to be used by another. In order for remoting to work, there are some rules:

- A class whose objects should be remotely instantiated must derive from the `MarshalByRefObject` class, to allow access to it across application domains.
- Every remoting session must have a server, which will provide the objects, and a client which will use the objects.
- In order for a client to be able to access a remote object, the server must register it and make it visible.
- Data passed using remoting must be serializable, as remoting relies on serialization of objects.

We need to be able to execute methods on the first application instance through any new instance. It means, we need some object to be instantiated on the server (which is the first application instance), and make use of it on the clients (any new instance of the application). This object, in turn, will execute methods on the first application instance. It will be our proxy object.

Now that we have a proxy object, residing on the first instance, we need to somehow tell it to call methods in the first application instance, whether on the main form, or anything else, depending on the application's design. So, for a class to be able to receive messages from new instances, it must implement an interface containing methods to process the messages. Then, when the proxy object is instantiated, it receives an object which implements the interface.

That's about it! Now, whenever a client (new application instance) needs to call the methods on the server (first application instance), it will use the proxy object as if it was its own object, and invoke the methods.

The main idea - Put to code

Let's begin with the interface:

C#

```
public interface ISingleInstanceEnforcer
{
    void OnMessageReceived(MessageEventArgs e);
    void OnNewInstanceCreated(EventArgs e);
}
```

As you can see, the `OnMessageReceived()` method received a `MessageEventArgs` argument. I will not discuss this argument, as you can find it in the provided source code. However, I *will* say that the `MessageEventArgs` class must be marked as

Serializable. The client is the one sending the message, so the message object is instantiated on the client's side, and is supposed to be sent to the server, which means it will be serialized. Remember the rules? Data passed using remoting must be serializable.

Next is our proxy:

C#

```
internal class SingleInstanceProxy : MarshalByRefObject
{
    #region Member Variables

    private ISingleInstanceEnforcer enforcer;

    #endregion

    #region Construction / Destruction

    public SingleInstanceProxy(ISingleInstanceEnforcer enforcer)
    {
        if (enforcer == null)
            throw new ArgumentNullException("enforcer",
                                            "enforcer cannot be null.");

        this.enforcer = enforcer;
    }

    #endregion

    #region Overriden MarshalByRefObject Members

    public override object InitializeLifetimeService()
    {
        return null;
    }

    #region Properties

    public ISingleInstanceEnforcer Enforcer
    {
        get
        {
            return enforcer;
        }
    }

    #endregion
}
```

This class is pretty self explanatory, except for two things: firstly, notice that the class inherits MarshalByRefObject. Again, if we go back to the rules... This class should be instantiated on the server, but accessed from clients, so it must derive from MarshalByRefObject. Secondly, the class is defined with the internal modifier. This is not a requirement, but as this class will be used by no one except for the SingleInstanceTracker class, I decided it should not be externally visible.

-- Update (17/11/07): --

Another important thing is the overridden InitializeLifetimeService() method. Every MarshalByRefObject object has a life-time (i.e., a specification when it should be collected by the GC). This life-time gets reset every time the object is referenced. But, if the object is not used for a certain amount of time, it gets garbage collected.

So, why should we care about it? When the proxy object is instantiated for the first time by the first application instance, it's manually instantiated by us, using the one and only constructor, which receives an ISingleInstanceEnforcer object. If we don't use the object for a certain amount of time, the GC will collect it. The next time we try to remotely access the proxy object, the framework would try and instantiate a new proxy object with a parameterless constructor (which does not exist). The solution is to make the object live forever. This is done by overriding the `InitializeLifetimeService()` method and returning `null`.

Many thanks to **Thomas Schoeniger** and **byates** for reporting the bug and posting the solution. :)

-- End of Update --

We've established that if this is the first instance, the application should create a server, and "publish" a `SingleInstanceProxy` object, so that every instance coming next will create a client and be able to access this object.

Here is the new constructor code for the `SingleInstanceTracker`, followed by an explanation:

C#

```
public SingleInstanceTracker(string name)
    : this(name, null) { }

public SingleInstanceTracker(string name,
       SingleInstanceEnforcerRetriever enforcerRetriever)
{
    if (string.IsNullOrEmpty(name))
        throw new ArgumentNullException("name",
              "name cannot be null or empty.");

    try
    {
        singleInstanceMutex = new Mutex(true, name, out isFirstInstance);

        // Do not attempt to construct the IPC channel
        // if there is no need for messages

        if (enforcerRetriever != null)
        {
            string proxyObjectName = "SingleInstanceProxy";
            string proxyUri = "ipc://" + name + "/" + proxyObjectName;

            // If no previous instance was found, create a server channel which
            // will provide the proxy to the first created instance

            if (isFirstInstance)
            {
                // Create an IPC server channel to listen for SingleInstanceProxy

                // object requests

                ipcChannel = new IpcServerChannel(name);
                // Register the channel and get it ready for use

                ChannelServices.RegisterChannel(ipcChannel, false);
                // Register the service which gets the SingleInstanceProxy object,

                // so it can be accessible by IPC client channels

                RemotingConfiguration.RegisterWellKnownServiceType(
                                      typeof(SingleInstanceProxy),
                                      proxyObjectName,
                                      WellKnownObjectMode.Singleton);

                // Attempt to retrieve the enforcer from the delegated method

                ISingleInstanceEnforcer enforcer = enforcerRetriever();
                // Validate that an enforcer object was returned
                if (enforcer == null)
                    throw new InvalidOperationException("The method delegated by the
                              enforcerRetriever argument returned null.
                              The method must return an
                              ISingleInstanceEnforcer object.");

                // Create the first proxy object

                proxy = new SingleInstanceProxy(enforcer);
                // Publish the first proxy object so IPC clients
                // requesting a proxy would receive a reference to it

                RemotingServices.Marshal(proxy, proxyObjectName);
            }
            else
            {
                // Create an IPC client channel to request
                // the existing SingleInstanceProxy object.

                ipcChannel = new IpcClientChannel();
                // Register the channel and get it ready for use

                ChannelServices.RegisterChannel(ipcChannel, false);

                // Retreive a reference to the proxy object which
                // will be later used to send messages

                proxy = (SingleInstanceProxy)
                         Activator.GetObject(typeof(SingleInstanceProxy),
                                             proxyUri);

                // Notify the first instance of the application
                // that a new instance was created

                proxy.Enforcer.OnNewInstanceCreated(new EventArgs());
            }
        }
    }
    catch (Exception ex)
    {
        throw new SingleInstancingException("Failed to instantiate a
                                             new SingleInstanceTracker object.
                                             See InnerException for more details.", ex);
    }
}
```

In order to make a channel usable, it must be registered. This is done by calling the `ChannelServices.RegisterChannel()` method in both branches of the `if (isFirstInstance)` condition block, either to register the server channel or the client channel.

In order to make a class remotlly usable, it must also be registered. Here, we have two choices when using the `RemotingConfiguration.RegisterWellKnownServiceType()` method... either specify `WellKnownObjectMode.SingleCall` or `WellKnownObjectMode.Singleton`. The latter means that every client's request to use the registered class will result in a reference to the same single object. This is exactly what is needed here... The proxy object to be instantiated once, on the server, and a reference to it on every client. So, we've got our server and client registration code, and the proxy's registration code. We need to create a proxy object, but to do that, we first have to retrieve an `ISingleInstanceEnforcer` object. I chose to make the constructor receive a delegate to a method which returns such an object. After constructing the first (and only) instance of `SingleInstanceProxy`, we have to publish it on the channel, so that clients requesting a proxy would end up receiving a reference to it, and we do that by calling the `RemotingServices.Marshal()` method. On the client side, to retrieve the reference to the proxy object, we use the `Activator.GetObject()` method, specifying the object's URI.

Done! Now, to send messages to the first instance, a simple call to one of `proxy.Enforcer`'s methods is made. The constructor now immediately notifies the first instance that another instance was created, by calling `proxy.Enforcer.OnNewInstanceCreated()`.

Lastly, here is the code which sends a message to the first instance:

C#

```
public void SendMessageToFirstInstance(object message)
{
    if (disposed)
        throw new ObjectDisposedException("The SingleInstanceTracker object
                                           has already been disposed.");

    if (ipcChannel == null)
        throw new InvalidOperationException("The object was constructed with
                                             the SingleInstanceTracker(string name)
                                             constructor overload, or with
                                             the SingleInstanceTracker(string name,
                                             SingleInstanceEnforcerRetriever
                                             enforcerRetriever) constructor overload,
                                             with enforcerRetriever set to null,
                                             thus you cannot send messages to
                                             the first instance.");

    try
    {
        proxy.Enforcer.OnMessageReceived(new MessageEventArgs(message));
    }
    catch (Exception ex)
    {
        throw new SingleInstancingException("Failed to send message to the first
                                             instance of the application.
                                             The first instance might have
                                             terminated.", ex);
    }
}
```

Using the Code

Choose the class which should process messages, and make it implement the `ISingleInstanceEnforcer` interface. In the demo project, I used the main form as the enforcer, and implemented it like so:

C#

```
public partial class MainForm : Form, ISingleInstanceEnforcer
{
    private delegate void OnMessageReceivedInvoker(MessageEventArgs e);

    public MainForm()
    {
        InitializeComponent();
    }

    #region ISingleInstanceEnforcer Members

    void ISingleInstanceEnforcer.OnMessageReceived(MessageEventArgs e)
    {
        OnMessageReceivedInvoker invoker = delegate(MessageEventArgs eventArgs)
        {
            string msg = eventArgs.Message as string;
            if (string.IsNullOrEmpty(msg))
                MessageBox.Show("A non-textual message has been received.",
                                "Message From New Instance",
                                MessageBoxButtons.OK,
                                MessageBoxIcon.Information);
            else
                MessageBox.Show(msg,
                                "Message From New Instance",
                                MessageBoxButtons.OK,
                                MessageBoxIcon.Information);
        };

        if (InvokeRequired)
            Invoke(invoker, e);
        else
            invoker(e);
    }

    void ISingleInstanceEnforcer.OnNewInstanceCreated(EventArgs e)
    {
        MessageBox.Show("New instance of the program has been created.",
                        "Notification From New Instance",
                        MessageBoxButtons.OK, MessageBoxIcon.Information);
    }

    protected virtual void OnMessageReceived(MessageEventArgs e)
    {
        ((ISingleInstanceEnforcer)this).OnMessageReceived(e);
    }

    protected virtual void OnNewInstanceCreated(EventArgs e)
    {
        ((ISingleInstanceEnforcer)this).OnNewInstanceCreated(e);
    }

    #endregion
}
```

I explicitly implemented the interface so it would be possible for me to create the same methods the interface contains, but declared them as `protected virtual`, in order to mimic the design of event handling in the .NET Framework. After completing the implementation of the `ISingleInstanceEnforcer` interface, it is time to use the `SingleInstanceTracker` class. And, where would be a better place than right before the main form gets instantiated, right? So, let's checkout the modified `Main()` method...

C#

```
[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);

    SingleInstanceTracker tracker = null;

    try
    {
        // Attempt to create a tracker

        tracker = new SingleInstanceTracker("SingleInstanceSample",
                                            new SingleInstanceEnforcerRetriever
                                                (GetSingleInstanceEnforcer));

        // If this is the first instance of the application, run the main form
        if (tracker.IsFirstInstance)
            Application.Run((MainForm)tracker.Enforcer);
        else // This is not the first instance of the application, so do

             // nothing but send a message to the first instance

            tracker.SendMessage("Hello first instance! this is the new
                                 instance!\nI will now terminate if
                                 you don't mind...");
    }
    catch (SingleInstancingException ex)
    {
        MessageBox.Show("Could not create a SingleInstanceTracker object:\n" +
                        ex.Message + "\nApplication will now terminate.");

        return;
    }
    finally
    {
        if (tracker != null)
            tracker.Dispose();
    }
}

private static ISingleInstanceEnforcer GetSingleInstanceEnforcer()
{
    return new MainForm();
}
```

As you can see, it is now a matter of a few lines of code to determine if the application is already running, and pass an object to its first instance. First, a `SingleInstanceTracker` object is created, then it is "interrogated" in order to determine whether this is the first instance, and if needed, send a message. Note that once it is decided that this is the first application instance, the main form is extracted from the `Enforcer` property, instead of being instantiated. The reason for this is that the `SingleInstanceTracker`'s constructor has already called the `GetSingleInstanceEnforcer()` method which instantiated the form. If a new form is created instead of being extracted from the `Enforcer` property, it will not receive the messages, because it is not the one associated with the proxy object.

Closing Words

The code does not provide a real-time check for a previous instance, but rather a snapshot of the check at the moment of instantiation. However, this is not a problem, as the check is normally made only once, when the application starts. This code can also be modified to make the first instance be able to send messages to the new instances, make the `ISingleInstanceEnforcer.OnNewInstanceCreated()` receive an `EventArgs` object which defines a `Cancel` property, or anything else, but the current implementation is enough for my needs.

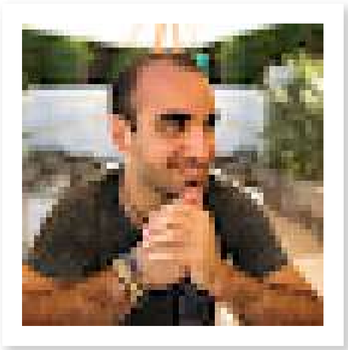
Any suggestions, questions, or constructive criticism are more than welcome... Thanks for reading, and happy coding!

History

- 22/7/07 - Updated source code (provided an old one by mistake).
- 17/11/07 - Fixed a bug causing the proxy object to be collected after a few minutes (source code updated as well).

License

About the Author



Shy Agam
Software Developer
Israel

No Biography provided

Comments and Discussions

28 messages have been posted for this article Visit **https://www.codeproject.com/Articles/19682/A-Pure-NET-Single-Instance-Application-Solution** to post and view comments on this article, or click **here** to get a print view with messages.