

# Generic Messaging

## Reference Documentation

Version 0.5.x

<http://slagd.com>

## Table of Contents

1. Preface
  - 1.1. What is GenericMessaging?
  - 1.2. Why do I need GenericMessaging?
2. Architecture
  - 2.1. General Explanation
  - 2.2. MessageHandler
  - 2.3. GenericMessagingClient
  - 2.4. GenericMessagingServer
  - 2.5. Lifecycle of a request
  - 2.6. Limitations
3. Implementation
  - 3.1. Setting up the project
  - 3.2. Configuration
  - 3.3. Service Contract
  - 3.4. Client Side
    - 3.4.1. Wrapped
    - 3.4.2. Extended
    - 3.4.3. Overview
  - 3.5. Server Side
    - 3.5.1. Wrapped
    - 3.5.2. Extended
    - 3.5.3. Hosting in IIS
    - 3.5.4. Overview
4. Extension Points
  - 4.1. Source Code
  - 4.2. GenericMessaginClient

- 4.3. GenericMessagingServer
- 4.4. MessageHandler
- 4.5. Graph Walker Workers
- 5. Future and Notes

# 1 Preface

At the time of this writing I am working for a small company called Memory Express. We are primarily a computer retailer and I and several other colleagues are responsible for upkeep and maintenance on the existing software as well as designing new software systems as they become needed. Because of the size of our team, time is always at a premium and doing things efficiently is paramount. I started off doing internal data communications with .net 2.0 web services. It was relatively easy, however as I began to accumulate more and more services I began to run into several key problems.

1. Maintainability becomes a major factor when you need to have a client object, a server object and numerous web methods for each one of those.
2. Loss of OO principles was quickly evident as web methods forced me to be much more procedural than I would have liked.

With the advent of WCF I began to evaluate ways of creating a more elegant way of .net internal communications. I was surprised to find out that there was still no good way of seamlessly communicating across the AppDomain boundary. I stress the word “seamless” because there are a lot of ways to go about it. As my dissatisfaction with the current situation increased, so did my desire to create something better. After a significant amount of research I decided to base off of the WCF architecture and thus the GenericMessaging project was born.

## 1.1 What is Generic Messaging?

GenericMessaging is in short a library that expands the capabilities of the Windows Communication Foundation (WCF). WCF is a fairly extensive architecture that was introduced with .net 3.5 as their new “recommended” way for .net programs to communicate with each other and with the rest of the world. It is configurable and extensible (mostly) and better yet it aims to align itself more strictly with the SOA paradigm that seems to be all the rage. This means that you can more easily communicate with non .net services through means of a messaging contract. While this is the recommended way of doing things, especially with external communications, there are numerous occasions when building .net to .net communications that this is unnecessary and results in large amounts of extra work. A Car POCO on the server side should be the same thing as a Car POCO on the client side. There is not a big need to define an extensive set of contracts because the same class can (and I argue should) be used on both sides of the communication architecture. WCF recognizes this and so has provided a means of serializing and deserializing .net classes called the

NetDataContractSerializer. This is their answer to the people who have been using .net Remoting up till now.

## 1.2 Why do I need Generic Messaging?

While WCF provides a means of sending .net types over the wire it seems to fall sadly short in terms of actually providing a simple way of “transparently” bridging the communications gap. The main fallback is the lack of support for “Open” Generics. Generics were introduced with .net 2.0 and provide a significant improvement in improving class structure when dealing with differing types of data. Suffice to say that if you are not using Generics then you probably don’t need the GenericMessaging library. Let’s take look at the following code.

```
class Program
{
    static void Main(string[] args)
    {
        IList<Car> cars = GetDataItems<Car>(null);

        Console.ReadKey();
    }

    static public IList<T> GetDataItems<T>(object queryObject)
    {
        //implementation goes here
    }
}
```

One might expect to find this kind of code in a Data Access class or more likely in an ORM layer. Without generics, especially in the data layer, we would need to write a separate set of CRUD methods for each persistent type we need to transfer and save. With generics we can take advantage of a modern ORM to handle data access as well as taking advantage of generic Business Layer classes.

The other shortfall of WCF is when handling reference types. When you mark a method parameter as “ref” using WCF the return reference is just copied over the original reference resulting in an almost useless scenario. Let’s consider the following example.

```
public class Car
{
    public string Name;
}

class Program
{
    static void Main(string[] args)
    {
        Car car = new Car();
    }
}
```

```

        car.Name = "Mercedes";

        Car currentCar = car;

        SomeMethod(ref car);

        Console.WriteLine(currentCar.Name);
        Console.ReadKey();
    }

    static public void SomeMethod(ref Car car)
    {
        car.Name = "BMW";
    }
}

```

From this example what will the program print out? If you guessed “**BMW**” you would be correct. This is due to pointers or references which .net tries hard to hide from you. When we copy “car” into “currentCar” all we are really doing is copying the address that “car” is located at, and when we pass “car” into “SomeMethod” the same thing happens. This ref behaviour is how c# works with compiled code. When working with ref parameters over WCF things are a little bit different, let's look at some code.

```

    static public void SomeMethod(ref Car car)
    {
        car = new Car();
        car.Name = "BMW";
    }
}

```

Really the only thing that has changed here is that we are initializing a new car inside of “SomeMethod”. When we run this program the output is now “**Mercedes**”. Why is this the case? Because the new object (car) now exists at a different reference than the old “car” did and “currentCar” is still pointing to that address. We have effectively nullified some of the benefits of passing “car” by reference. Anything that pointed to the old reference will still see the old object even though we updated the reference passed to our method.

Ok so what does this have to do with WCF? The second way of doing things is how WCF or serialization in general works. When deserializing an object using any of the standard .net serializers a new object **MUST** be created. There is no way to deserialize into an existing object. That means that everytime you send something over the wire, including a ref parameter, a new object is created on the return. Even if that new object's pointer is copied over the old one, references to the old object will still return old data. Normally this is not too much of a problem, but there are instances where it can be a big problem. The GenericMessaging library attempts to alleviate this issue by copying values back to their original objects on return. More details about how this is accomplished can be found in the architecture section.



## 2 Architecture

So now that I've explained a bit about it, it's time to examine how this actually works. Do you really need to know? No, you can use this library and not care. If this is you skip right ahead to implementation. However I know that quite a few people especially early adopters have a good understanding of the technology and before adopting something they want to know what's going on under the covers. If you are really hard core just head on over to <http://slagd.com> and use the svn link to get at the source code. Even if you do have an insatiable desire to go source snooping you will find that reading the following section will give you a much better understanding of the classes you look at. Understanding the GenericMessaging library first requires an understanding of WCF.

First it might be good to clarify that GenericMessaging does not aim to be an be-all to end-all. Its just a lightweight library meant to do its job and do it well. It does however aim to be extensible so that you can use it to solve your own particular problem.

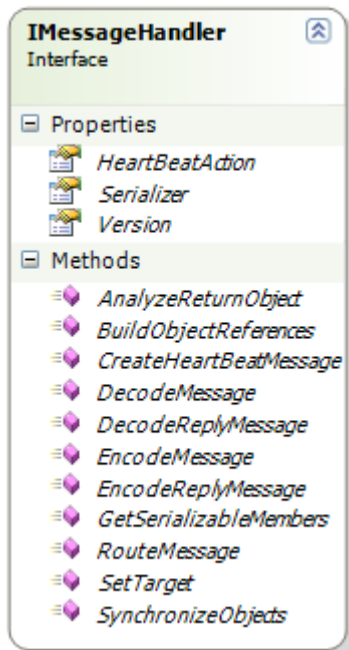
When we dig through all of the onion layers of WCF we come down to a simple messaging architecture. Raw message functionality is implemented by the Message class as defined in System.Runtime.Serialization. This is what is exchanged at the base by the standard WCF architecture. What I came to find out through research is that the ordinary developer – with a bit of work – can leverage these messages to create an entirely custom communication architecture. This is what is done in GenericMessaging, and it is at it's heart defined by this simple contract.

```
[ServiceContract(
    Namespace = "GenericMessaging",
    SessionMode = SessionMode.Required,
    CallbackContract = typeof(IMessageCallback))]
internal interface IInternalMessageService
{
    /// <summary>
    /// Entry point for remote message
    /// </summary>
    /// <param name="message"></param>
    [OperationContract(Action = "*", ReplyAction = "*", IsOneWay = false)]
    Message ToServerMessage(Message message);
}
```

There is only one method that is ever called. It's given a raw message and returns a raw message. The rest of the library is just classes to help support this.

The rest of GenericMessaging is composed of 3 main parts along with quite a few supporting pieces. We'll look at these pieces one by one and get a basic overview of how this library works.

## 2.2) MessageHandler



Before we can talk about the client class or the server class, we must first talk about the MessageHandler that is common to both of these pieces. The MessageHandler is really the most integral part of the GenericMessaging library. It is responsible for...

- Serializing and deserializing request and response objects into WCF “Messages”
  - Building a list of the object references that need to be updated from a “ref” method call
  - Routing the message to the appropriate method on the server side. (Later on the client side too)
  - Analyze the return object and resynchronize “ref” object back to their references.

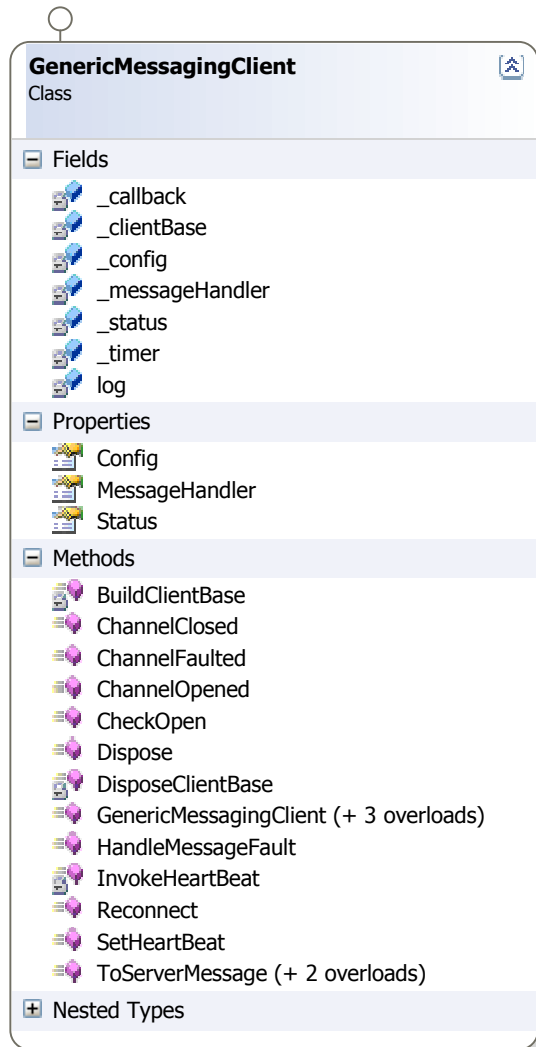


IMessageHandler is implemented by the generic MessageHandler<TContract> class. The MessageHandler class takes the contract you wish implement as a parameter. It also needs a concrete object of that type

in order to route its messages to that object. How and where you choose to implement your contract interface is up to you, although I will present several suggestions in the implementation section.

The MessageHandler class has another little trick up its sleeve. It’s called the ObjectGraphWalker, and it’s responsible for going through an object graph and performing some work on each item that it finds. You can use the IObjectGraphWalkerWorker to insert your own work items into the graph walker. We will elaborate more on this later.

## 2.3 ) GenericMessagingClient



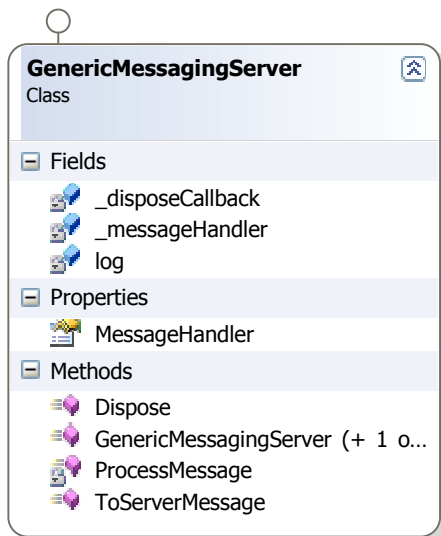
to expose it publicly.

This is the main class for the client side, and you will be interacting with this class for the majority of time you spend with this library. This client is a fairly simple wrapper around `RawMessageClient` and `RawDuplexMessageClient` which in turn extend WCF's `ClientBase` class and the `DuplexClientBase` class. The reason that we need to wrap this base class rather than just extending it is because when a WCF channel goes into the faulted state, it must be disposed of and recreated in order to attempt a reconnect. Why this is, I have no idea, but that is the reason this class is wrapped so that we can reconnect if it becomes faulted.

There are two ways of implementing this class on your client side. You can extend it, or you can wrap it in your own object. Both methods are supported and will be talked about in a little more detail in both the implementation, and the extension sections of this guide.

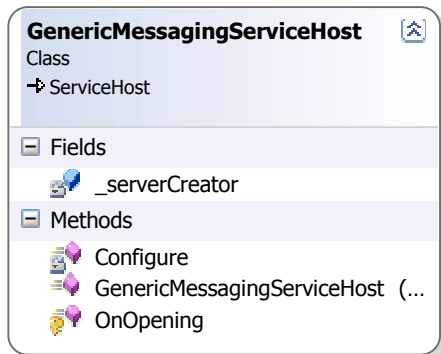
It's worthwhile to note that the callback portion of this class is not in use as of yet. It's still in experimental stage and as such I've chosen not

## 2.4) GenericMessagingServer



The server portion is really comprised of two parts. The `GenericMessagingServer` is the workhorse that contains a `MessageHandler` and is responsible for using that to route the incoming raw messages to the appropriate method. It will then process the return values and pass them back to the client. The server class is really the lightest part of the library right now as most of its work is done by the `MessageHandler`.

The `GenericMessagingServiceHost` extends the `WCF ServiceHost` to provide hooks for custom configuration as well as to provide support for an `IServerCreator` object. This is useful if you need more control over how your server classes are instanced.



## 2.5 Lifecycle

So what actually happens in terms of the lifecycle of a remote method?

1. A method is called on the class that implements the service contract. That method uses the "RemoteMethod" attribute and is marked as extern.
2. Handling of that method is passed to the RemoteMethod attribute. Method information is gathered and used along with the parameter values to form a MessageBody.
3. The MessageBody is passed to the GenericMessagingClient.
4. A list of object – Guid references are built for "ref" parameters
5. The MessageBody object is serialized to a WCF Message by the MessageHandler.
6. The WCF Message is sent through the WCF channel provided by ClientBase to the server
7. The WCF Message is received by the server
8. The server uses the MessageHandler to deserialize the Message into a MessageBody object
9. The server passes the MessageBody object to the "RouteMessage" method of the MessageHandler
10. The MessageHandler examines the MessageBody object and finds the correct method to execute from the TContract interface being handled.
11. If the appropriate method is generic the generic parameters are filled in from the MessageBody
12. The appropriate method is executed by the MessageHandler with the appropriate parameters.
13. The return value and any "out" or "ref" parameters are captured by the MessageHandler and packed into a ReplyMessageBody object.
14. The ReplyMessageBody is passed back to the GenericMessageServer
15. The server uses the MessageHandler to serialize the ReplyMessageBody into a WCF Message
16. The server passes the WCF Message back to the client
17. The client receives the Message and uses its MessageHandler to deserialize it back to a ReplyMessageBody object.
18. If the ReplyMessageBody is an exception then that exception is thrown,
19. The parameters in the ReplyMessageBody are processed. For "ref" parameters the objects are synchronized by MessageHandler. For "out" parameters the deserialized reference is simply copied over top of the old one. All parameters that are returned are passed through the "AnalyzeObject" method of the MessageHandler.
20. The return value is then passed through the "AnalyzeObject" method and then returned.

21. The return value and other “ref” and “out” parameters are returned by the original “RemoteMethod” that was called.

## 2.6 Limitations

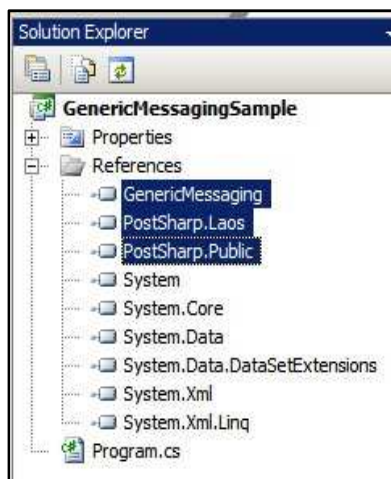
While this library does have some good functionality there are some things that it does not do. There are some things that are inherent simply by its nature and there are other things that are simply not implemented yet for reasons of time and practicality.

- (Inherent) This library uses the NetDataContractSerializer and all types and exceptions used must be serializable.
- (Implemented) Sessions are used when creating the channel.
- (Implemented) The two types of bindings used are the NetTcpBinding and the WsDualHttpBinding
- (Implemented) All methods are synchronous only.
- (Implemented) No duplex communication available yet.

## 3 Implementation

This is where we get to the fun part. Utilizing the GenericMessaging library in your own project is remarkably simple (Or so I think). In this section we will go through the steps necessary to get a project up and running in the logical order one would normally complete these steps. Along the way I will try to explain the various options that are available if you should choose to deviate from the norm. This is meant to be a cross between a quick start and a reference guide so please bear with me and skip the parts you don't need.

### 3.1 Setting up a project



This is where everyone starts. I will be using VS2008 for this project. There are actually three references you need to add in order to use GenericMessaging. You will need to add references to GenericMessaging.dll, PostSharp.Laos.dll, and PostSharp.Public.dll. The reason for adding the PostSharp references is that the "RemoteMethod" attribute uses this library in order to weave some code in at compile time.

I'd like to take this moment to say that [PostSharp](#) is an excellent tool for generating code at compile time and I would heartily recommend that you take a look at it.

### 3.2 Configuration

When I started out developing this library I was using the standard WCF configuration for everything. It was versatile and allowed for all different sorts of configuration. However as time went on and the design became solidified I realized that a proper configuration only utilized a tiny portion of the WCF configuration and so was needlessly complicated as well as being very prone to configuration failure. I decided to go with a dumbed down configuration that would limit a user's choices to things that I actually knew worked. If the configuration option that you need is not there, please contact me and I can probably add it in fairly quickly.

The configuration is based off of the GenericMessagingConfig and the ChannelConfig objects and their interfaces. You can choose to fill these structures out manually or you can use the app/web.config to assist you with this. They both accomplish the same task. Let's take a look at the structure for the app.config file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
```

```

        <section name="genericMessagingConfig"
type="GenericMessaging.Cfg.ConfigurationSectionHandler, GenericMessaging"/>
    </configSections>
    <genericMessagingConfig>
        <channel name="Default">
            <transport
address="net.tcp://webservices.localhost:4533/RemoteSession" type="tcp"/>
            <settings heartbeatInterval="60"/>
            <limits maxReceivedMessageSize="5242880"
maxStringLength="5242880" maxArrayLength="5242880" timeout="60" />
        </channel>
    </genericMessagingConfig>
</configuration>

```

This represents the basic structure of the configuration, and it mirrors the hierarchy of the configuration classes. The values expressed here are fairly self explanatory and almost all of them have defaults provided in case you don't wish to fill them out. The simplest possible config would look something this.

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="genericMessagingConfig"
type="GenericMessaging.Cfg.ConfigurationSectionHandler, GenericMessaging"/>
    </configSections>
    <genericMessagingConfig>
        <channel name="Channel2">
            <transport
address="http://webservices.localhost:4532/RemoteSession" type="http"/>
        </channel>
    </genericMessagingConfig>
</configuration>

```

All that you really need for a workable config is a channel element with the transport section filled out. "Address" is the address of the other end of the channel and "type" is the type of channel you want to create. There are only two possible values for type and they are "tcp" and "http". You may have also noticed that there is no separate config element for servers vs. clients. This is because the vast majority of the config at this point is the same for both clients and servers. There are some elements that may be ignored depending on if it is used client side or server side. One of those elements is the heartbeat. This is used to manually send a message ever X seconds to keep the channel alive. This element is only applicable on the client side. However if you use it server side it will not throw an exception and instead will simply be ignored.

There is a brief note I wanted throw in here about the channel name. This is simply to allow the configuration system to differentiate between multiple channel configs. It has no bearing on which channel to connect to on the other side. Which channel it will attempt to

connect to on the server side has everything to do with the address in the transport section of the config.

### 3.3 Service Contract

Before we can create the client or server we need to create a contract that specifies the operations that we want to implement. Unlike WCF though, we don't need to decorate this interface with any special attributes. This library is a bit more loose and accepting, but I do recommend that you make it disposable as when dealing with sessions it is a good idea to properly close both sides. Let's look at the contract I've chosen to use for the sample project.

```
/// <summary>
/// Defines the operations for the sample
/// </summary>
public interface ISampleContract : IDisposable
{
    /// <summary>
    /// Just returns the item passed to it
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="item"></param>
    /// <returns></returns>
    T GetItem<T>(T item);

    /// <summary>
    /// Shows a non generic overload
    /// </summary>
    /// <returns></returns>
    string GetItem();
}
```

That's all there is to it. We will be using this interface later as we build the client and the server.

### 3.4 Client Side

So you've made your configuration and you want to get started on programming your client. Firstly, as alluded to in the "Architecture" section, you have a choice to make. You can extend the `GenericMessagingClient` or you can wrap it. You can also mix and match, meaning you can extend on the client side and wrap on the server side or visa-versa. So let me show you both ways.

#### 3.4.1 Wrapped

##### Includes

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

using GenericMessaging;
using GenericMessaging.Cfg;
```

There is not much to see here except for the GenericMessaging and GenericMessaging.Cfg includes.

## Class Structure

```
/// <summary>
/// Client sample using wrapping technique
/// </summary>
public class WrappedClient : ISampleContract, IMessagingServiceClient
{
}
```

First we need to make sure our class implements the messaging contract that we defined earlier. That means that we will be using this class as the entry point for communications to our server. The second interface that we need to implement is IMessagingServiceClient. This interface ensures that we have a “ToServerMessage” method in this class.

## Private fields and Constructors

```
private GenericMessagingClient _messageClient;

public WrappedClient(string channelName)
    : this(GenericMessagingConfig.CreateFromConfig().GetChannel(channelName))
{
}

public WrappedClient(IChannelConfig config)
{
    //create the message handler
    IMessageHandler messageHandler = new
    MessageHandler<ISampleContract>(this);

    //create the inner client
    _messageClient = new GenericMessagingClient(messageHandler, config);
}
```

The constructor and its overload takes a channel name or an actual IChannelConfig object and uses this information to build an underlying message client. The most interesting code inside the constructor is the creation of the MessageHandler. The quirk about the message handler is that it requires a reference to the concrete object that implements our

messaging contract. In this case it is “this”. So when we create the underlying messaging client we pass it our new message handler.

## Methods

```
[RemoteMethod]
public extern T GetItem<T>(T item);

[RemoteMethod]
public extern string GetItem();

public object ToServerMessage(string action, IMessageBody body)
{
    return _messageClient.ToServerMessage(action, body);
}

public void Dispose()
{
    _messageClient.Dispose();
}
```

The most interesting part about the methods section are methods marked as “RemoteMethod”. This is the attribute that passes that method along through the “ToServerMessage” message to the server. It is possible to format the messages yourself and pass them manually to the message client, but that is a lot of work. It’s best just to mark them as “extern” and tag them with the “RemoteMethod” attribute as it will do all of the heavy lifting for you. I’ve also put a “Dispose” method as recommended here to clean up the underlying channel. It’s worthwhile to note that calling dispose on the underlying client will also dispose of our server instance as they are both linked by sessions. This ensures that you keep continuity between client and server.

### 3.4.2 Extended

Let’s take a look at the extension method which is remarkably similar except for a few key points.

## Includes

The includes are exactly the same as above.

## Class structure

```
public class ExtendedClient : GenericMessagingClient, ISampleContract
{
}
}
```

Let's note that even using the extended technique, our class still must implement the contract that we defined for our messaging. Oh and as indicated by the title "Extended" we extend this class directly from the `GenericMessagingClient`.

## Constructors

```
public ExtendedClient(string channelName)
    : this(GenericMessagingConfig.CreateFromConfig().GetChannel(channelName))
{
}

public ExtendedClient(IChannelConfig config)
    : base(new MessageHandler<ISampleContract>(null), config)
{
    //because we passed null into the message handler we must rectify that
    now
    MessageHandler.SetTarget(this);
}
```

Again the constructors are remarkable similar to the wrapped technique. The important difference comes in the creation of the `MessageHandler`. Because we need to pass a reference to the concrete implementation of our message contract into the `MessageHandler` and yet we need to pass the `MessageHandler` in the constructor, we will need to pass a null reference first and then later in the constructor we will have to set the target. This is kind of a backwards method of doing things but it does work and it will have to do until I can find something better.

## Methods

```
[RemoteMethod]
public extern T GetItem<T>(T item);

[RemoteMethod]
public extern string GetItem();
```

Because the `GenericMessagingClient` already implements things like "ToServerMessage" and "Dispose", we don't need to duplicate them here. That means our class only has to implement the methods defined in the message contract. Again we mark these methods as "extern" and tag them with the "RemoteMethod" attribute.

### 3.4.3 Overview

That's everything involved in setting up the client. Simple, right? Whether you pick wrapped or extended is up to you. Wrapped offers a bit better abstraction from the underlying technology, but Extended is a bit lighter and allows overrides of certain underlying methods.

## 3.5 Server Side

As with the client side there are two ways of implementing server side object. However unlike the client side, the server side doesn't follow the wrapped or extended techniques distinctly as there is the concept of a ServiceHost that gets thrown in by WCF. So after that scary disclaimer let's start with the easiest way first.

### 3.5.1 Extended

We are going to call this extended because it does actually extend the GenericMessagingServer however the reasons for extending it are not as much about adding functionality as it is about providing a constructor with no parameters. The main benefit about a parameterless constructor is that the service host can instantiate it much easier. You can make the decision later about its usefulness for you.

#### Includes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using GenericMessaging;
```

#### Class structure

```
/// <summary>
/// Demonstrates how to create a message server using the extension technique
/// </summary>
public class ExtendedServer : GenericMessagingServer, ISampleContract
{
}
}
```

Simple extension of the GenericMessagingServer, and make sure you implement that messaging contract that you defined earlier.

#### Constructors

```
public ExtendedServer()
    : base(new MessageHandler<ISampleContract>(null))
{
    MessageHandler.SetTarget(this);
}
```

There is only one parameterless constructor here. We use it to generate a MessageHandler and pass it along to the base class. Again we need to use the deferred initialization technique here to give the MessageHandler a concrete instance. I know a few of you are starting to recognize the limitations of this technique already. A parameterless constructor doesn't give you a lot of options when passing dependencies into your server class.

If this is the case you will want to keep reading as I will be providing a way in the next section to pass whatever parameters you want. For now lets just keep things simple.

## Methods

```
public T GetItem<T>(T item)
{
    return item;
}

public string GetItem()
{
    return "Hello from server!";
}
```

Here we just simply implment the methods for our message contract. This is just example code and you can do whatever you want here.

## Service Host

```
//get the config
IChannelConfig config =
GenericMessagingConfig.CreateFromConfig().GetChannel("Sample");

GenericMessagingServiceHost host = new
GenericMessagingServiceHost(typeof(ExtendedServer), config);

host.Open();

Console.WriteLine("The service is ready.");
Console.WriteLine("Press <ENTER> to terminate service.");
Console.WriteLine();
Console.ReadLine();

host.Close();
```

Unlike the client, which is self sufficient, the server must be served up by a service host. This means that regardless of whether you wrap or extend you must still use a service host to get your service up and running. The `GenericMessagingServiceHost` is an extended version of WCF's `ServiceHost` and allows for some additional functionality. Ok so what if you want to use IIS? This is possible but requires you to write some additional code at this point, and I will cover that after the next section.

### 3.5.2 Wrapped (ok I'm just calling it wrapped)

Wrapped is not really wrapped, instead it is a standalone class that implments your service contract. While it is not quite as simple to set up, it is more simple to implement and definitely provides more versatility than the aforementioned method. Let's dive in.

## Includes

There are no special includes needed, YAY!

## Class Structure

```
/// <summary>
/// Demonstrates how to create a message server using the wrapping technique
/// </summary>
public class WrappedServer : ISampleContract
{
}
}
```

Just implement your message service interface;

## Constructors

There are no special constructors in our simple implementation. You can, however, put whatever special constructors you need in this class because you will be able to have more control over how it is instanced.

## Methods

```
public T GetItem<T>(T item)
{
    return item;
}

public string GetItem()
{
    return "Hello from server!";
}

public void Dispose()
{
    //does nothing right now
}
```

There is nothing special here, just implement the methods defined in your message contract. Oh and we also have a dispose method so that we can clean up this object. We don't use it right now, but it's quite likely that a more complex class would need to be disposed of.

## Service Host

If you thought the simplicity of that last bit was too good to be true then yes you are correct. Because we are not extending from the `GenericMessagingServer` we cannot use our type to build a service host from. Why is this? This is because our class is not really a service,

and the real service interface lies within the GenericMessagingServer. So we need to find a way around this and that's where we get to the IServerCreator interface.

```
// Summary:
//     Defines a class that helps to create service server instances
public interface IServerCreator
{
    // Summary:
    //     Gets the service type
    Type ServiceType { get; }

    // Summary:
    //     Creates a new server instance
    GenericMessagingServer CreateServer();
}
```

This simple little interface will help us get our custom class instantiated. Let's take a look at the concrete implementation you will need to create and then things may start to make a little more sense.

```
/// <summary>
/// Demonstrates how to create a server creator
/// </summary>
public class WrappedServerCreator : IServerCreator
{
    #region IServerCreator Members

    public GenericMessagingServer CreateServer()
    {
        WrappedServer server = new WrappedServer();

        return new GenericMessagingServer(new
        MessageHandler<ISampleContract>(server), server.Dispose);
    }

    public Type ServiceType
    {
        get { return typeof(GenericMessagingServer); }
    }

    #endregion
}
```

What this class does is to hook into the service host and provide concrete implementations of your service class for it to work with. This implementation is fairly simple but your implementation may be more complex if you have dependencies that need to be injected into your service classes. The real work goes on inside of the "CreateServer" method where we create an instance of the WrappedServer class and pass that into a MessageHandler which in turn is passed into a new instance of the GenericMessagingServer. That instance is then passed off to WCF which hosts it. You will have noticed that we can also pass in a server disposed delegate which gets called as soon as the GenericMessagingServer is

disposed of. This allows us to perform cleanup work inside of our custom class. The `ServiceType` property is just there to tell the service host what the real type of the underlying service is. In our case that would be the `GenericMessagingServer`. The reason that this property exists is that in some cases you may be both extending the `GenericMessagingServer` and wrapping it as well and this gives you that flexibility. All in all the `IServerCreator` is there to allow you to do more complex instantiations of your service object. Let's say for example that you wanted to use `GenericMessaging` alongside `NHibernate`. Well, every time you made a new session you would want to pass in a new `ISession` to support your methods. This would allow you to do just that.

After the `IServerCreator` is made, the actual implementation is actually fairly simple. We simply pass the server creator along with the config as parameters into our `GenericMessagingServiceHost`.

```
//get the config
IChannelConfig config =
GenericMessagingConfig.CreateFromConfig().GetChannel("Sample");

//create a server creator
IServerCreator creator = new WrappedServerCreator();

//prepare the host
GenericMessagingServiceHost host = new GenericMessagingServiceHost(creator,
config);

host.Open();

Console.WriteLine("The service is ready.");
Console.WriteLine("Press <ENTER> to terminate service.");
Console.WriteLine();
Console.ReadLine();

host.Close();
```

### 3.5.3 Hosting in IIS

Ok so I promised you an explanation of how to host this in IIS. When hosting WCF from IIS you are used to seeing something like this at the top of your `.svc` file.

```
<% @ServiceHost Service="MyService" %>
```

IIS backs this line with something called the `ServiceHostFactory`. It is responsible for creating service hosts when IIS asks for them, and it doesn't have a clue about our new types and whatnot. That means that we are going to have to extend it, which is not a big deal. The reason I don't provide a generic implementation of this right now is the same reason why I don't provide an implementation of `IServerCreator`. There are just too many possibilities and it should be a fairly trivial thing for someone to write themselves.

```

public class SampleServiceHostFactory : ServiceHostFactory
{
    protected override System.ServiceModel.ServiceHost CreateServiceHost(Type
serviceType, Uri[] baseAddresses)
    {
        return new GenericMessagingServiceHost(serviceType, "Sample",
baseAddresses[0]);
    }
}

```

See, that wasn't too bad now, was it? The service host factory is responsible for giving a service host instance to IIS, and is activated by the @ServiceHost directive. That's where the bad news comes in. The @ServiceHost directive is **NOT** modifiable or extensible. This is why we are hard coding our channel name directly into the instantiation of the GenericMessagingServiceHost rather than properly passing it as a parameter. This is unfortunate but is unavoidable in my opinion. If you were using an IServerCreator object you would have to build that inside here as well. Please note that when using IIS to host your services, you shouldn't be specifying an address in the transport section and you also must specify "http" as the type, resulting in a config that looks something like the following.

```

<channel name="Sample">
    <transport address="" type="http"/>
    <limits maxReceivedMessageSize="5242880"
maxStringContentLength="5242880" maxArrayLength="5242880" timeout="30" />
</channel>

```

Now when we create our .svc file it would look something like this.

```

<%@ ServiceHost Language="C#" Debug="true"
Factory="GenericMessagingSample.SampleServiceHostFactory"
Service="GenericMessagingSample.ExtendedServer" %>

```

We can see from this line that we are now pointing the ServiceHost to the new factory which we just created. The service parameter still points to the type of the host and because we are using the extended host for this example we can just pass that as the service parameter.

### 3.5.4 Overview

The hosting part usually proves to be trickier than the client part simply because you usually have less granular control over how your server side objects are provided. Using an IServerCreator can help to alleviate some, but not all of these challenges.

## 4 Extension

Even with an extensive framework (not saying this one is), there are usage scenarios that are not covered. That's where extensibility comes into play. I've tried to build a few extensibility points into the GenericMessaging framework where I can see potential for additions. In this section we will talk about a few of them.

### 4.1 Source Code

Of course the most complete way to learn about and extend the GenericMessaging library is to view and/or modify the source code. You can find the source code online at <http://slagd.com>. There you will find the links to the subversion repositories where the latest source is kept. Simply use your favorite svn client to download it and use VS 2008 to compile it.

### 4.2 GenericMessagingClient

Modifying the source can be a bit of a brute force solution to your problems and is usually a last ditch solution. That's why the GenericMessagingClient has lots of virtual methods and is not sealed. I'll just list the methods here.

```
/// <summary>
/// Checks to see if this channel is good by sending a heartbeat
/// </summary>
/// <returns>True if the channel is open</returns>
public virtual bool CheckOpen()

/// <summary>
/// Attempts to reconnect to the server
/// Use this if the channel faults
/// </summary>
public virtual void Reconnect()

/// <summary>
/// Called when the channel is no longer available
/// </summary>
public virtual void ChannelFaulted(object sender, EventArgs e)

public virtual void ChannelClosed(object sender, EventArgs e)

public virtual void ChannelOpened(object sender, EventArgs e)

/// <summary>
/// sends a message to the server, gives a typed return
/// </summary>
/// <typeparam name="TReturn"></typeparam>
/// <param name="action"></param>
/// <param name="body"></param>
```

```

/// <returns></returns>
public virtual TReturn ToServerMessage<TReturn>(string action, IMessageBody
body)

/// <summary>
/// sends a message to the server
/// </summary>
/// <param name="action"></param>
/// <param name="body"></param>
/// <returns></returns>
public virtual object ToServerMessage(string action, IMessageBody body)

/// <summary>
/// Handles a fault in a message
/// </summary>
/// <param name="message"></param>
public virtual Exception HandleMessageFault(Message message)

```

The “ToServerMessage” method is the most interesting one as the bulk of the work that occurs when a message is sent or received happens here. It is also possible to mess things up by not properly extending these methods. However I have taken the position (unlike MS) that you guys are probably more intelligent than I and are more that capable of analyzing things and designing a solution that works, and who am I to stand in your way. Besides it’s open source and so No Warranty.

### 4.3 GenericMessagingServer

You can extend the server part as well, but there are far less interesting methods.

```

/// <summary>
/// Does the processing for an inbound message
/// </summary>
/// <param name="message"></param>
/// <returns></returns>
[LogDebug(LogEnter=true,LogExit=true)]
protected virtual Message ProcessMessage(Message message)

```

This is where the main handling for an inbound server message takes place. You can override this message but beware this is tapping into very low level functionality and is not really recommended

### 4.4 MessageHandler

Ok this is where all of the fun extension points lie. You can extend or replace the MessageHandler and then pass your modified version into the client or server when you create it. Let’s look at a few of the extension points.

```

/// <summary>
/// Routes a message to a particular object
/// Tries to match a method in the object to the action of the message
/// </summary>
/// <param name="action"></param>
/// <param name="body"></param>
/// <returns></returns>
public virtual IReplyMessageBody RouteMessage(string action, IMessageBody
body)

```

This is the big Kahula of the MessageHandler. It is responsible for analyzing the action and the parameters in the message body and then invoking the appropriate method with the parameters supplied. After execution has completed it returns the return values in the reply message body, or if there is an exception thrown it is responsible for storing that exception in the reply message body. You could replace this functionality if you wanted some custom routing scenario.

```

/// <summary>
/// Attempts to synchronize a list of old objects with new objects
/// </summary>
/// <param name="oldReferences"></param>
/// <param name="newReferences"></param>
public virtual void SynchronizeObjects(IDictionary<object, Guid>
oldReferences, IDictionary<object, Guid> newReferences)

```

This method is called when the return value is received by the client and is responsible for synchronizing “ref” parameters.

```

/// <summary>
/// Provided to allow subclasses to apply hooks to perform
/// special operations on return objects
/// </summary>
/// <param name="returnObject">This is the root return object</param>
/// <param name="workers">Additional workers to add in addition to the custom
ones already defined</param>
/// <param name="attributes"></param>
protected virtual void AnalyzeReturnObject(object returnObject,
GraphWorkerList workers, IEnumerable<CustomAttribute> attributes)

```

This method is called on the client side as well. When a method returns this allows you to inspect reply values and optionally change things inside of them. There are several overloads to this method but this is the deepest. The most interesting part about this method is the GraphWorkerList that is passed to it. Graph workers can either be inserted here or by using the CustomReturnObjectWorkers list that is exposed as a property on the MessageHandler class. We will talk a little more about graph workers in the next section.

There are few other extension points in the MessageHandler but they are mostly minor things.

## 4.5 Object Graph Workers

In my opinion this is the most useful extension, but that is probably because I use it myself in the NHibernate.Remote project. Basically when a return object graph comes back to the client there is sometimes a need to inspect in and possibly perform some fixups and such. A return object graph could come from a return value, an out or a ref parameter. The MessageHandler incorporates a reflection based analyzer that walks the object graph and optionally performs some work on each item that it finds. What work it does is up to you as defined by the IObjectGraphWalkerWorker interface. (Note that the object graph walker will not walk the graph if there are no workers present).

The IObjectGraphWalkerWorker is fairly simple and is defined in the GenericMessaging.Serialization namespace.

```
/// <summary>
/// Used by the object graph walker to define how work gets done
/// </summary>
public interface IObjectGraphWalkerWorker
{
    /// <summary>
    /// Does some processing work on a graph object
    /// </summary>
    /// <param name="graphObj"></param>
    void ProcessObject(object graphObj, Type type);
}
```

As the object graph walker walks the object graph it carries with it a list of workers. When it finds a non-primitive object it will call each worker in turn to allow it to perform some work on that item.

Let's take a look at an example. In NHibernate.Remote when lazy loaded collections or proxies are transferred to the client, obviously their internal sessions are no longer valid. If you tried to activate them it would just throw an error. So what we need to do is to create a fake ISession on the client and then every time one of those objects is transferred we will need to update it's internal session to point to the valid session of the client. Still following? Good. To do that we will make an object graph worker.

```
public class UpdateLazyItemWorker : IObjectGraphWalkerWorker
{
    #region private fields

    private ISessionImplementor _session;

    #endregion

    #region constructors
```

```

public UpdateLazyItemWorker(ISessionImplementor session)
{
    if (session == null)
        throw new ArgumentNullException("session");

    _session = session;
}

#endregion

#region IObjectGraphWalkerWorker Members

public void ProcessObject(object graphObj, System.Type type)
{
    if (typeof(INHibernateProxy).IsAssignableFrom(type))
    {
        //this is a lazy entity
        INHibernateProxy proxy = (INHibernateProxy)graphObj;
        proxy.HibernateLazyInitializer.Session = _session;
    }
    //is this a lazy collection?
    else if (typeof(IPersistentCollection).IsAssignableFrom(type))
    {
        //just update this collection
        IPersistentCollection lazyCollection =
(IPersistentCollection)graphObj;
        lazyCollection.SetCurrentSession(_session);
    }
}

#endregion
}

```

You can see that every time we get an object we check to see if it is a lazy collection or proxy object and if it is then we update it's associated session. That way these objects will still work on the client side the same as they worked on the server side. There is a bit of overhead associate with the reflection, however it is acceptable in the senarios in which I have tested it.

I've extended the MessageHandler in the NHibernate.Remote project and so I pass my custom worker while in the constructor like this.

```

base.CustomReturnObjectWorkers.Add(new
UpdateLazyItemWorker(_session.GetFakeSession()));

```

There are other places to extend the GenericMessaging library but these are the main one (or the ones I can think of right now).

## 5 Future and Notes

GenericMessaging is just meant to be a simpler way of getting your job done. As time goes on I'm sure there will be more features added to it. Or if you have added new feature or improved stability on it, I would be happy to incorporate your work and take full credit (oops sorry, I meant to say give YOU full credit).

I've marked this version at .5 because there are still some rough edges and I'm not happy with some of the structure but I think that its worthwhile to release it now. Please feel free to comment on the code or the quality because I'm always wanting to learn.

Daniel Guenter

<http://slagd.com>

nicao@slagd.com