.NET Remoting

Overview
- Introduction to .NET remoting
- Remoting basics
- Remoting events

References
- Fergal Grimes, “Microsoft .NET for Programmers”, Manning, 2002
What Is Remoting?

Remoting is the process of programs or components interacting across certain boundaries (e.g. processes or machines).

The remoting infrastructure is an abstract approach to interprocess communication. Much of the system functions without drawing attention to itself.

The .NET Remoting technology provides the foundation for distributed applications.

.NET Remoting replaces DCOM.
Copies vs. References

Cross-process communication requires a server object whose functionality is provided to callers outside its process, a client that makes calls on the server object, and a transportation mechanism to ferry the calls from one end to the other.

One way for the client to call a server object is to make a copy of the object and move it to the client process, where the copy's methods can be invoked directly.

Many objects, however, cannot or should not be copied and moved to some other process for execution. In these situations, the server process should pass to the client process a reference to the server object, not a copy of the object.
Remote Objects

Remote objects provide the ability to execute methods on remote servers, passing parameters and receiving return values.

A remote object always stays at the server, only a reference to it is passed around among other machines.

Clients can use this reference to call the server object. These calls do not execute in the client process. Instead, the remoting system collects all information about the call and sends it to the server process, where it is interpreted.
Mobile Object

- When mobile objects pass a context boundary, they are serialized (marshaled) into a general representation (either binary of XML), and then deserialized in the other context involved in the process.

- Server and client both hold copies of the same object.

- Methods executed on mobile objects will always be carried out in the local context, and no message will travel between processes or machines.

- After serialization and deserialization, the copied objects are indistinguishable from regular local objects, and there is no distinction between a server object and a client object.
Simplified Architecture

Client
- Client Object
- Proxy
- Formatter
- Transport Channel

Server
- Server Object
- Dispatcher
- Formatter
- Transport Channel

Com S 430
Using Remoting

Using object references to communicate between server objects and clients is the heart of remoting.

If configured properly, one only needs to create a new instance of the remote object using `new` (or some instance-creation function). As a result, the client will receive a reference to the server object, which can be used to call the object’s methods as though the object runs in your process and not on a separate computer.
The remoting system uses proxy objects to create the impression that a server object runs in the client's process (i.e., .NET remoting uses the so-called transparent networking approach).

When a client creates an instance of a remote type, the remoting infrastructure creates a proxy object that looks to the client exactly like the remote type.

If a client calls a method on that proxy, then the remoting system receives the call, routes it to the server process, invokes the server object, and returns the return value to the client proxy, which returns the result to the client object.
Transport Channels

- A channel is a type that takes a stream of data, creates a package according to a particular network protocol, and sends the package to another computer.

- Some channels can only receive information, others can only send information, and others, such as the default TcpChannel and HttpChannel classes, can be used in either direction.
  - A TcpChannel provides an implementation for a sender-receiver channel that uses the TCP protocol to transmit messages.
  - A HttpChannel provides an implementation for a sender-receiver channel that uses the HTTP protocol to transmit messages.
First Example

- The first remoting application exposes a server-side `MarshalByRefObject` in `Singleton` mode.
- The object is called `CustomerManager`, and it provides a method to load a `Customer` object (a `ByValue` object) from some fictitious database.
- The resulting object is then passed as a copy to the client.
When using remote objects, both client and server must have access to the same interface definitions and serializable objects that are passed by value!

In general, a .NET remoting project will consist at least of three assemblies:

- A shared assembly, which contains serializable objects and interfaces or base classes to `MarshalByRefObject`.
- A server assembly that implements the `MarshalByRefObject`.
- A client assembly that consumes them.
ICustomerManager is the interface that will be implemented by the remoting server. Both client and server will use this interface to exchange a remoting object.

```java
public interface ICustomerManager {
    ICustomer GetCustomer(int aCustomerId);
}
```
The Data Object

The data object needs to be passed as a copy (by value), so it must be marked with the *Serializable* attribute.

The class Customer defines four properties: *FirstName, LastName, and DateOfBirth*, and *Age* to calculate a customer’s age.

Occasionally Customer objects will print a message to the console, so that one can see in which context a specific operation is executing.
public interface ICustomer
{
    string FirstName { get; set; }
    string LastName { get; set; }
    DateTime DateOfBirth { get; set; }
    int Age { get; }
}
The Class Customer

[Serializable]
public class Customer : ICustomer
{
    // public properties …

    public Customer()
    {
        Console.WriteLine( "Customer object created!" );
    }
}
Implementing the Server

- The server is implemented as a console application.
- The server assembly defines a reference to both System.Runtime.Remoting and the newly created shared assembly (a DLL, namespace Shared).
- The server will have to access to following namespaces:
  - System.Collections
  - System.Runtime.Serialization.Formatters
  - System.Runtime.Remoting
  - System.Runtime.Remoting.Channels
  - Shared
class CustomerManager : MarshalByRefObject, ICustomerManager
{
    public CustomerManager() { … } 

    public ICustomer GetCustomer( int aCustomerId )
    {
        ICustomer tmp = new Customer();
        tmp.FirstName = "James";
        tmp.LastName = "Bond";
        tmp.DateOfBirth = new DateTime(1962, 6, 13);
        return tmp;
    }
}

// .NET 1.1 requirement
BinaryServerFormatterSinkProvider lServerProvider =
    new BinaryServerFormatterSinkProvider();
lServerProvider.TypeFilterLevel = TypeFilterLevel.Full;
BinaryClientFormatterSinkProvider lClientProvider =
    new BinaryClientFormatterSinkProvider();
IDictionary lProperties = new Hashtable();
lProperties["port"] = 1234;

HttpChannel lChn = new HttpChannel( lProperties, lClientProvider, lServerProvider );
ChannelServices.RegisterChannel( lChn );

RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(CustomerManager), // type
    "CustomerManager.soap", // objectUri
    WellKnownObjectMode.Singleton ); // mode
public class BasicRemoting {

    public static void SetupHttpChannel()
    {
        // channel properties
        IDictionary lProperties = new Hashtable();
        RegisterHttpChannel( lProperties );
    }

    public static void SetupHttpChannel( int Port )
    {
        // channel properties
        IDictionary lProperties = new Hashtable();
        lProperties["port"] = Port;
        RegisterHttpChannel( lProperties );
    }

    ...
}
private static void RegisterHttpChannel(IDictionary Properties)
{
    // sink setup
    BinaryServerFormatterSinkProvider IServerProvider =
        new BinaryServerFormatterSinkProvider();
    IServerProvider.TypeFilterLevel = TypeFilterLevel.Full;
    BinaryClientFormatterSinkProvider IClientProvider =
        new BinaryClientFormatterSinkProvider();

    // register fresh HTTP channel
    HttpChannel lChn = new HttpChannel(Properties, IClientProvider, IServerProvider);
    ChannelServices.RegisterChannel(lChn);
}
public static void SetupServer( Type RemoteType, 
    string URI, 
    WellKnownObjectMode Mode )
{
    RemotingConfiguration.RegisterWellKnownServiceType( 
        RemoteType, // Remote Object Type 
        URI, // Remote Object Uri 
        Mode ); // Remote Object Mode 
}
class ServerMain
{
    static void Main(string[] args)
    {
        BasicRemoting.SetupHttpChannel(1234);
        BasicRemoting.SetupServer(typeof(CustomerManager),
                                "CustomerManager.soap",
                                WellKnownObjectMode.Singleton);
        Console.WriteLine("Server started.");
        Console.WriteLine("Greetings and felicitation.");
        // The server will keep running until any key is hit
        Console.Read();
    }
}
class ClientMain
{
    static void Main(string[] args)
    {
        BasicRemoting.SetupHttpChannel();

        ICustomerManager lMgr =
            (ICustomerManager)Activator.GetObject(
                typeof(ICustomerManager),
                "http://localhost:1234/CustomerManager.soap" );

        Console.WriteLine( "Reference to CustomerManager acquired!" );

        ICustomer lCust = lMgr.GetCustomer( 007 );
        int lAge = lCust.Age;
        Console.WriteLine( "Customer {0} {1} is {2} years old.",
                            lCust.FirstName, lCust.LastName, lAge );
        Console.Read();
    }
}
Server started.
Greetings and felicitation.
CustomerManager created.
Customer object created!

Reference to CustomerManager acquired!
Calculating age of James, born on 6/13/1962
Customer James Bond is 49 years old.
Types of Remoting

There are two very different types of remote interaction between components:

- Serializable objects that are passed as a copy to the remote process,
- Server-side (remote) objects that allow clients to call their methods.
ByValue Objects

- Marshalling objects by value means to serialize their state (instance variables), including all objects, which are referenced by instance variables, to some persistent form which they can be deserialized in a different context.

- The ability to serialize objects is provided by the .NET framework when a class is annotated with the `Serializable` attribute or if it implements `ISerializable`.

- When a ByValue object contains references to other objects, these objects have to be either serializable or MarshalByRefObjects.
A MarshalByRefObject is a remote object that runs on the server and accepts method calls from clients.

The instance data is stored in the server’s memory and methods are executed in the server’s application domain.

MarshalByRefObject can be categorized into two groups:

- Server-activated objects (SAOs),
- Client-activated objects (CAOs).
Server-Activated Objects

- Server-activated objects are somewhat comparable to stateless Web Services.

- When a client requests a reference to a SAO, no message will travel to the server. Only when methods are called on this remote reference will the server be notified.

- Depending on the configuration of the SAOs, the server decides whether a new instance needs to be created or an existing object is reused, when a client requests a SAO.
Singleton or SingleCall

SAOs can be marked as

- **Singleton**
  One instance serves the requests of all clients in a multithreaded fashion.

- **SingleCall**
  A new instance is created for each request and destroyed afterwards.
Second Example

To illustrate the differences between the two kinds of services we create a new assembly Shared.dll, which contains an interface to a very simple remote object that allows the storage and retrieval of stateful information.

```csharp
namespace Shared
{
    public interface IRemoteObject
    {
        int Value { set; get; }
    }
}
```
**Simple Client**

- The client provides the means for opening a connection to the server and tries to set and retrieve the instance values of the server-side remote object.

- The client needs to have access to the following namespaces:
  - System
  - System.Collections
  - System.Runtime.Serialization.Formatters
  - System.Runtime.Remoting
  - System.Runtime.Remoting.Channels
  - Shared
static void Main(string[] args) {
    ...
    IRemoteObject lObj = (IRemoteObject)Activator.GetObject(
        typeof(IRemoteObject),
        "http://localhost:1234/RemoteObject.soap" );
    Console.WriteLine( … );
    int tmp = lObj.Value;        // Get initial value
    Console.WriteLine( … );
    Console.WriteLine( … );
    lObj.Value = 42;              // Set new value
    tmp = lObj.Value;            // Get updated value
    Console.WriteLine( … );
}

SingleCall Objects

- For SingleCall objects the server will create a single object, execute the method, and destroy the object again.

- SingleCall objects are registered at the server using the following statement:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(<RemoteObjectClass>),
    "<ObjectURI>",
    WellKnownObjectMode.SingleCall );
```

- Note, objects of this kind cannot maintain any state!
class RemoteObject : MarshalByRefObject, IRemoteObject
{
    private int fValue; // private state

    public RemoteObject() {}
    public RemoteObject(int aValue) {} // set initial value

    public int Value {
        set { Console.WriteLine("…");
             fValue = value; }

        get { Console.WriteLine("…");
             return fValue; }
    }
}
class ServerMain
{
    static void Main(string[] args)
    {
        Console.WriteLine("ServerStartup.Main(): Server started.");
        //
        // Remoting setup …
        //
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(RemoteObject), // type
            "RemoteObject.soap", // objectUri
            WellKnownObjectMode.SingleCall ); // mode
        // Keep server alive
        Console.Read();
    }
}
Every interaction with the server triggers the instantiation of a new remote object!
Client Output

Client.Main(): Reference to remote object acquired.
Client.Main(): Original server-side value is "0".
Client.Main(): Set value to 42.
Client.Main(): New server-side value is "0".
Singleton Objects

For Singleton objects the server will create only one object at any given time. When the server receives a request, it checks whether the corresponding object already exists. If not, a new object is created.

Singleton objects are registered at the server using the following statement:

```csharp
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(<RemoteObjectClass>),
    "<ObjectURI>",
    WellKnownObjectMode.Singleton);
```

Note, objects of this kind maintain a state!
class ServerMain
{
    static void Main(string[] args)
    {
        Console.WriteLine("ServerStartup.Main(): Server started.");
        //
        // Remoting setup ...
        //
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(RemoteObject), // type
            "RemoteObject.soap", // objectUri
            WellKnownObjectMode.Singleton ); // mode

        // Keep server alive
        Console.Read();
    }
}
Singleton Server Output

Every interaction with the server is dispatched to the same remote object!
Client Output

The remote object maintains a state!
Client-Activated Objects

- A client-activated object (CAO) behaves mostly the same way as does a “normal” .NET object.

- A client-activated object’s lifetime is managed by the same lifetime service used by SAOs.

- CAOs are so-called stateful objects. An instance variable that has been set by the client can be retrieved again and will contain the correct value.
CAO Creation

- CAOs are explicitly created by the client using the `new` operator, so they can have distinct constructors like normal .NET objects do.

- Unfortunately, this form of object creation has one serious drawback: one cannot use shared interfaces or base classes. This means that either
  - one has to ship the compiled objects to the client deployment side or
  - one has to use the tool “SoapSuds” to extract the metadata, an approach that will only export the default constructor.
Soapsuds Tool (Soapsuds.exe)

Soapsuds.exe performs the following functions:

- It creates XML schemas describing services exposed in a common language runtime assembly.
- It creates runtime assemblies to access services described by XML schemas. A schema definition can be a local file or it can be dynamically downloaded from the Internet.

Example:

```
soapsuds -ia:server -nowp -oa:metadata.dll
```

Com S 430
The CAO Remote Object

// must be public in order to extract meta-data
public class RemoteObject : MarshalByRefObject
{
    private int fValue;

    public RemoteObject() { … } public RemoteObject( int aValue ) { … fValue = aValue; … }

    public int Value
    {
        set { … fValue = value; … }
        get { … return fValue; }
    }
}

One can use a non-default constructor!
class ServerMain
{
    static void Main(string[] args)
    {
        Console.WriteLine("ServerStartup.Main(): Server started."");

        // Remoting setup ...

        RemotingConfiguration.ApplicationName = "CAOServer";
        RemotingConfiguration.RegisterActivatedServiceType(typeof(RemoteObject));

        // Keep server alive
        Console.Read();
    }
}
Soapsuds & Ildasm

RemoteObject type information
Metadata Extraction

Command line:

> soapsuds -ia:CAOServer -nowp -oa:RemoteObject.dll

Visual Studio .NET 2003
The Client

```csharp
static void Main(string[] args) {
    // Remoting setup ...

    RemotingConfiguration.RegisterActivatedClientType(
        typeof(RemoteObject),
        "http://localhost:1234/CAOServer" );

    Console.WriteLine( "Client.Main(): Creating first object." );
    RemoteObject lObj1 = new RemoteObject(); lObj1.Value = 47;
    Console.WriteLine( "lObj1.Value: {0}.", lObj1.Value  );
    Console.WriteLine( "lObj2.Value: {0}.", lObj2.Value );
    Console.WriteLine( "lObj1.Value: {0}.", lObj1.Value  );
    Console.WriteLine( "lObj2.Value: {0}.", lObj2.Value );
    ...
```
The Server Output

Objects created on the server!
The Client Output

Client.Main(): Creating first object.
Client.Main(): Creating second object.
Obj1.Value: 47.
Obj2.Value: 4711.
Encapsulate RemoteObject in a shared library:
The Client

```csharp
static void Main(string[] args) {
    // Remoting setup …

    RemotingConfiguration.RegisterActivatedClientType(
        typeof(RemoteObject),
        "http://localhost:1234/CAOServer"
    );

    Console.WriteLine( "Client.Main(): Creating first object." );
    RemoteObject lObj1 = new RemoteObject(); lObj1.Value = 47;
    Console.WriteLine( "lObj1.Value: {0}.", lObj1.Value  );
    Console.WriteLine( "lObj2.Value: {0}.", lObj2.Value );
    ... }
```
CAO2 Server

ServerStartup.Main(): Server started.
RemoteObject.Value(set): old 0 new 47.
RemoteObject.Constructor: .ctor called with 4711.
RemoteObject.Value(get): current 47.
CAO2 Client

Client.Main(): Creating first object.
Client.Main(): Creating second object.
Obj1.Value: 47.
Obj2.Value: 4711.
Abstract Factory

**Intent:**
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Collaborations:**
- Normally a single instance of a ConcreteFactory class is created at runtime. This concrete factory creates product objects having a particular implementation.
- To create different product objects, clients should use a different concrete factory.
- `AbstractFactory` defers creation of product objects to its `ConcreteFactory` subclass.
Structure of Abstract Factory

- **AbstractFactory**
  - CreateProductA()
  - CreateProductB()

- **ConcreteFactory1**
  - CreateProductA()
  - CreateProductB()

- **ConcreteFactory2**
  - CreateProductA()
  - CreateProductB()

- **AbstractProductA**
  - ProductA1
  - ProductA2

- **AbstractProductB**
  - ProductB1
  - ProductB2

Com S 430
namespace FactoryDesignPattern
{
    class ProductClass
    {
        public ProductClass() { … }
        public ProductClass(int aInitValue) { … }
    }

    class FactoryClass
    {
        public ProductClass GetNewInstanceOf() {
            return new ProductClass();
        }
        public ProductClass GetNewInstanceOf(int aInitValue) {
            return new ProductClass(aInitValue);
        }
    }
}
Abstract Factory Client

using FactoryDesignPattern;

class ClientClass
{
    public static void Main()
    {
        // create a new factory
        ProductFactory IFactory = new ProductFactory();

        // create objects
        ProductClass IObj1 = IFactory.GetInstance();
        ProductClass IObj2 = IFactory.GetInstance(4);
    }
}
namespace Shared
{
    public interface IRemoteObject
    {
        int Value{ get; set; }
    }
}

public interface IRemoteFactory
{
    IRemoteObject GetNewInstance();
    IRemoteObject GetNewInstance( int aInitValue );
}

The Remote Object

class RemoteObject : MarshalByRefObject, IRemoteObject
{
    private int fValue; // private state

    public RemoteObject() {}
    public RemoteObject( int aValue ) {} // set initial value

    public int Value {
        set { Console.WriteLine( … );
             fValue = value; }
        get { Console.WriteLine( … );
             return fValue; } }
}
The Remote Object Factory

class RemoteFactory : MarshalByRefObject, IRemoteFactory
{
    // The factory defines a default constructor
    public RemoteFactory() { … }

    public IRemoteObject GetNewInstance()
    {
        return new RemoteObject(); // ByRef-remoting object
    }

    public IRemoteObject GetNewInstance( int aInitValue )
    {
        return new RemoteObject( aInitValue );
    }
}
class ServerMain
{
    static void Main(string[] args)
    {
        Console.WriteLine( "ServerStartup.Main(): Server started." );
        HttpChannel lChn = new HttpChannel( 1234 );
        ChannelServices.RegisterChannel( lChn );
        RemotingConfiguration.RegisterWellKnownServiceType( typeof(RemoteFactory), // type
            "RemoteFactory.soap", // objectUri
            WellKnownObjectMode.Singleton ); // mode

        // Keep server alive
        Console.Read();
    }
}
static void Main(string[] args) {

    IRemoteFactory IFactory = (IRemoteFactory)Activator.GetObject(
        typeof(IRemoteFactory),
        "http://localhost:1234/RemoteFactory.soap" );

    Console.WriteLine( "..." );

    IRemoteObject IObj1 = IFactory.GetInstance();
    IObj1.Value = 42;    // Set new value

    // create a new RemoteObject with startup value "4711"
    IRemoteObject IObj2 = IFactory.GetInstance( 4711 );
    Console.WriteLine( "IObj1.Value: {0}.", IObj1.Value );
    Console.WriteLine( "IObj2.Value: {0}.", IObj2.Value );

}
RemoteObject Constructor: New object created.
RemoteObject.Value(set): old 0 new 42.
RemoteObject(Constructor): .ctor called with 4711.
RemoteObject.Value(get): current 42.

Client.Main(): Reference to remote factory object acquired.
10bj1.Value: 42.
10bj2.Value: 4711.
Remoting Problems

In .NET Remoting, the intuitive way of solving a problem might not be the correct one.

Example:
You have to implement a type of broadcast application in which a number of clients register themselves at the server as listeners and other clients can send messages that will be broadcast to all listening clients.
Changing Roles

When an event occurs, client and server will change roles. This means that the client in reality becomes the server (for the callback method), and then server will act as a client and try to contact the “real” client.
namespace Shared
{

    public delegate void MessageArrivedHandler(string aMessage);

    public interface IBroadcaster
    {
        void BroadcastMessage(string aMessage);

        event MessageArrivedHandler MessageArrived;
    }

}
Remote Broadcaster

```csharp
public class Broadcaster : MarshalByRefObject, IBroadcaster
{
    public event MessageArrivedHandler MessageArrived;

    public void BroadcastMessage( string aMessage )
    {
        // call the delegate to notify all listeners
        Console.WriteLine( "Will broadcast message: {0}" , aMessage );
        MessageArrived( aMessage );
    }
}
```

We use the standard technique to register Broadcaster as a Singleton remote object.
The Client Event Handler

A client defines an event handler (callback).

```csharp
public class EventHandler : MarshalByRefObject
{
    public void HandleMessage( String aMessage )
    {
        Console.WriteLine( "Received: {0}", aMessage );
    }
}
```
The Broadcast Client

```csharp
static void Main(string[] args)
{
    ...

    Console.WriteLine( "Registering event at server." );
    EventHandler lHandler = new EventHandler();
lObj.MessageArrived +=
    new MessageArrivedHandler( lHandler.HandleMessage );

    Console.WriteLine( "Event registered. Waiting for messages." );

    Console.Read();

    ...
}
```
Output

Microsoft Development Environment

An unhandled exception of type 'System.Runtime.Serialization.SerializationException' occurred in mscorlib.dll

Additional information: Cannot find the assembly Client, Version=1.0.2112.36908, Culture=neutral, PublicKeyToken=null.
Solution

- We introduce an intermediate wrapper \textit{MarshalByRefObject} (including the implementation) that will be located in Shared.dll.

- The wrapper object is created in the client’s context and provides an event that can be used to call back the “real” client.

- The server receives a delegate to the wrapper’s \textit{HandleMessage()} method. This method activates the wrapper’s \textit{MessageArrived} event, which will be handled by the client.
The System

Client 1

Wrapper object

HandleMessage()

Client object

Server

Event delegate

HandleMessage()
namespace Shared
{
    public class BroadcastEventWrapper : MarshalByRefObject
    {
        public event MessageArrivedHandler MessageArrived;

        public void HandleMessage(string aMessage)
        {
            // forward the message to the client
            MessageArrived(aMessage);
        }
    }
}
class ClientMain
{
    static void Main(string[] args) {
        // create event wrapper in client's context
        BroadcastEventWrapper lEventWrapper = new BroadcastEventWrapper();
        // register the local handler with the "remote" handler
        EventHandler lHandler = new EventHandler();
        lEventWrapper.MessageArrived +=
            new MessageArrivedHandler(lHandler.HandleMessage);
        // register wrapper with remote event
        IObj.MessageArrived +=
            new MessageArrivedHandler(lEventWrapper.HandleMessage);
        ...
    }
}
Event Initiator

```csharp
static void Main(string[] args)
{
    HttpChannel lChn = new HttpChannel();
    ChannelServices.RegisterChannel( lChn );

    IBroadcaster lObj = (IBroadcaster)Activator.GetObject(
        typeof(IBroadcaster),
        "http://localhost:1234/Broadcaster.soap" );

    lObj.BroadcastMessage( "Hello World!" );

    Console.Read();
}
```

Additional information: This remoting proxy has no channel sink which means either the server has no registered server channels that are listening, or this application has no suitable client channel to talk to the server.
Enable Client Sink

```csharp
class ClientMain
{
    
    static void Main(string[] args)
    {
        HttpChannel lChn = new HttpChannel(0);
        ChannelServices.RegisterChannel( lChn );

        IBroadcaster lObj = (IBroadcaster)Activator.GetObject( typeof(IBroadcaster),
            "http://localhost:1234/Broadcaster.soap" );
    }
}
```

Must be 0 to create a channel sink!
Output

ServerStartup.Main(): Server started.
Will broadcast message: Hello World!

Registering event at server.
Event registered. Waiting for messages.
Received: Hello World!
A Chat Application

A chat application (chat room) consists of:

- A chat server that broadcasts messages to all participating clients.
- Clients that register themselves at the server in order to exchange messages with other clients.

We need to implement:

- Shared.dll that provides type definitions used by both the server and the client.
- A server console application.
- A GUI client.
namespace Shared
{
    public delegate void MessageArrivedHandler( IChatter aSender, string aMessage );

    public interface IBroadcaster
    {
        void BroadcastMessage( IChatter aSender, string aMessage );
        event MessageArrivedHandler MessageArrived;
    }

    public interface IChatter
    {
        string Name { get; }
    }
}
[Serializable] // ByVal remoting object
public class Chatter : IChatter
{
    private string fName;

    public Chatter() : this("<NONAME>") { }

    public Chatter(string aName) { fName = aName; }

    public string Name
    {
        get { return fName; }
    }
}

Use standard name
public class BroadcastEventWrapper : MarshalByRefObject
{
    public event MessageArrivedHandler MessageArrived;

    public void HandleMessage( IChatter aSender, string aMessage )
    {
        // forward the message to the client
        MessageArrived( aSender, aMessage );
    }
}
public class Broadcaster : MarshalByRefObject, IBroadcaster
{
    public event MessageArrivedHandler MessageArrived;

    public void BroadcastMessage( IChatter aSender, string aMessage )
    {
        // call the delegate to notify all listeners
        if ( MessageArrived != null )
        {
            Console.WriteLine( … );
            MessageArrived( aSender, aMessage );
        }
        else
        {
            Console.WriteLine( … );
        }
    }
}
Chat Client

The remoting initialization is the same as in broadcast client.
public class NameForm : System.Windows.Forms.Form
{
    ... public string ChatName { get { return fNameTextBox.Text; } } ...
}
private void HandleMessage( IChatter aChatter, string aMessage )
{
    // sequence "\r\n" is required to add a newline
    fMessages.AppendText( aChatter.Name + ": " + aMessage + "\r\n" );
}
Output

ServerStartup.Main(): Server started.
Will broadcast message: Hi Paul, how are you? from Markus
Will broadcast message: I'm fine. from Paul

Chat

Message:
Markus: Hi Paul, how are you?
Paul: I'm fine.

Chat

Message:
Paul: I'm fine.

Chat
Still Problems

One client has become unreachable!
The Solution

```csharp
public class Broadcaster : MarshalByRefObject, IBroadcaster
{
    public event MessageArrivedHandler MessageArrived;

    public void BroadcastMessage( IChatter aSender, string aMessage )
    {
        MessageArrivedHandler lHandler = null;
        foreach( Delegate lDelegate in MessageArrived.GetInvocationList() )
        {
            try {
                lHandler = (MessageArrivedHandler)lDelegate;
                lHandler( aSender, aMessage );
            }
            catch { MessageArrived -= lHandler; }
        }
    }
}
```
Output

Unreachable client removed
Lifetime

Lease has expired!
public class Broadcaster : MarshalByRefObject, IBroadcaster
{
    ...

    // unlimited lifetime
    public override object InitializeLifetimeService()
    {
        return null;
    }

    ...
}