Sistemi Distribuiti
Corso di Laurea in Ingegneria

Prof. Paolo Nesi

PARTE 10: .net Remoting
Department of Systems and Informatics
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-4796523, fax: +39-055-4796363

Lab: DISIT, Sistemi Distribuiti e Tecnologie Internet
nesi@dsi.unifi.it nesi@computer.org
www: http://www.dsi.unifi.it/~nesi

Distributed Applications

- Intranet model
  - .NET Remoting
  - .NET to .NET

- Internet model
  - Web services
  - HTTP to HTTP
  - Ad hoc services for .NET clients
.NET Remoting

Set of services that enable applications to communicate
- Applications can reside on the same computer
- On different computers in the same LAN
- Across the world in “very” different networks and running on heterogeneous platforms

Enable communication between objects in different AppDomains or processes
- Different transportation protocols
- Different serialization formats
- Object lifetime schemes
- Modes of object creation

AppDomains

Managed code runs in an application domain
- AppDomains are separate units of processing that the CLR recognizes in a running process
- AppDomains are separated from one another
  - Similar to process boundaries but more lightweight

Diagram of AppDomains and related concepts:
- dotnet.exe
- Unmanaged Stub
- CLR Host
- Managed Code
- Default AppDomain
- Isolation
- Others
Isolation

- Managed code must get through a verification process before it can be run
  - Code that passed the test is said to be type-safe
- The certainty to run type-safe code allows the CLR to provide a level of isolation as strong as the process boundary, but more cost-effective
- The CLR enforces isolation by preventing direct calls between objects in different AppDomains
- .NET Remoting refers to the set of system services to access objects between domains

Remoting and AppDomains

- .NET Executable
  - Stub
  - Instantiates the CLR host and inject code in the default AppDomain
- Process primary thread
  - Default AppDomain
  - Other AppDomains
- Remoting (local or remote machine)

- .NET Executable
  - Stub
  - Instantiates the CLR host and inject code in the default AppDomain
- Process primary thread
  - Default AppDomain
  - Other AppDomains
- Remoting
Marshaling

- Marshal by value
  - Object is instantiated on the client and filled with data coming from the server
  - State of the object downloaded
- Marshal by reference
  - The object lives on the server and any calls take place remotely
  - Input parameters and return value travel over the network

Marshaling

- MarshalByRef
  - Client
    - Proxy
    - stub
      - Object, method, and params
    - AppDomain
  - Server
    - Proxy
    - stub
      - Object
    - Return values
    - AppDomain
Remote Components

- Class derived from MarshalByRef
- Public methods
- Work with serializable classes
- Application Host
  - IIS, NT service, custom app
  - Requires manual activation
  - Publish your components

Client Applications

- Need a reference to the server component
- Mark the remote type as "well-known"
  - Tell the run-time the type lives remotely
  - JIT compiler adds code on-the-fly to transform local calls into remote calls
  - Everything happens transparently
- Instantiated through operator new (if client activated)
Serializable objects

- When runtime can make a copy of the object, an object can marshal-by-value to other AppDomains
  - Add `SerializableAttribute`
  - Or implement the `ISerializable` interface

```csharp
[Serializable]
class Foo {
    . . .
}
```

- Clients in foreign AppDomains receive clone

---

MarshalByRef objects

- When the object's class derives, directly or indirectly, from `MarshalByRefObject`, the runtime creates a proxy to the object

```csharp
[Serializable]
class Foo : System.MarshalByRefObject {  // MBRO overrides
    . . .
    // Serializable
}
```

- Clients in foreign AppDomains receive proxy
- How does a client specify what proxy?
Typical Remoting scenario

- Clients must connect to servers
  - Server needs to publish an object
    - I listen on this TCP channel and that HTTP channel
    - I have a service called "MathService"
    - When client connects to MathService using a supported channel, give the client [ a | this ] Calculator instance
  - Clients need to specify the desired object
    - Connect to the "MathService" on server "LightningFast" using protocol HTTP on port 80

Writing a server

- Assumption: You have an assembly containing MarshalByRefObject types you wish to make available for use via remoting
- A server then becomes simply a host app
  - Loads the managed types
  - Configures remoting to expose the types
- Can write your own host
- Can use IIS as an already available HTTP host
Server design and implementation

- Design considerations
  - Decide which channels/ports to support
  - Select an activation model
  - Decide how clients get type metadata

- Implementation
  - Select/write appropriate host
  - Configure remoting system activation mode
  - Register channels

Channels

- A channel transports messages to and from a remote object
  - A server selects the channels upon which it listens for requests
  - A client selects the channels it wishes to use to communicate with the server

- Runtime provides built-in channel types
  - HTTP and TCP channels
  - You can write custom channels
ChannelServices.RegisterChannel

```csharp
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

ChannelServices.RegisterChannel (new HttpChannel());
ChannelServices.RegisterChannel (new TcpChannel(4242));
```

**Activation requests**

- Server also chooses how it wants to process activations
  requests for a type
- Two forms of server activation
  - WellKnownObjectMode.SingleCall
  - WellKnownObjectMode.Singleton
- One form of client activation
  - Client activated object (CAO)
SingleCall objects

- Server’s remoting layer creates one SingleCall object per method call
  - Each object services one and only one request
    - Created on-demand as method calls arrive
    - Lifetime limited to method call
- Useful in stateless applications
- Best choice for load-balanced applications

Singleton objects

- Server’s remoting layer creates one Singleton object
  - Sole object handles method calls from all clients
    - Lifetime equals server lifetime
- Useful in stateful applications
  - Can only store client-independent state
- Best choice where overhead of creating and maintaining objects is substantial
Client activated objects

- Server activated is one activation model
- Client activated is quite different
- Each client activation creates one object
  - Object's lifetime extends until the earliest event:
    - Client drops reference to object
    - Object's lease expires
  - Similar to COM coclass activation semantics
  - Can store per-client state, receive constructor args

Lease based lifetime

- Client activated object's lifetime controlled by a lease on the object
  - Leases have a lease time
  - Remoting infrastructure drops reference to object when lease expires
- Each method call renewes the lease
  - Use default renew on call time
- Clients can renew the lease using the proxy
- Sponsors can renew the lease
Well-known objects

- Server activated types are "well-known"
  - Server tells remoting
    - Here's a type
    - Here's how and when to instantiate the type
    - Here's the name (end point) a client will use to contact the type
  - Clients tell remoting
    - Connect to this server machine
    - Get the (known) type at this end point (name)

Well-known objects

- Server registers a well-known type using
  `RegisterWellKnownServiceType` specifying
  - The type being registered
  - The end point name known to clients
  - The activation model
RegisterWellKnownServiceType

```csharp
using System.Runtime.Remoting;
...
WellKnownServiceTypeEntry WKSTE =
    new WellKnownServiceTypeEntry(typeof(WiseOwl.Calculator),
    "EphemeralCalc",
    WellKnownObjectMode.SingleCall);
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);
```

Server activation example

```csharp
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

class RemotingHost {
    static void Main(string[] args) {
        RemotingConfiguration.ApplicationName = "WiseOwlMathService";
        WellKnownServiceTypeEntry WKSTE =
            new WellKnownServiceTypeEntry(typeof(WiseOwl.Calculator),
            "SharedCalc",
            WellKnownObjectMode.Singleton);

        RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);
        ChannelServices.RegisterChannel(new HttpChannel(9000));
        ChannelServices.RegisterChannel(new TcpChannel(4242));
        Console.ReadLine();
    }
}
```
Well-known object URLs

- Server-activated objects are published at a URL
  - The URL is "well-known" to the client
    ⇒ Such types are called well-known types
    ⇒ The URL is called the well-known object URL

```
ProtocolScheme://ComputerName:Port/PossibleApplicationName/ObjectUri
```

- When IIS is the server's host:
  - PossibleApplicationName becomes virtual dir name
  - ObjectUri should end in ".rem" or ".soap"
  - A TcpChannel requires the port number

Remoting config file

- All hard-coded remoting information in prior example can reside in external config file
  - Default filename is executable plus ".config"
  - E.g. RuntimeHost.exe is RuntimeHost.exe.config
- Host must tell remoting to use the config file
  - RemotingConfiguration.Configure (file)
- Server code simplifies…
Registering Client Activated Objs

- Server registers a client activated type using `RegisterActivatedServiceType`
  - Specifies the type being registered

```csharp
using System.Runtime.Remoting;
ActivatedServiceTypeEntry ASTE = new ActivatedServiceTypeEntry(typeof(WiseOwl.Calculator));
RemotingConfiguration.RegisterActivatedServiceType(ASTE);
```
Client activation example

```csharp
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

class RemotingHost {
    static void Main(string[] args) {
        RemotingConfiguration.ApplicationName = "WiseOwlMathService";
        ActivatedServiceTypeEntry ASTE =
            new ActivatedServiceTypeEntry(typeof(WiseOwl.Calculator));
        RemotingConfiguration.RegisterActivatedServiceType(ASTE);
        ChannelServices.RegisterChannel(new HttpChannel(9000));
        ChannelServices.RegisterChannel(new TcpChannel(4242));
        Console.ReadLine();
    }
}
```

Client activation URLs

- Client activated objects do not need a unique URL for each object

```
ProtocolScheme://ComputerName:Port/PossibleApplicationName
```

- PossibleApplicationName becomes virtual directory name when hosted in IIS
- A TcpChannel requires the port number
Remoting clients

- A client wants to use a remote object
  - Well-known objects exist at a URL
    - Client obtains proxy using `Activator.GetObject`
    - Client can also obtain proxy using `new`
  - Client activated object factory exists at a URL
    - Client requests factory to instantiate object and return a proxy to it using `Activator.CreateInstance`
    - Client can also make same request using `new`

---

Activator.GetObject

- `GetObject` returns a proxy for the well-known type served at the specified location

```vbnet
Dim o as Object = Activator.GetObject (GetType (WiseOwl.Calculator),
"http://localhost:9000/WiseOwlMathService/SharedCalc")

WiseOwl.Calculator c = CType (o, WiseOwl.Calculator)

c.Add (42);
```

- No network traffic until first method call!
  - Proxy built on client from metadata
  - Server activates object on first method call
**RemotingServices.Connect**

- GetObject is a thin wrapper over
    - RemotingServices.Connect

```csharp
Using System.Runtime.Remoting;
static object o RemotingServices.Connect (Type classToProxy, string url);
```

  - RemotingServices.Unmarshal

**RemotingServices.Unmarshal**

- RemotingServices.Unmarshal does the real work
  - Checks that type is MarshalByRef or Interface
  - Find or creates identity based on URI
  - Creates envoy and channel sink chains
  - Gets or creates transparent and real proxy
- End results is client gets a proxy
How does it work?

- Client receives a proxy object when activating a remote object
  - Client "thinks" proxy is actual object
- Proxy is instance of `TransparentProxy` class
  - Mimics inheritance hierarchy of remote object
- A call on the proxy:
  - passes control to actual object when in same domain
  - creates an `IMessage` to object in different domain
  - Passes message to a `RealProxy` instance

RealProxy class

- `RealProxy` forwards msg to remote object
  - Handles all method calls to remote object
- `RealProxy` class can be extended, customized
  - For example, load balancing method calls, client side validation/processing of certain calls
- `TransparentProxy` cannot be replaced
  - Internal remoting implementation class
IsTransparentProxy

- You can call IsTransparentProxy on any object reference to see if it is a proxy
  - Normally, you don't care

```csharp
using System.Runtime.Remoting;

bool RemotingServices.IsTransparentProxy (Object o);
```

ObjRef

- Runtime creates an ObjRef when you register an object with the remoting services
- An ObjRef contains all information required to locate and access a remote object
  - the strong name of the class
  - the class's hierarchy (its parents)
  - the names of all interfaces the class implements
  - the object URI
  - details of all available registered channels
How to remote existing objects

- Server/client activation expose "new" objects
  - What about existing objects?
- RemotingService.Marshal
  - Accepts a MarshalByRefObject
  - Registers it with remoting
  - Returns an ObjRef
- RemotingService.Unmarshal
  - Accepts an ObjRef
  - Returns a proxy

---

RemotingServices.Marshal

```csharp
// On the server, we find our hero about to go public...  
WiseOwl.Calculator calc = new WiseOwl.Calculator();
  // Let the remoting layer (and the world) know about it
  ObjRef or = RemotingServices.Marshal(calc, "ExplicitCalculator");
  // Clients can now connect to the Calculator as a Well-known object
  // prot://machine:port/WiseOwlMathServices/ExplicitCalculator
  // Alternatively, we can serialize and send the ObjRef to a client
  // System.IO.FileStream fs =
  //     new System.IO.FileStream("C:\ObjRef.xml", System.IO.FileMode.Create);
  sf.Serialize(fs, or);
  fs.Close();
```
RemotingServices.Unmarshal

```vbnet
' There’s an ObjRef serialized to a file using the SOAP formatter
' Open a stream on the file
Dim fs As System.IO.FileStream
   fs = New System.IO.FileStream("C:\ObjRef.xml", System.IO.FileMode.Open)
' Deserialize the object in the file
Dim o As Object = sf.Deserialize(fs)
fs.Close() ' Done with the file
' Object is really an WiseOwl.Calculator
Dim c As WiseOwl.Calculator = CType(o, WiseOwl.Calculator)
' Use the Calculator
   c.Add(21)
```

.NET Remoting

- .NET supports two basic forms of remoting
  - Web Services
    - An entry point into application specified by an URL
    - `<protocol>:\<machine>:\<port>\<URI>`
    - E.g. `http://localhost:4242/SomeServiceName`
    - Uses SOAP data types
  - CLR Object Remoting
    - Builds on Web Services
    - Uses native CLR data types
### Web Services

- Expose WebService endpoints from any process over any transport using any payload encoding
- Process types include console apps, graphical applications, NT Services, IIS
- Built-in Transports include HTTP and TCP
  - Extensible via pluggable channels
- Built-in encodings include SOAP and Binary
  - Extensible via pluggable serialization formatters
- Supports SOAP 1.1 (~XSD) type system

### CLR Object Remoting

- Full CLR type system fidelity
  - class hierarchies, constructors, delegates, interfaces, methods, overloaded methods, properties, fields
- Supports additional features over Web Services
  - Marshal by value (make a copy)
  - Marshal by reference (pass an ObjRef)
- Maintains distributed object identity
- Provides object activation semantics
- Allows control over object lifetime using leases
- Permits out of band info using CallContext
Remoting via Web Services

- Any client \(\leftrightarrow\) .NET using HTTP/SOAP
  - .NET object exposed as a Web Service by hosting in IIS
  - Web Services use well-defined interfaces called contracts
    - Contracts are described using a Web Services Description Language (WSDL) file
  - Any client that can consume a WSDL file can make SOAP calls to the remote object as per the contract
  - Firewall friendly, client OS agnostic
  - XML Schema data (XSD) types used in contract

Remoting via HTTP

- .NET client \(\leftrightarrow\) .NET using HTTP/SOAP
  - HTTP channel uses the SOAP formatter
    - Can override this default behavior
    - Less efficient than binary formatter
  - Firewall friendly
  - Uses CLR type system, not XSD
Remoting via TCP

- .NET client ↔ .NET using TCP channel
  - TCP channel uses the binary formatter
    - Can override this default behavior
    - More efficient than SOAP formatter
  - Raw socket connection used for transport
    - Mostly useful behind a firewall, not through one

Remoting via DCOM

- .NET client ↔ COM ↔ .NET
  - .NET client to COM uses RCW and COM interop
  - DCOM used when COM object is remote
  - COM to .NET uses CCW and COM interop
    - DCOM used when .NET server is remote
- Useful in heterogeneous environments
Using IIS as the host server

- IIS provides activation
  - Server does not have to be manually started
  - IIS uses web.config remoting config file

```xml
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall" type="WiseOwl.Calculator,MathObjects" objectUri="EphemeralCalc.rem"/>
        <wellknown mode="Singleton" type="WiseOwl.Calculator,MathObjects" objectUri="SharedCalc.soap"/>
        <activated type="WiseOwl.Calculator,MathObjects"/>
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

---

```csharp
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

namespace RemotingSamples {
    public class Sample {
        public static int Main(string[] args) {
            // Create and register a new TCP channel
            TcpChannel chan = new TcpChannel(8085);
            ChannelServices.RegisterChannel(chan);

            // The RegisterWellKnownServiceType method allows registering an object for future activation
            RemotingConfiguration.RegisterWellKnownServiceType
                (Type.GetType("RemotingSamples.HelloServer, object"),
                 "SayHello", WellKnownObjectMode.Singleton);

            System.Console.WriteLine("Hit  to exit...");
            return 0;
        }
    }
}
```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

namespace RemotingSamples {
    public class HelloServer : MarshalByRefObject {
        public HelloServer() {
            Console.WriteLine("HelloServer activated");
        }

        public String HelloMethod(String name) {
            Console.WriteLine("Hello.HelloMethod : {0}", name);
            return "Hi there " + name;
        }
    }
}

namespace RemotingSamples {
    public class Client {
        public static int Main(string [] args) {
            TcpChannel chan = new TcpChannel();
            ChannelServices.RegisterChannel(chan);
            // Il client localizza l'oggetto remoto passando tipo e URL
            if (obj == null)
                System.Console.WriteLine("Could not locate server");
            else
                Console.WriteLine(obj.HelloMethod("Carlo"));
            return 0;
        }
    }
}
References

- [http://www.dotnetremoting.cc](http://www.dotnetremoting.cc)