Cst556 Distributed Applications for .NET with Mono

4. Remoting with .NET
4.a .NET Remoting: Outcomes and References

4.a.1 Motivation and Outcomes

- What is .NET Remoting and how can I develop distributed applications using Remoting Object Servers and their Clients in C#? Remoting provides for object-level communication between client and server.

- Section Outcomes
  - To understand what it means to say that remoting allows distributed applications to be developed at the object level rather than via TCP data protocols.
  - To understand the differences between SingleCall, Singleton and Client-Activated remote object servers and the distributed applications best suited for each. To be able to develop C# .NET applications that utilize each.
  - To understand the leasing mechanism for controlling lifetimes of Remote Objects.
  - To understand limitations on and handling of parameter and return types for remoting applications.
4.a.2 References for .NET Remoting

- C# Web Services (course textbook) Chapter 6 on .NET Remoting Architecture
- C# for Java Developers by Jones and Freeman, Microsoft Press, Chapter 15
4.a.3 Index to Example Programs for .NET Remoting

- Server-activated: Section 4.d.4 Example Server-Activated Pencil Server
- Single-call: Section 4.d.4 Example Server-Activated Pencil Server
- Client-activated: Section 4.d.6 Example Client-Activated PencilServer
- User-defined parameter types: Section 4.f.2 Remoting Example with User-Defined Parameter Types
- Using leases: Section 4.f.4 Leases in GroupMgr Remoting Example
- Remote Server that Registers and calls-back clients for notification: Section 4.g.2 Solution Approach
4.b Introduction to Distributed Objects

4.b.1 What are Distributed Objects?

- **Distributed objects** as realized through .NET Remoting allow components of distributed applications to communicate at the object level.
  - Components, which may reside in different processes and/or on different machines are viewed as separate executing parts of an application.
  - Example components are **browser** and **http server**, **email client** and **imap server**, as well as their subparts.
  - Components send each other messages in an object-oriented way, consisting of a method to be invoked together with arguments for the method. The result of a message may be a return value.
  - Arguments, call **synchronization** and **return values** are all **handled by platform**; not by the application developer.

- Without distributed objects, sending a message to a remote machine:
  - Current **protocols** (http, ftp, telnet) use **stream sockets** with **data protocols**, or RPC
  - Current protocols are data centric, difficult to maintain, enhance, and use.
4.b.2 Vision of Distributed Object Computing

- Applications are **federations** of cooperating distributed objects.
  - Configuration may be **dynamic** - objects can discover each other and communicate to perform some service for each other.
  - Distributed Objects can **survive network outages** and/or follow **mobile paradigms** where connectivity is constantly changing.
  - Distributed Objects may be **service providers** that register themselves with multiple registries. Potential clients may contact the registry and discover information about compatible (and maybe competing) services.

- View the Internet as an **Object Bus**

- Distributed Object technologies remove the underlying **middleware** issues from applications and their developers.
  - How to implement message passing and argument synchronization?
  - What if objects are not “in-process”?
  - Distinction between objects within LAN versus WAN
4.b.3 .NET Remoting and ASP.NET Web Service Competitors

• **Distributed COM** and its evolutions by Microsoft
  - Multi-language, single platform
  - Constrained functionally by not including complete object model

• **Common Object Request Broker** by Object Management Group
  - Multi-language, multi-platform,
  - Sacrificed ease of use and functionality.
  - Lagging Pass-By-Value implementations (serialization) hindered usage.

• **Remote Method Invocation** in SUN’s Java
  - Single language, multiple platform
  - Easy to use and conceptually clean, although with automatic dynamic class loading security must be carefully addressed for each application.
  - Incurs burden of net-wide reference counting and garbage collection.
  - Lots of Intranet application including J2EE and other web server-specific support applications.
4.b.4 ASP.NET Web Services versus .NET Remoting

- .NET Framework includes two mechanisms for distributed application development:
  - ASP.NET Web Services
  - .NET Remoting

- Remoting can use different transport mechanisms
  - A native binary TCP based protocol
  - SOAP and HTTP
    - Object references and a proxy object are utilized by the client
    - Remoting is a fast (binary formatted remoting is twice as fast as web services) solution for .NET clients and servers (no interoperability).

- ASP.NET Web Services expose web methods using SOAP and HTTP through an IIS web server.
  - Developed based on open standards (WSDL, UDDI, SOAP)
  - Intended to allow across development platforms (for example, Java client accessing an ASP.NET Web Service).
4.c  Background of .NET Applications

4.c.1  View of ASP.NET Remoting

- Multiple .NET applications can run in the same operating system process
  - Application isolation (for transactions, security, for example) is achieved by defining Application Domains within a process for each application.
  - An Application Domain is the .NET common language runtime’s version of a “light weight” process for executing an application.
  - See the System.AppDomain class

- ASP.NET hosts all ASP.NET applications on a given machine within a single process.

- Each Application Domain consists of Assemblies
  - Assemblies can be loaded under program control.
  - The Application Domain can be unloaded under program control.
  - An Assembly can be shared by all Application Domains (code not data), which is called a Domain-Neutral Assembly

- Objects in one Application Domain can’t interact with objects in another except through Remoting.
4.c.2  .NET Base Class Library

- The Base Class Library is spread over two assemblies
  - MSCorLib.dll
  - System.dll
- MSCorLib.dll is always loaded as a domain neutral assembly
  - To keep it as small as possible, there are no references outside the core library.
- .NET provides that classes can be configured to require services such as:
  - synchronization,
  - security,
  - just-in-time activation, and
  - transactions
- These available services make up a runtime environment called the Context
  - Classes (objects) within an Application Domain are associated with a Context that defines the resources and constraints on its execution.
4.c.3 Communication outside the Application Domain

- Any communication outside the application domain requires either
  - **Marshal by Value Objects**, which are serializable objects
  - **Marshal by Reference Objects**, which are called **remote objects**
- Any objects that are neither **remote** nor **serializable** are limited to the application domain in which they are created.
- **Serializable objects** -implement **ISerializable** or are marked `[Serializable]`
  - **Marshalled** as they leave an Application by **serializing** the object - (creating a copy) and then transporting them through a stream.
  - **Unmarshalled** on entering a new Application Domain by **deserializing**.
- **Remote objects**
  - Extend **System.MarshalByRefObject**
  - When a **remote object reference** is imported into an Application Domain, a **proxy object** is created in the receiving Domain.
  - The **proxy** has a method for each method in the original object.
  - **Proxy** object methods handle calling methods and receiving results utilizing a **Tcp** or **Http** channel to another process/application domain.
4.c.4 Calling a Remote Object’s Methods

• Client method calls resolve to the proxy object.
  - The proxy object directs the call through the Remoting layer, which marshals the parameters and communicates all call information to the receiving application domain through a defined channel.
  - On the object server side, the Remoting layer unmarshals the call and arguments, calls the method and then marshals and returns the results.
4.d Implementing with .NET

4.d.1 Types of .NET Remote Objects

- **Server-Activated** remote objects - also called **Well Known Objects**
  - The client must have a **URL naming the object** (well known to client)
  - A **server may create and register the remote object**, associating it with either a TCP or HTTP channel for communication.

- Two types of **Server-Activated** objects
  - **Singleton object** - one object exists and is used to handle requests from all clients. Used when one client’s information may be displayed in another, such as a chat server. **Hard to scale** across multiple servers.
  - **Single Call object** - a **new object** is created to handle each client call. Used for **stateless** remote objects where an isolated service is to be performed for each client call. For example calculating a function, such as the energy consumed by an air conditioner.

- **Client-Activated objects** - A single object is created and used to service multiple calls from a single client.
  - Used to retain client session information through multiple calls, such as shopping cart applications.
4.d.2  Developing an Application with .NET Remoting

- **Server**
  - Define the server-side classes and methods that will be exposed to the internet (published).
  - Provide implementations of the classes and develop interfaces or proxies that isolate the server implementation classes from clients.

- **Registering the server-type**
  - Based on the characteristics of the application, decide on the appropriate deployment paradigm - communication platform (tcp or http) and remote object type (Server-Activated or Client-Activated)
  - Create a separate console application (or Windows Service) to register the server channels and type(s)

- Consider **server object lifetime** - if leases are involved and are to be manually maintained, create a client application to sponsor the lease

- **Client**
  - Define the client applications that will use the service. Provide their user-interface as well as connection to server code.
4.d.3 Publishing a Server-Activated Singleton

- Either (or both) HTTP and TCP can be used to communicate between the client and the server.
  - **Http channels** use SOAP and have slower performance. Use Http when clients and server are located on different networks. Http channels have specific support for using proxies and SOAP over HTTP makes traversing Internet proxies easier.
  - **Tcp channels** use binary encoding which offers a performance benefit but is less likely to be successfully transmitted across all Internet proxies.

- A single Application Domain may define more than one channel
  - For Example, publishing a server-activated singleton Group Server:

```csharp
TcpChannel tcpChan = new TcpChannel(8888);
ChannelServices.RegisterChannel(tcpChan);
HttpChannel httpChan = new HttpChannel(8889);
ChannelServices.RegisterChannel(httpChan);
RemotingConfiguration.RegisterWellKnownServiceType(typeof(GrpMgr),
    "GrpSrv", WellKnownObjectMode.Singleton)
...

- A client may obtain a reference to the group manager through TCP with:

  GrpMgr gm=(GrpMgr)Activator.GetObject(typeof(GrpMgr),"tcp://localhost:8888/GrpSrv")
```
4.d.4  Example Server-Activated Pencil Server

• Suppose we have a pencil server. Clients share pencils by contacting the server to borrow or return some number of pencils.

• Here’s a jar file containing the entire example: dotNetRemotingPencil.jar

• Create the server implementation class as a library.
  - See: PencilServerImpl.cs in the jar file.
  - msc /t:library /out:PencilServer.dll PencilServerImpl.cs

• Create an application to register and initiate a single server object
  - See: RegisterPencilServer.cs in the jar file.
  - msc /reference:PencilServer.dll RegisterPencilServer.cs

• The example comes with an Ant build.xml file and readme.txt
  - See the readme.txt file for instructions on building and running the example. Note that this example includes server activated singleton and single call as well as client activated (session).
4.d.5 SingleCall and Client-Activated Remote Objects

- **Server-Activated SingleCall** - Activates a new object for each method call. Client calls a method which initiates server to instantiate a new object.
  - The object is activated for a single call. Upon completion, the remote object is released. Registration indicates the type singleton or single-call

```
HttpChannel httpChan = new HttpChannel(8889);
ChannelServices.RegisterChannel(httpChan);
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(GroupMgr),
    "GroupServer",
    WellKnownObjectMode.SingleCall)
```

- On the client side, obtaining an object may be done the same as:

```
ChannelServices.RegisterChannel(new HttpClientChannel());
GroupMgr gm = (GroupMgr)Activator.GetObject(
    typeof(GroupMgr),
    "http://localhost:8090/GroupServer");
```

- **Client-Activated** - Server registers the type; activation is initiated in the client. A single object instance serves all of a client’s calls.

```
ChannelServices.RegisterChannel(new TcpServerChannel(8889));
RemotingConfiguration.RegisterActivatedServiceType(typeof(GroupMgr));
```

- One way a client can create the object is using `Activator.CreateInstance`:

```
GroupMgr gm = (GroupMgr)Activator.CreateInstance(typeof(GroupMgr), null,
    new object[] { new UrlAttribute("tcp://localhost:8889") });
```
4.d.6 Example Client-Activated PencilServer

- Download the Ant project (same as the prior example), see: `dotNetRemotingPencil.jar`
  - Follow the directions in the `readme.txt` file to run client-activated.

- Differences and Observations:
  - Observe `session` functionality by starting the server. Next create two or more clients in separate command prompt windows. Note that each client has its own remote server object (and its own supply of pencils.)
  - The life-time of a client-activated remote object matches the life-time of the client’s remote reference to the server.

- Directly building the application requires the following compiles:
  - `msc /t:library /out:PencilServer.dll PencilServerImpl.cs`
  - `msc /reference:PencilServer.dll RegisterPencilServer.cs`
  - `msc /reference:PencilServer.dll PencilClient.cs`
  - Attempting to deploy `PencilClient.exe` without `PencilServer.dll` in the folder will result in an attempt to download the library from the server.
4.d.7 Configuration by XML

- Instead of using method calls and their parameters to specify `RemotingConfiguration`, it can be done by reading XML from a file.
  - Reading from an XML file allows for modifying aspects of the configuration without having to recompile and rebuild the application.
  - Suppose the Server-Activated configuration is in file: `GroupConfig.xml`

```xml
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode=”Singleton” type=”GroupMgr, GroupMgr” objectUri=”GroupServer”/>
      </service>
      <channels>
        <channel ref=”http” port=”8889”/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

- The server registration program could specify this configuration with:

```csharp
RemotingConfiguration.Configure(“GroupConfig.xml”);
```
4.e Lifetimes for Server-Activated Remote Servers

4.e.1 Lifetime of Remote Objects

- **Client-Activated** and **Server-Activated** objects aren’t active forever, unless explicitly specified this way by the developer.
  - Default idle period is 5 minutes. Released after 5 minutes of inactivity.
  - A new object is instantiated when a client subsequently calls a method on the object.
  - Leases are used to manage lifetime, which avoids the overhead of network wide reference counting of remote object references.

- **Server-Activated SingleCall** remote objects have a lifetime that is limited to the duration of a single method call. The object is initialized to service a single method call and released after the method completes.

- How is the lifetime of a Server-Activated Singleton and a Client-Activated remote object determined?
  - When a new instance is created, the **Remoting System** calls the **MarshalByRefObject** method **InitializeLifetimeService**, which returns an object implementing **ILease**
4.e.2 Manually Controlling Leases

- A Remote object can override the `InitializeLifetimeService` method to control aspects of the lease.
  
  - The remote server’s base class (MarshalByReference Object) contains the method `InitializeLifetimeService`. The remote server’s implementation can override this method to provide server-specific functionality.
  
  - `InitialLeaseTime` determines how long a server-activated object will exist when its initially registered.
  
  - `RenewOnCallTime` determines how long the object will exist after a client makes a method call.

```csharp
public override object InitializeLifetimeService() {
    ILease myROLease = (ILease)base.InitializeLifetimeService();
    if (myROLease.CurrentState == LeaseState.Initial) {
        myROLease.InitialLeaseTime = TimeSpan.FromHours(3); //initially 3 hours of idle time till release
        myROLease.RenewOnCallTime = TimeSpan.FromMinutes(10); //after a method call extend 10 mins
    }
    return myROLease;
}
```

- Returning `null` or setting `InitialLeaseTime` to zero, configures the object with a never expiring lease.

  - For example: `myROLease.InitialLeaseTime = TimeSpan.Zero`
4.e.3 Sponsoring a Lease

- A client of a remote object can sponsor the objects lease.
- Sponsors implement **ISponsor** and register interest in lease expiration
  - The sponsor is a client that manages lease renewal on behalf of the remote object.

```csharp
ILease myROLEase = (ILease)RemotingServices.GetLifetimeService(myRO);
myROLEase.Register(this); // this code may appear in a method that publishes the service.
```

- the **ISponsor.Renewal** method is called before the lease expires. It returns a **TimeSpan** indicating how long the lease should be extended.

- The sponsor must be a serializable or a MarshalByRefObject class
  - When registering the sponsor with remoting, the Sponsor is used as an argument to a method in the Remoting Server.
  - When the Sponsor is serializable, it is serialized and copied to the server and subsequently executed on the server.
  - When the sponsor is a remote object, then it acts as a separate client.
  - Example of a server registered with a lease: **userParms.jar**
4.f  .NET Remoting Method Parameters

4.f.1  Arguments and Returns for Remote Objects

- What are the valid parameter and return types for Remote Object Calls?
  - **Primitive values**, such as int and boolean.
  - **.NET defined Serializable objects** are passed by copy/value, such as `String` or `Array` classes. They are serialized upon **marshalling** and deserialized when **unmarshalled**.
  - **Remote objects**. Classes that extend `System.MarshalByRefObject`
  - **User-Defined serializable** or **remote classes**. Must provide class information to the client. For serializable, the entire class must be available to the client. For remotes, the client needs access to the proxy.
4.f.2 Remoting Example with User-Defined Parameter Types

- Consider the problem of representing and serving information on groups of people, such as Cst556, Faculty, and Students. See: userParms.jar

- The diagram above shows assemblies in this application and their references

- Example with multiple clients accessing and manipulating information about groups of persons. Changes made by one client can be seen by all.
  - Characteristic of a class of problems in distributed applications, such as email, listserves, and chatrooms.
4.f.3 Comments on the GroupMgr Example

- Implementation of this group application in .NET Remoting includes:
  - Singleton object published in the remoting framework (GroupMgr).
  - A number of remote Groups live in the application, but are not published.
  - Serialized objects are copied and passed to clients (Person)

- Constraint in deploying with .NET Remoting Clients. GroupMgrClient does not bind a Proxy for the remotes. Instead it references GroupMgr.dll and Person.dll libraries.
  - A distributed application may include a user-defined serializable class as a parameter or return from a Remote method, such as Person. By design, the client and the server may need access to the full class definition. They may be designed to rely on methods of the shared class. For example, Person’s Age property get and set methods.
  - The soapsuds tool does not treat serializable user-defined classes as one would hope. Soapsuds generates a stub-class definition for user-defined serializable parameter (or return) types (Person). The stub promotes all properties to public visibility, and does not include methods; Person’s Age property get and set methods, for example.
4.f.4 Leases in GroupMgr Remoting Example

- The example consists of three console applications
  - **RegisterGroupMgr** - registers the GroupMgr as a well know type. **SponsorGroupMgr** object is created as the renewal sponsor for GroupMgr
  - **GroupMgrClient** - provides a client user interface for managing groups

- Upon instantiation, **GroupMgr** sets its initial (and on method call) lease times to 15 seconds. It also creates the SponsorGroupMgr object to manage its lease.
  - After 15 seconds of idle time, the object is released except that the sponsor has been registered to renew the lease.
  - Whenever the lease expires, **Renewal** is called to provide a **TimeSpan**
  - Observe what happens to the state of defined groups when you run the client without (comment out) the sponsor.
  - See: **userParms.jar**
4.f.5 Notes and Issues on GroupMgr Example

- Consider the following points related to the example
  
  - How is name management handled for registered remote object types? Can a registered type be unregistered? Can a client on another machine obtain a listing of all registered types and their corresponding names?

- Controlling where methods execute in distributed applications.
  
  - Serializable objects versus marshal-by-reference objects as parameters and return values.

  - Note the windows in which the two following messages appear (client or server): new group for myGroup created, and accessed person miniMe’s age property. Why do they appear where they do?

- Using a proxy versus distributing the code or library version of required classes:

  - Note that the GroupMgr example builds clients using the library containing code for the server. Assuming we don’t want implementation code on the client, how could we better structure the solution?
4.f.6 Threads and Thread-Safe Server Objects

- Server-activated singleton objects should be thread-safe.
  - What happens if multiple requests for method(s) in a singleton server object arrive at the same time?
  - The remoting subsystem creates distinct threads to service each arriving request. What is the implication on any underlying shared data?
  - As a practical guideline, all marshal-by-reference classes should be thread-safe.

- In the Group Manager example:
  - The singleton **GroupMgr** object was not coded as thread safe. Observe that multiple threads (one for each client) can execute **Group** and **GroupMgr** methods concurrently. There is no enforced synchronization of these threads. Good exercise to make the classes thread safe.
  - For **SingleCall server-activated** remoting, is there any way to allocate objects from an existing pool of remote objects as requests arrive? Object pooling is a common approach to scaling the types of applications that SingleCall objects are intended to address.
4.g Call-Backs from Server to Client

4.g.1 The Problem

- Consider distributed applications where the server has a persistent state that may be accessed and modified by multiple clients.
  - Characteristics of group-ware, email or threaded message servers.
  - Multiple clients may be simultaneously displaying some aspect of the server’s current state.
  - If one client requests a server operation that modifies the state, then all other clients may display out-of-date information.

- For example, consider the PencilServer supply desk application.
  - Suppose clients are graphical user interfaces in which each client displays the current total of pencils in the supply desk.
  - If one client adds pencils to the supply, then the total displayed on all other clients will be out-of-date.

- **Solution**: The server acts as a client whenever the state of the supply desk changes. It calls all active clients notifying them that a change has occurred in the supply desk (a modify event has occurred). The server calls the client (commonly called a callback) so it can update its display.
4.g.2 Solution Approach

- In Java with RMI:
  - The client becomes a remote object by implementing the `Remote` interface. It also implements an interface known to the server, such as `PencilChangeListener`. The PencilChangeListener interface includes a method, such as `notifyPencilSupplyChanged`.
  - The client either extends one of the remote classes (`Activatable`, or `UnicastRemoteObject`) or it performs the static `ExportObject` method to let the RMI system know its a remote object.

- In C# .NET Remoting:
  - The server defines a public `delegate`, such as `NotifyPencilModified`.
  - The client creates a delegate object, associating the delegate with a method defined in the client. See example project: `callBackPencil.jar`

- In both the C# and Java solutions, the server:
  - Implements methods: `addListener` and `removeListener`; also the private method `callListener`. `CallListener` creates a `thread` for each client call.
  - In the client constructor, it calls the server to add itself as a listener; on exit the client calls the server to remove itself from the set of listeners.