.Net Remoting

Jim Fawcett

CSE681 – Software Modeling & Analysis

Fall 2004

References

- Programming Microsoft .Net, Jeff Prosise, Microsoft Press, 2002, Chap 15.
- http://samples.gotdotnet.com/quickstart/howto/

Distributed Computing under .Net

- In .Net, there are three levels of access to distributed computing machinery:
  - Low Level:
    - System.Net.Sockets
  - Intermediate Level
    - System.Runtime.InteropSerives
      - Access COM objects and Win32 API
    - System.Runtime.Remoting
      - Access channels and CLR activation
      - Channels based on TCP or HTTP over TCP
  - High Level
    - System.Web.Services
    - System.Web.UI

Distributed Computing under .Net

- System.Net.Sockets
  - Provides low-level access to socket objects
  - You create listeners and send and receive just like we did in the socket demonstration code.

- System.Runtime.Remoting
  - Provides access at a medium level of abstraction.
  - You create channels and proxies and do RPCs on remote objects
  - Data marshaling is much richer than under COM. You can send anything the CLR understands as long as it has a [serializable] attribute or derives from MarshalByRefObject.
    - Basically you just add those .Net identifiers and the CLR takes care of everything else.

Distributed Computing under .Net

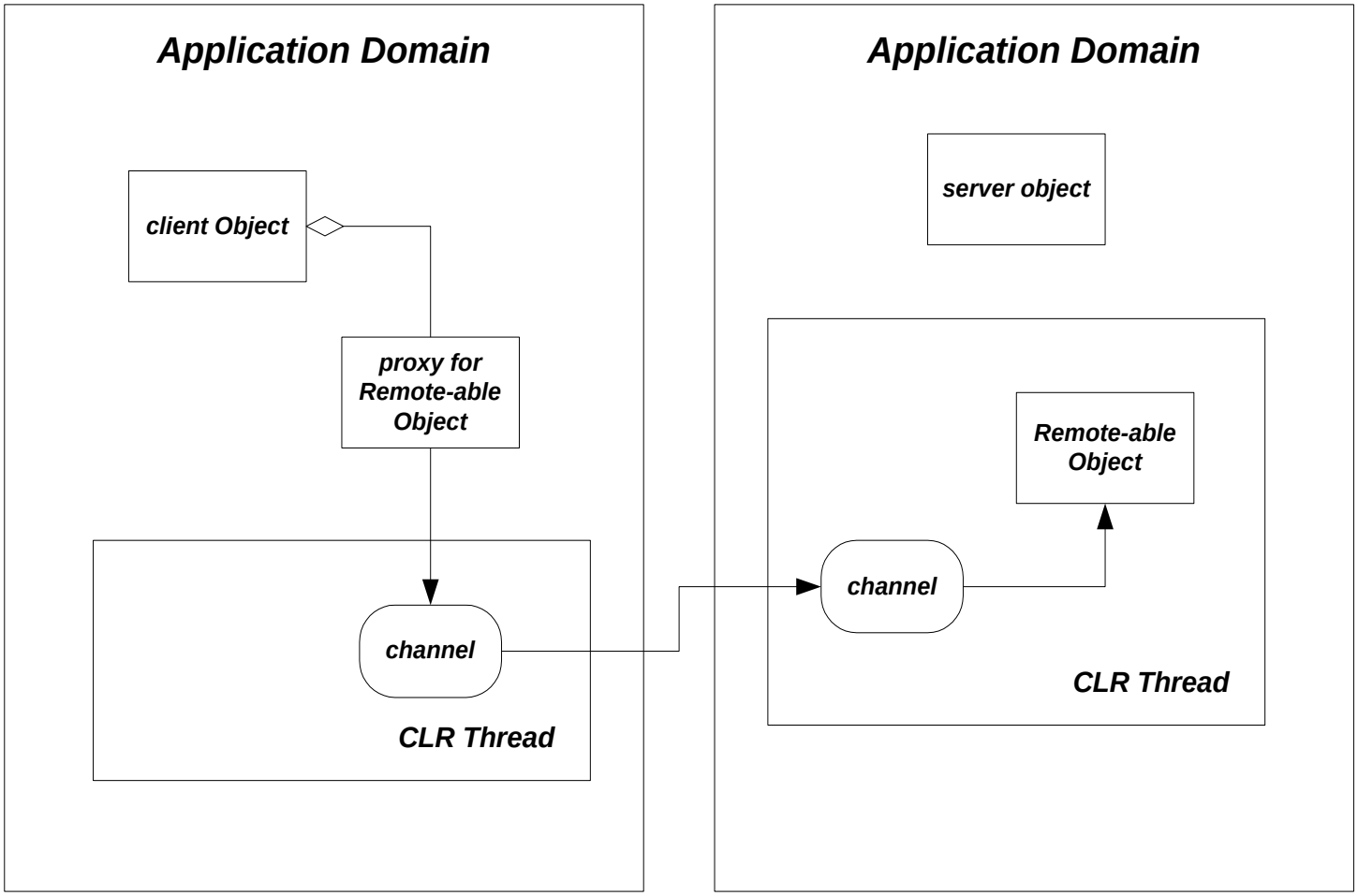
- System.Web.Services
  - Servers are hosted under IIS
  - Use HTTP-GET and HTTP-POST or higher level SOAP

- Simple Object Access Protocol (SOAP)
  - Wraps XML message in SOAP envelope (XML tags)
  - SOAP messages are interpreted by IIS and ASP
  - Typically use standard and/or custom COM components in ASP pages.
  - Active Server Pages (ASP) are XHTML pages with embedded server-side and client-side scripts that may access COM and C# objects for a significant part of their processing.

.Net Remoting

- Remoting supports a client's invocation of an object on a remote machine.
  - The server acts as a host for the remote object, loading it into memory and servicing client requests on a ***worker thread*** spawned by the server process's ***main thread***.
    - All of this is transparent to the designer.
  - The client makes calls as if the object were instantiated on the local machine.

Remoting Architecture



Server Supporting Remote-able Object

- Class of Remote-able object is derived from MarshalByRefObject.
  - Otherwise the object is oblivious of the remoting infrastructure.
- Server:
  - creates a TcpServerChannel
  - Registers Channel with ChannelServices
  - Registers Class of remote-able object with RemotingConfiguration
  - Then main server thread waits for client to shut it down.
- This can be done either programmatically or with a config file. We will demonstrate the former.

Client of Remote-able Object

- Client:
  - Creates TcpClientChannel
  - Registers channel with ChannelServices
  - Creates a proxy for remote object by calling Activator.GetObject
  - Uses proxy to invoke remote object:

```
string retVal = clnt.proxy.say(msg);
```

Remoting Server Code

This server's only role is to setup the channel, register the object, and wait while it is used by the client.

```
static void Main(string[] args)
{
   TcpServerChannel chan = new TcpServerChannel(8085);
   ChannelServices.RegisterChannel(chan);
   RemotingConfiguration.RegisterWellKnownServiceType(
     typeof(Hello),  // type of the remote object
     "HelloObj",
     WellKnownObjectMode.Singleton
   );
   System.Console.WriteLine("\n  Hit <enter> to exit...");
   System.Console.ReadLine();
}
```

Remotable Object Code

```
public class Hello : MarshalByRefObject
 {
   private int count = 0;

   public Hello()
   {
     Console.WriteLine("  construction of Hello Object");
   }
   public string say(string s)
   {
     ++count;
     Console.WriteLine("  " + s);
     string rtnMsg = "remote object received message #";
     rtnMsg += count.ToString();
     return (rtnMsg);
   }
 }
```

Just like any other class except that it derives from MarshalByRefObject

Client Code

Client sets up channel and constructs proxy. Then it uses object, as shown on next slide.

```
class client
 {
   private Hello proxy;

   //----< set up TCP channel >-----------------------------

   void SetUpChannel()
   {
     TcpClientChannel chan = new TcpClientChannel();
     ChannelServices.RegisterChannel(chan);
   }
   //----< activate remote object and return proxy >----------

   void ActivateRemoteObject()
   {
     proxy = (Hello)Activator.GetObject(
       typeof(Hello),
       "tcp://localhost:8085/HelloObj"
     );
     if(proxy == null)
       Console.WriteLine("can't activate object");
   }
```

```
static void Main(string[] args)
{
  client clnt = new client();
  clnt.SetUpChannel();
  clnt.ActivateRemoteObject();
  if (clnt.proxy == null)
  {
    System.Console.WriteLine(" -- Could not locate server -- "); return;
  }
  Console.Write("\n  To call remote object enter string");
  Console.WriteLine("\n  Just hit enter to quit:");
  try
  {
    while(true)
    {
      string test = "...";
      Console.Write("\n  > ");
      test = Console.ReadLine();
      if(test == "")
        break;

      // invoke remote object
      string retVal = clnt.proxy.say(test);

      // display string returned from remote object
      Console.Write("  ");
      Console.Write(retVal);
    }
  }
  catch(System.Exception e)
  {
    Console.WriteLine(e.Message);
  }
}
```

Here, client accesses remote object via its proxy.

HelloLib - Microsoft Visual C# .NET [design] - hello.cs

File Edit View Project Build Debug Tools Window Help

Object Browser | Activator Members | server.cs | hello.cs | client.cs

CSE791.Hello

```
// Application: CSE691\891 - Internet Programming, Summer 2002      /
// Author:      Jim Fawcett, CST 2-187, Syracuse University         /
//              (315) 443-3948, jfawcett@twcny.rr.com               /
////////////////////////////////////////////////////////////////////////
///
///  Operation:
///    Instances of the Hello class:
///      - accept and display a string sent by the caller
///      - return a numbered message to the caller
///
using System;
using System.Threading;

namespace CSE791
{
  public class Hello : MarshalByRefObject
  {
    private int count = 0;

    public Hello()
    {
      Console.WriteLine("  construction of Hello Object");
      Console.WriteLine(
        "  Hello Object: AppDomain {0} Thread {1} Context {2}",
        AppDomain.CurrentDomain.FriendlyName,
        Thread.CurrentThread.GetHashCode(),
        Thread.CurrentContext.ContextID
      );
    }
    public string say(string s)
    {
      ++count;
      Console.WriteLine("  " + s);
      return ("remote object received message #" + count.ToString());
    }
  }
}
```

C:\SU\CSE891\Su2002\Lecture7\RemotingDemo\client\bin\Debug\cl...

```
 ==========================================
  Client making call to remote Hello object
 ==========================================


 construction of Hello Object
 Hello Object: AppDomain client.exe Thread 6 Context 0
   local Hello object

 calling remote object
 Object reference a proxy?: True
 Client: AppDomain client.exe Thread 6 Context 0

 To call remote object enter string
 Just hit enter to quit:

 > first string
 remote object received message #1
 > second string
 remote object received message #2
 > third string
 remote object received message #3
 >
Press any key to continue_
```

C:\SU\CSE891\Su2002\Lecture7\RemotingDemo\server\bin\Debug\ser...

```
 ====================================================
  Server Supporting Remote Invocation of Hello Object
 ====================================================


 Server: AppDomain server.exe Thread 5 Context 0

 Hit <enter> to exit...
 construction of Hello Object
 Hello Object: AppDomain server.exe Thread 23 Context 0
 first string
 second string
 third string
```

Solution Explorer - HelloLib

- Solution 'RemotingProto' (3 pro
  - client
    - References
    - AssemblyInfo.cs
    - client.cs
  - HelloLib
    - References
    - AssemblyInfo.cs
    - hello.cs
  - server
    - References
    - AssemblyInfo.cs
    - server.cs

Output

Build

```
Build: 3 succeeded, 0 failed, 0 skipped
```

Task List | Output | Find Symbol Results | Index Results for Activator class, all members | Search Results

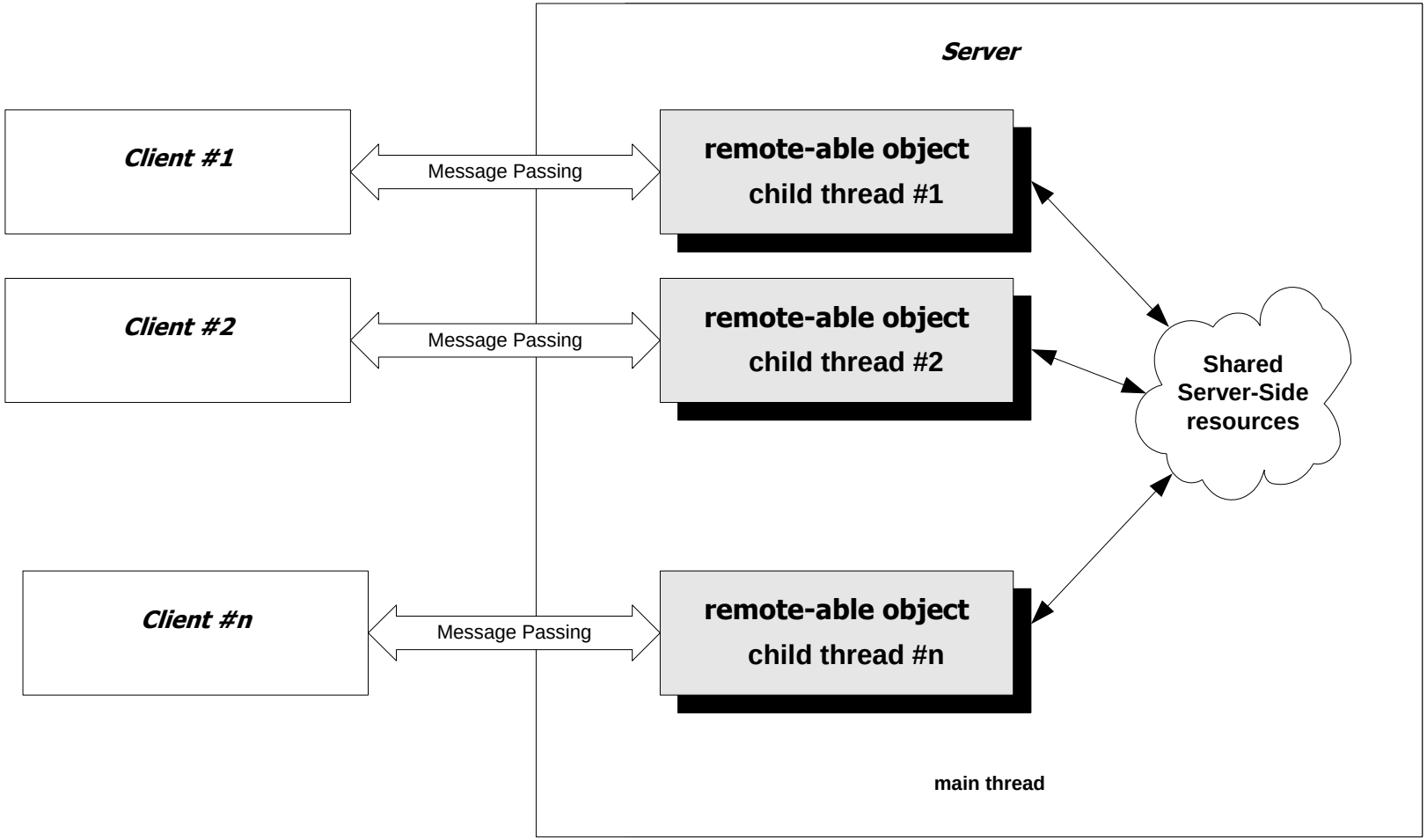
Build succeeded | Ln 26 | Col 3 | Ch 3 | INS

Start | Today !!! | Sticky ... | C:\SU\C... | CMD.EXE | sockets.... | Net Re... | HelloLib ... | C:\SU\C... | C:\SU\... | 9:54 PM

Multiple Clients

- Remote-able objects have one of three activation attributes:
  - <u>Client Activated</u>
    Object is created on first call, then lives a fixed amount of time – its lease – unless it is called again, in which case its lease is extended. Each client gets a new object running on its own thread.
  - <u>Singlecall</u>
    each client gets a new copy of the remote-able object on its own child thread, which exists for the duration of a single call.
  - <u>Singleton</u>
    All clients get a reference to the same remote-able object operating on the only child thread. Singletons also have a lease on life that is renewed on each subsequent call.

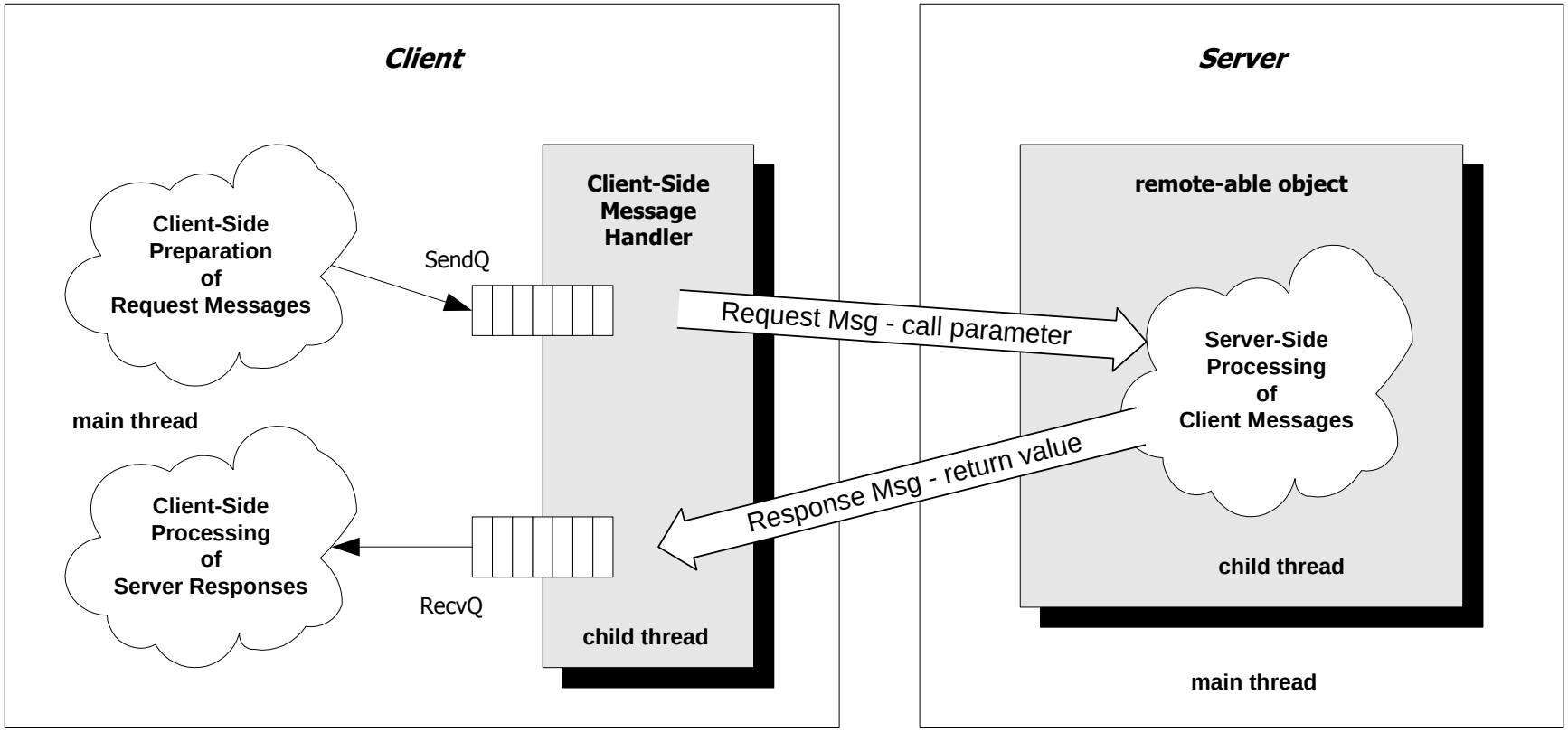
Multiple Clients



Message Passing

- Remoting can be used to construct a message-passing system with very little code.
  - Use a remote-able object for server-side processing.
  - The client main thread creates a child thread to handle sending and receiving messages.
    - The client threads share two First-In-First-Out (FIFO) queues.
    - To make a request of the server, the client main thread composes a request message and posts it to a sendQ shared with the child thread.
    - The child thread deQs the message and sends it as a parameter in a call to the remote server object.
    - When the server processing is complete, the server's response message is returned to the child thread.
    - The child thread posts the return value to the RecvQ.
    - The client main tread dequeues the response for processing.

Message Passing Architecture



Other Topics

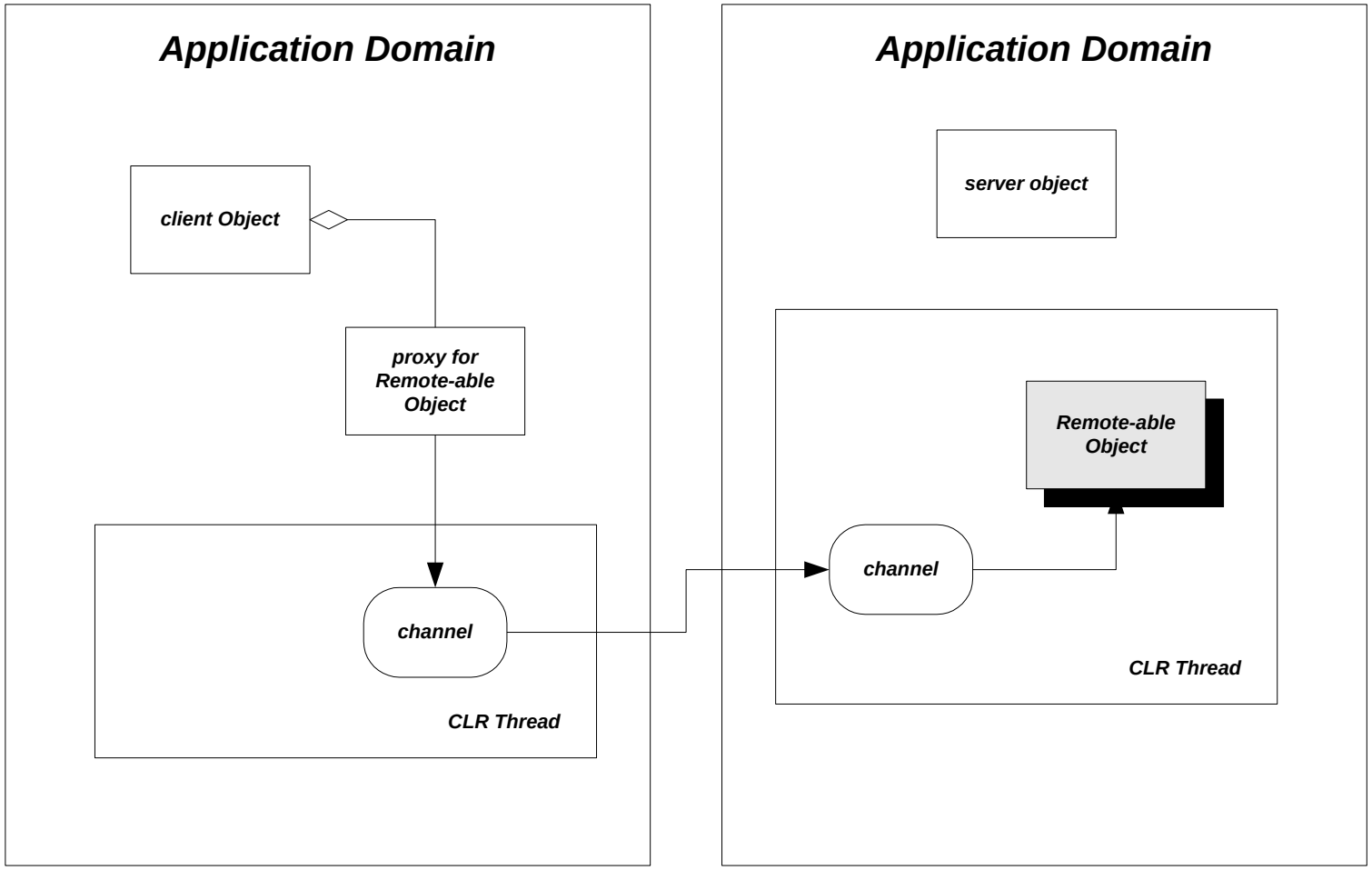
- Prosise discusses:
  - Asynchronous method calls
  - Handling remote events with delegates
  - Declarative configuration (using config files)
  - Client activated remote objects
  - Leases control lifetime of singleton and client activated objects.
  - IIS activation and HTTP channels

Building a Remoting Application

- First create a remote-able object:
  - Design an object to be invoked remotely, derived from MarshalByRefObject
  - Implement the class as a C# library – this creates a dll.
  - Any objects that you need to pass to that object need to be serializable.
    - The basic types like ints and strings already are serializable.
    - Your class objects need to have the [Serializable] attribute and contain only serializable data members.

    Or they can derive from MarshalByRefObject.

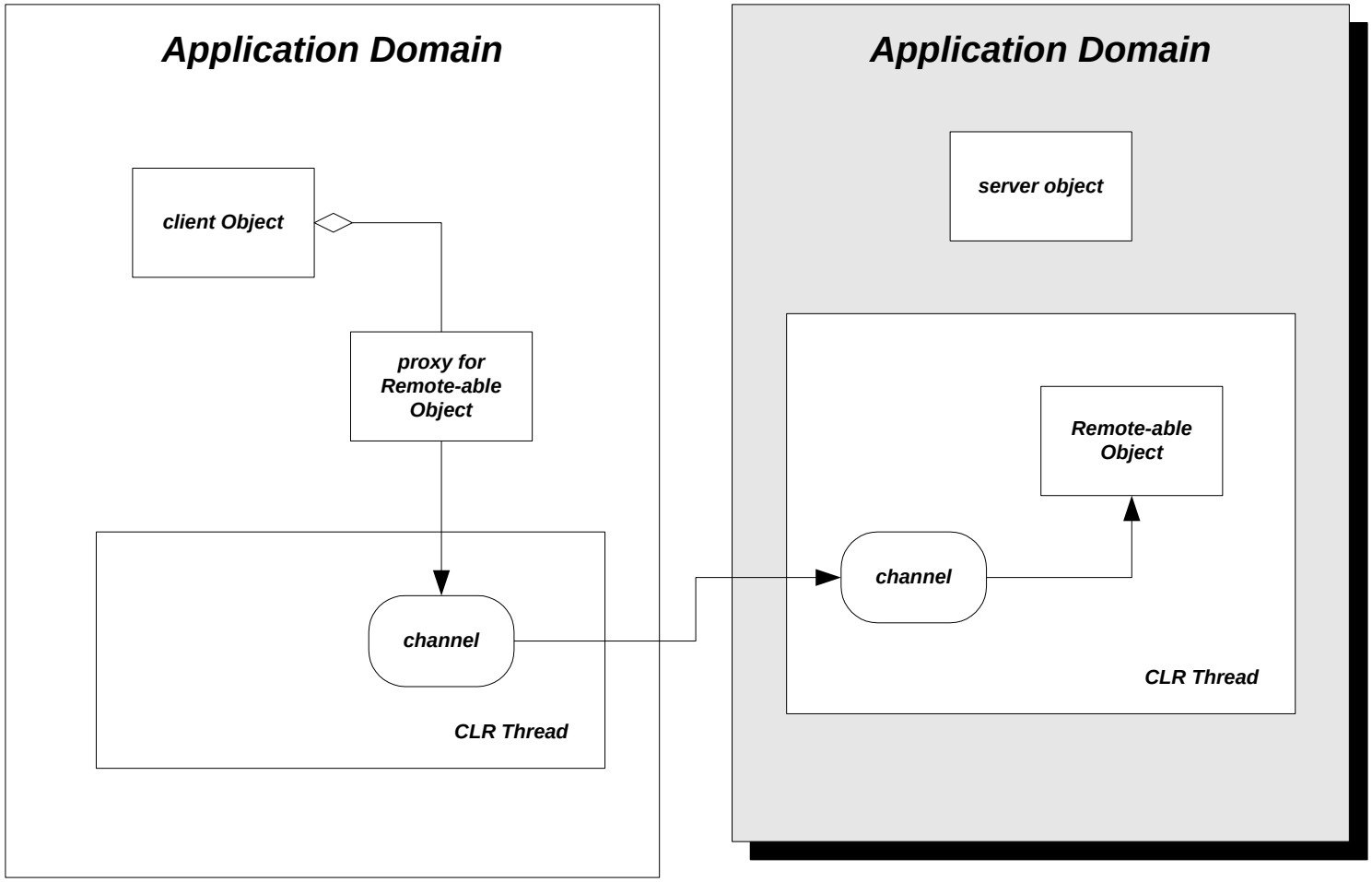
Define Remote Object



Building a Remoting Application

- Create a server:
  - Design a C# class, using a C# Console Application, or Empty Project in which you will create a WinForm.
  - In a member function – main will work:
    - Create a TcpServerChannel
    - Register the Channel with ChannelServices
    - Register the type of object clients want to use remotely by calling RegisterWellKnownServiceType
  - Then, block the main thread.
    - The object will be created by the CLR on its own thread and remote clients will access the object through the CLR.
    - You don't have to write any server code to support this access – the CLR takes care of it for you.
  - Create a reference to the remote-able object's assembly, build, and run the server.

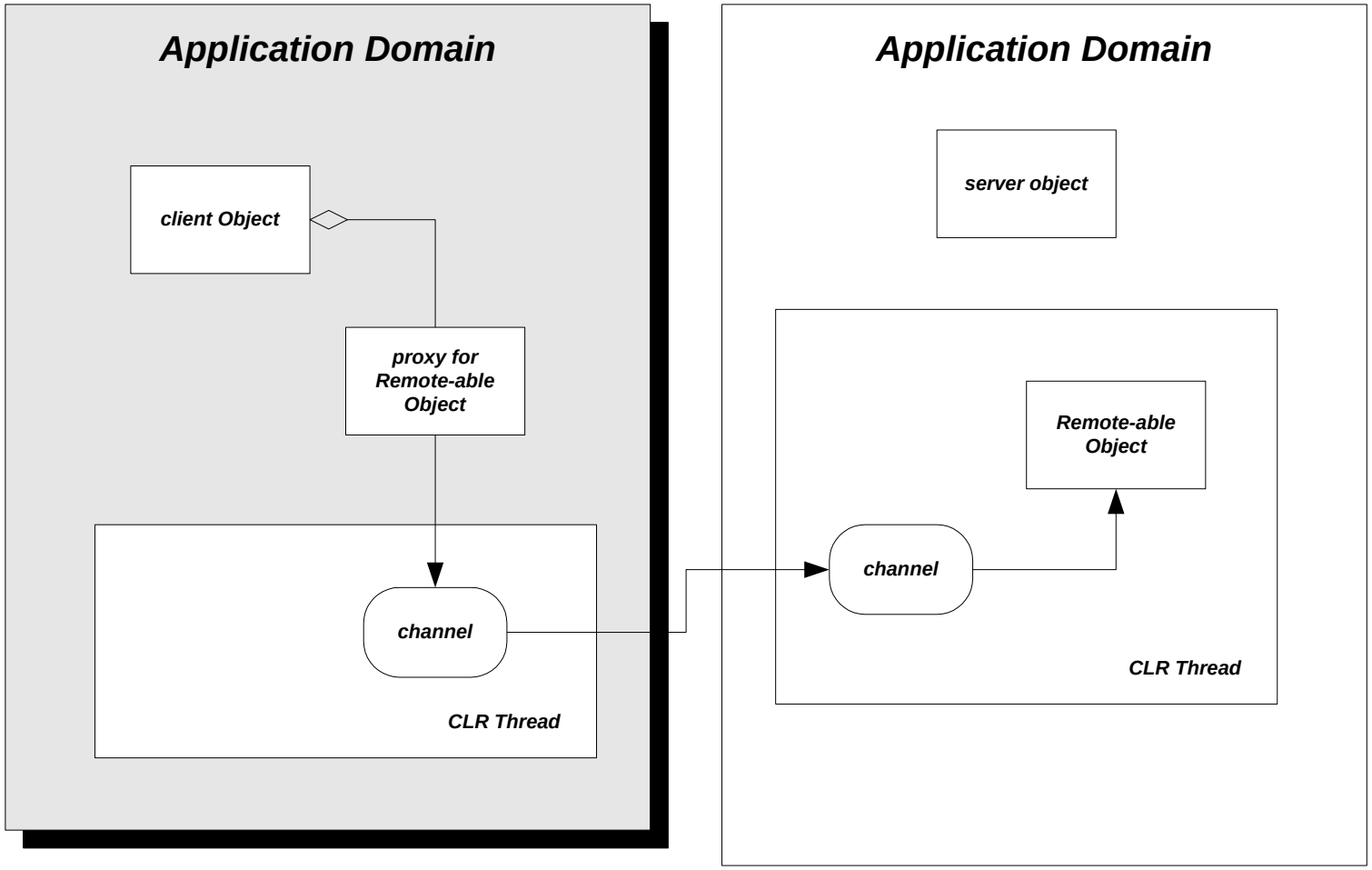
Define the Server



Building a Remoting Application

- Create a client:
  - Design a C# class, using a C# Console Application, or Empty Project in which you will create a WinForm.
  - In a member function – main will work:
    - Create a TcpClientChannel
    - Register the Channel with ChannelServices
  - In a member function – main will work:
    - Create a proxy to access the remote component. You do this by calling Activator.GetObject(…) and casting the result to the type of the remote object.
      - Note that both client and server need the assembly for the remote-able object.
  - Then, make calls on the remote object as needed.
    - You simply use the proxy as if it were the real object.
  - Create a reference to the remote-able object's assembly, build, and run the client.

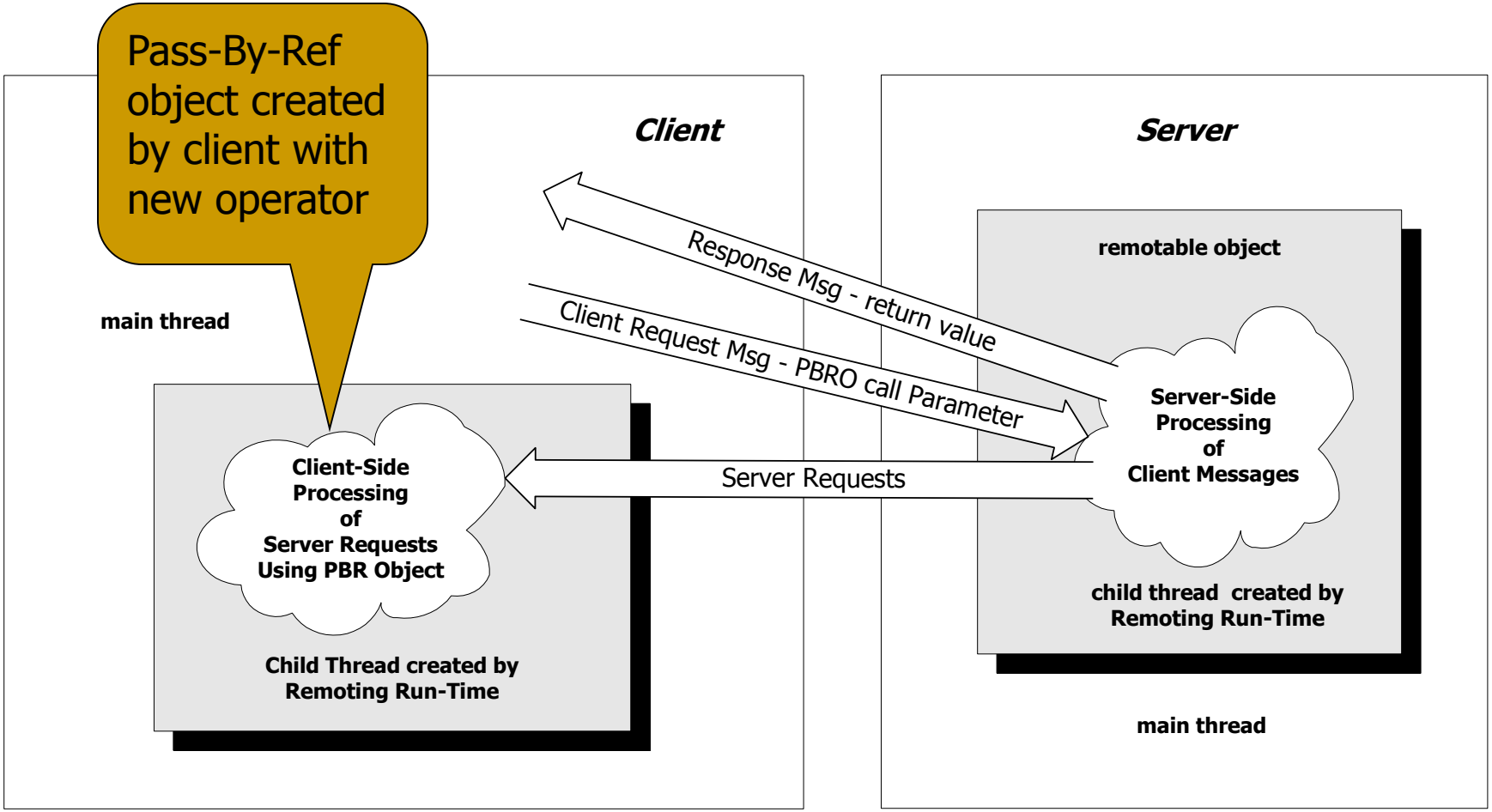
Define the Client



Passing Object Parameters to Remote Methods

- You pass an object to a remote method call:
  - By value
    - Object must be serializable.
    - That usually means that you simply decorate the class declaration with [serializable].
    - Object is declared by client, remoting channel serializes it on client and deserializes it on server.
  - By reference
    - Object must derive from MarshalByRefObject.
    - Client creates object and uses it in method call.
    - Remoting channel activates object on client, using clr thread, and manipulates it to reflect actions by server.

Pass-By-Reference Objects with Remoting



Deployment

- Configuration files
- Server Deployment with Windows Services
- Server Deployment with IIS
- Client Deployment with IIS

Deployment Issues

- Change in server location
  - Does the client hard-code the location and port of remote objects on the server?
- Uses of the application
  - Will this application be used in other ways? For instance, LAN vs Internet use.
- New/additional remotable objects
  - Will we be adding remotable objects after we have built the application?
- Web deployment

Configuration Files

- Rather than hard-code the registration of remote objects and their channels, we can use a configuration file.
- Using a configuration file allows us to do the following without recompiling the server or client:
  - Change the type of channel that is used
  - Add additional remotable objects
  - Change the lifetime settings of remotable objects
  - Add message sinks or formatters to the server or client
- This functionality is available through the System.Runtime.Remoting assembly.

Configuration Files (cont)

- A configuration file is an XML document that is loaded by the server or client.
- Use two different configuration files for the client and the server.
- On the server, load the configuration file using RemotingConfiguration.Configure(“MyServer.exe.config”);
- On the client, load the configuration file using RemotingConfiguration.Configure(“MyClient.exe.config”);
- After loading the configuration file on the client, simply call new on the remotable object class to create a proxy.

Configuration Files (cont)

- Content and structure

```
<configuration>
    <system.runtime.remoting>
        <application>
            <lifetime />
            <channels />
            <service />
            <client />
        </application>
    </system.runtime.remoting>
</configuration>
```

Configuration Files (cont)

- Lifetime
  - The <lifetime> tag allows you to change the lifetime of your remotable objects.
  - Valid attributes:
    - leaseTime – This is the initial lease time that an object will have to live before it is destroyed.
    - sponsorshipTimeout – The time to wait for a sponsor's reply.
    - renewOnCallTime – This is the additional lease time that is added with each call on the remote object.
    - leaseManagerPollTime – Specifies when the object's current lease time will be checked.
  - Note that these apply to Singleton and Client-Activated objects only.

Configuration Files (cont)

- Channels
  - The <channels> element contains the channels that your application will be using. We declare channels with the <channel> tag.
  - The <channel> tag specifies the type, port, and other properties for a particular channel.
  - Valid attributes:
    - ref – “http” or “tcp”
    - displayName – Used for .NET Framework Configuration Tool
    - type – if ref is not specified, contains namespace, classname, and assembly of the channel implementation.
    - port – server side port number. Use 0 on the client if you want to get callbacks from the server.
    - name – Unique names to specify multiple channels (use “”)
    - priority – Sets priority of using one channel over another.

Configuration Files (cont)

- Channels
  - Valid attributes (cont):
    - clientConnectionLimit – Number of simultaneous connections to a particular server (default = 2)
    - proxyName – name of the proxy server
    - proxyPort – port of the proxy server
    - suppressChannelData – specifies whether a channel will add to the ChannelData that is sent when an object reference is created
    - useIpAddress – specifies whether the channel should use IP addresses in URLs rather than hostname of the server
    - listen – setting for activation hooks into listener service
    - bindTo – used with computers that have multiple IP addresses
    - machineName – overrides useIpAddress
    - rejectRemoteRequests (tcp only) – sets local communication only

Configuration Files (cont)

- Providers
  - Sink and formatter providers allow the user to specify the manner in which messages are generated and captured by the framework for each channel.
  - Both the client and server may specify settings for
  - The tags <serverProviders></serverProviders> and <clientProviders></clientProviders> contain the individual settings for each provider or formatter that you wish to set.
  - You can specify one formatter and multiple provider settings.
  - You must place the settings in the order shown:

Configuration Files (cont)

- Example channel entry for a server:

```
<channels>
    <channel ref="http" port="1234">
            <serverProviders>
                    <formatter ref="binary" />
                    <provider type="MySinks.Sample, Server" />
            </serverProviders>
    </channel>
</channels>
```

Configuration Files (cont)

- Providers (cont)
  - Available attributes for formatters and providers:
    - ref – “soap”, “binary”, or “wsdl”
    - type – if ref is not specified, contains namespace, classname, and assembly of the sink provider implementation.
    - includeVersions (formatter only) – specifies whether version information is included with object requests
    - strictBinding (formatter only) – specifies whether the server must use an exact type and version for object requests

Configuration Files (cont)

- Service
  - The <service> tag is used in the server's configuration file to specify the remote objects that will be hosted.
  - Contains <wellknown /> and <activated /> entries for server-activated objects (SAOs) and client-activated objects (CAOs), respectively.
  - Valid attributes for <wellknown />
    - type – Specifies the namespace, classname, and assemblyname of the remote object.
    - mode – Singleton or SingleCall
    - objectUri – Important for IIS hosting (URIs must end in .rem or .soap, as those extensions can be mapped into the IIS metabase.
    - displayName – Optional, used by .NET Framework configuration tool.
  - Valid attributes for <activated />
    - type – Specifies the namespace, classname, and assemblyname of the remote object.

Configuration Files (cont)

- Client
  - The <client> tag is used in the client's configuration file to specify the types of remote objects that it will use.
  - Contains attribute for the full URL to the server if using CAOs.
  - Contains <wellknown /> and <activated /> entries for server-activated objects (SAOs) and client-activated objects (CAOs), respectively.
  - Valid attributes for <wellknown />
    - url – The full URL to the server's registered object
    - type - Specifies the namespace, classname, and assemblyname of the remote object.
    - displayName – Optional, used by .NET Framework configuration tool
  - Valid attributes for <activated />
    - type – Specifies the namespace, classname, and assemblyname of the remote object.

Configuration Files (cont)

- Usage notes:
  - Errors in your configuration file cause the framework to instantiate a local copy of the remote object rather than a proxy when you call new on it. Check the IsTransparentProxy method to be sure you are using a remote object.
  - When you specify assembly names in your <wellknown /> and <activated />, don't include the extension (.dll or .exe).
  - You only have to specify the features that you want/need in your configuration file.
  - You don't have to use the <channel /> setting on the client if you use the default "http" or "tcp" channels on the server. You must specify a port on the server.

Server Deployment with IIS

- If you are concerned about security, then IIS hosting is the best way to go.
- Authentication and encryption features are available through IIS.
- Remote objects are now hosted in IIS; there is no Main() in the server.
- Updates to the server are easy: just copy over the remote object assembly and web.config file. IIS will automatically read the new data.

Server Deployment with IIS

- Procedure:
  - Create a class library for your remotable objects
  - Build the assembly for the class library
  - Create a web.config file for the server
  - Create a virtual directory on the host machine
  - Set the desired authentication methods for the directory
  - Place the web.config file in the virtual directory
  - Create a /bin directory in the virtual directory
  - Place the remotable object assembly in the virtual directory
  - Create a client and configuration file

Client Deployment with IIS

- By placing a WinForm application in a virtual directory, we can stream it to clients.
- When a URL is selected by a client machine, an HTTP request is sent to the server, which streams the application back to the client.
- The application is then stored in the browser cache and also the .NET download cache.
- The runtime opens the application automatically and also makes requests for additional assemblies and files as necessary.
- Be sure to put any remoting configuration files in the virtual directory with the client application.

.Net Remoting

The End