Systems for Design and Implementation

2015-2016

Course 3

Contents

Remote Procedure Call

- *Remote procedure call* (RPC) is an inter-process communication technology that allows a computer program to cause a subroutine or procedure to execute in another address space without the programmer explicitly coding the details for this remote interaction.
- The programmer writes almost the same code whether the subroutine is local to the executing program, or remote.
- When the software is written using object-oriented paradigm, RPC may be referred to as remote invocation or remote method invocation.
- RPC is a popular paradigm for implementing the client-server model of distributed computing.
- A RPC is initiated by the client sending a request message to a known remote server in order to execute a specified procedure using supplied parameters. A response is returned to the client where the application continues along with its process.
- While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution).

.NET Remoting

- *Remoting* is the process of programs or components interacting across certain boundaries.
- These boundaries may be different processes or different machines.
- In the .NET Framework, Remoting technology provides the foundation for distributed applications.
- Remoting implementations generally distinguish between *remote objects* and *mobile objects*.
- A *remote object* provides the ability to execute methods on remote servers, passing parameters and receiving return values. The remote object will always "stay" at the server, and only a reference to it is passed around among other machines.
- *Mobile objects* pass a context boundary. They are serialized into a general representation (i.e., a binary or XML) in one context and then they are deserialized in the other context involved in the process. Server and client both hold copies of the same object.

.NET Remoting

- There are two very different kinds of objects when it comes to remoting: objects that are passed by reference and objects that are passed by value.
- *MarshalByRefObject*s allow the ability to execute remote method calls on the server side. These objects live on the server and only a so-called *ObjRef* will be passed around.
- The client does not usually have the compiled objects in one of its assemblies. Only an interface or a base class is available to it.
- Every method, including property gets/sets, will be executed on the server.
- The .NET Framework automatically generates proxy objects that take care of all remoting tasks. The client calls the remote method the same way it calls a local one.

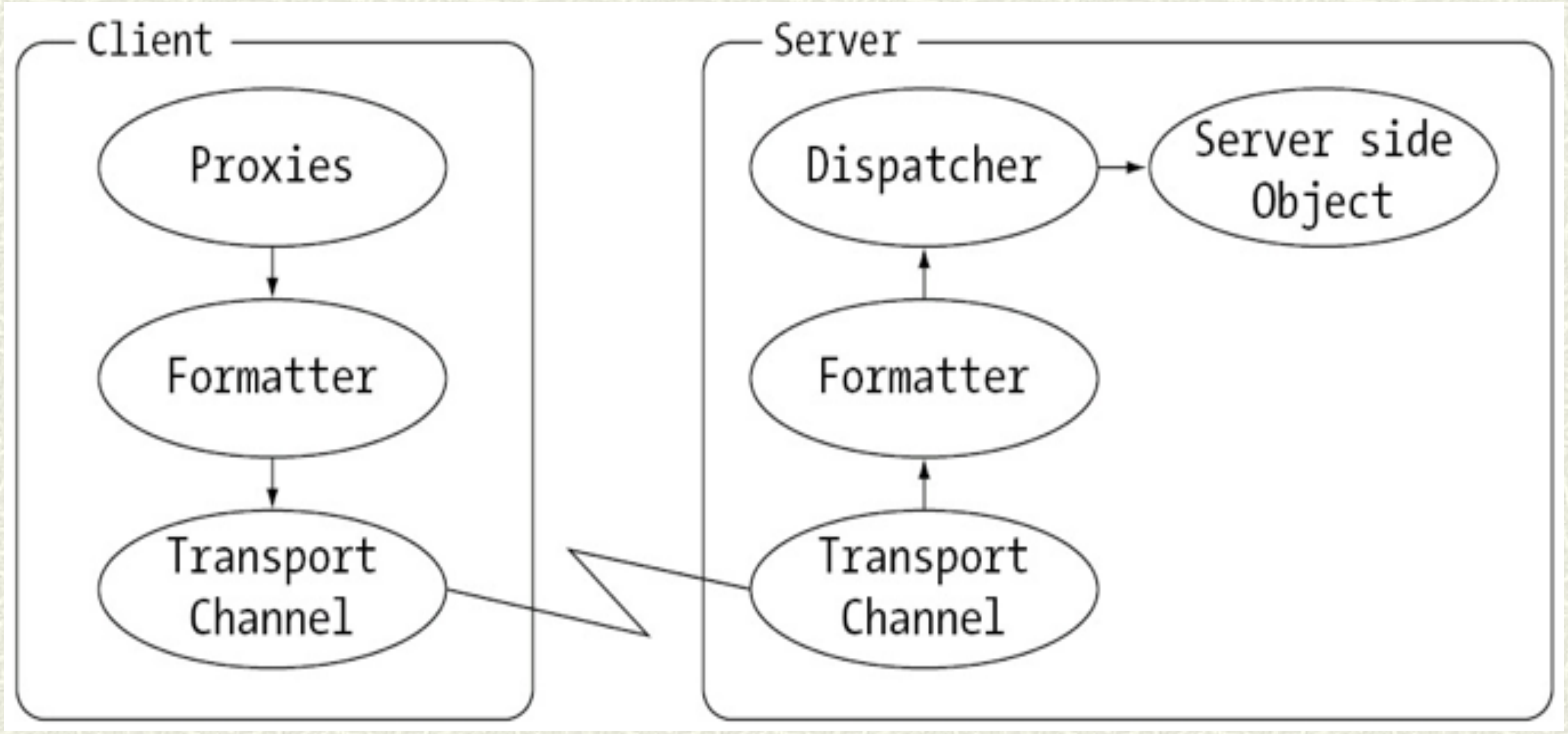
Passed by value objects in Remoting

- When these objects are passed over remoting boundaries (as method parameters or return values), they are serialized and, then, restored as a copy on the other side of the communications channel.
- After this re-construction, there is no notation of client or server for this kind of object; each one has its own copy, and both run absolutely independently.
- Methods called on these objects will execute in the same context as the origination of the method call.
- The only requirement for an object to be passable by value is that it supports serialization (`Serializable` attribute or `ISerializable` interface).

Remark:

Both the server and the clients must have the compiled object corresponding to the pass by value object in their assemblies (for serialization and deserialization).

Remoting


Client
Proxies
Formatter
Transport Channel
Server
Dispatcher
Server side Object
Formatter
Transport Channel

Architecture of a Remoting Application

- At least three assemblies are needed for any Remoting application:
  - A *shared assembly* (.dll) that contains serializable objects and interfaces or base classes to MarshalByRefObjects.
  - A *server assembly* (.exe) that implements the MarshalByRefObjects.
  - A *client assembly* (.exe) that uses the MarshalByRefObjects.

Remark

Both the server assembly and the client assembly have a reference to the shared assembly.

System.Runtime.Remoting

- The `System.Runtime.Remoting` namespace provides classes and interfaces that allow developers to create and configure distributed applications.
- The `RemotingConfiguration` class contains static methods for interfacing with configuration settings. The `Configure` method allows developers to configure the remoting infrastructure through the use of XML formatted configuration files.
- The `RemotingServices` class provides a number of methods to help in using and publishing remoted objects.
- The `ObjRef` class holds all the relevant information required to activate and communicate with a remote object. It is a serializable representation of an object that is transmitted to a remote location using a channel, where it is unmarshaled and can be used to create a local proxy of the remoted object.

System.Runtime.Remoting

- The `System.Runtime.Remoting.Channels` namespace contains classes that support and handle channels and channel sinks, which are used as the transport medium when a client calls a method on a remote object.
- The `System.Runtime.Remoting.Channels.Http` namespace contains channels that use the HTTP protocol to transport messages and objects to and from remote locations.
- The `System.Runtime.Remoting.Channels.Ipc` namespace defines a communication channel for remoting that uses the interprocess communication (IPC) system of the Windows operating system.
- It does not use network communication, so it is much faster than the HTTP and TCP channels, but it can only be used for communication between application domains on the same physical computer.

System.Runtime.Remoting

- The `System.Runtime.Remoting.Channels.Tcp` namespace contains channels that use the TCP protocol to transport messages and objects to and from remote locations. By default, the TCP channels encode objects and method calls in binary format for transmission.
  - `TcpClientChannel` implements a client channel that uses the TCP protocol to transmit messages (for remote calls).
  - `TcpServerChannel` implements a server channel for remote calls that uses the TCP protocol to transmit messages.
  - `TcpChannel` provides a channel implementation that uses the TCP protocol to transmit messages.
- The `TcpChannel` class combines the `TcpClientChannel` class and the `TcpServerChannel` class, and can be used to both receive and send messages using the TCP protocol.

Remoting Example

- A simple remoting application:
  - The remote object has one method that transforms a string to uppercase and appends the date and the time when it was transformed.
  - The client calls the method on the remote object and prints the result.

Remoting Example –Shared assembly

```
namespace RemotingServices
{
    public interface IServer{
        String transform(String txt);
   }
}
```

It is compiled as a library: `RemotingServices.dll`

Remoting Example –Server assembly

```
using RemotingServices;
namespace RemotingServer
{
    class ServerImpl: MarshalByRefObject, IServer {
        public string transform(string txt)
         {
            Console.WriteLine("Processing text: "+txt);
             return txt.ToUpper() +' ' +DateTime.Now;
         }
   }
}
```

Remoting Example –Server assembly

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

namespace RemotingServer{
    class RemotingServer {
        static void Main(string[] args) {
            TcpChannel channel=new TcpChannel(55555);
            ChannelServices.RegisterChannel(channel,false);
            RemotingConfiguration.RegisterWellKnownServiceType(
                                         typeof(ServerImpl),
                                         "StringConvertor",
                                         WellKnownObjectMode.Singleton);
            // the server  keeps running until keypress.
            Console.WriteLine("Server started ...");
            Console.WriteLine("Press <enter> to exit...");
            Console.ReadLine();
        }
    }
}
```

Remoting Example –Client assembly

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using RemotingServices;
namespace RemotingClient{
  class RemotingClient{
     static void Main(string[] args){
       try{
          TcpChannel channel = new TcpChannel();
          ChannelServices.RegisterChannel(channel, false);
IServer server = (IServer)Activator.GetObject(typeof(IServer),
                      "tcp://localhost:55555/StringConvertor");
           String txt = "Ana are mere.";
           Console.WriteLine("Calling remote method "+txt);
           String response = server.transform(txt);
           Console.WriteLine("Response:"+response);
       }catch(Exception e){…}
}}}
```

Types of Remoting

- There are two very different types of remote interaction between components:
  - One uses serializable objects that are passed as a copy to the remote process (ByValue objects).
  - The second employs server-side (remote) objects that allow the client to call their methods (MarshalByRefObjects).
- Remarks:
  - ByValue objects are not remote objects. All methods on these objects are executed locally (in the same context) to the caller.
  - The compiled class has to be available to the client and the server.
  - When a ByValue object holds references to other objects, these objects have to be either serializable or MarshalByRefObjects; otherwise, an exception is thrown.

MarshalByRefObjects

- A *MarshalByRefObject* is a remote object that runs on the server and accepts method calls from the client.
- Its data is stored in the server's memory and its methods executed in the server's AppDomain.
- *MarshalByRefObjects* can be categorized into two groups: server-activated objects (SAOs) and client-activated objects (CAOs):
  - *Server-Activated Objects* 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.
  - *Client-Activated Objects* A client-activated object behaves mostly the same way as does a "normal" .NET object. When a creation request on the client is encountered (using `Activator.CreateInstance()` or the `new` operator), an activation message is sent to the server, where a remote object is created. On the client, a proxy that holds the `ObjRef` to the server object is created.

Server-Activated Objects

- The server decides whether a new instance of a SAO will be created or an existing object will be reused, when a client calls a method on the SAO.
- SAOs can be marked as either *Singleton* or *SingleCall*. In the first case, one instance serves the requests of all clients in a multithreaded fashion.
- When using objects in *SingleCall* mode, a new object will be created for each request and destroyed afterwards.

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(<MRO>),  "<URL>", WellKnownObjectMode.SingleCall);

RemotingConfiguration.RegisterWellKnownServiceType(
         typeof(<MRO>), "<URL>", WellKnownObjectMode.Singleton);
```

Published SA Objects

- When using either *SingleCall* or *Singleton* objects, the necessary instances are created dynamically during a client's request.
- If an already created object must be published (i.e., it does not have a default constructor) `RemotingServices.Marshal()` should be used.
- After publishing, the object behaves like a *Singleton* SAO. The only difference is that the object has to already exist at the server before publication.

```
 RemoteObject obj = new RemoteObject(<actual params list>);
RemotingServices.Marshal(obj,“IdentificationString");
```

Lifetime of a remote object

- Each server-side object is associated with a *lease* upon creation.
- This lease will have a time-to-live counter (five minutes by default) that is decremented in certain intervals. In addition to the initial time, a defined amount (two minutes in the default configuration) is added to this time to live upon every method call a client places on the remote object.
- When this time reaches zero, the framework looks for any *sponsors* registered with this lease.
- A sponsor is an object running on the server itself, the client, or any machine reachable via a network that will take a call from the .NET Remoting Framework asking whether an object's lifetime should be renewed or not.
- When the sponsor decides that the lease will not be renewed or when the framework is unable to contact any of the registered sponsors, the object is marked as timed out and then garbage collected.
- When a client still has a reference to a timed-out object and calls a method on it, it will receive an exception.

Lifetime of a remote object

- To change the default lease times, you can override InitializeLifetimeService() from the MarshalByRefObject.

```
class MyRemoteClass: MarshalByRefObject, IRemoteInterface   {
  public override object InitializeLifetimeService(){
    ILease lease = (ILease)base.InitializeLifetimeService();
    if (lease.CurrentState == LeaseState.Initial){
      lease.InitialLeaseTime = TimeSpan.FromMilliseconds(10M);
      lease.SponsorshipTimeout = TimeSpan.FromMilliseconds(10M);
      lease.RenewOnCallTime = TimeSpan.FromMilliseconds(10H);
    }
        return lease;
 }
      // rest of implementation ...
}
```

Lifetime of a remote object

- To make a Singleton remote object live forever, the overriden method must return null.

```
class InfinitelyLivingSingleton: MarshalByRefObject{
   public override object InitializeLifetimeService()    {
      return null;
   }
    // ...
}
```