.NET Remoting

Tomáš Matoušek

&

Ladislav Prošek

tmd.havit.cz/teaching/csharp.htm

Recall: Prerequisities

- AppDomain
  - “light-weight process”
  - provides isolation between applications
    - cheaper than process
    - isolation enforced by the fact that instances are created, live and die in a single AppDomain

- Serialization
  - process of converting an object (or object graph) into a contiguous stream of bytes and vice versa
  - used for persistence, remoting, ...
  - performed by formatters
    - System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
    - System.Runtime.Serialization.Formatters.Soap.SoapFormatter
  - serializable types must by decorated with [Serializable]
  - tweaking: ISerializable, IDeserializationCallback, surrogates, ...

Introduction

- .NET Remoting is a generic and extensible interprocess (inter-AppDomain) communication system

- related technologies
  - RPC (Remote Procedure Call)
  - Java RMI (Remote Method Invocation)
  - OMG CORBA (Common Object Request Broker Architecture)
  - MS DCOM (Distributed Component Object Model)
  - Web Services

- distinct features
  - no IDL - metadata will do!
  - choice of communication protocol (TCP, HTTP, ..., write-your-own)
  - choice of data format (SOAP, binary, ..., write-your-own)

Remoting vs. Web Services

- XML Web Services
  - simple, loosely coupled programming model
  - XML Schema type system
  - broad cross-platform reach
    - standards: HTTP, XML, XSD, SOAP, WSDL
  - based on ASP.NET, services hosted in IIS

- .NET Remoting
  - tightly coupled and more complex programming model
  - CLR type system
  - limited reach – all participants should be built using .NET
  - services hosted in any AppDomain
    - Windows Service, IIS, console application, WinForm application, ...

Remotable types

- marshal-by-value (MBV)
  - all serializable types
  - can be used as parameters of remote calls
    - copy of the object is passed to the remote application
    - however, methods on them will always be executed locally!
- marshal-by-reference (MBR)
  - types derived from System.MarshalByRefObject
  - can be used as parameters of remote calls
    - a proxy is passed to the remote application
    - calls on the proxy are marshaled back to the AppDomain where the MBR object lives – methods are executed remotely
- you always need an MBR object to make remote calls

Activation

- Server activated objects (SAO)
  - reachable via a well-known URL
  - two lifetime modes
    - Singleton – one instance serves the requests of all clients - stateful
    - SingleCall – a new instance is created for each request - stateless
  - constructed implicitly by the server when a method is invoked (for the first time)
  - must have the default (parameterless) constructor

- Client activated objects (CAO)
  - constructed explicitly by the client using **new** or Activator.CreateInstance with an arbitrary constructor
  - methods are then invoked on this “private” instance
  - stateful

Server activated objects

- we would like to make the following class remotely accessible:

```
public class PrintService : MarshalByRefObject
{
    public void Print(string str)
    {
        Console.WriteLine(str);
    }
}
```

- server code:

```
ChannelServices.RegisterChannel(new TcpChannel(4321)); // the server will listen on port 4321
RemotingConfiguration.RegisterWellKnownServiceType(typeof(PrintService), "MyPrintService",
      WellKnownObjectMode.SingleCall);
```

- client code:

```
ChannelServices.RegisterChannel(new TcpChannel());
PrintService service = (PrintService)Activator.GetObject(typeof(PrintService),
      "tcp://localhost:4321/MyPrintService"); // creates a proxy
service.Print("Hello");
```

- alternative client code:

```
ChannelServices.RegisterChannel(new TcpChannel());
RemotingConfiguration.RegisterWellKnownClientType(typeof(PrintService),
      "tcp://localhost:4321/MyPrintService");
PrintService service = new PrintService(); // creates a proxy
service.Print("Hello");
```

Client activated objects

- server code:

```
ChannelServices.RegisterChannel(new TcpChannel(4321)); // the server will listen on port 4321
RemotingConfiguration.RegisterActivatedServiceType(typeof(SimpleService));
```

- client code:

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

// activates the remote object and creates a proxy:
PrintService service = (PrintService)Activator.CreateInstance(typeof(PrintService),
      new object[0], new object[] { new UrlAttribute("tcp://localhost:4321/") });
service.Print("Hello");
```

- alternative client code:

```
ChannelServices.RegisterChannel(new TcpChannel());
RemotingConfiguration.RegisterActivatedClientType(typeof(PrintService),
      "tcp://localhost:4321/");
PrintService service = new PrintService(); // activates the remote object and creates a proxy
service.Print("Hello");
```

- Behind the scenes there is a well-known factory SAO with URL tcp://localhost:4321/RemoteActivationService.rem that activates the CAOs.

Configuration files

- Remoting infrastructure can configure itself automatically according to an XML file
  - initialization code on both server and client then shrinks to something like

```
RemotingConfiguration.Configure("MyCfg.config");
```

- Sample configuration file:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown
               type="PrintService, PrintService, Version=..., ..."
               url="tcp://localhost:4321/MyPrintService"
            />
         </client>
         <channels>
            <channel ref="tcp" port="0" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Lifetime

- When should the objects be “deactivated”?

- .NET Remoting implements the lease-based object lifetime
  - when a MBR object is remoted (a reference to it is passed to another AppDomain), it is associated with a lease object implementing the ILease interface
  - the lease keeps a time-to-live counter that is periodically decremented in certain intervals by the lease manager and incremented/adjusted upon every remote call to the MBR
  - when the counter reaches 0, the MBR is removed from the table of active objects and eventually garbage collected

- time intervals
  - initial lease time (default = 5 minutes)
  - renew-on-call time (default = 2 minutes)
  - lease manager poll time (default = 10 seconds)

Lifetime sponsors

- it is often desirable to keep objects alive even if there are no calls
- sponsors (objects implementing the ISponsor interface) can be registered with leases in order to be notified when the time-to-live counter is low and have a chance to increase it
- sponsors can be remote objects as well
  - it is a good idea to register a sponsor that lives in client's AppDomain so that the server virtually pings the client to ensure that it is still alive
- sample client code:

```
class Sponsor : MarshalByRefObject, ISponsor
{
    public TimeSpan Renewal(ILease lease)
    {
        return TimeSpan.FromMinutes(1);
    }
}

// r is a remote MBR object
((ILease)r.GetLifetimeService()).Register(new Sponsor());
```

CallContext

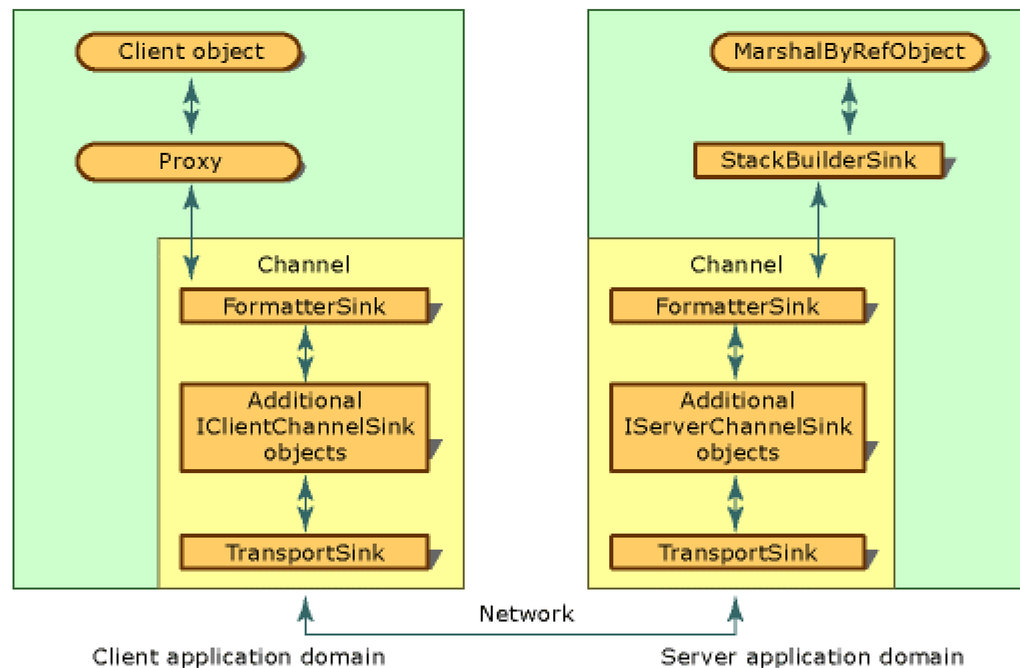
- System.Runtime.Remoting.Messaging.CallContext represents a logical thread context
  - provides named data slots that can store arbitrary data
    - should be either MBV or MBR objects and implement the ILogicalThreadAffinative interface
  - unlike [ThreadStatic] and thread local store (TLS), CallContext is preserved across remote calls
  - useful for passing configuration, transactional context, ...

```
[Serializable]
public class UserContext : ILogicalThreadAffinative
{
    public string UserName;
    public string Password;
    ...
}

CallContext.SetData("UserContext", new UserContext(...));
server.DoFoo();
// the UserContext will be passed to the server as a hidden parameter,
// the server can do CallContext.GetData("UserContext") to access it
```

Channels

- components that are responsible for passing messages across remoting boundaries
- consist of a sink chain
  - formatter sinks serialize calls (represented by IMessage) into streams and back
  - transport sinks transport streams to the other side
  - custom sinks intercept calls and do whatever you want them to do
    - encryption, compression, logging, ...



Channels (cont.)

- BCL comes with two channels
  - System.Runtime.Remoting.Channels.Tcp.TcpChannel
    - uses tcp:// URL prefix (scheme)
    - default sink chain
      - client side: BinaryClientFormatterSink → TcpClientTransportSink
      - server side: TcpServerTransportSink → BinaryServerFormatterSink
  - System.Runtime.Remoting.Channels.Http.HttpChannel
    - uses http:// scheme
    - default sink chain
      - client side: SoapClientFormatterSink → HttpClientTransportSink
      - server side: HttpServerTransportSink → SoapServerFormatterSink

- both channels can be configured to use a custom sink chain (e.g. SOAP formatter with TCP or binary formatter with HTTP)
  - just create a chain of sink providers (except for the transport sink which will be appended automatically) and pass it to the channel's constructor

```
Hashtable properties = new Hashtable(); properties[“port”] = 8080;
new HttpChannel(properties, new BinaryClientFormatterSinkProvider(),
               new BinaryServerFormatterSinkProvider());
```

Common problems

- CAO becomes unreachable after 5 minutes of inactivity
  - this is how the lifetime service behaves by default
  - solution: override MBR.InitializeLifetimeService() and/or register sponsors
- callbacks don't work
  - if the channel is instantiated using the parameterless constructor, only the client part is created
  - solution: use the constructor that takes port as parameter and pass 0
    ```
    new TcpChannel(0);
    ```
- I am getting SerializationException: "Because of security restrictions, the type ... cannot be accessed."
  - this is a new security feature of Framework 1.1
  - solution: set the TypeFilterLevel property of the formatter to "Full"

- beware of the slight semantic shift when passing MBV reference types as 'in' args

```
[Serializable] class A { int x; }

void f(A a)
{ a.x++; }
```

  - when f is called locally, caller sees that x was changed
  - when f is called remotely, the change is not propagated back to caller!
  - solution: `void f(ref A a)`