.NET Remoting

V 1.0

Agenda

Part 0 - Introduction

Distributed Objects

Part I - .NET Remoting

Overview

Activation

Leases

Configuration

Limitations

Part 0

Distributed Object Systems

Motivation

Why do we need a “Distributed Object System”?

- different systems
- load balancing / efficiency / cost
  - distribute work among many machines
  - too many objects to fit on one machine
  - too many request to be processed by one machine
  - cluster is cheaper than a supercomputer
- administrative
  - customer and provider are different entities (e.g. B2B)
- security
  - different parts of the system require different protection
  - database behind firewall, webserver on internet
- specialization
  - machine may be optimized / configured for particular task

ELCA

Problems

L.E.A.F. by ELCA

- object references
- pointers
- different type systems
- references vs. values
- distributed garbage collection
- object location
- naming, naming service
- object activation
- data protocol
- message protocol

Definitions

Serialization

conversion of an object‘s instance into a byte stream

Deserialization

conversion of a stream of bytes into an object‘s instance

Marshaling

gathering and conversion (may require serialization) to an appropriate format of all relevant data, e.g in a remote method call; includes details like name representation.

Existing Solutions

RPC / XDR

first implementation of distributed procedure calls (before OO times) [SUN]

CORBA / IIOP

standard for distributed object systems [OMG]

Java RMI

java framework for distributed java objects; RMI / IIOP is based on a subset of CORBA [SUN]

.NET Remoting

distributed object framework in .NET. [Microsoft]

WebServices

distributed method calls (not a distributed object system!) over SOAP + HTTP

ELCA

Part I

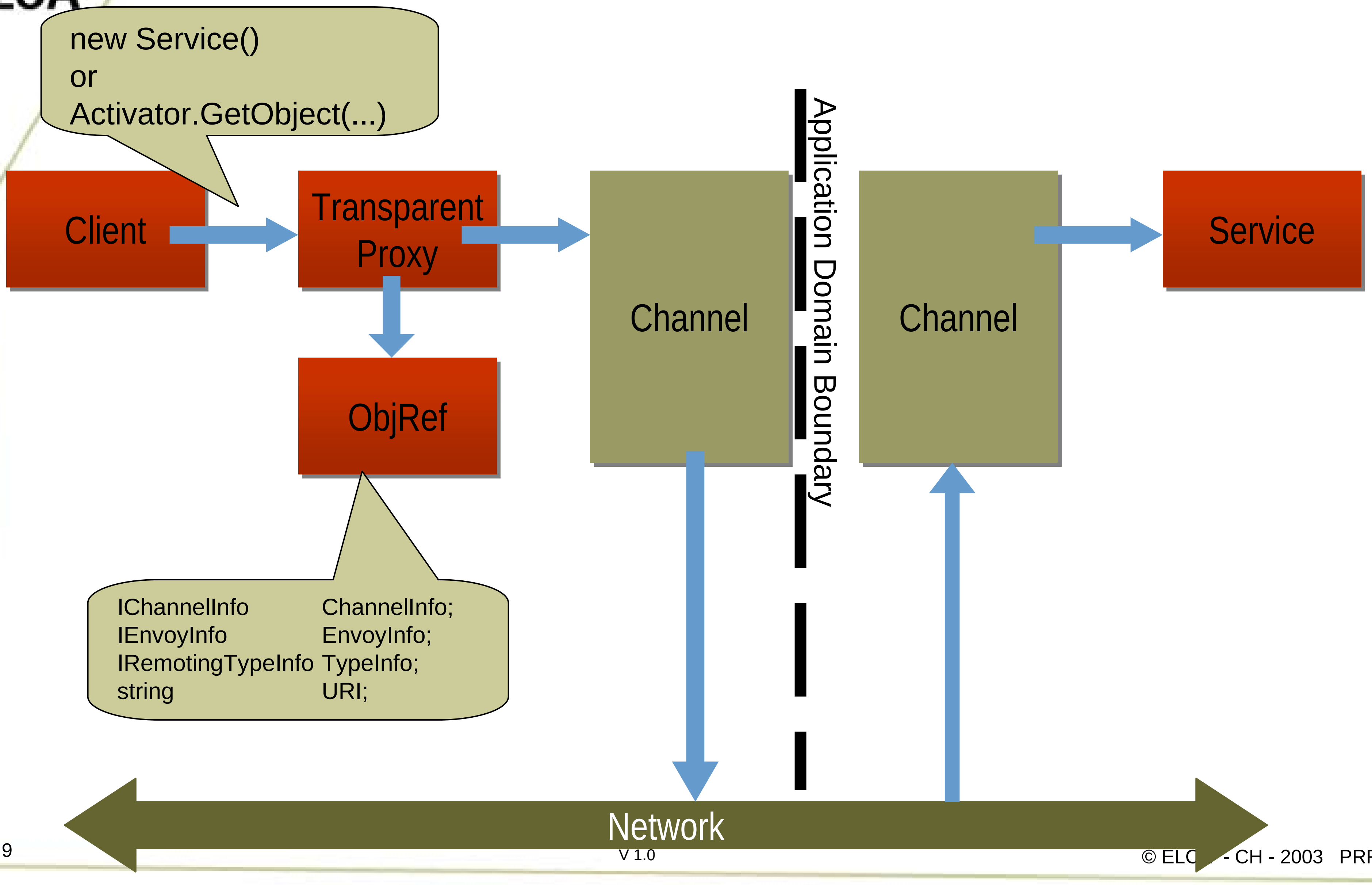
.NET Remoting

References

Teach yourself .NET remoting.....

- Visual Studio .NET / MSDN online documentation
- Ingo Rammer; **Advanced .NET Remoting**; APress
- http://www.dotnetremoting.cc/

Remoting Overview



new Service()
or
Activator.GetObject(...)

Client

Transparent Proxy

ObjRef

Channel

Application Domain Boundary

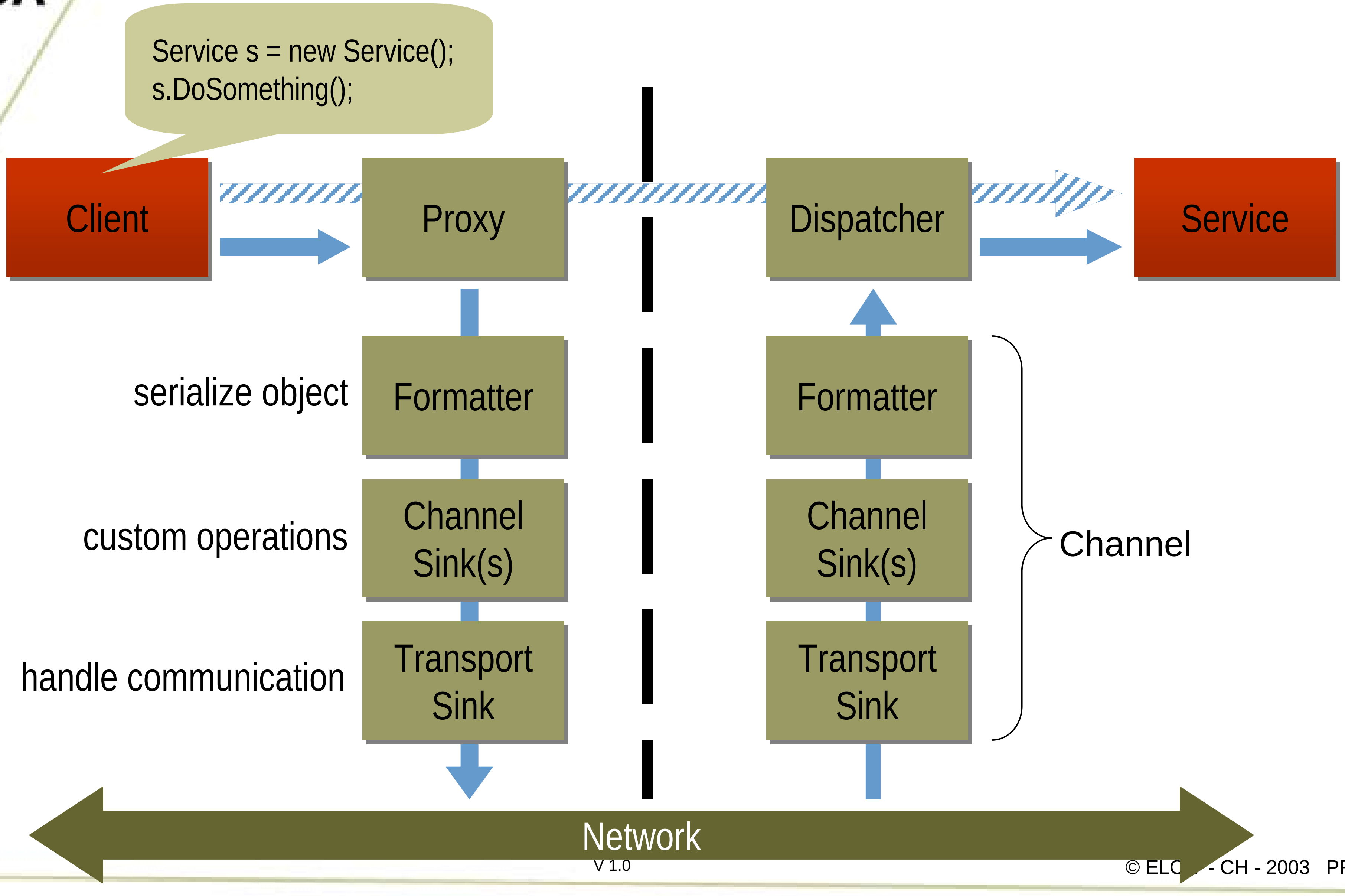
Channel

Service

```
IChannelInfo        ChannelInfo;
IEnvoyInfo          EnvoyInfo;
IRemotingTypeInfo   TypeInfo;
string              URI;
```

Network

Channels Overview



```
Service s = new Service();
s.DoSomething();
```

Channel Components

Channel Formatters

SOAP

- serialization using SOAP
- customizable
  - [Element(name="...")]
  - [Attribute(name="...")]
  - [XmlIgnore]
- SOAP-incompatible!
- slow

binary

- binary serialization
- customizable
  - [NonSerializable]
  - ISerializable Interface

custom

Transport Sinks

- TCP
- HTTP
- HTTPS
  - requires hosting in IIS!
- custom

Notes

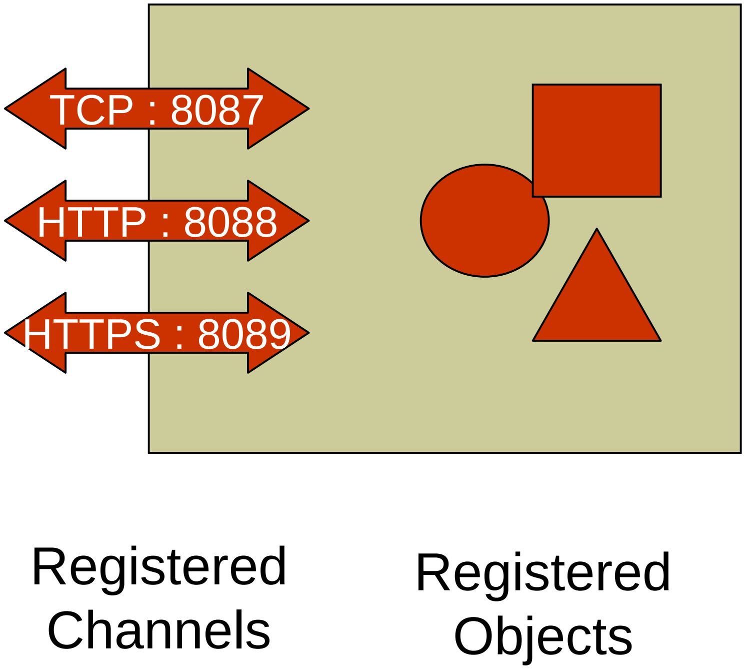
- Formatters and Transport sinks freely combinable
- SOAP+HTTP not compatible with WebServices

Channel Configuration

Objects and Channels are both registered to the remoting system

Channels and Objects are orthogonal:

- There is no way to force an object to use a given channel
- There is no way to limit a channel to a set of machines



Channel Registration

Code Configuration

XML Configuration

```
HttpChannel channel = new HttpChannel();
ChannelServices.RegisterChannel(channel);
```

```
<channels>
        <channel ref=„http“/>
</channels>
```

Client-side channel with SOAP formatter on HTTP transport-sink

```
TcpChannel channel = new TcpChannel(1234);
ChannelServices.RegisterChannel(channel);
```

```
<channels>
        <channel ref=„tcp“ port=„1234“ />
</channels>
```

Server-side channel with binary formatter on TCP transport-sink at port 1234

```
ListDictionary prop = new ListDictionary();
prop.Add("port", 4321);

HttpChannel channel = new HttpChannel(
        prop,
        new BinaryClientFormatterSinkProvider(),
        new BinaryServerFormatterSinkProvider());

ChannelServices.RegisterChannel(channel);
```

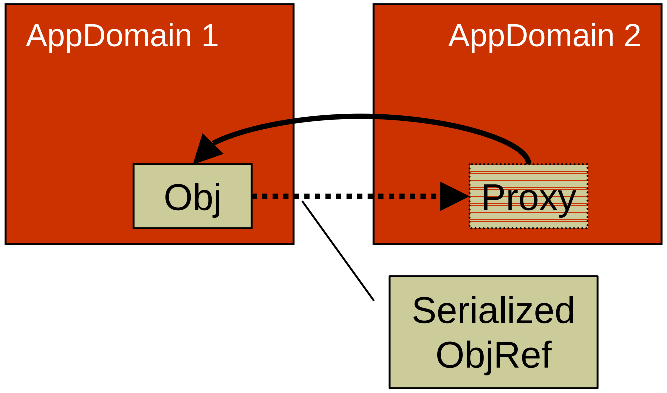
```
<channels>
        <channel ref=„http“ port=„4321“>
                <serverProviders>
                        <formatter ref=„binary“ />
                </serverProviders>
        </channel>
</channels>
```

Server-side channel with binary formatter on HTTP transport-sink at port 4321

Object Marshaling

MarshalByRefObjects

- remoted by reference
- client receives an ObjRef object, which is a“pointer“ to the original object

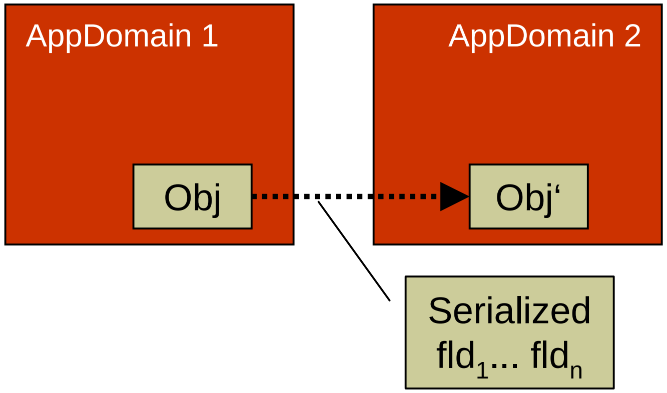


[Serializable]

- all fields of instance are cloned to the client
- [NonSerialized] fields are ignored

ISerializable

- object has method to define own serialization



Remoting Activation

Server-Side Activation (Well-Known Objects)

Singleton Objects

- only one instance is allocated to process all requests

SingleCall Objects

- one instance per call is allocated

“stateless”

Client-Side Activation

Client Activated Objects

- the client allocates and controls the object on the server

“stateful”

Scenario

Service Object

```
interface IMyRemoteObject {
        void DoThis(...);
        void DoThat(...);
}

public class MyRemoteObject: MarshalByRefObject, IMyRemoteObject {
        public MyRemoteObject() {...}
        public void DoThis(...) {...}
        public void DoThat(...) {...}
}
```

Server-Activated Objects

Client

```
IMyRemoteObject obj = (IMyRemoteObject) Activator.GetObject(
            typeof(IMyRemoteObject),
            „http://localhost:1234/MyRemoteObject.soap“);
```

Server (SingleCall)

```
RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(MyRemoteObject),
            „MyRemoteObject.soap“,
            WellKnownObjectMode.SingleCall);
```

the parameterless constructor is called on instantiation

Server (Singleton)

```
RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(MyRemoteObject),
            „MyRemoteObject.soap“,
            WellKnownObjectMode.Singleton);
```

the parameterless constructor is called on instantiation

```
MyRemoteObject obj = new RemoteObject(XYZ);
RemotingServices.Marshal(obj, „MyRemoteObject.soap“);
```

initialized object

Client Activated Objects

Client

```
RemotingConfiguration.RegisterActivatedClientType(
        typeof(MyRemoteObject), „http://localhost:1234/MyServer“);

MyRemoteObject obj = new MyRemoteObject();
```

Server

```
RemotingConfiguration.ApplicationName = „MyServer“;
RemotingConfiguration.RegisterActivatedServiceType(typeof(MyRemoteObject));
```

Remarks

- allocation with new ⇒ class must be present!
  - same class
  - stub (created with soapsuds): limited to the default constructor
  - workaround: factory pattern

Using Configuration Files (I)

Code

```
RemotingConfiguration.Configure(„server.exe.config“);
```

Config

```
<configuration>
   <system.runtime.remoting>
         <application>
               <channels>
                     <channel ref=„http“ port=„1234“ />
               </channels>
               <service>
                     <wellknown mode=„Singleton“
                           type=„Server.CurstomerManager, Server“
                           objectUri=„CustomerManager.soap“ />
               </service>
         </application>
   </system.runtime.remoting>
</configuration>
```

Same as:

```
HttpChannel channel = new HttpChannel(1234);
ChannelServices.RegisterChannel(channel);
```

Same as:

```
RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(CustomerManager), „CustomerManager.soap“,
      WellKnownObjectMode.SingleCall);
```

Using Configuration Files (II)

Advantages

- flexibility: change configuration without recompilation

Disadvantages

- checks are done at run-time instead of compile-time
- wrong config may be „correct“ (no exception), system will use local type instead of remote

Leases

distributed GC implementation:

- time-to-live counter for each object
  - initial lifetime per object
  - increment counter at every access
  - at time-out, collect object

- avoid keeping references from server to client or pinging the client (not always possible)

```
public override object InitializeLifetimeService() {
   ILease lease = (ILease)base.InitialiyeLifetimeService();
   if (lease.CurrentState == LeaseState.Initial) {
       lease.InitialLeaseTime = TimeSpan.FromMinutes(5);
       lease.RenewOnCallTime = TimeSpan.FromMinutes(2);
   }
   return lease;
}
```

return null to disable object collection

Limitations

Remoting has the following limitations:

Server-Activated Objects

- object configuration limited to the default constructor
  - Singleton can be configured using RemotingServices.Marshal
  - SingleCall requires different implementation classes

Client-Activated Objects

- class must be instantiated, no access over interface
- class hierarchy limitations
- use Factory Pattern
  - to get interface reference
  - to allow parametrization of the constructor

Furthermore...

- interface information is lost when passing an object reference to another machine

Deployment Options

Problem

object metadata must be visible on the client

Shared Implementation

- deploy the class dll on client and server
- bad design

Shared Interfaces

- deploy the interface dll on client and server
- good design
- implementation restriction when using multiple servers

Shared Base Class

- deploy the abstract class dll on client and server
- restricted to Activator.GetObject()

SoapSuds

- generate metadata with SoapSuds
- only default constructor supported
- does not work with ISerializable types (implementation not reflected)