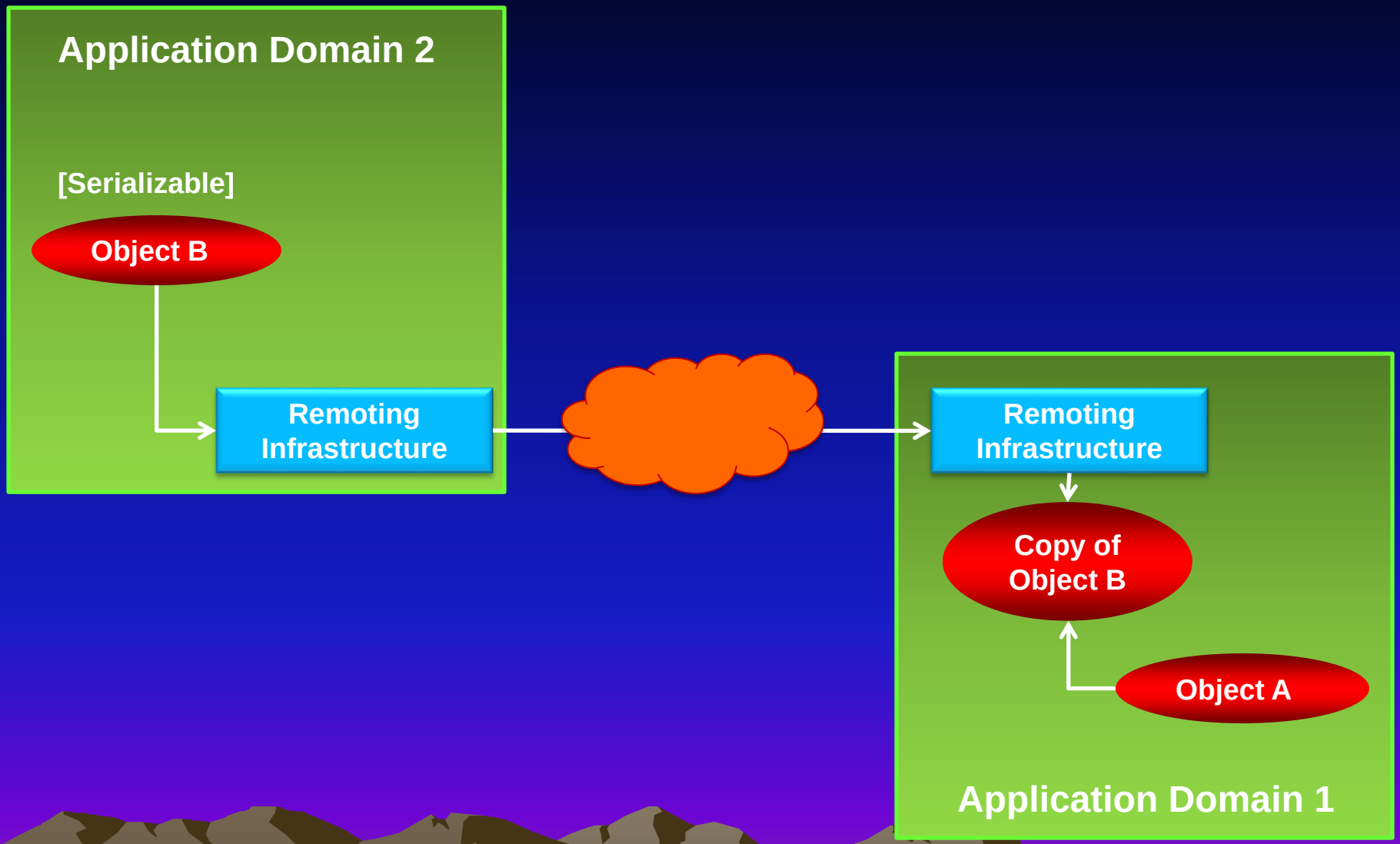
Distribution and Integration Technologies

.NET Remoting

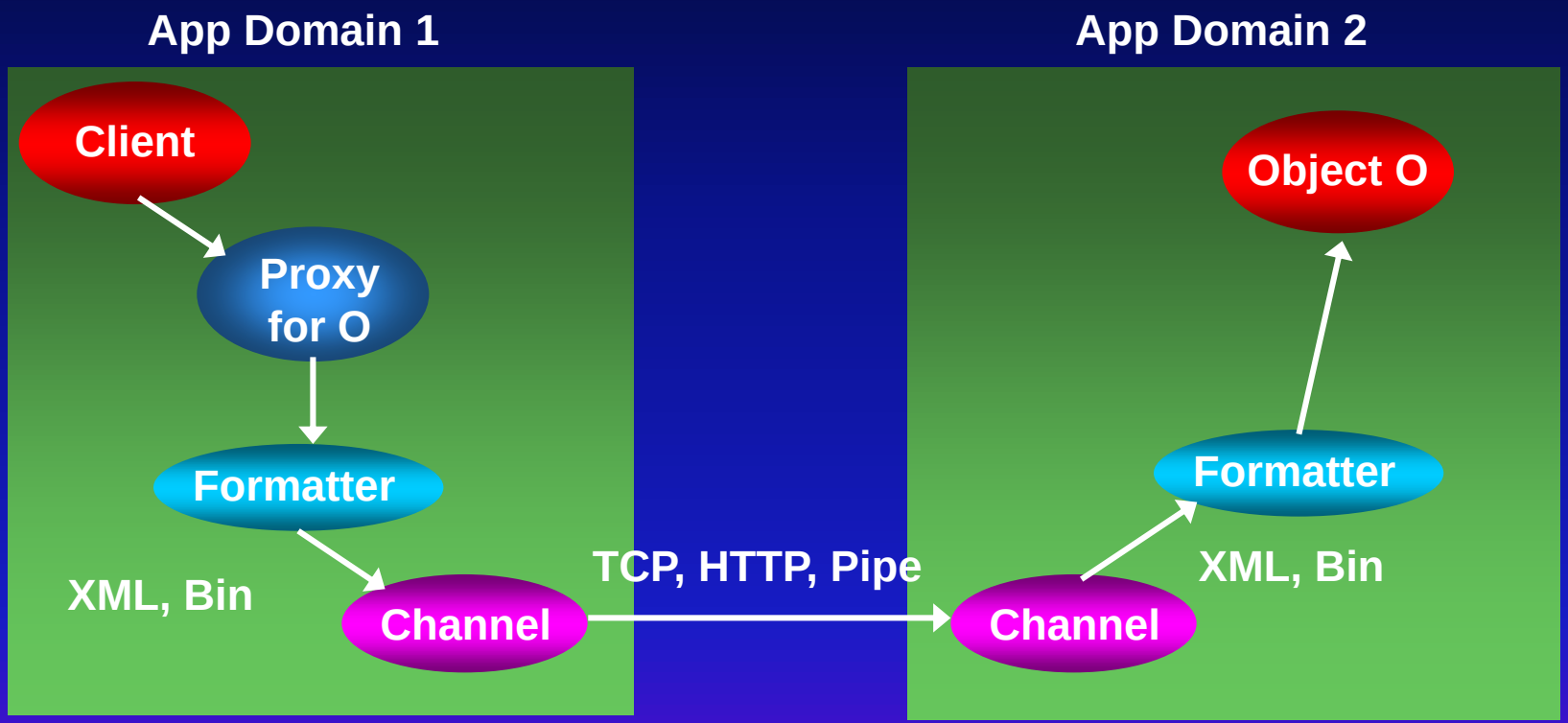
.NET Remoting

- Moving objects between *app. domains*
- Access objects in another *app. domain*
- Characteristics:
  - Transparent access without "plumbing" code
  - Automatic creation of *proxies* in the client
  - Selectable communication channels and encoding
    - Channel: TCP, HTTP, Pipe
    - Encoding: SOAP (XML), Binary
  - Controlled disposal using a *leasing* mechanism
  - Several remote activation models
    - Singe call, Singleton, Client activated

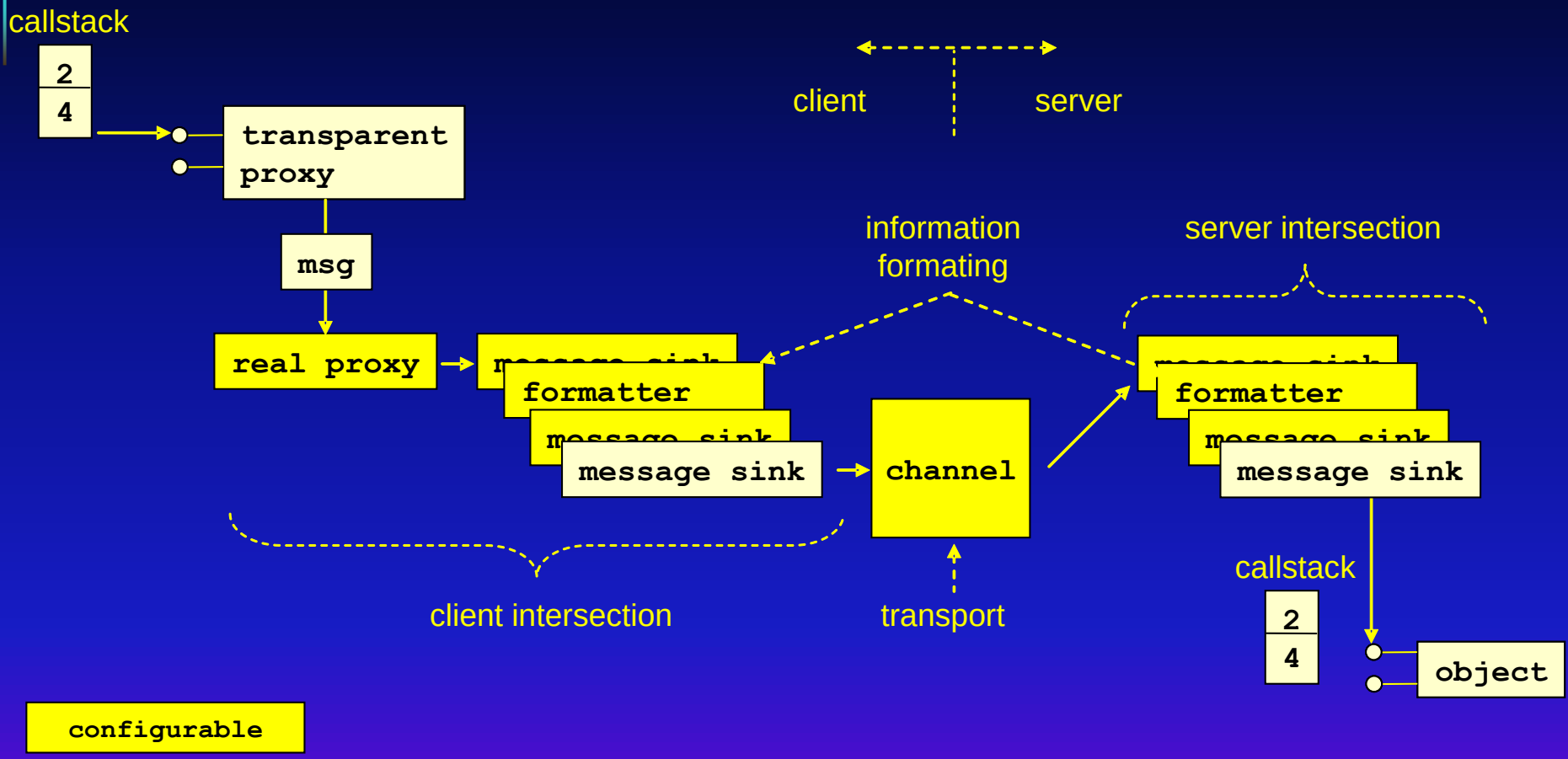
Moving objects (serialization)



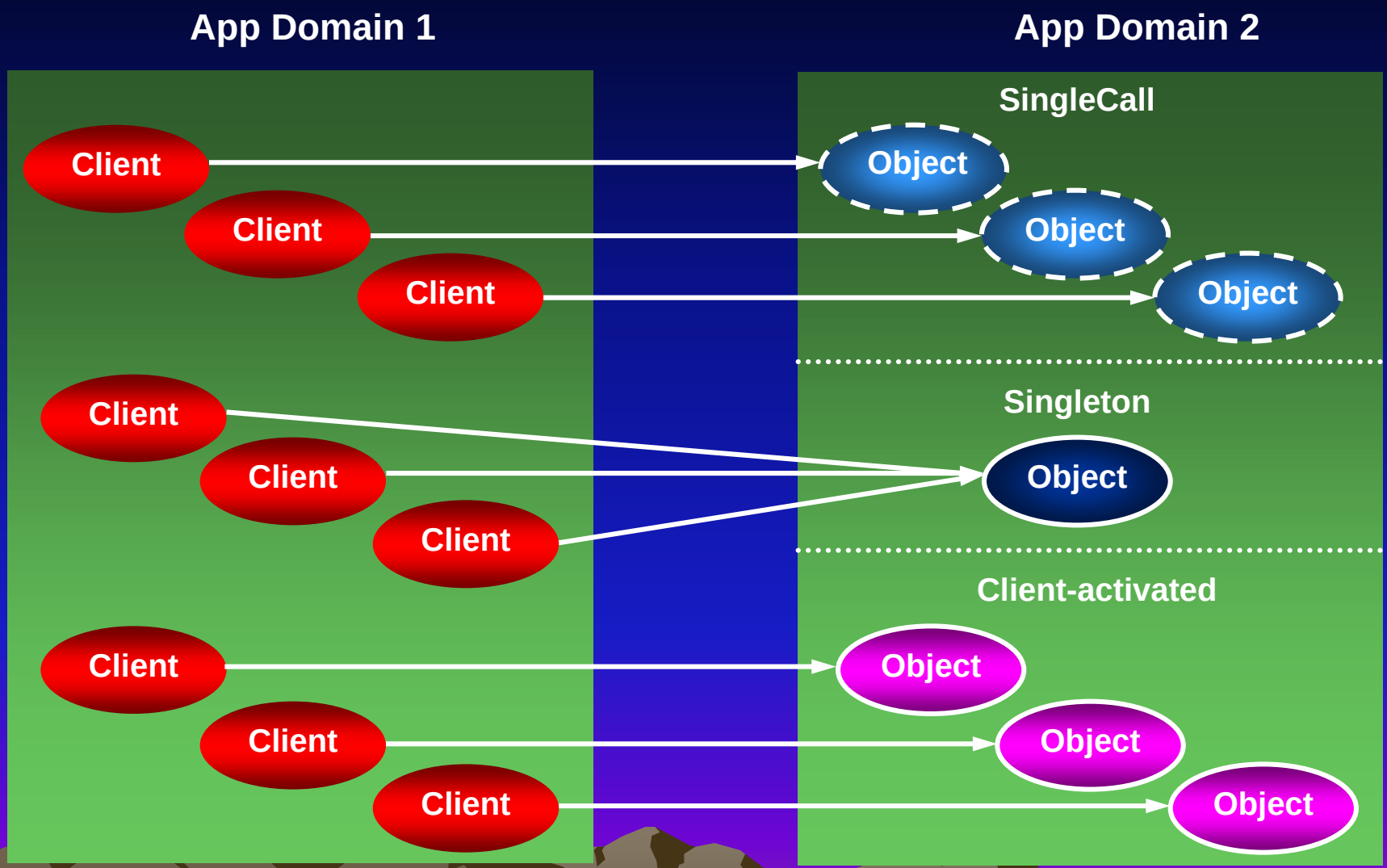
.NET Remoting - architecture



.NET Remoting – architecture (details)



.NET Remoting - activation



Programming

- Allowing the server process to use *remoting*
  - Register one or more channels (TCP, HTTP, Pipe)
    - Can be static (configuration file) or dynamic (server code)
- Allowing a class to be used with *remoting* (server)
  - Supply the potential to be accessed remotelly
  - Register the type (class) for remote access
    - Can be static (configuration file) or dynamic (code)
  - Register the activation mode
    - Singleton, SingleCall, Client Activated
    - Can be static (configuration file) or dynamic (code)
  - If needed, configure the activation time of life (*leasing*)
    - Can be based in the application (server) or in the object
- Remote object activation (in the client)
  - Explicit (calling *Connect* or *CreateInstance*) or
  - Implicit (new operator)

Channel registration

```
class MathServer {
  static void Main() {
    RemotingConfiguration.Configure("MathServer.exe.config", false);
    ...
  }
}
```

Static

```
<!-- MathServer.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="mathsrv">
      <channels>
        <channel ref="tcp" port="999"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Configuration file

Dynamic

```
class MathServer {
  static void Main() {
    TcpChannel chan = new TcpChannel(999);
    ChannelServices.RegisterChannel(chan, false);
    ...
  }
}
```

Remotly activatable class

- Must derive directly or indirectly from MarshalByRefObject
- The server must proactively register the remotely accessible types
  - Specify that this type has permission to be remotely accessed
  - What is the activation semantics
  - What is the access address through an URI

```
class Calc : MarshalByRefObject, ICalc
{
  public int Add( int a, int b )
  {
    return(a + b);
  }
}
```

Static registration of remote classes

```
class Calc : MarshalByRefObject { ... }
class Accumulator : MarshalByRefObject { ... }

class MathServer {
  static void Main() {
    RemotingConfiguration.Configure("MathServer.exe.config", false);
    ...
  }
}
```

```
<!-- MathServer.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="mathsrv">
      <service>
        <wellknown type="Calc, MathServer" mode="Singleton"
                   objectUri="calcEndpoint" />
        <activated type="Accumulator, MathServer" />
      </service>
      <channels>
        <channel ref="tcp" port="999"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Programmed registration of remote classes

```
class Calc : MarshalByRefObject { ... }
class Accumulator : MarshalByRefObject { ... }

class MathServer {
  static void Main() {
    ...
    RemotingConfiguration.ApplicationName = "mathsrv";

    RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(Calc), "calcEndpoint",
      WellKnownObjectMode.Singleton
    );

    RemotingConfiguration.RegisterActivatedServiceType(
      typeof(Accumulator)
    );
    ...
  }
}
```

Explicit activation in the client

```
class MathClient {
  static void Main() {
    ...
    ICalc c = (ICalc) RemotingServices.Connect(
          typeof(ICalc),
          "tcp://localhost:999/mathsrv/calcEndpoint"
      );

    int sum = c.Add(2, 4);

    object[] attrs = {new UrlAttribute("tcp://localhost:999/mathsrv")};
    ObjectHandle oh =
      Activator.CreateInstance("MathServer", "Accumulator", attrs);
    IAccumulator a = (IAccumulator)oh.Unwrap();

    a.Add(2);
    a.Add(4);
    sum = a.Sum;
  }
}
```

It is only needed to know, at the client, the interfaces: ICalc and IAccumulator

Activation using the *new* operator

```
class MathClient {
  static void Main() {
    RemotingConfiguration.Configure("MathClient.exe.config");
    Calc c = new Calc();
    int sum = c.Add(2, 4);
    Accumulator a = new Accumulator();
    a.Add(2);
    a.Add(4);
    sum = c.Sum;
}
```

```
<!-- MathClient.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="mathclient">
      <client url="tcp://localhost:999/mathsrv" >
        <activated type="Accumulator, MathServer" />

        <wellknown type="Calc, MathServer"
                   url="tcp://localhost:999/mathsrv/calcEndpoint" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Remoting – Simple Demo

```
using System;
using System.Runtime.Remoting;

class Client
{
  static void Main(string[] args)
  {
    int v = 7;
    RemotingConfiguration.Configure(
                                        "Client.exe.config");
    RemObj obj = new RemObj();
    Console.WriteLine(obj.Hello());
    Console.ReadLine();
    Console.WriteLine(obj.Modify(ref v));
    Console.WriteLine("Client after: {0}", v);
  }
}
```

```
class Server  {
  static void Main( )  {
    RemotingConfiguration.Configure(
                                      "Server.exe.config");
    Console.WriteLine("Press return to exit");
    Console.ReadLine();
  }
}
```

```
using System;

public class RemObj: MarshalByRefObject
{
  public RemObj()
  {
    Console.WriteLine("Constructor called");
  }

  public string Hello()
  {
    Console.WriteLine("Hello called");
    return "Hello .NET client!";
  }

  public string Modify(ref int val)
  {
    string s = String.Format("Received: {0}", val);
    Console.WriteLine("Modify called");
    Console.WriteLine(s);
    val += 10;
    return s;
  }
}
```

Remoting –Simple Demo

Server configuration

```
<configuration>
  <system.runtime.remoting>
    <application name="Server">
      <service>
        <wellknown
          mode="SingleCall"
          type="RemObj, RemObj"
          objectUri="RemObj" />
      </service>
      <channels>
        <channel ref="tcp server" port="9000" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Client configuration

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
          type="RemObj, RemObj"
          url="tcp://localhost:9000/Server/RemObj"
        />
      </client>
      <channels>
        <channel ref="tcp client" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Bidirectional remoting

Remote object – Singleton

```
delegate NewSegHandler
delegate LastSegHandler
delegate ClearStrokes

class Paper
    private Strokes[ ]
    private Current[ ]

    event NewSegment
    event LastSegment
    event ClearAll

    GetStrokes( )
    GetCurrentStroke( )
    DrawSegment ( )   →   NewSegment
    EndStroke( )      →   LastSegment
    Clear( )          →   ClearAll
```

Client – NetDraw (GUI – Windows.Forms)

class NetDraw

- Creates the remote object
- Obtains Strokes[ ] and Current [ ]
- Register the event handlers for the events NewSegment, LastSegmen and ClearAll
- Tracks mouse movements and keyboard to draw and delete strokes and invokes, when appropriate, DrawSegment( ), EndStroke( ) and Clear( )
- In the event handlers the appropriate actions are executed

Demo NetDraw

Leasing specification for the application

```
<!-- server remoting configuration file -->
<configuration>
  <system.runtime.remoting>
    <application name="mathsrv">
      ...
      <!-- values shown below are the defaults -->
      <!-- unit-of-time suffixes: MS,S,M,H,D -->
      <lifetime
        leaseManagerPollTime = "10S"
        leaseTimeout = "5M"
        renewOnCallTime = "2M"
        sponsorshipTimeout = "2M" />
    </application>
  </system.runtime.remoting>
</configuration>
```

in the server configuration file

in the server code

```
using System;
using System.Runtime.Remoting.Lifetime;

class Server {
  static void Main() {
    ...
    LifetimeServices.LeaseManagerPollTime = Timespan.FromSeconds(10);
    LifetimeServices.LeaseTime = TimeSpan.FromMinutes(5);
    LifetimeServices.RenewOnCallTime = TimeSpan.FromMinutes(2);
    LifetimeServices.SponsorshipTimeout = TimeSpan.FromMinutes(2);
  }
}
```

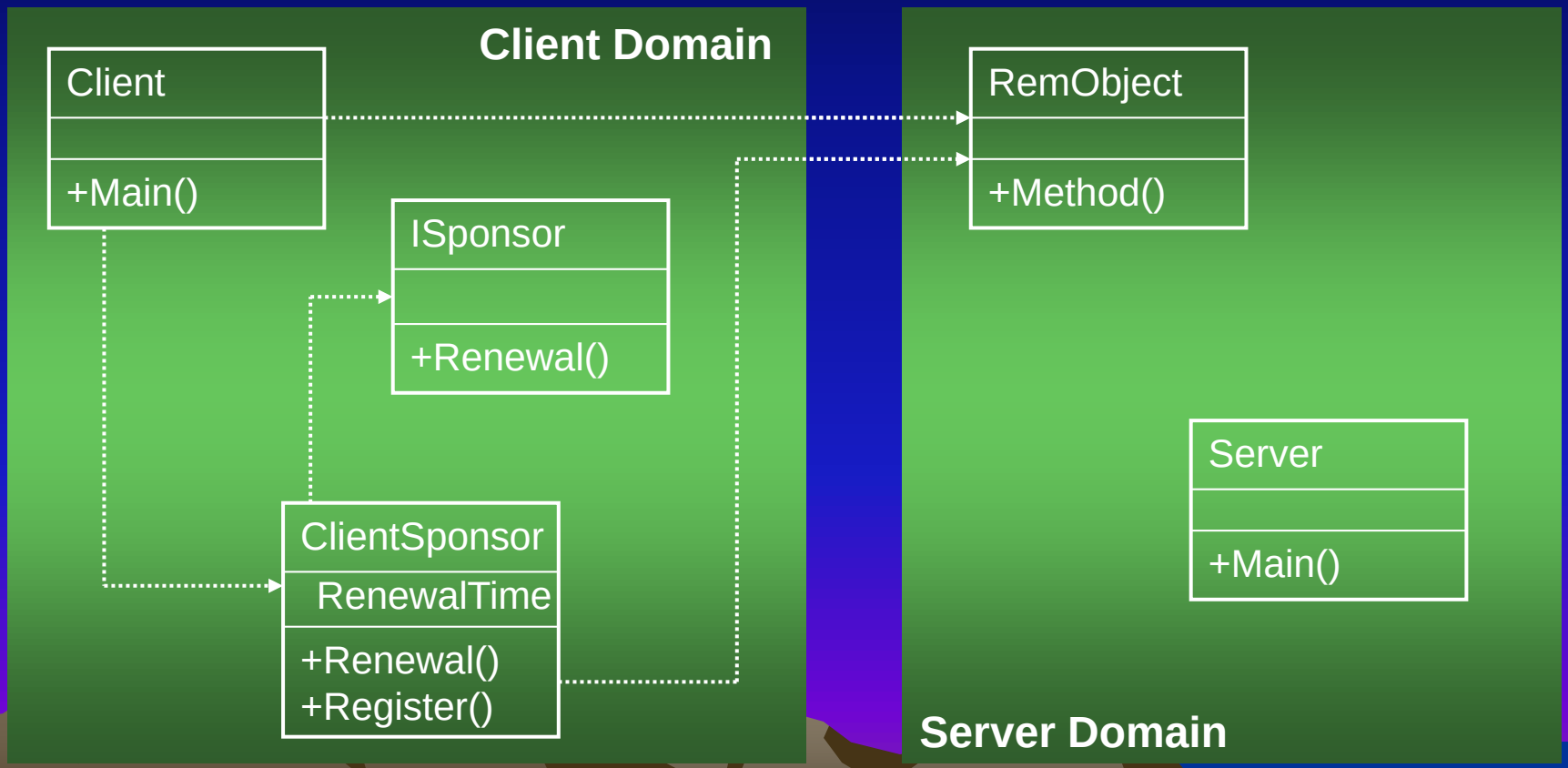
Remote object time of life

```
using System;
using System.Runtime.Remoting.Lifetime;

class Calc : MarshalByRefObject
{
  public override object InitializeLifetimeService()
  {
    ILease myLease = (ILease)base.InitializeLifetimeService();

    if( myLease.CurrentState == LeaseState.Initial )
    {
      myLease.InitialLeaseTime = TimeSpan.FromMinutes(15);
      myLease.RenewOnCallTime = TimeSpan.FromMinutes(5);
    }
    return(myLease);

    // Alternatively, may return null to request an
    // infinite lease on life.
  }
}
```

in the remote object code

Leasing and *Sponsors*

- **When an object is created its time of life is fixated (default 5 min.)**
- **Each access garantees a new minimum time of life (default 2 min.)**
- **When the time of life ends and before the object removal, it is verified if one or more *sponsors* were registered, and if so they are called.**



Sponsor use in the client

```
class Client
{
  static void Main()
  {
    RemotingConfiguration.Configure("Client.exe.config");
    RemObj obj = new RemObj();

    ClientSponsor sponsor = new ClientSponsor();
    sponsor.RenewalTime = TimeSpan.FromMinutes(5);
    sponsor.Register(obj);

    . . .
    obj.method( );
    . . .

  }
}
```

New lease time granted by this sponsor

For this remote object (usually client activated, but can also be a singleton)

Nuisances in .NET Remoting

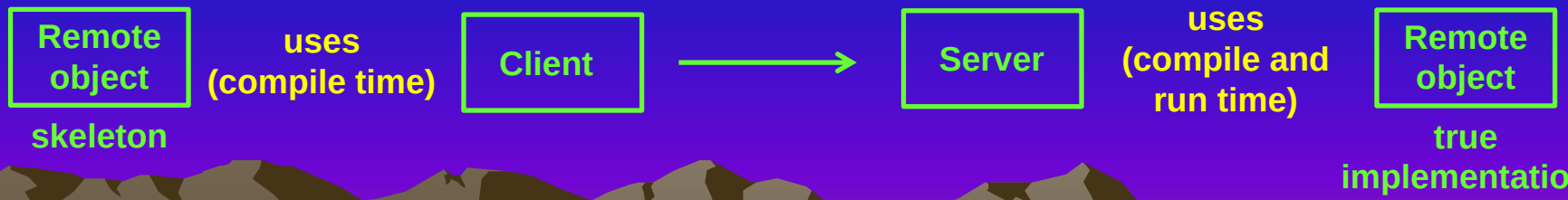
When a remote reference is instantiated in a client, it can use a configuration file and the new operator. The configuration file is preferred, avoiding recompilations whenever we change the server location.

But the new operator demands that the client has access to a remote object definition in compile time. Also the alternative Activator.CreateInstance() method for client activated objects demands the same.

One way of complying with the compiler needs, is to have a copy of the remote object implementation assembly in the client, but that is inconvenient and breaks the rule of distributed computing of maintaining separated development between client and remote code, except for an interface (or contract).

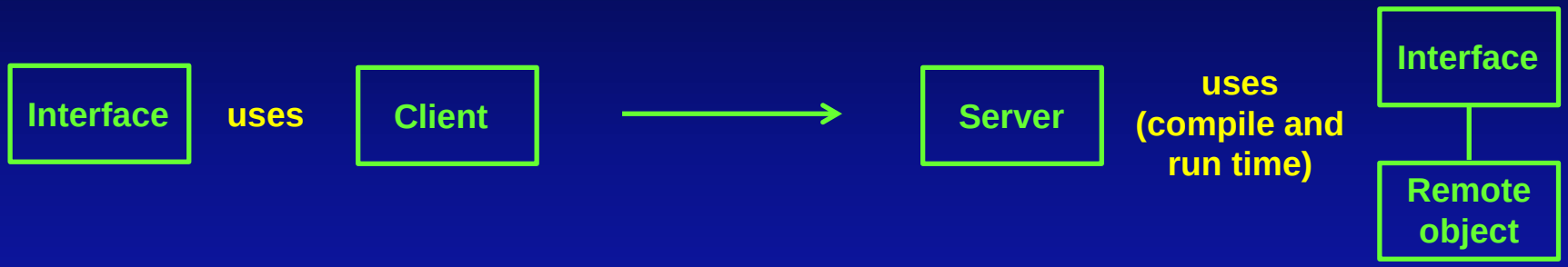
How to solve:

1) Build two assemblies of the remote objects with the same name. The one destined to the client only has a skeleton of the public methods available on the remote object.
   Still has the inconvenient of being easy to have dephased versions of the two assemblies.



Interface based remote objects

2) The remote object implements an interface. The client connects to the remote object knowing only the interface. It's only possible for well known objects (singe call or singletons) using RemotingServices.Connect().



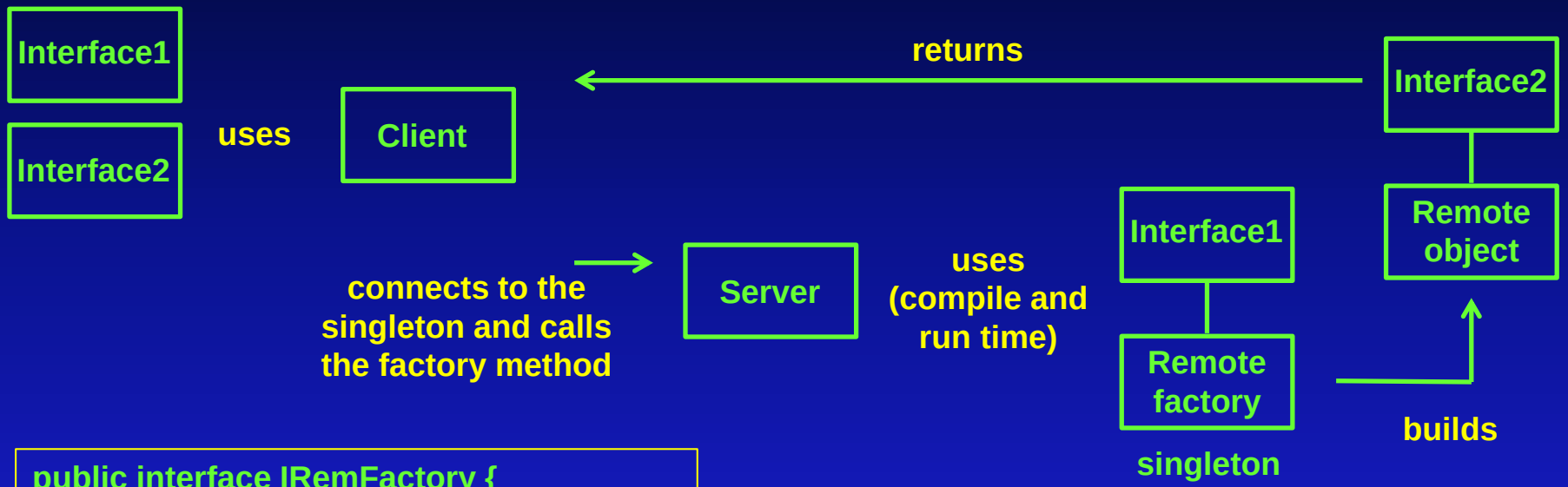
But this can preclude the use of the configuration file (and the new operator). To continue using the config file we need an our own implementation of a new-like method.

```
class RNew {
  private static Hashtable types = null;

  private static void InitTypeTable() {
    types = new Hashtable();
    foreach (WellKnownClientTypeEntry entry in
        RemotingConfiguration.GetRegisteredWellKnownClien
      types.Add(entry.ObjectType, entry);
  }
```

```
  public static object New(Type type) {
    if (types == null)
      InitTypeTable();
    WellKnownClientTypeEntry entry =
                (WellKnownClientTypeEntry) types[type];
    if (entry == null)
      throw new RemotingException("Type not found!");
    return RemotingServices.Connect(type, entry.ObjectUrl);
  }
}
```

Factory for client activated objects

3) The previous pattern can only be used for well known remote objects (single call or singleton). The direct activation of a client activated object requires always an object (not an interface). But, fortunately, a well known object can return a client activated one, even using only an interface.



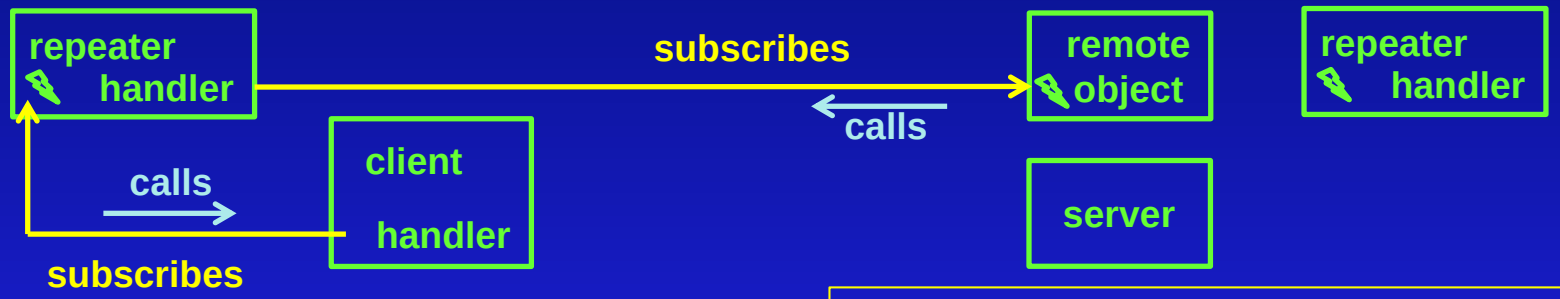

```
public interface IRemFactory {
    IRemObj BuildNew();
}

class MyRemFactory :
        MarshalByRefObject, IRemFactory {
    public IRemObj BuildNew() {
        return new MyRemObj();
    }
}
```

```
public interface IRemObj {
    void DoSomeWork();
}

class MyRemObj :
        MarshalByRefObject, IRemObj {
    public void DoSomeWork() {
        …;
    }
}
```

Remote events and handlers

4) When a client subscribes a remote event the server calls a client handler when the event is fired. For compiling and running the server it should have access to the handler implementing assembly, which is very inconvenient.

In order to avoid this inconvenience a repeater class should be defined and present in both client and server. This repeater should define an event with the same signature of the remote object's event and also a handler subscribing the remote object event and firing the repeater event. The client subscribes the repeater event. Whenever the remote event is fired the repeater handler runs and fires its own event, calling the client handle. This satisfies the compiler demands for both the client and server, having only to share the repeated event.




```
public class EventRepeater : MarshalByRefObject {
  public event RemDelegate repEvent;

  public EventRepeater(IRemote rem) {
     rem.remEvent += Repeater;
  }
```

```
public override object InitializeLifetimeService() {
   return null;   // infinite lease time
}

public void Repeater(…) {
  if (repEvent != null)  repEvent(…);
}
}
```