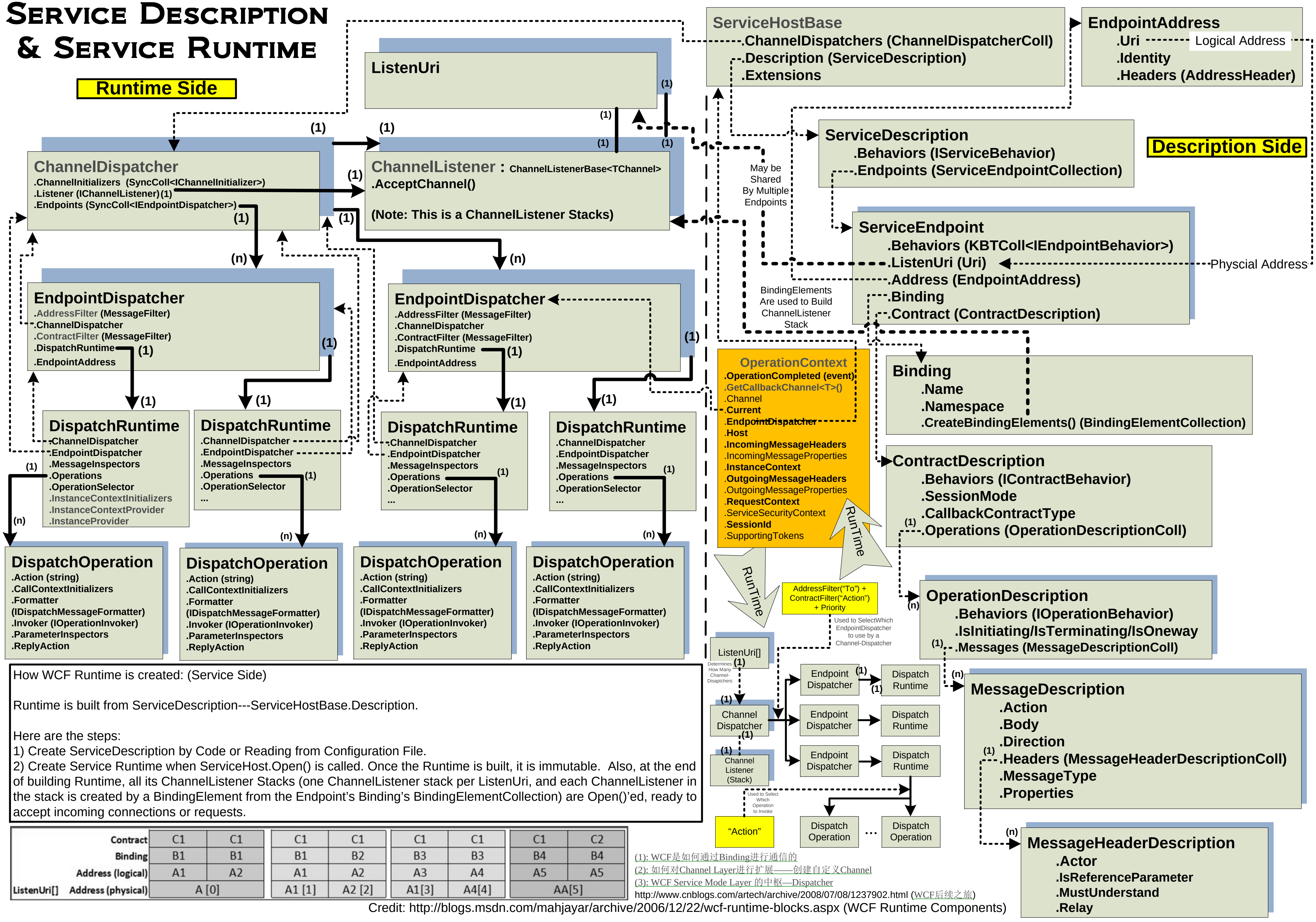
Service Description & Service Runtime



Runtime Side

Description Side

ListenUri

ServiceHostBase
.ChannelDispatchers (ChannelDispatcherColl)
.Description (ServiceDescription)
.Extensions

EndpointAddress
.Uri — Logical Address
.Identity
.Headers (AddressHeader)

ServiceDescription
.Behaviors (IServiceBehavior)
.Endpoints (ServiceEndpointCollection)

ServiceEndpoint
.Behaviors (KBTColl<IEndpointBehavior>)
.ListenUri (Uri) — Physcial Address
.Address (EndpointAddress)
.Binding
.Contract (ContractDescription)

May be Shared By Multiple Endpoints

BindingElements Are used to Build ChannelListener Stack

ChannelDispatcher
.ChannelInitializers (SyncColl<IChannelInitializer>)
.Listener (IChannelListener) (1)
.Endpoints (SyncColl<IEndpointDispatcher>)

ChannelListener : ChannelListenerBase<TChannel>
.AcceptChannel()
(Note: This is a ChannelListener Stacks)

EndpointDispatcher
.AddressFilter (MessageFilter)
.ChannelDispatcher
.ContractFilter (MessageFilter)
.DispatchRuntime
.EndpointAddress

EndpointDispatcher
.AddressFilter (MessageFilter)
.ChannelDispatcher
.ContractFilter (MessageFilter)
.DispatchRuntime
.EndpointAddress

DispatchRuntime
.ChannelDispatcher
.EndpointDispatcher
.MessageInspectors
.Operations
.OperationSelector
.InstanceContextInitializers
.InstanceContextProvider
.InstanceProvider

DispatchRuntime
.ChannelDispatcher
.EndpointDispatcher
.MessageInspectors
.Operations
.OperationSelector
...

DispatchRuntime
.ChannelDispatcher
.EndpointDispatcher
.MessageInspectors
.Operations
.OperationSelector
...

DispatchRuntime
.ChannelDispatcher
.EndpointDispatcher
.MessageInspectors
.Operations
.OperationSelector
...

DispatchOperation
.Action (string)
.CallContextInitializers
.Formatter
(IDispatchMessageFormatter)
.Invoker (IOperationInvoker)
.ParameterInspectors
.ReplyAction

DispatchOperation
.Action (string)
.CallContextInitializers
.Formatter
(IDispatchMessageFormatter)
.Invoker (IOperationInvoker)
.ParameterInspectors
.ReplyAction

DispatchOperation
.Action (string)
.CallContextInitializers
.Formatter
(IDispatchMessageFormatter)
.Invoker (IOperationInvoker)
.ParameterInspectors
.ReplyAction

DispatchOperation
.Action (string)
.CallContextInitializers
.Formatter
(IDispatchMessageFormatter)
.Invoker (IOperationInvoker)
.ParameterInspectors
.ReplyAction

OperationContext
.OperationCompleted (event)
.GetCallbackChannel<T>()
.Channel
.Current
.EndpointDispatcher
.Host
.IncomingMessageHeaders
.IncomingMessageProperties
.InstanceContext
.OutgoingMessageHeaders
.OutgoingMessageProperties
.RequestContext
.ServiceSecurityContext
.SessionId
.SupportingTokens

RunTime

RunTime

Binding
.Name
.Namespace
.CreateBindingElements() (BindingElementCollection)

ContractDescription
.Behaviors (IContractBehavior)
.SessionMode
.CallbackContractType
.Operations (OperationDescriptionColl)

OperationDescription
.Behaviors (IOperationBehavior)
.IsInitiating/IsTerminating/IsOneway
.Messages (MessageDescriptionColl)

MessageDescription
.Action
.Body
.Direction
.Headers (MessageHeaderDescriptionColl)
.MessageType
.Properties

MessageHeaderDescription
.Actor
.IsReferenceParameter
.MustUnderstand
.Relay

AddressFilter("To") + ContractFilter("Action") + Priority

Used to SelectWhich EndpointDispatcher to use by a Channel-Dispatcher

ListenUri[]

Determines How Many Channel-Disaptchers

Channel Dispatcher

Channel Listener (Stack)

Endpoint Dispatcher → Dispatch Runtime

Endpoint Dispatcher → Dispatch Runtime

Endpoint Dispatcher → Dispatch Runtime

Used to Select Which Operation to Invoke

"Action"

Dispatch Operation ... Dispatch Operation

How WCF Runtime is created: (Service Side)

Runtime is built from ServiceDescription---ServiceHostBase.Description.

Here are the steps:
1) Create ServiceDescription by Code or Reading from Configuration File.
2) Create Service Runtime when ServiceHost.Open() is called. Once the Runtime is built, it is immutable. Also, at the end of building Runtime, all its ChannelListener Stacks (one ChannelListener stack per ListenUri, and each ChannelListener in the stack is created by a BindingElement from the Endpoint's Binding's BindingElementCollection) are Open()'ed, ready to accept incoming connections or requests.

	Contract	C1	C1	C1	C1	C1	C1	C1	C2
	Binding	B1	B1	B1	B2	B3	B3	B4	B4
	Address (logical)	A1	A2	A1	A2	A3	A4	A5	A5
ListenUri[]	Address (physical)	A [0]		A1 [1]	A2 [2]	A1[3]	A4[4]	AA[5]	

(1): WCF是如何通过Binding进行通信的
(2): 如何对Channel Layer进行扩展——创建自定义Channel
(3): WCF Service Mode Layer 的中枢—Dispatcher
http://www.cnblogs.com/artech/archive/2008/07/08/1237902.html (WCF后续之旅)

Credit: http://blogs.msdn.com/mahjayar/archive/2006/12/22/wcf-runtime-blocks.aspx (WCF Runtime Components)

Haha, the WCF Basics:

A) Service & Client
B) Client Callback
C) New Behavior

A

Service Side

Step 1) Define a Service Contract (Interface)

```
using System.ServiceModel;
[ServiceContract]
public interface IMath {  //a WCF contract defined using an interface
  [OperationContract] int Add(int x, int y);
}
```

Step 2) Implement the Service

```
public class MathService : IMath {
  public int Add(int x, int y) { return x + y; }
}
```

Step 3) Host the Service

```
public class WCFServiceApp {
  // (1) Create ServiceHost and add Endpoints Imperatively
  public void DefineEndpointImperatively() {

    ServiceHost sh = new ServiceHost(typeof(MathService));

    sh.AddServiceEndpoint(
      typeof(IMath), //contract type
      new WSHttpBinding(), //one of the built-in bindings
      "http://localhost/MathService/Ep1"); //endpoint's address

    sh.Open(); // create and open the service runtime
  }

  // (2) Create ServiceHost & add Endpoints Declaratively via Config File
  public void DefineEndpointDeclaratively() {

    ServiceHost sh = new ServiceHost (typeof(MathService));
    sh.Open(); // create and open the service runtime
  }
}
```

Create ServiceHost Imperatively

Create ServiceHost Declaratively

Configuration File (App.config)

```
<configuration> <!-- configuration file used by above Declarative code -->
<system.serviceModel>
  <services>

    <service type="MathService"> <!-- refers to service type -->
    <endpoint
      address="http://localhost/MathService/Ep1"
      binding="wsHttpBinding"
      contract="IMath"/>
    </service>
  </services>
….
```

Client Side (Service Consumer)

```
using System.ServiceModel;
public interface IMath { // this contract is generated by svcutil.exe from the service's metadata
  [OperationContract] public int Add(int x, int y) { return x + y; }
}

public class MathProxy : IMath {
  … // generated by svcutil.exe from service's metadata, & the generated config is not shown
}

public class WCFClientApp {
  // (1) Use Generated Proxy
  public void SendMessageToEndpointUsingProxy() {
    //this uses a proxy class that was created by svcutil.exe from the service's metadata
    MathProxy proxy = new MathProxy("ServiceEndpoint"); int result = proxy.Add(35, 7);
}
  // (2) Use ChannelFactory<TChannel>
  public void SendMessageToEndpointUsingChannel() {

    ChannelFactory<IMath> factory=new ChannelFactory<IMath>(new WSHttpBinding(),
      new EndpointAddress("http://localhost/MathService/Ep1"));

    IMath channel=factory.CreateChannel();

    // Another way to use ChannelFactory
    // IMath channel = ChannelFactory<IMath>.CreateChannel(new WSHttpBinding(),
    //                                   new EndpointAddress("http://localhost/MathService/Ep1"));
    int result=channel.Add(35,7); factory.Close();
  }
}
```

Nowadays, Proxy is named as <ServiceType>Client()

1) Using Service Proxy **One Way**

2) Using ChannelFactory<Ichannel> **2nd Way**

Adding Service Endpoints in Code Here. To Use Endpoints in Config File: ...new ChannelFactory<Imath>("ServiceEndpoint");

Note: The difference between using Service Proxy and ChannelFactory<TChannel> is that Service Proxy is generated by svcutil.exe. But, if one have access to the interface that defines the Service Contract, one can use the ChannelFactory<IServiceContract> to create a channel factory which can be used to create a channel (factory.CreateChannel()) to communicate with the Service hosted by a Service Host.

```
<system.serviceModel>
  <client>
    <endpoint name="ServiceEndpoint"
      address="http://localhost/MathService/Ep1"
      binding="wsHttpBinding"
      contract="IMath"/>
  </client>
</system.serviceModel>
```

Client Callback Implementation

```
// Step 1) Implement the Callback on Client

public class ClientCallBack:IClientCallback {
      public void Calculate(answer) {
            …
      }
}
```

```
// Step 2) Get an instance of InstanceContext
InstanceContext ic =

      new InstanceContext(new ClientCallback());
```

```
// (3A) Get a Client Proxy to make calls to Service
DuplexServiceClient client =
      new DupliexServiceClient(ic, "ServiceEpt");
client.Add(100);
```

One Way

```
// (3B) Using DuplexChannelFactory<IChannel>
var def =
new DuplexChannelFactory<IServerContract>(ic,
                              "ServiceEnpt");
IServerContract ch = def.CreateChannel();
ch.Add(100);
```

2nd Way

```
// (3B') Adding Endpoint usng static CreateChannel
var def =
new DuplexChannelFactory<IServerContract>(ic);

var ea = new EndpointAddress("http://….");
var bb = new BasicHttpBinding();

IClientCallback ch = dcf.CreateChannel(ea, bb);
ch.Add(100);
```

2nd Way

B

Service Side (To Use Callback Channel):

```
      OperationContext.Current.GetCallbackChannel<IClientCallback>();
```

http://idunno.org/archive/2008/05/29/wcf-callbacks-a-beginners-guide.aspx

C

Adding New Behavior

```
[AttributeUsageAttribute(AttributeTargets.Class, AllowMultiple=false, Inherited=false)]
public class InspectorBehavior : Attribute, IServiceBehavior {
  public void ApplyBehavior(ServiceDescription description, Collection<DispatchBehavior> behaviors) {
    Console.WriteLine("-------- Endpoints ---------");
    foreach (ServiceEndpoint endpoint in description.Endpoints) {
      Console.WriteLine("--> Endpoint");
      Console.WriteLine("Address: {0}", endpoint.Address);
      Console.WriteLine("Binding: {0}", endpoint.Binding.GetType().Name);
      Console.WriteLine("Contract: {0}", endpoint.Contract.ContractType.Name); Console.WriteLine();
    }
    Console.WriteLine("-------- Service Behaviors --------");
    foreach (IServiceBehavior behavior in description.Behaviors) {
      Console.WriteLine("--> Behavior");
      Console.WriteLine("Behavior: {0}", behavior.GetType().Name); Console.WriteLine();
    }
  }
}
```

2nd Way Add Behavior using Attribute

```
[InspectorBehavior]
public class MathService : IMath{
  public int Add(int x, int y) { return x + y; }
}
```

One Way Add Behavior Imperatively

```
ServiceHost sh = new ServiceHost(typeof(MathService));
sh.AddServiceEndpoint( typeof(IMath),  new WSHttpBinding(),"http://localhost/MathService/Ep1");

InspectorBehavior behavior = new InspectorBehavior(); //Add the behavior imperatively
sh.Description.Behaviors.Add(behavior);

sh.Open();
```

Credit: http://www.yassers.com/content/soa/WCFArchOverview.aspx

Gotchas in WCF Metadata Publishing & Others!

WCF is great; however, it has many strange Gotchas! Publishing Metadata is one:
1) (Not WCF related). On Vista, MUST do this as administrator: netsh http add urlacl url=http://+:<port>/+ user=kenny
eg: netsh http add urlacl url=http://+:8088/+ user=kenny
this will open up all the URL (with specific port) on localhost; so that you won't need to do this for each URL (for given port number)

eg: http://localhost:8088/PATH2/MEX (ok) http://localhost:8088/PATH3/MEX (ok) http://localhost:8099/PATH2/MEX (not ok; need to do another netsh for port 8099) NOTE: Same applied to baseAddresses!

2) Turn off the firewall!

3) Now, Metadata Publishing is built-in! There are two ways to get the Metadata.

A) **Basic**: Using HTTP-Get style. eg: svcutil /t:metadata http://localhost:8088/Service or http://localhost:8088/Service?wsdl on a browser or just http://localhost:8088/Service on a browser But, this is not enabled by default. Needs to add <serviceMetadata> to the serviceBehaviors!

eg:

```
    <service name=ServiceNamespace.ServiceName behaviorConfiguration="serviceBehavior">
        ... // define service endpoints here
    </service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior name="serviceBehavior"> // matching the about behaviorConfiguration="serviceBehavior"
            <serviceMetadata httpGetEnabled="true" httpGetUrl="" />
        </behavior>
    </serviceBehaviors>
</behaviors>
```

app.config MUST use NS.Name scheme!!!!

Must Specify, event it is empty relative string!

B) **Advance**: Using a dedicated Metadata Exchange (MEX) Endpoint. This endpoint needs to be created just like regular service endpoint, which has ABC's, except the contract is always "IMetadataExchange", and a set of predefined binding just for Metadata Exchange. Note: the binding can be custom too, just like bindings for regular service. In addition, it requires to have a <serviceMetadata/> (an empty one is fine) behavior set too. If there is no <serviceMetadata/>, the serviceHost won't start!; it will throw an exception for you.

eg:

```
    <service ... behaviorConfiguration="serviceBehavior">
        ... // other service endpoints
        <endpoint address="Mex" binding="mexHttpBinding" contract="IMetadataExchange"/> // this is a relative address which requires a corresponding
        base address
        <host>
            <baseAddresses>
                <add baseAddress="http://localhost:8088/Service"/>
            </baseAddresses>
        </host>
    </service>
```

Always this Interfaces

relative to this base addr

BaseAddress is Tricky; DON'T do Naked BaseAddress, like http://localhost:8088; you need something like http://localhost:8088/XYZ

Gotcha: You must add both MEX endpoint and serviceMetadataBehavior together in either configuration file or in code; you CAN NOT add one in configuration and add the other in code. Credit: http://msdn.microsoft.com/en-us/library/ms730243.aspx

HEADS-UP: Unless using absolute addresses, for each protocol type's relative address, there must be a matching base address! This rule applies to both regular service endpoint and metadata endpoint.
eg: If we have a relative path using tcp protocol like in ... address="**kenny**" binding="netTcpBinding" contract=..., then we need to have a corresponding tcp base address! <add baseAddress="net.tcp://localhost:8088/Service"/>; if we have same (or different) relative using http protocol like in ... address="kenny" binding="basicHttpBinding" contract..., then we need another corresponding http base address! <add baseAddress="http://localhost:8089/Service"/>

Gotcha: DO NOT use "naked" Service's Namespace in Metadata Endpoint's URI!!!! Otherwise, MEX endpoint won't work!
Let say, we have a Service called "Service1" and its namespace is "**HelloWorldService**". We CAN NOT use the following base address for the MEX endpoint!
eg:

```
<endpoint address="Mex" binding="basicHttpBinding" contract="IService1"/> // endpoint (BTW, the address is case-insensitive!)
...
<add baseAddress="http://localhost:8088/HelloWorldService"> // endpoint's base address with "naked" service's namespace
...
<serviceMetadata/> // needed serviceMetadataBehavior
```

svcutil /t:metadata http://localhost:8088/HelloWorldService/Mex would give following error:

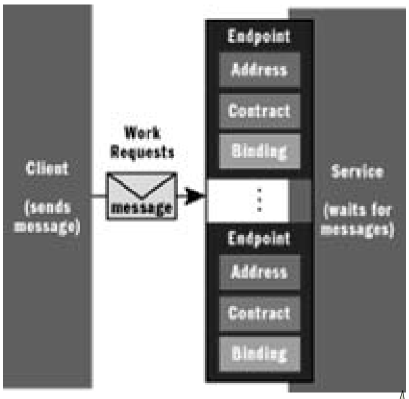
```
c:\Users\Kenny\Desktop\tmp>svcutil /t:metadata http://localhost:8088/HelloWorldService/Mex
Microsoft (R) Service Model Metadata Tool
…[deleted]
WS-Metadata Exchange Error
URI: http://localhost:8088/HelloWorldService/Mex
Metadata contains a reference that cannot be resolved: 'http://localhost:8088/HelloWorldService/Mex'.
The HTTP service located at http://localhost:8088/HelloWorldService/Mex is too busy.
The remote server returned an error: (503) Server Unavailable.

HTTP GET Error
URI: http://localhost:8088/HelloWorldService/Mex
There was an error downloading 'http://localhost:8088/HelloWorldService/Mex'.
The request failed with HTTP status 503: Service Unavailable.
If you would like more help, type "svcutil /?"
```

Now, change the base address to:
eg:

```
<endpoint address="Mex" binding="basicHttpBinding" contract="IService1"/> // endpoint
...
<add baseAddress="http://localhost:8088/HelloWorldService123"/> // endpoint's base address, w/o "naked" service's namespace
or
<add baseAddress="http://localhost:8088/HelloWorldService/123"/> // endpoint's base address, w/o "naked" service's namespace
...
<serviceMetadata/> // needed serviceMetadataBehavior
```

svcutil /t:metadata http://localhost:8088/HelloWorldService123/Mex would WORK!

```
c:\Users\Kenny\Desktop\tmp>svcutil /t:metadata http://localhost:8088/HelloWorldService123/Mex
Microsoft (R) Service Model Metadata Tool
…
Attempting to download metadata from 'http://localhost:8088/HelloWorldService123/Mex'
using WS-Metadata Exchange or DISCO.
Saving downloaded metadata files...
c:\Users\Kenny\Desktop\tmp\tempuri.org.wsdl
c:\Users\Kenny\Desktop\tmp\tempuri.org.xsd
c:\Users\Kenny\Desktop\tmp\schemas.microsoft.com.2003.10.Serialization.xsd
```



Element	Class or Interface
Endpoint	System.ServiceModel.ServiceEndpoint
Address	System.Uri
Binding	System.ServiceModel.Binding
Contract	Interfaces annotated with System.ServiceModel attributes

ABCs of WCF (Service + Service hosting)

SvcConfigEditor
SvcTraceViewer

Define a Contract

Default is: http://tempuri.org

Catch All

```
[ServiceContract(Namespace="http://example.org/echo/")]
public interface IEchoService
{
   [OperationContract(Action="urn:echo:string")]
   string Echo(string msg);
   [OperationContract(Action="*")]
   Message EchoMessage(Message msg);
}
```

Define DataContract for Complex Data Types

```
[DataContract(Namespace="http://example.org/person")]
public class Person
{
   [DataMember(Name="first", Order=0)] public string First;
   [DataMember(Name="last", Order=1)] public string Last;
   [DataMember(IsRequired=false,Order=2)]
         private string id;
   ...
}
```

Use DataContractSerializer

Use DataContract in ServiceContract

```
[ServiceContract(Namespace="http://example.org/echo/")]
public interface IEchoService
{
   ... // previous methods omitted

   [OperationContract] Person EchoPerson(Person p);
}
```

Force to use XmlSerializer

```
[ServiceContract]
[XmlSerializerFormat]
public interface IEchoService { ... }
```

Have Control on where to put Things in SOAP Msg

```
[MessageContract]
public class EchoPersonMessage
{
   [MessageBody]    public Person Person;
   [MessageHeader] public Authorization Authorization;
}
```

Use the Type Message defined by MessageContract

```
[ServiceContract]
public interface IEchoService
{
   ... // previous methods omitted

   [OperationContract]
   void EchoPerson(EchoPersonMessage msg);
}
```

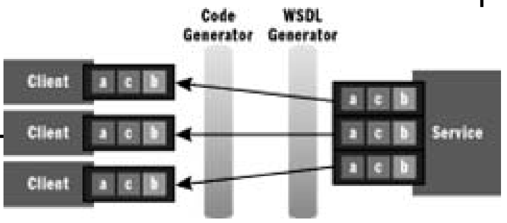
Sample Service (EchoService)

Host the Service

```
class Program
{
  static void Main(string[] args)
  {
    using (ServiceHost host = new ServiceHost(
      typeof(EchoService), new Uri("http://localhost:8080/echo"))) {
      // Add the service endpoints in Code
      host.AddServiceEndpoint(typeof(IEchoService),
         new BasicHttpBinding(), "svc");
      host.AddServiceEndpoint(typeof(IEchoService),
         new NetTcpBinding(), "net.tcp://localhost:8081/echo/svc");
      host.Open(); // Starts the ChannelListener Stacks

      Console.WriteLine(
         "{0} is open and has the following endpoints:\n",
         host.Description.ServiceType);

      int i=1;
      foreach (ServiceEndpoint end in host.Description.Endpoints){
         Console.WriteLine("Endpoint #{0}", i++);
         Console.WriteLine("Address: {0}",
                    end.Address.Uri.AbsoluteUri);
         Console.WriteLine("Binding: {0}", end.Binding.Name);
         Console.WriteLine("Contract: {0}\n", end.Contract.Name);
      }
      Console.WriteLine(
         "The following EndpointListeners are active:\n");
      foreach (EndpointListener l in host.EndpointListeners)
         Console.WriteLine(l.Listener.Uri.AbsoluteUri);

      // keep the process alive
      Console.ReadLine();
    }
  }
}
```

Configuration File

```
<configuration>
  <system.serviceModel>
    <services>
      <service type="ServiceLibrary.EchoService">
        <endpoint address="svc" binding="basicHttpBinding"
              contract="ServiceLibrary.IEchoService"/>
        <endpoint address="net.tcp://localhost:8081/echo/svc"
              binding="netTcpBinding"
              contract="ServiceLibrary.IEchoService"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```



Credit: http://msdn.microsoft.com/en-us/magazine/cc163647(printer).aspx
(Learn the ABCs of Programming Windows Communication Foundation)

ABC 2 of WCF
1) Adding Service Endpoints in Code
and
2) In Configuration File

Client

1 Adding Service Endpoints In Code

```
using System;
using System.ServiceModel;
using ServiceLibrary;

class Program
{
    static void Main(string[] args)
    {
        try { // Add service endpoints on client
            ServiceEndpoint httpEndpoint = new ServiceEndpoint(ContractDescription.GetContract(

                typeof(IEchoService)),new BasicHttpBinding(), new EndpointAddress("http://localhost:8080/echo/svc"));

            ServiceEndpoint tcpEndpoint= new ServiceEndpoint(ContractDescription.GetContract(
                typeof(IEchoService)),new NetTcpBinding(), new EndpointAddress("net.tcp://localhost:8081/echo/svc"));

            IEchoService svc = null;

            // create channel factory based on HTTP endpoint
            using (ChannelFactory<IEchoService> httpFactory = new ChannelFactory<IEchoService>(httpEndpoint)) {
                svc = httpFactory.CreateChannel(); // create channel object for designated endpoint
                Console.WriteLine("Invoking HTTP endpoint: {0}", svc.Echo("Hello, world")); // invoke service operation
            }
            // create channel factory based on TCP endpoint
            using (ChannelFactory<IEchoService> tcpFactory = new ChannelFactory<IEchoService>(tcpEndpoint)) {
                svc = tcpFactory.CreateChannel(); // create channel proxy for endpoint
                Console.WriteLine("Invoking TCP endpoint: {0}", svc.Echo("Hello, world")); // invoke service operation
            }
        }
        catch (Exception e){
            Console.WriteLine(e);
        }
    }
}
```

2b Using Endpoints in Config File

```
IEchoService svc = null;
using (ChannelFactory<IEchoService> httpFactory = new ChannelFactory<IEchoService>("httpEndpoint"))
{
    svc = httpFactory.CreateChannel();
    Console.WriteLine("Invoking HTTP endpoint: {0}",svc.Echo("Hello, world"));
}
```

2a Define Service Endpoint in Config File

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="httpEndpoint" address="http://localhost:8080/echo/svc"
        binding="basicHttpBinding" contract="ServiceLibrary.IEchoService"/>
      <endpoint name="tcpEndpoint" address="net.tcp://localhost:8081/echo/svc"
        binding="netTcpBinding" contract="ServiceLibrary.IEchoService"/>
    </client>
  </system.serviceModel>
</configuration>
```

WCF Addressing

You can have one **base address per transport type**

```
ServiceHost host = new ServiceHost(
      typeof(CalculatorService),
      new Uri("http://localhost:8080/calcservice"),       // base HTTP address
      new Uri("net.tcp://localhost:8081/calcservice")); // base TCP address
```

By default, WCF uses base HTTP address to expose metadata when GET retrieval has bee enabled (via the <serviceMetadata> behavior). You can also specify a specific address by behavior's **httpGetUrl** property.

Clients have no awareness of the service's base addresses!

When Service is hosted by IIS, base address is always the virtual directory of IIS. Specifying absolute addresses need to be careful to match IIS's virtual directory; otherwise, exception will be throw at the start of the Service; thus, it really only makes sense to use relative address when Service is hosted by IIS.

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalculatorService">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8080/calcservice"/>
            <add baseAddress="net.tcp://localhost:8081/calcservice"/>
          </baseAddresses>
        </host>
        <endpoint binding="basicHttpBinding" contract="ISimpleMath"/>
         <endpoint address="secure"
                 binding="wsHttpBinding" contract="ISimpleMath"/>
        <endpoint binding="netTcpBinding" contract="ISimpleMath"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Using This as Base Address

Using This as Base Address

Two reasons to have multiple bindings for a Service:

1) Need to have multiple different bindings
2) Need to support multiple Contracts.

When exposing multiple endpoints with **different Bindings, each endpoint address must be unique. This is because each endpoint (really each Binding) translates into a different listener stack and thus channel stack.**

When exposing multiple endpoints with different Contracts, endpoint address can be the same. In this case, both endpoints end up sharing the same listener and channel stack; dispatcher dispatches message to different set of methods.

[Wrong Way! To Add Bindings with the Same Address]

```
ServiceHost host = new ServiceHost(typeof(CalculatorService));

host.AddServiceEndpoint(
    typeof(ISimpleMath), new WSHttpBinding(),
    "http://localhost:8080/calc");

host.AddServiceEndpoint(
    typeof(IScientific), new WSHttpBinding(),  // <<< this is new binding
    "http://localhost:8080/calc");
host.Open();
...
```

Wrong

[**Correct Way! To Add Bindings with the Same Address**]

Correct

```
ServiceHost host = new ServiceHost(typeof(CalculatorService))
WSHttpBinding wsbinding = new WSHttpBinding();

host.AddServiceEndpoint(
    typeof(ISimpleMath), wsbinding, "http://localhost:8080/calc");

host.AddServiceEndpoint(
    typeof(IScientific), wsbinding, "http://localhost:8080/calc");
host.Open();
...
```

Specify Physical Address by ListenUri in Code

```
host.AddServiceEndpoint(
    typeof(ISimpleMath), wsbinding, // binding
    "urn:calcservice:simplemath", // address is logical
    new Uri("http://localhost:8080/calc")); // physical (listenUri)

host.AddServiceEndpoint(
    typeof(IScientific), wsbinding, // binding
    "urn:calcservice:scientific", // address is logical
    new Uri("http://localhost:8080/calc")); // physical (listenUri)
```

ListenUriMode = ListenUriMode.Unique or .Explicit (Default)

Service
(Physical Address)

```
SimpleMathClient client = new SimpleMathClient(
    "WSHttpBinding_ISimpleMath");

client.Endpoint.Behaviors.Add(new ClientViaBehavior(
        new Uri("http://localhost:8080/calcservice")));

double sum = client.Add(3, 4);
```

Specify Physical Address in Client (In Code)

Client
(Physical Address)

Clients have no notion of a ListenUri. As far as clients know, there's a single address for each endpoint. Since the WSDL definition contains the logical address for each endpoint, svcutil.exe embeds the logical address in the client configuration and will use it for transmission by default. When a service has been configured with a different physical address, you'll need an **out-of-band mechanism** to inform clients of the physical address to use.

Different Bindings => Must Have Different Addresses

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalculatorService">
        <endpoint address="http://localhost:8080/calcservice"
                 binding="basicHttpBinding" contract="ISimpleMath"/>
        <endpoint address="http://localhost:8080/calcservice/secure"
                 binding="wsHttpBinding" contract="ISimpleMath"/>
        <endpoint address="net.tcp://localhost:8081/calcservice"
                 binding="netTcpBinding" contract="ISimpleMath"/>
      </service>
 ...
```

Different Contracts => Can Have Same Addresses

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalculatorService">
        <endpoint address="http://localhost:8080/calcservice"
                 binding="wsHttpBinding" contract="ISimpleMath"/>
        <endpoint address="http://localhost:8080/calcservice"
                 binding="wsHttpBinding" contract="IScientific"/>
</service>
 ...
```

[Absolute Address Example]

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalculatorService">
        <endpoint address="http://localhost:8080/calcservice"
                 binding="basicHttpBinding" contract="ISimpleMath"/>
        <endpoint address="http://localhost:8080/calcservice/secure"
                 binding="wsHttpBinding" contract="ISimpleMath"/>
        <endpoint address="net.tcp://localhost:8081/calcservice"
                 binding="netTcpBinding" contract="ISimpleMath"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Specify Physical Address by ListenUri in Config

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalculatorService">
        <endpoint address="urn:calcservice:simplemath"
                 listenUri="http://localhost:8080/calcservice"
                 binding="wsHttpBinding" contract="ISimpleMath"/>
        <endpoint address="urn:calcservice:scientific"
                 listenUri="http://localhost:8080/calcservice"
                 binding="wsHttpBinding" contract="IScientific"/>
      </service>
      ...
```

Specify Physical Address in Client (In Config)

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="WSHttpBinding_ISimpleMath" address="urn:calcservice:simplemath"
           binding="wsHttpBinding" contract="Client.localhost.ISimpleMath"
           behaviorConfiguration="Via" />
      ...
    </client>
    <behaviors>
      <endpointBehaviors>
        <behavior name="Via"> <clientVia viaUri="http://localhost:8080/calcservice"/></behavior>
      </endpointBehaviors>
    </behaviors>
    ...
```

Credit: http://msdn.microsoft.com/en-us/magazine/cc163412.aspx **(WCF Addressing In Depth)**

WCF Addressing 2

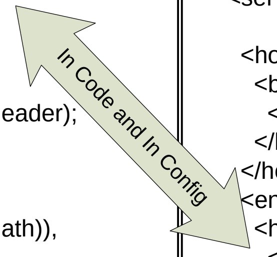
[Custom Address Header in SOAP]

```
<s:Envelope xmlns:s=”http://www.w3.org/2003/05/soap-envelope”
            xmlns:a=”http://www.w3.org/2005/08/addressing”>
  <s:Header>
    <a:Action s:mustUnderstand=”1”>http://example.org/calc/Add</a:Action>
    <a:MessageID>urn:uuid:db1ba7a6-ca2e-494b-b390-9f621e8b90f4</a:MessageID>
    <a:ReplyTo>
      <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
    </a:ReplyTo>
    <a:To s:mustUnderstand=”1”>http://localhost:8080/calcservice</a:To>
    <basic a:IsReferenceParameter=”true” xmlns=”http://example.org/level”/>
  </s:Header>
  <s:Body>
  ...
```

[Custom Address Header in WSDL]

```
...
<wsdl:service name=”CalculatorService”>
  <wsdl:port name=”WSHttpBinding_ISimpleMath” binding=”tns:WSHttpBinding_ISimpleMath”>
    <soap12:address location=”http://localhost:8080/calcservice”/>
    <wsa10:EndpointReference>
      <wsa10:Address>http://localhost:8080/calcservice</wsa10:Address>
      <wsa10:ReferenceParameters>
        <basic xmlns=”http://example.org/level”/>
      </wsa10:ReferenceParameters>
      ...
    </wsa10:EndpointReference>
  </wsdl:port>
  ...
</wsdl:service>
...
```

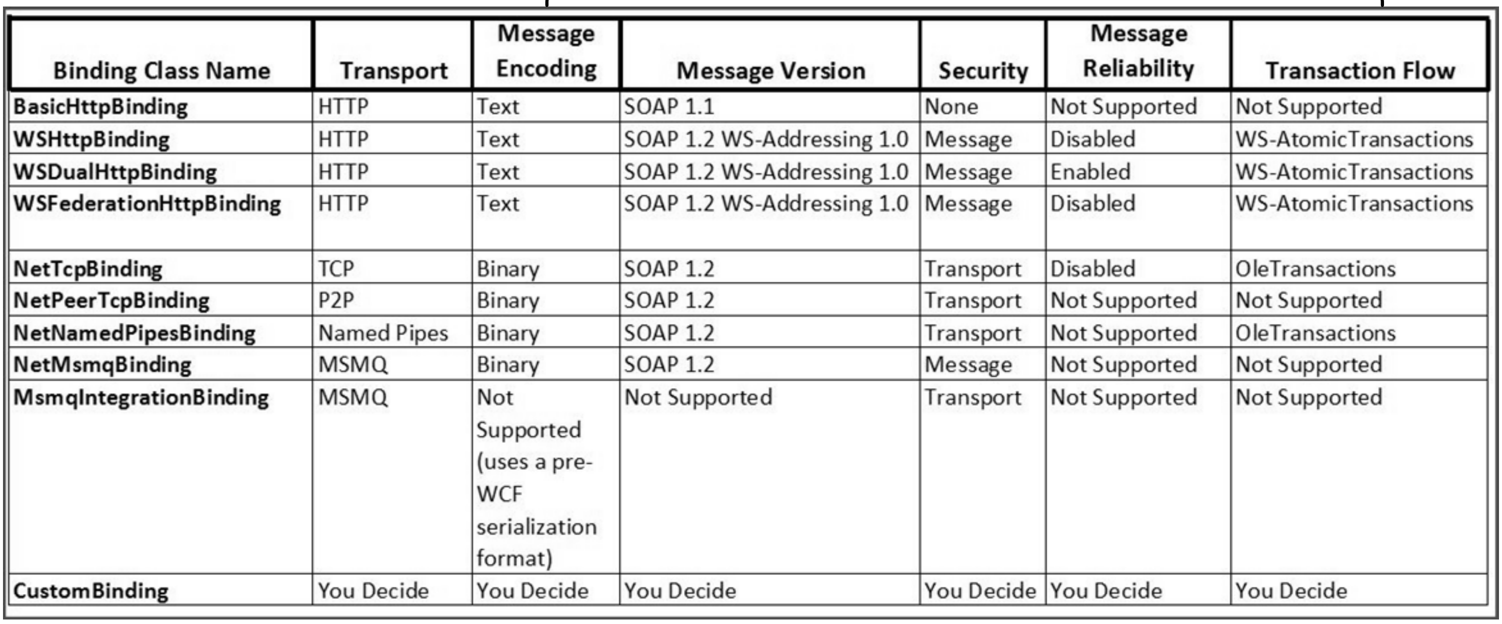
```
ServiceHost host = new ServiceHost(typeof(CalculatorService));

AddressHeader header=
AddressHeader.CreateAddressHeader(“basic”,
    “http://example.org/level”, null);

EndpointAddress ea = new EndpointAddress(
  new Uri(“http://localhost:8080/calcservice/foobar”), header);

host.Description.Endpoints.Add(
  new ServiceEndpoint(
    ContractDescription.GetContract(typeof(ISimpleMath)),
    new WSHttpBinding(), ea));
...
```



```
<configuration>
  <system.serviceModel>
    <services>
      <service name=”CalculatorService”
               behaviorConfiguration=”metadata”>
        <host>
          <baseAddresses>
            <add baseAddress=”http://localhost:8080/calcservice”/>
          </baseAddresses>
        </host>
        <endpoint binding=”wsHttpBinding” contract=”ISimpleMath”>
          <headers>
            <basic xmlns=”http://example.org/level”/>
          </headers>
        </endpoint>
        <endpoint binding=”wsHttpBinding” contract=”IScientific”>
          <headers>
            <premium xmlns=”http://example.org/level”/>
          </headers>
        </endpoint>
      </service>
  ...
```

WCF Binding

Default Property Values



Binding Class Name	Transport	Message Encoding	Message Version	Security	Message Reliability	Transaction Flow
BasicHttpBinding	HTTP	Text	SOAP 1.1	None	Not Supported	Not Supported
WSHttpBinding	HTTP	Text	SOAP 1.2 WS-Addressing 1.0	Message	Disabled	WS-AtomicTransactions
WSDualHttpBinding	HTTP	Text	SOAP 1.2 WS-Addressing 1.0	Message	Enabled	WS-AtomicTransactions
WSFederationHttpBinding	HTTP	Text	SOAP 1.2 WS-Addressing 1.0	Message	Disabled	WS-AtomicTransactions
NetTcpBinding	TCP	Binary	SOAP 1.2	Transport	Disabled	OleTransactions
NetPeerTcpBinding	P2P	Binary	SOAP 1.2	Transport	Not Supported	Not Supported
NetNamedPipesBinding	Named Pipes	Binary	SOAP 1.2	Transport	Not Supported	OleTransactions
NetMsmqBinding	MSMQ	Binary	SOAP 1.2	Message	Not Supported	Not Supported
MsmqIntegrationBinding	MSMQ	Not Supported (uses a pre-WCF serialization format)	Not Supported	Transport	Not Supported	Not Supported
CustomBinding	You Decide	You Decide	You Decide	You Decide	You Decide	You Decide

```
using (ServiceHost host = new ServiceHost(typeof(ChatService)))
{
    host.AddServiceEndpoint(typeof(IChat),
        new BasicHttpBinding(), “http://localhost:8080/chat”);
    host.AddServiceEndpoint(typeof(IChat),
        new WSHttpBinding(), “http://localhost:8080/chat/secure”);
    host.AddServiceEndpoint(typeof(IChat),
        new NetTcpBinding(), “net.tcp://localhost:8081/chat”);

    host.Open();

    ... // remaining code ommitted for brevity
}
```

Using Binding In Code

Using Binding In Config File

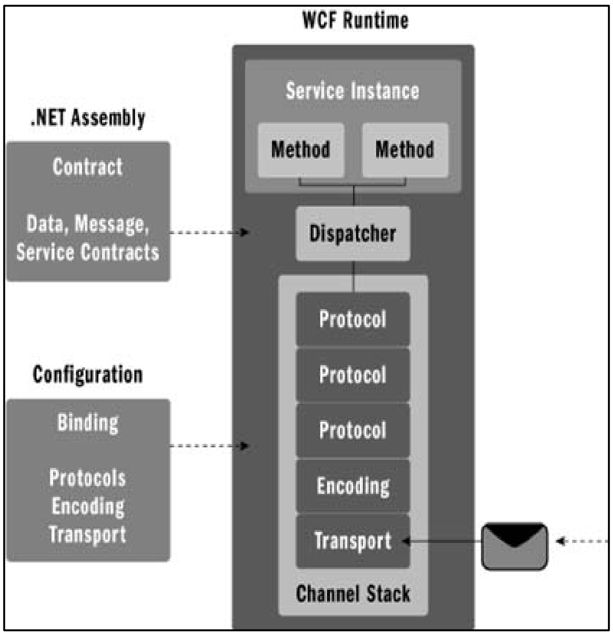
```
<configuration>
  <system.serviceModel>
    <services>
      <service name=”ChatService”>
        <endpoint
          address=”http://localhost:8080/chat”
          binding=”basicHttpBinding”
          contract=”IChat”/>
        <endpoint
          address=”http://localhost:8080/chat/secure”
          binding=”wsHttpBinding”
          contract=”IChat”/>
        <endpoint
          address=”net.tcp://localhost:8081/chat”
          binding=”netTcpBinding”
          contract=”IChat”/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

```
BasicHttpBinding basicHttpBinding = new BasicHttpBinding();
basicHttpBinding.MessageEncoding = WSMessageEncoding.Mtom;
basicHttpBinding.Security.Mode = BasicHttpSecurityMode.Transport;

host.AddServiceEndpoint(
    typeof(IChatService),
    basicHttpBinding,
    “http://localhost:8080/chat”);
...
```

Change Binding Properties In Code

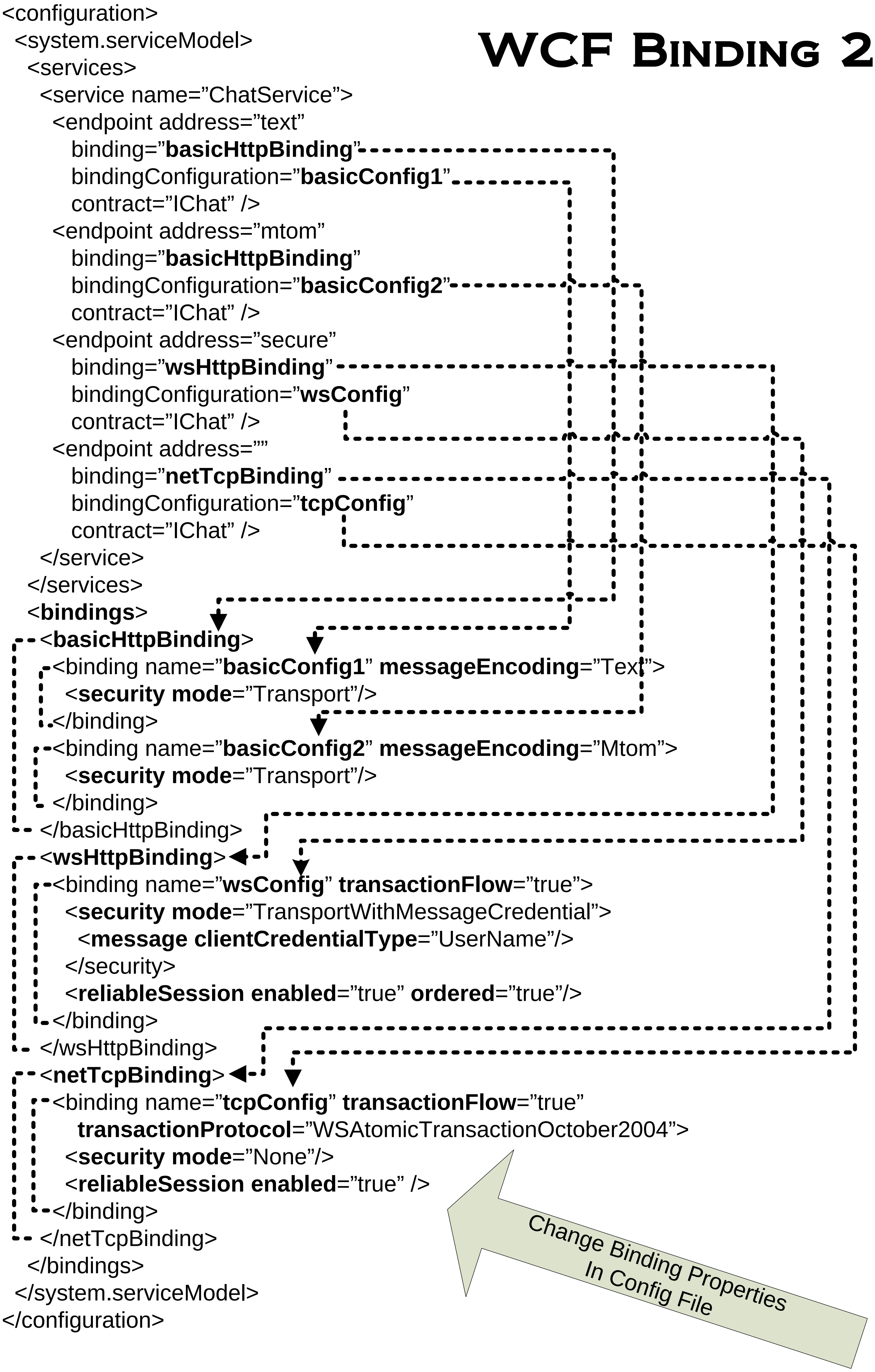
Change Binding Properties In Config File

```
<configuration>
  <system.serviceModel>
    <services>
      <service name=”ChatService”>
        <endpoint address=”http://localhost:8080/chat”
          binding=”basicHttpBinding”
          bindingConfiguration=”basicConfig”
          contract=”ChatLibrary.IChat” />
        ...
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding name=”basicConfig”
                    messageEncoding=”Mtom”>
          <security mode=”Transport”/>
        </binding>
      </basicHttpBinding>
      ...
    </bindings>
  </system.serviceModel>
</configuration>
```



Credit: http://msdn.microsoft.com/en-us/magazine/cc163394(printer).aspx (WCF Binding in Depth)

WCF Binding 2



```
<configuration>
  <system.serviceModel>
    <services>
      <service name=”ChatService”>
        <endpoint address=”text”
           binding=”basicHttpBinding”
           bindingConfiguration=”basicConfig1”
           contract=”IChat” />
        <endpoint address=”mtom”
           binding=”basicHttpBinding”
           bindingConfiguration=”basicConfig2”
           contract=”IChat” />
        <endpoint address=”secure”
           binding=”wsHttpBinding”
           bindingConfiguration=”wsConfig”
           contract=”IChat” />
        <endpoint address=””
           binding=”netTcpBinding”
           bindingConfiguration=”tcpConfig”
           contract=”IChat” />
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding name=”basicConfig1” messageEncoding=”Text”>
          <security mode=”Transport”/>
        </binding>
        <binding name=”basicConfig2” messageEncoding=”Mtom”>
          <security mode=”Transport”/>
        </binding>
      </basicHttpBinding>
      <wsHttpBinding>
        <binding name=”wsConfig” transactionFlow=”true”>
          <security mode=”TransportWithMessageCredential”>
            <message clientCredentialType=”UserName”/>
          </security>
          <reliableSession enabled=”true” ordered=”true”/>
        </binding>
      </wsHttpBinding>
      <netTcpBinding>
        <binding name=”tcpConfig” transactionFlow=”true”
            transactionProtocol=”WSAtomicTransactionOctober2004”>
          <security mode=”None”/>
          <reliableSession enabled=”true” />
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Change Binding Properties
In Config File

Binding Terminologies

Binding = defines which communication protocols are used to communicate with WCF services. It is constructed of a set of components called binding elements that stack one on top of the other to create the communication infrastructure. See endpoint.

Binding Element = represents a particular piece of the binding, such as a transport, an encoding, an implementation of an infrastructure-level protocol (such as WS-ReliableMessaging), or any other component of the communication stack.

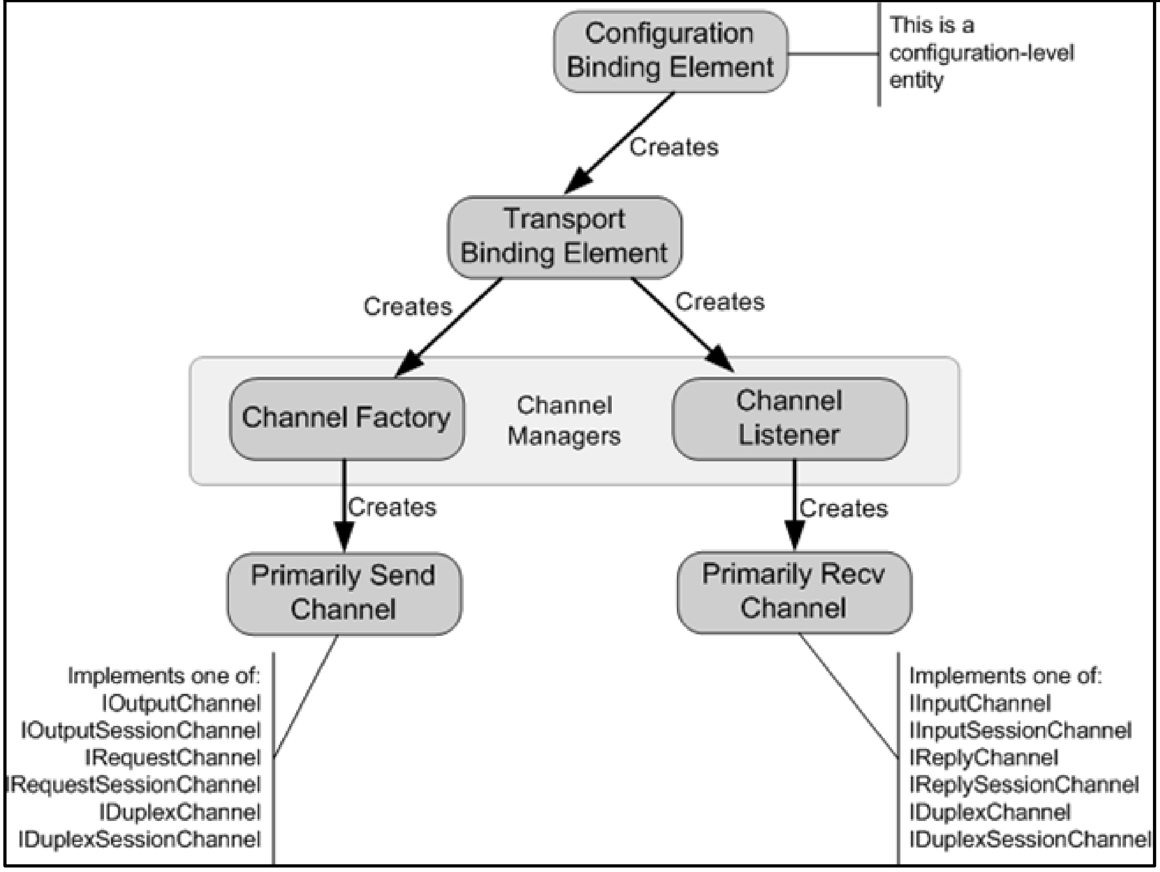
Channel = A concrete implementation of a binding element. The binding represents the configuration, and the channel is the implementation associated with that configuration. Therefore, there is a channel associated with each binding element. Channels stack on top of each other to create the concrete implementation of the binding: the channel stack.

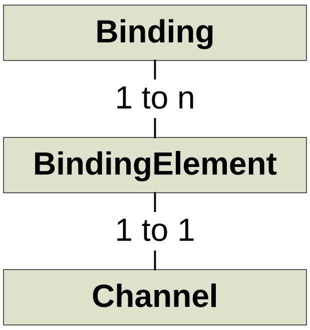
Security = in WCF includes **confidentiality** (encryption of messages to prevent eavesdropping), **integrity** (the means for detection of tampering with the message), **authentication** (the means for validation of servers and clients), and **authorization** (the control of access to resources). These functions are provided by either leveraging existing security mechanisms, such as TLS over HTTP (also known as HTTPS), or by implementing one or more of the various WS-* security specifications.

Transport Security Mode = Security can be provided by one of three modes: **transport mode**, **message security mode**, and **transport with message credential mode**. The transport security mode specifies that confidentiality, integrity, and authentication are provided by the transport layer mechanisms (such as HTTPS). When using a transport like HTTPS, this mode has the advantage of being efficient in its performance, and well understood because of its prevalence on the Internet. The disadvantage is that this kind of security is applied separately on each hop in the communication path, making the communication susceptible to a man in the middle attack.

Transport with message credential security mode = This mode uses the transport layer to provide confidentiality, authentication, and integrity of the messages, while each of the messages can contain multiple credentials (claims) required by the receivers of the message.

Message Security Mode = Message security mode specifies that security is provided by implementing one or more of the security specifications, such as the specification named "Web Services Security: SOAP Message Security" (available at http://go.microsoft.com/fwlink/?LinkId=94684). Each message contains the necessary mechanisms to provide security during its transit, and to enable the receivers to detect tampering and to decrypt the messages. In this sense, the security is encapsulated within every message, providing end-to-end security across multiple hops. Because security information becomes part of the message, it is also possible to include multiple kinds of credentials with the message (these are referred to as *claims*). This approach also has the advantage of enabling the message to travel securely over any transport, including multiple transports between its origin and destination. The disadvantage of this approach is the complexity of the cryptographic mechanisms employed, resulting in performance implications.





Confidentiality

Integrity

Authentication

Authorization

http://www.cnblogs.com/artech/archive/2008/07/08/1237902.html

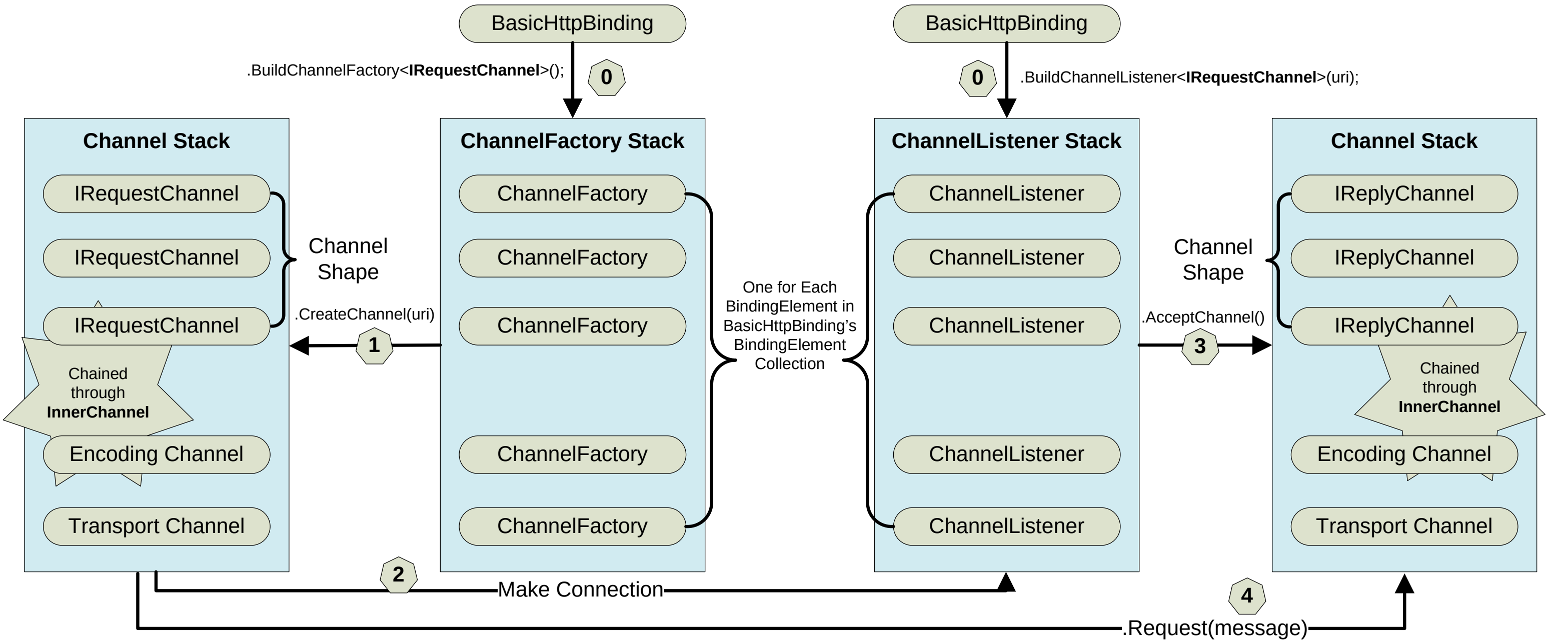
Channel Layer and Binding

Client-Side

```
static void Main(string[] args)
{
  EndpointAddress address = new EndpointAddress(
                    "http://127.0.0.1:9999/MessagingViaBinding");
  BasicHttpBinding binding = new BasicHttpBinding();

  IChannelFactory<IRequestChannel> chananelFactory =
                    binding.BuildChannelFactory<IRequestChannel>();
  chananelFactory.Open();

  IRequestChannel channel =
                              chananelFactory.CreateChannel(address);
  channel.Open();

  Message requestMessage = Message.CreateMessage(
      MessageVersion.Soap11,
      "http://artech/messagingviabinding",
      "Manually created for Demo Purpose.");

  Message replyMessage = channel.Request(requestMessage);
  Console.WriteLine("Receive a reply message:\n{0}", replyMessage);
  channel.Close();
  chananelFactory.Close();
  Console.Read();
}
```

Service-Side

```
static void Main(string[] args)
{
  Uri address = new Uri("http://127.0.0.1:9999/MessagingViaBinding");
  BasicHttpBinding binding = new BasicHttpBinding();

  IChannelListener<IReplyChannel> channelListener =
               binding.BuildChannelListener<IReplyChannel>(address);
  channelListener.Open(); // <<<< Listening Here! (Block)

  IReplyChannel channel = channelListener.AcceptChannel();
  channel.Open(); // <<< Creates Channel Stack and start accepting Messages

  Console.WriteLine("Begin to listen  ");

  while (true)
  {
    RequestContext context=channel.ReceiveRequest(new TimeSpan(1,0,0));

    Console.WriteLine("Receive a request message:\n{0}", context.RequestMessage);
    Message replyMessage = Message.CreateMessage(
        MessageVersion.Soap11,
        "http://artech.messagingviabinding",
        "Manually created reply message for Demo purpose");
    context.Reply(replyMessage);
  }
}
```



How Channel Stack is Created by a Binding:

1) **ChannelListener stack and ChannelFactory stack are created:**
Binding.**BuildChannelListener<TChannel>(uri)** creates **ChannelListener<TChannel>** (on Service Side) and Binding.**BuildChannelFactory<TChannel>()** creates **ChannelFactory<TChannel>** (on Client Side), for each BindingElement in Binding's BindingElementCollection.

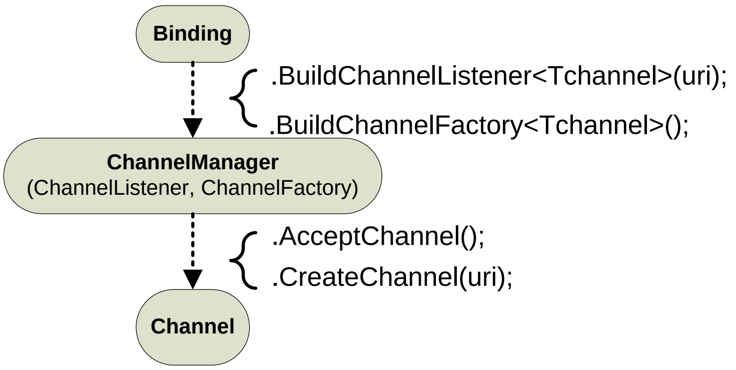
2) **ChannelListen stack and ChannelFactory stack create Channel Stack:**
In turn, each **ChannelListener**.AcceptChannel() creates a **Channel** (one specified by TChannel---one of those Service-Side Channel Shapes) to communicate with Client, and each **ChannelFactory**.CreateChannel(**uri**) creates a **Channel** (one specified by TChannel---one of those Client-Side Channel Shapes) to make connection to and communicate with Service.

Because a Channel is created by ChannelListener and ChannelFactory which are created by a BindingElement, in order to create a new Custom Channel, one needs to 1) Create a two new Custom Channel classes---one for Sending and one for Receiving, then 2) Create a Custom BindingElement added to the Binding (new Binding class or CustomBinding class), in the Custom BindingElement, implement BuildChannelListener<TChannel> (BindingContext ctx) and BuildChannelFactory<TChannel>(BindingContext ctx) which returns ChannelListener<TChannel> and ChannelFactory<TChannel> , respectively.

Channel Shapes

Client-Side / Service-Side

IRequestChannel / IReplyChannel
IRequestSessionChannel / IReplySessionChannel
IOutputChannel / IInputChannel
IOutputSessionChannel / IInputSessionChannel
IDuplexChannel / IDuplexChannel
IDuplexSessionChannel / IDuplexSessionChannel



Binding
.BuildChannelListener<Tchannel>(uri);
.BuildChannelFactory<Tchannel>();
ChannelManager
(ChannelListener, ChannelFactory)
.AcceptChannel();
.CreateChannel(uri);
Channel

Channel Layer and Binding 2

CommunicationObject vs **ICommunicationObject**

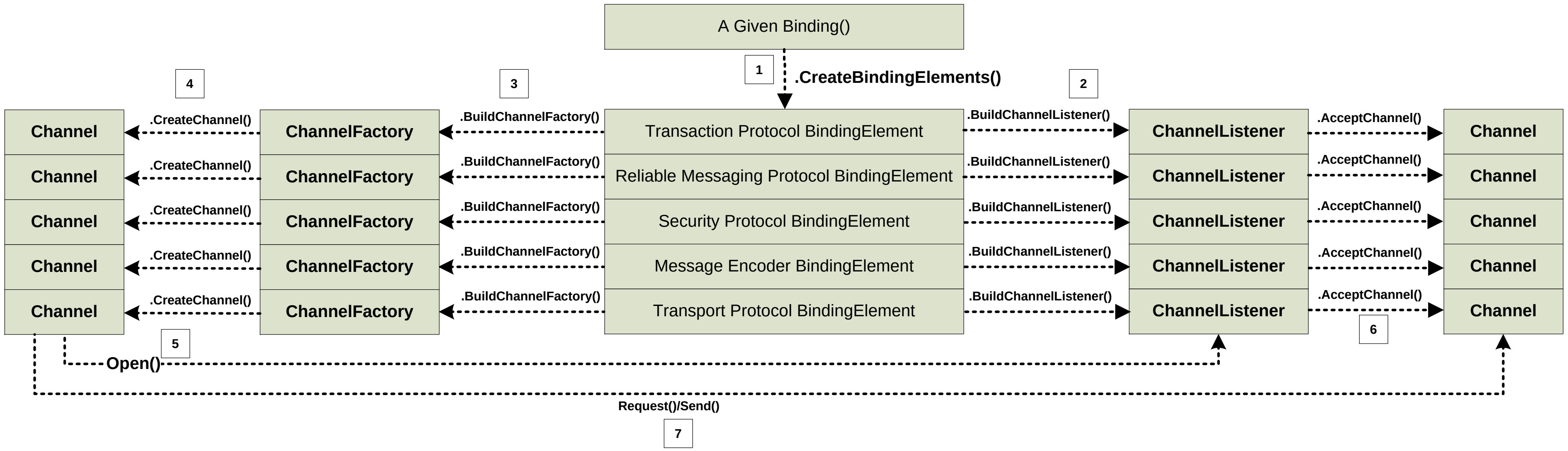
CommunicationObject implement ICommunicationObject, which define standard communication state-machine and events (Closing, Closed, Faulted, Opening, Opened).

Channel vs **Channel Shapes**

Each Channel is one of Channel Shapes (IOutputChannel - IInputChanel, etc.); these Interfaces have both Sync and Async verions of Send() and Receive() on Client and Service side respectively.

IOutputChannel.Send()/BeginSend() – IInputChannel.Receive()/BeginReceive()
IRequestChannel.Request()/BeginRequest() – IReplyChannel.ReceiveRequest/BeginReceiveRequest()
IDuplexChannel: IOutputChannel, IInputChannel

How Channel Stack is Created



A Given Binding()

1 .CreateBindingElements()

2 3 4

Transaction Protocol BindingElement
Reliable Messaging Protocol BindingElement
Security Protocol BindingElement
Message Encoder BindingElement
Transport Protocol BindingElement

.BuildChannelFactory() → ChannelFactory → .CreateChannel() → Channel

.BuildChannelListener() → ChannelListener → .AcceptChannel() → Channel

5 Open()

6

Request()/Send()

7

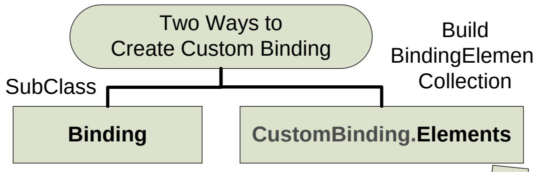
http://www.cnblogs.com/artech/archive/2008/07/09/1238626.html

Create WCF “Custom Binding” Using CustomBinding

Protocol	BindingElement Class	Configuration Element
Transaction Flow	TransactionFlowBindingElement	<transactionFlow/>
Reliable Messaging	ReliableSessionBindingElement	<reliableSession/>
Security	SecurityBindingElement	<security/>

Message Enoding	BindingElement Class	Configuration Element
Text	TextMessageEncodingBindingElement	<textMessageEncoding/>
MTOM	MtomMessageEncodingBindingElement	<mtomMessageEncoding/>
Binary	BinaryMessageEncodingBindingElement	<binaryMessageEncoding/>

Transport	BindingElement Class	Configuration Element
HTTP	HttpTransportBindingElement	<httpTransport/>
HTTPS	HttpsTransportBindingElement	<httpsTransport/>
TCP	TcpTransportBindingElement	<tcpTransport/>
Named pipes	NamedPipeTransportBindingElement	<namedPipeTransport/>
MSMQ	MsmqTransportBindingElement	<msmqTransport/>
MSMQ	MsmqIntegrationBindingElement	<msmqIntegration/>
P2P	PeerTransportBindingElement	<peerTransport/>


Two Ways to Create Custom Binding
SubClass
Build BindingElement Collection
Binding
CustomBinding.Elements

Order in **BindingElementCollection**:

= Transaction Flow (Not Required)
= Reliable Messaging (Not Required)
= Message Security (Not Required)
= Composite Duplex (Not Required)
= **Message Encoding** (Required*)
= Transport Security (Not Required)
= **Transport** (Required)

* Default is provided if not given

```
// instantiate message encoding element and configure
TextMessageEncodingBindingElement text = new TextMessageEncodingBindingElement();
text.MessageVersion = MessageVersion.Soap11WSAddressingAugust2004;

// instantiate transport element and configure
HttpTransportBindingElement http = new HttpTransportBindingElement();
http.TransferMode = TransferMode.Streamed;
http.UseDefaultWebProxy = true;

CustomBinding myHttpBinding = new CustomBinding();
myHttpBinding.Name = “myHttpBinding”;
myHttpBinding.Elements.Add(text); myHttpBinding.Elements.Add(http);

host.AddServiceEndpoint(typeof(IChat), myHttpBinding, “http://localhost:8080/chat/custom”);
...
```

Building BindingElement Collection In Code

```
<configuration>
  <system.serviceModel>
    <services>
      <service name=”ChatService”>
        <endpoint address=”custom” contract=”IChat”
          binding=”customBinding” bindingConfiguration=”myBasicHttpBindingConfiguration”/>
      </service>
    </services>
    <bindings>
      <customBinding>
        <binding name=”myBasicHttpBindingConfiguration”>
          <textMessageEncoding messageVersion=”Soap11WSAddressingAugust2004”/>
          <httpTransport useDefaultWebProxy=”true” transferMode=”Streamed”/>
        </binding>
      </customBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Building BindingElement Collection In Config

Another Example

```
 ...
    <bindings>
      <customBinding>
        <binding name=”myBasicHttpBindingConfiguration”>
          <textMessageEncoding messageVersion=”Soap11WSAddressingAugust2004”/>
          <httpTransport allowCookies=”true” useDefaultWebProxy=”true” transferMode=”Streamed”/>
        </binding>
        <binding name=”myWSHttpBindingConfiguration”>
          <transactionFlow/> <reliableSession ordered=”true”/> <security authenticationMode=”SspiNegotiated”/>
          <binaryMessageEncoding/>
          <httpTransport/>
        </binding>
        <binding name=”myNetTcpBindingConfiguration”>
          <transactionFlow/> <textMessageEncoding/> <windowsStreamSecurity/> <tcpTransport/>
        </binding>
      </customBinding>
    </bindings>
```

BindingElements

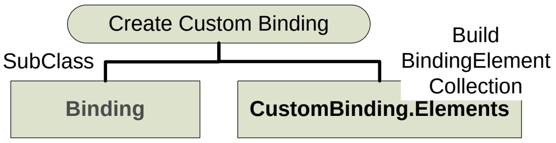
BindingElements

BindingElements

Create WCF Custom Binding, by subclassing Binding 2

1 New Custom Binding --- **NetHttpBinding**

```
public class NetHttpBinding : Binding
{
   private var binary = new BinaryMessageEncodingBindingElement();
   private var http = new HttpTransportBindingElement();

   public override BindingElementCollection CreateBindingElements() {
      return new BindingElementCollection(
                                        new BindingElement[] { binary, http });
   }
   public TransferMode TransferMode{
      get { return http.TransferMode; }
      set { http.TransferMode = value; }
   }
   public bool UseDefaultWebProxy{
      get { return http.UseDefaultWebProxy; }
      set { http.UseDefaultWebProxy = value; }
   }
   public override string Scheme { get { return “http”; } }
}
```



Using the New **NetHttpBinding** in Code 2

```
...
NetHttpBinding netHttp = new NetHttpBinding();
netHttp.TransferMode = TransferMode.Streamed;
netHttp.UseDefaultWebProxy = true;

host.AddServiceEndpoint(typeof(IChat), netHttp,
              “http://localhost:8080/chat/nethttp”);
...
```

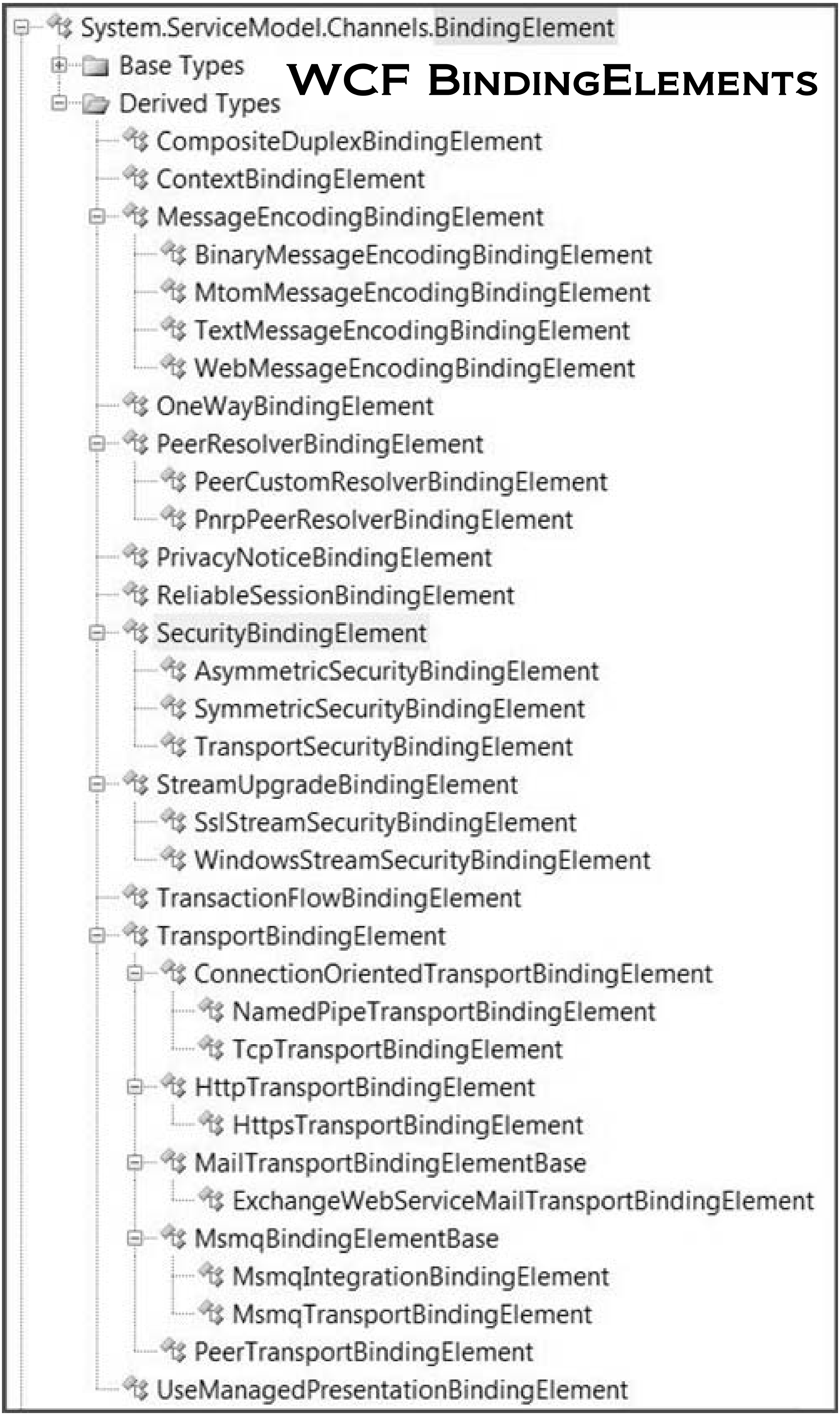
2' Making Newly Created NetHttpBinding Available to Config File

```
public class NetHttpBindingConfigurationElement : StandardBindingElement
{
   public NetHttpBindingConfigurationElement(string configurationName) : base(configurationName) { }
   public NetHttpBindingConfigurationElement(): this(null) { }

   protected override Type BindingElementType {get { return typeof(NetHttpBinding); }}

   [ConfigurationProperty(“transferMode”, DefaultValue = TransferMode.Buffered)]
   public TransferMode TransferMode{
      get { return ((TransferMode)(base[“transferMode”])); }
      set { base[“transferMode”] = value; }
   }
   [ConfigurationProperty(“useDefaultWebProxy”, DefaultValue = false)]
   public bool UseDefaultWebProxy{
      get { return ((bool)(base[“useDefaultWebProxy”])); }
      set { base[“useDefaultWebProxy”] = value; }
   }

   protected override ConfigurationPropertyCollection Properties{
      get {
         ConfigurationPropertyCollection properties = base.Properties;
         properties.Add(new ConfigurationProperty(“transferMode”, typeof(TransferMode), TransferMode.Buffered));
         properties.Add(new ConfigurationProperty(“useDefaultWebProxy”, typeof(bool), true));
         return properties;
      }
   }
   protected override void InitializeFrom(Binding binding) {
      base.InitializeFrom(binding);
      NetHttpBinding netHttpBinding = ((NetHttpBinding)(binding));
      this.TransferMode = netHttpBinding.TransferMode;
      this.UseDefaultWebProxy = netHttpBinding.UseDefaultWebProxy;
   }
   protected override void OnApplyConfiguration(Binding binding) {
      if (binding == null)
         throw new System.ArgumentNullException(“binding”);
      if (binding.GetType() != typeof(NetHttpBinding))
         throw new System.ArgumentException(
                     “Invalid binding type – expected NetHttpBinding”);

      NetHttpBinding netHttpBinding = ((NetHttpBinding)(binding));
      netHttpBinding.TransferMode = this.TransferMode;
      netHttpBinding.UseDefaultWebProxy = this.UseDefaultWebProxy;
   }
}
public class NetHttpBindingSection : StandardBindingCollectionElement<NetHttpBinding,
NetHttpBindingConfigurationElement>
{ }
```

Using the **NetHttpBinding** In Config File 3

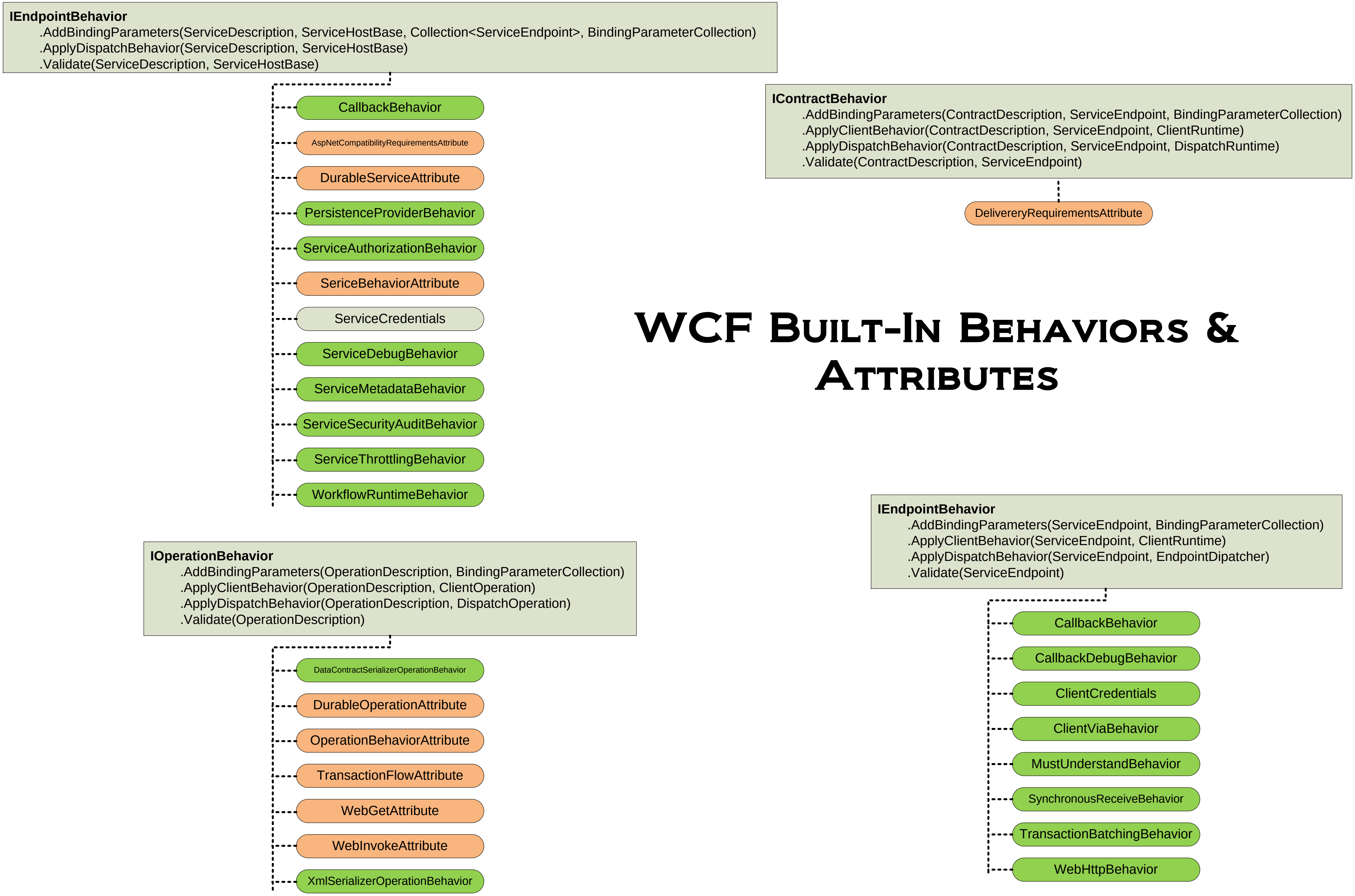
```
...
    <endpoint address=”nethttp” contract=”ChatLibrary.IChat”
      binding=”netHttpBinding”
      bindingConfiguration=”myNetHttpBindingConfiguration”/>
...
  <bindings>
    <netHttpBinding>
      <binding name=”myNetHttpBindingConfiguration”
        transferMode=”Streamed” useDefaultWebProxy=”true”/>
    </netHttpBinding>
  </bindings>
  <extensions>
    <bindingExtensions>
      <add name=”netHttpBinding”
        type=”NetHttpBindingSection, NetHttpBinding” />
    </bindingExtensions>
  </extensions>
 </system.serviceModel>
</configuration>
```

WCF BindingElements



- System.ServiceModel.Channels.BindingElement
  - Base Types
  - Derived Types
    - CompositeDuplexBindingElement
    - ContextBindingElement
    - MessageEncodingBindingElement
      - BinaryMessageEncodingBindingElement
      - MtomMessageEncodingBindingElement
      - TextMessageEncodingBindingElement
      - WebMessageEncodingBindingElement
    - OneWayBindingElement
    - PeerResolverBindingElement
      - PeerCustomResolverBindingElement
      - PnrpPeerResolverBindingElement
    - PrivacyNoticeBindingElement
    - ReliableSessionBindingElement
    - SecurityBindingElement
      - AsymmetricSecurityBindingElement
      - SymmetricSecurityBindingElement
      - TransportSecurityBindingElement
    - StreamUpgradeBindingElement
      - SslStreamSecurityBindingElement
      - WindowsStreamSecurityBindingElement
    - TransactionFlowBindingElement
    - TransportBindingElement
      - ConnectionOrientedTransportBindingElement
        - NamedPipeTransportBindingElement
        - TcpTransportBindingElement
      - HttpTransportBindingElement
        - HttpsTransportBindingElement
      - MailTransportBindingElementBase
        - ExchangeWebServiceMailTransportBindingElement
      - MsmqBindingElementBase
        - MsmqIntegrationBindingElement
        - MsmqTransportBindingElement
      - PeerTransportBindingElement
    - UseManagedPresentationBindingElement

WCF Built-In Behaviors & Attributes



IEndpointBehavior
.AddBindingParameters(ServiceDescription, ServiceHostBase, Collection<ServiceEndpoint>, BindingParameterCollection)
.ApplyDispatchBehavior(ServiceDescription, ServiceHostBase)
.Validate(ServiceDescription, ServiceHostBase)

- CallbackBehavior
- AspNetCompatibilityRequirementsAttribute
- DurableServiceAttribute
- PersistenceProviderBehavior
- ServiceAuthorizationBehavior
- SericeBehaviorAttribute
- ServiceCredentials
- ServiceDebugBehavior
- ServiceMetadataBehavior
- ServiceSecurityAuditBehavior
- ServiceThrottlingBehavior
- WorkflowRuntimeBehavior

IContractBehavior
.AddBindingParameters(ContractDescription, ServiceEndpoint, BindingParameterCollection)
.ApplyClientBehavior(ContractDescription, ServiceEndpoint, ClientRuntime)
.ApplyDispatchBehavior(ContractDescription, ServiceEndpoint, DispatchRuntime)
.Validate(ContractDescription, ServiceEndpoint)

- DelivereryRequirementsAttribute

IOperationBehavior
.AddBindingParameters(OperationDescription, BindingParameterCollection)
.ApplyClientBehavior(OperationDescription, ClientOperation)
.ApplyDispatchBehavior(OperationDescription, DispatchOperation)
.Validate(OperationDescription)

- DataContractSerializerOperationBehavior
- DurableOperationAttribute
- OperationBehaviorAttribute
- TransactionFlowAttribute
- WebGetAttribute
- WebInvokeAttribute
- XmlSerializerOperationBehavior

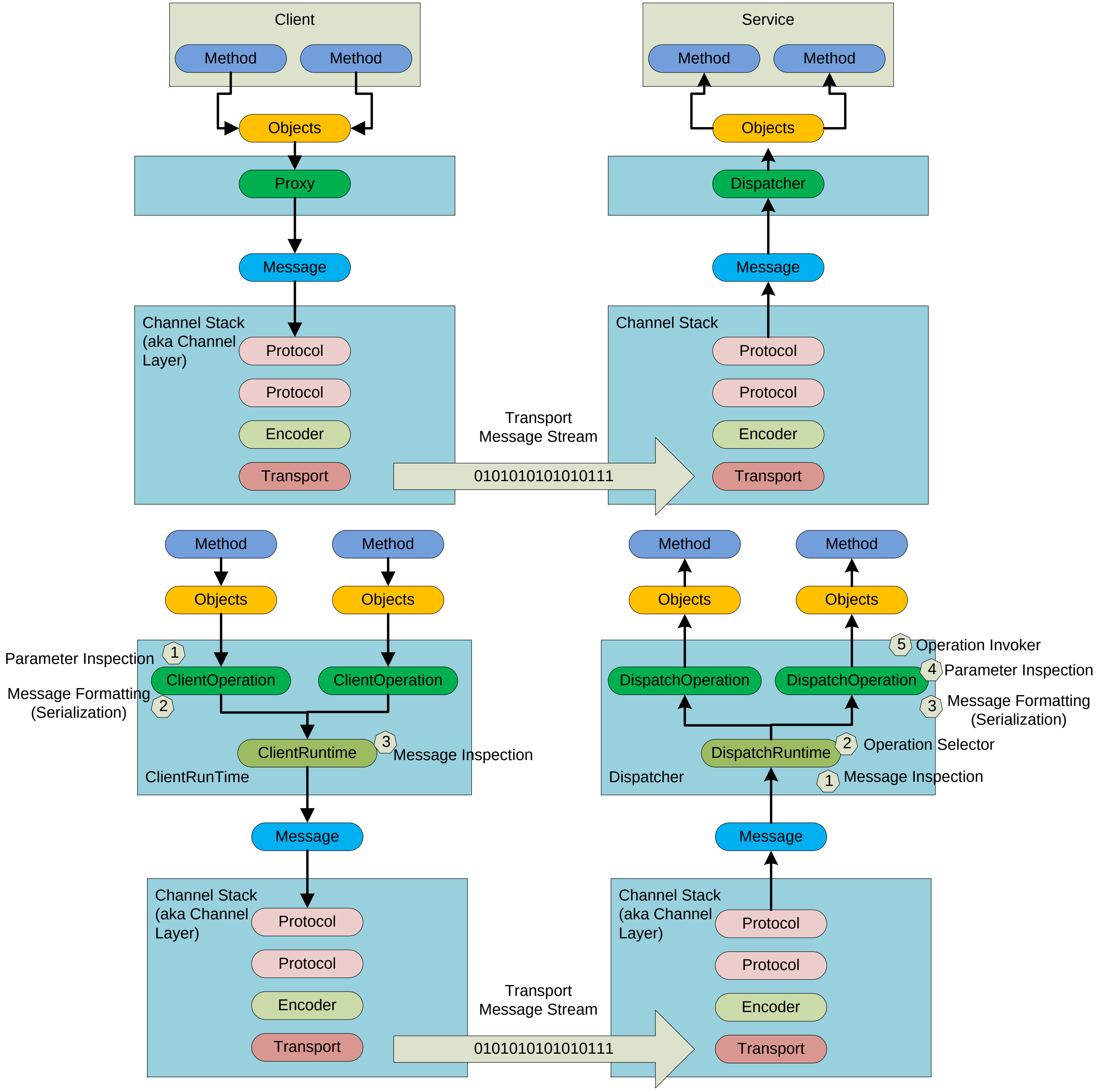
IEndpointBehavior
.AddBindingParameters(ServiceEndpoint, BindingParameterCollection)
.ApplyClientBehavior(ServiceEndpoint, ClientRuntime)
.ApplyDispatchBehavior(ServiceEndpoint, EndpointDipatcher)
.Validate(ServiceEndpoint)

- CallbackBehavior
- CallbackDebugBehavior
- ClientCredentials
- ClientViaBehavior
- MustUnderstandBehavior
- SynchronousReceiveBehavior
- TransactionBatchingBehavior
- WebHttpBehavior

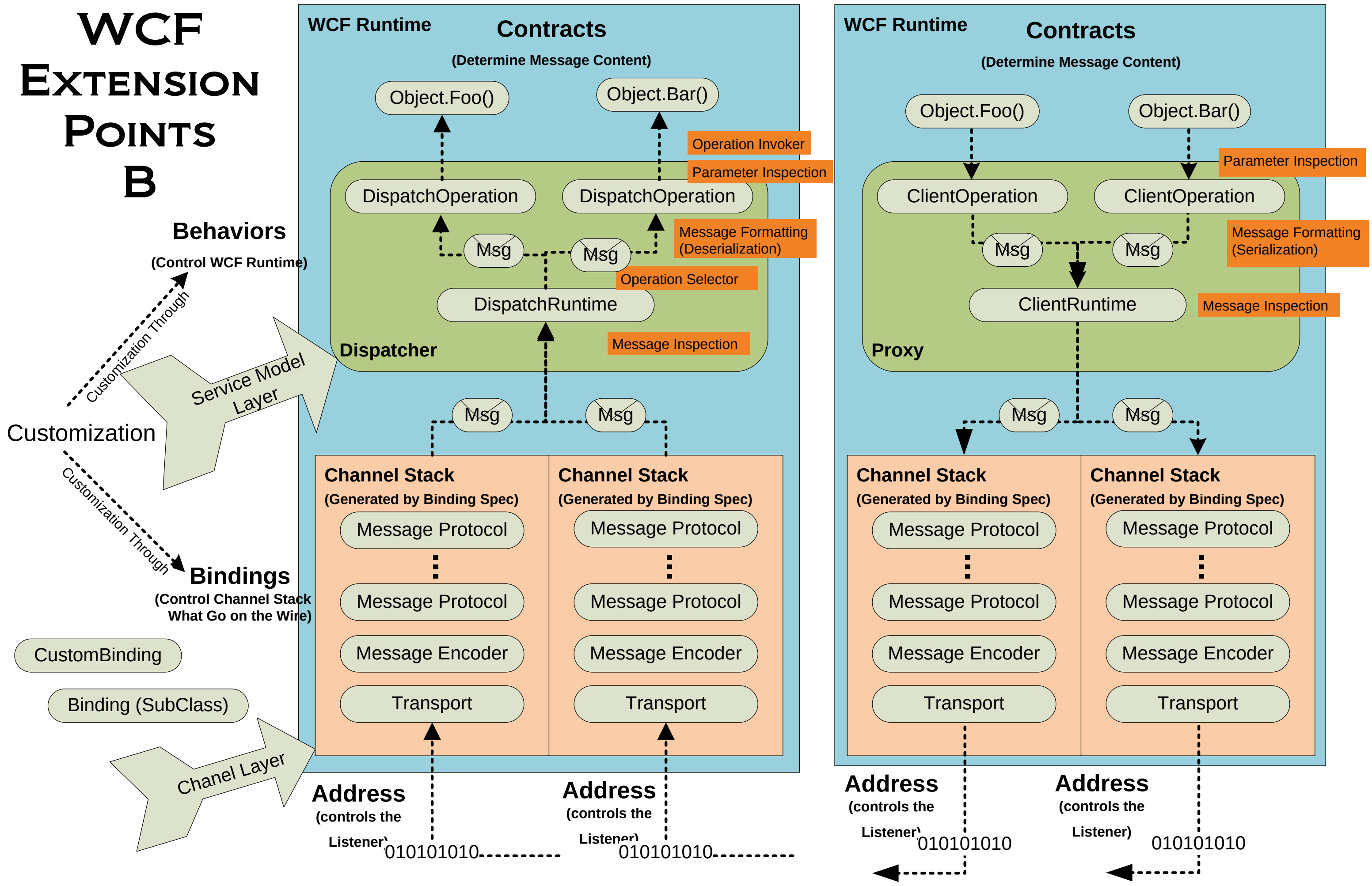
Client Side (Sender) =
Proxy Invocation

WCF RunTime Extension Points & Behavior Customizations A

Service Side (Receiver) =
Dispatcher (Service Dispatching)



WCF Extension Points B



	Interceptor Interface		
Stage	Dispatcher Side	ProxySide	Description
Operation Invoker	IOperationInvoker	N/A	Called to invoke the operation
Parameter Inspection	IParameterInspector	IParameterInspector	Called before and after invocation to inspect and modify parameter values
Message Formatting	IDispatchMessageFormatter	IClientFormatter	Called to perform serialization and deserialization
Operation Selection	IDispatchOperationSelector	IClientOperationSelector	Called to select the operation to invoke for the given message
Message Inspection	IDispatchMessageInspector	IClientMessageInspector	Called after receive or before send to inspect and replace message contents

```
[ServiceContract]
public interface IZipCodeService
{
    [OperationContract]
    string Lookup(string zipcode);
}
```

```
public class ZipCodeInspector : IParameterInspector
{
    int zipCodeParamIndex;
    string zipCodeFormat = @"\d{5}-\d{4}";

    public ZipCodeInspector() : this(0) { }
    public ZipCodeInspector(int zipCodeParamIndex) {
        this.zipCodeParamIndex = zipCodeParamIndex;
    }
    public object BeforeCall(string operationName, object[] inputs) {
        string zipCodeParam = inputs[this.zipCodeParamIndex] as string;
        if (!Regex.IsMatch(zipCodeParam, this.zipCodeFormat, RegexOptions.None))
            throw new FaultException("Invalid zip code format. Required format: #####-####");
        return null;
    }
    ... // AfterCall is empty
}
```

Validate ZipCode format of #####-####

To customize WCF Runtime behavior for Service Dispatching and Client Proxy Invocation, one needs to write Custom Behaviors that can be applied declaratively to the Services and Client.

Credit: http://msdn.microsoft.com/en-us/magazine/cc163302(printer).aspx (Extending WCF with Custom Behaviors)

WCF Extensions Points C (Continued)

```
public class ConsoleMessageTracer : IDispatchMessageInspector, IClientMessageInspector
{
   private Message TraceMessage(MessageBuffer buffer) {
      Message msg = buffer.CreateMessage();
      Console.WriteLine("\n{0}\n", msg);
      return buffer.CreateMessage();
   }
   public object AfterReceiveRequest(ref Message request, IClientChannel channel, InstanceContext instanceContext) {
      request = TraceMessage(request.CreateBufferedCopy(int.MaxValue));
      return null;
   }
   public void BeforeSendReply(ref Message reply, object correlationState) {
      reply = TraceMessage(reply.CreateBufferedCopy(int.MaxValue));
   }

   public void AfterReceiveReply(ref Message reply, object correlationState) {
      reply = TraceMessage(reply.CreateBufferedCopy(int.MaxValue));
   }
   public object BeforeSendRequest(ref Message request,  IClientChannel channel) {
      request = TraceMessage(request.CreateBufferedCopy(int.MaxValue));
      return null;
   }
}
```

```
IDispatchMessageInspector:
    AfterReceiveRequest();
    BeforeSendReply();
```

```
IClientMessageInspector:
    AfterReceiveReply();
    BeforeSendRequest();
```

```
public class ZipCodeCacher : IOperationInvoker
{
   IOperationInvoker innerOperationInvoker;
   Dictionary<string, string> zipCodeCache = new Dictionary<string, string>();

   public ZipCodeCacher(IOperationInvoker innerOperationInvoker) {
      this.innerOperationInvoker = innerOperationInvoker;
   }

   public object Invoke(object instance, object[] inputs, out object[] outputs) {
      string zipcode = inputs[0] as string;
      string value;

      if (this.zipCodeCache.TryGetValue(zipcode, out value)) {
         outputs = new object[0];
         return value;
      } else {
         value = (string)this.innerOperationInvoker.Invoke(instance, inputs, out outputs);
         zipCodeCache[zipcode] = value;
         return value;
      }
   }
   ...
   // remaining methods elided
   // they simply delegate to innerOperationInvoker
}
```

WCF Extensions Via Beahviors D

Scope	Interface	Service (Service OnlY)	Endpoint (Client & Service)	Contract (Client & Service)	Operation (Client & Service)	Validate()	AddingBindingParameters()	ApplyClientBehavior()	ApplyDispatchBehavior()
Service	IServiceBehavior	X	X	X	X	Y	Y		Y
Endpoint	IEndpoitnBehavior		X	X	X	Y	Y	Y	Y
Contract	IContractBehavior			X	X	Y	Y	Y	Y
Operation	IOperationBehavior				X	Y	Y	Y	Y
		Potential Impact							

```
public class ZipCodeInspector : IParameterInspector {
    int zipCodeParamIndex;
    string zipCodeFormat = @"\d{5}-\d{4}";
    public ZipCodeInspector() : this(0) { }
    public ZipCodeInspector(int zipCodeParamIndex)
        {this.zipCodeParamIndex = zipCodeParamIndex;}
    public object BeforeCall(string operationName, object[] inputs) {
        string zipCodeParam = inputs[this.zipCodeParamIndex] as string;
        if (!Regex.IsMatch(zipCodeParam, this.zipCodeFormat, RegexOptions.None))
            throw new FaultException("
                    Invalid zip code format. Required format: #####-####");
        return null;
    }
    ... // AfterCall is empty
}
```

```
public class ZipCodeCacher : IOperationInvoker {
    IOperationInvoker innerOperationInvoker;
    Dictionary<string, string> zipCodeCache = new Dictionary<string, string>();
    public ZipCodeCacher(IOperationInvoker innerOperationInvoker) {
        this.innerOperationInvoker = innerOperationInvoker;
    }
    public object Invoke(object instance, object[] inputs, out object[] outputs) {
        string zipcode = inputs[0] as string;
        string value;
        if (this.zipCodeCache.TryGetValue(zipcode, out value)) {
            outputs = new object[0];
            return value;
        } else {
            value = (string)this.innerOperationInvoker.Invoke(instance, inputs, out outputs);
            zipCodeCache[zipcode] = value;
            return value;
        }
    }
    // remaining methods elided
    // they simply delegate to innerOperationInvoker
}
```

```
public class ZipCodeValidation : Attribute, IOperationBehavior{
    public void ApplyClientBehavior(OperationDescription operationDescription,
                                    ClientOperation clientOperation) {
        clientOperation.ParameterInspectors.Add(new ZipCodeInspector());
    }
    public void ApplyDispatchBehavior(OperationDescription operationDescription,
                                      DispatchOperation dispatchOperation) {
        dispatchOperation.ParameterInspectors.Add(new ZipCodeInspector());
    }
    ... // remaining methods empty
}
public class ZipCodeCaching : Attribute, IOperationBehavior{
    public void ApplyDispatchBehavior(OperationDescription operationDescription,
                                      DispatchOperation dispatchOperation) {
        dispatchOperation.Invoker = new ZipCodeCacher(dispatchOperation.Invoker);
    }
    ... // remaining methods empty
}
```

A behavior is a special type of class that extends runtime behavior during the ServiceHost (Service side) or ChannelFactory (Client side) initialization process.

Four (4) types of behaviors, applying behaviors at four corresponding scopes:

= Service
= Contract
= Endpoint
= Operation

Like Extension Points, Behaviors are also modeled by different Interfaces that share the same set of methods (with different signatures, though) with an exception that IServiceBehavior does not have an ApplyClientBehavior() method.

The set of methods in these different Interfaces is:

= Validate()
= AddingBindingParameters()
= ApplyClientBehavior()
= ApplyDispatchBehavior()

ApplyDispatchBehavior() and ApplyClientBehavior() are core methods that are used to insert custom extensions into the Dispatcher and Proxy, respectively. When WCF Runtime calls them, it provides the DispatchRuntime and DispatchOperation or ClientRuntime and ClientOperation objects to which Runtime Extensions can be injected into.

Take away

[A Behavior in X Scope insert a Runtime Extension at Y Stage]

X --- Could be any-one-of or combination-of Service, Endpoint, Contract or Operation (Scopes).

Y --- Could be any-one-of or combination-of Extension Points (Stages).

WCF Extensions Via Beahviors E (more examples)

```
public class ConsoleMessageTracer : IDispatchMessageInspector, IClientMessageInspector
{
    private Message TraceMessage(MessageBuffer buffer) {
        Message msg = buffer.CreateMessage();
        Console.WriteLine("\n{0}\n", msg);
        return buffer.CreateMessage();
    }
    public object AfterReceiveRequest(ref Message request, IClientChannel channel, InstanceContext instanceContext) {
        request = TraceMessage(request.CreateBufferedCopy(int.MaxValue));
        return null;
    }
    public void BeforeSendReply(ref Message reply, object correlationState) {
        reply = TraceMessage(reply.CreateBufferedCopy(int.MaxValue));
    }

    public void AfterReceiveReply(ref Message reply, object correlationState) {
        reply = TraceMessage(reply.CreateBufferedCopy(int.MaxValue));
    }
    public object BeforeSendRequest(ref Message request,  IClientChannel channel) {
        request = TraceMessage(request.CreateBufferedCopy(int.MaxValue));
        return null;
    }
}
```

```
public class ConsoleMessageTracing : Attribute, IEndpointBehavior, IServiceBehavior
{
    void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint endpoint, ClientRuntime clientRuntime) {
        clientRuntime.MessageInspectors.Add(new ConsoleMessageTracer());
    }
    void IEndpointBehavior.ApplyDispatchBehavior(ServiceEndpoint endpoint, EndpointDispatcher endpointDispatcher) {
        endpointDispatcher.DispatchRuntime.MessageInspectors.Add(new ConsoleMessageTracer());
    }

    ... // remaining methods empty

    void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription desc, ServiceHostBase host) {
        foreach (ChannelDispatcher cDispatcher in host.ChannelDispatchers)
            foreach (EndpointDispatcher eDispatcher in cDispatcher.Endpoints)
                eDispatcher.DispatchRuntime.MessageInspectors.Add(new ConsoleMessageTracer());
    }
    ... // remaining methods empty
}
```

IServiceBehavior:

```
AddBindingParameters(ServiceDescription, ServiceHostBase, Collection<ServiceEndpoint>, BindingParameterCollection)
ApplyDispatchBehavior(ServiceDescription, ServiceHostBase)
Validate(ServiceDescription, ServiceHostBase)
```

IContractBehavior:

```
AddBindingParameters(ContractDescription, ServiceEndpoint, BindingParameterCollection)
ApplyClientBehavior(ContractDescription, ServiceEndpoint, ClientRuntime)
ApplyDispatchBehavior(ContractDescription, ServiceEndpoint, DispatchRuntime)
Validate(ContractDescription, ServiceEndpoint)
```

IEndpointBehavior:

```
AddBindingParameters(ServiceEndpoint, BindingParameterCollection)
ApplyClientBehavior(ServiceEndpoint, ClientRuntime)
ApplyDispatchBehavior(ServiceEndpoint, EndpointDispatcher)
Validate(ServiceEndpoint)
```

IOperationBehavior:

```
AddBindingParameters(OperationDescription, BindingParameterCollection)
ApplyClientBehavior(OperationDescription, ClientOperation)
ApplyDispatchBehavior(OperationDescription, DispatchOperation)
Validate(OperationDescription)
```

Behavior Interfaces Signatures

WCF Runtime is Generated!

WCF Runtime is generated according to specifications of Service Endpoints on both sides of Service and Client:

On Service Side, it consists of the Channel Listener Stack (to create IInputChannel and IReplyChannel, …), the Dispatcher [ChannelDispatcher + EndpointDipstacher + DispatchRuntime + DispatchOperation].

On Client Side, it consists of Channel Factory Stack (to create IOutputChannel and IRequestChannel, …), ClientRuntime and ClientOperation.

When ServiceHost (Service Side) or ChannelFactory (Client Side) is constructed, the WCF Runtime reflects over the service types, reads the configuration file, and starts building an in-memory description of the service. Within the ServiceHost, this service description is accessible via the host.**Description** property of type ServiceDescription [host = new ServiceHost(..)]. Within the ChannelFactory, it is accessible via the factory.**Endpoint** property of type ServiceEndpoint [factory = ChannelFactory<IContract>.CreateFactory(..)]. **Note**: The Client Side description is limited to one target endpoint to which it wants to talk.

Starting from the ServiceDescription, ServiceEndpoint, ContractDescription, and OperationDescription can be found.

ServiceDescription.Behaviors property (a collection of IServiceBehavior) that models a collection of Service Behaviors.
ServiceEndpoint.Behaviors property (a collection of IEndpointBehavior) that modeles a collection of Endpoint Behaviors.
ContractDescription.Behaviors property (a collection of IContractBehavior) that models a collection of Contract Behaviors.
OperationDecription.Behaviors property (a collection of IOperationBehavior) that models a collection of Operation Behaviors.

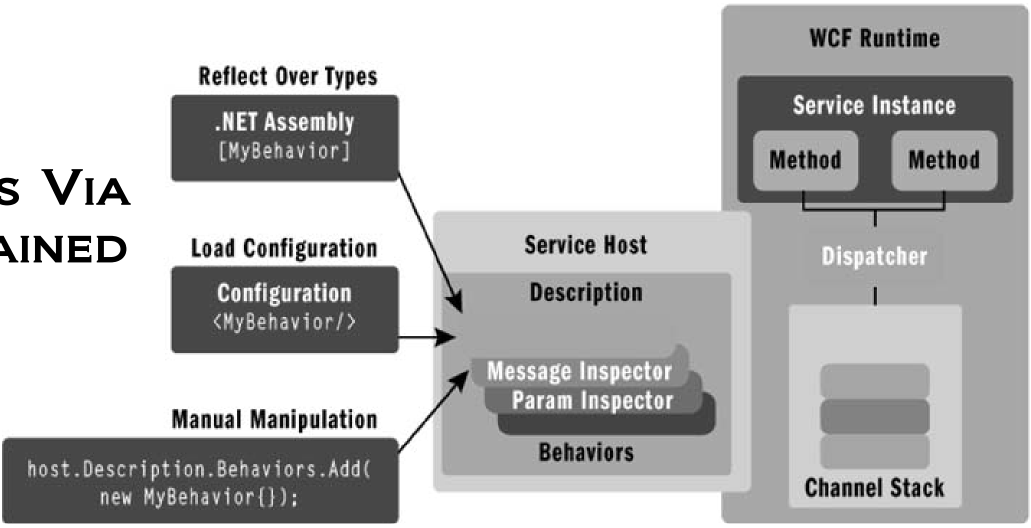
Inserting Runtime Extensions into WCF Runtime via Behaviors

During the ServiceHost and ChannelFactory construction process, these behavior collections are automatically populated with any behaviors that are found in the code (via Attributes) or within the configuration file. Behaviors can be added to these collections manually afterwards, before serviceHost.Open() or channelFactory.CreateChannel() is called.

Then, the Magic Happens: once you open the ServiceHost's Open() (via ICommunicationObject.Open) or ChannelFactory's CreateChannel(), the WCF runtime walks through the ServiceDescription and gives each behavior a chance to insert its dispatcher or proxy extensions by calling ApplyDispatchBehavior() or ApplyClientBehavior() with appropriate parameters.

Note: **Once this process is complete, you can NOT add additional behaviors or extensions to the WCF runtime.**

WCF Extensions Via Beahviors Explained F



Adding Behaviors by Attributes

```
[ServiceContract] public interface IZipCodeService {
    [ZipCodeCaching] [ZipCodeValidation]
    [OperationContract] string Lookup(string zipcode);
}
[ConsoleMessageTracing]
public class ZipCodeService : IZipCodeService
{  ...}
```

Contract behavior attributes can be applied to service contract interfaces or to the service class. When applied to a service class, you may want to restrict the contract behavior to take affect only when an endpoint uses a particular contract. You can control this by implementing IContractBehaviorAttribute on your contract behavior attribute class and specifying the desired contract via the **TargetContract** property.

Adding Behaviors Manually

```
// A Behavior---ConsoleMessageTracing---is added to host's Service
// Behavior collection after ServiceHost  is constructed.
ServiceHost host = new ServiceHost(typeof(ZipCodeService));
host.Description.Behaviors.Add(new ConsoleMessageTracing());
```

```
// A Behavior---ConsoleMessageTracing---is added to host's Endpoint
// Behavior collection after ServiceHost is constructed.
ServiceHost host = new ServiceHost(typeof(ZipCodeService));
foreach (ServiceEndpoint se in host.Description.Endpoints)
    se.Behaviors.Add(new ConsoleMessageTracing());
```

```
// A Behavior---ConsoleMessageTracing---is added to Client's Service
// Endpoint Behavior Collection after ClientProxy is created
ZipCodeServiceClient client = new ZipCodeServiceClient();
client.ChannelFactory.Endpoint.Behaviors.Add(
    new ConsoleMessageTracing());
```

WCF Extensions Via Beahviors G

After the reflection process is complete, the runtime also inspects the application configuration file and loads the information found in the <system.serviceModel> section into the ServiceDescription. WCF provides a <behaviors> section for configuring service and endpoint behaviors. Any service/endpoint behaviors found in this section are automatically added to the ServiceDescription.
In order to place custom behaviors within this configuration section, you must first write a class that derives from BehaviorElementExtension, like this one:

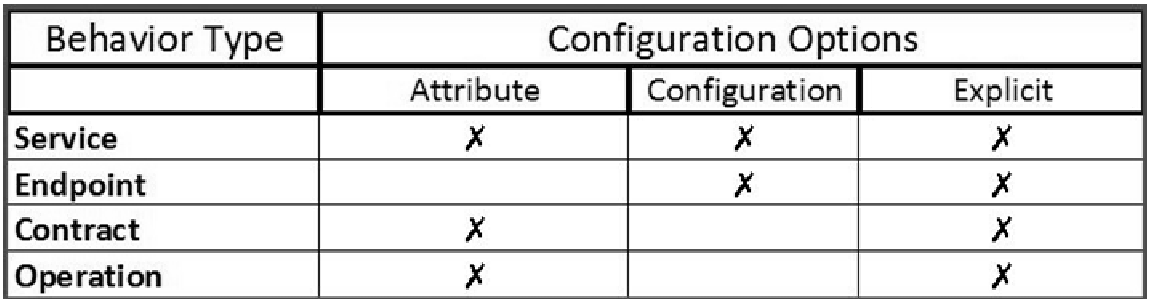
```
public class ConsoleMessageTracingElement : BehaviorExtensionElement
{
   public override Type BehaviorType {
      get { return typeof(ConsoleMessageTracing); }
   }
   protected override object CreateBehavior(){
      return new ConsoleMessageTracing();
   }
}
```

Then you must register your BehaviorExtensionElement in the <extensions> section and map it to an element name. With that in place, you can use your registered element name within the <behaviors> section in order to configure the behavior.

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ZipCodeServiceLibrary.ZipCodeService"   behaviorConfiguration="Default">
        <endpoint binding="basicHttpBinding" contract="ZipCodeServiceLibrary.IZipCodeService"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="Default">
          <serviceMetadata httpGetEnabled="true"/>
          <consoleMessageTracing/>
        </behavior>
      </serviceBehaviors>
      <endpointBehaviors>
        …
      </endpointBehaviors>
    </behaviors>
    <extensions>
      <behaviorExtensions>
        <add name="consoleMessageTracing" type="Extensions.ConsoleMessageTracingElement, Extensions,
                                                              Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null"/>
      </behaviorExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

Caveats

You can add service, contract, or operation behaviors using attributes, but you can't use them to add endpoint behaviors. You can add service and endpoint behaviors via the configuration file, but you can't use it to add contract or operation behaviors. Finally, you can manually add any type of behavior to the **ServiceDescription**.



Behavior Type	Configuration Options		
	Attribute	Configuration	Explicit
Service	✗	✗	✗
Endpoint		✗	✗
Contract	✗		✗
Operation	✗		✗

Behavior Validation and Binding Configuration

In addition to adding custom runtime extensions, behaviors are also designed to let you perform two additional tasks: custom validation and binding configuration. The **Validate()** method gives you a chance to perform custom validation on the ServiceDescription after it has been initialized but before the rest of the runtime has been built. This is your chance to traverse the ServiceDescription tree (or ServiceEndpoint on the client side) and validate it against your own criteria. If something doesn't meet your requirements, you can throw an exception to prevent the ServiceHost/ChannelFactory from opening.

The following service behavior validates the ServiceDescription to ensure that no endpoints use BasicHttpBinding:

```
public class NoBasicEndpointValidator : Attribute, IServiceBehavior
{
    public void Validate(ServiceDescription desc, ServiceHostBase host) {
        foreach (ServiceEndpoint se in desc.Endpoints)
            if (se.Binding.Name.Equals("BasicHttpBinding"))
                throw new FaultException("BasicHttpBinding is not allowed");
    }
    ... //remaining methods empty
}
```

WCF Extensions Via Beahviors H

AddBindingParameters() gives you a chance to add additional binding parameters during runtime initialization. The binding parameters are supplied to the underlying channel layer in order to influence the creation of the channel stacks. Custom binding elements have access to these binding parameters and can be designed to look for them

Sharing State Between Extensions

Once you begin employing multiple extensions within the dispatcher/proxy, you will need to learn how to share state across them. Thankfully, WCF provides extension objects that can be used to store user-defined state.

Where you store an extension object determines how long it will stick around. You can store it globally on the ServiceHost, on an InstanceContext, or on an OperationContext. Each of these classes provides an Extensions collection that manages objects derived from IExtension<T> (where T is ServiceHostBase, InstanceContext, or OperationContext, depending on the collection). ServiceHost extension objects remain in memory for the lifetime of the ServiceHost while InstanceContext and OperationContext extension objects only remain in memory for the lifetime of the service instance or operation invocation. Your custom dispatcher/proxy extensions can use these collections to store (and look up) user-defined state throughout the pipeline.

Final Words

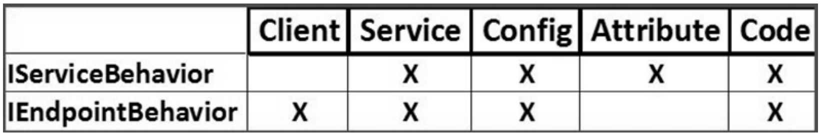
WCF provides a powerful extensibility architecture that allows for significant runtime customizations. It provides some key extensibility stages throughout the dispatcher/proxy for performing tasks such as parameter inspection, message formatting, message inspection, operation selection, and invocation. You write these custom extensions by implementing the appropriate extension interface, and then you can apply your extension to the dispatcher/proxy via a custom behavior.

There are some more advanced extensibility points made available on the dispatcher . They deal with matters such as **instancing**, **concurrency**, **addressing**, and, of course, **security**. Although the built-in [ServiceBehavior] and [OperationBehavior] behaviors do supply most of what you'll need in these areas, you can write custom behaviors to extend those aspects of the runtime when they don't provide everything you need.

Tips: IServiceBehavior vs IEndpointBehavior

1. ServiceBehavior applies only on service while EndpointBehavior applies on both **client and service**.
2. ServiceBehavior can be specified via config/attribute/code while endpointbehavior can be specified via config/code.
3. ServiceBehavior has access to all ServiceEndpoints dispatch runtime and so could modify all dispatch runtimes while Endpointbehavior gets called with the runtime for that endpoint only.

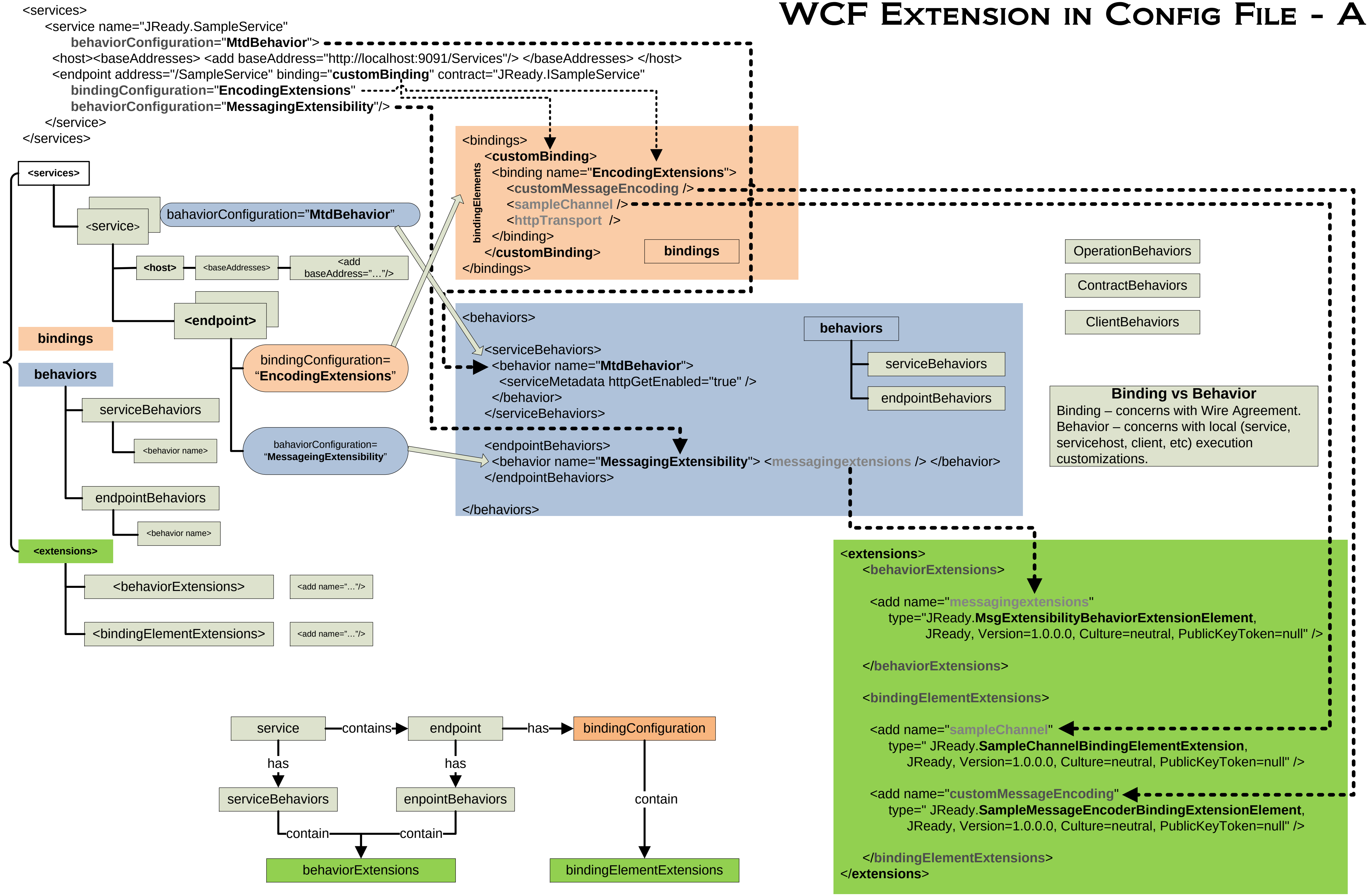
Look at it this way, ServiceBehavior lets you access runtime parameters for all endpoints while Endpointbehavior lets you access runtime components only for that endpoint. So if you have a need to extend functionality that spawns the entire contract (or multiple contracts) then use ServiceBehavior and if you are interested in extending one specific endpoint then use Endpointbehavior.



	Client	Service	Config	Attribute	Code
IServiceBehavior		X	X	X	X
IEndpointBehavior	X	X	X		X

When Open() is called on a ServiceHost, the runtime is built from the ServiceDescription and the list of ServiceEndpoints specified. Once the Open() step completes, the runtime is immutable and any attempt to modify any components results in an exception.

WCF Extension in Config File - A



```
<services>
    <service name="JReady.SampleService"
        behaviorConfiguration="MtdBehavior">
     <host><baseAddresses> <add baseAddress="http://localhost:9091/Services"/> </baseAddresses> </host>
     <endpoint address="/SampleService" binding="customBinding" contract="JReady.ISampleService"
        bindingConfiguration="EncodingExtensions"
        behaviorConfiguration="MessagingExtensibility"/>
    </service>
</services>
```

```
<bindings>
   <customBinding>
    <binding name="EncodingExtensions">
      <customMessageEncoding />
      <sampleChannel />
      <httpTransport  />
    </binding>
   </customBinding>
</bindings>
```

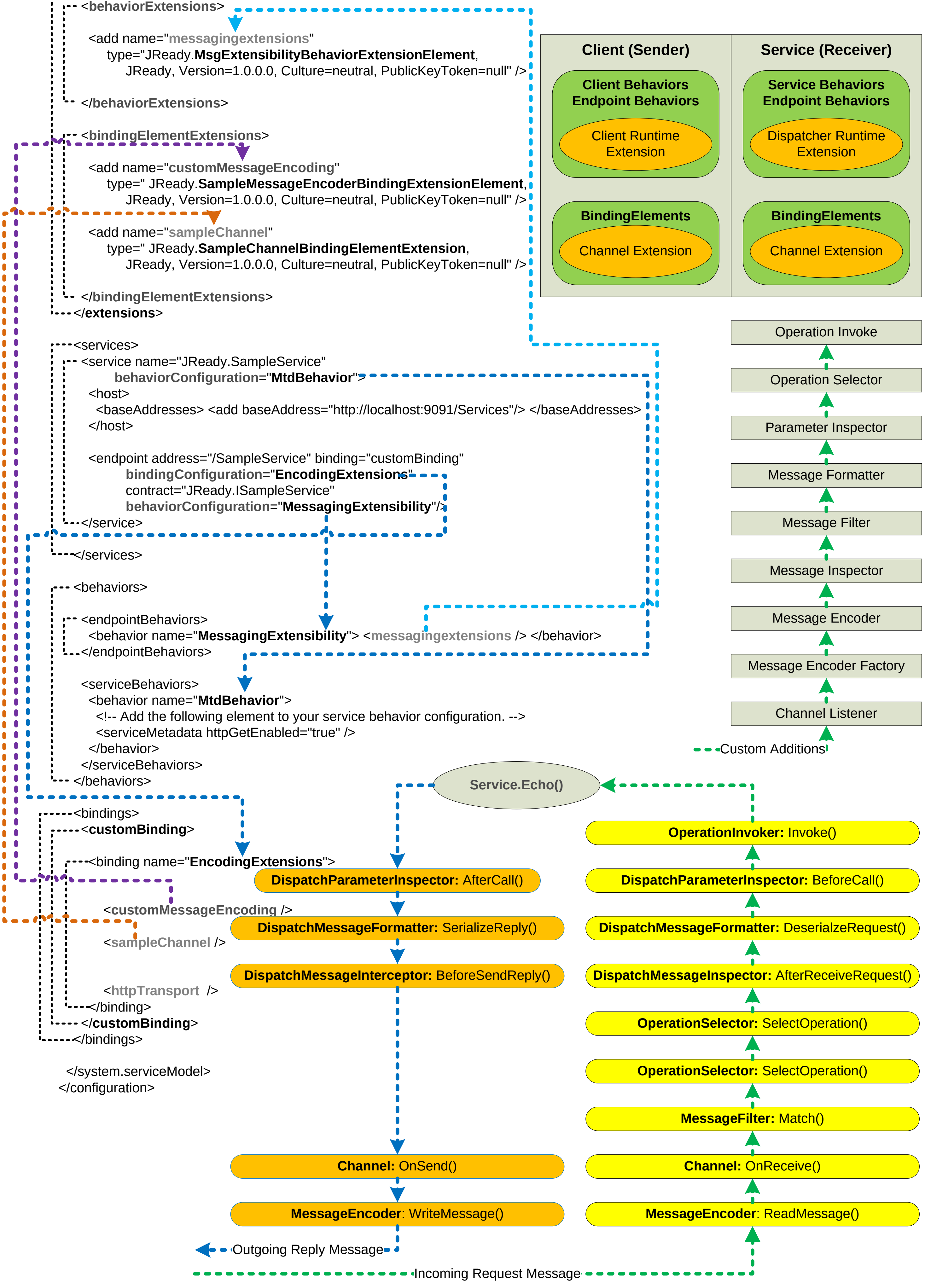
```
<behaviors>
   <serviceBehaviors>
    <behavior name="MtdBehavior">
      <serviceMetadata httpGetEnabled="true" />
    </behavior>
   </serviceBehaviors>

   <endpointBehaviors>
    <behavior name="MessagingExtensibility"> <messagingextensions /> </behavior>
   </endpointBehaviors>

</behaviors>
```

```
<extensions>
   <behaviorExtensions>

    <add name="messagingextensions"
       type="JReady.MsgExtensibilityBehaviorExtensionElement,
              JReady, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

   </behaviorExtensions>

   <bindingElementExtensions>

    <add name="sampleChannel"
       type=" JReady.SampleChannelBindingElementExtension,
          JReady, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

    <add name="customMessageEncoding"
       type=" JReady.SampleMessageEncoderBindingExtensionElement,
          JReady, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

   </bindingElementExtensions>
</extensions>
```

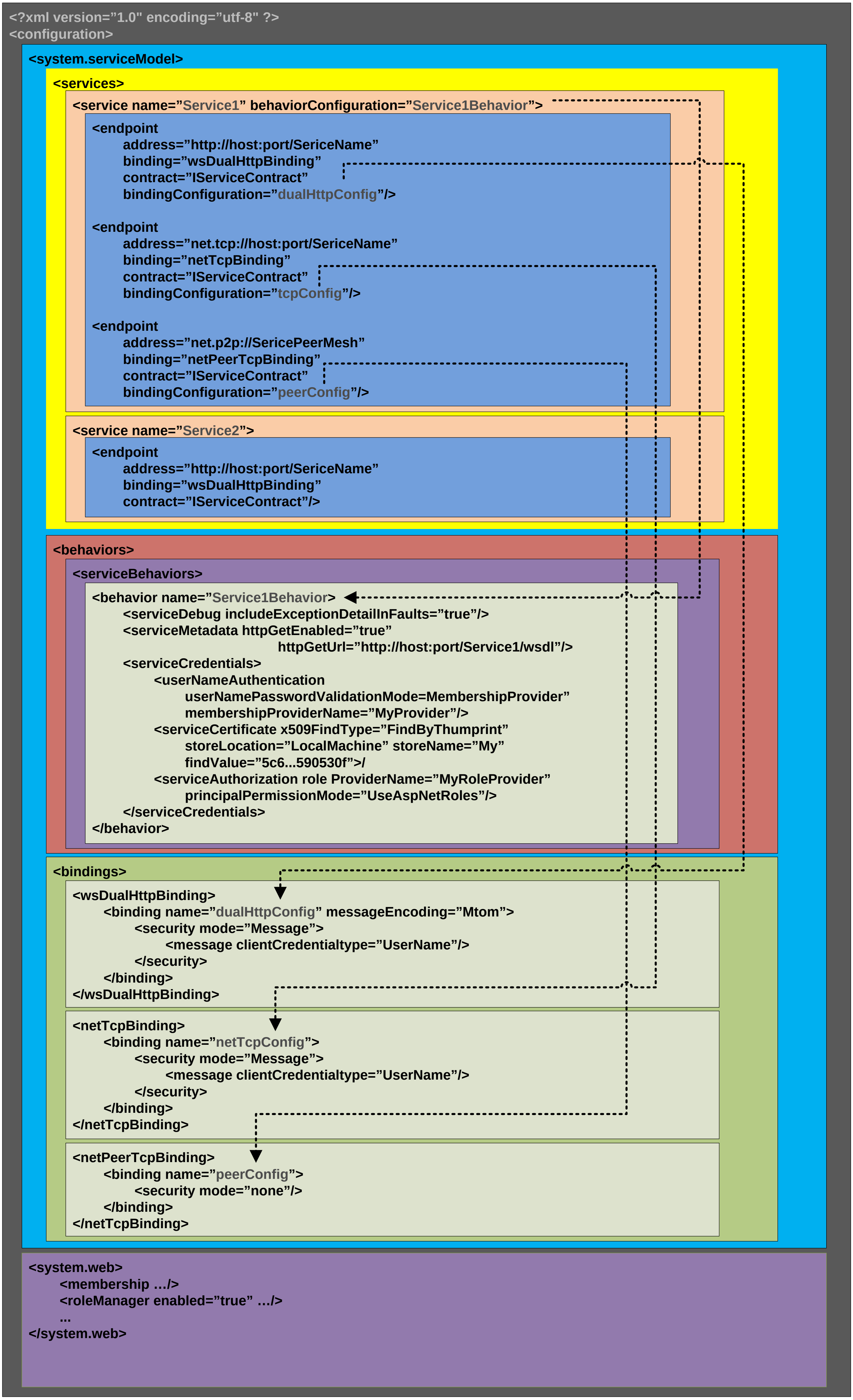
Binding vs Behavior
Binding – concerns with Wire Agreement.
Behavior – concerns with local (service, servicehost, client, etc) execution customizations.

WCF Extension Points in config - B

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>

        <add name="messagingextensions"
          type="JReady.MsgExtensibilityBehaviorExtensionElement,
            JReady, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

      </behaviorExtensions>

      <bindingElementExtensions>

        <add name="customMessageEncoding"
          type=" JReady.SampleMessageEncoderBindingExtensionElement,
            JReady, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

        <add name="sampleChannel"
          type=" JReady.SampleChannelBindingElementExtension,
            JReady, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

      </bindingElementExtensions>
    </extensions>

    <services>
      <service name="JReady.SampleService"
          behaviorConfiguration="MtdBehavior">
        <host>
          <baseAddresses> <add baseAddress="http://localhost:9091/Services"/> </baseAddresses>
        </host>

        <endpoint address="/SampleService" binding="customBinding"
            bindingConfiguration="EncodingExtensions"
            contract="JReady.ISampleService"
            behaviorConfiguration="MessagingExtensibility"/>
      </service>
    </services>

    <behaviors>

      <endpointBehaviors>
        <behavior name="MessagingExtensibility"> <messagingextensions /> </behavior>
      </endpointBehaviors>

      <serviceBehaviors>
        <behavior name="MtdBehavior">
          <!-- Add the following element to your service behavior configuration. -->
          <serviceMetadata httpGetEnabled="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    <bindings>
      <customBinding>
        <binding name="EncodingExtensions">

          <customMessageEncoding />

          <sampleChannel />


          <httpTransport  />
        </binding>
      </customBinding>
    </bindings>

  </system.serviceModel>
</configuration>
```



App.Config Example



```
<?xml version=”1.0" encoding=”utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name=”Service1” behaviorConfiguration=”Service1Behavior”>
        <endpoint
              address=”http://host:port/SericeName”
              binding=”wsDualHttpBinding”
              contract=”IServiceContract”
              bindingConfiguration=”dualHttpConfig”/>

        <endpoint
              address=”net.tcp://host:port/SericeName”
              binding=”netTcpBinding”
              contract=”IServiceContract”
              bindingConfiguration=”tcpConfig”/>

        <endpoint
              address=”net.p2p://SericePeerMesh”
              binding=”netPeerTcpBinding”
              contract=”IServiceContract”
              bindingConfiguration=”peerConfig”/>
      <service name=”Service2”>
        <endpoint
              address=”http://host:port/SericeName”
              binding=”wsDualHttpBinding”
              contract=”IServiceContract”/>
    <behaviors>
      <serviceBehaviors>
        <behavior name=”Service1Behavior>
              <serviceDebug includeExceptionDetailInFaults=”true”/>
              <serviceMetadata httpGetEnabled=”true”
                                        httpGetUrl=”http://host:port/Service1/wsdl”/>
              <serviceCredentials>
                    <userNameAuthentication
                          userNamePasswordValidationMode=MembershipProvider”
                          membershipProviderName=”MyProvider”/>
                    <serviceCertificate x509FindType=”FindByThumprint”
                          storeLocation=”LocalMachine” storeName=”My”
                          findValue=”5c6...590530f”>/
                    <serviceAuthorization role ProviderName=”MyRoleProvider”
                          principalPermissionMode=”UseAspNetRoles”/>
              </serviceCredentials>
        </behavior>
    <bindings>
      <wsDualHttpBinding>
            <binding name=”dualHttpConfig” messageEncoding=”Mtom”>
                  <security mode=”Message”>
                        <message clientCredentialtype=”UserName”/>
                  </security>
            </binding>
      </wsDualHttpBinding>
      <netTcpBinding>
            <binding name=”netTcpConfig”>
                  <security mode=”Message”>
                        <message clientCredentialtype=”UserName”/>
                  </security>
            </binding>
      </netTcpBinding>
      <netPeerTcpBinding>
            <binding name=”peerConfig”>
                  <security mode=”none”/>
            </binding>
      </netTcpBinding>
  <system.web>
        <membership .../>
        <roleManager enabled=”true” .../>
        ...
  </system.web>
```

The **EndpointDispatcher** and the System.ServiceModel.Dispatcher.**DispatchRuntime** classes expose the runtime customization points for endpoints in a service. The **EndpointDispatcher** ccontrols which messages it can process. The **DispatchRuntime** has a large number of properties used to insert custom extensions into the **endpoint-wide runtime**.

The **EndpointDispatcher** object is responsible for processing messages from a System.ServiceModel.Dispatcher.**ChannelDispatcher** when the destination address of a message matches the **AddressFilter** property and the **message action** matches the **ContractFilter** property. If two **EndpointDispatcher** objects can accept a message, the **FilterPriority** property value determines the higher priority endpoint.

Use the **EndpointDispatcher** object to configure or extend the process of receiving messages from the associated **ChannelDispatcher**, converting from message objects to objects used as parameters, and invoking an endpoint operation; same can be said for the reverse process.

Typically, the EndpointDispatcher for an endpoint is obtained by implementing the IEndpointBehavior interface, but you can access the EndpointDispatcher from the other behavior interfaces.

You can use the following **EndpointDispatcher** properties:

The **AddressFilter** property allows you to get or set a **MessageFilter** object that the **ChannelDispatcher** uses to identify whether the endpoint can process a particular message.

The **ChannelDispatcher** property gets the associated **ChannelDispatcher** object, which sends and receives messages to and from the **EndpointDispatcher** and which can be used to inspect or modify other channel-related values and behaviors.

The **ContractFilter** gets the **MessageFilter** object that is used to identify whether a message is destined for this contract.

The **ContractName** and **ContractNamespace** properties return the name and namespace of the endpoint contract.

The **DispatchRuntime** property returns the **DispatchRuntime** object that you can use to modify run-time values or insert custom run-time extensions for the entire endpoint.

The **EndpointAddress** property gets the address of the endpoint.

The **FilterPriority** property returns the priority of the composite filter that the **ChannelDispatcher** uses to establish which endpoint is to handle the message.

How ServiceModel (aka WCF RunTime) Work Flow

DEFAULT ACTION

Action for your [OperationContract]

Question: What is the “Action” header for my OperationContract?

Answer: Two ways to specify it:

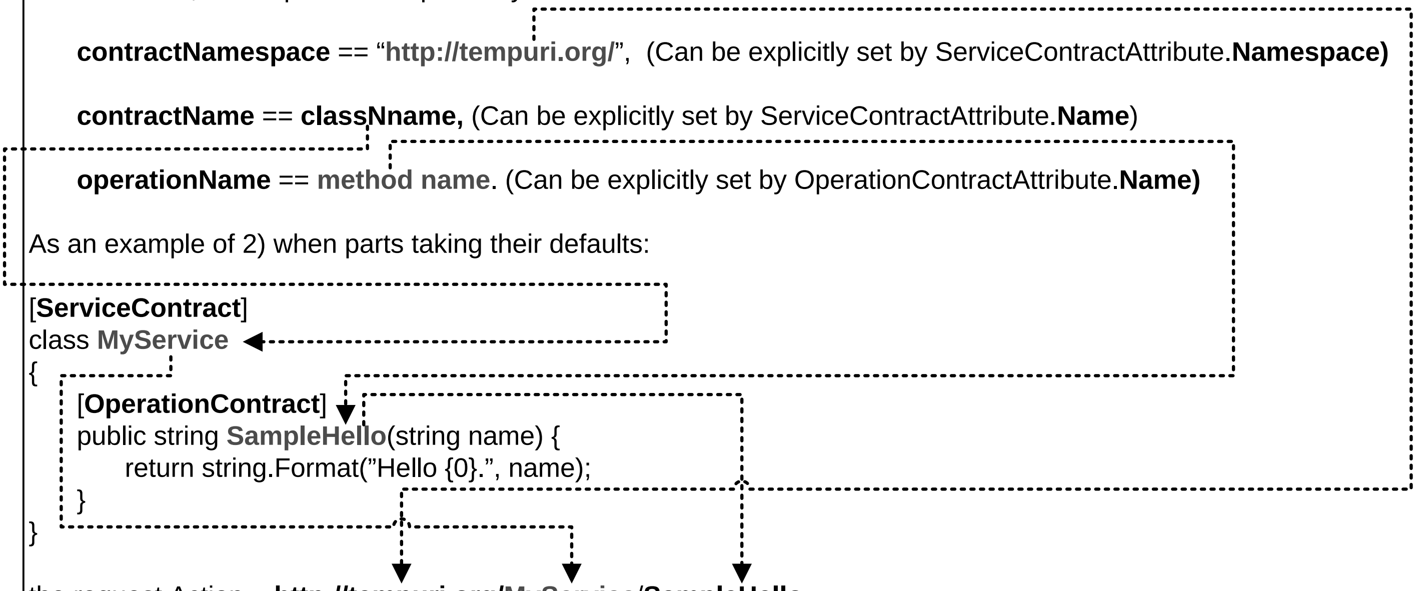
1) Explicitly:

[OperationContract(Action="**myAction**", ReplyAction="**myReplyAction**")]

2) Implicitly:

Action = **contractNamespace** + “/” +**contractName** + “/” + **operationName**.
ReplyAction = **contractNamespace** + “/” + **contractName** + “/” + **operationName + “Response”**

Their defaults, if not specified respectively:



contractNamespace == “**http://tempuri.org/**”, (Can be explicitly set by ServiceContractAttribute.**Namespace)**

contractName == **classNname,** (Can be explicitly set by ServiceContractAttribute.**Name**)

operationName == **method name**. (Can be explicitly set by OperationContractAttribute.**Name)**

As an example of 2) when parts taking their defaults:

```
[ServiceContract]
class MyService
{
    [OperationContract]
    public string SampleHello(string name) {
        return string.Format(”Hello {0}.”, name);
    }
}
```

the request Action = **http://tempuri.org/MyService/SampleHello**,
response ReplyAction = http://tempuri.org/MyService/**SampleHelloResponse**.

Credit: http://kennyw.com/work/indigo/78

OperationContext

OperationContext

- .Channel (IContextChannel)
- .Current (OperationContext)
- .EndpointDispatcher (*)
  - .AddressFilter (MessageFilter)
  - .ChannelDispatcher(*)
  - .ContractFilter (MessageFilter)
  - .ContractName (string)
  - .ContractNamespace (string)
  - .DispatchRuntime (*)
  - .EndpointAddress (*)
    - .AnonymousUri
    - .Headers (AddressHeaderColllection)
    - .Identity (EndpointIdentity)
    - .IsAnonymous
    - .IsNone
    - .NoneUri
    - .Uri
  - .FilterPriority (int)
- .Extensions (IExtesionCollection)
- .HasSupportingTokens (bool)
- .Host (ServiceHostBase)
- .IncomingMessageHeaders (MessageHeaders)
  - .Action (string)
  - .Count (int)
  - .FaultTo (EndpointAddress)
  - .From (EndpointAddress)
  - .MessageId (UniqueId)
  - .MessageVersion (*)
  - .RelatesTo (UniqueId)
  - .ReplyTo (EndpointAddress)
  - .this[int]
  - .To (Uri)
  - .UnderstoodHeaders (*)
- .IncomingMessageProperties (MessageProperties)
  - .AllowOutputBatching (bool)
  - .Count (int)
  - .Encoder (MessageEncoder)
  - .IsFixedSize (bool)
  - .IsReadOnly (bool)
  - .Keys (ICollection<string>)
  - .Security (SecurityMessageProperty)
  - .Values (ICollection<object>)
  - .Via (Uri)
- .IncomingMessageVersion (MessageVersion)
- .InstanceContext (*)
  - .DefaultCloseTimeout (TimeSpan)
  - .DefaultOpernTimeout (TimeSpan)
  - .Extensions (IExtensionCollection)
  - .Host (ServiceHostBase)
  - .IncomingChannels (ICollection<Ichannel>)
  - .ManualFlowControlLimit (int)
  - .OutgoingChannels (ICollection<Ichannel>)
  - .SynchronizationContext (*)
- .IsUseContext (bool)
- .OutgoingMessageHeaders (MessageHeaders)
- .OutgoingMessageProperties (MessageProperties)
- .RequestContext (*) – RequstMessage (Message)
- .ServiceSecurityContext (*)
- .SessionId (string)
- .SupportingTokens ((Collection<SupportingTokenSpecification>)

WCF Channels AutoOpen vs Open Semantics

Sample Apps to illustrate the differences between AutoOpen and Open---with respect to Async calls.

Here is a simple contract that has a Sync and Async version and the client will invoke a operation on the async contract 4 times. Lets see the difference in actual behavior when the channel is auto opened and explicitly opened.

```
[ServiceContract(Name="ISyncService")]
public interface IAsyncService
{
    [OperationContract(AsyncPattern=true, Action="SayHello")]
    IAsyncResult BeginSayHello(int id, AsyncCallback callback, Object obj);
    void EndSayHello(IAsyncResult ar);
}

[ServiceContract]
public interface ISyncService
{
    [OperationContract(Action="SayHello")] void SayHello(int id);
}
```

Here is the contract implementation. Notice the multiple concurrency that should allow concurrent processing of multiple messages.

```
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Multiple, InstanceContextMode=InstanceContextMode.Single)]
public class SyncService : ISyncService
{
    public void SayHello(int id)
    {
        Console.WriteLine("SayHello invoked with Id: {0}", id);
        Thread.Sleep(500); // Simulate doing some work.
    }
}
```

Finally the client code to invoke the service.

```
IAsyncService channel = ChannelFactory<IAsyncService>.CreateChannel(binding, new EndpointAddress(ServiceAddress));
for (int i = 0; i < 4; i++)
{
    channel.BeginSayHello(i, SayHelloCallback, channel);
}
```

This implies the "auto" open behavior. Lets see the output.

```
Issued all asynchronus requests. Waiting for them to complete.
SayHello invoked with Id: 0
SayHello invoked with Id: 1
SayHello invoked with Id: 2
SayHello invoked with Id: 3

All async requests completed
```

As you can see no matter how many times the client is run the server processes all the async invocations sequentially and in the order it was issued. This defeats the purpose of issuing async calls and having multiple concurrency mode on the server.

Lets see the output when the channel is explicitly opened.

```
((IChannel)channel).Open(); // added before the loop calling the Async method----BeginSayHello()
```

Output is:

```
Issued all asynchronus requests. Waiting for them to complete.
SayHello invoked with Id: 0
SayHello invoked with Id: 2
SayHello invoked with Id: 1
SayHello invoked with Id: 3

All async requests completed
```

As you can see the output is truly async and truly concurrent.

Credit: http://blogs.msdn.com/mahjayar/archive/2006/11/19/wcf-channels-autoopen-vs-open-semantics.aspx

IInstanceContextProvider interface:

InstanceContext **GetExistingInstanceContext**(Message msg, IContextChannel ch);
void **InitializeInstanceContext**(InstanceContext instCtx, Message msg, IContextChannel ch);

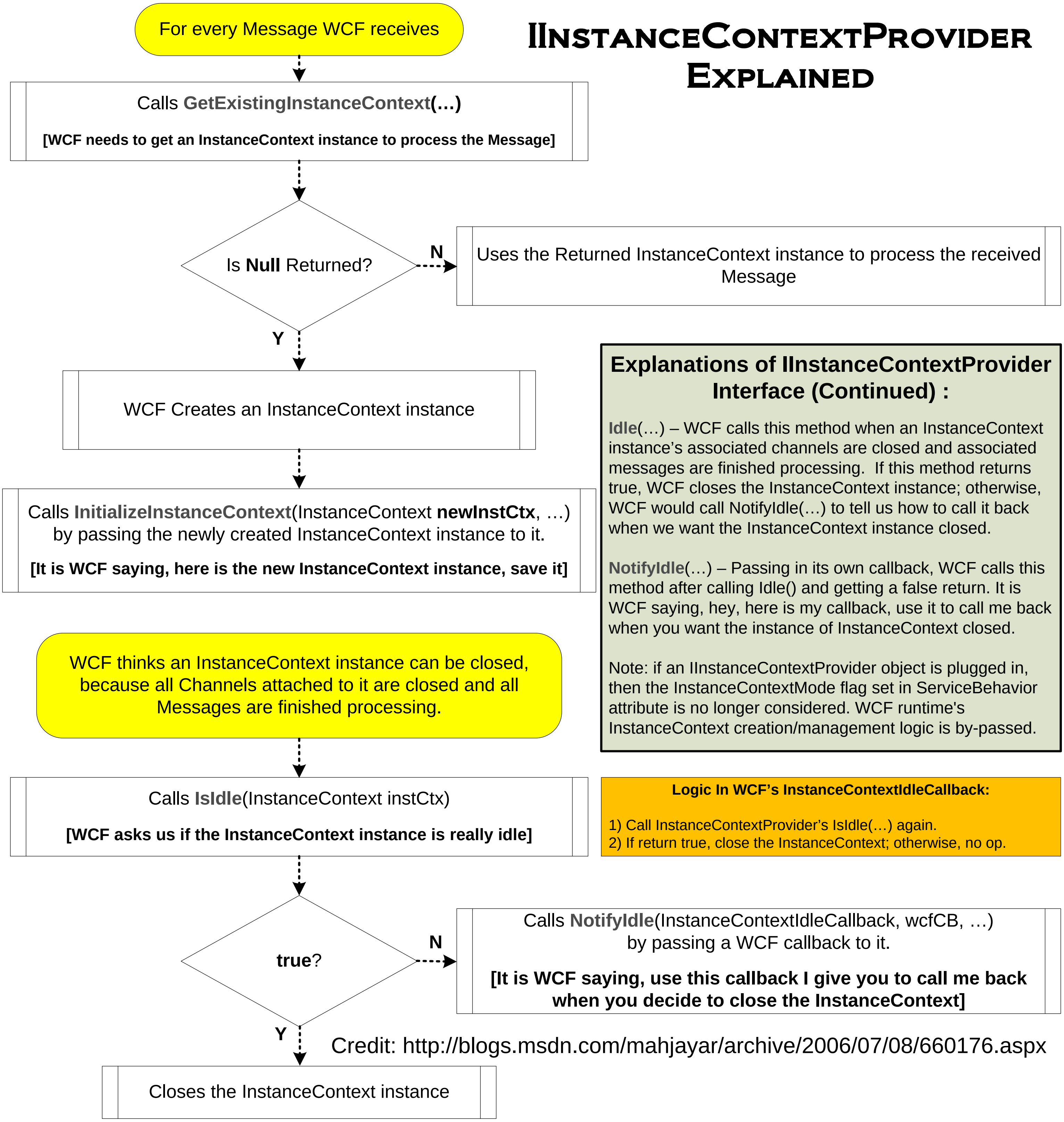
bool **IsIdle**(InstanceContext instCtx);
void **NotifyIdle**(InstanceContextIdleCallback callback, InstanceContext instCtx);

Explanations of IInstanceContextProvider interface:

GetExistingInstanceContext(…) – For every Message WCF receives, it calls this method, trying to get an InstanceContext instance. It uses the InstanceContext instance returned by this method to process the received message. However, if a null is returned by this method, then WCF would create a brand new InstanceContext instance to serve the received message; in addition, after creating an InstanceContext instance, WCF calls IntitializeInstanceContext(…), by passing the newly created InstanceContext instanceto it. [InstanceContext instancing here is subject to the InstanceContext throttling mechanism.]

InitializeInstanceContext(…) – WCF calls this method after creation of a brand new instance of InstanceContext, to notify us that a new one has been created, and here it is.

IInstanceContextProvider Explained



For every Message WCF receives

Calls **GetExistingInstanceContext(…)**

[WCF needs to get an InstanceContext instance to process the Message]

Is **Null** Returned?

N

Uses the Returned InstanceContext instance to process the received Message

Y

WCF Creates an InstanceContext instance

Calls **InitializeInstanceContext**(InstanceContext **newInstCtx**, …) by passing the newly created InstanceContext instance to it.

[It is WCF saying, here is the new InstanceContext instance, save it]

WCF thinks an InstanceContext instance can be closed, because all Channels attached to it are closed and all Messages are finished processing.

Calls **IsIdle**(InstanceContext instCtx)

[WCF asks us if the InstanceContext instance is really idle]

true?

N

Calls **NotifyIdle**(InstanceContextIdleCallback, wcfCB, …) by passing a WCF callback to it.

[It is WCF saying, use this callback I give you to call me back when you decide to close the InstanceContext]

Y

Closes the InstanceContext instance

Explanations of IInstanceContextProvider Interface (Continued) :

Idle(…) – WCF calls this method when an InstanceContext instance's associated channels are closed and associated messages are finished processing. If this method returns true, WCF closes the InstanceContext instance; otherwise, WCF would call NotifyIdle(…) to tell us how to call it back when we want the InstanceContext instance closed.

NotifyIdle(…) – Passing in its own callback, WCF calls this method after calling Idle() and getting a false return. It is WCF saying, hey, here is my callback, use it to call me back when you want the instance of InstanceContext closed.

Note: if an IInstanceContextProvider object is plugged in, then the InstanceContextMode flag set in ServiceBehavior attribute is no longer considered. WCF runtime's InstanceContext creation/management logic is by-passed.

Logic In WCF's InstanceContextIdleCallback:

1) Call InstanceContextProvider's IsIdle(…) again.
2) If return true, close the InstanceContext; otherwise, no op.

Credit: http://blogs.msdn.com/mahjayar/archive/2006/07/08/660176.aspx

IInstanceContextProvider vs IInstanceContextInitializer

June CTP introduces a new extension (IInstanceContextProvider) to enable InstanceContext sharing. One common question being raised internally iswhat are the difference between

IInstanceContextProvider.**InitializeInstanceContext**(**InstanceContext** newInstCtx**, Message msg, IContextChannel ch)**

and

IInstanceContextInitializer.**Initialize**(**InstnaceContext** newInstCtx**, Message** msg**)**.

Both these methods are called whenever a new InstanceContext is created and so the question is why did IInstanceContextProvider duplicate this method?

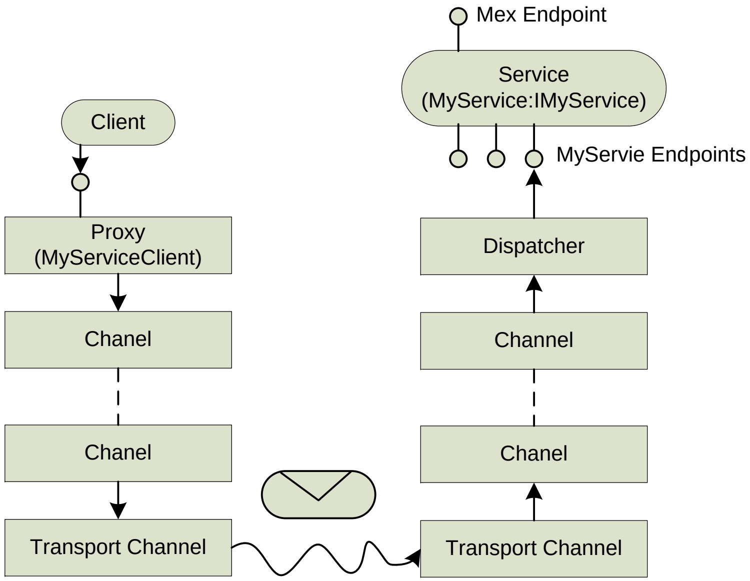
Let me talk about why this method was introduced in IInstanceContextProvider. To implement sharing users need to know when a new InstanceContext is created so they can wire up the caching logic. So if we did not provide this API, users would need to implement IInstanceContextInitializer just to get this notification. Also, for sharing, one needs to associate the Channel on which the message came from with the InstanceContext created, so that the InstanceContext lives while that Channel is functional. One look at the IInstanceContextInitializer.Initialize parameters tells us that it currently doesn’t pass the Channel. So one possible solution is to change Initialize signature. But we felt that the sharing extension should be all that the users need to worry about when designing sharing and the API's in that extension should be designed such that all necessary information is available to the implementer. So we decided to go with one interface for the extension.

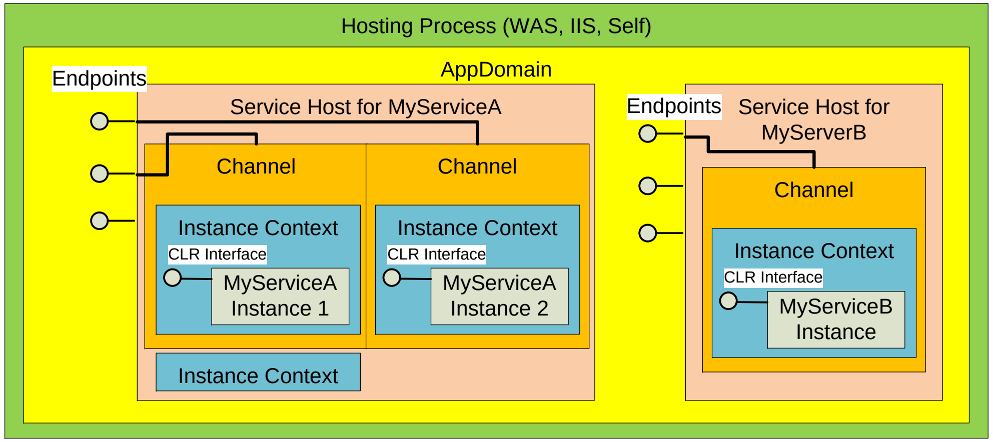
The one may wonder that IInstanceContextInitializer is redundant and that it can be removed. The reason its still there is because both these interfaces are there for two very different scenarios. Think of them like two kind of saws, one a chain saw and the other a low power precision saw. IInstanceContextProvider is the heavy duty one which lets you bypass InstanceContext lifetime logic, enable sharing and InstanceContext leasing, while IInstanceContextInitializer is for more finer things like adding extensions to newly created InstanceContext.

Also, for a given Endpoint, you can have exactly one IInstanceContextProvider while it can have any number of IInstanceContextInitializers, because EndpointDipatcher->DispatchRuntime.InstanceContextInitializers returns a Collection object---a Collection of IInstanceContextInitializer objects, while .InstanceContextProvider returns a single IInstanceContextProvider object.

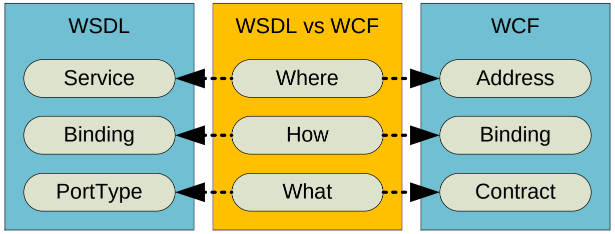
http://blogs.msdn.com/mahjayar/archive/2006/07/17/668859.aspx

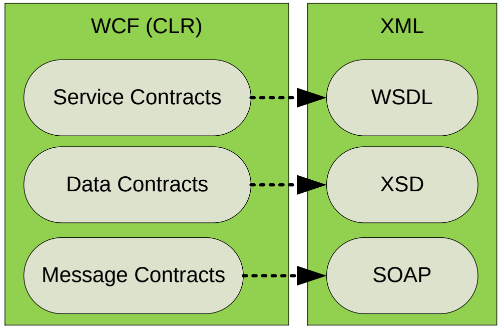
WCF Architecture and Main Concepts - 1





Each .Net host process can have many AppDomains; each AppDomain can have zero or more Service Host instances; each Service Host instance is dedicated to a particular Service Type (the Type implement the Service), and has zero or more Instance Contexts, the innermost execution scope of the service instance; each Instance Context is associated with zero or one Service instance, meaning it could also be empty.



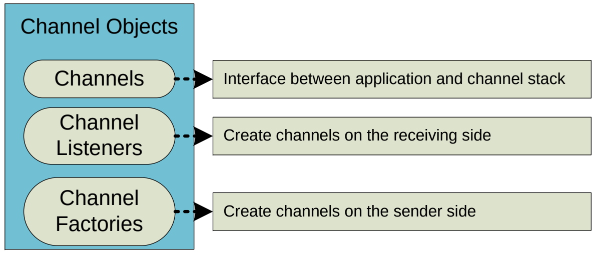


Type in Service

- Typed
- Typed Message
- Message

Message Exchange Pattern

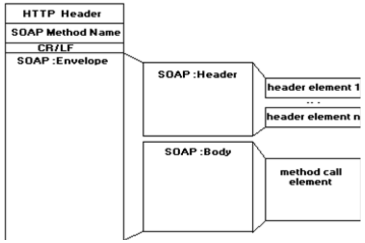
- Req/Reply
- One-Way
- Duplex

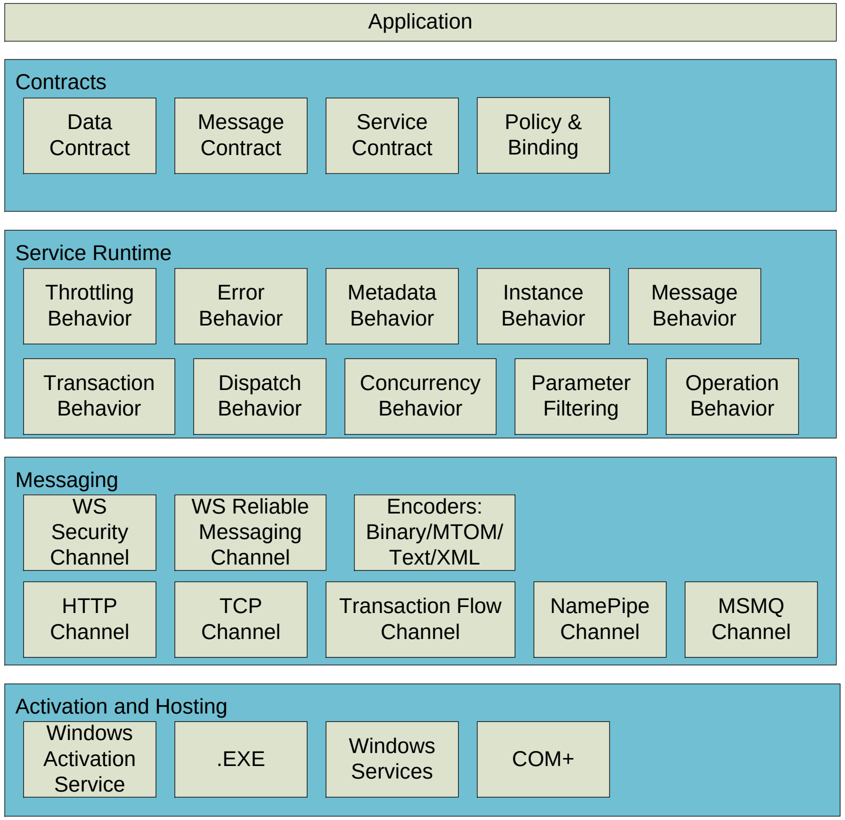


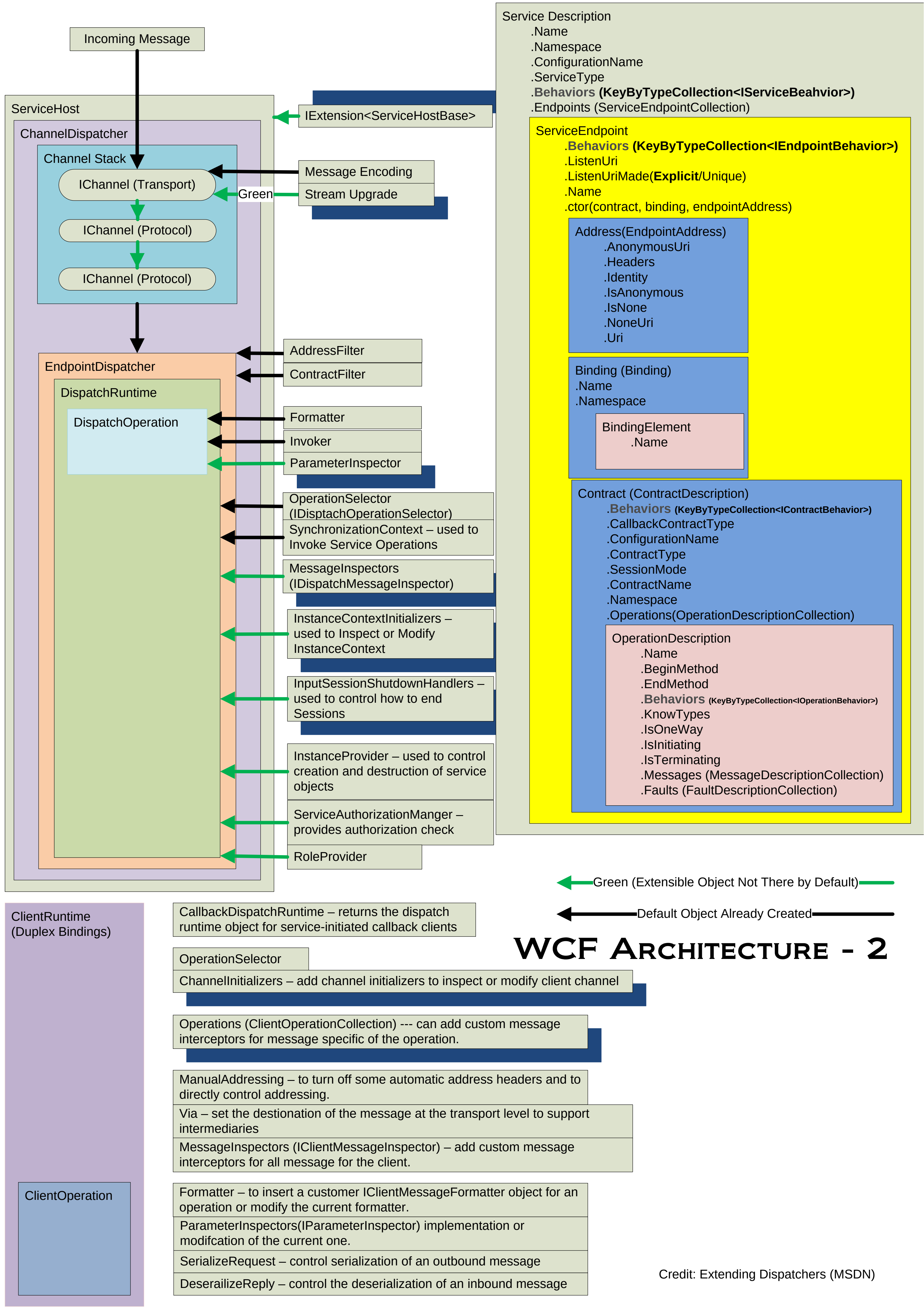
Channels: Interface between application and channel stack

Channel Listeners: Create channels on the receiving side

Channel Factories: Create channels on the sender side







Incoming Message

ServiceHost

ChannelDispatcher

Channel Stack

IChannel (Transport)

IChannel (Protocol)

IChannel (Protocol)

IExtension<ServiceHostBase>

Message Encoding

Stream Upgrade

Green

EndpointDispatcher

DispatchRuntime

DispatchOperation

AddressFilter

ContractFilter

Formatter

Invoker

ParameterInspector

OperationSelector (IDisptachOperationSelector)

SynchronizationContext – used to Invoke Service Operations

MessageInspectors (IDispatchMessageInspector)

InstanceContextInitializers – used to Inspect or Modify InstanceContext

InputSessionShutdownHandlers – used to control how to end Sessions

InstanceProvider – used to control creation and destruction of service objects

ServiceAuthorizationManger – provides authorization check

RoleProvider

Service Description
.Name
.Namespace
.ConfigurationName
.ServiceType
.**Behaviors (KeyByTypeCollection<IServiceBeahvior>)**
.Endpoints (ServiceEndpointCollection)

ServiceEndpoint
.**Behaviors (KeyByTypeCollection<IEndpointBehavior>)**
.ListenUri
.ListenUriMade(**Explicit**/Unique)
.Name
.ctor(contract, binding, endpointAddress)

Address(EndpointAddress)
.AnonymousUri
.Headers
.Identity
.IsAnonymous
.IsNone
.NoneUri
.Uri

Binding (Binding)
.Name
.Namespace

BindingElement
.Name

Contract (ContractDescription)
.**Behaviors (KeyByTypeCollection<IContractBehavior>)**
.CallbackContractType
.ConfigurationName
.ContractType
.SessionMode
.ContractName
.Namespace
.Operations(OperationDescriptionCollection)

OperationDescription
.Name
.BeginMethod
.EndMethod
.**Behaviors (KeyByTypeCollection<IOperationBehavior>)**
.KnowTypes
.IsOneWay
.IsInitiating
.IsTerminating
.Messages (MessageDescriptionCollection)
.Faults (FaultDescriptionCollection)

Green (Extensible Object Not There by Default)

Default Object Already Created

ClientRuntime (Duplex Bindings)

CallbackDispatchRuntime – returns the dispatch runtime object for service-initiated callback clients

OperationSelector

ChannelInitializers – add channel initializers to inspect or modify client channel

Operations (ClientOperationCollection) --- can add custom message interceptors for message specific of the operation.

ManualAddressing – to turn off some automatic address headers and to directly control addressing.

Via – set the destionation of the message at the transport level to support intermediaries

MessageInspectors (IClientMessageInspector) – add custom message interceptors for all message for the client.

ClientOperation

Formatter – to insert a customer IClientMessageFormatter object for an operation or modify the current formatter.

ParameterInspectors(IParameterInspector) implementation or modifcation of the current one.

SerializeRequest – control serialization of an outbound message

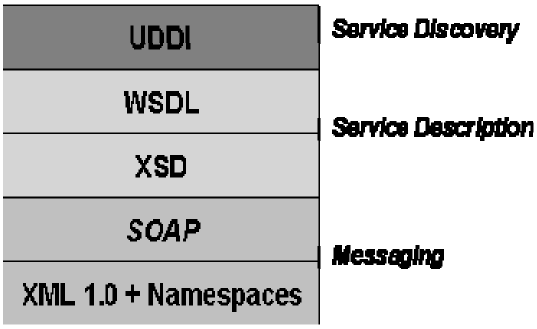
DeserailizeReply – control the deserialization of an inbound message

WCF Architecture - 2

Credit: Extending Dispatchers (MSDN)

WSDL

```
<!-- WSDL definition structure -->
<definitions name="MathService"
   targetNamespace="http://example.org/math/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- abstract definitions -->
   <types> …  <!-- a container for abstract types defined using XML Schema -->
   <message> … <!-- an abstract message may consist of multiple <part>'s, each may be of a different type  -->
  <portType> … <!-- an abstract set of operations supported by one or more endpoints (aka interface) -->
                    <!-- Operations are defined by an exchange of messages of <input> or <output> -->
  <!-- concrete definitions -->
  <binding> … <!-- a concrete protocol and data format specification for a particular portType (aka interface) -->
  <service> … <!-- a collection of related endpoints, where an endpoint is a combination of a binding and an address (URI) -->
</definition>
```



```
<definitions .... >
   <types>
      <xsd:schema .... />*
   </types>
</definitions>
```

```
<definitions .... >
   <message name="nmtoken"> *
      <part name="nmtoken" element="qname"? type="qname"?/> *
   </message>
</definitions>
```

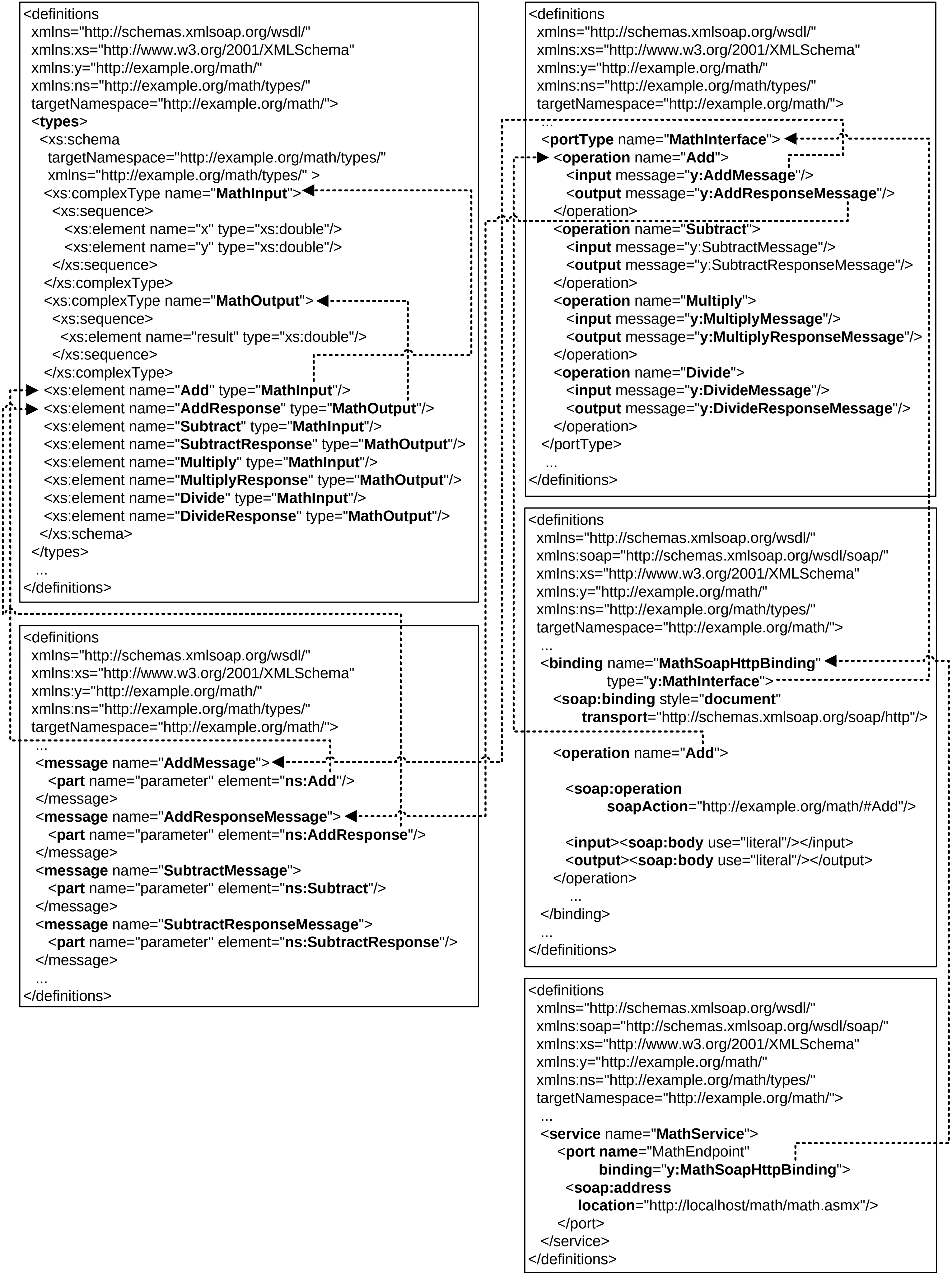
```
<definitions .... >
   <portType name="nmtoken">
      <operation name="nmtoken" .... /> *
   </portType>
</definitions>
```

```
<wsdl:definitions .... >
   <wsdl:binding name="nmtoken" type="qname"> *
      <-- extensibility element providing binding details --> *
      <wsdl:operation name="nmtoken"> *
        <-- extensibility element for operation details --> *
        <wsdl:input name="nmtoken"? > ?
           <-- extensibility element for body details -->
        </wsdl:input>
        <wsdl:output name="nmtoken"? > ?
           <-- extensibility element for body details -->
        </wsdl:output>
        <wsdl:fault name="nmtoken"> *
           <-- extensibility element for body details -->
        </wsdl:fault>
      </wsdl:operation>
   </wsdl:binding>
</wsdl:definitions>
```

```
<definitions .... >
   <service .... > *
      <port name="nmtoken" binding="qname"> *
        <-- extensibility element defines address details -->
      </port>
   </service>
</definitions>
```

Credit: http://msdn.microsoft.com/en-us/library/ms996486.aspx (Understanding WSDL)

WSDL – 2 (How Elements are Connected)



```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">
  <types>
   <xs:schema
    targetNamespace="http://example.org/math/types/"
    xmlns="http://example.org/math/types/" >
    <xs:complexType name="MathInput">
     <xs:sequence>
       <xs:element name="x" type="xs:double"/>
       <xs:element name="y" type="xs:double"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="MathOutput">
     <xs:sequence>
      <xs:element name="result" type="xs:double"/>
     </xs:sequence>
    </xs:complexType>
    <xs:element name="Add" type="MathInput"/>
    <xs:element name="AddResponse" type="MathOutput"/>
    <xs:element name="Subtract" type="MathInput"/>
    <xs:element name="SubtractResponse" type="MathOutput"/>
    <xs:element name="Multiply" type="MathInput"/>
    <xs:element name="MultiplyResponse" type="MathOutput"/>
    <xs:element name="Divide" type="MathInput"/>
    <xs:element name="DivideResponse" type="MathOutput"/>
   </xs:schema>
  </types>
  ...
</definitions>
```

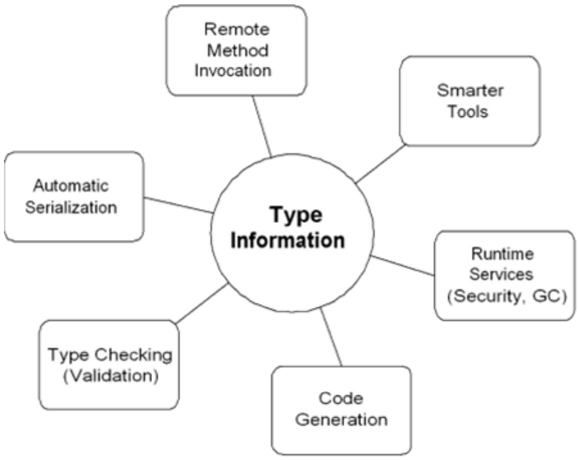
```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">
   ...
   <message name="AddMessage">
     <part name="parameter" element="ns:Add"/>
   </message>
   <message name="AddResponseMessage">
     <part name="parameter" element="ns:AddResponse"/>
   </message>
   <message name="SubtractMessage">
     <part name="parameter" element="ns:Subtract"/>
   </message>
   <message name="SubtractResponseMessage">
     <part name="parameter" element="ns:SubtractResponse"/>
   </message>
   ...
</definitions>
```

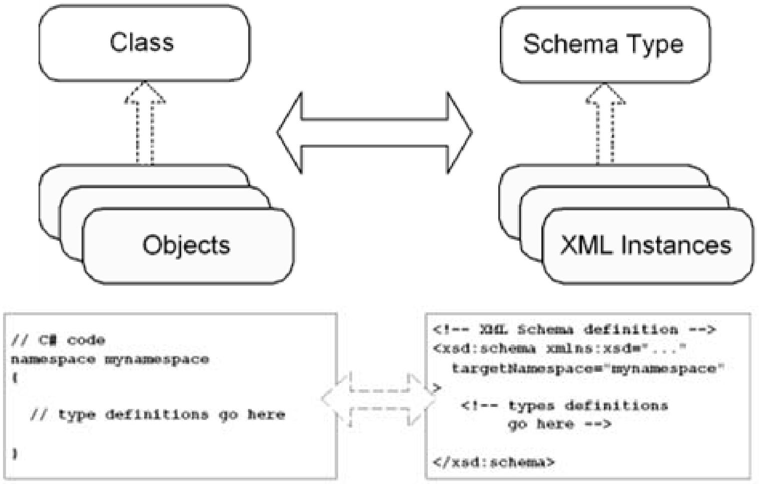
```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">
   ...
  <portType name="MathInterface">
    <operation name="Add">
      <input message="y:AddMessage"/>
      <output message="y:AddResponseMessage"/>
    </operation>
    <operation name="Subtract">
      <input message="y:SubtractMessage"/>
      <output message="y:SubtractResponseMessage"/>
    </operation>
    <operation name="Multiply">
      <input message="y:MultiplyMessage"/>
      <output message="y:MultiplyResponseMessage"/>
    </operation>
    <operation name="Divide">
      <input message="y:DivideMessage"/>
      <output message="y:DivideResponseMessage"/>
    </operation>
  </portType>
   ...
</definitions>
```

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">
   ...
  <binding name="MathSoapHttpBinding"
              type="y:MathInterface">
    <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="Add">

      <soap:operation
            soapAction="http://example.org/math/#Add"/>

      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
      ...
  </binding>
   ...
</definitions>
```

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">
   ...
  <service name="MathService">
     <port name="MathEndpoint"
           binding="y:MathSoapHttpBinding">
      <soap:address
        location="http://localhost/math/math.asmx"/>
     </port>
  </service>
</definitions>
```

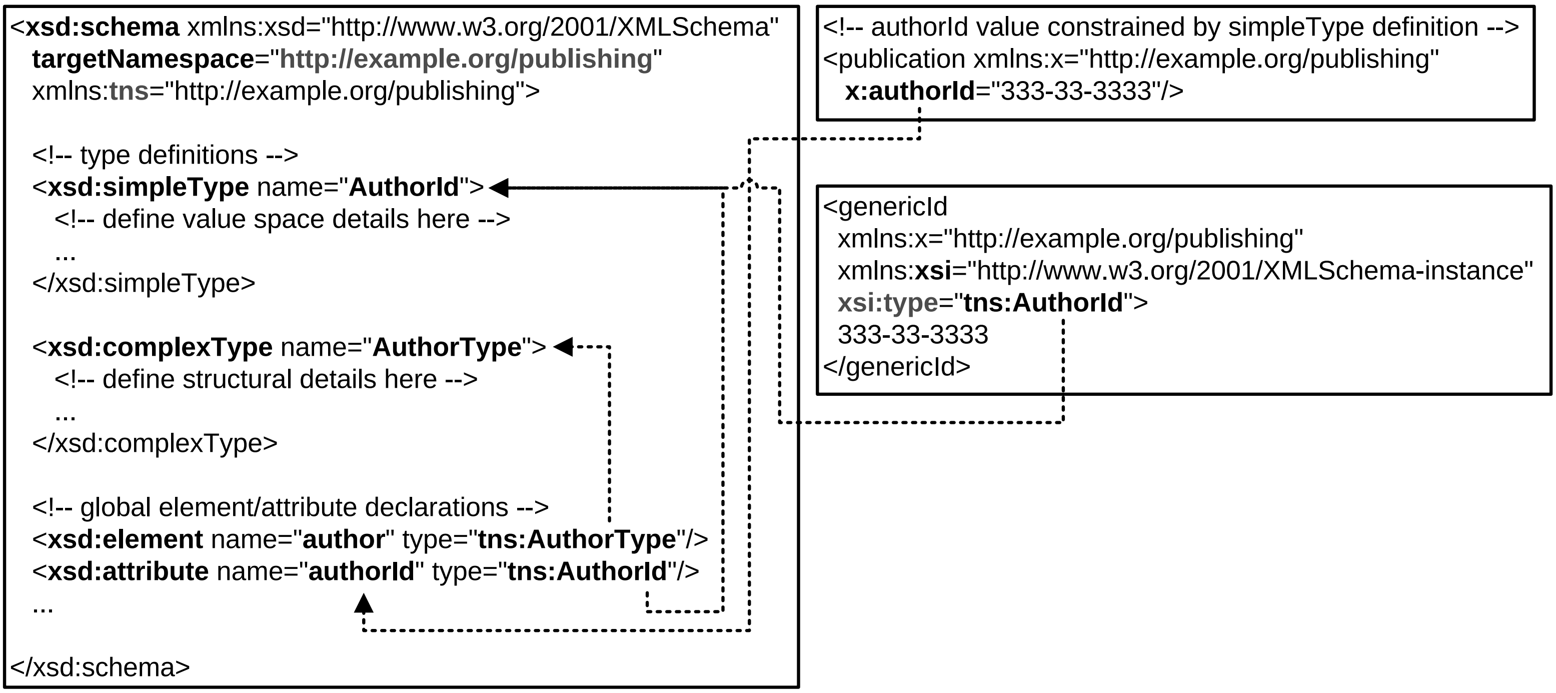
XML Schema (Defining a Namespace---Valid Elements and Types in it)





```
// C# code
namespace mynamespace
{

  // type definitions go here

}
```

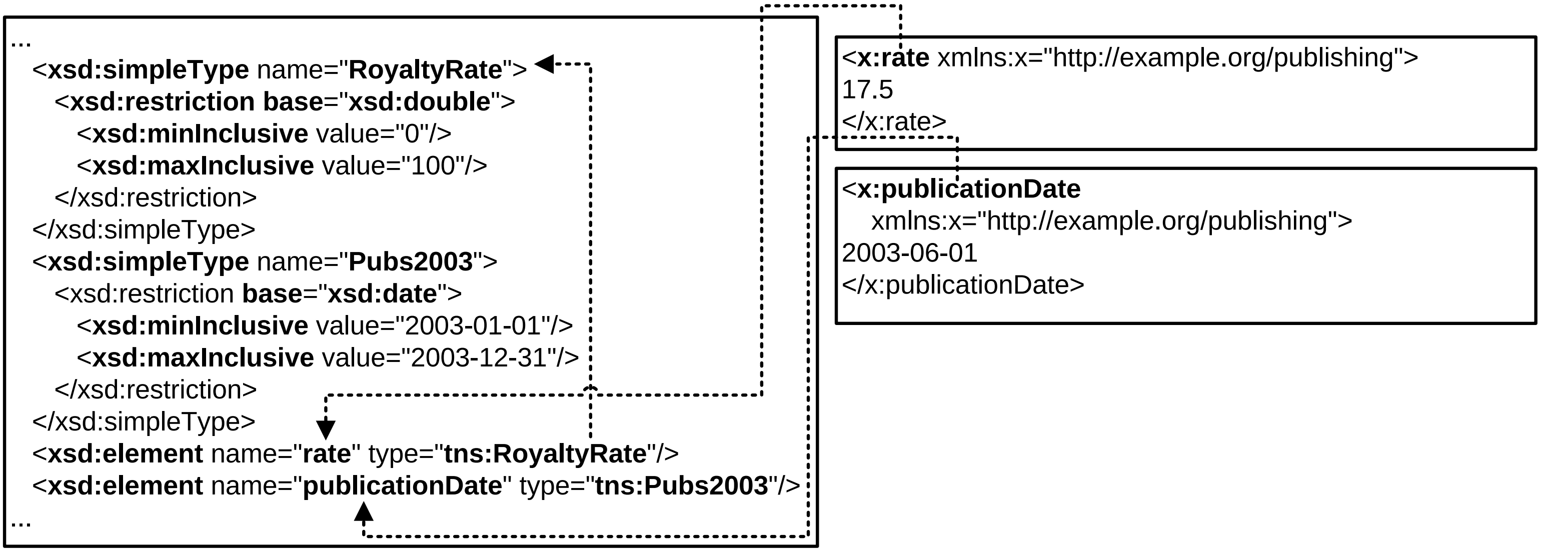
```
<!-- XML Schema definition -->
<xsd:schema xmlns:xsd="..."
  targetNamespace="mynamespace"
>
   <!-- types definitions
        go here -->

</xsd:schema>
```



```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.org/publishing"
  xmlns:tns="http://example.org/publishing">

  <!-- type definitions -->
  <xsd:simpleType name="AuthorId">
    <!-- define value space details here -->
    ...
  </xsd:simpleType>

  <xsd:complexType name="AuthorType">
    <!-- define structural details here -->
    ...
  </xsd:complexType>

  <!-- global element/attribute declarations -->
  <xsd:element name="author" type="tns:AuthorType"/>
  <xsd:attribute name="authorId" type="tns:AuthorId"/>
  ...

</xsd:schema>
```

```
<!-- authorId value constrained by simpleType definition -->
<publication xmlns:x="http://example.org/publishing"
  x:authorId="333-33-3333"/>
```

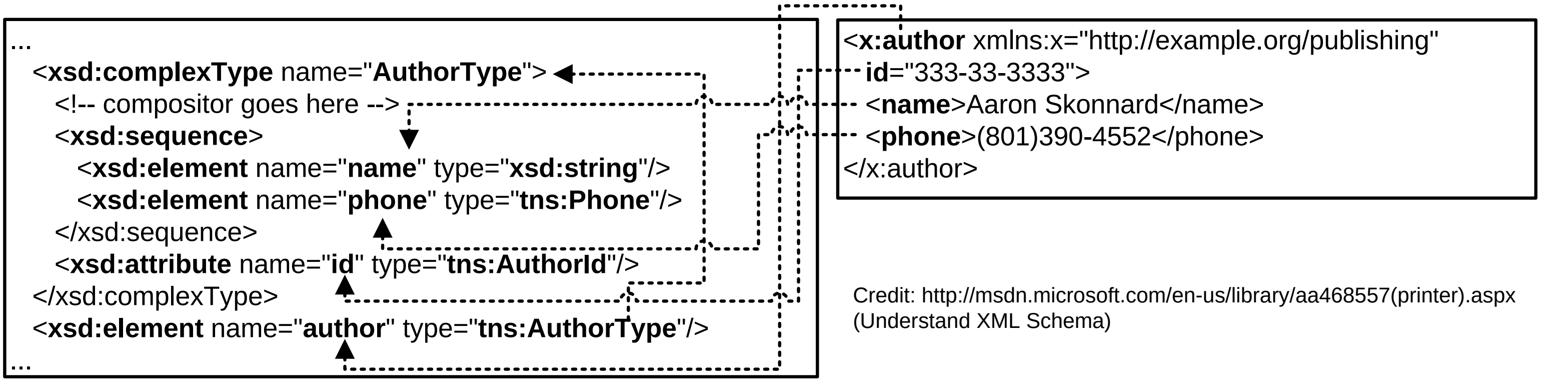
```
<genericId
  xmlns:x="http://example.org/publishing"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="tns:AuthorId">
  333-33-3333
</genericId>
```



```
...
  <xsd:simpleType name="RoyaltyRate">
    <xsd:restriction base="xsd:double">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="Pubs2003">
    <xsd:restriction base="xsd:date">
      <xsd:minInclusive value="2003-01-01"/>
      <xsd:maxInclusive value="2003-12-31"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="rate" type="tns:RoyaltyRate"/>
  <xsd:element name="publicationDate" type="tns:Pubs2003"/>
...
```

```
<x:rate xmlns:x="http://example.org/publishing">
17.5
</x:rate>
```

```
<x:publicationDate
   xmlns:x="http://example.org/publishing">
2003-06-01
</x:publicationDate>
```



```
...
  <xsd:complexType name="AuthorType">
    <!-- compositor goes here -->
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="phone" type="tns:Phone"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="tns:AuthorId"/>
  </xsd:complexType>
  <xsd:element name="author" type="tns:AuthorType"/>
...
```

```
<x:author xmlns:x="http://example.org/publishing"
  id="333-33-3333">
  <name>Aaron Skonnard</name>
  <phone>(801)390-4552</phone>
</x:author>
```

Credit: http://msdn.microsoft.com/en-us/library/aa468557(printer).aspx
(Understand XML Schema)

SOAP

```
<soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header> <!-- optional -->
    <!-- header blocks go here... -->
  </soap:Header>
  <soap:Body>
    <!-- payload or Fault element goes here... -->
  </soap:Body>
</soap:Envelope>
```

General Format

```
<soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <x:TransferFunds xmlns:x="urn:examples-org:banking">
   <from>22-342439</from>
   <to>98-283843</to>
   <amount>100.00</amount>
  </x:TransferFunds>
 </soap:Body>
</soap:Envelope>
```

Example

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <soap:Fault>
   <faultcode>soap:Server</faultcode>
   <faultstring>Insufficient funds</faultstring>
   <detail>
    <x:TransferError xmlns:x="urn:examples-org:banking">
     <sourceAccount>22-342439</sourceAccount>
     <transferAmount>100.00</transferAmount>
     <currentBalance>89.23</currentBalance>
    </x:TransferError>
   </detail>
  </x:TransferFunds>
 </soap:Body>
</soap:Envelope>
```

Example

Credit: http://msdn.microsoft.com/en-us/library/ms995800(printer).aspx (Undstanding SOAP)

SOAP 1.1 XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/" >

  <!-- Envelope, header and body -->
  <xs:element name="Envelope" type="tns:Envelope" />
  <xs:complexType name="Envelope" >
    <xs:sequence>
      <xs:element ref="tns:Header" minOccurs="0" />
      <xs:element ref="tns:Body" minOccurs="1" />
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:complexType>

  <xs:element name="Header" type="tns:Header" />
  <xs:complexType name="Header" >
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##other"
     processContents="lax" />
  </xs:complexType>

  <xs:element name="Body" type="tns:Body" />
  <xs:complexType name="Body" >
    <xs:sequence>
      <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##any"
     processContents="lax" />
  </xs:complexType>

  <!-- Global Attributes -->
  <xs:attribute name="mustUnderstand" default="0" >
     <xs:simpleType>
     <xs:restriction base='xs:boolean'>
      <xs:pattern value='0|1' />
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="actor" type="xs:anyURI" />

  <xs:simpleType name="encodingStyle" >
    <xs:list itemType="xs:anyURI" />
  </xs:simpleType>

  <xs:attribute name="encodingStyle" type="tns:encodingStyle" />
  <xs:attributeGroup name="encodingStyle" >
    <xs:attribute ref="tns:encodingStyle" />
  </xs:attributeGroup>

  <xs:element name="Fault" type="tns:Fault" />
  <xs:complexType name="Fault" final="extension" >
    <xs:sequence>
      <xs:element name="faultcode" type="xs:QName" />
      <xs:element name="faultstring" type="xs:string" />
      <xs:element name="faultactor" type="xs:anyURI" minOccurs="0" />
      <xs:element name="detail" type="tns:detail" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="detail">
    <xs:sequence>
      <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##any" processContents="lax" />
  </xs:complexType>
</xs:schema>
```

[ServiceContract]

Attribute on a Type or Interface

.CallbackContract
.ConfigurationName
.HasProtectionLevel
.Name
.Namespace
.ProtectionLevel
.SessionMode

[DataContract]

(data passed through the service ops)

.IsReference
.Name
.Namespace

MessageContract **(msg's serialized format)**

.HasProtectionLevel
.IsWrapped
.ProtectionLevel
.WrapperName
.WrapperNamespace

FaultContract **(err msg format)**

.Action
.DetailType
.HasProtectionLevel
.Name
.Namespace
.ProtectionLevel

System.Runtime.Serialization
System.Xml.Serialization

[OperationContract]

Attribte on Interface or Type Methods.

.Action
.AsyncPattern
.HasProtectionLevel
.IsInitiating
.IsOneWay
.IsTerminating
.Name
.ProtectionLevel
.ReplyAction

[DataMember]

.IsRequired
.Name
.Order

OperationContext

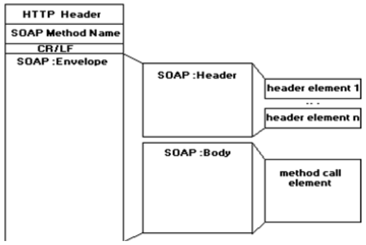
.Channel
.Current (OperationContext)
.EndpointDispatcher
.Extensions
.HasSupportingTokens
.Host
.IncomingMessageHeaders
.IncomingMessageProperties
.IncomingMessageVersion
.InstanceContext
.IsUseContext
.OutgoingMessageHeaders
.OutgoingMessageProperties
.RequestContext
.ServiceSecurityContext
.SessionId
.SupportingTokens

[ServiceBehavior] Attribute on Type

.AddressFilterMode
.AutomaticSessionShutdown
.ConcurrencyMode
.ConfigurationName
.IgnoreExtensionDataObject
.IncludeExceptionDetailInFaults
.InstanceContextMode
.MaxItemsInObjectGraph
.Name
.Namespace
.ReleaseServiceInstanceOnTransactionComplete
.TransactionAutoCompleteOnSessionClose
.TransactionIsolationLevel
.TransactionTimeout
.UseSynchronizationContext
.ValidateMustUnderstand

CalbackBehavior

.AutomaticSessionShutdown
.ConcurrencyMode
.IgnoreExtensionDataObject
.IncludeExceptionDetailInFaults
.MaxItemsInObjectGraph
.TransactionIsolationLevel
.TransactionTimeout
.UseSynchronizationContext
.ValidateMustUnderstand



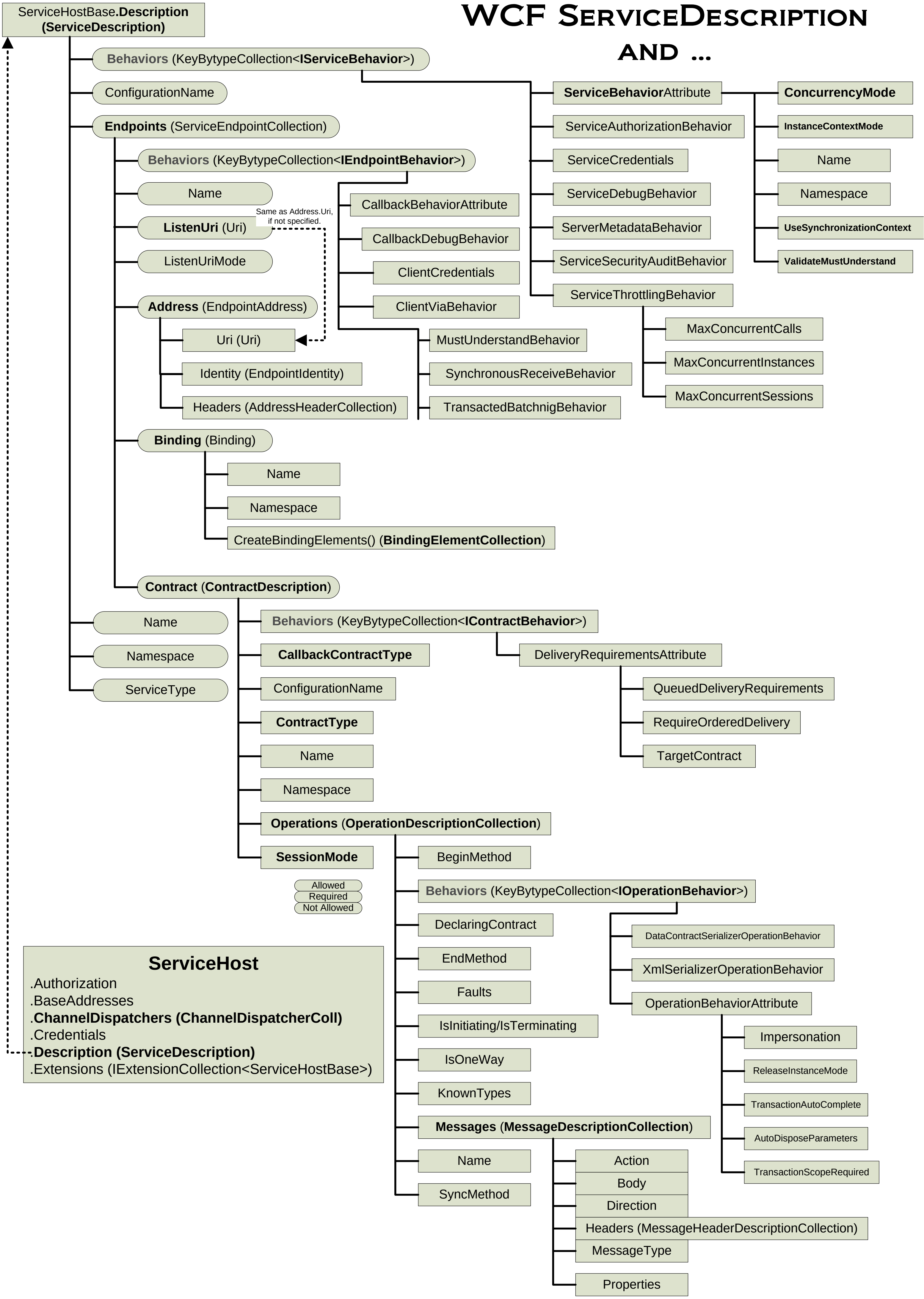
CommunicationObject

.DefaultCloseTimeout
.DefaultOpenTimeout
.IsDisposed
.State
.ThisLock
.Closed (event)
.Closing (event)
.Faulted (event)
.Opened (event)
.Openning (event)

ServiceDescription

.Behaviors
.ConfigurationName
.Endpoints
.Name
.Namespace
.ServiceType

WCF ServiceDescription and ...



ServiceHostBase.**Description** **(ServiceDescription)**

Behaviors (KeyBytypeCollection<**IServiceBehavior**>)

ServiceBehaviorAttribute

ConcurrencyMode

InstanceContextMode

Name

Namespace

UseSynchronizationContext

ValidateMustUnderstand

ServiceAuthorizationBehavior

ServiceCredentials

ServiceDebugBehavior

ServerMetadataBehavior

ServiceSecurityAuditBehavior

ServiceThrottlingBehavior

MaxConcurrentCalls

MaxConcurrentInstances

MaxConcurrentSessions

ConfigurationName

Endpoints (ServiceEndpointCollection)

Behaviors (KeyBytypeCollection<**IEndpointBehavior**>)

CallbackBehaviorAttribute

CallbackDebugBehavior

ClientCredentials

ClientViaBehavior

MustUnderstandBehavior

SynchronousReceiveBehavior

TransactedBatchnigBehavior

Name

ListenUri (Uri)

Same as Address.Uri, if not specified.

ListenUriMode

Address (EndpointAddress)

Uri (Uri)

Identity (EndpointIdentity)

Headers (AddressHeaderCollection)

Binding (Binding)

Name

Namespace

CreateBindingElements() (**BindingElementCollection**)

Contract (**ContractDescription**)

Behaviors (KeyBytypeCollection<**IContractBehavior**>)

DeliveryRequirementsAttribute

QueuedDeliveryRequirements

RequireOrderedDelivery

TargetContract

CallbackContractType

ConfigurationName

ContractType

Name

Namespace

Operations (**OperationDescriptionCollection**)

BeginMethod

Behaviors (KeyBytypeCollection<**IOperationBehavior**>)

DataContractSerializerOperationBehavior

XmlSerializerOperationBehavior

OperationBehaviorAttribute

Impersonation

ReleaseInstanceMode

TransactionAutoComplete

AutoDisposeParameters

TransactionScopeRequired

DeclaringContract

EndMethod

Faults

IsInitiating/IsTerminating

IsOneWay

KnownTypes

Messages (**MessageDescriptionCollection**)

Action

Body

Direction

Headers (MessageHeaderDescriptionCollection)

MessageType

Properties

Name

SyncMethod

SessionMode

Allowed

Required

Not Allowed

Name

Namespace

ServiceType

ServiceHost

.Authorization

.BaseAddresses

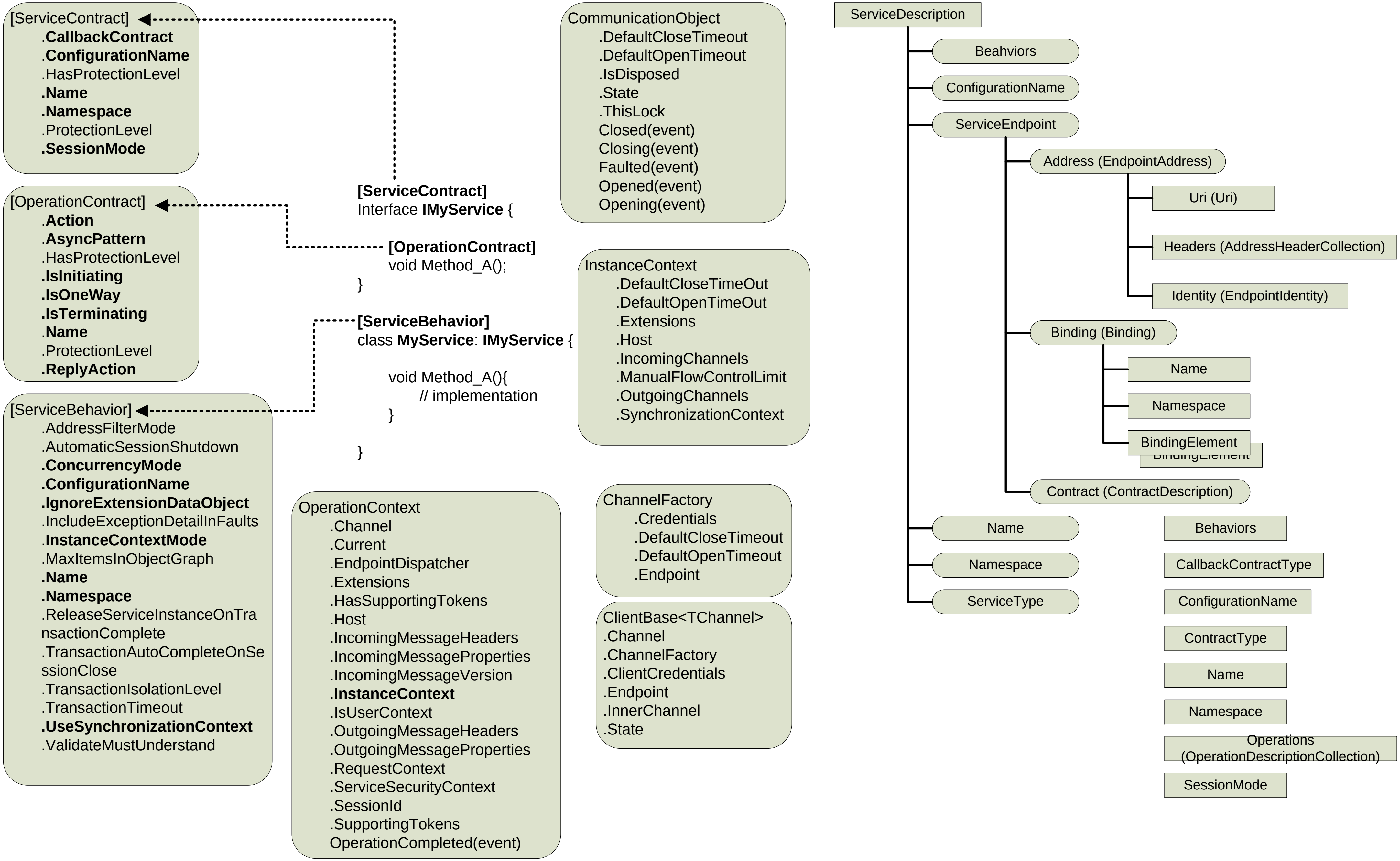
.ChannelDispatchers (ChannelDispatcherColl)

.Credentials

.Description (ServiceDescription)

.Extensions (IExtensionCollection<ServiceHostBase>)

WCF Basics



[ServiceContract]
- .**CallbackContract**
- .**ConfigurationName**
- .HasProtectionLevel
- .**Name**
- .**Namespace**
- .ProtectionLevel
- .**SessionMode**

[OperationContract]
- .**Action**
- .**AsyncPattern**
- .HasProtectionLevel
- .**IsInitiating**
- .**IsOneWay**
- .**IsTerminating**
- .**Name**
- .ProtectionLevel
- .**ReplyAction**

[ServiceBehavior]
- .AddressFilterMode
- .AutomaticSessionShutdown
- .**ConcurrencyMode**
- .**ConfigurationName**
- .**IgnoreExtensionDataObject**
- .IncludeExceptionDetailInFaults
- .**InstanceContextMode**
- .MaxItemsInObjectGraph
- .**Name**
- .**Namespace**
- .ReleaseServiceInstanceOnTransactionComplete
- .TransactionAutoCompleteOnSessionClose
- .TransactionIsolationLevel
- .TransactionTimeout
- .**UseSynchronizationContext**
- .ValidateMustUnderstand

```
[ServiceContract]
Interface IMyService {

    [OperationContract]
    void Method_A();
}

[ServiceBehavior]
class MyService: IMyService {

    void Method_A(){
        // implementation
    }

}
```

OperationContext
- .Channel
- .Current
- .EndpointDispatcher
- .Extensions
- .HasSupportingTokens
- .Host
- .IncomingMessageHeaders
- .IncomingMessageProperties
- .IncomingMessageVersion
- .**InstanceContext**
- .IsUserContext
- .OutgoingMessageHeaders
- .OutgoingMessageProperties
- .RequestContext
- .ServiceSecurityContext
- .SessionId
- .SupportingTokens
- OperationCompleted(event)

CommunicationObject
- .DefaultCloseTimeout
- .DefaultOpenTimeout
- .IsDisposed
- .State
- .ThisLock
- Closed(event)
- Closing(event)
- Faulted(event)
- Opened(event)
- Opening(event)

InstanceContext
- .DefaultCloseTimeOut
- .DefaultOpenTimeOut
- .Extensions
- .Host
- .IncomingChannels
- .ManualFlowControlLimit
- .OutgoingChannels
- .SynchronizationContext

ChannelFactory
- .Credentials
- .DefaultCloseTimeout
- .DefaultOpenTimeout
- .Endpoint

ClientBase<TChannel>
- .Channel
- .ChannelFactory
- .ClientCredentials
- .Endpoint
- .InnerChannel
- .State

ServiceDescription
- Beahviors
- ConfigurationName
- ServiceEndpoint
  - Address (EndpointAddress)
    - Uri (Uri)
    - Headers (AddressHeaderCollection)
    - Identity (EndpointIdentity)
  - Binding (Binding)
    - Name
    - Namespace
    - BindingElement
  - Contract (ContractDescription)
- Name
- Namespace
- ServiceType

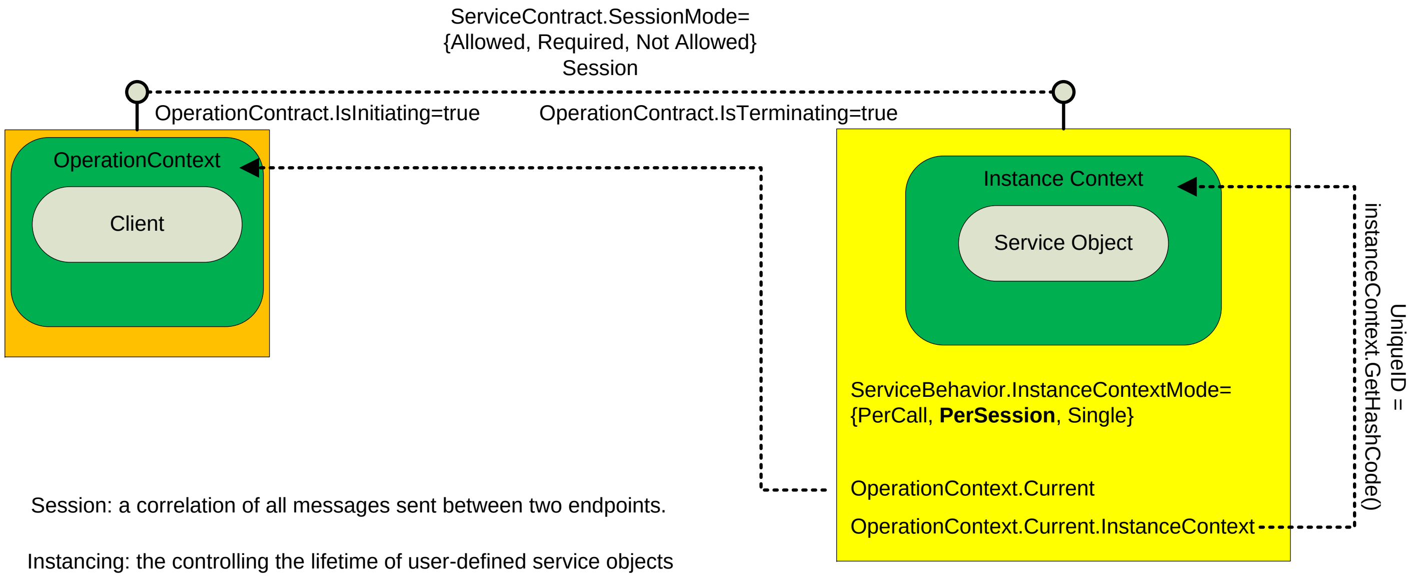
BindingElement

- Behaviors
- CallbackContractType
- ConfigurationName
- ContractType
- Name
- Namespace
- Operations (OperationDescriptionCollection)
- SessionMode

// "this" represents an object that implements the callback contract IMyDuplexServiceCallback
InstanceContext ic = new InstanceContext(this); var m_ChannelFactory = new
DuplexChannelFactory<IMyDuplexService>(ic);

IMyServiceCallback callback =
OperationContext.Current.GetCallbackChannel<IMyServiceCallback>()

WCF Session, Instancing, Concurrency



ServiceContract.SessionMode=
{Allowed, Required, Not Allowed}
Session

OperationContract.IsInitiating=true

OperationContract.IsTerminating=true

ServiceBehavior.InstanceContextMode=
{PerCall, **PerSession**, Single}

OperationContext.Current

OperationContext.Current.InstanceContext

UniqueID =
instanceContext.GetHashCode()

Session: a correlation of all messages sent between two endpoints.

Instancing: the controlling the lifetime of user-defined service objects and their related InstanceContext objects.

To do this I need an **IExtension** that defines the values I want to store, an **IInstanceContextProvider** that adds the IExtension to the service instance when it is created, and an **Attribute** that implements **IContractBehavior** that we can use on the service.

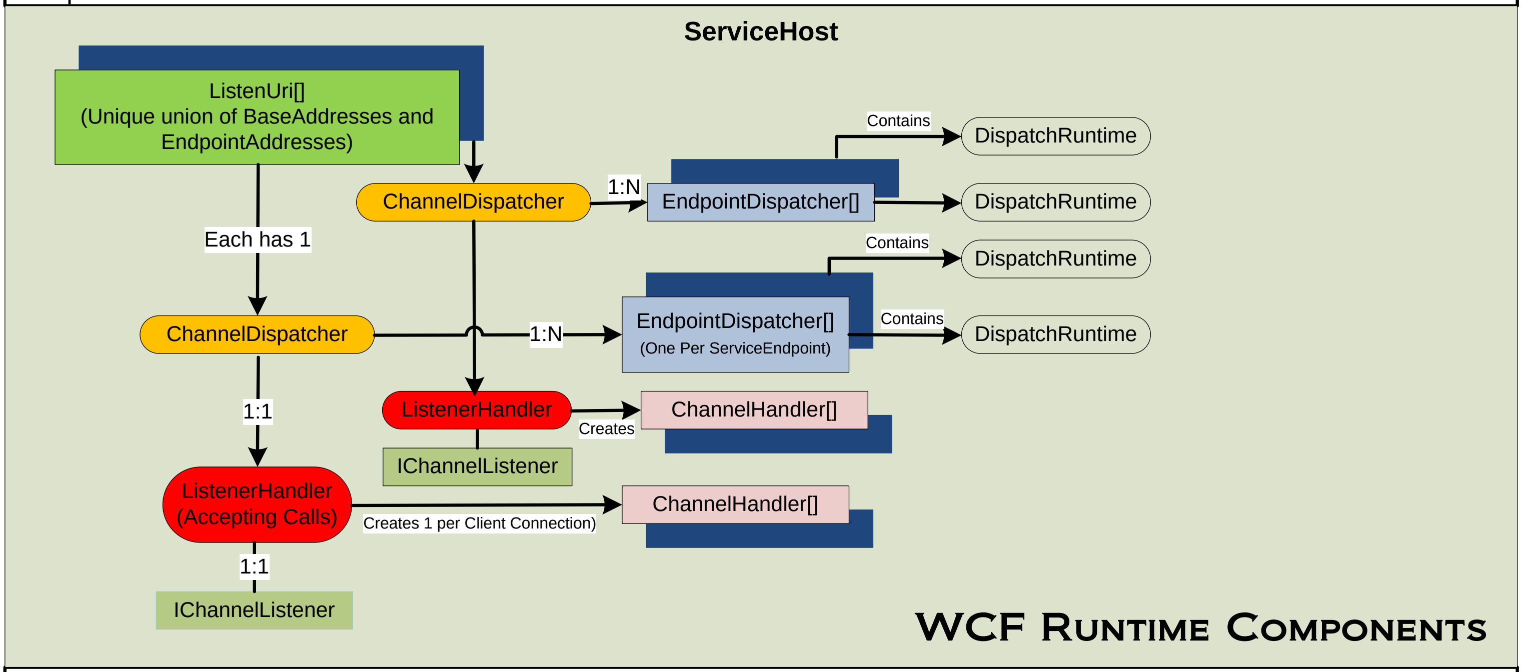
IExtension<InstanceContext>

IInstanceContextProvider

IInstanceContextInitializer

Credits: http://www.danrigsby.com/blog/index.php/2008/05/24/extending-wcf-instancecontext-to-store-custom-state/

WCF Runtime is BUILT according to "Specification" (The ServiceDescription), and once built, it is IMMUTABLE!
When Open() is called on a ServiceHost, the runtime is built from the ServiceDescription and the list of ServiceEndpoints specified. Once the Open() step completes, the runtime is immutable and any attempt to modify any components results in an exception.



Based on your base addresses [set when creating ServiceHost(), one per transport type] and EndpointAddress, the service host contains a list of listen URI's [Unique union of Base Addresses and EndpointAddresses] and each URI will have its own listener. There is a ChannelDispatcher associated with each ListenUri and each ChannelDispatcher can contain 1 or more EndpointDispatcher object. Most often a ChannelDispatcher will contain exactly one EndpointDispatcher but for the case where the user has hosted multiple endpoints all using one listen URI (**The ServiceEndpoints have same Contract, same Binding object, different EndpointAddresses, and the same ListenUri)**. In that case, there will be exactly one ChannelDispatcher for that Uri but it will contain all the EndpointDispatchers for all the ServiceEndpoints. Basically, each ServiceEndpoint added to the host will have one EndpointDispatcher associated with it. Each EndpointDispatcher contains the DispatchRuntime class that contains extension points InstanceContextProvider, InstanceContextInitializers etc.

Each ChannelDispatcher contains one ListenerHandler which, as its name suggests, handles the underlying IListener. The ListenerHandler is the one determines when to accept a new channel (Client's Connection). When the ListenerHandler accepts a new Channel, it will wrap it with a ChannelHandler object. ChannelHandler handles messages off one particular channel in the same way ListenerHandler handles connection requests from one listener.

Let's see a sample breakdown of how a channel is accepted and messages are read till the channel closes. Note that I have tried to simplify this as explaining the actual process would be very verbose.

1. ServiceHost is opened and all Listeners are ready to accept.
2. ListenerHandler object opens and issues a pending Accept() call.
3. When a client connects to the listener a callback is fired on the ListenerHandler notifying availability of channel.
4. The ListenerHandler creates a ChannelHandler object and associated the channel with that handler.
5. The ChannelHandler object is registered. Once registered, the ChannelHandler will try to read messages from that channel. In other words, it issues a pending Receive on that channel.
6. When the client sends a message, ChannelHandler is notified of that message via a callback.
7. The handler then determines the EndpointDispatcher that the message is addressed to.
8. Then using the DispatchRuntime of that EndpointDispatcher, it determines the Operation that the runtime should invoke and schedules the invocation.
9. It keeps reading Messages from the channel till null is received (Denoting a client initiated Close())

So we refer to the **ListenerHandler's** way of reading channels as **ChannelPump** and similarly the **ChannelHandler's** behavior is referred as **MessagePump**. It's the responsibility of the individual handlers to ensure that their respective pumps are never stalled. A stalled pump will mean either no new connections are accepted or no new message will be read off a channel.

The **ListenerHandler** uses the **ServiceThrottleBehavior.MaxConcurrenctConnections** throttle to determine when to pause and restart the **ChannelPump** and the **ChannelHandler** will use the **ServiceThrottleBehavior.MaxConcurrenctCalls** throttle to determine when to pause and restart the **MessagePump**.

Credit: http://blogs.msdn.com/mahjayar/archive/2006/12/22/wcf-runtime-blocks.aspx (WCF Runtime Components)

WCF Shared ListenUri

An example showing the ability for multiple ServiceEndpoint's on a service to share a common listen URI.

```
[ServiceContract]
public interface IService
{
	[OperationContract]
	String SayHello(String s);
}
```

Now we create a ServiceHost which will expose two endpoints.

```
Binding binding = new NetTcpBinding(); // Binding must be the same (same Binding type and same Binding object)
Uri commonListenUri = "net.tcp://localhost:9999/CommonListen";
ServiceHost sh = new ServiceHost(typeof(MyIServiceImplementation), new Uri("net.tcp://localhost:8888/Sample)); // this Uri is the baseAddress

sh.AddServiceEndpoint(typeof(IService), binding, "/Endpoint1", commonListenUri);
sh.AddServiceEndpoint(typeof(IService), binding, "/Endpoint2", commonListenUri); // <== Note we are reusing the same binding object
```

EndpointAddresses: (Just URNs if ListenUri is used)
net.tcp://localhost:8888/Sample/Endpoint1
net.tcp://localhost:8888/Sample/Endpoint2

So to enable sharing of listen uri, two endpoints needs to **share the binding object** and pass in the listen uri as an argument to AddServiceEndpoint call. Its that simple.

Now lets see how a client can connect to endpoint "/Endpoint1" and execute an operation.

```
using (ChannelFactory<IService> factory= new ChannelFactory<IService>(new NetTcpBinding(), new EndpointAddress("net.tcp://localhost:8888/Endpoint1")))
{
	factory.Endpoint.Behaviors.Add(new ViaUriBehavior("net.tcp/localhost:9999/CommonListen"));
	IService proxy = factory.CreateChannel();
	Console.WriteLine(proxy.SayHello("Hello from client"));
}
```

Adding a ViaBehavior marks the underlying transport to connect to that Uri but actual **EndpointAddress specified in the factory is sent as the "To" header in the message. On the service, we now have one listener listening on the listen uri and decides which endpoint should process the message based on the "To" header----THIS IS the FLEXIBLE/POWER of WCF; now you can build some semantics into the “To” header, doing different things based on the “To” header.**

If you are using svcutil to generate the proxy class then you would add the ViaUriBehavior on [**GeneratedProxyObjectInstance**].Endpoint.Behaviors.

Note: To share a listen uri, all the endpoints in the group should use the **same binding object** and should specify the same binding object in the AddServiceEndpoint call, if not, exception would be thrown, like the following:

```
// Don’t Do It.
sh.AddServiceEndpoint(typeof(IService), new NetTcpBinding(), "/Endpoint1", commonListenUri);
sh.AddServiceEndpoint(typeof(IService), new NetTcpBinding(), "/Endpoint2", commonListenUri);  // using a different Binding object.
```

Credit: http://blogs.msdn.com/mahjayar/archive/2006/03/02/542339.aspx

WCF RM (Reliable Messaging) See How it Works

Seq ...4e11: Message 1

```
<soapenv:Envelope>
    <soapenv:Header>
        <wsa:MessageID> ... </wsa:MessageID>
        <wsa:To> ... </wsa:To>
        <wsa:Action> ... </wsa:Action>
        <wsa:From> ... </wsa:From>
        <wsrm:Sequence soapenv:mustUnderstand="1">
            <wsu:Identifier> http://www.ibm.com/guid/a8f7151a091b50a42b38e04437774e11 </wsu:Identifier>
            <wsrm:MessageNumber>1</wsrm:MessageNumber>
        </wsrm:Sequence>
    </soapenv:Header>
    <soapenv:Body> ... </soapenv:Body>
</soapenv:Envelope>
```

Seq ...4e11: Message 2

```
<soapenv:Envelope>
    <soapenv:Header>
        <wsa:MessageID> ... </wsa:MessageID>
        <wsa:To> ... </wsa:To>
        <wsa:Action> ... </wsa:Action>
        <wsa:From> ... </wsa:From>
        <wsrm:Sequence soapenv:mustUnderstand="1">
            <wsu:Identifier> http://www.ibm.com/guid/a8f7151a091b50a42b38e04437774e11 </wsu:Identifier>
            <wsrm:MessageNumber>2</wsrm:MessageNumber>
        </wsrm:Sequence>
    </soapenv:Header>
    <soapenv:Body> ... </soapenv:Body>
</soapenv:Envelope>
```

Seq ...4e11: Message 3

```
<soapenv:Envelope>
    <soapenv:Header>
        <wsa:MessageID> ... </wsa:MessageID>
        <wsa:To> ... </wsa:To>
        <wsa:Action> ... </wsa:Action>
        <wsa:From> ... </wsa:From>
        <wsrm:Sequence soapenv:mustUnderstand="1">
            <wsu:Identifier> http://www.ibm.com/guid/a8f7151a091b50a42b38e04437774e11 </wsu:Identifier>
            <wsrm:MessageNumber>3</wsrm:MessageNumber>
            <wsrm:LastMessage/>
        </wsrm:Sequence>
    </soapenv:Header>
    <soapenv:Body> ... </soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope>
```

Ack: Seq ...4e11: Message 1 and 3

```
    <soapenv:Header>
        <wsa:MessageID> ... </wsa:MessageID>
        <wsa:To> ... </wsa:To>
        <wsa:Action> ... </wsa:Action>
        <wsa:From> ... </wsa:From>
        <wsrm:SequenceAcknowledgement>
            <wsuu:Identifier> http://www.ibm.com/guid/a8f7151a091b50a42b38e04437774e11 </wsuu:Identifier>
            <wsrm:AcknowledgementRange Lower="1" Upper="1"/>
            <wsrm:AcknowledgementRange Lower="3" Upper="3"/>
        </wsrm:SequenceAcknowledgement>
    </soapenv:Header>
    <soapenv:Body> ... </soapenv:Body> (response body)
</soapenv:Envelope>
```

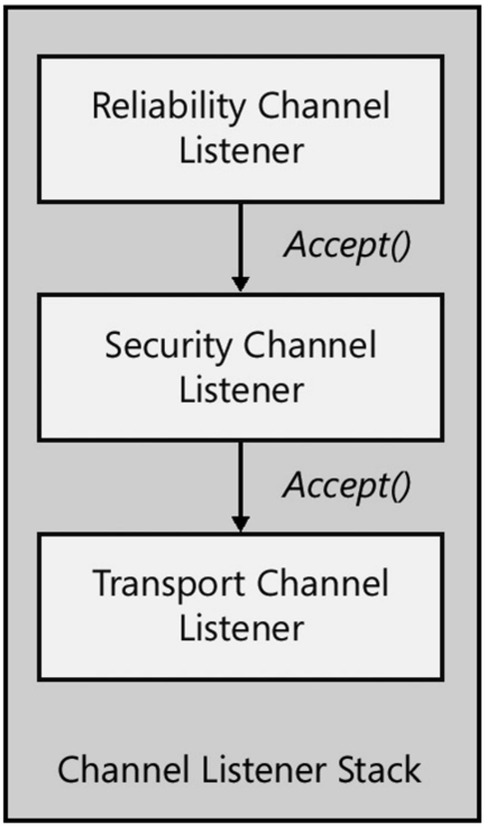
Ack: Seq ...4e11: Message 1, 2, and 3

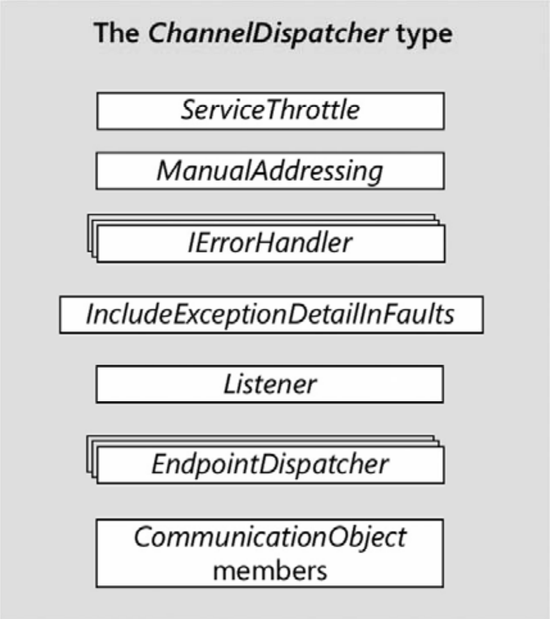
```
<soapenv:Envelope>
    <soapenv:Header>
        <wsa:MessageID> ... </wsa:MessageID>
        <wsa:To> ... </wsa:To>
        <wsa:Action> ... </wsa:Action>
        <wsa:From> ... </wsa:From>
        <wsrm:SequenceAcknowledgement>
            <wsuu:Identifier> http://www.ibm.com/guid/a8f7151a091b50a42b38e04437774e11 </wsuu:Identifier>
            <wsrm:AcknowledgementRange Lower="1" Upper="3"/>
        </wsrm:SequenceAcknowledgement>
    </soapenv:Header>
    <soapenv:Body> ... </soapenv:Body> (response body)
</soapenv:Envelope>
```

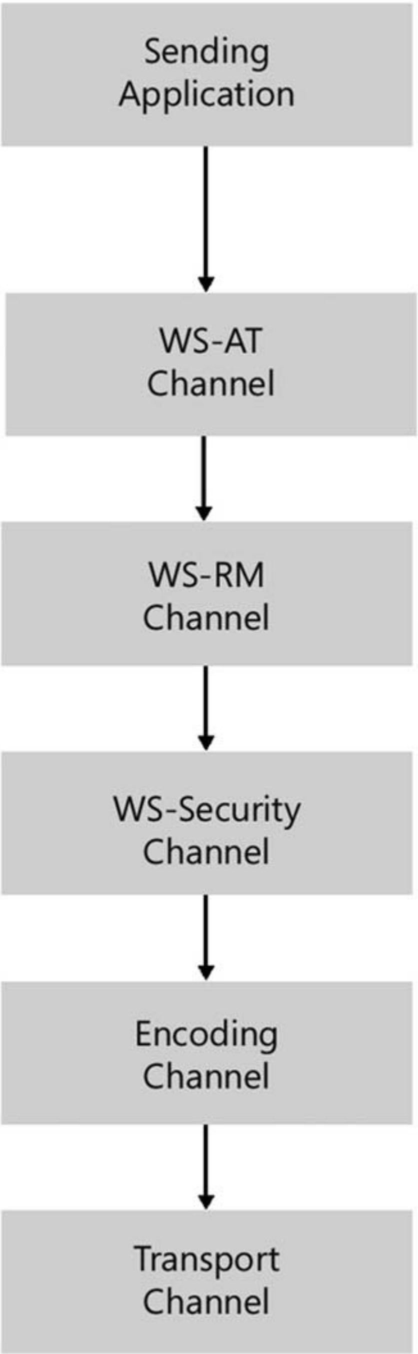
Indigo is a communication platform, that developers use to write computer programs that exchange information with other programs (on the same machine, across the internet, etc). Programs send *Messages* to each other over *Channels*. I’m responsible for making sure these Channels get the Messages from A to B .

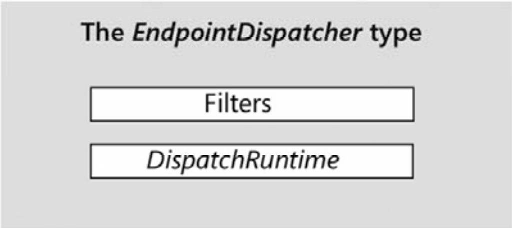
Credit: http://kennyw.com/work/indigo/5

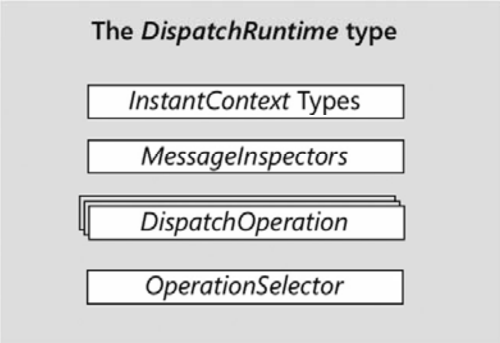
Binding: A binding is simply an ordered list of **binding elements**. Credit: http://kennyw.com/work/indigo/4

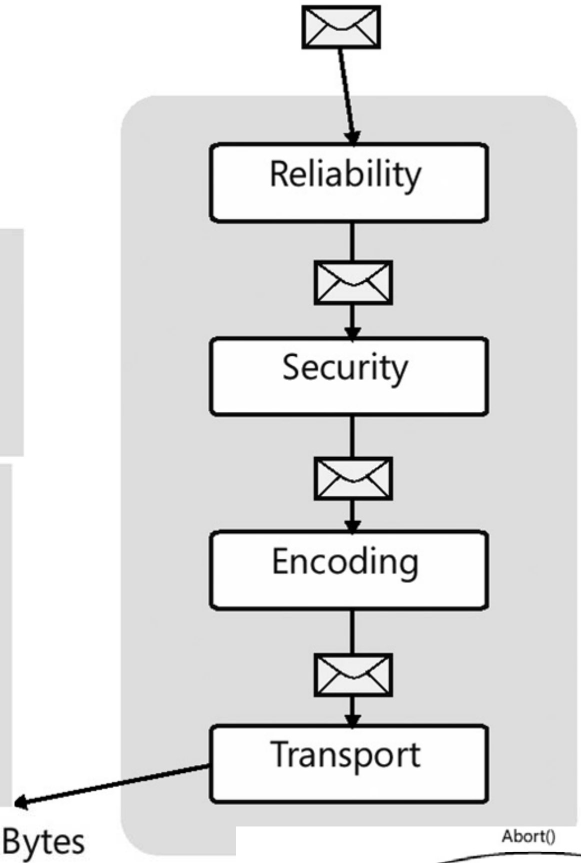

Reliability Channel Listener
Accept()
Security Channel Listener
Accept()
Transport Channel Listener
Channel Listener Stack

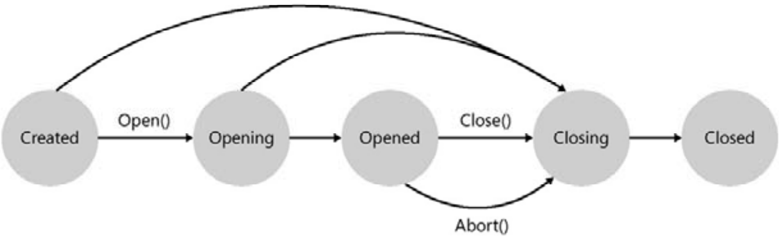

The ChannelDispatcher type
ServiceThrottle
ManualAddressing
IErrorHandler
IncludeExceptionDetailInFaults
Listener
EndpointDispatcher
CommunicationObject members

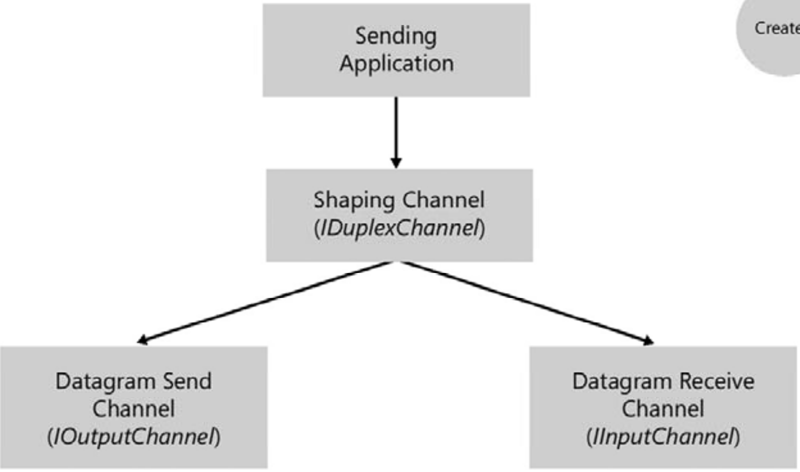

Sending Application
WS-AT Channel
WS-RM Channel
WS-Security Channel
Encoding Channel
Transport Channel

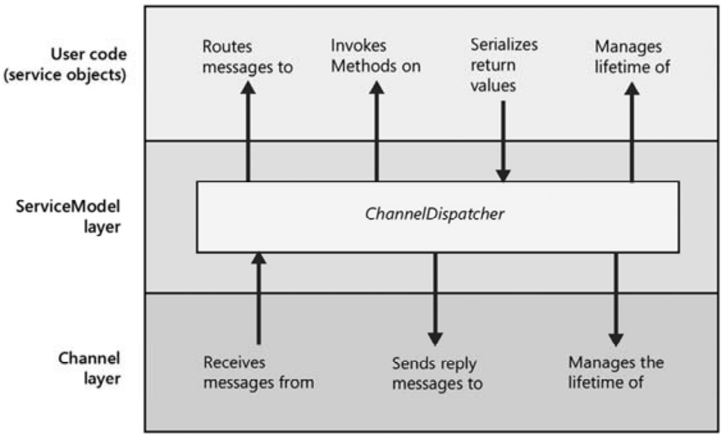

The EndpointDispatcher type
Filters
DispatchRuntime

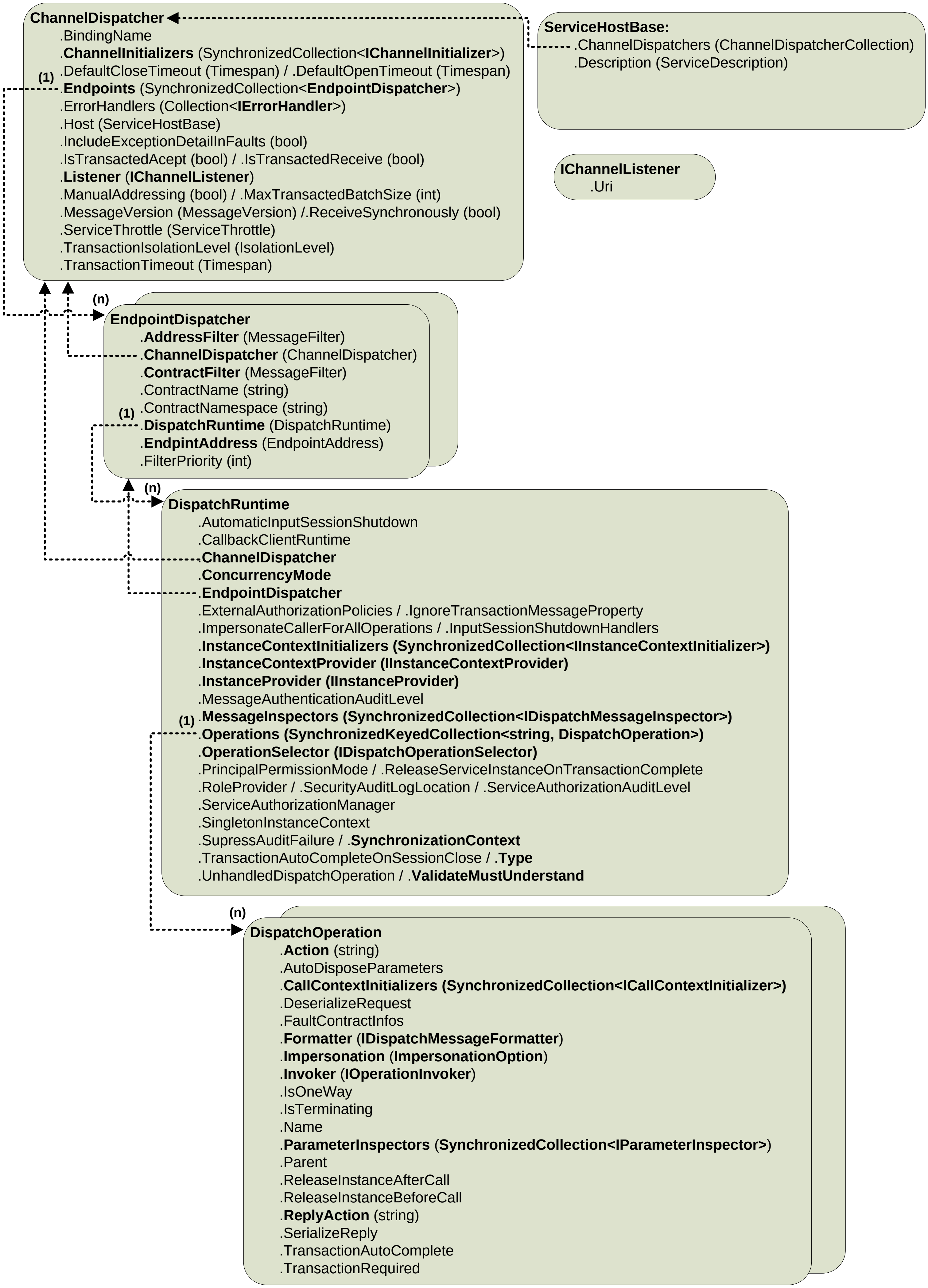

The DispatchRuntime type
InstantContext Types
MessageInspectors
DispatchOperation
OperationSelector


Reliability
Security
Encoding
Transport
Bytes


Abort()
Created
Open()
Opening
Opened
Close()
Closing
Closed
Abort()


Sending Application
Shaping Channel (IDuplexChannel)
Datagram Send Channel (IOutputChannel)
Datagram Receive Channel (IInputChannel)


User code (service objects)
Routes messages to
Invokes Methods on
Serializes return values
Manages lifetime of
ServiceModel layer
ChannelDispatcher
Channel layer
Receives messages from
Sends reply messages to
Manages the lifetime of



ChannelDispatcher

.BindingName

.**ChannelInitializers** (SynchronizedCollection<**IChannelInitializer**>)

.DefaultCloseTimeout (Timespan) / .DefaultOpenTimeout (Timespan)

.**Endpoints** (SynchronizedCollection<**EndpointDispatcher**>)

.ErrorHandlers (Collection<**IErrorHandler**>)

.Host (ServiceHostBase)

.IncludeExceptionDetailInFaults (bool)

.IsTransactedAcept (bool) / .IsTransactedReceive (bool)

.**Listener** (**IChannelListener**)

.ManualAddressing (bool) / .MaxTransactedBatchSize (int)

.MessageVersion (MessageVersion) /.ReceiveSynchronously (bool)

.ServiceThrottle (ServiceThrottle)

.TransactionIsolationLevel (IsolationLevel)

.TransactionTimeout (Timespan)

ServiceHostBase:

.ChannelDispatchers (ChannelDispatcherCollection)

.Description (ServiceDescription)

IChannelListener

.Uri

EndpointDispatcher

.**AddressFilter** (MessageFilter)

.**ChannelDispatcher** (ChannelDispatcher)

.**ContractFilter** (MessageFilter)

.ContractName (string)

.ContractNamespace (string)

.**DispatchRuntime** (DispatchRuntime)

.**EndpintAddress** (EndpointAddress)

.FilterPriority (int)

DispatchRuntime

.AutomaticInputSessionShutdown

.CallbackClientRuntime

.**ChannelDispatcher**

.**ConcurrencyMode**

.**EndpointDispatcher**

.ExternalAuthorizationPolicies / .IgnoreTransactionMessageProperty

.ImpersonateCallerForAllOperations / .InputSessionShutdownHandlers

.**InstanceContextInitializers (SynchronizedCollection<IInstanceContextInitializer>)**

.**InstanceContextProvider (IInstanceContextProvider)**

.**InstanceProvider (IInstanceProvider)**

.MessageAuthenticationAuditLevel

.**MessageInspectors (SynchronizedCollection<IDispatchMessageInspector>)**

.**Operations (SynchronizedKeyedCollection<string, DispatchOperation>)**

.**OperationSelector (IDispatchOperationSelector)**

.PrincipalPermissionMode / .ReleaseServiceInstanceOnTransactionComplete

.RoleProvider / .SecurityAuditLogLocation / .ServiceAuthorizationAuditLevel

.ServiceAuthorizationManager

.SingletonInstanceContext

.SupressAuditFailure / .**SynchronizationContext**

.TransactionAutoCompleteOnSessionClose / .**Type**

.UnhandledDispatchOperation / .**ValidateMustUnderstand**

DispatchOperation

.**Action** (string)

.AutoDisposeParameters

.**CallContextInitializers (SynchronizedCollection<ICallContextInitializer>)**

.DeserializeRequest

.FaultContractInfos

.**Formatter** (**IDispatchMessageFormatter**)

.**Impersonation** (**ImpersonationOption**)

.**Invoker** (**IOperationInvoker**)

.IsOneWay

.IsTerminating

.Name

.**ParameterInspectors** (**SynchronizedCollection<IParameterInspector>**)

.Parent

.ReleaseInstanceAfterCall

.ReleaseInstanceBeforeCall

.**ReplyAction** (string)

.SerializeReply

.TransactionAutoComplete

.TransactionRequired

Binding

.BuildChannelFactory<TChannel>(params object[])
.BuildChannelFactory<Tchannel>(BindingParameterCollection)
.BuildChannelListener<Tchannel>(params object[])
.BuildChannelListener<Tchannel>(BindingParameterCollection)
.BuildChannelListener<Tchannel>(uri, params object[])
.CanBuildChannelFactory<Tchannel>(params object[])
.CanBuildChannelListener<Tchannel>(params object[])
.CreateBindingElements()
.GetProperty<T>(BindingParameterCollection)
.CloseTimeout
.MessageVersion
.Name
.Namespace
.OpenTimeout
.ReceiveTimeout
.Scheme
.SendTimeout

Steps to Create a New Custom Channel

```
public class MyRequestChannel :ChannelBase, IRequestChannel { // TX
  private IRequestChannel InnerChannel {get;set;}
  public MyRequestChannel(ChannelManagerBase channleManager,
                         IRequestChannel innerChannel) : base(channleManager) {
    this.InnerChannel = innerChannel;
  }
  protected override void OnAbort() {
    Console.WriteLine("MyRequestChannel.OnAbort()");
    this.InnerChannel.Abort();
  }
  protected override IAsyncResult OnBeginClose(TimeSpan timeout,
                                              AsyncCallback callback,
                                              object state) {
    Console.WriteLine("MyRequestChannel.OnBeginClose()");
    return this.InnerChannel.BeginClose(timeout, callback, state);
  }
  protected override IAsyncResult OnBeginOpen(TimeSpan timeout,
                                              AsyncCallback callback,
                                              object state) {
    Console.WriteLine("MyRequestChannel.OnBeginOpen()");
    return this.InnerChannel.BeginOpen(timeout, callback, state);
  }
  protected override void OnClose(TimeSpan timeout) {
    Console.WriteLine("MyRequestChannel.OnClose()");
    this.Close(timeout);
  }
  protected override void OnEndClose(IAsyncResult result) {
    Console.WriteLine("MyRequestChannel.OnEndClose()");
    this.InnerChannel.EndClose(result);
  }
  protected override void OnEndOpen(IAsyncResult result) {
    Console.WriteLine("MyRequestChannel.OnEndOpen()");
    this.InnerChannel.EndOpen(result);
  }
  protected override void OnOpen(TimeSpan timeout) {
    Console.WriteLine("MyRequestChannel.OnOpen()");
    this.InnerChannel.Open(timeout);
  }
  public IAsyncResult BeginRequest(Message message,
                                  TimeSpan timeout,
                                  AsyncCallback callback,
                                  object state) {
    Console.WriteLine("MyRequestChannel.BeginRequest()");
    return this.BeginRequest(message, timeout, callback, state);
  }
  public IAsyncResult BeginRequest(Message message,
                                   AsyncCallback callback,
                                  object state) {
    Console.WriteLine("MyRequestChannel.BeginRequest()");
    return this.InnerChannel.BeginRequest(message, callback, state);
  }
  public Message EndRequest(IAsyncResult result) {
    Console.WriteLine("MyRequestChannel.EndRequest()");
    return this.InnerChannel.EndRequest(result);
  }
  public EndpointAddress RemoteAddress {
    get {
      Console.WriteLine("MyRequestChannel.RemoteAddress");
      return this.InnerChannel.RemoteAddress;
    }
  }
  public Message Request(Message message, TimeSpan timeout) {
    Console.WriteLine("MyRequestChannel.Request()");
    return this.InnerChannel.Request(message, timeout);
  }
  public Message Request(Message message) {
    Console.WriteLine("MyRequestChannel.Request()");
    return this.InnerChannel.Request(message);
  }
  public Uri Via {
    get {
      Console.WriteLine("MyRequestChannel.Via)");
      return this.InnerChannel.Via;
    }
  }
}
```

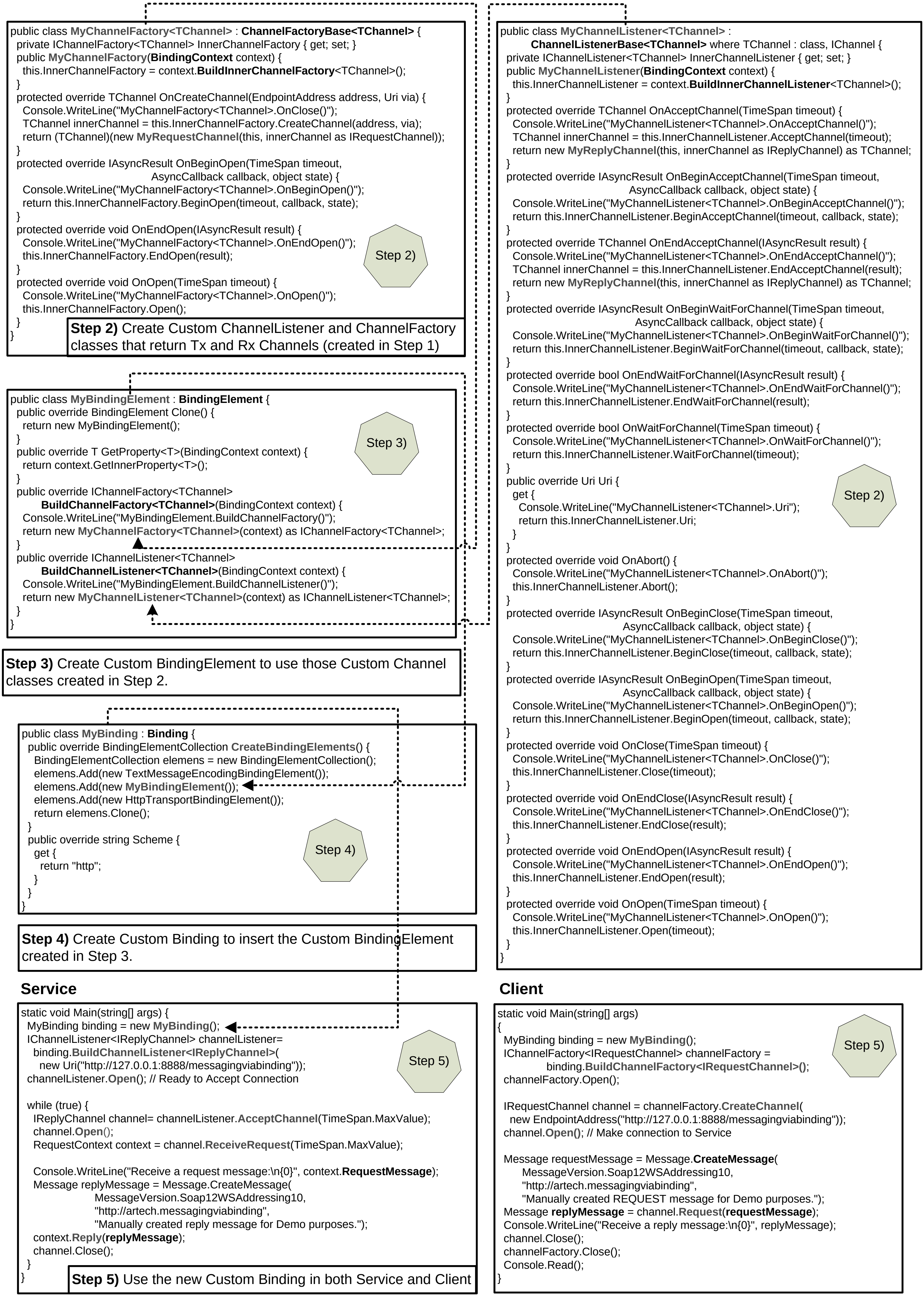
Step 1)

Step 1) Create Two Custom Channels: Tx & Rx Channels

```
public class MyReplyChannel: ChannelBase, IReplyChannel { // RX Channel
  private IReplyChannel InnerChannel { get; set; }
  public MyReplyChannel(ChannelManagerBase channelManager,
     IReplyChannel innerChannel):base(channelManager) {
    this.InnerChannel = innerChannel;
  }
  protected override void OnAbort() {
    Console.WriteLine("MyReplyChannel.OnAbort()"); this.InnerChannel.Abort();
  }
  protected override IAsyncResult OnBeginClose(TimeSpan timeout,
                                     AsyncCallback callback, object state) {
    Console.WriteLine("MyReplyChannel.OnBeginClose()");
    return this.InnerChannel.BeginClose(timeout, callback, state);
  }
  protected override IAsyncResult OnBeginOpen(TimeSpan timeout,
                                     AsyncCallback callback, object state) {
    Console.WriteLine("MyReplyChannel.OnBeginOpen()");
    return this.InnerChannel.BeginOpen(timeout, callback, state);
  }
  protected override void OnClose(TimeSpan timeout) {
    Console.WriteLine("MyReplyChannel.OnClose()"); this.Close(timeout);
  }
  protected override void OnEndClose(IAsyncResult result) {
    Console.WriteLine("MyReplyChannel.OnEndClose()");
    this.InnerChannel.EndClose(result);
  }
  protected override void OnEndOpen(IAsyncResult result) {
    Console.WriteLine("MyReplyChannel.OnEndOpen()");
    this.InnerChannel.EndOpen(result);
  }
  protected override void OnOpen(TimeSpan timeout) {
    Console.WriteLine("MyReplyChannel.OnOpen()");
    this.InnerChannel.Open(timeout);
  }
  public IAsyncResult BeginReceiveRequest(TimeSpan timeout,
                                  AsyncCallback callback, object state) {
    Console.WriteLine("MyReplyChannel.BeginReceiveRequest()");
    return this.InnerChannel.BeginReceiveRequest(timeout, callback, state);
  }
  public IAsyncResult BeginReceiveRequest(AsyncCallback callback, object state) {
    Console.WriteLine("MyReplyChannel.BeginReceiveRequest()");
    return this.InnerChannel.BeginReceiveRequest(callback, state);
  }
  public IAsyncResult BeginTryReceiveRequest(TimeSpan timeout,
                                    AsyncCallback callback, object state) {
    Console.WriteLine("MyReplyChannel.BeginTryReceiveRequest()");
    return this.InnerChannel.BeginTryReceiveRequest(timeout, callback, state);
  }
  public IAsyncResult BeginWaitForRequest(TimeSpan timeout,
                                  AsyncCallback callback, object state) {
    Console.WriteLine("MyReplyChannel.BeginWaitForRequest()");
    return this.InnerChannel.BeginWaitForRequest(timeout, callback, state);
  }
  public RequestContext EndReceiveRequest(IAsyncResult result) {
    Console.WriteLine("MyReplyChannel.EndReceiveRequest()");
    return this.InnerChannel.EndReceiveRequest(result);
  }
  public bool EndTryReceiveRequest(IAsyncResult result,
                           out RequestContext context) {
    Console.WriteLine("MyReplyChannel.EndTryReceiveRequest()");
    return this.InnerChannel.EndTryReceiveRequest(result, out context);
  }
  public bool EndWaitForRequest(IAsyncResult result) {
    Console.WriteLine("MyReplyChannel.EndWaitForRequest()");
    return this.InnerChannel.EndWaitForRequest(result);
  }
  public System.ServiceModel.EndpointAddress LocalAddress {
    get {
      Console.WriteLine("MyReplyChannel.LocalAddress");
      return this.InnerChannel.LocalAddress;
    }
  }
  public RequestContext ReceiveRequest(TimeSpan timeout) {
    Console.WriteLine("MyReplyChannel.ReceiveRequest()");
    return this.InnerChannel.ReceiveRequest(timeout);
  }
  public RequestContext ReceiveRequest() {
    Console.WriteLine("MyReplyChannel.ReceiveRequest()");
    return this.InnerChannel.ReceiveRequest();
  }
  public bool TryReceiveRequest(TimeSpan timeout, out RequestContext context) {
    Console.WriteLine("MyReplyChannel.TryReceiveRequest()");
    return this.InnerChannel.TryReceiveRequest(timeout, out context);
  }
  public bool WaitForRequest(TimeSpan timeout) {
    Console.WriteLine("MyReplyChannel.WaitForRequest()");
    return this.InnerChannel.WaitForRequest(timeout);
  }
}
```

Step 1)

http://www.cnblogs.com/artech/archive/2008/07/09/1238626.html [原创]WCF后续之旅(2): 如何对Channel Layer进行扩展——创建自定义Channel



```
public class MyChannelFactory<TChannel> : ChannelFactoryBase<TChannel> {
  private IChannelFactory<TChannel> InnerChannelFactory { get; set; }
  public MyChannelFactory(BindingContext context) {
    this.InnerChannelFactory = context.BuildInnerChannelFactory<TChannel>();
  }
  protected override TChannel OnCreateChannel(EndpointAddress address, Uri via) {
    Console.WriteLine("MyChannelFactory<TChannel>.OnClose()");
    TChannel innerChannel = this.InnerChannelFactory.CreateChannel(address, via);
    return (TChannel)(new MyRequestChannel(this, innerChannel as IRequestChannel));
  }
  protected override IAsyncResult OnBeginOpen(TimeSpan timeout,
                                        AsyncCallback callback, object state) {
    Console.WriteLine("MyChannelFactory<TChannel>.OnBeginOpen()");
    return this.InnerChannelFactory.BeginOpen(timeout, callback, state);
  }
  protected override void OnEndOpen(IAsyncResult result) {
    Console.WriteLine("MyChannelFactory<TChannel>.OnEndOpen()");
    this.InnerChannelFactory.EndOpen(result);
  }
  protected override void OnOpen(TimeSpan timeout) {
    Console.WriteLine("MyChannelFactory<TChannel>.OnOpen()");
    this.InnerChannelFactory.Open();
  }
}
```

Step 2) Create Custom ChannelListener and ChannelFactory classes that return Tx and Rx Channels (created in Step 1)

```
public class MyBindingElement : BindingElement {
  public override BindingElement Clone() {
    return new MyBindingElement();
  }
  public override T GetProperty<T>(BindingContext context) {
    return context.GetInnerProperty<T>();
  }
  public override IChannelFactory<TChannel>
        BuildChannelFactory<TChannel>(BindingContext context) {
    Console.WriteLine("MyBindingElement.BuildChannelFactory()");
    return new MyChannelFactory<TChannel>(context) as IChannelFactory<TChannel>;
  }
  public override IChannelListener<TChannel>
        BuildChannelListener<TChannel>(BindingContext context) {
    Console.WriteLine("MyBindingElement.BuildChannelListener()");
    return new MyChannelListener<TChannel>(context) as IChannelListener<TChannel>;
  }
}
```

Step 3) Create Custom BindingElement to use those Custom Channel classes created in Step 2.

```
public class MyBinding : Binding {
  public override BindingElementCollection CreateBindingElements() {
    BindingElementCollection elemens = new BindingElementCollection();
    elemens.Add(new TextMessageEncodingBindingElement());
    elemens.Add(new MyBindingElement());
    elemens.Add(new HttpTransportBindingElement());
    return elemens.Clone();
  }
  public override string Scheme {
    get {
      return "http";
    }
  }
}
```

Step 4) Create Custom Binding to insert the Custom BindingElement created in Step 3.

Service

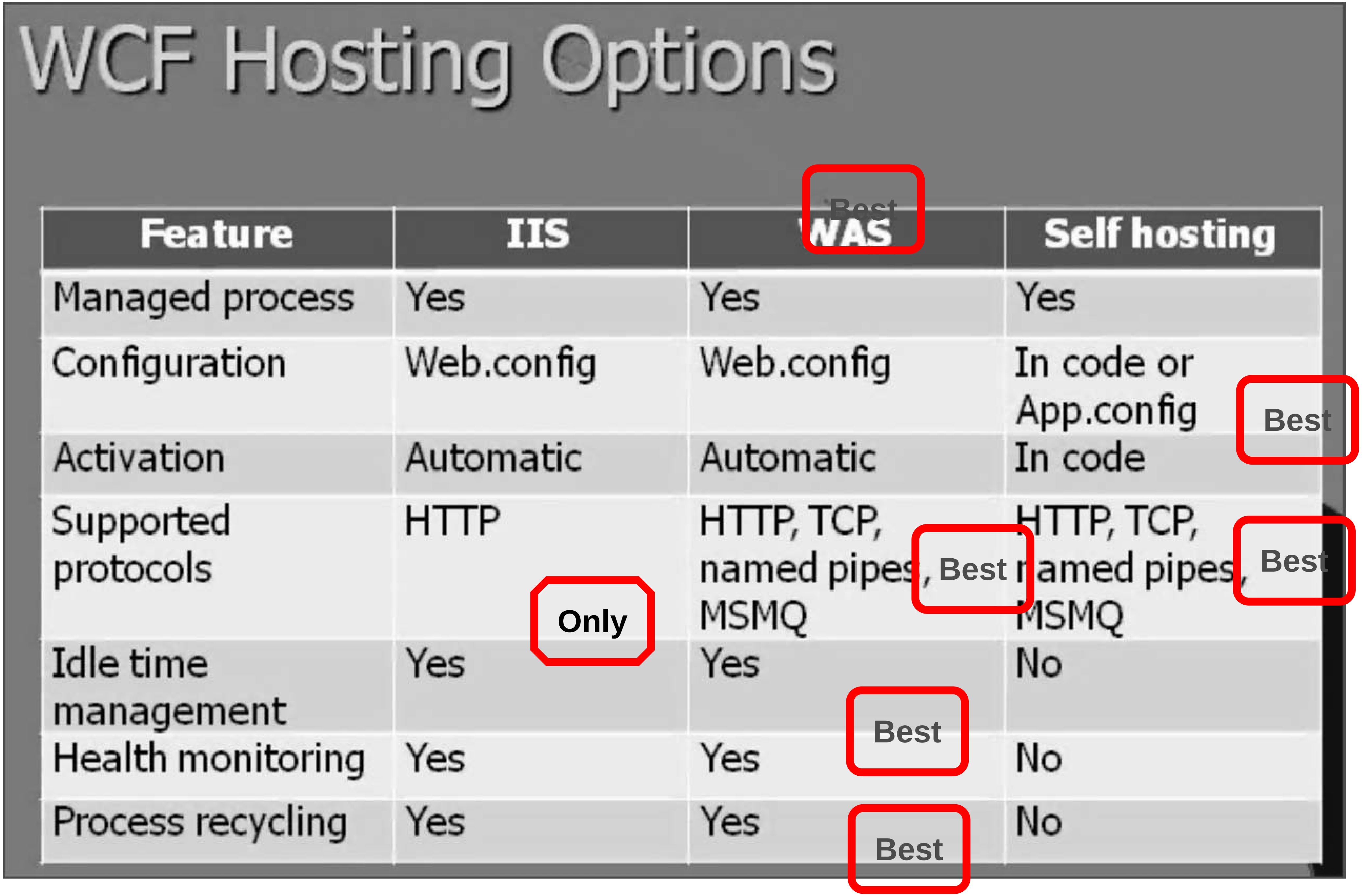
```
static void Main(string[] args) {
  MyBinding binding = new MyBinding();
  IChannelListener<IReplyChannel> channelListener=
    binding.BuildChannelListener<IReplyChannel>(
      new Uri("http://127.0.0.1:8888/messagingviabinding"));
  channelListener.Open(); // Ready to Accept Connection

  while (true) {
    IReplyChannel channel= channelListener.AcceptChannel(TimeSpan.MaxValue);
    channel.Open();
    RequestContext context = channel.ReceiveRequest(TimeSpan.MaxValue);

    Console.WriteLine("Receive a request message:\n{0}", context.RequestMessage);
    Message replyMessage = Message.CreateMessage(
                    MessageVersion.Soap12WSAddressing10,
                    "http://artech.messagingviabinding",
                    "Manually created reply message for Demo purposes.");
    context.Reply(replyMessage);
    channel.Close();
  }
}
```

Step 5) Use the new Custom Binding in both Service and Client

```
public class MyChannelListener<TChannel> :
        ChannelListenerBase<TChannel> where TChannel : class, IChannel {
  private IChannelListener<TChannel> InnerChannelListener { get; set; }
  public MyChannelListener(BindingContext context) {
    this.InnerChannelListener = context.BuildInnerChannelListener<TChannel>();
  }
  protected override TChannel OnAcceptChannel(TimeSpan timeout) {
    Console.WriteLine("MyChannelListener<TChannel>.OnAcceptChannel()");
    TChannel innerChannel = this.InnerChannelListener.AcceptChannel(timeout);
    return new MyReplyChannel(this, innerChannel as IReplyChannel) as TChannel;
  }
  protected override IAsyncResult OnBeginAcceptChannel(TimeSpan timeout,
                              AsyncCallback callback, object state) {
    Console.WriteLine("MyChannelListener<TChannel>.OnBeginAcceptChannel()");
    return this.InnerChannelListener.BeginAcceptChannel(timeout, callback, state);
  }
  protected override TChannel OnEndAcceptChannel(IAsyncResult result) {
    Console.WriteLine("MyChannelListener<TChannel>.OnEndAcceptChannel()");
    TChannel innerChannel = this.InnerChannelListener.EndAcceptChannel(result);
    return new MyReplyChannel(this, innerChannel as IReplyChannel) as TChannel;
  }
  protected override IAsyncResult OnBeginWaitForChannel(TimeSpan timeout,
                              AsyncCallback callback, object state) {
    Console.WriteLine("MyChannelListener<TChannel>.OnBeginWaitForChannel()");
    return this.InnerChannelListener.BeginWaitForChannel(timeout, callback, state);
  }
  protected override bool OnEndWaitForChannel(IAsyncResult result) {
    Console.WriteLine("MyChannelListener<TChannel>.OnEndWaitForChannel()");
    return this.InnerChannelListener.EndWaitForChannel(result);
  }
  protected override bool OnWaitForChannel(TimeSpan timeout) {
    Console.WriteLine("MyChannelListener<TChannel>.OnWaitForChannel()");
    return this.InnerChannelListener.WaitForChannel(timeout);
  }
  public override Uri Uri {
    get {
      Console.WriteLine("MyChannelListener<TChannel>.Uri");
      return this.InnerChannelListener.Uri;
    }
  }
  protected override void OnAbort() {
    Console.WriteLine("MyChannelListener<TChannel>.OnAbort()");
    this.InnerChannelListener.Abort();
  }
  protected override IAsyncResult OnBeginClose(TimeSpan timeout,
                              AsyncCallback callback, object state) {
    Console.WriteLine("MyChannelListener<TChannel>.OnBeginClose()");
    return this.InnerChannelListener.BeginClose(timeout, callback, state);
  }
  protected override IAsyncResult OnBeginOpen(TimeSpan timeout,
                              AsyncCallback callback, object state) {
    Console.WriteLine("MyChannelListener<TChannel>.OnBeginOpen()");
    return this.InnerChannelListener.BeginOpen(timeout, callback, state);
  }
  protected override void OnClose(TimeSpan timeout) {
    Console.WriteLine("MyChannelListener<TChannel>.OnClose()");
    this.InnerChannelListener.Close(timeout);
  }
  protected override void OnEndClose(IAsyncResult result) {
    Console.WriteLine("MyChannelListener<TChannel>.OnEndClose()");
    this.InnerChannelListener.EndClose(result);
  }
  protected override void OnEndOpen(IAsyncResult result) {
    Console.WriteLine("MyChannelListener<TChannel>.OnEndOpen()");
    this.InnerChannelListener.EndOpen(result);
  }
  protected override void OnOpen(TimeSpan timeout) {
    Console.WriteLine("MyChannelListener<TChannel>.OnOpen()");
    this.InnerChannelListener.Open(timeout);
  }
}
```

Client

```
static void Main(string[] args)
{
  MyBinding binding = new MyBinding();
  IChannelFactory<IRequestChannel> channelFactory =
             binding.BuildChannelFactory<IRequestChannel>();
  channelFactory.Open();

  IRequestChannel channel = channelFactory.CreateChannel(
    new EndpointAddress("http://127.0.0.1:8888/messagingviabinding"));
  channel.Open(); // Make connection to Service

  Message requestMessage = Message.CreateMessage(
       MessageVersion.Soap12WSAddressing10,
       "http://artech.messagingviabinding",
       "Manually created REQUEST message for Demo purposes.");
  Message replyMessage = channel.Request(requestMessage);
  Console.WriteLine("Receive a reply message:\n{0}", replyMessage);
  channel.Close();
  channelFactory.Close();
  Console.Read();
}
```



WCF Hosting Options

Feature	IIS	Best WAS	Self hosting
Managed process	Yes	Yes	Yes
Configuration	Web.config	Web.config	In code or App.config Best
Activation	Automatic	Automatic	In code
Supported protocols	HTTP Only	HTTP, TCP, named pipes, MSMQ Best	HTTP, TCP, named pipes, MSMQ Best
Idle time management	Yes	Yes	No
Health monitoring	Yes	Yes Best	No
Process recycling	Yes	Yes Best	No

What is GetProperty<T> for?

This is What Microsoft Says:

T **GetProperty**<**T**>() where **T** : class
Member of System.ServiceModel.Channels.IChannel

Summary: Returns a typed object requested, if present, from the appropriate layer in the channel stack.

Type Parameters: *T*: The typed object for which the method is querying.

Return Values: The typed object T requested if it is present or null if it is not.

Following is from a Blog:

Aggregated Interface Implementation

Let's say I have an object myObject and a few interfaces. However, myObject doesn't implement any of those interfaces, but it contains a list called *elements* that contain various other objects, each of which might implement those interfaces. Now lets say I want to get a handle to one of those interfaces - to make things more concrete, let's say we want the ISecurityProvider interface. Using the current WCF model, we would do something like this:

```
ISecurityProvider sec = myObject.GetProperty<ISecurityProvider>();
```

Which in turn would do something like this:

```
T GetProperty<T>() where T : class {
   T prop = null;
   foreach (object o in elements)  {
      if (o is T) {
         prop = (T)o;
         break;
      }
   }
   return prop;
}
```

(Remember, this is a simplified version. The actual WCF implementation is more complicated).

Now, what I would like to see, as a simplified and cleaner syntax, is the ability to do this:

```
ISecurityProvider sec = myObject as ISecurityProvider;
```

This is much clearer, adheres to the interface-implementation paradigm, and still allows me to make late-bound changes to my object's implementation. Ideally, the casting operation would internally call the GetProperty<T> method. The definition would go something like this:

```
public static explicit operator <T> (Aggregator agg) {
      return agg.GetProperty<T>();
}
```

http://weblogs.asp.net/avnerk/archive/2006/08/08/Aggregated-Interface-Implementation.aspx

VisualStudio Project Organizations

Basic Steps to Create WCF Service and Client

1) Create Service Interface decorated with [**ServiceContract**] and, optionally, specifying a Client Callback Interface.

2) Create **DataContracts** used by both Service Interface and Client Callback Interface.

3) Create a Service class that implements the Service Interface; decorate the Service class with [**ServiceBehavior**]

4) Create a project that host the above implemented Service.

5) Create app.conf to enable metadata

6) Create a project that implement the Service Client

7) Add a Service Reference to generate the Service's Proxy client

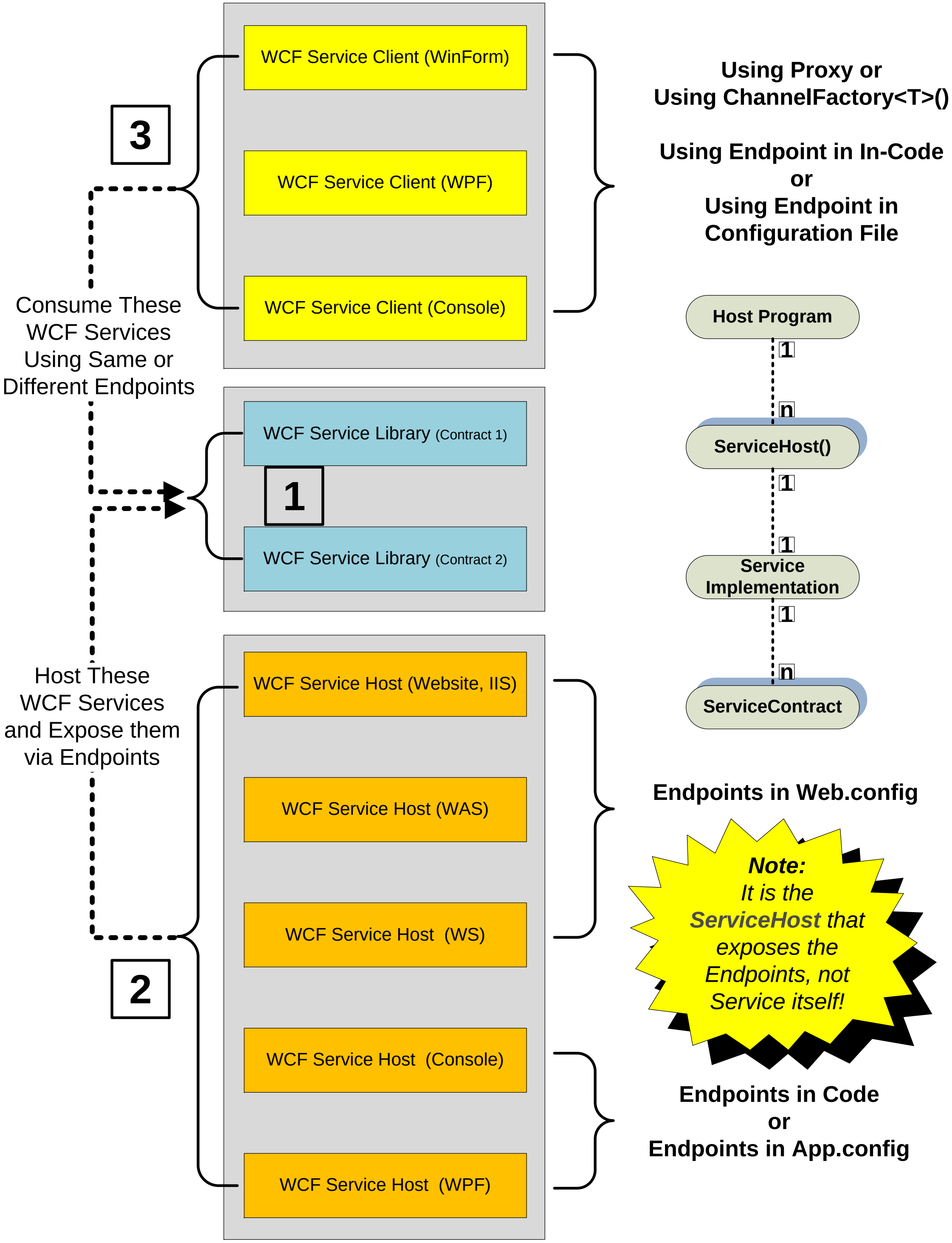
8) Instantiate the service proxy client and open it and start invoking Service operations.
======
Steps creating WCF Service and Client:
1) Create a Service Library Project
1') Create a Empty Project and add "WCF Service" class item

2) Rename app.conf to xapp.conf, so that VS won't create a host for the Service
3) Code IService interface
4) Implement the IService in a Service class
5) Create a Host project (like a Console)
6) Instantiate ServiceHost instance

7) Add Endpoints in Code (host.AddServiceEndpoint(...) or host.Description.Endpoints.Add(...))
7') Add Endpoints in Configuration File

8) Enable Service Metadata in Configuration file
9) Create Client Project
10) Add Service Reference
11) Instantiate the ServiceClient instance
12) Use the ServiceClient instance to invoke service operations

VisualStudio Projects



WCF Service Client (WinForm)

WCF Service Client (WPF)

WCF Service Client (Console)

3

Consume These WCF Services Using Same or Different Endpoints

Using Proxy or Using ChannelFactory<T>()

Using Endpoint in In-Code or Using Endpoint in Configuration File

WCF Service Library (Contract 1)

1

WCF Service Library (Contract 2)

Host Program

1

n

ServiceHost()

1

1

Service Implementation

1

n

ServiceContract

Host These WCF Services and Expose them via Endpoints

2

WCF Service Host (Website, IIS)

WCF Service Host (WAS)

WCF Service Host (WS)

Endpoints in Web.config

Note: *It is the **ServiceHost** that exposes the Endpoints, not Service itself!*

WCF Service Host (Console)

WCF Service Host (WPF)

Endpoints in Code or Endpoints in App.config

Many WCF runtime objects support adding new state, they do so through a common extension model which uses three interfaces:

1) **IExtensibleObject**<T>, that any extensible runtime object (the to-be-extended) must implement.
2) **IExtension**<T>, that the extensions (the to-extend) must implement, along with necessary custom states and logics.
3) **IExtensionCollection**<T>, type of Extensions property of IExtensionObject<T>, is a collection that holds all of the extensions for the instance of the extensible runtime object.

Step 1: To make an object of type T extensible, let it implement the IExtensibleObject<T> interface, which provides a place (its Extensions property) for the extensions to be attached . The use of unusual recursive, generic type definition is to make sure that only extensions designed for the extensible object can be added to extensible object's Extensions property.

1

```
public interface IExtensibleObject<T> where T : IExtensibleObject<T> {  // Yep, a Recursive Definition, so T is IExtensibleObject<T> ☺
   IExtensionCollection<T> Extensions { get; }  // extensions are collected here.
}
```

3

WCF Service Runtime Object Extension Model

An extension collection works like a standard collection, with the additional ability to find an extension in the collection, based on the type of the extension.

```
public interface IExtensionCollection<T> : ICollection<IExtension<T>>,  IEnumerable<IExtension<T>>, IEnumerable where T : IExtensibleObject<T> {
   E Find<E>();
   Collection<E> FindAll<E>();
}
```

3

Step 2: Have the extension implement IExtension<IExtensibleObject<T>>, which has two methods---Attach() and Detach().

This Extension can only be added to Extensible object of type IExtensibleObject<T>

2

```
public interface IExtension<T> where T : IExtensibleObject<T> {
   void Attach(T owner); // Attach() is called before the extension is about to be added to an extension collection
   void Detach(T owner); // Detach() is called before the extension is about removed from the collection.
}
```

Attach() is called before the extension is about added to an extension collection. **Detach()** is called before the extension is about removed from the collection. The owner (the extensible object that the extension is added to or removed from) is passed to Attach() and Detach().

Summary of Service Extension Model

1) To make "**T**" an Extensible Object, let "**T : IExtensibleObject<T>**"
2) Then, create an "**E** : **IExtension**<**T**>"

How to Use Them:

```
T t = new T(); // instantiate T
E e = new E(): // instantiate E
t.Extensions.Add(e); // Add the Extension
E ee = t.Extensions.Find<E>();  // Retrieve back the Extension
```

An Example:

T

```
class MyToBeExtendedObject : IExtensibleObject<MyToBeExtendedObject> { // ToBeExtended = Extensible
     …
}
class MyExtensionObject : IExtension<MyToBeExtendedObject> {// MyToBeExtendedObject is Extended/Attached to by this MyExtensionObject.
     ...
}
myToBeExtendedObject = new MyToBeExtendedObject();
myExtensionObject = new MyExtensionObject();

myToBeExtendedObject.Extensions.Add(myExtensionObject)); // Add the Extension object to the Extensible object
MyExtensionObject retrievedMyExtensionObject = myToBeExtendedObject.Extensions.Find<MyExtensionObject>()
```

The <T> Triangle

ExtensibleObjects in WCF

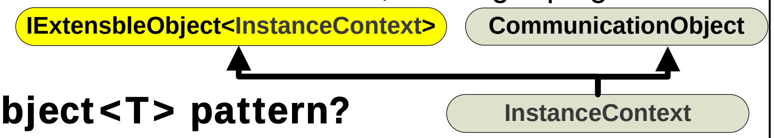
IClientChannel
IContextChannel
IDuplexContextChannel
InstanceContext
IServiceChannel
OperationContext
ServerHostBase

http://msdn.microsoft.com/en-us/library/ms733816.aspx (the IExtensibleObject<T> pattern)
http://msdn.microsoft.com/en-us/library/ms586697.aspx (the IExtensibleObject<T> interface)

Source: http://blogs.msdn.com/drnick/archive/2008/12/16/service-extension-model.aspx (Service Extension Model)

Updated: 3/12/2010

This is an example of extending (attaching) custom InstanceContext extension to an instanceContext object (which is an Extensible object), inside a custom InstanceContextInitializer, which get plug-in through an EndpointBehvior.



Q: Why we need to have an IExtensionObject<T> pattern?

A: Because we need a way to extend WCF runtime objects, such as instanceContext. But, **WE CAN NOT extend WCF runtime classes by the traditional OO sub-classing way**. Why? Because WCF runtime objects are instantiated for us by the WCF runtime. Their instantiations are opaque to us---there is no way for us to tell WCF runtime to instantiate our subclasses of WCF runtime classes.
So in order to extend these WCF runtime objects (ONLY AFTER their instantiations!), to add states and custom logics, etc, we need this IExtensionObject<T> pattern to do the job for us.

```
public class MyInstanceContextInitializer : IInstanceContextInitializer {

  public void Initialize(InstanceContext instanceContext, Message message) {
    MyInstanceContextExtension extension = new MyInstanceContextExtension();

    // Add your custom InstanceContext extension that will let you associate  state with this instancecontext
    instanceContext.Extensions.Add(extension);
  }
}
```

WCF Runtime gives us this instance of instanceContext!

Example of WCF Runtime Object Extension Model

3/12/2010

```
// Create an Extension that will attach to each InstanceContext and let it
// retrieve the Id or whatever state you want to associate
public class MyInstanceContextExtension : IExtension<InstanceContext> {

  String instanceId;   //Associate an Id with each Instance Created.

  public MyInstanceContextExtension() { this.instanceId = Guid.NewGuid().ToString(); }

  public String InstanceId { get { return this.instanceId; } }

  public void Attach(InstanceContext owner) { Console.WriteLine("Attached to new InstanceContext."); }
  public void Detach(InstanceContext owner) { Console.WriteLine("Detached from InstanceContext."); }
}
```

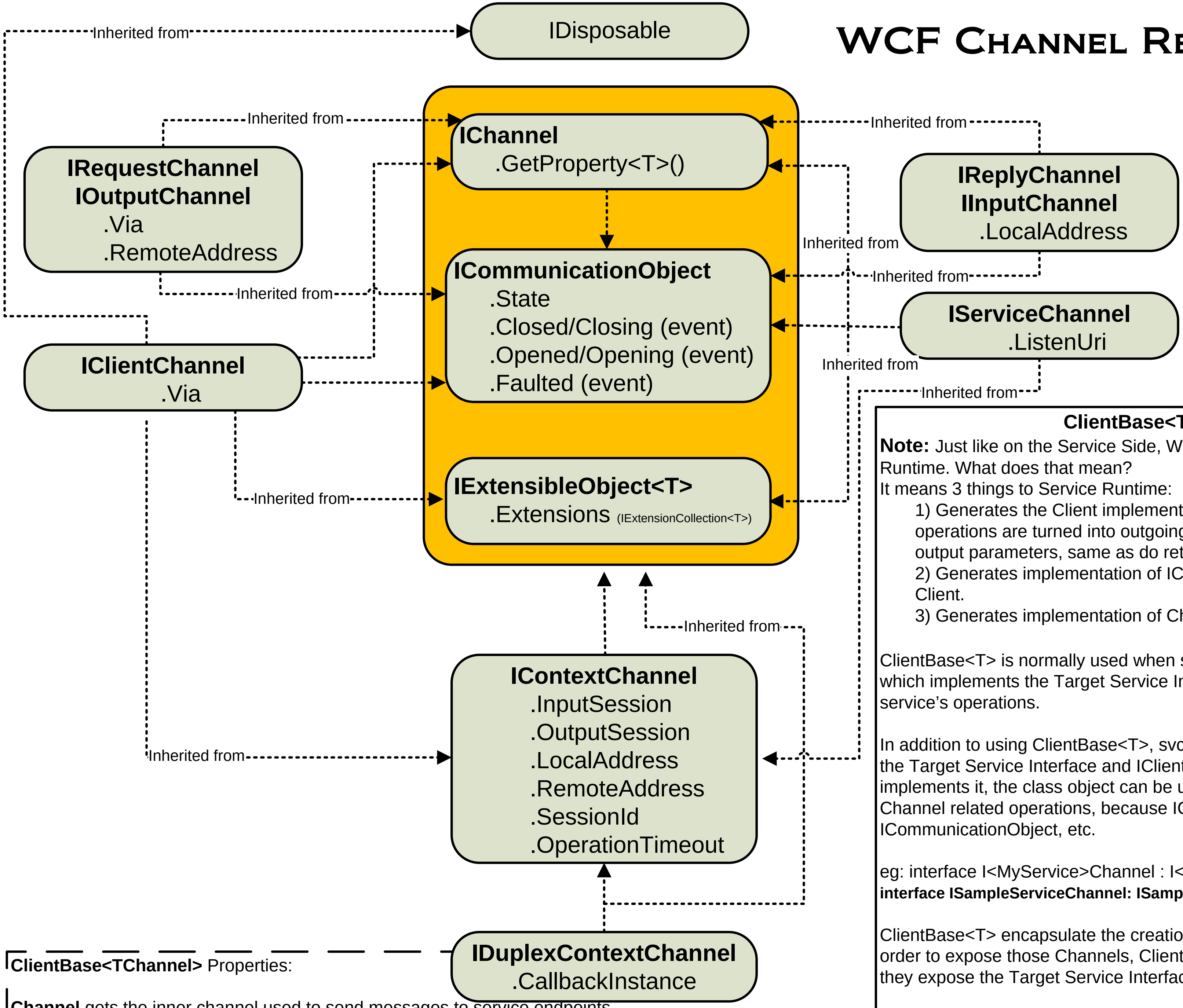
When an extension is added to the collection, Attach is called before it goes into the collection; when removed from the collection, Detach is called before it is removed

```
// How to Plug-In MyInstanceContextInitializer through the IEndpointBehavior

public class InstanceInitializerBehavior : IEndpointBehavior {

  public void AddBindingParameters(ServiceEndpoint serviceEndpoint,
                                   BindingParameterCollection bindingParameters) { }

  // Apply the custom IInstanceContextProvider to the EndpointDispatcher.DispatchRuntime
  public void ApplyDispatchBehavior(ServiceEndpoint serviceEndpoint,
                                    EndpointDispatcher endpointDispatcher) {
    MyInstanceContextInitializer myICintializer = new MyInstanceContextInitializer();
    endpointDispatcher.DispatchRuntime.InstanceContextInitializers.Add(myICinitializer);
  }
  public void ApplyClientBehavior(ServiceEndpoint serviceEndpoint, ClientRuntime behavior) {    }
  public void Validate(ServiceEndpoint endpoint) {    }
}
```

```
// How to Create a BehaviorExtensionElement to be used in {app,web}.conf file.

public class InstanceInitializerBehaviorExtensionElement : BehaviorExtensionElement {
  public override Type BehaviorType { get { return typeof(InstanceInitializerBehavior); } }
  protected override object CreateBehavior() { return new InstanceInitializerBehavior(); }
}
```

http://msdn.microsoft.com/en-us/library/ms733816.aspx (the IExtensibleObject<T> pattern)
http://msdn.microsoft.com/en-us/library/ms586697.aspx (the IExtensibleObject<T> interface)

WCF Channel Related Interfaces



ClientBase<TChannel> Properties:

Channel gets the inner channel used to send messages to service endpoints.
ChannelFactory gets the underlying ChannelFactory<TChannel> object.
ClientCredentials gets the client credentials used to call an operation.
Endpoint gets the target endpoint for the service to which the client can connect.
InnerChannel gets the underlying IClientChannel implementation.
State gets the current state of the ClientBase<TChannel> object.

ClientBase<T> vs ChannelFactory<T>

Note: Just like on the Service Side, WCF Client is generated by the WCF's Service Runtime. What does that mean?
It means 3 things to Service Runtime:

1) Generates the Client implementation of Service Contract, so that calls to Service operations are turned into outgoing messages, and response messages are turned into output parameters, same as do return values.
2) Generates implementation of IClientChannel, the Channel Control Interface for WCF Client.
3) Generates implementation of ChannelStatck based on specified Endpoint's Binding.

ClientBase<T> is normally used when svcutil.exe is used to generated the Client Proxy which implements the Target Service Interface, so that one can use it to directly invoke service's operations.

In addition to using ClientBase<T>, svcutil.exe also create a Helper interface to combines the Target Service Interface and IClientChannel interface; it is helpful because if a class implements it, the class object can be used to invoke Service operations and manipulate Channel related operations, because IClientChannel inherits IChannel, ICommunicationObject, etc.

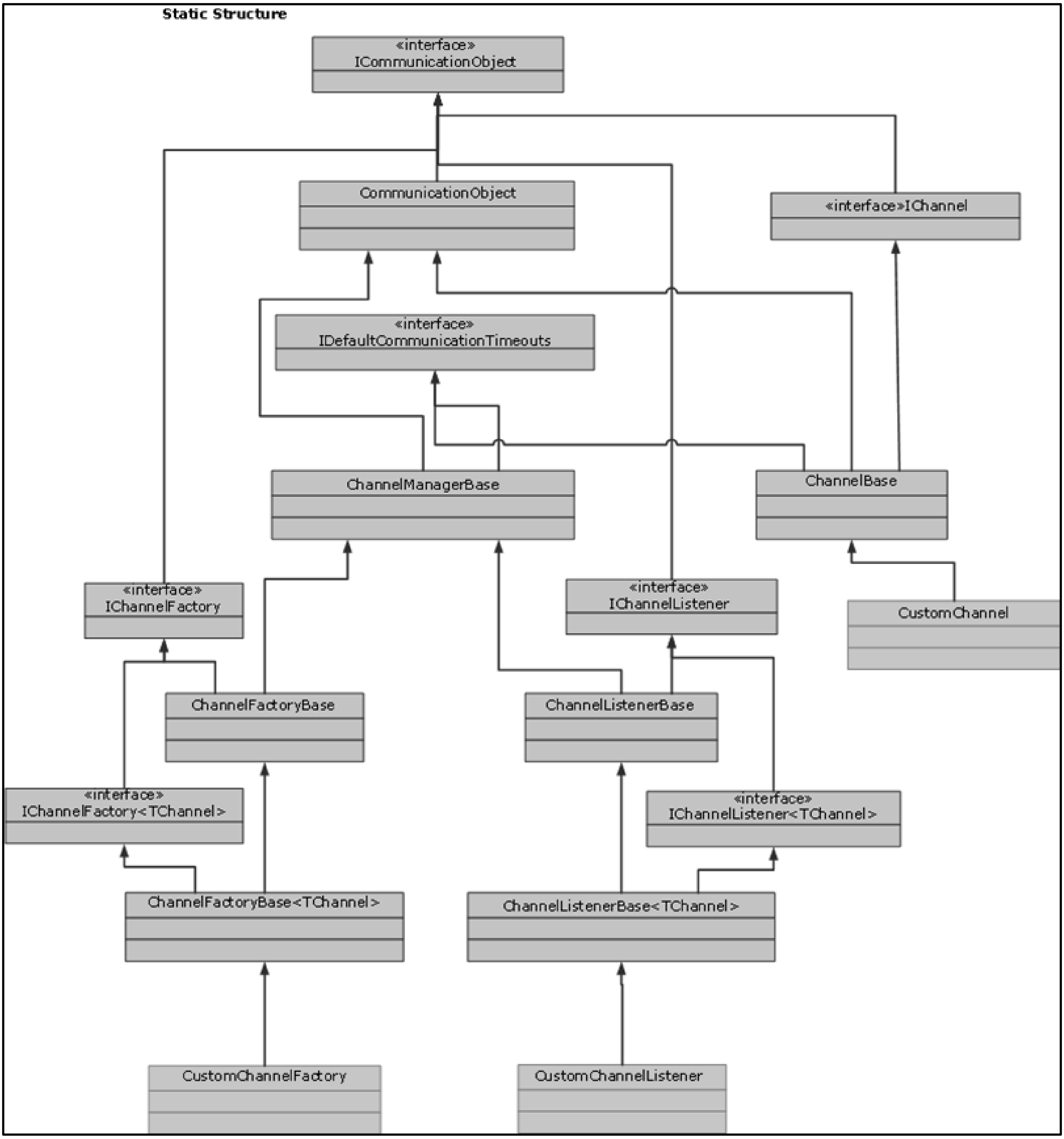
eg: interface I<MyService>Channel : I<MyService> : IClientChannel {}
interface ISampleServiceChannel: ISampleService : IClientChannel{}

ClientBase<T> encapsulate the creations of Channels---by using ChannelFactory<T>. In order to expose those Channels, ClientBase<T> has Channel and InnerChannel properties; they expose the Target Service Interface and Channel related operations respectively.

Channel gets the inner channel of TChannel type, used to send messages to service endpoints

InnerChannel gets the underlying IClientChannel implementation that is used to connect to Service (aka, target service)

WCF Channel Model



http://msdn.microsoft.com/en-us/library/ms729840.aspx

The Right Way to Dispose a Client Proxy and Channel

```
// Represents a code block containing WCF proxy client method calls.
public delegate void UseProxyDelegate<TInterface, TClass>(TClass proxy);

// Represents a code block containing WCF client method calls using a ChannelFactory
public delegate void UseChannelFactoryDelegate<TInterface>(TInterface channel);

// Provides methods for dealing with the "broken" IDisposable implementation for WCF clients.
// Credit: "http://geekswithblogs.net/DavidBarrett/archive/2007/11/22/117058.aspx"
// WCF Clients and the "Broken" IDisposable Implementation

// Credit: "http://www.iserviceoriented.com/blog/post/Indisposable+-+WCF+Gotcha+1.aspx"
// Indisposable: WCF Gotcha #1

public static class Service {

  /// "TInterface", The type of the interface.
  /// "TClass", The type of the class.
  /// "proxy", The WCF client proxy.
  /// "codeBlock", The code block containing WCF client method calls.

  public static void UseProxy<TInterface, TClass>(             // A WCF client via proxy
                         ClientBase<TInterface> proxy,
                         UseProxyDelegate<TInterface, TClass> codeBlock)
                           where TInterface : class
                           where TClass : ClientBase<TInterface>, TInterface {

    if (proxy == null) throw new ArgumentNullException("proxy", "proxy is null.");
    if (codeBlock == null) throw new ArgumentNullException("codeBlock", "codeBlock is null.");

    bool closed = false;
    try {
      codeBlock(proxy as TClass);
      proxy.Close();
      closed = true;
    } finally {
      if (!closed)
        proxy.Abort();
    }
  }

  // "TInterface", The type of the interface.
  // "channelFactory", The WCF client channel factory.
  // "codeBlock", The code block containing WCF client method calls.

  public static void UseChannelFactory<TInterface>(      // A WCF client via "ChannelFactory"
                         ChannelFactory<TInterface> channelFactory,
                         UseChannelFactoryDelegate<TInterface> codeBlock)
                           where TInterface : class {

    if (channelFactory == null) throw new ArgumentNullException("channelFactory", "channelFactory is null.");
    if (codeBlock == null) throw new ArgumentNullException("codeBlock", "codeBlock is null.");

    IClientChannel channel = channelFactory.CreateChannel() as IClientChannel;
    bool closed = false;
    try {
      codeBlock(channel as TInterface);
      channel.Close();
      closed = true;
    } finally {
      if (!closed)
        channel.Abort();
    }
  }
}
```

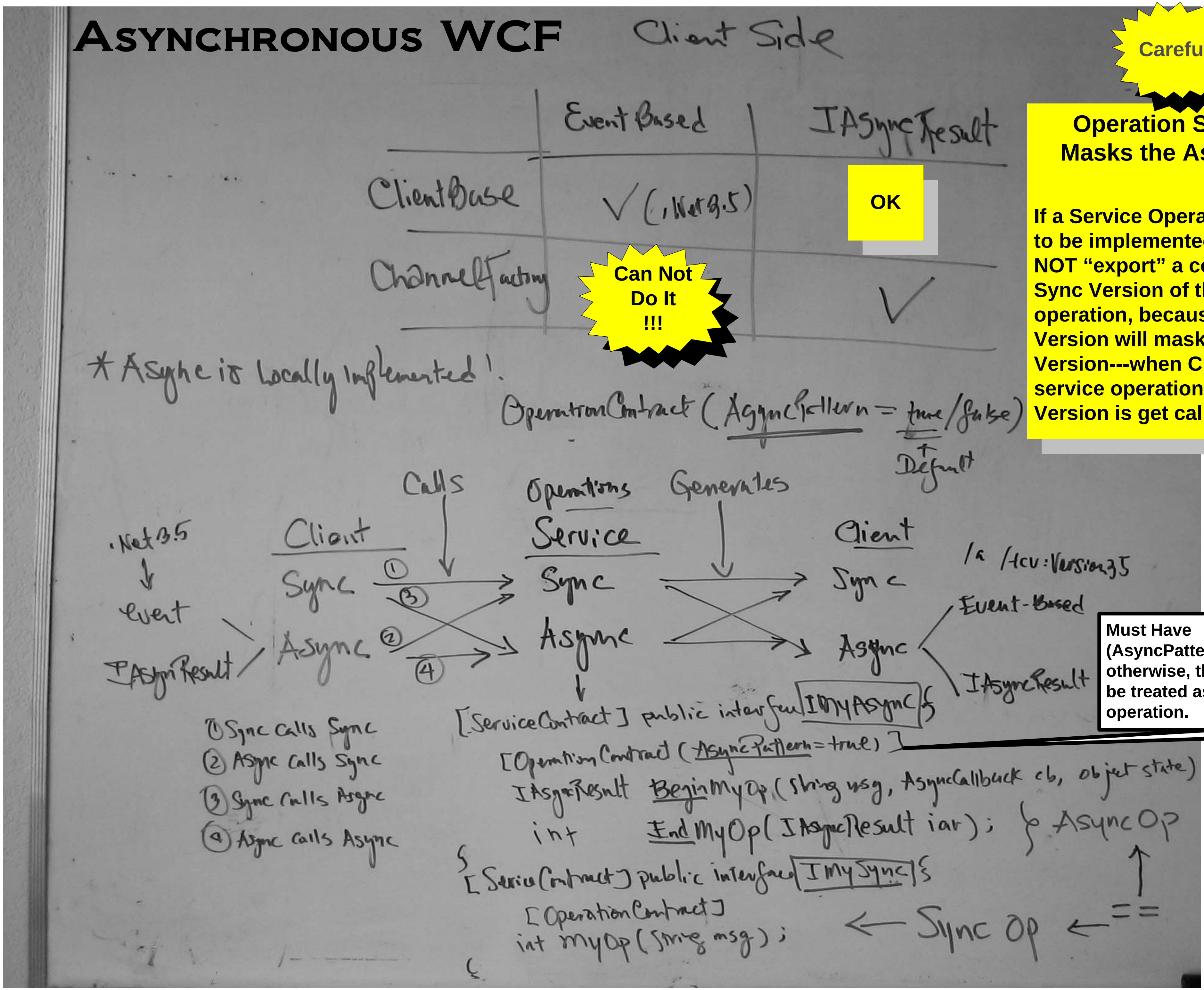
Use Cases

```
NetNamedPipeBinding binding = new NetNamedPipeBinding();
EndpointAddress endpoint = new EndpointAddress("net.pipe://localhost/Calculator");


ClientBase client = new CalculatorClient(binding, endpoint);
Service.UseProxy(client, delegate(CalculatorClient calculator) {
                      int result = calculator.Add(1, 2);
                      Console.WriteLine(result);
                  });

ChannelFactory<ICalculator> channelFactory = new ChannelFactory<ICalculator>(binding, endpoint);
Service.UseChannelFactory(channelFactory, delegate(ICalculator calculator) {
                                  int result = calculator.Add(1, 2);
                                  Console.WriteLine(result);
                              });
```

The Simplist Way to Dispose a Client Proxy or Channel

```
// Credit: http://elegantcode.com/2009/07/13/handy-wcf-techniques/

// The Extension Method!
public static void SafeClose(this ICommunicationObject proxy) {
  try {
    proxy.Close();
  } catch {
    proxy.Abort();
  }
}

// The fancy Generic Proxy that uses the Extension Method
public class SafeProxy<Service> : IDisposable where Service : ICommunicationObject {
  private readonly Service _proxy;

  public SafeProxy(Service s) {      // Ctor()
    _proxy = s;
  }

  public Service Proxy {
    get { return _proxy; }
  }

  public void Dispose() {
    if (_proxy != null)
      _proxy.SafeClose();  // calls the Extension Method
  }
}

// Application of the Generic, and fancy, Proxy---kind of Proxy of Proxy
using (var proxyVar = new SafeProxy(new Proxy())) {
        proxyVar.Proxy.MakeAnOperationCall();
}
```

Asynchronous WCF



Client Side

	Event Based	IAsyncResult
ClientBase	✓ (.Net 3.5)	OK
ChannelFactory	Can Not Do It !!!	✓

* Async is Locally Implemented!

OperationContract(AsyncPattern = true/false)

↑ Default

1. Sync calls Sync
2. Async calls Sync
3. Sync calls Async
4. Async calls Async

```
[ServiceContract] public interface IMyAsync {
  [OperationContract(AsyncPattern=true)]
  IAsyncResult BeginMyOp(string msg, AsyncCallback cb, object state)
  int          EndMyOp(IAsyncResult iar);   } AsyncOp
}                                                ↑
[ServiceContract] public interface IMySync {
  [OperationContract]
  int myOp(String msg);   ← Sync Op  ← ==
{
```

Careful!

Operation Sync Version Masks the Async Version!

If a Service Operation is intended to be implemented Async, then DO NOT "export" a corresponding Sync Version of the same operation, because the Sync Version will mask the Async Version---when Client calls the service operation, only the Sync Version is get called!

Must Have (AsyncPattern=true); otherwise, the operation will be treated as regular operation.

DispatchRuntime

http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.dispatchruntime.aspx

Use the **DispatchRuntime** class either to modify the default behavior of a service or individual endpoint, or to insert objects that implement custom modifications to one or both of the following service processes:

The transformation of incoming messages into objects and releasing those objects as method invocations on a service object.

The transformation of objects received from the response to a service operation invocation into outbound messages.

In WCF, the channel- and endpoint- dispatchers are the service components responsible for accepting new channels, receiving messages, method dispatch and invocation, and response processing. Each endpoint exposed by a **ServiceHost** object has one endpoint dispatcher and an associated channel dispatcher; in addition, each client that participates in duplex communication also has an endpoint dispatcher and channel dispatcher for each callback endpoint.

The **DispatchRuntime** enables you to intercept and extend the channel or endpoint dispatcher for all messages across a particular contract, even when a message is not recognized. When a message arrives that does not match any messages declared in the contract it is dispatched to the operation that was returned by the **UnhandledDispatchOperation** property. To intercept or extend across all messages for a particular operation, see the **DispatchOperation** class.

There are four (4) main areas of dispatcher extensibility exposed by the **DispatchRuntime** class:

Dispatch components use the properties of the **DispatchRuntime** and those of the associated channel dispatcher returned by the **ChannelDispatcher** property to customize how the channel dispatcher accepts and closes channels. This category includes the **ChannelInitializers** and **InputSessionShutdownHandlers** properties.

Message components are customized for each message processed. This category includes the **MessageInspectors**, **OperationSelector**, **Operations**, and the **ErrorHandlers** properties.

Instance components customize the creation, lifetime, and disposal of instances of the service type. For more information about service object lifetimes, see the **InstanceContextMode** property. This category includes the **InstanceContextInitializers** and the **InstanceProvider** properties.

Security-related components can use the following properties:

- **SecurityAuditLogLocation** indicates where audit events are written.
- **ImpersonateCallerForAllOperations** controls whether the service attempts to impersonate using the credentials provided by the incoming message.
- **MessageAuthenticationAuditLevel** controls whether successful message authentication events are written to the event log specified by SecurityAuditLogLocation.
- **PrincipalPermissionMode** controls how the **CurrentPrincipal** property is set.
- **ServiceAuthorizationAuditLevel** specifies how the auditing of authorization events is performed.
- **SuppressAuditFailure** specifies whether to suppress non-critical exceptions that occur during the logging process.

Typically custom extension objects are assigned to a DispatchRuntime property or inserted into a collection by a service behavior (an object that implements **IServiceBehavior**), a contract behavior (an object that implements **IContractBehavior**), or an endpoint behavior (an object that implements **IEndpointBehavior**). Then the installing behavior object is added to the appropriate collection of behaviors either programmatically or by implementing a custom **BehaviorExtensionElement** object to enable the behavior to be inserted using an application configuration file.

A **ChannelDispatcher** object associates an **IChannelListener** at a particular URI (called a listen URI) with an instance of a service. Each **ServiceHost** object can have many **ChannelDispatcher** objects, each associated with a different listener and listen URI for that service.

When a message arrives, the **ChannelDispatcher** queries each of the associated **EndpointDispatcher** objects whether the endpoint can accept the message, and passes the message to the one that can. The **EndpointDispatcher** object is responsible for processing messages from a ChannelDispatcher when the destination address of a message matches the AddressFilter property and the message action matches the ContractFilter property.

http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.channeldispatcher.aspx

All properties that control the lifetime and behavior of a channel session are available for inspection or modification on the **ChannelDispatcher** object. In addition to the **EndpointDispatcher**, these include custom **IChannelInitializer** objects, the **IChannelListener**, the **ServiceHost**, the associated and **InstanceContext**.

The **EndpointDispatcher** and the **DispatchRuntime** classes expose the runtime customization points for endpoints in a service. The **EndpointDispatcher** can be used to control which messages it can process and some endpoint-related information. The **DispatchRuntime** has a large number of properties used to insert custom extensions into the endpoint-wide runtime.

The **EndpointDispatcher** object is responsible for processing messages from a **ChannelDispatcher** when the destination address of a message matches the **AddressFilter** property and the message action matches the **ContractFilter** property. If two **EndpointDispatcher** objects can accept a message, the **FilterPriority** property value determines the higher priority endpoint.

Use the **EndpointDispatcher** object to configure or extend the process of receiving messages from the associated **ChannelDispatcher**, converting from message objects to objects used as parameters, and invoking an endpoint operation as well as the reverse process.

Typically, the **EndpointDispatcher** for an endpoint is obtained by implementing the **IEndpointBehavior** interface, but you can access the **EndpointDispatcher** from the other behavior interfaces.

You can use the following **EndpointDispatcher** properties:

Channel- and Endpoint Dispatchers

The **AddressFilter** property allows you to get or set a **MessageFilter** object that the **ChannelDispatcher** uses to identify whether the endpoint can process a particular message.

The **ChannelDispatcher** property gets the associated **ChannelDispatcher** object, which sends and receives messages to and from the **EndpointDispatcher** and which can be used to inspect or modify other channel-related values and behaviors.

The **ContractFilter** gets the **MessageFilter** object that is used to identify whether a message is destined for this contract.

The **ContractName** and **ContractNamespace** properties return the name and namespace of the endpoint contract.

The **DispatchRuntime** property returns the **DispatchRuntime** object that you can use to modify run-time values or insert custom run-time extensions for the entire endpoint.

The **EndpointAddress** property gets the address of the endpoint.

The **FilterPriority** property returns the priority of the composite filter that the **ChannelDispatcher** uses to establish which endpoint is to handle the message.

http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.endpointdispatcher.aspx

ICallContextInitializer (added to DispatchOperation.**CallContextInitializers**) defines the methods that enable the initialization and recycling of thread-local storage with the thread that invokes user code.

IChannelInitializer (added to ChannelDispatcher.**ChannelInitializers** or ClientRuntime.**ChannelInitializers**) defines the interface to notify a service or client when a channel is created. This can be used to track all outstanding sessions, for instance, so the service can send messages on them.

IClientMessageFormatter (Formatter) defines methods that are used to control the conversion of messages into objects and objects into messages for client applications.

IClientMessageInspector (added to **MessageInspectors)**defines a message inspector object that can be added to the MessageInspectors collection to view or modify messages.

IClientOperationSelector (OperationSelector) defines the contract for an operation selector.

IDispatchMessageFormatter (Formatter) defines methods that deserialize request messages and serialize response messages in a service application.

IDispatchMessageInspector (added to DispatchRuntime**.MessageInspectors)** defines the methods that enable custom inspection or modification of inbound and outbound application messages in service applications.

IDispatchOperationSelector (OperationSelector) defines the contract that associates incoming messages with a local operation to customize service execution behavior.

IErrorHandler (ErrorHandlers) allows an implementer to control the fault message returned to the caller and optionally perform custom error processing such as logging.

IInputSessionShutdown (added to DispatchRuntime**.InputSessionShutdownHandlers)** defines the contract that must be implemented to shut down an input session.

IInstanceContextInitializer (added to DispatchRuntime.**InstanceContextInitializers)** defines the methods necessary to inspect or modify the creation of InstanceContext objects when required.

IInstanceContextProvider (DispatchRuntime.**InstanceContextProvider**) implements to participate in the creation or choosing of a **InstanceContext** object, especially to enable shared sessions.

IInstanceProvider (InstanceProvider) declares methods that provide a service object or recycle a service object for a Windows Communication Foundation (WCF) service.

IInteractiveChannelInitializer (added to ClientRuntime.**InteractiveChannelInitializers)** defines the methods that enable a client application to display a user interface to collect identity information prior to creating the channel.

IMessageFilterTable<TFilterData> defines the contract that a filter table must implement to inspect messages with query criteria derived from one or more filters.

System.ServiceModel.Dispatcher Namespace

IOperationInvoker declares methods that take an object and an array of parameters extracted from a message, invoke a method on that object with those parameters, and return the method's return value and output parameters.

IParameterInspector (added to ClientOperation.**ParameterInspectors or** DispatchOperation.**ParameterInspectors)** defines the contract implemented by custom parameter inspectors that enables inspection or modification of information prior to and subsequent to calls on either the client or the service.

http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.aspx

Taking Action on Client Close

How do I clean up resources on the server when a duplex client closes its half of the connection?

Duplex services sometimes need to be a little bit more aggressive cleaning up after clients. Unlike with other channel shapes, a duplex client can decide to stop sending requests but continue to receive responses from the server. When that happens, the polite thing for the client to do is to close its half of the connection so that the server knows that no more requests are coming.

The server has to do some extra work to handle this half close case. Since a half close doesn't permit the server to dispose of the channel or the instance yet, cleanup may need to be split into multiple parts so that the request infrastructure can be torn down sooner. The server also needs some way to detect that a half close is taking place. You can't rely on the client sending a terminating message because the client might time out or decide not to send the message. The server should try to avoid letting client problems tie up resources unnecessarily.

The IInputSessionShutdown extensibility point allows a service to run code on client disconnection regardless of whether the disconnection was graceful or unexpected. You can insert your code by writing a behavior that adds to the InputSessionShutdownHandlers collection of the DispatchRuntime. There are two methods that will get call backs depending on whether a particular disconnection was graceful or unexpected.

```
public interface IInputSessionShutdown
{
   void ChannelFaulted(IDuplexContextChannel channel);
   void DoneReceiving(IDuplexContextChannel channel);
}
```

In the case of unexpected disconnections it may take some time for the server to realize that the client has disconnected. However, this is better than always having to wait for the full idle timeout to expire.

https://blogs.msdn.com/drnick/archive/2008/01/07/taking-action-on-client-close.aspx