Juval Lowy

WCF ESSENTIALS

Discover Mighty Instance Management Techniques for Developing WCF Apps

Instance management refers to a set of techniques used by Windows® Communication Foundation to bind a set of messages to a service instance. It's necessary because applications widely differ in their needs for scalability, performance, throughput, transactions, and queued calls, and there simply isn't a one-size-fits-all solution to address these varied demands. Understanding instance management is essential to developing scalable and consistent service-oriented applications. This article provides the rationale for various instance management modes, offers guidelines for when and how to use them, and discusses some related topics such as behaviors, contexts, demarcating operations, and instance deactivation.

By and large, the service instance mode is strictly a service-side implementation detail that should not manifest itself on the client side. To support that and a few other local service-side aspects, Windows Communication Foundation defines the notion of behaviors. A behavior is a local attribute of a service that does not affect its communication patterns.

The ServiceBehaviorAttribute configures service behaviors—a behavior that affects all endpoints of the service and is applied directly on the service implementation class. As shown in **Figure 1**, the attribute defines the InstanceContextMode property of the enum type InstanceContextMode whose value controls which instance mode is used for the service.

Per-Call Services

Per-call services are the Windows Communication Foundation default instantiation mode. When the service type is configured for per-call activation, a service instance, a common language runtime (CLR) object, exists only while a client call is in progress. Every client request gets a new dedicated service instance. **Figure 2** illustrates how this single-call activation works.

This article is based on a prerelease version of WinFX. All information herein is subject to change.

This article uses the following technologies:
Windows Communication Foundation

This article discusses:
- Per-call services
- Per-session services
- Shareable services
- Instance deactivation

Code download available at:
msdn.microsoft.com/msdnmag/code06.aspx

Juval Lowy is a software architect with IDesign providing Windows Communication Foundation training and architecture consulting. Juval is currently working on a comprehensive Windows Communication Foundation book. He is also the Microsoft Regional Director for the Silicon Valley. Contact Juval at www.idesign.net.

Figure 1 InstanceContextMode

```
public enum InstanceContextMode
{
    PerCall, PerSession, Shareable, Single
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, ...
{
    public InstanceContextMode InstanceContextMode {get;set;}
    ... // More members
}
```

1. The client calls the proxy and the proxy forwards the call to the service.
2. Windows Communication Foundation creates a service instance and calls the method on it.
3. After the method call returns, if the object implements IDisposable, then Windows Communication Foundation calls IDisposable.Dispose on it.

In the classic client-server programming model, using languages such as C++ or C#, every client gets its own dedicated server object. The fundamental problem with this approach is that it doesn't scale well. The server object may hold expensive or scarce resources such as database connections, communication ports, or files. Imagine an application that has to serve many clients. Typically, these clients create the objects they need when the client application starts and dispose of them when the client application shuts down. What impedes scalability with the client-server model is that the client applications can hold onto objects for long periods of time, while actually using the objects for only a fraction of that time. If you allocate an object for each client, you will tie up such crucial or limited resources for long periods of time and you will eventually run out of resources.

A better activation model is to allocate an object for a client only while a call is in progress from the client to the service. That way, you have to create and maintain only as many objects in memory as there are concurrent calls, not as many objects as there are outstanding clients. Between calls, the client holds a reference on a proxy that doesn't have an actual object at the end of the wire. The obvious benefit is that you can now dispose of the expensive resources the service instance occupies long before the client disposes of the proxy. By that same token, acquiring the resources is postponed until they are actually needed by a client. The second benefit is that forcing the service instance to reallocate or connect to its resources on every call caters very well to transactional resources and transactional programming because it eases the task of enforcing consistency with the instance state.

To configure a service type as a per-call service, you can apply the ServiceBehavior attribute with the InstanceContextMode property set to InstanceContextMode.PerCall:

```
[ServiceContract]
interface IMyContract {...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract {...}
```

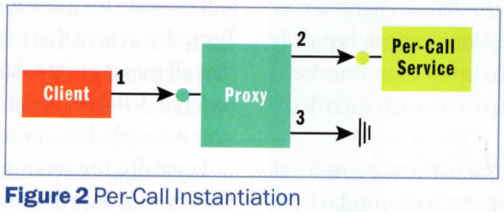

Figure 2 Per-Call Instantiation

Because InstanceContextMode.PerCall is the default value for the property, there is no need to actually apply the attribute. Consequently, the following two definitions are equivalent:

```
class MyService : IMyContract {...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract {...}
```

Although in theory you can use per-call instance activation mode on any service, in practice, you need to design the service and its contracts to support per-call activation mode from the ground up. The main problem is that the client doesn't know it's getting a new instance each time.

If a per-call service must be state-aware, meaning that state from one call must persist to a future call, the implementation itself must proactively manage the state, giving the client the illusion of a continuous session. An instance of a per-call service is created just before every method call and destroyed immediately after each call. Therefore, at the beginning of each call, the object should initialize its state from values saved in some storage, and at the end of the call it should return its state to the storage. Such storage is typically either a database or the file system, but it can also be volatile storage like static variables. However, not all of the object's state can be saved as-is. For example, if the state contains a database connection, the object must reacquire the connection at construction or at the beginning of every call and dispose of the connection at the end of the call or in its implementation of IDisposable.Dispose.

There are no hard-and-fast rules as to when and to what extent you should trade some performance for a lot of scalability.

Using per-call instance mode has one important implication for operation design: every operation must include a parameter to identify the service instance whose state needs to be retrieved. Examples for such parameters are the account number for bank account service, the order number for services processing orders, and so on. **Figure 3** shows a template for implementing a per-call service.

The MyMethod operation accepts a parameter of type Param (a pseudo-type invented for this example) that identifies the instance, as shown in the following:

```
public void MyMethod(Param objectIdentifier);
```

The instance then uses the identifier to retrieve its state and to save the state back at the end of the method call. Any piece of state that is common to all clients can be allocated at the constructor and disposed of in Dispose.

Also, the per-call activation mode works best when the amount of work to be done in each method call is small and there are no more activities to complete in the background once a method returns. For this reason, you should not spin off background threads or dispatch asynchronous calls back into the instance because the object will be discarded once the method returns.

Per-call services clearly offer a trade-off in performance (the overhead of reconstructing the instance state on each method call) with scalability (holding onto the state and the resources it ties in). There are no hard-and-fast rules as to when and to what extent you should trade some performance for a lot of scalability. You may need to profile your system and ultimately design some services to use per-call activation and redesign some not to use it.

Per-Session Services

Windows Communication Foundation can maintain a private session between a client and a particular service instance. When the client creates a new proxy to a service configured as session-aware, the client gets a new dedicated service instance that is independent of all other instances of the same service. That instance will remain in service usually until the client no longer needs it. Each private session uniquely binds a proxy to a particular service instance. Note that the client session has one service instance per proxy. If the client creates another proxy to the same or a different endpoint, that second proxy will be associated with a new instance and session.

Figure 3 Implementing a Per-Call Service

```
[DataContract]
class Param {...}

[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(Param objectIdentifier);
}

class MyPerCallService : IMyContract, IDisposable
{
    public void MyMethod(Param objectIdentifier)
    {
        GetState(objectIdentifier);
        DoWork();
        SaveState(objectIdentifier);
    }
    void GetState(Param objectIdentifier) {...}
    void DoWork() {...}
    void SaveState(Param objectIdentifier) {...}
    public void Dispose() {...}
}
```

Because the service instance remains in memory throughout the session, it can maintain state in memory, and the programming model is very much like that of the classic client-server model. Consequently, it also suffers from the same scalability and transaction issues as the classic client-server model. A service configured for private sessions cannot typically support more than a few dozen (or perhaps up to a few hundred) outstanding clients due to the cost associated with each such dedicated service instance.

Supporting a session has two facets: contractual and behavioral. The contractual facet is required across the service boundary because the client-side Windows Communication Foundation runtime needs to know it should use a session. The ServiceContract attribute offers the Boolean property Session:

```
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class,
    Inherited=false)]
public sealed class ServiceContractAttribute : Attribute
{
    public bool Session {get;set;}
    ... // More members
}
```

Session defaults to false. To support sessions, you need to set Session to true at the contract level:

```
[ServiceContract(Session = true)]
interface IMyContract {...}
```

To complete the configuration, you need to instruct Windows Communication Foundation to keep the service instance alive throughout the session and to direct the client messages to it. This local behavior facet is achieved by setting the InstanceContextMode property of the ServiceBehavior attribute to InstanceContextMode.PerSession, as shown in the following:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
class MyService : IMyContract {...}
```

The session typically terminates when the client closes the proxy, which notifies the service that the session has ended. If the service supports IDisposable, then the Dispose method will be called. **Figure 4** shows a contract and service configured to use a private session and their client. As you can see from the output, the client got a dedicated instance.

The per-session instance mode provides for a session at the application level between the client and a service instance. This session is only as reliable as the underlying channel session. Consequently, a service that implements a session-aware contract requires that all the endpoints that expose the contract use bindings that support reliable transport session. This constraint is verified at service load time, yielding InvalidOperationException if there is a mismatch. Make sure to use a binding that supports reliability and that you explicitly enable it at both the client and the service, either programmatically or administratively. One exception to this rule is the named pipes binding. This binding is considered a reliable transport and has no need for the reliable messaging protocol. Besides, all calls will be on the same machine anyway.

Just as reliable transport session is optional, so is ordered delivery of messages, and Windows Communication Foundation will provide for a session even when ordered delivery is disabled. Typically, a client that interacts with a session-aware service expects that all messages are delivered in the order they are sent. As such, ordered delivery is enabled by default when reliable transport session is enabled, so no additional setting is required.

Typically, the session will end once the client closes the proxy. However, in case the client fails to terminate gracefully or encounters a communication problem, each session also has an idle time timeout that defaults to 10 minutes—the session will automatically terminate after 10 minutes of inactivity from the client, even if the client still intends to use the session. Once the session has terminated due to the idle timeout, if the client tries to use its proxy, the client will get a CommunicationObjectFaultedException.

Both the client and the service can configure a different timeout by setting a different value in the binding. The bindings that support reliable transport-level session provide the ReliableSession property with the InactivityTimeout property used for configuring the idle timeout. For example, the following shows the code that is required to programmatically configure an idle timeout of 25 minutes for the TCP binding:

```
NetTcpBinding tcpSessionBinding = new NetTcpBinding();

tcpSessionBinding.ReliableSession.Enabled = true;

tcpSessionBinding.ReliableSession.InactivityTimeout =
    TimeSpan.FromMinutes(25);
```

Here is the equivalent configuration setting using a config file:

```
<netTcpBinding>
    <binding name="TCPSession">
        <reliableSession enabled="true" inactivityTimeout="00:25:00"/>
    </binding>
</netTcpBinding>
```

If both the client and the service configure timeouts, then the shorter timeout prevails.

Shareable Services

Windows Communication Foundation does not allow you to pass object references across service boundaries. Objects are technology-specific entities, and sharing objects goes against the grain of service-oriented, technology-neutral interactions. However, sometimes one client may want to share the current state of its session with another client. The solution is a shareable service. A shareable service behaves much like a per-session service, with one important additional aspect: the instance has a unique ID, and when a client establishes a session with a shareable service, the client can pass a logical reference to that instance to another client. The second client will establish an independent session but will share the same instance. Also, each of these sessions may use different inactivity timeouts, and expire independently of any other session.

You configure a shareable service by setting the InstanceContextMode property to InstanceContextMode.Sharable:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Sharable)]
class MyService : IMyContract {...}
```

There are two ways to take advantage of a shareable service and both are variations on the same technique. The first is to simply duplicate a proxy in the same app domain and the second is to pass an endpoint address over Windows Communication Foundation to another client.

Passing around an instance reference presents an interesting challenge to Windows Communication Foundation: what if the first client creates the instance, passes a reference to it to a second client, and closes its proxy before the second client has had the chance to create its own proxy using the reference? To deal with that, Windows Communication Foundation employs a linger timeout that defaults to one minute. Windows Communication Foundation keeps track of the sessions wired up to the same instance. Closing a proxy by any one of the clients does not dispose of the service instance. The instance is disposed of when the last session ends and the linger timeout has expired. Windows Communication Foundation also allows for configuring the linger timeout. In the service host config file, add a behaviorConfiguration property to the service element, pointing to a behavior section. In that section, set the instanceContextIdleTimeout property to the desired linger timeout:

```
<services>
    <service type="MyShareableService"
     behaviorConfiguration="ShortLingerBehaviour">
    ...
    </service>
</services>
<behaviors>
    <behavior name="ShortLingerBehaviour"
     instanceContextIdleTimeout="00:05:00" />
</behaviors>
```

If the linger timeout is not good enough for your application, then the participating clients need to coordinate among themselves when it is permissible to close the proxies.

Duplicating a Proxy

Duplicating a proxy is handy when two clients in the same app domain want to share a service instance. If the two clients share a proxy reference, then the clients need to coordinate which is responsible for closing the proxy and when. This need for coordination will introduce an unwanted degree of coupling between the two clients. The solution is to have the two clients use separate proxies and yet point to the same instance. Because the two proxies will have independent sessions, each client can close its proxy without affecting the other.

To duplicate a proxy, create a proxy to the service as usual and then explicitly resolve the service instance by creating an endpoint address reference. This contains the address of the service and some additional addressing headers information required to connect to the existing instance. To obtain an endpoint reference, use the ResolveInstance method of the IClientChannel interface:

```
public interface IClientChannel : ...
{
    EndpointAddress ResolveInstance();
    ... // More members
}
```

You obtain IClientChannel through the InnerChannel property of the ClientBase<T> proxy base class.

Figure 4 Per-Session Service and Client

Service Code

```
[ServiceContract(Session = true)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
class MyService : IMyContract,IDisposable
{
    int m_Counter = 0;

    MyService()
    {
        Trace.WriteLine("MyService.MyService()");
    }

    public void MyMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }

    public void Dispose()
    {
        Trace.WriteLine("MyService.Dispose()");
    }
}
```

Client Code

```
MyContractProxy proxy = new MyContractProxy();
proxy.MyMethod();
proxy.MyMethod();
proxy.Close();
```

Output

```
MyService.MyService()
Counter = 1
Counter = 2
MyService.Dispose()
```

Figure 5 Duplicating a Proxy

```
MyContractProxy proxy1 = new MyContractProxy();
proxy1.MyMethod();
IClientChannel channel = proxy1.InnerChannel;

//For simplicity, just grab binding from first proxy
Binding binding = proxy1.Endpoint.Binding;
EndpointAddress address = channel.ResolveInstance();

MyContractProxy proxy2 = new MyContractProxy(binding,address);
proxy2.MyMethod();

proxy1.Close();
proxy2.Close();
```

Calling ResolveInstance requires an exchange with the service to both verify it is configured for sharing and to obtain the instance ID. If the service is not configured for sharing, ResolveInstance throws a CommunicationException. Once you have the endpoint address, you create a new proxy with it. You can use the proxy constructor that takes the endpoint section name from the config file and the address, in which case the address in the config file will be ignored. Alternatively, you can use the proxy constructor that takes a binding and address objects. You can also set the endpoint address explicitly post-construction through the Endpoint property of the proxy. Regardless of which method you choose to create the proxy, make sure the endpoint address's transport matches that of the binding used. Now you have two distinct proxies that share the same service instance, albeit in separate sessions. **Figure 5** demonstrates this technique.

Using the same definitions as in **Figure 4** except with the service configured for InstanceContextMode.Sharable, **Figure 5** yields the same output as **Figure 4**.

Sharing an Instance

If the service is configured as shareable, Windows Communication Foundation lets you share an instance of that service across service boundaries, meaning you can pass a reference to a shared instance from one client to another over Windows Communication Foundation. The receiving side needs to expose a contract with an operation that takes EndpointAddress as a parameter. This sort of interaction is illustrated in **Figure 6**. In this figure, the client has a session with an instance of the shareable Service A:

1. The client obtains an endpoint address from Service A.
2. The client then passes the address over Windows Communication Foundation to an instance of Service B.
3. Service B constructs a proxy using the endpoint address and calls the instance of Service A.

The B instance can also pass the endpoint address to another internal client, or pass it to another service, and so on.

It is worth noting that while the original client and the A instance need to have a session, and while the B instance (or any client that uses the address) and the A instance need to have a session, the client and the B instance do not need to have a session at all.

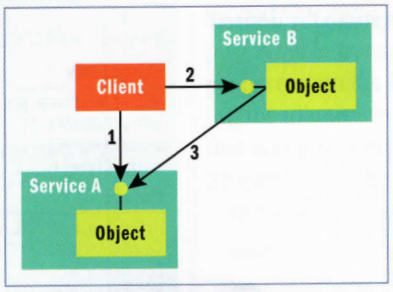

Figure 6 Passing a Reference

As I just mentioned, the receiving end must implement some kind of a contract that can accept an EndpointAddress as a parameter, as the following example shows:

```
[ServiceContract]
interface ISomeContract
{
    [OperationContract]
    void PassReference(EndpointAddress10 serviceAddress);
}
```

EndpointAddress is not serializable. However, you can safely work around this by using EndpointAddress10, which corresponds to the wire-format of WS-Addressing 1.0.

Assume for a moment that the service implementing ISomeContract is called MyOtherService and is defined as follows:

```
class MyOtherService : ISomeContract
{
   public void PassReference(EndpointAddress10 serviceAddress)
   {
      MyContractProxy proxy = new MyContractProxy();
      proxy.Endpoint.Address = serviceAddress.ToEndpointAddress();
      proxy.MyMethod();
      proxy.Close();
   }
}
```

Using the same definitions as in **Figure 4**, but with the service configured for InstanceContextMode.Sharable, the following code yields the same output:

```
MyContractProxy proxy1 = new MyContractProxy();
proxy1.MyMethod();

IClientChannel channel = proxy1.InnerChannel;
EndpointAddress serviceAddress = channel.ResolveInstance();

SomeContractProxy proxy2 = new SomeContractProxy();
proxy2.PassReference(EndpointAddress10.FromEndpointAddress
   (serviceAddress));

proxy1.Close();
proxy2.Close();
```

I find that a shareable service instance is a cumbersome instance management technique that probably harbors more harm than good. It does offer a singular benefit for some specific sharing scenarios, but it raises complicated issues with regard to session management and potential lifecycle coupling if the linger timeout is inadequate for your application. Try to avoid this technique and find a design solution that does not require you to pass a service reference or share a service instance's state.

Singleton Services

The singleton service is the ultimate shareable service. When a service is configured as a singleton, all clients get connected to the same single well-known instance independently of each other, regardless of which endpoint of the service they connect to. The singleton service lives forever, and is only disposed of once the host shuts down. The singleton is created exactly once when the host is created.

Using a singleton does not require clients to maintain a session with the singleton instance, or to use a binding that supports transport-level session. If the contract that the client consumes has a session, by closing the proxy the client will only terminate the session, not the

singleton instance. In addition, the session will never expire. If the singleton service supports contracts without a session, those contracts will not be per-call—they too will be connected to the same instance. With a singleton service, you can't call ResolveInstance. The reason is obvious: by its very nature the singleton is shared, and each client should simply create its own proxies to it.

You configure a singleton service by setting the InstanceContextMode property to InstanceContextMode.Single:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MySingleton : ...
{...}
```

Figure 7 demonstrates a singleton service with two contracts, one that requires a session and one that does not require one. As you can see from the client call, the calls on the two endpoints are routed to the same instance, and closing the proxies does not result in terminating the singleton.

Sometimes you may not want to create and initialize the singleton using just the default constructor. Perhaps initializing the scarce resource the singleton manages takes too much time and you would not want to penalize the first client. Perhaps initializing that state requires some custom steps or specific knowledge not available to the clients (or the clients should not be bothered). To support such scenarios, Window Communication Foundation allows you to create the singleton instance directly using normal CLR instantiation beforehand, initialize the instance, and then open the host with that instance in mind as the singleton service. The ServiceHost class offers a dedicated constructor that accepts an object:

```
public class ServiceHost : ServiceHostBase,...
{
    public ServiceHost(object singletonInstance,
        params Uri[] baseAddresses);
    public virtual object SingletonInstance { get; }
    ... // More members
}
```

Note that the object must be configured as a singleton. For example, consider the code in **Figure 8**. The class MySingleton will be first initialized and then hosted as a singleton. The client making the first call will be connected to the already initialized instance.

If you do initialize and host a singleton this way, you may also want to access it directly on the host side. Windows Communication Foundation enables downstream objects to reach back into the singleton directly using the SingletonInstance property of ServiceHost. Any party on the call chain leading down from an operation call on the singleton can always access the host through the operation context's read-only Host property:

```
public sealed class OperationContext : ...
{
    public ServiceHostBase Host {get;}
    ... // More members
}
```

Once you have the singleton reference you can interact with it directly, like shown here:

```
ServiceHost host = OperationContext.Current.Host as ServiceHost;
Debug.Assert(host != null);
MySingleton singleton = host.SingletonInstance as MySingleton;
Debug.Assert(singleton != null);
singleton.Counter = 388;
```

If no singleton instance is provided to the host, SingletonInstance returns null.

Having a singleton implies the singleton has some valuable state that you want to share across multiple clients. The problem is that when multiple clients connect to the singleton, they may all do so concurrently on multiple worker threads. The singleton must synchronize access to its state to avoid state corruption. This in turn means that only one client at a time can access the singleton. This may degrade responsiveness and availability to the point that the singleton is unusable as the system grows.

In general, use a singleton object if it maps well to a natural singleton in the application domain. A natural singleton is a resource that is by its very nature single and unique. Examples of natural singletons are a global logbook that all services should log their activities to, a single communication port, and a single mechanical motor. Avoid using a singleton if there is even the slightest chance that the business logic will allow more than one such service in the future, such as adding another motor or a second communi-

Figure 7 Singleton Service and Client

Service Code

```
[ServiceContract(Session=true)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}

[ServiceContract]
interface IMyOtherContract
{
    [OperationContract]
    void MyOtherMethod();
}

[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
class MySingleton : IMyContract, IMyOtherContract, IDisposable
{
    int m_Counter = 0;
    public MySingleton()
    {
        Trace.WriteLine("MyService.MyService()");
    }
    public void MyMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }
    public void MyOtherMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }
    public void Dispose()
    {
        Trace.WriteLine("MyService.Dispose()");
    }
}
```

Client Code

```
MyContractProxy proxy1 = new MyContractProxy();
proxy1.MyMethod();
proxy1.Close();

MyOtherContractProxy proxy2 = new MyOtherContractProxy();
proxy2.MyOtherMethod();
proxy2.Close();
```

Output

```
MyService.MyService()
Counter = 1
Counter = 2
```

Figure 8 Initializing and Hosting a Singleton

Service Code

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
class MySingleton : IMyContract
{
    int m_Counter = 0;

    //Accesses m_Counter
    public int Counter {get;set;}

    public void MyMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + Counter);
    }
}
```

Host Code

```
MySingleton singleton = new MySingleton();
singleton.Counter = 42;
ServiceHost host = new ServiceHost(singleton);
host.Open();
```

Client Code

```
MyContractProxy proxy = new MyContractProxy();
proxy.MyMethod();
proxy.Close();
```

Output

```
Counter = 43
```

cation port. The reason is clear: if your clients all depend on being implicitly connected to the well-known instance and more than one service instance is available, the clients would suddenly need to have a way to bind to the correct instance. This can have severe implications on the application's programming model.

Demarcating Operations

Sometimes when dealing with session contracts there is an implied order to operation invocations. Some operations cannot be called first while other operations must be called last. For example, consider this contract used to manage customer orders:

```
[ServiceContract(Session = true)]
interface IOrderManager
{
    [OperationContract]
    void SetCustomerId(int customerId);
    [OperationContract]
    void AddItem(int itemId);
    [OperationContract]
    decimal GetTotal();
    [OperationContract]
    bool ProcessOrders();
}
```

The contract has the following constraints: the client must first provide the customer ID against which items are added; then the total is calculated. When the order processing is complete, the session is terminated.

Windows Communication Foundation allows contract designers to designate contract operations as operations that can or cannot start or terminate the session using the IsInitiating and IsTerminating properties of the OperationContract attribute:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public bool IsInitiating {get;set;}
    public bool IsTerminating {get;set;}
    ... // More members
}
```

Using these properties may demarcate the boundary of the session, so I call this technique demarcating operations. Both a session-aware service and a singleton can implement a contract that uses demarcating operations to manage their client sessions.

The default values of these properties are IsInitiating set to true and IsTerminating set to false. Consequently these two definitions are equivalent, as shown here:

```
[ServiceContract(Session = true)]
interface IMyContract
{
    [OperationContract]
    void MyMethod()
}

[ServiceContract(Session = true)]
interface IMyContract
{
    [OperationContract(IsInitiating = true, IsTerminating = false)]
    void MyMethod()
}
```

By default, operations do not demarcate the session boundary—they can be called first, last, or between any other operation in the session. Using non-default values enables you to dictate that a method is not called first, or that it is called last, or both, to enforce the interaction constraints (see **Figure 9**).

When IsInitiating is set to true (the default), it means the operation will start a new session if it is the first method called by the client, but that it will be part of the ongoing session if another operation is called first. When IsInitiating is set to false, it means the operation can never be called as the first operation by the client in a new session, and the method can only be part of an ongoing session.

When IsTerminating is set to false (the default), the session continues after the operation returns. When IsTerminating is set to true,

Figure 9 Specifying Demarcating Operations

Service Code

```
[ServiceContract(Session = true)]
interface IOrderManager
{
    [OperationContract]
    void SetCustomerId(int customerId);

    [OperationContract(IsInitiating = false)]
    void AddItem(int itemId);

    [OperationContract(IsInitiating = false)]
    decimal GetTotal();

    [OperationContract(IsInitiating = false, IsTerminating = true)]
    bool ProcessOrders();
}
```

Client Code

```
OrderManagerProxy proxy = new OrderManagerProxy();
proxy.SetCustomerId(123);
proxy.AddItem(4);
proxy.AddItem(5);
proxy.AddItem(6);
proxy.ProcessOrders();
proxy.Close();
```

the session terminates once the method returns, and the client will not be able to issue additional calls on the proxy. Note that the client must still close the proxy because the operation does not dispose of the service instance—it simply rejects subsequent calls.

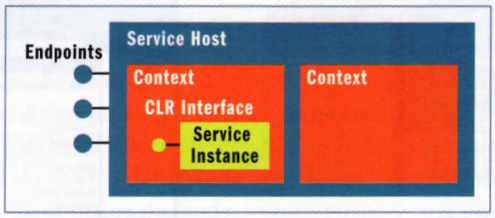

Figure 10 Contexts and Instances

Instance Deactivation

The session-aware service instance management technique as described so far connects a client (or clients) to a service instance. The real picture is more complex. Each service instance is hosted in a context, as shown in **Figure 10**.

Sessions actually correlate the client messages not to the instance, but to the context that hosts it. When the session starts, the host creates a new context. When the session ends, the context is terminated. By default, the lifeline of the context is the same as that of the instance it hosts. However, for optimization purposes, Windows Communication Foundation provides the service designer with the option of separating the two lifelines and deactivating the instance separately from its context. In fact, Windows Communication Foundation also allows the situation of a context that has no instance at all. I call this instance management technique context deactivation. The usual way for controlling context deactivation is through the ReleaseInstanceMode property of the OperationBehavior attribute:

```
public enum ReleaseInstanceMode
{
    None, BeforeCall, AfterCall, BeforeAndAfterCall
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute,...
{
    public ReleaseInstanceMode ReleaseInstanceMode {get;set;}
    ... // More members
}
```

The various values of ReleaseInstanceMode specify when to release the instance in relation to the method call: before, after, before and after, or not at all. When releasing the instance, if the service supports IDisposable, the Dispose method is called.

You typically apply instance deactivation only on some, but not all, of the service methods, or with different values on different methods. If you were to apply it uniformly, you would end up with a per-call-like service, so you might as well have configured it that way. If relying on instance deactivation assumes a certain call order, you can try and enforce that order using demarcating operations.

The default value for the ReleaseInstanceMode property is ReleaseInstanceMode.None. ReleaseInstanceMode.None means that the instance lifeline is not affected by the call, as shown in **Figure 11**.

When a method is configured with ReleaseInstanceMode.BeforeCall, if there is already an instance in the session, Windows Communication Foundation will deactivate it, create a new instance in its place, and let that new instance service the call (as shown in **Figure 11**). While the client blocks, the incoming thread handles the process of deactivating the instance and calling dispose before the call occurs. This ensures that the deactivation is indeed done before the call and not concurrently. ReleaseInstanceMode.BeforeCall is designed to optimize methods like Open, which acquire some valuable resources and yet want to release the previously allocated resources. Instead of acquiring the resource when the session starts, you wait on the Open method, and then both release the previously allocated resources and allocate new ones. After Open is called you are ready to start calling other methods on the instance. These methods are typically configured with ReleaseInstanceMode.None.

When a method is configured with ReleaseInstanceMode.AfterCall, Windows Communication Foundation deactivates the instance after the call (see **Figure 11**). This is designed to optimize methods (such as Close) that clean up valuable resources the instance holds, without waiting for the session to terminate. ReleaseInstanceMode.AfterCall is typically applied on methods called after methods that have been configured with ReleaseInstanceMode.None.

When a method is configured with ReleaseInstanceMode.BeforeAndAfterCall it has the combined effect , as its name implies, of ReleaseInstanceMode.BeforeCall and ReleaseInstanceMode.AfterCall. If the context has an instance before the call is made, then

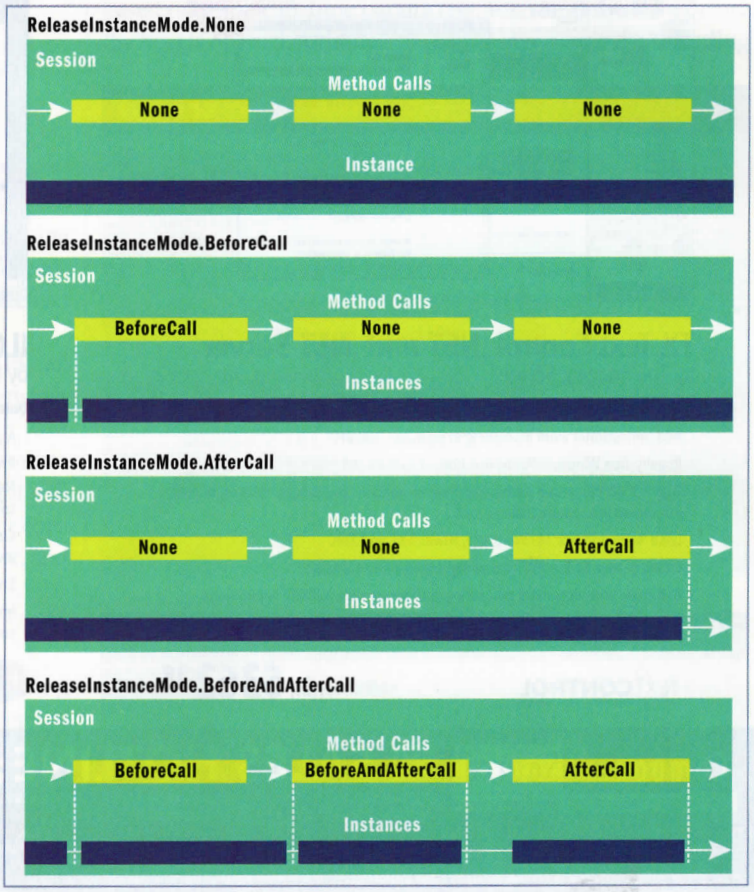

Figure 11 Comparing Instance Deactivation Modes

Windows Communication Foundation deactivates that instance just before the call, creates a new instance to service the call, and deactivates the new instance after the call, as shown in **Figure 11**.

ReleaseInstanceMode.BeforeAndAfterCall may look superfluous at first glance, but it actually complements the other values. It is designed to be applied to methods called after methods marked with ReleaseInstanceMode.BeforeCall/None or before methods marked with ReleaseInstanceMode.AfterCall/None. Consider a situation where the session-aware service wants to benefit from state-aware behavior (like a per-call service), while holding onto resources just when needed to optimize resource allocation and security lookup. If ReleaseInstanceMode.BeforeCall was the only available option, there would be a period of time after the calls when the resources would still be allocated to the object but not in use. A similar situation would occur if ReleaseInstanceMode.AfterCall was the only available option, because there would be a period of time before the call when the resource would be wasted.

Instead of making a design-time decision on which methods to use to deactivate the instance, you can make a run-time decision to deactivate the instance after the method returns. You do that by calling the ReleaseServiceInstance method on the instance context. You obtain the instance context through the InstanceContext property of the operation context:

```
public sealed class InstanceContext : CommunicationObject,...
{
    public void ReleaseServiceInstance();
    ... // More members
}

public sealed class OperationContext : ...
{
    public InstanceContext InstanceContext {get;}
   ... // More members
}
```

Figure 12 demonstrates this technique.

Calling ReleaseServiceInstance has a similar effect as using ReleaseInstanceMode.AfterCall. When used in a method decorated with ReleaseInstanceMode.BeforeCall, it has a similar effect as using ReleaseInstanceMode.BeforeAndAfterCall.

Instance deactivation is an optimization technique, and like all such techniques it should be avoided in general cases. Consider using instance deactivation only after failing to meet both your performance and scalability goals and when careful examination and profiling has proven beyond a doubt that using it will improve the situation. If scalability and throughput are your concern, choose the simplicity of the per-call instancing mode, and avoid instance deactivation.

Throttling

While not a direct instance management technique, throttling enables you to restrain client connections and the load they place on your service. Throttling allows you to avoid maxing-out your service and the underlying resources it allocates and uses.

The default throttling setting is unlimited. When engaged, if the throttling settings you configured are exceeded, Windows Communication Foundation automatically places the pending callers in a queue and serves them out of the queue in order. Throttling is done per service type—that is, it affects all instances of the service and all its endpoints. This is done by associating the throttle with every channel dispatcher the service uses.

Windows Communication Foundation allows you to control the service consumption parameters shown in **Figure 13**. With a per-session service, Max Instances is both the total number of concurrently active instances and the number of concurrent sessions. With a shared-session service, Max Instances is just the total number of concurrently active instances (since each shared instance can have multiple sessions). With a per-call service, the number of instances is actually the same as the number of concurrent calls. Consequently, the maximum number of instances with a per-call service is the minimum of Max Instances and Max Concurrent Calls. Max Instances is ignored with a singleton service since it can only have a single instance anyway.

Configuring Throttling

Throttling is typically configured in the config file. This enables you to throttle the same service code differently over time or across deployment sites. The host can also programmatically configure throttling based on some run-time decisions.

Figure 14 shows how to configure throttling in the host config file. Using the behaviorConfiguration tag you add to your service a custom behavior that sets throttled values.

The host process can programmatically throttle the service based on some run-time parameters. You can only do so before the host is opened. Although the host can override the throttling

Figure 12 Using ReleaseServiceInstance

```
[ServiceContract(Session = true)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}

[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
class MyService : IMyContract,IDisposable
{
    public void MyMethod()
    {
        //Do some work then
        OperationContext.Current.InstanceContext.
            ReleaseServiceInstance();
    }

    public void Dispose() {...}
}
```

Figure 13 ServiceThrottle Parameters

Parameter	Description
Max Connections	The overall number of outstanding clients connected to the service.
Max Concurrent Calls	The total number of calls currently in progress across all service instances.
Max Instances	The total number of contexts concurrently alive. How instances map to contexts is a product of the instance context management mode.

Figure 14 Administrative Throttling

```
<system.serviceModel>
    <services>
        <service type = "MyService"
                 behaviorConfiguration = "ThrottledBehavior">
        ...
        </service>
    </services>
    <behaviors>
        <behavior name="ThrottledBehavior">
            <throttling
               maxConcurrentCalls = "12"
               maxConnections = "34"
               maxInstances = "56"
            />
        </behavior>
    </behaviors>
</system.serviceModel>
```

Figure 15 Programmatic Throttling

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

ServiceThrottlingBehavior throttle;
throttle = host.Description.Behaviors.Find<ServiceThrottlingBehavior>();

if(throttle == null)
{
    throttle = new ServiceThrottlingBehavior();
    throttle.MaxConcurrentCalls = 12;
    throttle.MaxConnections  = 34;
    throttle.MaxInstances    = 56;
    host.Description.Behaviors.Add(throttle);
}

host.Open();
```

Figure 16 Reading the Throttled Values

```
class MyService : ...
{
   public void MyMethod() //Contract operation
   {
      ChannelDispatcher dispatcher = OperationContext.Current.Host.
          ChannelDispatchers[0] as ChannelDispatcher;

      ServiceThrottle serviceThrottle = dispatcher.ServiceThrottle;

      Trace.WriteLine("MaxConcurrentCalls = " +
          serviceThrottle.MaxConcurrentCalls);
      Trace.WriteLine("MaxConnections = " +
          serviceThrottle.MaxConnections);
      Trace.WriteLine("MaxInstances = " + serviceThrottle.MaxInstances);
   }
}
```

behavior found in the config file by removing the configuration and adding its own, you typically should provide a programmatic throttling behavior only when there is no throttling behavior in the config file.

The ServiceHostBase class offers the Description property of the type ServiceDescription:

```
public class ServiceHostBase : ...
{
    public virtual ServiceDescription Description {get;}
    ... // More members
}
```

As its name implies, the service description is the description of the service with all its aspects and behaviors. ServiceDescription contains a property called Behaviors of the type KeyedByTypeCollection<I> with IServiceBehavior as the generic parameter:

```
public class KeyedByTypeCollection<I> : KeyedCollection<Type,I>
{
    public T Find<T>();
    public T Remove<T>();
    ... // More members
}
public class ServiceDescription
{
    public KeyedByTypeCollection<IServiceBehavior> Behaviors {get;}
}
```

IServiceBehavior is the interface that all behavior classes and attributes implement. KeyedByTypeCollection<I> offers the generic method Find<T>, which returns the requested behavior if it is in the collection and null otherwise. A given behavior type can only be found once in the collection at most.

Figure 15 shows how to set the throttled behavior programmatically. First the hosting code verifies that no service throttling behavior is provided in the config file. This is done by calling the Find<T> method of KeyedByTypeCollection<I> using ServiceThrottlingBehavior as the type parameter. ServiceThrottlingBehavior is defined in the System.ServiceModel.Design namespace:

```
public class ServiceThrottlingBehavior : IServiceBehavior
{
    public int MaxConcurrentCalls {get;set;}
    public int MaxConnections {get;set;}
    public int MaxInstances {get;set;}
    ... // More members
}
```

If the returned throttle is null, the hosting code creates a new ServiceThrottlingBehavior, sets its values, and adds it to the behaviors in the service description.

Reading Throttled Values

The throttled values can be read at run time by service developers for diagnostics and analysis purposes. The service instance can access at run time its throttled properties from its dispatcher. First, obtain a reference to the host from the operation context. The host base class ServiceHostBase offers the read-only ChannelDispatchers property—a strongly typed collection of ChannelDispatcherBase objects. Each item in the collection is of the type ChannelDispatcher. ChannelDispatcher offers the property ServiceThrottle which contains the configured throttled values (see **Figure 16**).

Note that the service can only read the throttled values, and has no way of affecting them. If the service tries to set throttled values it will get an InvalidOperationException.

Conclusion

While every application is unique when it comes to scalability, performance, and throughput, Windows Communication Foundation does offer canonical instance management techniques that are applicable across the range of applications, thus enabling a wide variety of scenarios and programming models. Understanding Windows Communication Foundation instance management and choosing the right activation mode is critical. Yet applying each mode is surprisingly easy to accomplish. In addition, you can augment the straightforward modes with demarcating operations and instance deactivation. ■