Invoking STA COM Objects from WCF

# Problem Statement

WCF threads (or any .Net managed threads) in reality are COM multithreaded apartment (MTA) threads. Any message received in WCF is processed in an MTA thread. Usually this is not an issue while running things in a managed world or within .Net framework. This becomes a huge issue if you’re dealing with WCF services invoking COM objects or sending messages to COM objects. COM objects run in single threaded apartment (STA) threads and as such provide serious performance bottlenecks due to MTA -> STA switching. For more details, see this: [Wicked Code: Running ASMX Web Services on STA Threads](http://msdn.microsoft.com/en-us/magazine/cc163544.aspx#S1). As such, there is a need to invoke WCF -> COM operations/messages on a STA thread.

# The Solution

The solution involves around creating a thread and setting its apartment state to be STA and then running the operation inside this thread. Something along these lines:

public object DoSomeSTAThing()

{

object returnValue = new object();

System.ServiceModel.OperationContext context = System.ServiceModel.OperationContext.Current;

var thread = new System.Threading.Thread(new System.Threading.ThreadStart(delegate

{

using (var scope = new System.ServiceModel.OperationContextScope(context))

{

// invoke COM objects here

// process messages

}

}));

// !! Important: Need to set aprtment state before the thread starts

thread.SetApartmentState(System.Threading.ApartmentState.STA);

thread.Start();

thread.Join();

return returnValue;

}

While this approach works fine, it puts the burden of implementing the thread mechanics on the caller. Each developer has to repeat the plumbing code all over the place wherever he’s dealing with COM objects. In short, it creates messy code.

Fortunately, WCF provides a mechanism to isolate the plumbing code from the actual method body resulting in cleaner code. Further, the app developer no longer has to repeat the threading mechanics. The ideal solution is to create an OperationBehavior to run the operation in STA.

An operation behavior is nothing more than a class that implements the **System.ServiceModel.Description.IOperationBehavior** interface. *Typically* it inherits from the **System.Attribute** class, but this isn't a strict requirement. Doing this does, however, allow the operation behavior to be applied declaratively to your operation implementation. Again, this is optional.

The only method we care about in our new operation behavior is the **ApplyDispatchBehavior** method. Using this method, we can install our operation invoker. Here's our behavior:

using System;

using System.ServiceModel.Channels;

using System.ServiceModel.Description;

using System.ServiceModel.Dispatcher;

namespace WCFWithSTADemo

{

[AttributeUsage(AttributeTargets.Method)]

public class STAOperationBehavior : Attribute, System.ServiceModel.Description.IOperationBehavior

{

#region Implementation of IOperationBehavior

/// <summary>

/// Implement to confirm that the operation meets some intended criteria.

/// </summary>

/// <param name="operationDescription">The operation being examined. Use for examination only. If the operation description is modified, the results are undefined.</param>

public void Validate(OperationDescription operationDescription)

{

// Nothing needs to be done here

// to force synchronous only execution, uncomment the following code:

// if (operationDescription.SyncMethod == null)

// throw new InvalidOperationException("The STAOperationBehaviorAttribute only works for synchronous method invocations.");

}

/// <summary>

/// Implements a modification or extension of the service across an operation.

/// </summary>

/// <param name="operationDescription">The operation being examined. Use for examination only. If the operation description is modified, the results are undefined.</param><param name="dispatchOperation">The run-time object that exposes customization properties for the operation described by <paramref name="operationDescription"/>.</param>

public void ApplyDispatchBehavior(OperationDescription operationDescription, DispatchOperation dispatchOperation)

{

dispatchOperation.Invoker = new STAOperationInvoker(dispatchOperation.Invoker);

}

/// <summary>

/// Implements a modification or extension of the client across an operation.

/// </summary>

/// <param name="operationDescription">The operation being examined. Use for examination only. If the operation description is modified, the results are undefined.</param><param name="clientOperation">The run-time object that exposes customization properties for the operation described by <paramref name="operationDescription"/>.</param>

public void ApplyClientBehavior(OperationDescription operationDescription, ClientOperation clientOperation)

{

// Only called when the contract is used on a client.

// Will do nothing here since we only care about the service implementation.

}

/// <summary>

/// Implement to pass data at runtime to bindings to support custom behavior.

/// </summary>

/// <param name="operationDescription">The operation being examined. Use for examination only. If the operation description is modified, the results are undefined.</param><param name="bindingParameters">The collection of objects that binding elements require to support the behavior.</param>

public void AddBindingParameters(OperationDescription operationDescription, BindingParameterCollection bindingParameters)

{

// Nothing needs to be done here

}

#endregion

}

}

Technically, we don't care about the operation behavior so much as we do the operation invoker we also need to create. We want to control the invocation of the operation.  For this we create an operation invoker; the operation behavior is only there to install the invoker.

An operation invoker is class which implements the **System.ServiceModel.Dispatcher.IOperationInvoker** interface. Among other things, the invoker allows us to hijack the WCF process and put our own logic around our operation implementation. This is a VERY useful thing to do. For example, whenever I want to implement my own security on services I'll create an invoker to do authorization.  If authorization is successful, I'll call the operation myself. Otherwise, I'll throw a security exception and WCF will deal with it from there.

For our purposes, we are going to use this hijacking ability to starting a new STA thread and call the operation from that thread. We are going to do this in the **Invoke** method.

using System;

using System.ServiceModel.Dispatcher;

using System.Threading;

using System.ServiceModel;

namespace WCFWithSTADemo

{

class STAOperationInvoker : IOperationInvoker

{

private IOperationInvoker \_iOperationInvoker;

public STAOperationInvoker(IOperationInvoker iOperationInvoker)

{

this.\_iOperationInvoker = iOperationInvoker;

}

#region Implementation of IOperationInvoker

/// <summary>

/// Returns an <see cref="T:System.Array"/> of parameter objects.

/// </summary>

/// <returns>

/// The parameters that are to be used as arguments to the operation.

/// </returns>

public object[] AllocateInputs()

{

return \_iOperationInvoker.AllocateInputs();

}

/// <summary>

/// Returns an object and a set of output objects from an instance and set of input objects.

/// </summary>

/// <returns>

/// The return value.

/// </returns>

/// <param name="instance">The object to be invoked.</param>

/// <param name="inputs">The inputs to the method.</param>

/// <param name="outputs">The outputs from the method.</param>

public object Invoke(object instance, object[] inputs, out object[] outputs)

{

Object result = null;

Object[] staOutputs = null;

OperationContext context = OperationContext.Current;

var thread = new Thread(new ThreadStart(delegate

{

using (var scope = new OperationContextScope(context))

{

result = \_iOperationInvoker.Invoke(instance, inputs, out staOutputs);

}

}));

thread.SetApartmentState(ApartmentState.STA);

thread.Start();

thread.Join();

//+

outputs = staOutputs;

//+

return result;

}

/// <summary>

/// An asynchronous implementation of

/// the <see cref="M:System.ServiceModel.Dispatcher.IOperationInvoker.Invoke(System.Object,System.Object[],System.Object[]@)" /> method.

/// </summary>

/// <returns>

/// A <see cref="T:System.IAsyncResult"/> used to complete the asynchronous call.

/// </returns>

/// <param name="instance">The object to be invoked.</param>

/// <param name="inputs">The inputs to the method.</param>

/// <param name="callback">The asynchronous callback object.</param>

/// <param name="state">Associated state data.</param>

public IAsyncResult InvokeBegin(object instance, object[] inputs, AsyncCallback callback, object state)

{

return \_iOperationInvoker.InvokeBegin(instance, inputs, callback, state);

}

/// <summary>

/// The asynchronous end method.

/// </summary>

/// <returns>

/// The return value.

/// </returns>

/// <param name="instance">The object invoked.</param>

/// <param name="outputs">The outputs from the method.</param>

/// <param name="result">The <see cref="T:System.IAsyncResult"/> object.</param>

public object InvokeEnd(object instance, out object[] outputs, IAsyncResult result)

{

return \_iOperationInvoker.InvokeEnd(instance, out outputs, result);

}

/// <summary>

/// Gets a value that specifies whether

/// the <see cref="M:System.ServiceModel.Dispatcher.IOperationInvoker.Invoke(System.Object,System.Object[],System.Object[]@)"/>

/// or <see cref="M:System.ServiceModel.Dispatcher.IOperationInvoker.InvokeBegin(System.Object,System.Object[],System.AsyncCallback,System.Object)"/> method is called by the dispatcher.

/// </summary>

/// <returns>

/// true if the dispatcher invokes the synchronous operation; otherwise, false.

/// </returns>

public bool IsSynchronous

{

get { return \_iOperationInvoker.IsSynchronous; }

}

#endregion

}

}

Now, to finish the job, all we need to do is apply the attribute to our operation (or use another mechanism if you didn't make the behavior an attribute). Here's what our operation looks like now:

[STAOperationBehavior]

public object DoSomeSTAThing()

{

object returnValue = new object();

// invoke COM objects here

// process messages

return returnValue;

}