The SOA Magazine

Feature Article

Consuming Services with WCF

by Herbjorn Wilhelmsen

Published: September 25, 2009 (SOA Magazine Issue XXXII: September 2009)

Download PDF

Digg This • De.licio.us • Slashdot • Technorati • StumbleUpon • Google Bookmark

Abstract: Microsoft's Windows Communication Foundation (WCF) is an effective framework for implementing services as well as service consumers. Whenever you deal with WCF communication objects you need to pay attention to the disposal of the resources that these objects hold. However, these disposal mechanisms are not that straightforward and are very much related to how resources need to be cleaned up. The how and why of cleaning up service resources is the topic of this article, which is comprised of pre-release content from the upcoming SOA with .NET & Azure book [REF-1], they are equally relevant for asynchronous communications.

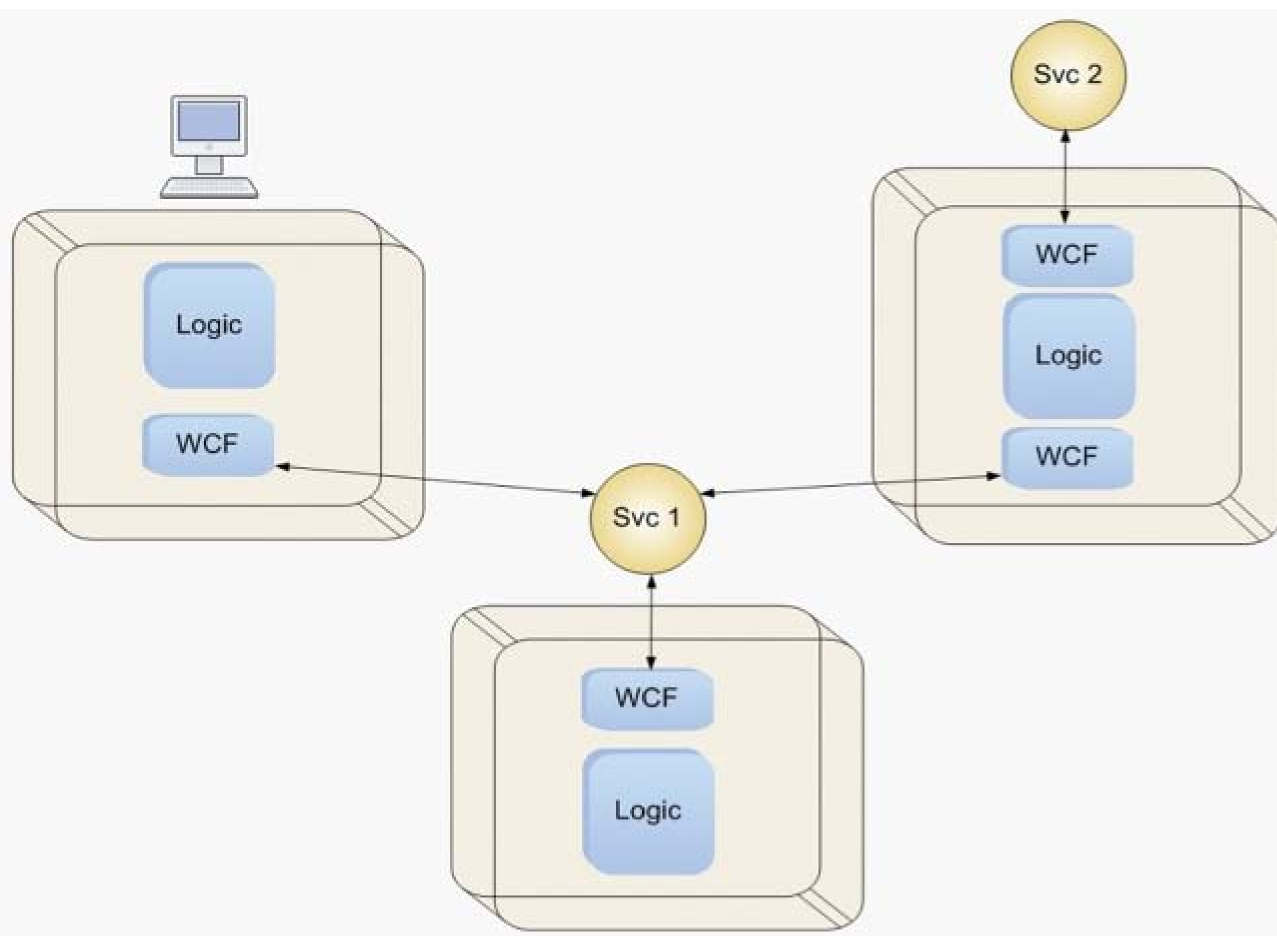

Figure 1: WCF can be used to receive requests and send response messages in services. It can also be used to send requests to and receive response messages from services.

Cleaning Up Resources

Cleaning up resources is something that has been important for as long as programming has existed. In unmanaged languages cleaning up memory resources is essential as it would otherwise lead to memory leaks that could eventually prevent your program from working properly or even bring down an entire machine. Another area where cleaning up resources is considered critical is in the world of database access. Closing database connections is a proven practice that nobody questions since omitting to do so could overpower the database server as it can only keep a limited number of connections open at any given time. One technique that has been developed over the years to optimize connection handling is connection pooling. Each client pools all of its connections as a common resource. As required an application pulls a connection out of the pool. Once you are finished with the connection it is put back into the pool. If you do not close your connection before releasing it, the connection will not be put back into the connection pool in an efficient manner. This leads to extra overhead each time your application needs to connect to a database.

In several respects closing the client connection in a WCF based service consumer is similar to closing a database connection. But there are some important differences. One major difference is that you have two slightly different methods to choose from when you want to close the connection.

The Proper Disposal and Closing of an ICommunicationObject

To consume a service using WCF you have two options: use a class that inherits from ClientBase or connect using the ChannelFactory class. If you generate a proxy using Add Service Reference or SvcUtil you will get a [ServiceName]Client class that inherits from the ClientBase generic class. Both ClientBase and ChannelFactory implement the ICommunicationObject interface which in turn specifies a Close() method.

The ICommunicationObject.Close() method

According to the MSDN documentation of the Close() method, the object implementing ICommunicationObject will not enter the closed state immediately after the Close() method has been called. Instead calling the Close() method will start a graceful transition that works as follows:

The ICommunicationObject immediately enters the Closing state (defined by the `CommunicationState.Closing` enum value) as soon as the Close() method has been called. It will remain in that state while unfinished work, such as sending or receiving buffered messages, is completed. Upon the completion of this work the state of the ICommunicationObject will be changed to the Closed state (defined by the `CommunicationState.Closed` enum value) and then the Close() method returns.

Another very interesting property of the Close() method is that it may throw exceptions. Documented exceptions are

- TimeoutException; will be thrown if the close timeout has elapsed before the Close() method returned.
- CommunicationObjectFaultedException; will be thrown if the Close() method was called on an ICommunicationObject that is in the Faulted state (defined by the `CommunicationState.Faulted` enum value).

Apart from these two exceptions the work that was initiated could throw exceptions too. This flow of events is illustrated in figure 2.

Figure 2: The Close() method's graceful transition to the Closed state.

The ICommunicationObject.Abort() method

The ICommunicationObject interface also specifies an Abort() method , using this method is obviously somewhat different from using the Close() method. Abort() will also transition an ICommunicationObject to the Closed state. However, this transition will not be graceful, as in the case of the Close() method, instead any unfinished work is ignored or terminated and the state will be changed to Closed before the method returns.

Abort() versus Close()

In practice there are only two differences between calling the Abort() and Close() methods:

1. The Close() method may throw exceptions while Abort() cannot throw exceptions.
2. The Close() method allows for unfinished work to be finished within a specified timeout, the Abort() method will cause any unfinished work to be aborted immediately.

This means that if you know for sure that all the work that you have initiated has completed then calling the Abort() method may be a good choice. It also means that if there is a chance that some of your work has not been completed, calling the Close() method is the more sensible choice.

It is fairly easy to create a service that clearly demonstrates the difference between Close() and Abort(). Create a service that communicates on a WS-SecureConversation channel and a client that connects to it. Trace the interaction with SvcTraceViewer and see what happens after calling the Close() and Abort() methods respectively. When you call the Close() method, the graceful transition will allow for a few more messages to be sent in order to remove the conversation token. These extra messages will not be sent if you instead call the Abort() method.

IDisposable for Cleaning Up Resources

The IDisposable interface specifies only one method: Dispose(). The sole purpose of this method is to release and clean up any unmanaged resources that an object has made use of.

A proven practice is to always call the Dispose() method as soon as you finish working with a class that implements IDisposable. The reason behind this is that the implementer of the class probably had a good reason for implementing the IDisposable interface. Typically classes that implement the IDisposable interface make use of unmanaged resources like memory or connections and providing a way of easily disposing of these resources is meant to make it easier for you to write code that uses these classes without getting into trouble. In some cases you may know for sure that the Dispose() method doesn't implement anything (e.g. by inspecting the source code with Red-Gate's .Net Reflector) but even in these cases you are still recommended to call the Dispose() method from your code. Doing so allows you to easily upgrade your program to make use of a newer version of the class with a minimum of fuss. The new version may use resources that need to be disposed of and therefore actually have logic implemented inside the Dispose() method. If you have not called the Dispose() method, you may get a lot of unforeseen problems with the newer version.

IDisposable and Its Relation to ClientBase and ChannelFactory

In its very first incarnation the ICommunicationObject interface implemented the IDisposable interface, but in its current version it does not. However, both ClientBase and ChannelFactory do implement the IDisposable interface. So, both ClientBase and ChannelFactory need to call either the Abort() method or the Close() method from inside their Dispose() method implementations. As a matter of fact, the WCF team has attempted callling both methods in different versions of the WCF framework.

In a very early release of WCF (at that time WCF went by the name of Indigo) the Abort() method was called from inside the Dispose() method. According to the Indigo team this was the first and foremost complaint on Beta 1 of WCF as it lead to cached messages not being sent. To mitigate this, the Indigo team attempted to make the Dispose() method smarter by calling the Close() method if the state of the ICommunicationObject was in the Opened state (defined by the `CommunicationState.Opened` enum value), and otherwise calling the Abort() method. Unfortunately, this way of implementing resource disposal lead to a situation where the Dispose() method could throw exceptions but would not always let you know if something went wrong!

Their ultimate descision was to remove the IDisposable interface from the ICommunicationObject interface, let ClientBase and ChannelFactory implement IDisposable and let both classes call the Close() method from inside their Dispose() methods. The end result is that the Dispose() method of the ClientBase and ChannelFactory classes can throw exceptions. The MSDN documentation for the Dispose() method clearly states that objects that implement the IDisposable interface should ensure that all held resources are freed, but as our discussion below will show this is not always the case with ClientBase and ChannelFactory.

Cleaning Up Resources with the Using Block

A very convenient way of making sure that you do not inadvertently forget to call the Dispose() method, e.g. due to an exception being thrown, is to use a using block. The using block is a short hand for writing a try-finally block. The two following code snippets show semantically identical code.

```
using (myObject)
{
    //some code here
}

myObject;
try
{
    //some code here
}
finally
{
    if (myObject != null)
    {
        myObject.Dispose();
    }
}
```

The using block internally emits IL-code that essentially wraps your code in a try-finally block. In the finally block the Dispose() method will be called. This is actually exactly what you want in the normal case as it means that the Dispose() method will be called even if an exception is thrown from your code inside the using block.

The beauty of the using block syntax is that it is compact and that if you use it you do not have to worry about the disposal of your resources. Unfortunately this also leads to many developers following the proven practice and wrapping ClientBase and ChannelFactory objects in using statements. They are used to writing code like this for their database connections and assume that it will work equally well for ClientBase and ChannelFactory. The problem is that calling the Close() method (remember that the Dispose() methods of both ClientBase and ChannelFactory call the Close() method) of an ICommunicationObject may result in an exception being thrown. This leads to two separate problems: The worst of them is that the connection may not be closed in some situations. The other is that if a CommunicationObjectFaultedException exception is thrown it will hide the exception that was originally thrown from inside your using block. To avoid these problems ClientBase and ChannelFactory objects should not be wrapped by a using block.

Cleaning Up Resources with the Try-Catch-Finally-Abort Pattern

A better way to handle the closing of an ICommunicationObject is to wrap the logic in a try block and to call the Close method from inside the try block. If something goes wrong in the try block, the Abort method should be called from inside a finally block. This approach is sometimes referred to as try-close-finally-abort and is very similar to the way the using block works. The code snippet below shows what this pattern looks like.

```
var finallyClient = new OrganizationClient();
try
{
    //method calls
    finallyClient.Close();
}
finally
{
    if (finallyClient.State != CommunicationState.Closed)
    {
        finallyClient.Abort();
    }
}
```

Please note that the proper criteria for calling the Abort() method is that the state of the channel is something other than Closed. If the channel has another state, it means that something went wrong (an

exception was thrown) either before the Close() method was called or during the execution of the Close() method. In both cases you want to call the Abort() method, in any other case you should not call the Abort() method.

To make this approach as efficient as possible, you should try to include only WCF code and code that is necessary for WCF calls to proceed as planned in the try block. Attempt to place other code before and / or after the try block where possible. Ideally the code inside the try block should only consist of a call to the Open() method, calls to the service, and finally a call to the Close() method. When the .NET runtime exits the try block you will have either finished all the work you need to get done or something went wrong with the calls that went via WCF and your work cannot be completed.

If everything went well you are home free. If something went wrong you can call the Abort() method knowing that you did as much as you could to complete the calls to the composed services.

This code will essentially defer any exception handling code until you have properly disposed of your resources, which is what you want for most scenarios. If it is not what you want, a slightly different approach is needed.

Handling Exceptions and Cleaning Up Resources with the Try-Close-Catch-Abort Pattern

The call to the Abort() method can be wrapped inside a catch block instead of in a finally block. If you need exception handling to take place before closing the channel, this is the correct choice. If you do not need exception handling before closing then the try-close-finally-abort pattern is more suitable as resources will be disposed of before any exception handling occurs. Whenever an exception is caught an exception stack has to be built, this is a time-consuming task. Put slightly differently, if you place the call to the Abort() method inside a catch block you have to wait for the exception stack to be built before you can release your resources.

The following code snippet shows how this can be done.

```
var catchClient = new OrganizationClient();
try
{
    //method calls
    catchClient.Close();
}
catch (Exception)
{
    //code that handles the exception
    catchClient.Abort();
}
```

A variation of this is also possible for cases when you have exception handling code that depends on the exact exception as illustrated in the next code snippet.

```
var catchClient = new OrganizationClient();
try
{
    //method calls
    catchClient.Close();
}
catch (ApplicationException)
{
    //code that handles ApplicationException
}
catch (Exception)
{
    //code that handles all other exceptions
}
finally
{
    if (catchClient.State != CommunicationState.Closed)
{
        catchClient.Abort();
    }
```

```
}
```

Although this pattern is used by quite a few systems, we find that in practice it is hard to find circumstances that justify prioritizing exception handling over resource disposal. It should only be considered if the channel needs to be used as a part of the exception handling or directly after it.

Cleaning Up Resources in a Convenient Way

Adding code for closing channels to all your WCF calls is tedious and error prone. It also defies the DRY (Don't-Repeat-Yourself) principle . As the behavior is essentially the same each and every time you call a service using WCF, this is a good opportunity to write a utility method that encapsulates the required behavior. The proposed utility methods will enable you to write code that looks and feels very similar to the kind of code you would write if you could wrap WCF calls in a using block. The code listing below shows a utility class called WcfClient.

```
public static class WcfClient
{
/// <summary>
/// Performs the action on the client (a <see
cref="ICommunicationObject"/>) using the try-close-finally-abort pattern
/// </summary>
/// <typeparam name="TClient">The type of client</typeparam>
/// <param name="client">The client</param>
/// <param name="action">The action to perform on the client</param>
public static void Using<TClient>(TClient client, Action<TClient> action)
where TClient : ICommunicationObject
{
    if (client == null)
    {
        throw new ArgumentNullException("client");
    }
    if (action == null)
    {
        throw new ArgumentNullException("action");
    }

    try
    {
        client.Open();
        action(client);
        client.Close();
    }
    finally
    {
        if (client.State != CommunicationState.Closed)
        {
            client.Abort();
        }
    }
}


/// <summary>
/// Creates a WCF client channel using a <see cref="ChannelFactory"/> and
performs the action on the channel using the try-close-finally-abort
pattern
/// </summary>
/// <typeparam name="TChannel">The type of channel</typeparam>
/// <param name="channelFactory">The channel factory</param>
/// <param name="action">The action to perform on the created
channel</param>
public static void Using<TChannel>(ChannelFactory<TChannel> channelFactory,
Action<TChannel> action) where TChannel : class
{
    if (channelFactory == null)
    {
        throw new ArgumentNullException("channelFactory");
```

```
    }
    if (action == null)
    {
        throw new ArgumentNullException("action");
    }

    TChannel clientChannel = channelFactory.CreateChannel() ;
    try
    {
        channelFactory.Open();
        action(clientChannel);
        channelFactory.Close();
    }
    finally
    {
        if (channelFactory.State != CommunicationState.Closed)
        {
            channelFactory.Abort();
        }
      clientChannel = null;
    }
}
}
```

All you need to do in order to properly dispose of your ICommunicationObjects is then to write code such as this (if you use a generated WCF client):

```
GetUsersResponse response = null;
WcfClient.Using(new OrganizationClient(), client =>
{
    response = client.GetUsers(new GetUsersRequest());
});
```

Please note that you are able to use any WCF specific classes that need to be used after the channel has been opened. For instance OperationContextScope as discussed in the Idempotency section of chapter 10.

```
GetUsersResponse response = null;
WcfClient.Using(new OrganizationClient(), client =>
{
    using (new OperationContextScope(client.InnerChannel))
    {
        OperationContext.Current.OutgoingMessageHeaders.MessageId =
        messageId;
        response = client.GetUsers(new GetUsersRequest());
    }
});
```

If you prefer to use the generic ChannelFactory class, instead your code would look similar to this:

```
GetUsersResponse response = null;
WcfClient.Using(new ChannelFactory("WSHttpBinding_IOrganization"), client
=>
{
    response = client.GetUsers(new GetUsersRequest());
});
```

How to Handle Connections when Consuming Services Using WCF

Handling and closing connections when you consume services using WCF is obviously an important matter as our discussion above has shown. However, its importance is related to the type of binding that is used.

Different bindings use different protocols. For some protocols it is very important to close the connection, this pertains to connection-full protocols. In such protocols subsequent calls to your consumed service will

happen on the same connection. To make this possible a transport session is maintained between calls. But this is not so for http as it is a stateless or connectionless protocol meaning that the connection is destroyed automatically. In early versions of http the connection was destroyed after each call, but since then it has been optimized to actually keep the connection alive so that multiple response / requests can be transmitted without renegotiation. From this it follows that closing the connection is not critical from the perspective of handling resources when you consume a service over http.

On the other hand, WCF allows you to configure which binding you want to use when you consume a service, and when you do so you may also change the protocol. If you were to consume a WCF service over http and not close your client connection, you would experience no problems at all. However, say that you get the opportunity to consume this service over tcp, which has considerably less overhead than http. You, or someone else several years later for that matter, change the configuration and verifies that the service still works. And it does! Great. But then after a short while you get strange problems. You may soon reach a threshold where the number of open channels blocks your application.

To avoid these kinds of problems, it seems wise to always close the connection as that will enable you to change the configuration and consume a service over different protocols as required without having to worry about changing your code.

There is one exception though: REST services. REST services are built upon principles that define how to use web standards and are per definition connection-less. The only relevant standards that are available today for implementing REST are http and URIs. A REST service built with http will most likely always use http. More to the point: You cannot merely change your configuration to consume a REST service over another protocol than http. Http is an indispensable part of a REST service, not just a transport protocol.

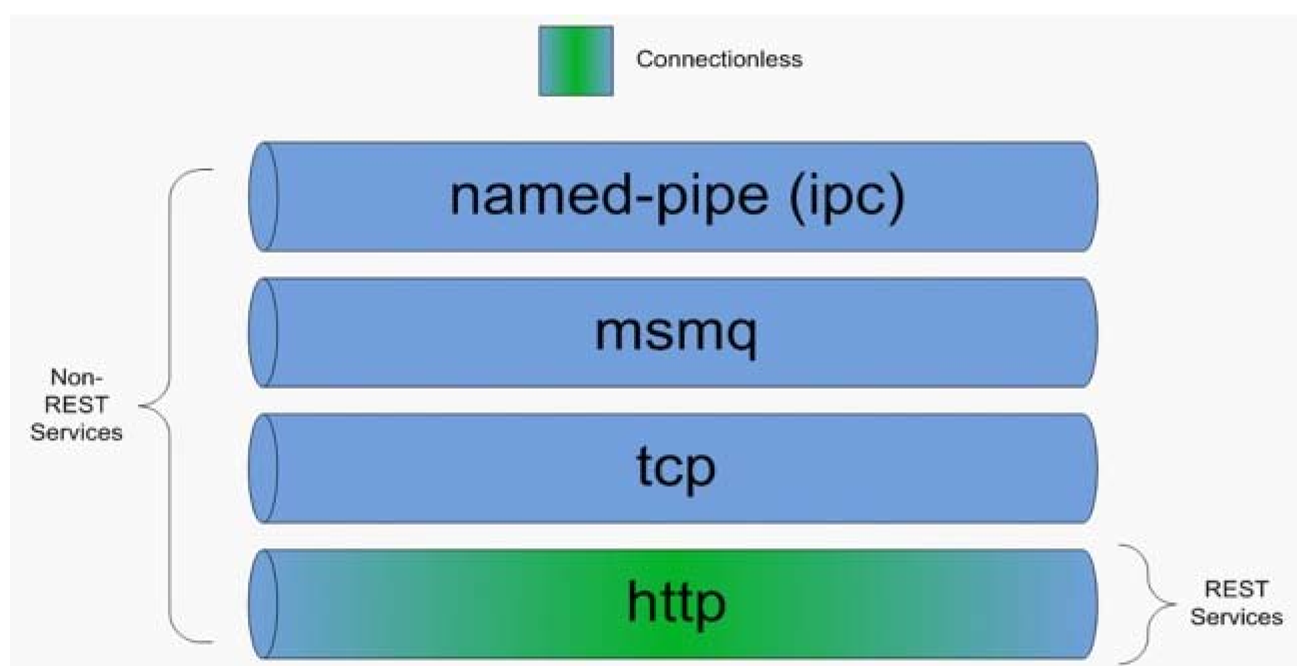

Figure 3: The protocols supported by standard WCF bindings. REST services always uses the http protocol which is connectionless.

So here is at last comes a recommendation: Make sure that you close your client connection whenever you consume a service using WCF, except when you consume a REST service.

Conclusion

Efficient disposal of resources is important and the importance of getting this right should be reflected in your code. Generally, the using block is a great way to ensure that resources are disposed of, but not so in the case of consuming services with the ICommunucationObject. Instead, you need to write custom code to call the Close() or Abort() methods as appropriate. Ideally you should dispose of resources before any exception handling code is executed.

The resources you release from your ICommunucationObject are the resources of the underlying channel.

Some protocols (stateless or connectionless) do not mandate that you close your client connection so this can be safely ommitted when you consume REST services over http.

Acknowledgements

I would like to thank Joshua Anthony (http://www.nsilverbullet.net/) and Scott Seely (http://www.scottseely.com/) for their valuable comments on this article.

References

[REF-1] "SOA with .NET", Prentice Hall/Pearson, ISBN: 0131582313 (scheduled for 2010).

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

Copyright © 2006-2009 SOA Systems Inc.