# WCF Channel Listeners

APRIL 9TH, 2007

Almost two months ago, I wrote a [couple](https://winterdom.com/weblog/2007/02/14/WritingAWCFTransportChannelPart1.aspx) of [entries](https://winterdom.com/weblog/2007/02/19/WritingAWCFTransportChannelPart2.aspx) on writing custom transport channels for Windows Communication Foundation. Those entries and a few that followed on defining URI schemes for transport channels were based on some stuff I was working on at the time. Unfortunately, I got sidetrack with some stuff I had at hand and the project was laid by the side way until this week, when started working on it again.

So last week I rewrote my previous [URI](https://winterdom.com/weblog/2007/02/20/URIsAndWCFTransportChannels.aspx) [scheme](https://winterdom.com/weblog/2007/02/21/URIsInWCFTransportsContinued.aspx), and even got a [basic] IOutputChannel implementation up and running. It's not much, but it feels good to have something working to build upon :-). From what I learned, The ChannelFactory and Output Channel (in it's various forms) is a fairly logical and even somewhat simple model (only made more complex because of the multitude of output channel models to choose from).

Today, though, I started pondering on how to create my ChannelListener and IInputChannel implementation in order to be able to expose WCF services through my custom transport channel. I've got mixed impressions about the model so far. ****Warning:**** There's a bit of ranting below (hopefully not much).

Let's consider for a minute here the shape that a ChannelListener has. Basically, a channel listener exposes a set of methods to the WCF runtime to interact with it, in a sort of state-oriented fashion: OnOpen(), OnWaitForChannel(), OnAcceptChannel(), OnClose() and OnAbort(). Each one also has corresponding BeginXXX() and EndXXX() methods following the asynchronous pattern [1].

At first sight, this set of methods seem make a lot of sense. If you're familiar with programming TCP/IP sockets, then you'll quickly draw some parallels between this model and the familiar Socket operations:

* OnOpen() -> Bind() + Listen()
* OnWaitForChannel() + OnAcceptChannel() -> Accept()
* OnClose() -> Close()

The OnWaitForChannel() and OnAcceptChannel() methods basically follow the pattern: 1) Wait until a connection has been accepted and then 2) create the input channel for the accepted connection, which is why I say that both together sort of relate to the single Accept() method in a Socket.

Having drawn this parallel one might think that this is very good! After all, we were able to draw an immediate parallel to an existing and common network model to base our channel on. However, I don't quite agree with this. Instead, I find the channel listener model to be a bit more awkward, complex and intrusive than I'd strictly necessary. Let me explain why.

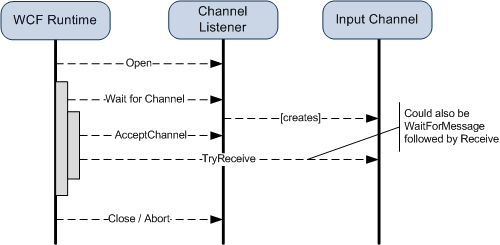
The first issue I have with this model is that it is pretty daunting. Part of this a lot to do with the fact that all operations have both synchronous and asynchronous variants (this is also true for channel factories, but they are simpler), making it seem far more complex than it really is to the new developer (a.k.a. "me"). However, it is not clear at all just when exactly the WCF runtime will decide to call the synchronous or the asynchronous variants of each method. It most certainly does not appear to be discussed in the documentation, or if it is, it's not very easy to find out.

A second and bigger issue is with the API model itself. I mentioned above that to me the model resembled the Sockets programming model. That's obviously good news if your transport protocol is built natively on top of TCP/IP Sockets. But what if it isn't? More to the point, what if the transport protocol has no notion at all of ****"accept"**** operation? Or what happens if that operation is hidden well beneath the guts of a "high"  level transport API that exposes a completely different model (a common thing for Line of Business (LOB) adapters)?

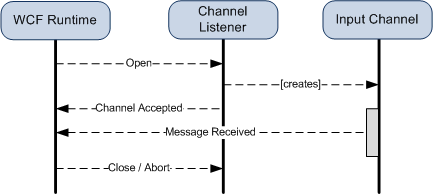
[Nicholas Allen](http://blogs.msdn.com/drnick/) solves the issue for his [File transport channel](http://blogs.msdn.com/drnick/archive/2006/05/02/585612.aspx) by simply throwing an exception from OnAccept(), which is probably not the best workaround this issue because of the performance implications.

More important than this, however, is that the documentation is not very clear on what the play between the WCF runtime (the service dispatcher in this case?) and the channel stack is with respect to WaitForChannel() and friends. Does it do a closed loop where WaitForChannel() is supposed to introduce waiting until a new client is ready to be serviced? What happen if you don't introduce a wait here and instead simply returns true? Does it cause the WCF engine to bog down creating (accepting) miriads of channels? If your channel has no accept model (i.e. if it has no notion of client-specific connections), should your channel listener create essentially a single input channel to do all its work?

One thing that I found somewhat surprising in the WCF model for input channels and channel listeners, however, is that the execution of both of them is driven by the WCF runtime. The channel and listener are not really autonomous, but instead respond to requests done from the WCF runtime, i.e. the runtime asks you to accept a connection, it asks you to create a channel, it asks you to receive a message and so on. In other words, it's a very ****imperative model****.



What's a bit uncomfortable about this is that, from a messaging perspective, things are the other way around: the listener and the input channel drive the execution of the service above them! (a ****reactive model****). My very uneducated guess is that this might have been done to keep the input and output models a bit more symmetric, as well as to make it easier to implement the WS-\* protocols built on the upper layers of the channel model (remember that things like WS-RM are implemented as layered channels sitting on top of the transport channel) [2].



If the model was reactive, then most of the async/sync versions would not be really needed to implement an input channel and channel factory, instead all it would need would be for WCF to give it a way to call back into the runtime to notify it that a new client session has been created/destroyed or a new message has arrived. This seems to me like a more natural model for the receive side, but perhaps it is just a personal preference :-).

In a following article I'll let you know the results of this excercise and how I worked around this issues for my own channel listener implementation.

****[1]**** Actually, the core interfaces are built on the async methods, while the sync versions are added then by the base classes, but looking at this the other way around makes it simpler to illustrate.

****[2]**** One could imagine performance and scalability reasons for doing it this way, as well. Perhaps the designers wanted to force channel developers to keep their hands off the threading model as much as possible so that the WCF runtime could be in control most of the time on how threads from the threadpool are used to service channels, but this is an even more unsubstantiated guess...

<https://winterdom.com/2007/04/09/wcfchannellisteners>