On calling AfxConnectionAdvise with bAddRef set to FALSE

devblogs.microsoft.com/oldnewthing/20231228-00

December 28, 2023

Raymond Chen

A customer had a question about the `AfxConnectionAdvise` function when you set `bAddRef` to `FALSE`.

According to the documentation,

> ```
> BOOL AFXAPI AfxConnectionAdvise(
>     LPUNKNOWN pUnkSrc,
>     REFIID iid,
>     LPUNKNOWN pUnkSink,
>     BOOL bRefCount,
>     DWORD FAR* pdwCookie);
> ```
>
> *bRefCount*
> TRUE indicates that creating the connection should cause the reference count of *pUnkSink* to be incremented. FALSE indicates that the reference count should not be incremented.

They are passing `FALSE` for `bAddRef` when they connect to an out-of-process COM server. What they found is that not only does this suppress the increment of the sink's reference count, it in fact does a spurious *decrement* of the reference count, causing the sink to be destroyed.

The customer looked at the source code for `AfxConnectionAdvise` and saw that if you pass `FALSE` for `bAddRef`, then it calls `Release` on the sink after a successful registration.

```
BOOL AFXAPI AfxConnectionAdvise(LPUNKNOWN pUnkSrc, REFIID iid,
    LPUNKNOWN pUnkSink, BOOL bRefCount, DWORD* pdwCookie)
{
    ASSERT_POINTER(pUnkSrc, IUnknown);
    ASSERT_POINTER(pUnkSink, IUnknown);
    ASSERT_POINTER(pdwCookie, DWORD);

    BOOL bSuccess = FALSE;

    LPCONNECTIONPOINTCONTAINER pCPC;

    if (SUCCEEDED(pUnkSrc->QueryInterface(
                    IID_IConnectionPointContainer,
                    (LPVOID*)&pCPC)))
    {
        ASSERT_POINTER(pCPC, IConnectionPointContainer);

        LPCONNECTIONPOINT pCP;

        if (SUCCEEDED(pCPC->FindConnectionPoint(iid, &pCP)))
        {
            ASSERT_POINTER(pCP, IConnectionPoint);

            if (SUCCEEDED(pCP->Advise(pUnkSink, pdwCookie)))
                bSuccess = TRUE;

            pCP->Release();

            // The connection point just AddRef'ed us.  If we don't want to
            // keep this reference count (because it would prevent us from
            // being deleted; our reference count wouldn't go to zero),
            // then we need to cancel the effects of the AddRef by calling
            // Release.

            if (bSuccess && !bRefCount)
                pUnkSink->Release();
        }

        pCPC->Release();
    }

    return bSuccess;
}
```

It is apparent from the comment “The connection point just AddRef’ed us” that `AfxConnectionAdvise` expects the `IConnectionPoint::Advise` method to increment the reference count of the sink, because it is doing a bonus `Release` to counteract that `AddRef`.

Is this a valid assumption?

No.

One case where `IConnectionPoint::Advise` would not increment the reference count on the sink is the somewhat pathological case where the source knows that it will never call the event sink, so there's no point remembering it.

It's like if somebody hands you a slip of paper and says, "Make sure to call this number if the antenna falls down," but your television set doesn't have an antenna at all. In that case, you can just throw away the slip of paper since you know you will never need it.

Now, as I noted, this is a rather pathological case. After all, a connection point container is unlikely to advertise a connection point that does nothing.[1]

The customer is encountering another case, though: Reference count aggregation through a proxy.

When a COM object is passed to another context (in this case, to another process), the client receives a proxy object. To reduce cross-context chatter, each proxy retains a single reference to the real object on the other side, and any local reference counts are aggregated in the proxy.

For example, suppose we have an object on the server with a single reference held by a client process.

Server	Client
	pObject
	⇓
Object refcount = 1	Object proxy refcount = 1

In these diagrams, heavy arrows are those that cross process boundaries.

If the client calls `pObject->AddRef()` to increment the reference count, COM merely increments the reference count of the proxy. It doesn't send an `AddRef` call over the wire to the server. After the client calls `AddRef`, we have this:

Server	Client
	pObject
	⇓
Object refcount = 1	Object proxy refcount = 2

Notice that the reference count of the proxy has gone up to 2, but there is no change to the reference count of the original object on the server. The proxy aggregates all the references from the client, and only a single reference count is retained from the proxy to the original object. When the proxy's reference count drops to zero, then the proxy is destroyed, and only then does the proxy send a `Release` to the original object.

Okay, now we can set up our story. Suppose you register the same sink against multiple connection points in a remote process. From the point of view of the sink, those remote processes are all clients. (Of course, from the point of view of the connection points, your process is the client.)

```
ComPtr<IDispatch> sink = ⟦ create a sink ⟧;
ComPtr<IConnectionPoint> point1 = ⟦ some remote object ⟧;
ComPtr<IConnectionPoint> point2 = ⟦ some remote object ⟧;

DWORD cookie1, cookie2;
point1->Advise(sink.Get(), &cookie1);
point2->Advise(sink.Get(), &cookie2);
```

This is what things look like before the first `Advise`:

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
`sink`	⇒	Sink Object refcount = 1	
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

The only reference to the sink object is the one you hold in the `sink` local variable.

After the first `Advise`, the Point1 object now holds a reference to the Sink Object, through its own proxy on their side.

You	Them

`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	Sink Object refcount = 2	Sink Proxy refcount = 1
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

With the second `Advise`, things get interesting:

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	Sink Object refcount = 2	Sink Proxy refcount = 2
			⇑
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

COM realizes that it already has a proxy for the sink object on their side, so instead of creating a second proxy, it just reuses the existing one. When `Point2` calls `AddRef` to retain a reference to the sink, the `AddRef` is cached by the proxy, and no `AddRef` happens on the original sink.

What we did above was roughly equivalent to calling `AfxConnectionAdvise` with `bAddRef` set to `TRUE`.

Now let's do it again with `bAddRef` set to `FALSE`.

```
ComPtr<IDispatch> sink = [ create a sink ];
ComPtr<IConnectionPoint> point1 = [ some remote object ];
ComPtr<IConnectionPoint> point2 = [ some remote object ];

DWORD cookie1, cookie2;
point1->Advise(sink.Get(), &cookie1);
sink->Release(); // New!

point2->Advise(sink.Get(), &cookie2);
sink->Release(); // New!
```

Everything is the same through the first `Advise`:

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	Sink Object refcount = 2	Sink Proxy refcount = 1
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

But this time, since `bAddRef` is `FALSE`, the `AfxConnectionAdvise` function tries to undo the `AddRef` performed by the connection point by doing a `sink->Release()`.

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	Sink Object refcount = 1	Sink Proxy refcount = 1
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

The sink object's reference count is back down to 1, so it's as if the sink proxy's `AddRef` had never occurred.

Now we do the second `Advise`:

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	Sink Object refcount = 1	Sink Proxy refcount = 2
			⇑
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

The `AddRef` from the second connection point to the sink is cached in the proxy and is not communicated to the original sink in the server process.

Nevertheless, `AfxConnectionAdvise` sees that `bAddRef` is set to `FALSE`, so it performs a second `sink->Release()`, and bad things happen:

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	Sink Object refcount = 0	Sink Proxy refcount = 2
			⇑
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

Oh no, the sink object's reference count has dropped to zero! This destructs the sink, leaving the `sink` variable and the sink proxy with pointers to freed memory.

You			Them
`point1`	⇒	Point1 Proxy refcount = 1	Point1 Object refcount = 1
			⇓
`sink`	⇒	(freed memory)	Sink Proxy refcount = 2
			⇑
`point2`	⇒	Point2 Proxy refcount = 1	Point2 Object refcount = 1

The original authors of `AfxConnectionAdvise` made an invalid assumption, namely that a successful `Advise` necessarily increments the reference count on the sink. In the case of remote connection points, the reference counts of those remote connection points are aggregated on the remote side, and only a single reference count is maintained on the server side. The `Release()` call thinks it is counteracting the `AddRef()` on the client side, but really it's counteracting a nonexistent `AddRef()` on the server side.

You break the COM rules at your own peril.

Moral of the story: This is a design flaw in `AfxConnectionAdvise`. Do not call it with `bAddRef` set to `FALSE`. Instead of playing games with COM reference counts trying to simulate a weak reference, use a proper weak reference to the sink.

[1] And it may end up not being a valid implementation anyway, because the connection point is expected to produce those clients in response to `IConnectionPoint::EnumConnections`, so it is obligated to retain references to them even though it has no use for them.