Async-Async: Reducing the chattiness of cross-thread asynchronous operations

devblogs.microsoft.com/oldnewthing/20190430-00

April 30, 2019

Raymond Chen

The Windows Runtime expresses the concept of asynchronous activity with the `IAsyncOperation<T>` and `IAsyncAction` interfaces. The former represents an operation that completes asynchronously with a result of type `T`. The latter represents an operation that completes asynchronously with no result; you can think of it as `IAsyncOperation<void>`. In fact, let's just treat it as such for the purpose of this discussion.

When you call a method like `DoSomethingAsync`, it returns an instance of the `IAsyncOperation` interface. All of the details of the `IAsyncOperation` interface are normally hidden from the developer by the language projection. If you are writing in C#, you see a `Task`; in JavaScript, you get a `Promise`. In C++/WinRT and C++/CX, you `co_await` `IAsyncOperation`, and the `co_await` machinery hides the details. In C++/CX, you can also convert the `IAsyncOperation` into a `concurrency::task`, and then schedule your continuation that way.

But today, we're going to look at how things work under the covers.

At the raw interface level, asynchronous operations work like this. In the diagrams, a solid arrow represents a call, and a dotted arrow represents the return from that call.

Client		Server
`DoSomethingAsync()`	→	Start operation
	⇠	return `IAsyncOperation`
`put_Completed(callback)`	→	
	⇠	return
`Release()`	→	
	⇠	release `IAsyncOperation`

… time passes …		
Operation completes		
	←	`callback.Invoke()`
`get_Status()`	→	
	⇠	return Completed (or Error)
`GetResults()`	→	
	⇠	return results
`callback` returns	⇢	`IAsyncOperations` is destroyed

When the client calls the `DoSomethingAsync()` method, the call is sent to the server, which starts the operation and returns an `IAsyncOperation` which represents the operation in progress.

The client calls the `IAsyncOperation::` `put_` `Completed` method to specify a callback that will be invoked when the operation is complete, thereby allowing the client to resume execution when the operation is complete. The server saves this callback and returns.

The client releases the `IAsyncOperation` , since it no longer needs it. The operation itself keeps the `IAsyncOperation` alive.

Time passes, and eventually the operation is complete.

The server invokes the callback to let it know that the operation is complete. The client receives a reference to the original `IAsyncOperation` as part of the callback. The client can interrogate the `IAsyncOperation` to determine whether the operation was successful or not, and if successful, what the result was.

Finally, when the callback returns, there are no more outstanding reference to the `IAsyncOperation` , so it destroys itself.

You may have noticed that this is a very chatty interface between the client and server. I mean, look at all those arrows!

Enter Async-Async.

Async-Async interposes layers on both the client and server which do local caching. The layer returns a fake async operation to the client and provides a fake client to the server.

Client	Client Layer	Server Layer

`DoSomethingAsync()`	→	create fake `IAsyncOperation`		
	⇠	return fake `IAsyncOperation`	→	fake client
`put_Completed(callback)`	→	save in fake `IAsyncOperation`		
	⇠	return		`put_Completed(private`
`Release()`	→			
	⇠	release fake `IAsyncOperation`		`Release()`
… time passes …				
Operation completes				
				`get_Status()`
				`GetResults()`
			⇠	return status and results
		cache status and results		
	←	`callback.Invoke()`		`private` returns
`get_Status()`	→			
	⇠	return cached status		
`GetResults()`	→			
	⇠	return cached results		
`callback` returns	⇢	fake `IAsyncOperation` is destroyed		

With Async-Async, the client's call to `DoSomethingAsync()` creates a fake `IAsyncOperation` on the client side. This fake `IAsyncOperation` makes a call out to the server to initiate the operation, but doesn't wait for the server to respond to the request. Instead, the fake `IAsyncOperation` immediately returns to the client.

As before, the client calls `IAsyncOperation::put_Completed` method to specify a callback that will be invoked when the operation is complete, thereby allowing the client to resume execution when the operation is complete. The fake `IAsyncOperation` saves this callback and returns.

The client releases the fake `IAsyncOperation`, since it no longer needs it. The operation itself keeps the `IAsyncOperation` alive.

Meanwhile, the request from the fake `IAsyncOperation` reaches the server, where a fake client is constructed. This fake client asks the real server to start the operation, and it registers its own private callback to be notified when the operation is complete, and then it releases the `IAsyncOperation`.

Time passes, and eventually the operation is complete.

The server invokes the callback to notify the fake client that the operation is complete. The fake client immediately retrieves the status and result, and transmits both to the fake `IAsyncOperation`, thereby completing the asynchronous call that was initiated by the fake `IAsyncOperation` at the start.

The fake client then returns from its callback, and everything on the server side is now all done.

Meanwhile, the fake `IAsyncOperation` has received the operation's status and result and invokes the client's callback. As before, the client calls the `IAsyncOperation::get_Status()` method to find out whether the operation was successful or not, and it calls the `IAsyncOperation::GetResults()` method to obtain the results of the asynchronous operation from the fake `IAsyncOperation`. The client returns from its callback, and everything on the client side is now all done.

This interface is much less chatty. There is only one call from the client to the server (to start the operation), and only one call from the server back to the client (to indicate the status and result of the operation). All the rest of the calls are local and therefore fast.

From the client's perspective, Async-Async takes asynchronous operations and makes them even more asynchronous: Not only does the operation itself run asynchronously, even the *starting* of the operation takes place asynchronously. This gives control back to the client sooner, so it can do productive things like, say, running other ready tasks.

Note that Async-Async comes into play only when the method call needs to be marshaled. If the client and server are on the same thread, then there is no need for Async-Async because the calls are all local already.

Async-Async was introduced in Windows 10, and it is enabled for nearly all Windows-provided asynchronous operations. There are some methods that do not use Async-Async because they need to start synchronously; UI operations fall into this category.

You can enable Async-Async for your own asynchronous operations by adding the `[remote_async]` attribute to your methods.

```
runtimeclass Awesome
{
  [remote_async]
  Windows.Foundation.IAsyncAction BeAwesomeAsync();
};
```

Although Async-Async is intended to be transparent to the client, there are some things to be aware of. We'll look at those next time.

Raymond Chen

Follow