# Inside C++/WinRT: Coroutine completion handlers: Disconnection

devblogs.microsoft.com/oldnewthing/20230201-00

February 1, 2023

Raymond Chen

C++/WinRT relies on the `Completed` delegate to tell it when a Windows Runtime asynchronous operation is complete. However, it's possible that the `IAsyncAction` or `IAsyncOperation` provider tears itself down without ever calling the `Completed` handler. This typically happens when the provider is running in another process that crashes (or at least disconnects from you). It never calls its completion handler, and the coroutine simple gets leaked.

Here's what you see in the debugger:

```
contoso!winrt::impl::implements_delegate<AsyncActionCompletedHandler,lambda_xxxx>::Rel

combase!<lambda_yyy>::operator()+0xd7
combase!ObjectMethodExceptionHandlingAction<<lambda_yyy> >+0xe
combase!CStdIdentity::ReleaseCtrlUnk+0x64
combase!CStdMarshal::DisconnectWorker_ReleasesLock+0x6e7
combase!CStdMarshal::DisconnectAndReleaseWorker_ReleasesLock+0x35
combase!CStdMarshal::DisconnectForRundownIfAppropriate+0xc9
combase!CRemoteUnknown::RundownOidWorker+0x241
combase!CRemoteUnknown::RundownOid+0x65
RPCRT4!Invoke+0x73
RPCRT4!NdrStubCall2+0x3db
RPCRT4!NdrStubCall3+0xee
combase!CStdStubBuffer_Invoke+0x6f
combase!InvokeStubWithExceptionPolicyAndTracing::__l6::<lambda_zzz>::operator()+0x22
combase!ObjectMethodExceptionHandlingAction<<lambda_zzz> >+0x4d
combase!InvokeStubWithExceptionPolicyAndTracing+0xe1
combase!DefaultStubInvoke+0x268
combase!SyncServerCall::StubInvoke+0x41
combase!StubInvoke+0x303
combase!ServerCall::ContextInvoke+0x517
combase!ComInvokeWithLockAndIPID+0x9a9
combase!ThreadInvokeReturnHresult+0x17b
combase!ThreadInvoke+0x193
RPCRT4!DispatchToStubInCNoAvrf+0x22
RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x1b4
RPCRT4!RPC_INTERFACE::DispatchToStub+0xb3
RPCRT4!RPC_INTERFACE::DispatchToStubWithObject+0x188
RPCRT4!LRPC_SBINDING::DispatchToStubWithObject+0x23
RPCRT4!LRPC_SCALL::DispatchRequest+0x14c
RPCRT4!LRPC_SCALL::QueueOrDispatchCall+0x253
RPCRT4!LRPC_SCALL::HandleRequest+0x996
RPCRT4!LRPC_SASSOCIATION::HandleRequest+0x2c3
RPCRT4!LRPC_ADDRESS::HandleRequest+0x17c
RPCRT4!LRPC_ADDRESS::ProcessIO+0x939
RPCRT4!LrpcIoComplete+0x109
ntdll!TppAlpcpExecuteCallback+0x157
ntdll!TppWorkerThread+0x72c
KERNEL32!BaseThreadInitThunk+0x1d
ntdll!RtlUserThreadStart+0x28
```

The way we address this is to have the completion handler detect that it was never invoked. If that happens, then it simply invokes itself. On resumption, the coroutine will call `GetResults()` on the asynchronous operation, and that will throw the appropriate RPC error.

Keeping track of whether the handler was invoked requires a custom destructor, so we'll convert the lambda to a C++ class first, so that we can add a destructor. This conversion is mechanical.

```
// Original lambda
[
    handle,
    this,
    context = resume_apartment_context()
](auto&& ...)
{
    resume_apartment(context.context, handle,
        &failure);
});

// Converted to explicit class

template<typename Awaiter>
struct disconnect_aware_handler
{
    disconnect_aware_handler(Awaiter* awaiter,
        coroutine_handle<> handle) noexcept
        m_awaiter(awaiter), m_handle(handle) {}

    template<typename...Args>
    void operator()(Args&&...)
    {
        resume_apartment(m_context.context, m_handle,
            &m_awaiter->failure);
    }

private:
    Awaiter* m_awaiter;
    coroutine_handle<> m_handle;
    resume_apartment_context m_context;
};

template<typename Async>
struct await_adapter
{
    ⟦ ... ⟧

    void await_suspend(coroutine_handle<> handle) const
    {
        auto extend_lifetime = async;
        async.Completed(
            disconnect_aware_handler(this, handle));
    }

    ⟦ ... ⟧
};
```

Okay, now we can add a destructor that calls the `operator()` if it had never been called. We'll factor the body into a method `Complete()` and use the null-ness of the `m_handle` to tell us whether the operator has been invoked yet.

```
template<typename Awaiter>
struct disconnect_aware_handler
{
    disconnect_aware_handler(Awaiter* awaiter,
        coroutine_handle<> handle) noexcept
        m_awaiter(awaiter), m_handle(handle) {}

    ~disconnect_aware_handler()
    {
        if (m_handle) Complete();
    }

    template<typename...Args>
    void operator()(Args&&...)
    {
        Complete();
    }

private:
    Awaiter* m_awaiter;
    coroutine_handle<> m_handle;
    resume_apartment_context m_context;

    void Complete()
    {
        resume_apartment(m_context.context,
            std::exchange(m_handle, {}),
            &m_awaiter->failure);
    }
};
```

If you try this, though, it fails miserably: The delegate constructor moves the functor into the newly-constructed delegate, but `coroutine_handle`'s move constructor simply copies the coroutine handle. This means that when the delegate constructor moves the functor, the temporary functor destructs and says, "Oh no, I was never invoked! I must have been disconnected!", and it resumes the coroutine. And then when the coroutine completes for real, the invoke occurs a second time, and we have resumed a running coroutine, which is illegal.

We need custom move operators that null out the coroutine handle in the moved-from object. This is another case where we could have used the `movable_primitive` template type, but C++/WinRT just writes it out by hand.

```
    disconnect_aware_handler(disconnect_aware_handler&& other) noexcept
        : m_context(std::move(other.m_context))
        , m_awaiter(std::exchange(other.m_awaiter, {}))
        , m_handle(std::exchange(other.m_handle, {})) { }
```

We null out the `m_awaiter` just for good measure.

If you see a coroutine resumption from `disconnect_aware_handler`'s destructor when debugging, then that is a sign that the coroutine is resuming due to a disconnection from the Windows Runtime asynchronous operation provider.