Ready… cancel… wait for it! (part 3)

devblogs.microsoft.com/oldnewthing/20110204-00

February 4, 2011

Raymond Chen

A customer reported that their application was crashing in RPC, and they submitted a sample program which illustrated the same crash as their program. Their sample program was actually based on the AsyncRPC sample client program, which was nice, because it provided a mutually-known starting point. They made quite a few changes to the program, but this is the important one:

```
// old code:
// status = RpcAsyncCancelCall(&Async, FALSE);
// new code:
 status = RpcAsyncCancelCall(&Async, TRUE);
```

(It was actually more complicated than this, but this is the short version.)

The program was crashing for the same reason that Wednesday's I/O cancellation program was crashing: The program issued an asynchronous cancel and didn't wait for the cancel to complete. In this case, the crash occurred when the RPC call finally completed and RPC went about cleaning up the call based on the information in the now-freed `RPC_ASYNC_STATE` structure.

The error was probably caused by the not-very-helpful name for that last parameter to `RpcAsyncCancelCall` : `fAbortCall` , and the accompanying documentation which says, "In an abortive cancel (*fAbortCall* is TRUE), the **RpcAsyncCancelCall** function sends a cancel notification to the server and client side and the asynchronous call is canceled immediately, not waiting for a response from the server." Compare this to a nonabortive cancel, where "the **RpcAsyncCancelCall** function notifies the server of the cancel and the client waits for the server to complete the call."

Obviously, it's faster if you don't wait for the server to respond, right? Let's pass `TRUE` , so that the function cancels the asynchronous call immediately without waiting for the server. Wow, look at how fast our program runs now!

Unfortunately, the documentation doesn't make it sufficiently clear that when you issue a cancellation, you still have to wait for the operation to complete before you can clean up all the resources associated with that operation. Another way of looking at that last parameter is to think of it as `fAsync`. If you pass `fAsync = TRUE`, then the `RpcAsyncCancelCall` function issues the cancellation and returns before the operation completes. If you pass `fAsync = FALSE`, then the `RpcAsyncCancelCall` function issues the cancellation and waits for the operation to complete before returning.

If you switch from a synchronous cancel to an asynchronous cancel, then you become responsible for keeping the `RPC_ASYNC_STATE` valid until the cancellation completes. In this case, the customer was using the `RpcNotificationTypeEvent` notification type, which means that they need to wait for the `Async.u.hEvent` to become signaled before they can free the `RPC_ASYNC_STATE`.

The customer confirmed the fix and closed the support case. Another problem solved.

Three months later, the customer reopened the case, reporting that after they released a new version of their program with the aforementioned fix, they were nevertheless getting WinQual crashes which looked exactly like the ones that they were having before they applied the fix. It appears that the fix wasn't working.

Upon closer investigation, it turns out that the customer originally did apply the fix as recommended: They added a `WaitForSingleObject(Async.u.hEvent, INFINITE)` call before destroying the `Async` object to ensure that the cancellation was complete. However, they became frustrated that sometimes the cancellation would take a long time to complete, so they changed it to

```
WaitForSingleObject(Async.u.hEvent, 5000); // wait up to 5 seconds
```

The customer explained, "After the wait fails due to timeout, we just proceed as normal and call `RpcAsyncCompleteCall` and free the the `RPC_ASYNC_STATE`. Is that wrong?"

Um, yeah. Changing the `WaitForSingleObject` from an infinite wait to one with a timeout means that you just reintroduced the bug that the `WaitForSingleObject` was originally supposed to fix! If the cancellation takes more than 5 seconds, then your code will continue and free the `RPC_ASYNC_STATE`, just like it did when you didn't wait at all.

"How long can I wait before assuming that the event will simply never get signaled?"

There is no such duration after which you can safely abandon the operation. Even if the event doesn't get signaled for 30 minutes (say because the computer is thrashing its guts out), it may get signaled at 30 minutes and 1 second.

"But we don't want our program to get stuck waiting for the server."

Great. It's fine to have your program continues running after issuing the cancellation, even if the RPC call hasn't completed. Just don't free the `RPC_ASYNC_STATE` until the call is complete. and if you set things up so that your completion event takes the form of a callback, you can just make the callback free the `RPC_ASYNC_STATE`. Then you don't have to keep track of the asynchronous call any more; the system will merely call you when it's finished, and then you can free the state structure.

Bonus RPC chatter: (For the purpose of this discussion, I'll use the term *RPC operation* instead of *RPC call* so we don't have confusion between function calls and RPC calls.) A colleague explained the lifetime of an RPC operation as follows:

Submit phase	You call into the MIDL-generated stub.	You cannot call `Rpc-AsyncCancelCall` during the submit phase.
	The stub does magic RPC stuff.	
	The stub returns control back to the caller.	
Pending phase	RPC is waiting for the response to the operation. The operation remains in this phase until the operation completes or is cancelled.	You can call `RpcAsync-CancelCall` to cancel the RPC operation and accelerate the transition to the Notified phase.
Notified phase	RPC informs the application of the result of the operation in a manner described by the `NotificationType` and `RPC_ASYNC_NOTIFICATION_INFO` members of the `RPC_ASYNC_STATE` structure.	You can call `RpcAsync-CancelCall` but it will have no effect since the operation is already complete.
Completion phase	The application calls the `RpcAsync-CompleteCall` function to clean up the resources used to track the RPC operation. You exit the completion phase when `RpcAsyncCompleteCall` returns something other than `RPC_S_ASYNC_CALL_PENDING.`	You cannot call `Rpc-AsyncCancelCall` after `RpcAsyncComplete-Call` indicates that the operation is complete, since that is the call that says "I'm all done!"

Raymond Chen

Follow