How can I tell whether a COM pointer to a remote object is still valid?

devblogs.microsoft.com/oldnewthing/20111116-00

November 16, 2011

Raymond Chen

A customer asked the rather suspicious question, "How do I check whether a pointer is valid in another process?" This question should make your head boggle with bewilderment. First of all, we've moved beyond `CrashProgramRandomly` to `CrashSomeOtherProgramRandomly`. Second of all, what the heck are you doing with a pointer in another process? You can't do anything with it! After some back-and-forth[1] we manage to tease the real question out of the customer: How can I tell whether a COM pointer to a remote object is still valid? The easy answer is "Don't worry. COM will take care of it." Just call the method on the object. If the remote object is not valid, you will get an error back, like `RPC_E_DISCONNECTED` or `RPC_E_SERVER_DIED` or `RPC_E_SERVER_DIED_DNE` or `HRESULT_FROM_WIN32(RPC_S_SERVER_UNAVAILABLE)`. When you get an error like that, you'll know that the remote object is no longer valid, and you can respond accordingly. What if you want your program to be a little proactive and prune dead remote objects instead of just noticing that they're dead the net time you want to use them? Some people "solve" this problem by performing a `QueryInterface` on a newly-generated interface ID. Since the IID has never been seen before, COM cannot consult its cache of previously-queried interfaces and must remote the call, at which point the death of the remote object will be detected. (The second rule for implementing `QueryInterface` exists in part so that COM can optimize `QueryInterface` of remote objects.) The problem with this technique is that by subverting the cache, you also end up polluting it. Each time you generate a new IID and do a dummy `QueryInterface` on it, you add another dummy entry to the `QueryInterface` cache. This wastes memory keeping track of interfaces that nobody will ever ask for again, and may even push out information about interfaces that your program actually uses! The COM folks tell me that your program should just accept the fact that the other process can go away at any time. Instead of making some sort of decision based on whether the other process is still there (since a response of "yeah, it's still here" could be wrong by the time you act on it), you should just call the method and accept that it may fail because the other process vanished while you weren't looking. **Footnote** [1] The customer first explained that their server process created an object and gave a pointer to that object to the client. The client then registered a callback object with the server, and the server wanted to check that the client object was still valid before invoking any methods on it. When asked, "Why not just use COM?" the customer

replied, “We are using COM. We create the object on the server via `CoCreateInstance`, then register the client object via a method on our interface.” The customer was under the impression that when a COM pointer refers to an object in another process, you just get that pointer from the other process. If you think about it, this makes no sense at all. How could any of your method calls work? You call `pRemoteObject->AddRef()` and the compiler is going to deference the `pRemoteObject` pointer, and then crash because the pointer would refer to memory in another process. I guess the customer was under the impression that some *magic voodoo* happens so that the CPU knows that “Oh wait, this pointer really belongs to another process, let me go fetch the memory from that other process. Okay, and now you want to call a function pointer in another process? Okay, um, let me magically merge the two processes together so the remote code running in that other process can access the objects in your process.” Or something.

When you have a COM pointer to an object in another process, the pointer that you have is a *proxy* which accepts method calls and *marshals* the call to the real object somewhere else.

Raymond Chen

Follow