# Understanding COM Apartments, Part II

**Rating:** 

**Jeff Prosise** (view profile)
April 13, 2001

Source: Codeguru.com - http://www.codeguru.com/cpp/com-tech/activex/apts/article.php/c5533/

In my previous column, I described the hows and whys of COM apartments. If you read it, you now know that when a thread calls COM's CoInitialize or CoInitializeEx function, that thread is placed in an apartment. And you know that when an object is created, it, too, is placed in an apartment, and that COM uses ThreadingModel values in the registry to determine what types of apartments to place in-proc objects in.

You also know that there are three types of apartments: single-threaded apartments, or STAs; multithreaded apartments, or MTAs; and neutral-threaded apartments, or NTAs. Windows 2000 supports all three apartment types; Windows NT 4.0 supports only two (STAs and MTAs). COM apartment types have the following distinguishing characteristics:

- STAs are limited to one thread each, but COM places no limit on the number of STAs per process. When a method call enters an STA, it is transferred to the STA's one and only thread. Consequently, an object in an STA will only receive and process one method call at a time, and every method call that it receives will arrive on the same thread. COM transfers an incoming call to an STA thread by posting a private message to the hidden window that serves that STA.
- MTAs are limited to one per process, but do not limit the number of threads that can run inside them. Method calls destined for an object in an MTA are transferred to arbitrary RPC threads as they enter the apartment. COM makes no attempt to serialize method calls bound for MTAs, so an MTA-based object can receive concurrent calls on concurrent threads. Because incoming calls are transferred to RPC threads, every call to an MTA-based object is likely to arrive on a different thread-even if every one of those calls comes from the same caller.
- NTAs were added to Windows 20000 as a performance optimization. Method calls entering STAs and MTAs incur thread switches that account for most of the overhead involved in interapartment method calls. Calls entering an NTA incur no thread switching. If an STA or MTA thread calls an NTA-based object in the same process, the thread temporarily leaves the apartment that it's in and executes code in the NTA.

Some of the information presented in last month's "Into the IUnknown" column must have seemed hopelessly abstract at the time. This month you'll see why understanding the arcana of apartments is important if you wish to avoid the pitfalls that afflict all too many COM programmers. It all boils down to understanding (and obeying) two sets of rules: one for COM clients, another for COM servers. Obey the rules and life with COM will be as joyous as that scene from "The Sound of Music" where Julie Andrews sings the movie's theme song while spinning like a top in an Alpine meadow. Break the rules, and you'll probably encounter insidious errors that are difficult to reproduce and even more difficult to diagnose. I get e-mail about these errors all the time. It's time to do something about them.

## Writing COM Clients That Work

Here are three rules for writing COM clients that work. Take them to heart and you'll avoid many of the crippling mistakes made by developers who write COM clients.

### Rule 1: Client Threads Must Call CoInitialize[Ex]

Before a thread does anything that involves COM, it should call either CoInitialize or CoInitializeEx to initialize COM. If a client application has 20 threads and 10 of those threads use COM, then each of those 10 threads should call CoInitialize or CoInitializeEx. Inside these API functions, the calling thread is assigned to an apartment. If a thread isn't assigned to an apartment, then COM is powerless to enforce the laws of COM concurrency for that thread. Don't forget, too, that a thread that calls CoInitialize or CoInitializeEx successfully needs to call CoUninitialize before it terminates. Otherwise, the resources allocated by CoInitialize[Ex] aren't freed until the process ends.

It sounds simple-remembering to call CoInitialize or CoInitializeEx before using COM-and it is simple. It's one function call. Yet you'd be amazed at how often this rule is broken. Most of the time the error manifests itself in the form of failed calls to CoCreateInstance and other COM API functions. But occasionally the problems don't show up until much later, and have no obvious connection to the client's failure to initialize COM.

Ironically, one of the reasons that developers sometimes don't call CoInitialize[Ex] is that Microsoft tells them they don't have to. MSDN contains a document advising developers that COM clients can sometimes avoid calling these functions. The document goes on to say that access violations may result. I recently got a call from a developer whose client threads' calls to Release were locking up or generating access violations. The reason? Some of the threads were placing method calls without first calling CoInitialize or CoInitializeEx. When calls to Release blow up, you have problems. Fortunately, fixing the problem was a simple matter of adding a few calls to CoInitialize[Ex].

Remember: It's never harmful to call CoInitialize[Ex], and it should be considered mandatory for any and all threads that call COM API functions or utilitize COM objects in any way.

## Rule 2: STA Threads Need Message Loops

Rule 2 isn't very obvious unless you understand the mechanics of single-threaded apartments. When a client places a call to an STA-based object, the call is transferred to the thread that lives in that STA. COM accomplishes the transfer by posting a message to the STA's hidden window. So what happens if the thread in that STA doesn't retrieve and dispatch messages? The call will disappear into the RPC channel and never return. It will languish in the STA's message queue forever.

When developers call me and ask why method calls aren't returning, the first thing I ask them is "Is the object that you're calling in an STA? And if so, does the thread that drives that STA have a message loop?" More often than not, the answer is "I don't know." If you don't know, you're playing with fire. When a thread calls CoInitialize, when it calls CoInitializeEx with a COINIT_APARTMENTTHREADED parameter, or when it calls MFC's AfxOleInit function, it's assigned to an STA. If objects are later created in that STA, those objects can't receive method calls from clients in other apartments if the STA's thread lacks a message pump. The message pump can be as simple as this:

```
MSG msg;
while (GetMessage (&msg, 0, 0, 0))
 DispatchMessage (&msg);
```

Be wary of placing a thread in an STA if it lacks these simple statements. One common scenario in which it happens is when an MFC application launches a worker thread (MFC worker threads are, by definition, threads that lack message pumps) and that thread calls AfxOleInit, which places it in an STA. You'll get away with it if the STA doesn't end up hosting any objects, or if it does host objects but the objects don't have clients in other apartments. But if that STA ends up hosting objects that export interface pointers to other apartments, calls placed through those interface pointers will never return.

## Rule 3: Never Pass Raw, Unmarshaled Interface Pointers Between Apartments

Suppose you're writing a COM client that has two threads. Both threads call CoInitialize to enter an STA, and one thread-thread A-uses CoCreateInstance to create a COM object. Thread A wants to share the interface pointer that it received from CoCreateInstance with thread B. So thread A sticks the interface pointer in a global variable and signals thread B that the pointer is ready. Thread B reads the interface pointer from the global variable and proceeds to place calls to the object through the interface pointer. What, if anything, is wrong with this picture?

The scenario that I just described is an accident waiting to happen. The problem is that thread A passed a raw, unmarshaled interface pointer to a thread in another apartment. Thread B should only communicate with the object through an interface pointer that has been marshaled to thread B's apartment. "Marshaling" in this context means giving COM the opportunity to create a new proxy in thread B's apartment so thread B can safely call out. The consequences of passing raw interface pointers between apartments range from extremely timing-dependent (and difficult to reproduce) data corruption errors to outright lockups.

Thread A can safely share an interface pointer with thread B if it marshals the interface pointer. There are two basic ways a COM client can marshal an interface to another apartment:

- The COM API functions CoMarshalInterThreadInterfaceInStream and CoGetInterfaceAndReleaseStream. Thread A calls CoMarshalInterThreadInterfaceInStream to marshal the interface pointer; thread B calls CoGetInterfaceAndReleaseStream to unmarshal it. Inside CoGetInterfaceAndReleaseStream, COM creates a new proxy in the caller's apartment. If the interface pointer doesn't need to be marshaled (if, for example, the two threads share an apartment), then CoGetInterfaceAndReleaseStream is smart enough not to create a proxy.
- The Global InterfaceTable (GIT), which debuted in Windows NT 4.0 Service Pack 3. The GIT is a per-process table that enables threads to safely share interface pointers. If thread A wants to share an interface pointer with other threads in the same process, it can use IGlobalInterfaceTable::RegisterInterfaceInGlobal to put the interface pointer in the GIT. Then, threads that want to use that interface pointer can call IGlobalInterfaceTable::GetInterfaceFromGlobal to retrieve it. The magic is that when a thread retrieves an interface pointer from the GIT, COM marshals the interface pointer to the retrieving thread's apartment.

Are there times when it's OK not to marshal an interface pointer that you intend to share with another thread? Yes. If the two threads share an apartment, which can only happen if the threads belong to the MTA, then they can share raw, unmarshaled interface pointers. But if in doubt, marshal. It's never harmful to call CoMarshalInterThreadInterfaceInStream and CoGetInterfaceAndReleaseStream or use the GIT because COM won't marshal the pointer if it doesn't have to.

# Writing COM Servers That Work

There's an equally binding set of rules that you should follow when you write COM servers. They're described below.

## Rule 1: Protect Shared Data in ThreadingModel=Apartment Objects

One of the most common misconceptions about COM programming is that marking an object ThreadingModel=Apartment absolves the developer from having to think about thread safety. It doesn't. When you register an in-proc object ThreadingModel=Apartment, you're implicitly promising COM that instances of that object (and other objects created from that DLL) access shared data in a thread-safe manner. That means you use critical sections or other thread synchronization primitives to ensure that only one thread can touch the data at a time. Data shared between object instances typically comes in three forms:

- Global variables declared in a DLL
- Static member variables in C++ classes
- Static local variables

Why is thread synchronization so important for ThreadingModel=Apartment objects? Consider the case in which two objects-object A and object B-are created from the same DLL. Suppose that both objects read and write a global variable declared in that DLL. Because the objects are marked ThreadingModel=Apartment, they might be created in separate STAs and run, therefore, on two different threads. But the global variable that they access is shared because it's only instanced into the process one time. If calls go out to objects A and B at about the same time, and if object A writes to that variable at the same time thread B reads from it (or vice versa), the variable could be corrupted-unless you serialize the threads' actions. Fail to provide a synchronization mechanism and you'll get away with it most of the time. But eventually the two threads will probably collide over the shared data, and there's no telling what might happen as a result.

Is it ever safe to allow COM objects to access shared data without synchronizing those accesses? Yes, under the following circumstances:

- Objects that have no ThreadingModel value registered (which we refer to as ThreadingModel=None or ThreadingModel=Single) all run on the same thread in the same STA and therefore can't collide over shared data.
- If you're sure that the objects will run in the same STA even though they're marked ThreadingModel=Apartment (for example, if they're all created by the same STA thread).

- If you're sure that the objects won't process method calls concurrently.

Absent these circumstances, make sure any ThreadingModel=Apartment objects that you write access shared data in a thread-safe manner. Only then can you rest assured knowing that you've fulfilled your end of the bargain.

## Rule 2: Objects Marked ThreadingModel=Free or ThreadingModel=Both Should be Wholly Thread-Safe

When you register an object ThreadingModel=Free or ThreadingModel=Both, the object either will be (Free) or could be (Both) placed in an MTA. Remember that COM does not serialize calls to MTA-based objects. Therefore, unless you know beyond the shadow of a doubt that these objects' clients will not place concurrent calls to them, the objects should be entirely thread-safe. That means your classes must synchronize access to their own non-static member variables in addition to synchronizing accesses to data shared by object instances. Writing thread-safe code isn't easy, but you must be prepared to do it if you plan to use the MTA.

## Rule 3: Avoid Using TLS in Objects Marked ThreadingModel=Free or ThreadingModel=Both

Some Windows developers use thread-local storage (TLS) as a temporary data store. Imagine that you're implementing a COM method and you want to cache some information about the current call where you can retrieve it when the next call arrives. You might be tempted to use TLS. And in an STA, it would work. But if the object you're writing resides in an MTA, you should avoid TLS like the plague.

Why? Because calls that enter an MTA are transferred to RPC threads. Each call is liable to arrive on a different RPC thread, even if every one of those calls originated from the same thread and the same caller. Thread B can't access thread A's thread-local storage, so if call number 1 arrives on thread A and the object tucks data away in TLS, and if call number 2 arrives on thread B and the object attempts to retrieve the data it wrote to TLS in call number 1, the data won't be there. Plain and simple.

Beware using TLS to cache data between method calls in MTA-based objects. It'll only work if all the calls come from the same thread in the same MTA that the object resides in.

# You're Kidding, Right?

Am I serious about all these rules? You bet. Roughly half of all bugs I find in COM-based applications stem from violations of the rules prescribed in this article. Even if you don't understand them, abide by them. The world will be a better place if you do.