Pool Your COM+ Objects

Drawing from a pool of ready-to-use objects can satisfy the thousands of clients clamoring for objects at peak times.

by Juval Lowy

WHAT YOU NEED

Windows 2000

Visual C++ 6.0 or Visual Studio.NET beta 1 or later

(Note: Visual Basic developers can use object pooling only in VB.NET)

Survival in today's Internet-centric world depends on your ability to handle a large number of clients. Dedicating one server object per client, as you do in the classic two-tier client/server model, drains critical resources quickly when you have thousands of clients hammering on your system at peak load. You can't realistically allocate resources such as a database connection, a system handle, or a worker thread for each client; you must recycle these resources and amortize the cost of creating them across many clients.

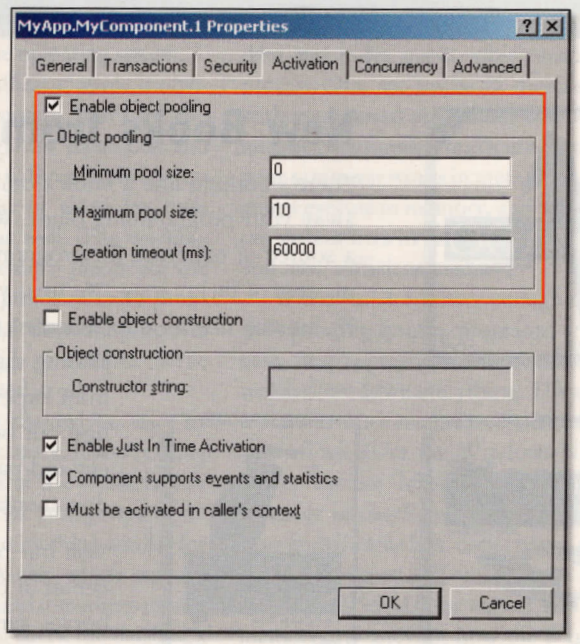

Figure 1 Configure the Pool Parameters. Use the Activation tab on your COM+ component's properties page to configure the way clients create and access instances of the component. You can configure object pooling—including pool size and creation timeout values—as well as enable Just In Time Activation.

Avoid paying this time and resource price for every client request by using object pooling—constructing a pool of already created objects that are ready to serve clients. Object pooling comes in handy even in situations where you have a smaller number of clients but sudden spikes in demand for objects. I'll explain when to use object pooling, how it works under COM+, and how to design your components to take advantage of it. I'll finish by briefly introducing you to object pooling in .NET.

You should use object pooling when instantiating an object is a costly operation, or when you need to control access to scant resources. Object pooling is most appropriate when the object initialization is generic enough that it doesn't require client-specific parameters. When using object pooling, strive to perform as much of the time-consuming work that's common to all clients in the object's constructor, such as acquiring connections (OLE DB, ADO, and ODBC), running initialization scripts, initializing external devices, creating file handles, fetching initialization data from files or across a network, and so on. Avoid using object pooling if constructing a new object doesn't take a lot of time because pool management incurs a fixed overhead every time the client creates or releases an object.

Implementing an object pool on your own is no small matter. You must write code that manages the pool size, returns objects to clients and back to the pool, denies client requests for objects if the pool is exhausted, handles concurrent access to the pool, and so on. You must also provide your client with a dedicated means of creating objects that goes through your pool. By doing so, you couple your clients to your pooling schema so any change to the pool mechanism affects all your clients. Client programmers will have to change their code, recompile, retest, and redeploy. Fortunately, COM+-configured

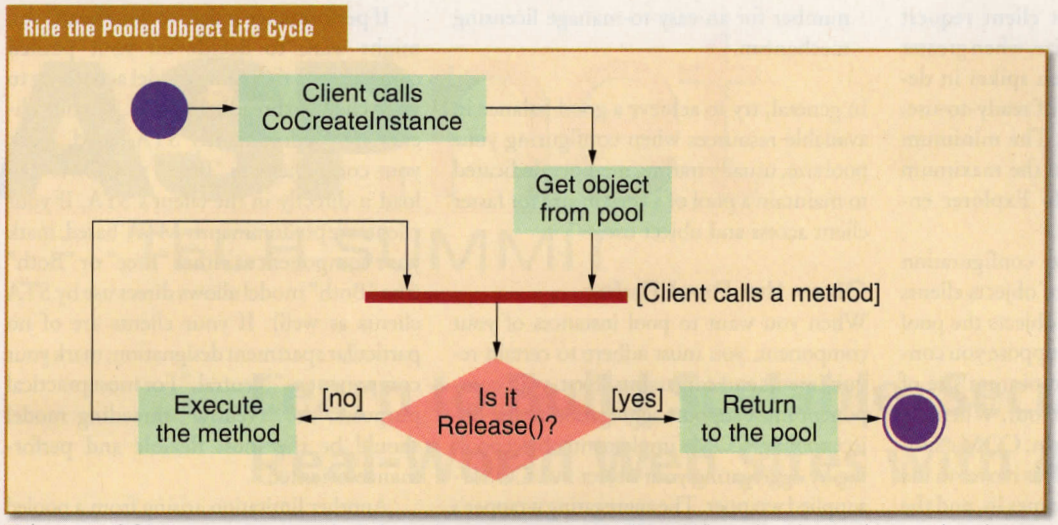

Figure 2 COM+ retrieves an object from the pool to serve a client. When the client releases the object, COM+ returns it to the pool instead of destroying it. This Unified Modeling Language (UML) activity diagram demonstrates that COM+ object pooling is designed to minimize the cost of creating an object, not the cost of using it (see Resources).

components can use the COM+ object pooling service—a simple and elegant generic pooling schema almost any component can use.

You control the way clients create and access instances of a COM+ component on its property page's Activation tab (see Figure 1). COM+ provides two instance-management services: object pooling and Just In Time Activation (JITA). I'll address JITA in a future article.

Neither technique is a COM+ innovation. What's new is the ease with which you can take advantage of the services. You can focus your development efforts on the domain problem at hand rather than the instance-management plumbing. Unlike custom object pooling solutions, the COM+ pooling mechanism isn't coupled to the client's code because COM+ "owns" CoCreateInstance() and New, and intercepts the Release() call the client makes on the object. So you can use the same client code for pooled and nonpooled objects.

COM+ maintains a pool for each component type. You pay the cost of creating a new object only once, then reuse the same instance with many clients. COM+ recycles the same object instance repeatedly for as long as the containing application runs. COM+ calls the object's constructor and destructor only once. The COM+ object pooling mechanism is designed to deal with numerous clients that create objects for every request and release their object references when the request processing completes rather than holding references on the objects.

Take a Dip in the Pool

Any COM+ application can host object pools, be it a server or a library application. A server application hosts one pool per machine. If you install proxies to that application on other machines, the pool can serve the local network. If you have a library application, however, COM+ creates a pool of objects per client process that loads the library application. As a result, two clients in different processes end up using two distinct pools. If you want only one pool of objects, configure your application to be a server application.

Take a look at the life of a pooled object (see Figure 2). When a client issues a request to create a component instance, and that component is configured to use object pooling, COM+ first checks the pool for an available object instead of creating the object. COM+ returns the object to the client if it finds one in the pool. If the pool has no object available and hasn't yet reached its maximum configured size, COM+ creates a new object and hands it back to the creating client. In any case, COM+ stays out of the way once a client gets a reference to the object. In every respect except one, the client's interaction with the object proceeds as an interaction with a nonpooled object. When the client calls the final release on the object (when the reference count goes down to zero), COM+ returns the object to the pool instead of releasing it. The client can hold on to the object for as long as it needs to. Object pooling minimizes the cost of creating an object, not the cost of using it.

Use object pooling by checking the "Enable object pooling" checkbox on the Activation tab (see Figure 1). Then configure the pool parameters: minimum and maximum pool size, and the object-creation timeout. The minimum pool size determines how many objects COM+ should keep in the pool, even when no clients want an object. When you first launch an application configured to contain pools of objects, COM+ creates the specified minimum number of objects for each pool. If the minimum pool size is zero, COM+ doesn't create

RESOURCES

- For an explanation of Unified Modeling Language (UML) activity diagrams: **UML Distilled, Second Edition** by Martin Fowler and Kendall Scott (Addison Wesley Longman, 1999, ISBN: 020165783X)
- MSDN Library: http://msdn.microsoft.com/library/default.asp

any objects until the first client request comes in. Minimum pool size, when greater than zero, mitigates sudden spikes in demand by offering a cache of ready-to-use, already initialized objects. The minimum pool size must be less than the maximum pool size, and the COM+ Explorer enforces this condition.

The maximum pool size configuration controls the total number of objects clients can create, not how many objects the pool can contain. For example, suppose you configure the pool to have a minimum size of zero and a maximum size of four. When the first creation request comes in, COM+ simply creates an object and hands it over to the client. If a second request comes in, and the first object is still tied up by the first client, COM+ creates a new object and hands it over to the second client. The same is true for the third and fourth clients. But when a fifth request comes along, four objects have already been created and the pool has reached its maximum potential size. If all objects are in use once you reach that limit, COM+ blocks further client requests until it returns an object to the pool, after which it hands over the object to the waiting client.

If, on the other hand, the client waits for the duration specified in the timeout and no client returns an object to the pool, COM+ unblocks the client and CoCreateInstance() returns the error code CO_E_ACTIVATIONFAILED_TIMEOUT. COM+ maintains a queue of waiting clients for each pool and services them on a first-come first-served basis as objects are returned to the pool. A creation timeout of zero causes all client calls to fail, regardless of pool state and object availability.

If the pool contains more objects than the configured minimum size, COM+ periodically cleans up the pool and destroys the surplus objects. Deciding on the minimum and maximum pool size configuration depends largely on the nature of your application and the work your objects perform. Consider these factors when configuring your pool size:

- Expected system load, including highs and lows.
- Performance profiling done on your product to optimize the resource usage.
- Various parameters captured during installation, such as user preferences and memory size.
- Number of licenses your customer has paid for. You can set the pool size to this number for an easy-to-manage licensing mechanism.

In general, try to achieve a good balance in available resources when configuring your pool size, usually trading memory dedicated to maintain a pool of a certain size for faster client access and object use.

Obey the Pool Rules

When you want to pool instances of your component, you must adhere to certain requirements and constraints. First, your component must support aggregation to use object pooling. COM+ implements object pooling by aggregating your object in a COM+-supplied wrapper. The aggregating wrapper's implementation of AddRef() and Release() manages the reference count and returns the object to the pool when the client has released its reference. When you import a COM component into a COM+ application, COM+ verifies that your component supports aggregation. If it doesn't, COM+ disables object pooling in the COM+ Explorer. If you implement your object using the Active Template Library (ATL), make sure your code doesn't contain the ATL macro DECLARE_NOT_AGGREGATABLE(), because it prevents your object from being aggregated. In particular, when using Visual C++ 6.0, the ATL wizard inserts this macro into your component's header file when generating Microsoft Transaction Server (MTS) components. You can remove this macro safely to enable object pooling.

Also pay attention to your pooled object's threading model: A pooled object should have no thread affinity. It should make no assumption about the identity of the thread it's executing on or use thread-local storage because the execution thread can differ each time the object leaves the pool to serve a client. So the pooled object can't use the single-threaded apartment (STA) model because STA objects always require execution on the same thread. If the component's threading model is marked as "Apartment" (STA), COM+ disables object pooling for that component when you import it into a COM+ application. A pooled object can use only the "Free" multi-threaded apartment (MTA), the "Both" threading model, or the neutral-threaded apartment (NTA). You can't develop pooled objects using Visual Basic 6.0 because all COM components developed in VB6 are STA-based and make use of thread-local storage, but you can develop them with Visual Basic.NET.

If performance is dear to your heart, you might want to decide on your pooled component's threading model according to your clients' threading model. If your clients are predominantly STA-based, mark your component as "Both" so COM+ can load it directly in the client's STA. If your clients are predominantly MTA-based, mark your component as either "Free" or "Both" (the "Both" model allows direct use by STA clients as well). If your clients are of no particular apartment designation, mark your component as "Neutral." For most practical purposes, the "Neutral" threading model should be the most flexible and performance-oriented.

Another limitation arising from a pooled object's inability to use STA: Pooled objects cannot display a user interface because all user interfaces require the STA message loop.

Let's look ahead to object pooling in .NET. With all its innovations and advanced concepts, .NET is only a component technology—it provides you with the means to build binary components rapidly, much as COM does. But it doesn't have component services such as object pooling, so it relies on COM+ to provide them. A .NET component that takes advantage of COM+ services must derive from the .NET base class ServicedComponent. You use the ObjectPooling attribute to configure every aspect of your .NET component's object pooling. This attribute enables or disables object pooling, as well as sets the minimum or maximum pool size and object-creation timeout. For example, write this C# code to enable your component's object pooling with a minimum pool size of three, maximum pool size of 10, and creation timeout of 20 seconds (20,000 milliseconds):

```
[ObjectPooling(MinPoolSize =
3,MaxPoolSize = 10,CreationTimeout =
20000)]
public class MyComponent
:ServicedComponent

{...}
```

Or, use this VB.NET code:

```
Public Class
<ObjectPooling(MinPoolSize:=
3,MaxPoolSize:=10,
CreationTimeout:=20000)> MyComponent
Inherits ServicedComponent
...
End Class
```

MinPoolSize, MaxPoolSize, and CreationTimeout are public properties of the ObjectPooling attribute class. If you don't specify values for these properties (all or just a subset), the default COM+ values go into effect when you register your .NET component with a COM+ application.

The ObjectPooling attribute class has a Boolean property called Enabled. The attribute's constructor sets Enabled to true if you don't specify a value for it. In fact, the attribute's constructor has a couple overloaded versions: a default constructor that sets Enabled to true, and a constructor that accepts a Boolean parameter. All constructors set the pool parameters to the default COM+ values. So these three C# statements are equivalent:

```
[ObjectPooling]
[ObjectPooling(true)]
[ObjectPooling(Enabled = true)]
```

And these VB.NET statements are equivalent as well:

```
<ObjectPooling()>
<ObjectPooling(True)>
<ObjectPooling(Enabled := True)>
```

As you might know already, .NET components rely on a garbage collector rather than reference counting to manage their life cycle. A .NET pooled object returns to the pool when it's garbage-collected.

Object pooling is a powerful instance-management service you can use for COM and .NET components alike. You'll find it most useful when creating objects proves costly in either time or resources (or both), and when the clients don't hold onto the objects for long periods of time—such as Internet clients, because of the stateless nature of HTTP. If the clients do hold onto their objects for awhile, consider combining object pooling with Just In Time Activation, which makes objects return to the pool between method invocations. VBPJ

About the Author

Juval Lowy is a seasoned software architect and the founder of IDesign, a consulting and training firm focused on COM/.NET design. Juval consults, conducts training classes, and delivers conference talks on component-oriented design and development. Reach him through www.componentware.net.

This article is based on excerpts from his book, **COM+ Services: The Definitive Guide – Mastering COM and .NET Component Services** [O'Reilly & Associates Inc., 2001, ISBN: 0596001037]. Reprinted with permission. Reach O'Reilly at www.oreilly.com.

GO ONLINE

Use these DevX Locator+ codes at www.vbpj.com to go directly to these related resources.

VB0107 Download all the code for this issue of **VBPJ**.

VB0107BB_T Read this article online. DevX Premier Club membership is required.

Want to subscribe to the Premier Club? Go to www.devx.com.