WHAT YOU NEED

Windows 2000

Visual C++ 6.0

Microsoft Platform SDK

Get Into a Log Rhythm

by Juval Lowy

Learn to use COM+ to create application logging for a robust application and faster time to market.

One of the most crucial actions you can take to achieve a robust application and faster time to market is to employ application logging. In this article, I present the COM+ logbook, a simple utility that enables you to log method calls, events, and errors, as well as various COM+ information. The logbook is your product flight recorder, and in a distributed COM+ environment, it's worth its weight in gold. It saved my skin whenever I tried to analyze why things didn't work the way they were supposed to (ever happen to you?).

Simply examining the log files lets you analyze what took place across machines and applications, and the source of the problem is almost immediately evident. The logbook is also useful in troubleshooting customer problems post-deployment; the customers simply send you the log files.

Logbook's Many Requirements

Logbook is a versatile server application that be used in any Win32 app (such as MFC or classic COM), not just COM+, and its only requirement is that the app runs on Windows 2000. Here are the requirements set for the logbook:

1. Trace the calling tree (the causality) from the original client down to the lowest components, across threads, process, and machines; thereby, tracing the logical thread of execution.
2. Log the times and locations of calls, events, and errors.
3. All the calls from all the applications should be interleaved—in order—in one log file.
4. Log the current COM+ execution context.
5. Be able to customize what is logged administratively (for example, events and errors, or just errors).
6. Be able to specify the log file name administratively.
7. Logging and tracing should be as easy as possible for the applications.
8. The log data should be accessible in two formats: HTML or XML.
9. The logbook application and the applications using it should have different lifelines.
10. Be able to turn logging on and off.

In addition to teaching how to use the logbook, this article explains how it works. The logbook makes elegant, simple use of many COM+ features such as event system, just-in-time activation (JITA), object pooling, idle time management, automatic deactivation of objects, synchronization, and the object constructor string. The logbook is a good example of the synergy you get from using multiple services together and how they cooperate.

The COM+ logbook is a COM+ server application that implements many requirements (see the sidebar, "Logbook's Many Requirements"). It can be used in any Win32 application (such as MFC or classic COM), not just COM+, and the only requirement is that the application runs on Windows 2000.

Figure 1 shows the tracing and logging entries, in HTML format. To view the XML format, examine the same entries in Logbook.xml; download the code from the *VCDJ* Web site (see the Go Online box for details).

The HTML log file is already well formatted and can be viewed by users as is. The XML log file is less presentable, but of course has many potential ways of presentation when applying various XSL schemas to it (such as presenting just errors, or only entries from specific machines, and so on).

Install, Use, and Configure the Logbook

First you need to install the logbook. Download the logbook.msi and the header file ComLogBook.h, and install the MSI file (double-click on it to import it to the COM+ catalog). That's it.

To enable your application to use the logbook, include the ComLogBook.h header file in your app. ComLogBook.h defines four helper macros for logging (see Table 1). The macros can be used independently of each other and in every combination. For example, to trace a method call into the logbook, pass the method name as a parameter to the LOGMETHOD() macro:

```
void CMyClass::MyMethod()
{
  LOGMETHOD("CMyClass::MyMethod");
  //Real work starts here
}
```

I recommend using LOGMETHOD() before doing anything else in the methods. The LOGMETHOD() macro will log, along with the method name, all the required information: call time, call location (machine, module, filename, and line number), and execution context (process, thread, context, transaction, and activity). Similarly, you can use LOGEVENT() to log events and LOGERROR() to log errors (download Listing 1 from the *VCDJ* Web site; see the Go Online box for details).

You configure the various logging options directly using the Component Services Explorer. After installing the logbook, you should have a new COM+ application called Logbook with three components: the HTML logger, the XML logger, and an event class (see Figure 2).

The main mechanism behind the logs is COM+ LCE (loosely coupled event). The macros publish the data as COM+ events, and the logbook components are persistent subscribers. Each logbook component has four persistent subscriptions in its Subscription folder: Errors Only, Methods Only, Events Only, and Log All.

Enabling or disabling a subscription allows you to control what is being logged and in what format. By default, both the HTML and the XML Log All subscriptions are enabled after installation. If, for example, you want to enable only HTML logging of events and errors, follow these steps: Go to the XML components, select the Log All subscription, display its properties (by right-clicking on it), go to the Options tab, and disable the subscription. Similarly, disable the HTML component's Log All subscription. Next, enable the HTML component's Errors Only and Events Only subscriptions. The HTML and XML components log to C:\Temp\Logbook.htm and C:\Temp\Logbook.xml, respectively, by default.

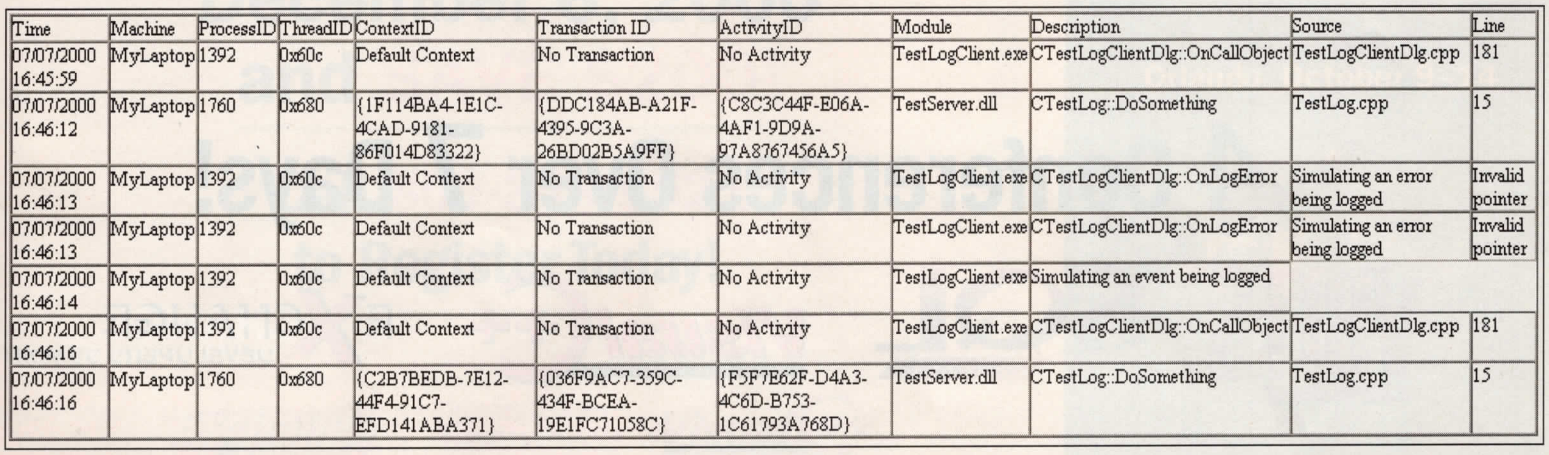
The filenames are provided as constructor strings to the components. To specify a different filename (for example, D:\MyLog.htm for the HTML component), display the HTML component properties and select the Activation tab (see Figure 3). Under Object Constructor, specify the new filename.

One interesting aspect of the logbook is that because it uses persistent subscriptions, its lifeline is independent from the applications using it. As a result, logs from many application runs are concatenated in the same file. If you want the logbook to start a new log file, you must manually shut down the logbook application (right-click on it in the COM+ explorer and select Shut Down). The next time an application publishes to the logbook, the logbook will clear the file and start fresh. You can shut down and restart the logbook even when the logging app is running.

MACRO NAME	DESCRIPTION
LOGMETHOD()	Traces a method call into the logbook.
LOGERROR()	Logs an error into the logbook.
LOGEVENT()	Logs an event into the logbook.
LOGERROR_AND_RETURN()	Logs an error into the logbook and returns in case of an error, or continues to run when no error has occurred.

Table 1 | Logging Macros Defined in ComLogBook.h. These four macros can be used independently of one another and in every possible combination. The macros automate collecting the information and the logging operation. Download the Logbook help file for detailed information on using the macros, as well as code samples; get the code from the *VCDJ* Web site (see the Go Online box for details).

LOGGING ENTRIES IN HTML



Time	Machine	ProcessID	ThreadID	ContextID	Transaction ID	ActivityID	Module	Description	Source	Line
07/07/2000 16:45:59	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnCallObject	TestLogClientDlg.cpp	181
07/07/2000 16:46:12	MyLaptop	1760	0x680	{1F114BA4-1E1C-4CAD-9181-86F014D83322}	{DDC184AB-A21F-4395-9C3A-26BD02B5A9FF}	{C8C3C44F-E06A-4AF1-9D9A-97A8767456A5}	TestServer.dll	CTestLog::DoSomething	TestLog.cpp	15
07/07/2000 16:46:13	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnLogError	Simulating an error being logged	Invalid pointer
07/07/2000 16:46:13	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnLogError	Simulating an error being logged	Invalid pointer
07/07/2000 16:46:14	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	Simulating an event being logged		
07/07/2000 16:46:16	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnCallObject	TestLogClientDlg.cpp	181
07/07/2000 16:46:16	MyLaptop	1760	0x680	{C2B7BEDB-7E12-44F4-91C7-EFD141ABA371}	{036F9AC7-359C-434F-BCEA-19E1FC71058C}	{F5F7E62F-D4A3-4C6D-B753-1C61793A768D}	TestServer.dll	CTestLog::DoSomething	TestLog.cpp	15

Figure 1 | The HTML log file shows the tracing and logging entries are already well formatted, and users can view them as is. Each entry contains the entry number (different numbers for methods calls, events, and errors), the call, error, and event time and location, machine name, process ID, thread ID, context ID, transaction ID, activity ID, the module name (the EXE or DLL name), the method name or the error/event description, the source filename, and the line number.

You already saw that the logbook uses COM+ events to pass the information collected on the logging application side to the logbook components. The components (the HTML and XML producers) implement the ILogbook interface—a custom interface with methods corresponding to what is being logged—method call, event, or error (see Figure 2, download Listing 2). The helper macros collect the information on the application side, pack it in a LOG_ENTRY struct, create a COM+ event class that also implements ILogbook, and fire the appropriate event. The logbook then receives the event, formats it appropriately (to HTML or XML), and writes it to the file.

Deciding to use COM+ events was, in a way, the easy part of the design. I faced a few other challenges. How do I channel all the events to the same logbook component? How do I collect all the tracing information I was interested in? To solve the first problem, I used COM+ instance management services. The components in the logbook application are configured to use object pooling and JITA to create the COM+ equivalent of a singleton.

Each component (HTML and XML) implements the IObjectControl interface and returns TRUE from IObjectControl::CanBePooled. The object pool is configured to have a minimum and maximum pool size of 1, which ensures there is always exactly one instance of the component of that type (see Figure 3).

INSTALLING THE COM+ LOGBOOK APPLICATION

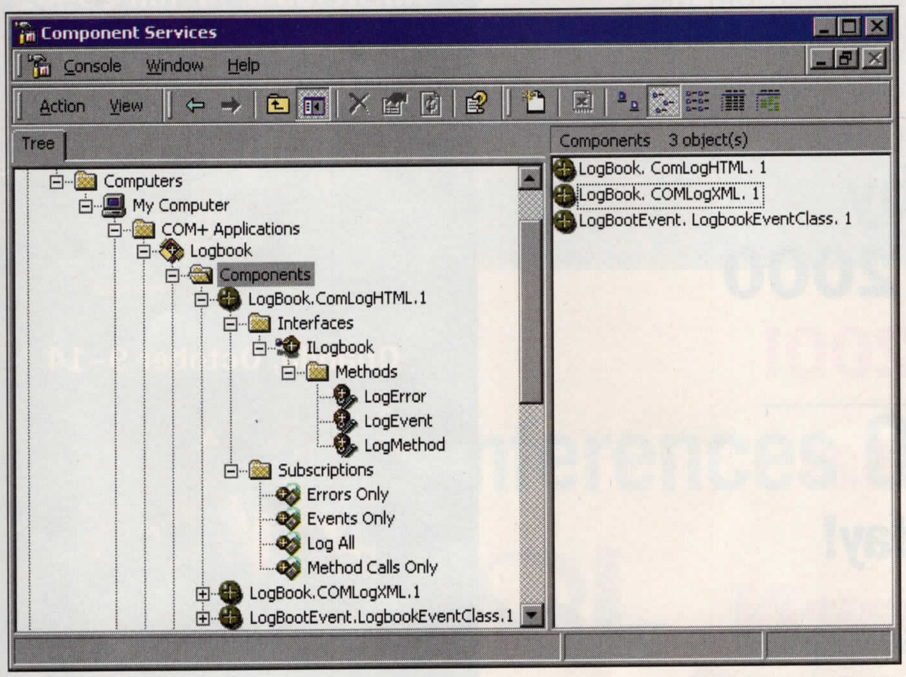


Figure 2 | After installing the logbook, you should have a new COM+ application called Logbook with three components: the HTML logger, the XML logger, and an event class. All three components implement the ILogBook interface with the methods LogError(), LogEvent(), and LogMethod(). The HTML and XML components have four subscriptions—one for each ILogBook method and one for all the methods on the interface.

When a logging client application publishes an event to the logbook, COM+ retrieves the logbook component from the pool, hands the event over to it, and, once the method of ILogBook returns, releases it back to the pool. But what would happen if a greedy application creates the logbook component directly and holds on to it? The maximum pool size is 1, so COM+ can't create another instance of the logbook component to publish the event to it; instead, it waits for the existing object to return to the pool, which doesn't happen because an existing application holds a reference to it. As a result, all attempts from other applications to publish to the logbook fail after the timeout specified in the Creation Timeout field (see Figure 3).

The solution: Use JITA. If the logbook component indicates to COM+ that it is willing to be deactivated (destroyed) and is configured to use JITA, COM+ will release the components (in this case back to the pool instead of a real release). The greedy app won't know the difference because it still has a reference to a valid proxy, which points to the logbook component interceptor. The next time the greedy client application tries to access the logbook, COM+ will detect it, retrieve the object from the pool, and hook it up with the interceptor; the greedy application's call will go through.

The logbook components ultimately are configured to use JITA. The thing is, a logbook component must still let COM+ know when to deactivate it. The logical place is at method boundaries, when it is done logging to the file. For that, the traditional way (a la Microsoft Transaction Server) was to get a hold of the object's context interface (IObjectContext), and call SetComplete().

COM+ provides a new interface to do the same (IContextState::SetDeactivateOnReturn()), but there is even a better way, one that doesn't require writing a single line of code: COM+ method auto deactivate. If you browse down to the method level in the logbook application and display the method properties on the General tab (see Figure 4), you can configure your method to deactivate the object automatically on return. When the method returns, COM+ will call IContextState::SetDeactivateOnReturn() on your behalf. Cool, isn't it?

The logbook components use the Both threading model. Synchronization is provided by having the component configured as Synchronization = Required (component properties, Concurrency tab). Note that JITA requires synchronization, so the only synchronization settings enabled are Required and Requires New.

One other configuration setting I used was to have COM+ leave the logbook app running when idle (Logbook application properties, Advanced tab). This keeps the pool alive even when no external clients are logging. As a result, all the logging is done to the same file, which also improves performance because it means you create a new application and object only once.

You already saw that the filename is passed as a constractor string. To access that string, the logbook components implement IObjectConstruct. COM+ queries for

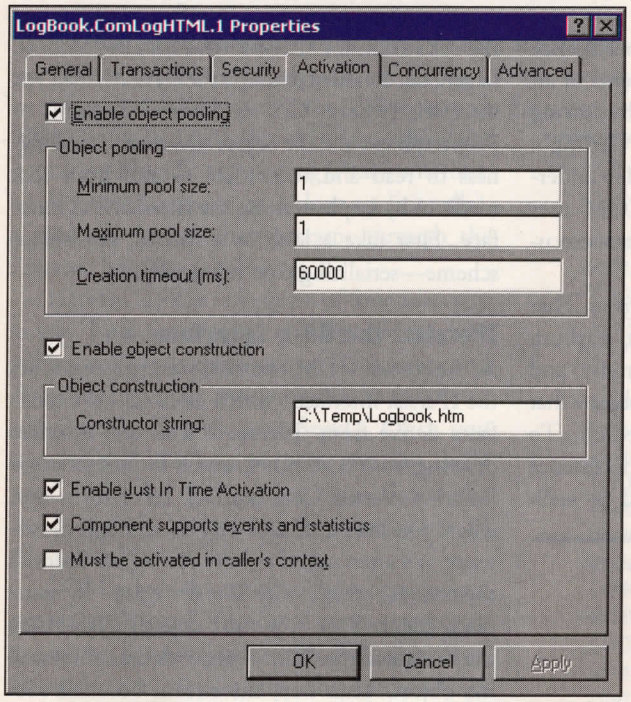
IObjectConstruct after creating the object, and passes to IObjectConstruct::Construct() a pointer to an IObjectConstruct String object. You can use that pointer to get the constructor string, in this case a filename (download Listing 3).

The other major challenge is collecting the information on the client side. Some of it, such as the line number, file, and module name, have nothing to do with COM+ and are simply neat programming tricks. If you're curious, look at the source files. But the execution context IDs are a different story. Fortunately, COM+ has an excellent infrastructure for retrieving the context, transaction, and activity ID—the IObjectContextInfo interface.

You can use IObjectContextInfo to retrieve the context, transaction, and activity ID. This is exactly what the helper macros shown in Table 1 do on the client side. Download Listing 4 for an example of how to get the current transaction ID. The macros use a helper class—CEventLogger—to collect the information and publish it to the logbook (download Listing 5).

The logbook implementation takes advantage of many COM+ services, and it's a good example of how those services cooperate. The logbook uses the Component Services Explorer for configuration, the same environment you use for configuring your components.

A LOGBOOK COMPONENT ACTIVATION TAB



[✓] Enable object pooling

Object pooling

Minimum pool size: 1

Maximum pool size: 1

Creation timeout (ms): 60000

[✓] Enable object construction

Object construction

Constructor string: C:\Temp\Logbook.htm

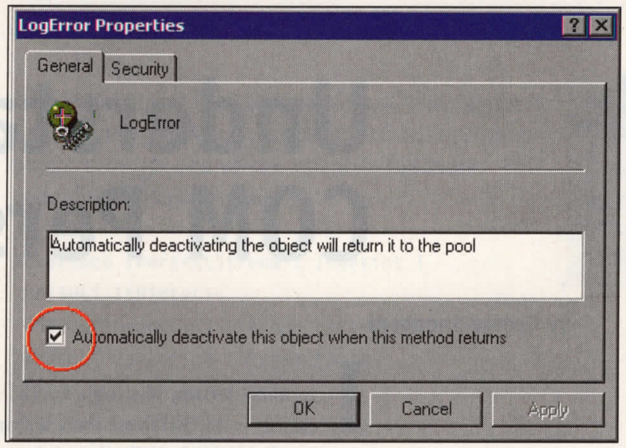
[✓] Enable Just In Time Activation

[✓] Component supports events and statistics

[ ] Must be activated in caller's context

Figure 3 | A logging component uses object pooling with exactly one object in the pool to ensure all logbook entries are serialized in order to the same file. Just-in-time activation (JITA) is used to return the object to the pool between invocations, even when an application holds a reference to it. The object constructor string is used to pass in the logbook filename.

ENABLING METHOD AUTO DEACTIVATION



Description:

Automatically deactivating the object will return it to the pool

[✓] Automatically deactivate this object when this method returns

Figure 4 | COM+ deactivates the object after each method call—in this case, receiving an event—when auto deactivation is enabled because you're using JITA. You must set this attribute on each method of ILogBook.

You can download a VC++ project that contains not only the source files of the logbook but also a demo client and server—both use the logbook—and a Windows 2000 help file for the logbook users. You can extend the logbook or improve on it (such as adding verbosity levels), but in any case, once you start enjoying the productivity boost of a flight recorder, you will find yourself asking one question: "How did I ever manage without it?" VCDJ

About the Author

Juval Lowy is a seasoned software architect. He spends his time publishing and conducting training classes and conference talks on object-oriented design and COM/COM+. He was an early adopter of COM and has unique experience in COM design. He is author of an up-and-coming book on COM+ and .NET (O'Reilly). E-mail him at idesign@componentware.net.

GO ONLINE

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

VC0011 Download all the code for this issue of *VCDJ*.

VC0011MT Download the code for this article separately. It includes all code listings, the COM+ installation file, the only header file required for a client to use the logbook, a Windows 2000 help file, and a VC++ project that contains the source files for the logbook and a demo client and server.

VC0011MT_T Read this article online. DevX Premier Club membership is required.

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