The performance of Web services applications in Windows 2000: monitoring ASP and COM+

Mark B. Friedman
Demand Technology
1020 Eighth Avenue South, Suite 6
Naples, FL USA 34102
markf@demandtech.com

Abstract.

Microsoft's proprietary ASP, COM+, and.Net software development technologies provide a powerful set of application runtime services that target the development of enterprise-class applications. When it comes to deploying ASP, COM+, and .Net applications, the Microsoft strategy is noticeably less coherent. Performance monitoring and capacity planning for the Microsoft application development platform is challenging due to significant gaps in the measurement data available and familiar problems correlating data from distributed transaction processing components running across multiple machines. This paper focuses on the current ASP and COM+ runtime environment and the problems that arise in monitoring the performance of web-based transaction processing applications that rely on these services.

Introduction.

Viewed as a whole, Microsoft's proprietary development technology is noteworthy for both its breadth and depth. Supplying both development tools and runtime services, Microsoft provides developers with an integrated software development platform for writing applications that run on top of its basic operating system services. ASP, COM+, and the .Net initiatives from Microsoft - combined with a powerful set of COM+ application runtime services - target the development of enterprise-class applications. According to one observer, the goal of the COM+ runtime services is "to simplify the development of highly concurrent systems."[1] By focusing on the productivity of software developers, Microsoft tries to make it easy to develop applications that run on its server platform.

When it comes to deploying ASP, COM+, and .Net applications and monitoring the performance of the mission-critical production systems built to exploit these services, the Microsoft strategy is noticeably less coherent. Performance monitoring and capacity planning for the Microsoft application development platform is challenging due to significant gaps in the measurement data and familiar problems correlating data from distributed transaction processing components running across multiple machines. While Microsoft and various third party vendors are working to fill these gaps, it appears likely that this environment will continue to prove challenging for the foreseeable future.

This paper focuses primarily on the current ASP and COM+ runtime environment and the problems that arise in monitoring the performance of transaction processing applications that rely on these services. It will discuss the performance monitoring limitations that are common to both environments that make it difficult to configure and tune large scale transaction processing application systems today. The current situation being far from hopeless, the paper will then describe the interfaces that are available that can be used to gather additional application runtime performance statistics. Finally, it will review promising new developments that should bring some relief to systems professionals with responsibility for maintaining adequate computer and network capacity for applications on the Microsoft platform.

Background.

Over the past several years, Microsoft Corporation has diligently pursued a strategy to develop an application development platform on top of its Windows NT operating system architecture. This application development platform

- relies on object-oriented programming paradigms,
- exploits a variety of industry-standard, open protocols like HTTP, XML and SOAP, and
- is wrapped into a proprietary framework called COM for building reusable code components.

COM, which stands for the Common Object Model, is an object-oriented programming (OOP) environment. COM is also the centerpiece of a broader application development framework designed to support large scale, enterprise applications. As COM has morphed into COM+ and now .Net, Microsoft has made available extensive COM component runtime services, including an SQL database management system (MS SQL Server) and a transaction monitor called Microsoft Transaction Server (mts) which can be readily accessed by COM programs.

Originating on Intel servers with inherently limited scalability, the Microsoft strategy embraces distributed processing. To enhance the scalability of its platform using a distributed processing model, Microsoft provides an inter-process or inter-computer message queuing service called MSMQ and three forms of clustering technology:

- Network Load Balancing (NLB), which operates a cluster of up to 32 front-end machines that share a single, virtual IP address,
- Component Load Balancing (CLB), which operates exclusively on COM+ middleware programs, and
- Microsoft Cluster Server (MCS), aka Wolfpack, which provides a high availability (HA), fault tolerant cluster involving two, similarly configured machines.

In addition, there is a broad range of Independent Software Vendors (ISVs) that provide a variety of pre-packaged COM components that support the Microsoft development platform. Together, these products assist a thriving community of application developers that work with Microsoft's proprietary application development technology.

Microsoft also builds a complete set of application development tools, including compilers, editors, debuggers, and a code performance profiler. In a break with a long tradition of industry practice to promote interoperability at the computer programming language level, Microsoft has chosen to build application development tools that specifically support its proprietary application development platform. Microsoft's latest extensions to the standard C++ programming language are designed to support COM+ program development. These proprietary language extensions are known as C# (pronounced "C Sharp").

Windows NT application servers. The client side of a COM+ transaction processing application normally uses the User Interface services associated with the Internet; namely HTTP running on top of TCP/IP. Of course, the original focus of the Microsoft Windows operating system was its Graphical User Interface (GUI), modeled on research using high resolution graphics, and mouse-manipulated menus that was performed at Xerox PARC circa 1980 that was later commercialized in the Xerox Star computer systems and derivative products at Apple Computer, including the Lisa and Macintosh computers. This is a much different lineage than the document-oriented markup languages such as SGML and the hypertext paradigm credited to Ted Nelson that the designers of HTML adopted. As many organizations are heavily invested with the Microsoft development platform at various stages in its evolution, it may be helpful to trace some of the major milestones in this line of development as Microsoft has shifted its aim over time to focus on web-enabled applications.

Early versions of Microsoft Windows grafted graphical elements onto Microsoft's DOS operating system (MS-DOS) that supported Intel-compatible 16-bit, and later 32-bit, computers. When a joint operating system development project between Microsoft and IBM that produced OS2 version 1 and 2 floundered, Microsoft initiated an ambitious effort to produce an advanced operating system on its own, which became Windows NT version 3, introduced to the public in 1992. As discussed in [2], the goals of the Windows NT development project were to build a reliable and secure computing platform on top of next generation computer hardware. The clear intent was ultimately to replace MS-DOS and early versions of Windows, which were welded on top of a DOS single-user kernel, as a foundation for future application development.

With Microsoft focused on its huge install base of desktop applications that exploited the Windows GUI interface, Windows NT had to be able to run Windows desktop apps. From an initial version of the OS that was essentially agnostic with regard to what GUI ran on top of it, Windows NT version 4 was redesigned and re-positioned as "a better Windows than Windows," as Microsoft standardized its GUI application programming interface (API) into a common specification called Win32 that was supported on all 32-bit forms of Windows, including Windows 9x and ME and Windows NT. From the beginning, Windows NT also supplied a significantly more elaborate set of application runtime services suited more to multi-user server-based applications than the traditional desktop applications associated with Microsoft Windows. The NT operating system's support for a program thread Scheduler with priority queuing and pre-emptive multitasking in version 3, followed by symmetric multiprocessing (SMP) support in version 4, made it far more suitable for enterprise class server applications than MS-DOS based versions of Windows. Following the initial delivery of its Back Office suite of server applications for Windows NT version 3.5, including MS SQL Server and MS Exchange, Microsoft succeeded in delivering an initial set of integrated runtime services capable of supporting large-scale, mission-critical, enterprise applications. Microsoft championed these internally-developed, multi-user server applications publicly to showcase the full range of its application runtime services [3].

Web services. In a well-publicized sudden change in direction [4], Microsoft maneuvered in the late 1990s to absorb Web-based technology and the services associated with the burgeoning Internet into its primary application development platform. Microsoft moved quickly on a number of fronts to incorporate Internet technology. These included providing support for the Internet communication protocols associated with TCP/IP as an alternative to the LAN-oriented communications protocols like IPX, developed by Novell, and NetBios that prevailed in the early days of MS-DOS and Windows. This transformation was so complete that by the time Windows 2000 was released that TCP/IP became the default networking protocol used by Microsoft-based clients [5].

In its effort to support TCP/IP networking protocols, Microsoft was able to move as rapidly as it did because it was able to adapt proven, open source software to the proprietary Windows NT environment. The availability of high quality, open source software to service the Internet application protocols, including FTP, SMTP, and HTTP code, assisted the effort to add Internet Mail support to Exchange, for example, and deliver a fully functional Web server called IIS (Internet Information Server). Ultimately, Microsoft bundled IIS into Server versions of every copy of its Windows NT (and 2000 and XP) operating system.

Meanwhile, Microsoft made a determined effort to stake out a leadership position with its Internet Explorer (IE) web browser. Exploiting Internet connectivity, browser software developed rapidly into a universal telecommunications client application, both widely available and capable of accessing the unprecedented range of "content" available on the public access World Wide Web and private "intranets." Microsoft entered the market for web browsers by licensing the source code for Mosaic, one of the more popular browsers of its day, which the company then integrated into Windows 9x and Windows 2000. Microsoft's decision to "bundle" IE into its operating systems was later challenged by its competitors, leading the US Department of Justice to prosecute Microsoft for violating the country's antitrust laws. (The court-imposed remedy for these actions is still being debated as this paper is being written.)

The legality of Microsoft's actions in this arena aside, the result of this aggressive program to incorporate Web technology into its operating systems was that Windows 2000 when it first became Generally Available was a model TCP/IP networking client, while Microsoft's bundled Internet Explorer browser program and IIS web server application were also both widely deployed. The extent to which the technology associated with the Internet is pervasive across every major computing platform today makes Microsoft's aggressive decision to bundle basic web services into its operating system software look infallibly prescient today. It was hardly a sure thing when Microsoft began the effort.

While Microsoft's actions to make IE the dominant browser program Windows-based clients use to access the Internet today have been widely analyzed and discussed, Microsoft's efforts to extend its development tools to support Internet technology are often ignored. Microsoft's initial stance suggested using web-based user interfaces only as a "thin client" alternative to the more functional Win32 GUI applications typified by Microsoft's own Office productivity apps. While single user desktop applications that rely on the Win32 GUI services, like MS Word and Excel, continue to play a key role in Microsoft's application portfolio, Microsoft's adoption of web technology is so complete in its current .Net platform that web browser-based application development is considered the norm, rather than the exception. Recognizing that the Internet protocols are capable of interconnecting client-server applications with a virtually unlimited range of client devices, including full-featured desktop computers, laptops, handheld devices like phones and PDAs, and even intelligent home and industrial appliances, Microsoft's focus in .Net is on building tools to develop and deliver web services applications almost exclusively. To appreciate the full scope of Microsoft's achievements in this area, it will help to consider the evolution of the Web server environment from delivery of primarily static information contained in read-only files to web services that perform dynamic information exchange.

Dynamic web content. The shift from the display of web content contained in static files to dynamic web content generated programmatically is still underway. Initially, web content was limited to delivering static pages in standard HTML (HyperText Markup Language) format. Functionally, the web browser sends an HTTP (hypertext transfer protocol) Get Request referencing a specific html file (or the site-defined default one) identified by a URL (Uniform Resource Locator) to a web server application (e.g., Apache, Netscape, Websphere, or IIS), which returns the appropriate file data using standard TCP/IP session-oriented connections. HTML codes inserted into the file contain formatting instructions for the browser so that the data returned can be displayed properly. The hypertext aspects of HTML permit one file to reference other files designed to be embedded in the display (gif or jpeg format graphics, for example), which are known as *inline files*. In this situation, several Get Requests from the client to the server are required before the full display can be con-

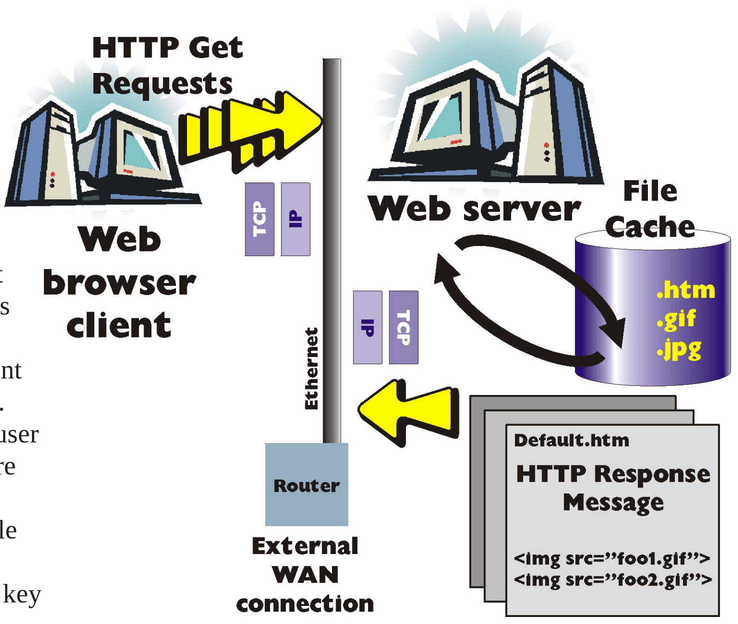


FIGURE 1. WEB SERVER PROCESSING OF STATIC HTML REQUESTS.

structed. See Figure 1. Embedded hypertext links also support transfer of control to related web pages upon user command.

From the standpoint of the web server, individual Get Requests are processed independently – it is both a *connectionless* and *sessionless* protocol. However, HTTP does sit in the protocol stack on top of a TCP-oriented session that is responsible for maintaining a unique connection between client and server. A unique TCP port number is assigned to each sessionless (i.e., stateless, no client state information is retained between successive interactions) connection, which apart from periodic TCP Keep Alive messages, can remain idle for minutes, hours, and even days with no other consequences.

While these basic building blocks of HTML-based web content are adequate for constructing useful and attractive web sites, static HTML technology is not powerful enough to build interactive web content that can be shaped by user input. The first step taken by the Internet community to augment static HTML with programmable capabilities to alter content dynamically was the ability of the web server to launch a program or script in response to user input using the Common Gateway Interface (CGI). With CGI scripts capable of processing forms, a rudimentary transaction processing capability became a reality using web browser clients to initiate programs executed on the web server, which then fashions an appropriate reply.

The HTML specification also provides a simple mechanism which CGI scripts can use to store and retrieve state information about a client session on the client side of the connection. A server, when returning an HTTP object to a client (the browser), may also send a piece of state information that the client will retain locally, usually by hardening it on disk. Included in that state object is a description of the range of URLs for which that state is valid. Any future HTTP requests made by the client which fall in that range will include a transmittal of the current value of the state object from the client back to the server. For no especially compelling reason, this state object is called a *cookie*. The ability to maintain persistent state information about the status of a client session extends the capabilities of Web-based client/server applications into the realm of transaction processing systems.

Rudimentary forms processing using a combination of CGI scripts and HTML cookies inevitably generated demand for even more Web programming capability to harness the potential power of this pervasive computing platform. Microsoft, in particular, has been active in developing several non-standard and proprietary extensions to the basic functions available in the original HTTP specification. These proprietary extensions are designed to improve web site programming using Microsoft's own development tools. Below, we will review the functional characteristics of the most important of these initiatives, including ISAPI, ASP, and COM+.

Microsoft's web application programming extensions.

ISAPI. In addition to also supporting standard CGI scripts, Microsoft's IIS web server supports a proprietary ISAPI (Internet Server Application Programming Interface) interface that allows processing of HTTP Get and Post Requests using Win32 programs packaged as DLLs (dynamic linked libraries). Because ISAPI extension DLLs are executables built by a compiler, they must be written in a programming language like C++. Unlike CGI requests which require a separate process to execute, ISAPI DLLs can be multithreaded. Depending on the level of *application protection* chosen for running ISAPI DLLs in IIS version 5.0, an ISAPI extension can be executed

- using a thread within the IIS **inetinfo.exe** address space thread pool,
- on a thread from a thread pool inside a container process known as **dllhost.exe**, or,
- on a thread in a separate process (which also happens to be an instance of **dllhost.exe**) dedicated to that execution instance (similar to CGI scripts).

The lowest level protection provides the highest level performance because the ISAPI extension DLL is dispatched on an existing thread from inetinfo's internal worker thread pool. However, this exposes the entire web site to problems if the ISAPI extension DLL misbehaves. Using a medium level of protection, IIS transfers control to a thread from a pool inside the **dllhost.exe** container process. At the highest level of protection available, each ISAPI DLL is executed in a separate, dedicated instance of **dllhost.exe**. This thread dispatching scheme is summarized in Table 1. **Inetinfo** invokes a COM Object called the Web Application Manager, or WAM (**wam.dll**), to provide a consistent linkage mechanism between IIS and the ISAPI extension DLL, where ever it happens to be loaded.

Besides the obvious relationship between the application protection level and web site reliability, the choice of where to run ISAPI extension applications also has distinct performance implications. At the highest protection level,

Application Protection Level	Thread execution context
Low	inetinfo.exe shared thread pool
Medium (Pooled)	dllhost.exe shared thread pool
High	dedicated dllhost.exe thread pool

TABLE 1. ISAPI APPLICATION PROTECTION LEVELS IN IIS 5.0.

ISAPI applications are run in dedicated processes reminiscent of the way CGI scripts are executed. IIS does provide a configuration parameter to allow the **dllhost** processes created to persist for some amount of time following execution of the ISAPI DLL so that not every new transaction provokes process creation and destruction. Using the medium protection level, a rogue ISAPI DLL cannot bring down IIS completely, but it certainly can damage any other applications that are sharing the **dllhost.exe** container process. Assuming process creation and destruction is not too big a factor, the CPU overhead considerations for a single shared **dllhost** container process are roughly comparable to running dedicated dllhost.exe processes, assuming there are sufficient process-level worker threads available to handle the workload in the case of a single instance of **dllhost**. In either case, WAM must perform an out-of-process call from **inetinfo** to **dllhos**t and back to process the request. These *out-of-process* calls are made using very expensive COM+ runtime plumbing from a WAM instance inside **inetinfo** to another WAM object inside **dllhost.exe**, as illustrated in Figure 2.[1]

Resource accounting is problematic when ISAPI extension applications execute in their default medium (or pooled) protection mode or use the lowest protection level. When ISAPI DLLs are all executing inside the same container process, it is very difficult to figure out which application is responsible for consuming which computer resources. While process level resource usage statistics are available for both CPU and Memory consumption, it is not possible to apportion that usage data among multiple applications running inside the same process accurately. Thread CPU usage data is also available, but it is not any more useful. At any point in time, any worker thread in the thread pool can be processing any application request, so it is difficult to know what to do with this data, too. Because thread pooling is used inside **dllhost.exe** (and, for that matter, inside **inetinfo.exe**, too, when the lowest protection level is specified), Thread performance Counters cannot be relied upon to identify any single application DLL for very long.[2] For an appreciation of the difficulties involved, see [6] for a procedure to determine which ISAPI application program is responsible for a runaway thread inside either **inetinfo** or **dllhost.exe**. This debugging procedure, which requires freezing the **dllhost.exe** process using WinDebug is clearly not suitable for trouble-shooting most high volume, production web sites.

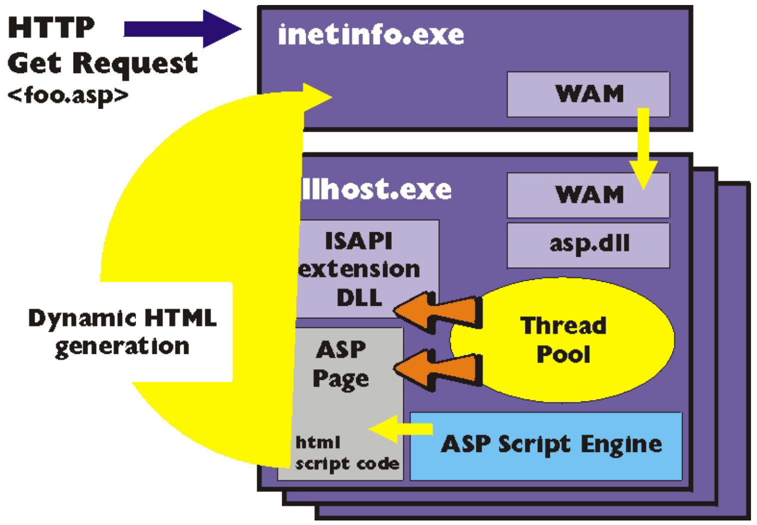


FIGURE 2. THE ARCHITECTURE OF ISAPI EXTENSION DLLS WHEN MEDIUM (OR HIGH) APPLICATION PROTECTION IS SET.

While running ISAPI extension DLLs in isolation solves one aspect of the resource accounting problem, it leaves another significant difficulty unresolved. With multiple copies of **dllhost.exe** often active, it is still almost impossible to correlate process level resource consumption statistics with individual applications, except via the debugging procedure referenced above. The process and thread level performance statistics that Windows NT/2000 provides do not identify which active application DLL is running inside each specific instance of the **dllhost** container process. One approach is to process the transaction-oriented data that is written to the web log and try to correlate that with the process-level statistics that the performance monitoring interface does provide. We will review the contents of the IIS extended format web logs in a moment. Another approach extends the standard performance monitoring instrumentation so that the active application DLLs running inside container processes can be identified [7].

Several tuning parameters are available to adjust the size of the IIS thread pool where ISAPI extentsion DLLs are executed. These include **MaxPoolThreads**, which determines the number of threads per processor in the **inetinfo** thread pool, and P**oolThreadLimit**, which sets an upper bound on the number of threads in the **inetinfo** process address space. Both these values are added to the Registry at the HKLM\System\CurrentControlSet\Services\InetInfo\Parameters key. The level of application protection influences the way **MaxPoolThreads** works. At the highest level of protection, each **dllhost.exe** container process allocates at most **MaxPoolThreads** per processor. At lower levels of protection, **MaxPoolThreads** applies to the single thread pool that all ISAPI extension applications share.

[1] The **mtx.exe** container process is used instead in IIS version 4.0.

[2] The use of I/O Completion Ports by the IIS-managed thread pool almost ensures that a dispatched thread cannot be expected to service any one application continuously. I/O Completion Ports are a mechanism to re-dispatch a worker thread that would otherwise block because it is performing I/O. A thread pooling application that utilizes an I/O Completion Port tends to make efficient use of its worker threads by maintaining a minimum number of active threads.

ASP. In an effort to make web application programming easier than the low level ISAPI interface, Microsoft subsequently introduced Active Server Pages (ASP), another proprietary web application programming technology that only Microsoft platforms support.[3] Active Server Pages contain a mixture of HTML codes and script code that is executed by an IIS facility during runtime. IIS supports scripts written in either VBscript or Javascript, or in PERL, Python, TCL, and REXX, assuming you have the appropriate language script interpreter program installed. The HTML code contained in an ASP page normally serves as a template that the script shapes dynamically, based on the current context, to render the final HTML response message that a web browser program client can understand and display. The following simple example illustrates the basic capability of ASP scripting to generate HTML on the fly:

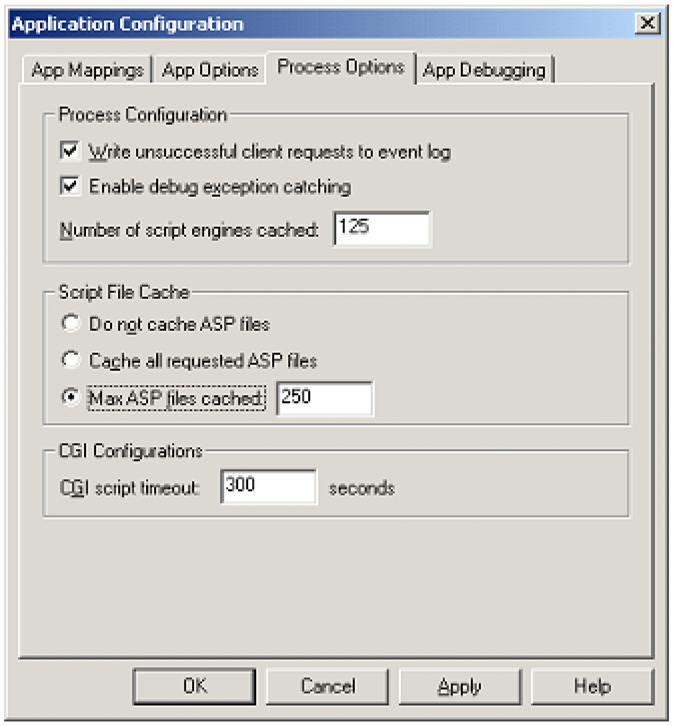
```
<% If Hour(Now) < 12 Then %>
   <FONT COLOR=YELLOW>Good Morning!</FONT>
<% ElseIf Hour(Now) < 18 Then %>
   <FONT COLOR=LIME>Good Afternoon!</FONT>
<% Else %>
   <FONT COLOR=ORANGE>Good Evening!</FONT>
<% End If %>
```

Here the script generates an appropriate Hello message based on the current time of day. Note: because the output of ASP scripts is generated dynamically, IIS sets HTTP cache-control to prevent browsers from caching HTML output generated by Active Server Pages scripts because there is no way to guarantee that an ASP page will look the same the next time it is requested. Since the HTML output created by ASP scripts cannot be cached locally by either the client browser or by intermediate cache engines and proxy servers, the performance of ASP applications depends almost exclusively on server capacity, along with the usual network performance factors that predominate when Internet protocols are used [8].

Microsoft implemented the Active Server Pages feature as an ISAPI extension DLL. Consequently, IIS processing for Get and Post Requests for ASP files (which are identified by an **asp** filename suffix) is quite similar to other ISAPI extensions. The ISAPI interface passes all ASP requests to **asp.dll**, which is also responsible for loading the appropriate script engine. Depending on the application protection level chosen, **asp.dll** is loaded *in-process* inside **inetinfo** or *out-of-process* inside **dllhost.exe**. See Figure 2 for a picture of this web application architecture, which illustrates the medium or pooled level of protection to execute ASP scripts using threads from a pool inside the **dllhost.exe** container process. Figure 2 also illustrates the use of **wam.dll** running inside both **inetinfo.exe** and **dllhost.exe** to link to ASP application scripts regardless of which process is hosting them. The presence of **wam.dll**, which is a COM component (discussed below), inside **dllhost.exe** can be used to associate a specific instance of the **dllhost** process with ASP application processing.

The first time an ASP script is executed, the script code must be interpreted by the Script engine. Compiled script code is saved and stored in a memory-resident cache called the Script Engine Cache, which saves time if the application is re-executed again soon. The size of the Script Engine Cache is limited by the amount of RAM installed and a tuning parameter that sets an upper limit on the number of compiled scripts that can be retained at any one time. See Figure 3. A performance Counter in the ASP Object called **Script Engines Cached** reports the current number of files in the Script Engine Cache. There is also a smaller Template Cache which is used to cache handles (pointers) to the compiled scripts themselves. Hit ratio statistics for the Template Cache are provided, but the Template Cache, which manages handles, is not a large consumer of RAM. Somewhat inexplicably, the more important (and considerably larger) Script Engine Cache is not instrumented. The best way to ensure that the Script Engine cache is effective is to inventory the number of ASP scripts that are being executed (this can be tabulated from the web log) and verify that the **Script Engines Cached** counter equals this number.

Since Active Server Pages were designed to make it easy to build transaction-processing applications, they are session-oriented. Each client that initiates an ASP page is



[✓] Write unsuccessful client requests to event log
[✓] Enable debug exception catching
Number of script engines cached: 125

○ Do not cache ASP files
○ Cache all requested ASP files
● Max ASP files cached: 250

CGI script timeout: 300 seconds

FIGURE 3. IIS 5.0 SETTINGS THAT CONTROL ASP COMPILED SCRIPT CACHING.

[3] A product called Chili!Soft ASP can host ASP pages and components on a variety of non-Microsoft Web servers without any changes to application script code.

a assigned a Session ID, which is stored by the client as an HTML cookie. The programmer can write an ASP event handler in **global.asa** to process the Session **OnStart** event that is triggered the first time a user runs any page in the site. The **OnStart** event is often used to establish a value for **TimeOut** property of the user Session and to store any additional identification data needed in the Session ID cookie. Utilizing the Session object to store session state creates special considerations that impact network load balancing schemes such as round-robin DNS that allow a series of IIS web servers to appear as a single virtual IP address. The proper technique for preserving Session state that works with network load balancing is known as *ASP session-aware load balancing*. This involves invoking the load balancing scheme to distribute the initial application request only and redirecting all subsequent requests to the computer assigned to that client session. The Network Load Balancing feature of Windows 2000 Advanced Server, which allows you to create an IIS server cluster containing up to 32 machines, provides an option to specify that all connections from the same client IP address be handled by a particular server to preserve Session state.

Because the internal architecture of ASP script processing is identical to ISAPI extension DLLs, capacity planners face the same difficulties trying to associate applications that use Active Server Pages technology with the computer resources they consume. With multiple copies of **dllhost.exe** often active, correlating process level resource consumption statistics with individual applications remains problematic. A combination of running ASP applications in isolated processes and analyzing the transaction-oriented data that is written to the web log is the only viable option currently available.

COM and MTS. Active Server Pages technology, where scripts and HTML coding can be embedded in the same file, is quite handy for crafting dynamic web pages. But developing scripts has its limitations when it comes to programming complex applications. Of particular concern is the fact that scripting languages do not support the type of modularization that developing complex applications usually requires. In the Microsoft framework, ASP provides the presentation layer for what Microsoft describes as a three-tiered approach to web application development. The second layer is for business logic, which Microsoft suggests should be performed using COM modules and MTS runtime services. (In Windows 2000 COM and MTS are effectively merged into a single runtime service called COM+.) A third and final layer is for the back-end database processing where all persistent state information about active transactions is usually stored. We will not attempt to discuss back-end database performance topics here; the interested Reader should refer to [9]. See Figure 4 for an illustration that shows how applications using this framework can be clustered for both high availability and performance. In this illustration, middle tier business logic is distributed across a series of server machines that are executing COM+ components. Component Load Balancing, a COM+ application clustering feature included in Application Center 2000, handles routing to the middle tier. Before we consider some of the performance implications of COM+, we need to understand what it is and what it does.

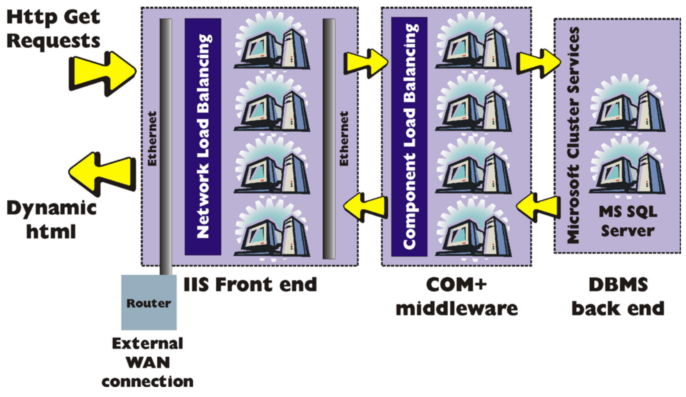


FIGURE 4. THE MICROSOFT FRAMEWORK FOR DEVELOPING SCALABLE, 3-TIERED WEB SERVICES APPLICATIONS.

Originally, COM was the runtime infrastructure associated with ActiveX components, a development technology introduced at the same time that Active Server Pages scripting was released.[4] It was designed initially as an object-oriented facility to supercede OLE (Object Linking and Embedding) as a way for one application in one process to call another application in a different process. (Think MS Word calling Excel to edit an Excel-generated chart embedded in a Word document.) As an interprocess communication (IPC) mechanism, the most salient feature of COM, compared to more conventional methods like RPC and the shared DLLs that were already in widespread use on the Microsoft platform, was its support for versioning. COM programs, which are also packaged as DLLs, support a required interface called **IUnknown** that allows the caller to discover the Methods and Properties of the program being called at runtime. The calling program can also use the **IUnknown** interface to verify the version of the module being called before transferring control to it.

A complete discussion of COM+ programming is well beyond the scope of the present paper. Readers interested in pursuing this topic can refer to [10] and any number of other good books on this subject. The discussion here will be limited to the use of COM+ as the middle tier of a

[4] With this momentary penchant for calling all its new features "Active" this or that (e.g., *Active Directory*), it seems likely that someone in the Microsoft Marketing department responsible for product naming was once brutalized by a high school English teacher for using passive voice.

transaction processing runtime environment, supporting concurrent execution of component programs. Enhancing the scalability of the Microsoft web services platform, modularized COM+ components can be executed in an extraordinary variety of configurations, either within the same computer or on a remote computer using the DCOM (Distributed COM) protocol. For example, when requested by a calling program, a COM+ program can execute in several different runtime environments, including

- *in process* in the same thread as the calling program,
- *in process* in a new thread separate from the calling program,
- in a new thread in a different process (*out of process*) in the same computer, and
- in a thread in a process running on a remote computer.

To a remarkable degree, a COM+ program can be written without regard to how the application is actually deployed across this range of computing environments. For example, whether a COM+ program executes in-process as a *library application* or in a separate container process (the ubiquitous **dllhost.exe**) as a *server application* is a decision made when the compiled runtime component is installed. When the COM+ component is installed, the System Administrator sets its *activation* property, which determines whether the program runs on that machine as a library application or a server application. For the most part, the program itself can be developed independently of this deployment decision.

Naturally, how the COM+ runtime infrastructure services a request that *activates* a component is important from a performance perspective. If the module requested is installed as a library application, COM+ merely transfers control in-line so that the component executes on the caller's thread. Obviously, this is the most efficient linkage. However, if the caller's thread is not set up to provide all the COM+ services the requested module requires, COM+ will transfer control instead to a new thread from the same process thread pool. If the module requested is installed as a server application, COM+ will transfer control from a thread in one process to a thread in another process, with the COM+ runtime being responsible for constructing a new **dllhost.exe** container process, if necessary. If the requested component is only available remotely – again, a configuration decision to isolate the programs incorporating the business logic from the presentation level, front end scripts – the dynamic module linkage is automatically resolved using the DCOM (Distributed COM) protocol, a close cousin of RPC.

Of course, there is a considerable difference in the overhead associated with these three variations. A capacity planning white paper published by Microsoft [10] references measurements taken comparing performance in all three environments: in-process calls, out-of-process calls in the same box, and calls to a remote computer, as follows:

	Calls per second	Relative speed
COM+ server application, run over a 10BaseT network	625	1
COM+ server application, out-of-process, same machine	1923	3.08
COM+ server application, in-process, same machine	3333	5.33

Table 2. The relative speed of COM+ calls.

As Table 2 illustrates, distributed processing explicitly trades off application response time, which elongates as a result, against overall throughput, always a tricky decision to make. Quite obviously, the overhead associated with a DCOM call to a component executing on a remote machine represents a serious performance penalty. However, if a single machine cannot sustain the transaction throughput required by the application, there may be little alternative in the Microsoft environment other than distributing the work across multiple machines. The distributed processing architecture depicted in Figure 6 may be the only feasible way to handle the transaction volume the application's user population generates.

A COM+ component is further characterized by a number runtime attributes, including its threading model, serialization requirements, concurrency level, and transactional support. These are all program attributes which are set declaratively so that the program itself can be written and executed (largely) independent of the implementation considerations. From this perspective, COM+ is essentially a runtime service that allows one program to call another independent of the calling program needing to establish the proper runtime environment for the called function to execute. Instead, COM+ runtime services provide the linkage from caller to callee, ensuring that the component requested executes in a context that offers all the services it requires.

Similar to other transaction monitors like Tuxedo and CICS, COM+ is also designed to simplify the development of scalable transaction processing applications by masking the inherent difficulties in building multithreaded application programs that execute correctly in that complex environment. COM+ applications can be written as if they are running single-threaded, but are executed in a runtime environment that supports multithreaded, concurrent processing. To accomplish this, the COM+ runtime has several noteworthy features, including multithreading, object pooling, and built-in transactional support. Next, we will consider each of these features briefly and their impact on the performance of web services applications.

Threading. In theory, COM+ is an attempt to let developers ignore serialization considerations in building scalable, multithreaded applications. In practice, serialization is too implementation-dependent to jump from the particular to the general in every case. But Microsoft's achievements in this most difficult area of application development technology should still be applauded. COM+

runtime services manage both an application's threading behavior and its serialization requirements, which are both established by the programmer declaratively.

COM+ programming supports the full range of threading options available to the Win32 application programmer. As discussed earlier, the COM+ routine can be loaded directly in the calling process and can execute in either the caller's thread or a separate thread. COM+ Objects can also be executed out-of-process, which means that they are loaded into a separate container process, similar to ISAPI extension DLLs. Each calling thread can acquire a separate copy of the COM+ module running in a dedicated single threaded *apartment*, or a single copy of the COM+ component can service multiple callers concurrently if it is running in a multithreaded apartment. Instead of invoking the serialization services of the Win32 API directly, the COM+ programmer sets the synchronization property of the application declaratively. For example, a database update program is set to run in a single threaded apartment with synchronization required, which ensures that only one database update at a time is performed.

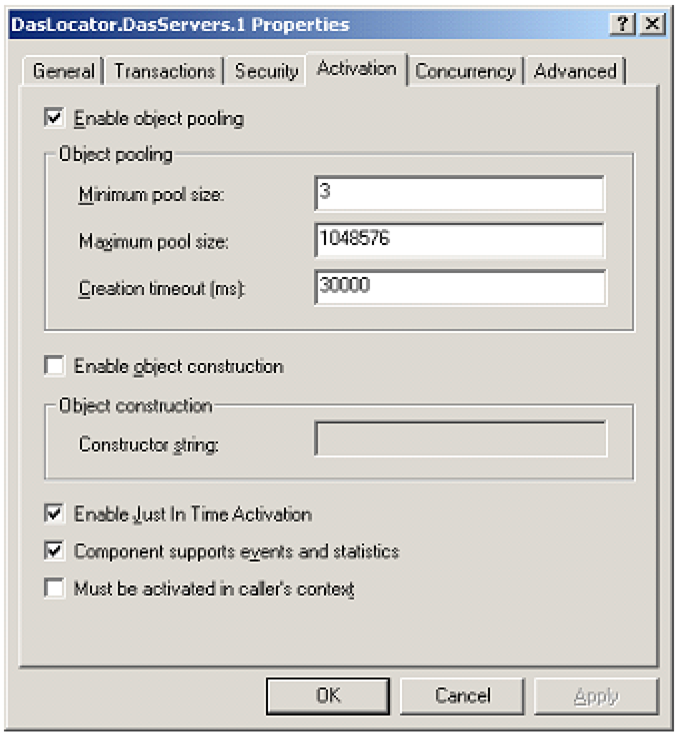
Thread pooling. At least one aspect of writing scalable, multithreaded applications can be generalized, namely, that a runtime environment that draws available resources from a common *thread pool* is an effective way to build client/server applications that scale from small machines with single processors to large machines with multiple processors. The general structure of a client/server-oriented thread pooling application is one that fields work requests and matches them to a pool of available processing threads. On larger systems that can handle more requests, one simply increases the size of the thread pool to increase application throughput. In addition to IIS and ASP, Microsoft has developed several other commercial, thread pooling applications for Windows 2000, including the network file sharing service known as Server, the SQL Server database engine, and the MS Exchange messaging and mail server.

Because thread pooling is so fundamental to multi-user server application scalability, Microsoft decided to offer developers a pre-packaged set of robust thread pooling services to incorporate into their applications. If they chose, developers can take advantage of COM+ runtime features that provide generic thread pooling, instead of painstakingly developing their own thread pooling logic. This feature of COM+ is known as *object pooling*. Using object pooling, the COM+ runtime environment is capable of managing the concurrency level of re-usable (and reentrant) multi-threaded components automatically.

A component that is enabled for object pooling establishes minimum and maximum pool limits that determine how many instances of the object can run concurrently. A third pooling parameter determines how long a request for a COM+ component that is already running at its maximum level will be queued before the request is timed out. These runtime parameters can be set by the System Administrator, assuming the application has its **ObjectPoolingEnabled** property set. See Figure 5.

In this example, the System Administrator has used the Component Services Explorer (CSE) applet to establish a lower limit on the number of instances of this component that the COM+ runtime environment will maintain in memory once the application is started. (CSE provides the administrative interface for all the COM+ program parameters that can be set at runtime.) The object pooling runtime parameters are logically equivalent to the tuning knobs that Microsoft provides for its internally-developed server applications. For example, earlier we discussed tuning parameters that can be used to override system defaults that control the behavior of IIS thread pooling. The Windows 2000 file Server and MS SQL Server DBMS also have similar thread pooling controls [12].

The reason for using object pooling is to improve performance. A pooled component is maintained by the COM+ runtime in a state that is ready to use, saving the time it takes to create a new **dllhost** container process and initialize the component inside it each time the component is requested by a caller. When a pooled object is requested, the COM+ runtime calls the object's **Activate** Method to transfer control from the Requestor program to the pooled component. Instead of terminating conventionally, the component calls **Deactivate** when it is finished processing the caller's request. To determine if a deacti-



[✓] Enable object pooling

Object pooling

Minimum pool size: 3

Maximum pool size: 1048576

Creation timeout (ms): 30000

[ ] Enable object construction

Object construction

Constructor string:

[✓] Enable Just In Time Activation

[✓] Component supports events and statistics

[ ] Must be activated in caller's context

FIGURE 5. OBJECT POOLING PARAMETERS ARE SET FOR EACH COM+ PROGRAM USING THE COMPONENT SERVICES EXPLORER (CSE) SNAP-IN.

vated component can then be returned to the pool where it can be re-used, the COM+ plumbing calls the component's **CanBePooled** Method. The component program returns **True** if it is okay to re-use the object instance. If the program returns a value of **False**, that instance of the COM+ object is unloaded and deleted. COM+ runtime services always maintain the minimum number of ready object instances specified and will manufacture a new instance of the object if a request occurs, all currently activated objects are busy, and the component is not running at its specified maximum.

Idle objects above the minimum concurrency level persist for about 5 seconds, according to [1]; there is, unfortunately, no runtime parameter to control the length of time idle objects are retained. Nor is any instrumentation provided that monitors the concurrency level of active pooled objects. However, assuming a pooled component is installed as a server application and the specific instance of the **dllhost** container process can be identified (as described in [7]), the **dllhost** process Thread Count is a valid indicator of concurrency.

Transactions. The final COM+ runtime service discussed here is transactional support. Within COM+, the term *transaction* is used to refer to a logically distinct unit of work with specific database recovery requirements.[5] If a COM+ program that is a transaction fails to complete properly, or *aborts*, the transaction support in COM+ ensures that all database updates associated with the failed transaction are backed out by any Resource Managers (typically, connections to a DBMS) that have already applied partial updates. COM+ implements the well-known *two-phase commit* process to ensure that database updates can be applied consistently even in a distributed database environment where updates to multiple Resource Managers need to be synchronized.

A COM+ component's transaction attribute is set declaratively, using the CSE, although there are alternate methods that are programmatic. Once the transaction attribute is set, then the COM+ runtime ensures that the component, when it is activated, is dispatched in a transactional context. Running inside a transaction, the component program calls the **SetComplete** or **SetAbort** Methods of the **ObjectContext** object to indicate either a successful or unsuccessful outcome. If the component issues a **SetComplete** to indicate successful completion status, the COM+ runtime communicates with the appropriate Resource Manager(s) to commit the changes that occurred within the transaction *boundary*. Alternatively, a call to **SetAbort** would signal the Resource Managers to roll back any database updates that were partially applied. The COM+ runtime effectively insulates the application programmer from having to understand many of the details of the commit/rollback logic required by various DBMSes, which can become quite complicated, especially in the case of distributed database transactions.

IIS extends COM+ transactional support to ISAPI extension DLLs, which by implication means that Active Server Pages can be transactions. Similar to COM components, an ASP application script becomes a transaction declaratively. In VBscript, for example, this is accomplished with a declaration at the beginning of the page, as follows:

```
<%@ LANGUAGE=VBSCRIPT TRANSACTION=REQUIRED %>
```

This declaration causes the COM+ runtime to create an instance of **ObjectContext** that the page's script can then reference by calling the **SetComplete** or **SetAbort** Methods. One ASP limitation is that the page (nee, script) is the boundary of the transaction – a transaction cannot span ASP pages. The transactional unit of work is always the ASP page which contains the **Transaction=Required** declaration. Building COM+ components from scratch, there is a great deal more flexibility. If a component has a **Transaction=Supported** attribute, for example, it can participate in the transaction context of the calling program. The COM+ runtime services, naturally, are responsible for implementing the transition between one COM+ component that is a not a transaction that calls another component that is. A concrete example is an ASP script that is not a transaction that calls a COM+ component program that is a transaction, which is then responsible for posting a customer order to a database. COM+ ensures that a transactional component always executes in the appropriate context.

Summary. The foregoing discussion highlights the COM+ runtime services that hide much of the complexity associated with developing and deploying scalable COM application component programs. The flexibility of the Microsoft Distributed Networking Architecture raises initial questions about which pieces of which applications should be deployed where. In addition, specific COM+ deployment decisions, e.g., object pooling, raise obvious capacity planning questions. Establishing minimum and maximum pool sizes and monitoring pooling concurrency levels are unmistakable concerns. Due to the degree that Microsoft has succeeded in hiding the details of the COM+ runtime plumbing from developers, it is difficult to formulate reliable answers to these questions today.

Monitoring the performance of COM+ components can be quite challenging, as we will discuss further below. In a

[5] The definition of a *transaction* from [13] is a unit of work defined as having the following properties: *atomic*, *consistent*, *isolated*, and *durable*, also depicted as ACID attributes. For example, the fact that transactions are atomic means that either all the work associated with a transaction is performed or none of it is. COM+'s use of the term *transaction* is consistent with this standard usage. Unfortunately, computer performance professionals often use the term *transaction* in a queuing model theoretic context as a way of denoting units of work that arrive from a client to a server for processing. This ambiguity can cause considerable confusion.

production environment, it is not unusual to see many instances of **dllhost.exe** running concurrently, each with multiple threads. To monitor and tune a complex web services environment, it is important to be able to determine which COM+ components are running where and map these applications to their resource consumption profile. We will take some preliminary steps in that direction in the next section where we discuss the current state of the performance data available for web services applications in ASP, COM+ and .Net.

Web application services performance monitoring.

Web application services using Active Server Pages scripts and COM+ components have well-understood performance monitoring requirements to support both tuning and capacity planning. The principal requirement is that measurement data be on hand that captures a client's-eye view of the human-computer interaction. In any client/server web application, the user participates in an interactive computer session broken into discrete request/response sequences. We denote each discrete request/response sequences as a *transaction*. (In this section it is necessary that we adopt a different definition of a transaction than COM+ uses.) Here, a transaction corresponds to an end user's perception of an individual request/response sequence. In the case of a typical web services application, consider one such sequence where the end user initiates a transaction by positioning the computer's mouse pointer to a button on a form marked "Submit" and clicking on it. From an end-to-end response time perspective, everything that happens next that involves the network and the computers involved in processing this client request represents processing associated with this transaction.

In any client/server transaction processing environment, we endeavor to measure the delay or *latency* involved in processing the user request on both the client and server machines, as well as assessing the network latency involved in the communication of the request and the server's reply. For the purpose of the present discussion, however, we will ignore the network latency associated with transporting a web client's request to a web server and back. This network latency is primarily a function of distance, but can be influenced by a variety of internetworking infrastructure issues that are beyond the scope of the present discussion. Here we intend to focus only on what happens to the request from the time it arrives at an IIS web server for processing until IIS returns an HTTP Response message.

Using the Microsoft web services application architecture, we have seen that the server-side processing of the client request is normally broken into a number of discrete processing components. The anatomy of a typical (and relatively simple) ASP/COM+ transaction is depicted in Figure 6. Upon pressing the Submit button, the end user initiates an HTTP Post Request referencing the **order.asp** script. IIS processes this HTTP Request, transferring control to the **dllhost** container process (via **wam**) where the **order.asp** script is executed. The script then calls a COM+ program in a separate **dllhost** container process that is responsible for updating the back-end database with information from the customer's order request. When the database updates complete, the COM+ transactional component makes a **SetComplete** Method call to commit the database update, and then returns to the script. The script then fashions an appropriate HTTP Response message, which is relayed back to IIS. Finally, IIS returns the HTTP Response message that the script generated and sends it back to the client.

We have simplified this picture of the transaction processing performed on the server considerably, leaving out the COM+ plumbing used to transfer control from the caller's thread to the called program and back again, the data base connection processing, the fact that additional **HTML**, **gif**, and **jpeg** files might be embedded in the HTML response message, the processing of the HTTP requests by lower levels of the TCP/IP protocol stack, and other salient details. Instead of making one call to a COM+ component, the order processing script might invoke several component modules. Moreover, there might well be multiple trips back and forth to the database. We have also chosen to ignore for now the possibility that the ASP script, the COM+ components, and the back-end database might all reside on different machines, causing the transaction to encounter additional networking delays.

Nevertheless, the simplified picture of a web services transaction should suffice to illustrate the need to instrument each discrete component involved in processing the transaction. Furthermore, the discrete component-level measurements themselves ultimately must be *correlated* to reflect all the processing that was performed. This correlation of independent measurements applied to discrete events and components is a major technical challenge on any of the major computer

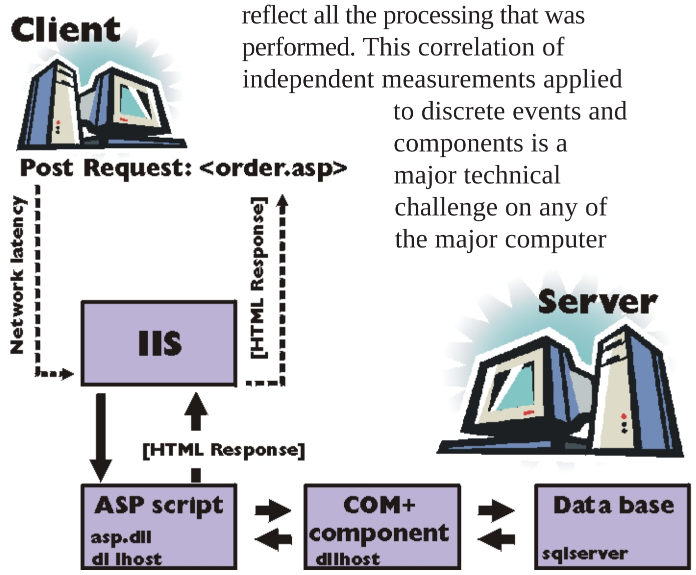


FIGURE 6. THE ANATOMY OF A WEB SERVICES APPLICATION *TRANSACTION*.

platforms [14]. The goal of this paper is to show just how big a challenge this problem poses for performance analysts responsible for web services applications built for the Microsoft platform. In the sections that follow, we will describe the measurement data that is currently available for HTTP Requests, ASP scripts, and COM+ components.

Web event logging. If an IIS application server is enabled to generate extended format web logs, it is possible to collect data on server response time for individual HTTP Requests. The **time taken** field in the log, which is *not* among the data fields enabled by default, reports internal processing time for each HTTP Request in milliseconds. Figure 7 shows sample output from the log for an HTTP Request. The date and time of the request, the IP address of the client, and the specific resource being requested are shown. At 5:11:28, a Get Request for **iccm.asp** is processed. According to the log, it took slightly more than 5 seconds to process this request. The HTTP Response message that **iccm.asp** generated evidently referenced several embedded graphic files, which accounts for a number of subsequent Get Requests from the same client. This spotlights one difficulty with this response time measurement data – it is difficult to tell *a priori* exactly which Get Requests correspond to a discrete transaction as perceived by the end user. If only the main in-line processing by the **iccm.asp** script code to create an HTTP Response message is significant, it is possible to consider just the **time taken** processing that Request. But, if you want to measure the end user experience, it makes sense to consider all the elements of the rendered web browser display. This suggests summarizing the **time taken** processing all the Get Requests associated with a single web page display. In this simple example with only one user request active, that seems like a relatively simple thing to do. In the log on a production web server that is servicing concurrent requests from multiple connections simultaneously, the processing of events from different user requests are intermixed. Under those more typical circumstances, the boundaries between end user requests are likely to blur.

A second consideration is problems interpreting the **time taken** data alongside the timestamp information in the log, which, according to Microsoft documentation, represents the time when the log record was written. If you sum up all **time taken** processing the **Get Requests** shown in Figure 8, the result is a total of 11.359 ms. That is difficult to reconcile with the timestamps of the log records reported in column 2 which show all the **Get Requests** being processed over a span of just 3 seconds.

A final concern is how the event data recorded in the IIS log corresponds to the performance data available from other sources, including the performance data the System Monitor provides. The log data should be well correlated with other web performance data since they are all derived by the same underlying measurement procedure. The HTTP method calls reported in the log, for example, should correlate well with the counters reported in the Web service Object, including **Get Requests/sec**, **Post Requests/sec**, etc. However, as of this writing, there is no published work that has analyzed the validity of this data rigorously.

Interval performance data. Using the System Monitor, or some similar tool, it is possible to report on several indicators of web server transaction load on an interval basis. For most Web servers, IP **Datagrams received/sec** and TCP **Segments received/sec** show quite similar request rates, as illustrated in Figure 8, which is an example showing TCP and IP activity for a busy commercial web site. This is because HTTP request packets are usually small enough to fit (< 1500 bytes) in a single Ethernet segment. That is something that can be verified against the **cs bytes** field in the web log, which reports length of the HTTP request.

Figure 9 illustrates the relationship between HTTP Method calls (GET, POST, etc.) and ISAPI calls, reported by the Web services performance Object, compared to the ASP **Requests/sec** Counter. In this instance, only about 1⁄10 of all IP packets received and processed by the web server represent HTTP requests. This reflects the normal amount of network traffic associated with establishing TCP connections, TCP Acknowledgement packets that must be processed, keep-alive messages and other overhead functions, some of which can be minimized by tuning, see [5]. However, all five indicators of the transaction load are autocorrelated. This is a useful finding in case not all the metrics happen to be available, for one reason or another. Moreover, as Figure 10 shows, the

```
date      time      c-ip          cs-method       Cs-uri-stem                                      sc-status  sc-bytes  cs-bytes  time-taken
5/16/2002 5:11:28   216.23.50.222  GET  /iccmFORUM/public/img/ICCMLink.gif                          200       4150      414       516
5/16/2002 5:11:28   216.23.50.222  GET  /iccm.asp                                                   200       42104     517       5015
5/16/2002 5:11:28   216.23.50.222  GET  /iccmFORUM/public/img/RedArrow.gif                          200       1076      414       218
5/16/2002 5:11:28   216.23.50.222  GET  /iccmFORUM/public/img/Clear.gif                             200       267       411       235
5/16/2002 5:11:29   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0101.gif200               17854     432       875
5/16/2002 5:11:29   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0102.gif200               17309     432       625
5/16/2002 5:11:29   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0103.gif200               9680      432       375
5/16/2002 5:11:29   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0104.gif200               17717     432       625
5/16/2002 5:11:30   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0106.gif200               16351     432       500
5/16/2002 5:11:30   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0105.gif200               16923     432       828
5/16/2002 5:11:30   216.23.50.222  GET  /iccmforum/public/Capacity/CPUResource/Wicks0107.gif200               7820      432       313
5/16/2002 5:11:30   216.23.50.222  GET  /iccmFORUM/public/img/innovation.gif                        200       3531      416       344
5/16/2002 5:11:30   216.23.50.222  GET  /iccmFORUM/public/img/Eval3x1.gif                           200       2772      413       219
5/16/2002 5:11:30   216.23.50.222  GET  /iccmFORUM/public/img/BlueArrow.gif                         200       297       415       328
5/16/2002 5:11:30   216.23.50.222  GET  /iccmFORUM/public/img/BlueArrowLeft.gif                     200       1077      419       343
```

FIGURE 7. OUTPUT FROM THE EXTENDED FORMAT IIS WEB LOG.

relationship between the transaction load and processor resource consumption at this site, for example, is quite obvious. Certainly, this is an unsurprising result on a machine that is dedicated to a web services application, as in this case.

Correlating the response time measurements reported in the web log against other transaction statistics available in System Monitor is problematic because, unfortunately, this critical measure of service levels is generally not available for Microsoft web services applications. The one measure of transaction response time that is available in the System Monitor is the ASP Request **Execution Time** and Request **Queue Time** counters. The value reported, which is also in milliseconds, is the time of the *last* ASP request that executed. In an environment such as the web site we have been discussing where ASP transactions arrive at a rate of up to 5 per second, the ASP Request **Execution Time** and Request **Queue Time** counters are properly viewed as a sampling mechanism.

As discussed in [12], there is sufficient ASP measurement data to calculate the mean response time of ASP Requests using Little's Law, as follows:

```
MEAN RESPONSE TIME =
(REQUESTS EXECUTING + REQUESTS QUEUED)/
REQUESTS/SEC
```

Figure 11 shows the estimated ASP Request mean response time for this site, calculated using the Little's Law formula. Note that this is a mean value, appropriate for many mean value analysis (MVA) performance modeling techniques, but limited for service level reporting. We recommend that the average response time calculated using this formula always be validated against the average **Execution Time** and **Queue Time** calculated from the sampled performance Counters.

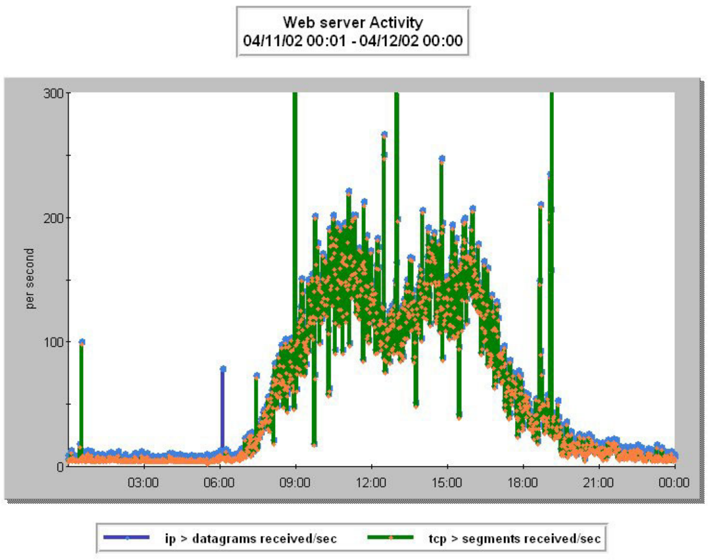


FIGURE 8. MEASUREMENTS FOR IP DATAGRAMS RECEIVED/SEC AND TCP SEGMENTS RECEIVED/SEC ARE USUALLY ISOMORPHIC DUE TO THE RELATIVELY SMALL SIZE OF HTTP REQUEST PACKETS.

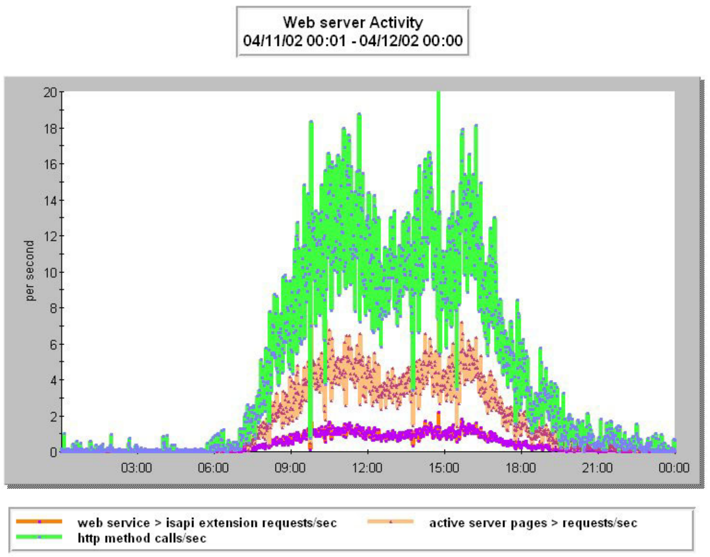


FIGURE 9. COMPARING THE RATE OF HTTP METHOD CALLS, ISAPI CALLS, AND ASP REQUESTS/SEC.

Transaction decomposition. As we have seen, both the web log event data and the interval performance data provide measures of overall ASP transaction response time, at least from the standpoint of the IIS server. But these measurements provide no insight into the impact of delays associated with the processing by middle tier COM+ components and/or calls to back-end database connections, assuming the web services application is architected to take advantage of these facilities. In the case of calls to COM+ applications, an event-oriented trace mechanism is available that can be used to determine the rate of requests for component activation and the service time of those requests. This facility is known as COM+ Events. At least one commercial software package currently uses the COM+ Events trace facility to generate transaction load and response statistics to support tuning and capacity planning. In contrast, no transaction-level or user level measurement data currently exists to monitor calls to an MS SQL Server 2000 back-end database. Moreover, there is no documented facility currently available in that product that could be exploited to gather the required SQL Server measurement data.

The good news is that at least the performance of COM+ components can be monitored. However, doing so requires a 3rd party package. When a COM+ component is installed with support for events and statistics enabled (reference the Concurrency tab in the COM component Property page), transaction load and service time information is forthcoming. The COM+ statistics that are available are provided using the COM+ Events tracing facility. When Events are enabled, each call and return to a COM+ interface method is traced, which corresponds to the arrival rate and completion rate of specific transactions, respectively. Using a correlation ID, a COM+ Event tracing application can also determine how long an interface method executed.

As trace facility exclusive to the COM+ runtime, COM+ Events are not integrated into the Windows Measurement Instrumentation (WMI) framework via either a WMI provider or a perflib DLL. There are, however, several specific system management tools that enable COM+ Event tracing and report the results. The most familiar is the Transaction Statistics display in CSE, illustrated in Figure 12. The Transaction Statistics display shows some interesting current and aggregate statistics. However, it is of very limited value for performance monitoring since what you see illustrated in Figure 12 is all you get. There is no ability to break out statistics by component, no ability to gather interval statistics, and no ability to access individual transaction-level data. It is worth noting that a COM+ component does not have to support Transactions in order to be counted in the Transaction Statistics display – this is one area where COM+'s use of the terminology is inconsistent.

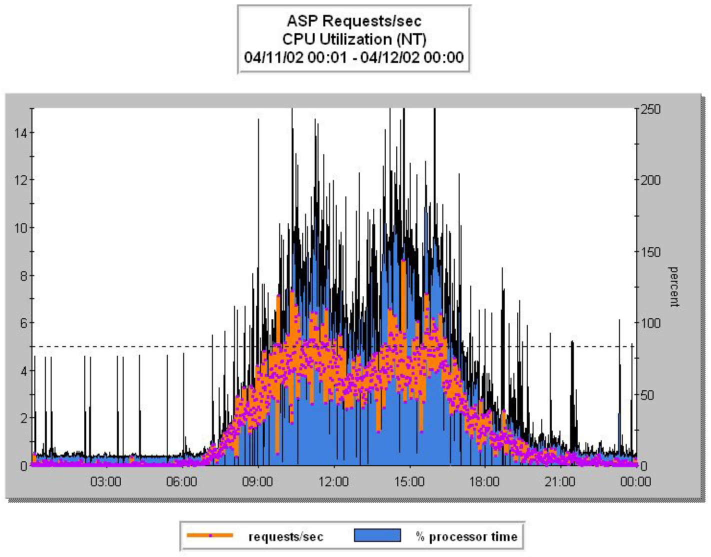


Figure 10. CPU utilization as a function of ASP Requests/sec.

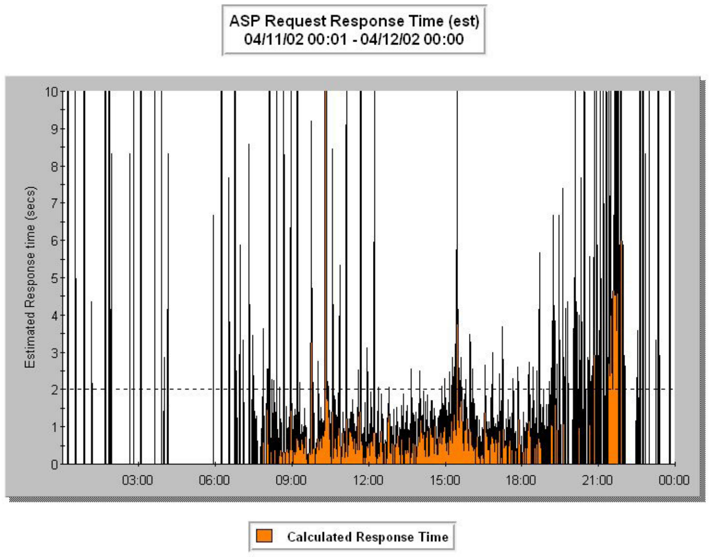
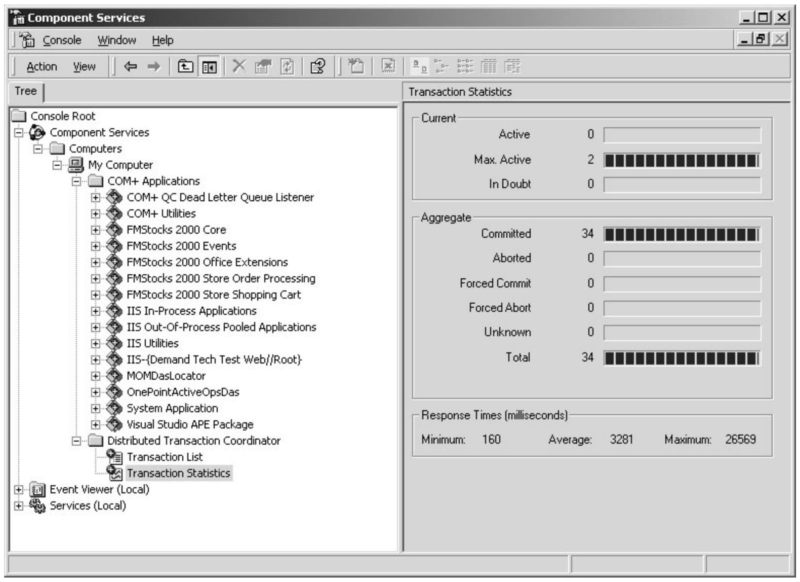


Figure 11. ASP Request response time estimated using Little's Law.

The Visual Studio Analyzer, an application performance profiling tool, also supports COM+ Event tracing to allow developers to determine how well their COM+ applications are performing. This is a useful tool that developers can use, but it is of little help in a production environment. The Component Load Balancing (CLB) clustering facility in Application Center 2000 is the other Microsoft application that relies on COM+ Event tracing. CLB's control component pulls transaction level statistics from each remote machine in the cluster every 800 milliseconds in order to make optimal routing decisions.

A third party COM+ performance monitoring application called AppMetrics, developed by Extremesoft, also exploits the COM+ Events trace facility. AppMetrics is the only robust performance monitoring application currently available for COM+ componentware. As Figure 13 illustrates, the AppMetrics software breaks out the transaction statistics by component, reporting both arrival rates and service times. When the AppMetrics data is collected for identifiable COM+ server applications, it is also possible to gather associated data on processor and memory utilization by collecting Process counters for those specific instances of **dllhost**. Together, this measurement data is robust enough to apply a variety of traditional performance engineering and modeling techniques for COM+ web services application programs.

While the AppMetrics COM+ transaction load and response time data supplies a critical piece of the performance monitoring puzzle, it does not encompass the full picture of web services application performance for the Microsoft architecture depicted back in Figure 4. As noted earlier, timing and resource utilization information on database calls per transaction is sorely lacking when MS SQL Server is the back-end DBMS. Moreover, the



```
Console Root
  Component Services
    Computers
      My Computer
        COM+ Applications
          COM+ QC Dead Letter Queue Listener
          COM+ Utilities
          FMStocks 2000 Core
          FMStocks 2000 Events
          FMStocks 2000 Office Extensions
          FMStocks 2000 Store Order Processing
          FMStocks 2000 Store Shopping Cart
          IIS In-Process Applications
          IIS Out-Of-Process Pooled Applications
          IIS Utilities
          IIS-{Demand Tech Test Web//Root}
          MOMDasLocator
          OnePointActiveOpsDas
          System Application
          Visual Studio APE Package
        Distributed Transaction Coordinator
          Transaction List
          Transaction Statistics
  Event Viewer (Local)
  Services (Local)
```

```
Current
        Active   0
   Max. Active   2
      In Doubt   0

Aggregate
     Committed  34
       Aborted   0
 Forced Commit   0
  Forced Abort   0
       Unknown   0
         Total  34

Response Times (milliseconds)
Minimum:   160     Average:   3281     Maximum:   26569
```

Figure 12. The Transaction Statistics display in Component Services Explorer.

measurement data we described that does exist for HTML Method calls, ASP scripts, and COM+ transactions cannot easily be correlated.

Nevertheless, even when the resource accounting data cannot be attributed to specific user transactions precisely, in many circumstances reasonable steps can be taken to apportion resource consumption per transaction approximately. Apportionment techniques are a well-established practice in similar situations on other platforms whenever resource accounting is not as complete as desired. Apportionment would permit capacity planners to employ a broad set of analytic techniques to resolve performance issues with confidence at sites deploying Microsoft's web services application platform.

Summary.

Capacity planners responsible for web services applications on the Microsoft Windows 2000 platform face difficult, but not unfamiliar problems with inadequate performance data. As Microsoft has rapidly evolved its software development platform for delivery of dynamic web-based content, deployment-oriented tools for web services application performance monitoring have lagged behind. This paper describes the basic architecture of web services applications on the Microsoft platform and discusses the performance monitoring data that can currently be gathered to manage the deployment of these applications.

Conceptually, the Microsoft platform consists of three application processing tiers that represent (1) the presentation layer, (2) the business logic, and (3) the back-end database processing. The presentation layer currently relies on Active Server Pages (ASP) scripts, a Microsoft-proprietary technology that permits HTML codes to be intermixed with VBscript or Javascript code to generate HTML Response messages. The business logic layer relies on COM+, a set of object-oriented Component Services that simplify the job of building multi-threaded transaction processing applications. COM+ is also a Microsoft-proprietary technology. The back-end database processing can be handled by any of a variety of DBMS engines, including Microsoft SQL Server and Oracle. Optionally, these processing components can be deployed in a wide variety of runtime environments, including multiple machines that are clustered for scalable performance.

Microsoft built ASP on top of an existing Internet Information Server (IIS) web server facility called ISAPI that was designed to allow web pages to be created programmably. Standard web server transaction logging facilities can be used to monitor the arrival rate and service time of ASP scripts, similar to the way other HTTP Method Calls are instrumented. In addition, interval-oriented ASP transaction statistics are also available from a standard Windows 2000 performance monitor, such as the bundled System Monitor application. The ASP transaction statistics include a measure of script execution service time and queue time, but these are properly understood as a sampling technique that measures the response time of the last ASP script execution. Using Little's Law, it is also possible to calculate average response times for ASP scripts.

Resource accounting for ASP application scripts is confounded by the Application Protection parameter, new in IIS version 5.0, which sets the execution environment of ASP scripts. They can execute inside the IIS **inetinfo.exe** process address space, inside a shared instance of the **dllhost.exe** container process, or in isolated instances of **dllhost**. It is not currently possible to determine which scripts are executing in which instances of the **dllhost** container process. This makes it impossible to associate process level resource utilization statistics such as CPU and Memory consumption with the execution of specific application scripts.

Depending on whether they are defined as *library* applications or *server* applications, COM+ component programs can execute either inside the calling calling application process or *out-of-process* inside the ubiquitous **dllhost.exe** container process. While the tools Microsoft provides are woefully inadequate to the task of monitoring COM+ transaction processing programs, 3rd party tools are beginning to rise to the challenge. One third party tool utilizes the COM+ Events tracing facility to track COM+ transaction arrival rates and service times by application. Another third party tool can determine which COM+

application modules are resident in which **dllhost.exe** container process so that adequate resource accounting can be performed. These are both steps in the right direction.

The capability to process web services applications using multiple-machine clusters presents a more formidable performance monitoring challenge. In clustered environments, measurement data from multiple machines needs to be integrated to capture all application processing components. Moreover, transaction-level measurement data from the three different processing tiers – the presentation layer, the business logic, and the backend DBMS – needs to be correlated. This transaction level data, where it exists today, is piecemeal. Currently, no facilities of the Microsoft web services application runtime environment are available that could be exploited to provide an integrated view of application performance. This defect may prove to be a significant obstacle to adoption of the Microsoft application development framework for enterprise-ready, mission critical applications.

References

[1] Tim Ewald, *Transactional COM+: Building scalable applications*. Boston, MA: Addison-Wesley, 2001.

[2] G. Pascal Zachary, *Showstoppers: the breakneck race to create Windows NT and the Next Generation at Microsoft*. New York, Simon and Schuster: 1994.

[3] "Writing great Windows NT server application," Microsoft Corporation, Jan 18, 1995.

[4] Ken Auletta, *World War 3.0: Microsoft vs. the U.S. Government, and the battle to rule the digital age*. New York: Broadway Books, 2000.

[5] Thomas Lee and Joseph Davies, *Windows 2000 TCP/IP Protocols and Services: Technical Reference*. Redmond, WA: Microsoft Press, 2001.

[6] Anand Rajagopaian, "Debugging distributed Web applications," *MSDN*, Jan 2001.

[7] *Performance SeNTry version 2.4 User's Manual*. Naples, FL: Demand Technology Software, 2002.

[8] Daniel A. Menasce and Virgilio A. F. Almeida, *Scaling for E-Business: technologies, models, performance, and capacity planning*. Upper Saddle River, NJ: Prentice-Hall PTR, 2000.

[9] Kalen Delaney, *Inside SQL Server 2000*. Redmond, WA: Microsoft Press, 2000.

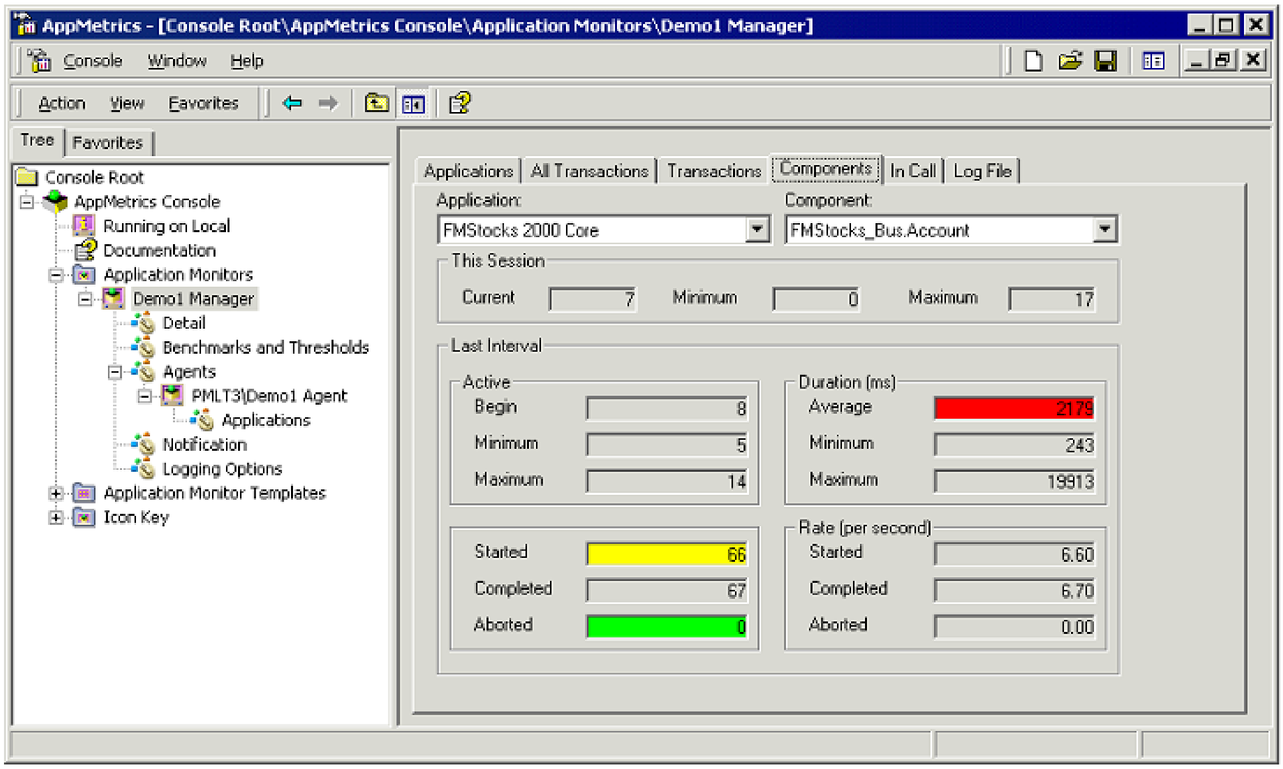
[10] Don Box, *Essential COM*. Boston, MA: Addison-Wesley, 1998.

[11] "Load Balancing COM+ Components," *MSDN*, March 2002.

[12] Mark Friedman and Odysseas Pentakalos, *Windows 2000 Performance Guide*, Sebastopol, CA: O'Reilly Associates, 2002.

[13] Jim Gray and Andreas Reuter, *Transaction Processing: Transaction Processing: concepts and techniques*. San Francisco, CA: Morgan Kaufmann, 1993.

[14] Mark W. Johnson, "Application Response Measurement (ARM) API, Version 2," CMG *Proceedings*, 2000. Also available at http://regions.cmg.org/regions/cmgarmw/marcarm.html.


AppMetrics - [Console Root\AppMetrics Console\Application Monitors\Demo1 Manager]

Application: FMStocks 2000 Core
Component: FMStocks_Bus.Account

This Session
Current 7 Minimum 0 Maximum 17

Last Interval

Active
Begin 8
Minimum 5
Maximum 14

Duration (ms)
Average 2179
Minimum 243
Maximum 19913

Started 66
Completed 67
Aborted 0

Rate (per second)
Started 6.60
Completed 6.70
Aborted 0.00

FIGURE 13. APPMETRICS COM+ TRANSACTION PERFORMANCE MONITORING.