Realtime publishers

The Five Essential Elements of Application Performance Monitoring

Don Jones

sponsored by

QUEST SOFTWARE®

Copyright Statement

Chapter 2: Tracking and Monitoring the User Transaction

Once you've accepted that the end user experience (EUE) is the ultimate top-level metric of your application's performance, how will you actually monitor it? More importantly, when the EUE isn't where you need it to be, how can you start finding the root cause of the problem?

The answer to both of these questions is the *user transaction*. I should probably take a minute to ensure we're on the same page with the term *transaction*. In the software development world, *transaction* usually means some group of operations that must be completed together or not completed at all. For example, in a financial application, you have to be sure to debit and credit the two sides of a ledger at the same time; if you only do one or the other, everything's messed up. A *user transaction* has a vaguely similar sort of meaning: It's a collection of discrete steps that a user undertakes to complete some higher-level task. The classic example is an e-commerce shopping cart, which not only involves numerous steps for the user but also typically includes numerous back-end steps: saving a shopping cart, processing a credit card, generating an order entry, creating an invoice, and so forth. User transactions typically impact multiple components in your application, and they are the top-level unit of work that results in the EUE that you monitor.

Monitoring the user transaction can be incredibly complicated simply because our applications have, over the years, evolved into pretty complex systems. In fact, monitoring application performance *used* to be a *lot* easier.

How Application Performance Monitoring Got Away from Us

So when did things go awry? When did our applications become so complex? When did monitoring become so difficult? When did we have to start worrying about a discrete EUE metric? The answers to these questions—as well as some of the solutions for better application performance monitoring (APM)—lies in the way our applications have evolved.

Remembering When IT Was Easy: Monolithic Applications

Figure 2.1 shows how the first applications worked: They ran on a single computer, were entirely self-contained, and typically interacted with one user at a time.

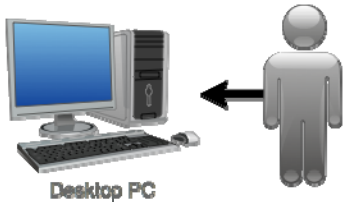

Figure 2.1: Monolithic applications.

That *self-contained* part is the real key: Everything the application needed to function existed in a single place, in a single big chunk of code, and couldn't be broken down into any smaller components. The first applications like this ran on mainframes, but most of us still have a few applications like this today. Simple applications like Windows Paint or Calculator are examples, but many businesses continue to use self-contained line of business applications.

Believe it or not, monitoring the EUE was actually easier with these kinds of applications, and we did is instinctively. "My application is slow," a user would complain. No problem—a developer could fire up the application on their own, and they'd be getting exactly the same experience as the user because the developer was using the exact same code in the exact same way.

Eventually, though, we needed our applications to share information. Some of the earliest ways for doing this were-file based databases that lived on a network file server and were accessed by otherwise-monolithic applications. In early examples, the database didn't consist of any code at all; the only thing keeping the application from being entirely self-contained was the network that it had to use in order to get to its data files. Figure 2.2 shows this minor evolution.

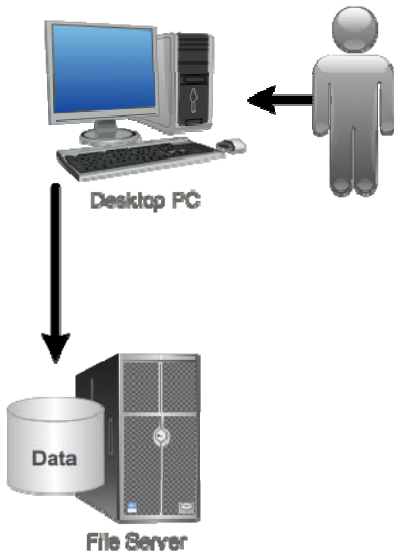

Figure 2.2: Application accessing data from a file server.

Even these applications were relatively easy to troubleshoot from a performance perspective. There were really only three things to look at: the application code, the network performance (which, back then, was always slow), and the number of people contending for the data on the file server. You could easily eliminate one or two of those factors and narrow down performance problems pretty quickly. In fact, this is where our entire concept of application performance troubleshooting came into being: Eliminate as many factors as possible and see if what's left is performing well, then start adding factors back in until you see the performance problem. Many folks today still try to troubleshoot their applications that way, but there's a problem: Our applications are a bit more complex than that, now.

When IT Started to Get Hard: Two- and Three-Tier Applications

We started to realize that one of the big problems with the shared-file database model is that you could only support a few people in an application. Once too many people started contending for access to the same file, performance went from "slow" to "awful" pretty quickly.

Big businesses, in fact, tended to stick with single-tier applications running on mainframes and accessed from terminals or terminal emulators. A surprising number of businesses still use this model for major line of business applications. The technology industry, however, started looking at a solution: Rather than having each user's application try to open a data file, why not create a single application whose job was to manage the data file? It could accept query and change requests from applications, and make those queries and changes on the application's behalf. The relational database management system (RDBMS) was born, as was client-server computing—shown in Figure 2.3.

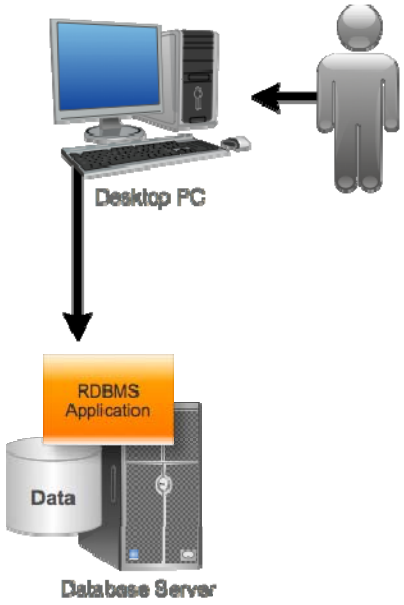

Figure 2.3: Client-server application.

Now, we could have a lot more people using the application at once—but troubleshooting performance became markedly more difficult. No longer was all of our code running in one place; now we had to worry about the client application's performance as well as the RDBMS' performance—as well as the network, the database server's disk subsystem, and so on. Still, it wasn't entirely impractical to eliminate one or two factors to test performance. Did the RDBMS respond quickly to ad-hoc queries made from a developer's toolset? If so, the problem was probably in the client application or the network. This is exactly where APM started to get away from us, though, and the problem was compounded when we had another bright idea: *n*-tier computing.

Now IT Is Difficult: Multi-Tier, Multi-Component, Distributed Applications

Under the client-server model, we realized that our client applications tended to be pretty heavy. They incorporated most of our business logic, and really did the bulk of the work for the application. Updating these applications was a pain, however, because we had to touch every user's computer. We decided to add another layer to the application—and sometimes many layers—that would just handle things like business logic, data validation, and so on. That would distribute some of the workload from the client applications as well as move some of the workload back a level from the database server, which was a very difficult tier to scale. Figure 2.4 shows what an *n*-tier application looks like.

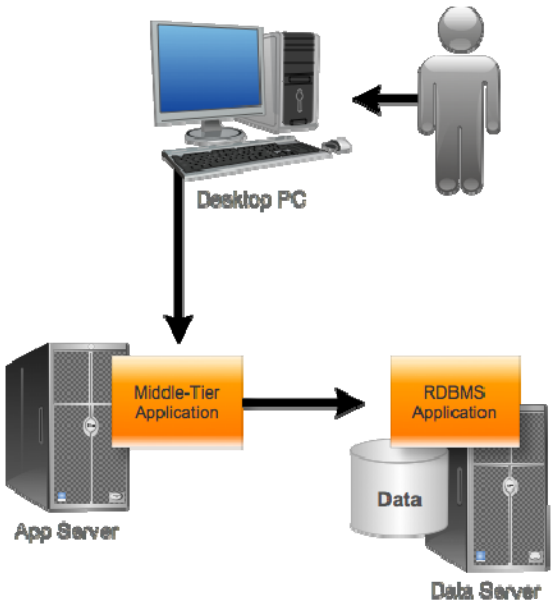

Figure 2.4: An example *n*-tier application.

We actually helped improve application performance with this architecture, at least in most cases. Now, rather than having every user hitting the RDBMS directly, groups of users would communicate with middle-tier application servers, and *those* would talk to the RDBMS. We were basically solving the same problem we'd had with the old file-based databases by adding layers to the application that would help distribute the workload.

Unfortunately, we didn't make ourselves immune to performance problems, and now we had a *lot* of factors in play. When someone said, "The application is slow," there were simply too many places to look, and it was becoming harder and harder to test individual components with any accuracy. The client application, its network connection to the application server, the application server itself, the code running on that server, the connection to the RDMBS, *that* server—there were a lot of balls in motion.

And we continued to add layers and components. In some cases, we did so to solve a known performance problem and to further distribute our growing workload. In other cases, new layers helped componentized specific functionality or business logic, making it easier to maintain or re-use. Figure 2.5 shows the result: multi-tier applications consisting of multiple components running across multiple physical and virtual computers—sometimes across clusters of computers. Users interact differently with different tiers; customers might access a Web site in their computer's Web browser, while partner companies might connect directly to certain middle-tier components.

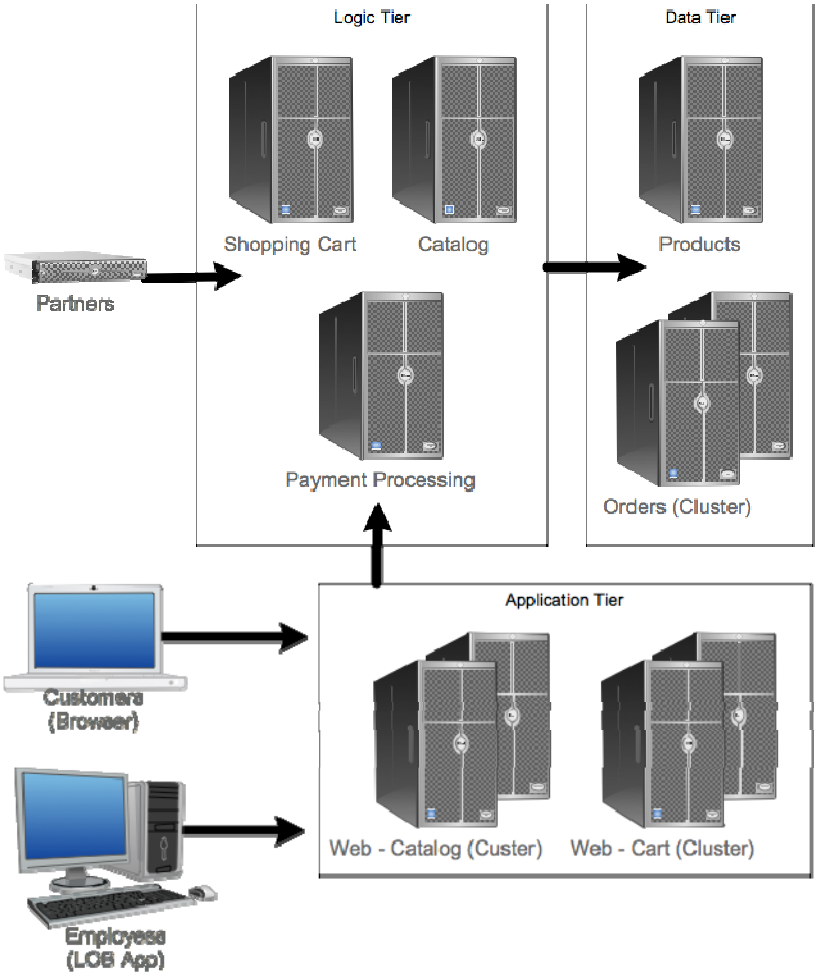

Figure 2.5: Modern, multi-tier distributed application.

Performance problems? Want to eliminate a factor or two to see if that helps? Forget it. Our applications have moved far beyond our ability to troubleshoot them using that old technique. In fact, there are so many ways to interact with an application that we can't even get a feel for the EUE using the old "try it yourself and see how it runs" technique. The applications have simply become too complex, too distributed, and too interdependent. APM is now officially difficult.

In order to regain control, we need to step back and start developing new ways of thinking about our applications.

Applications as a Stack

A more modern way of visualizing an application is to think of it as a collection of services, each provided by a different application tier or component. We also have to recognize that an individual tier or component might well provide the same services to *other,* independent applications. For example, a component that provides real-time shipping estimates to a shopping cart application might also provide shipping information to our distribution center, or even to partner companies who are shipping items directly to our customers. Our applications thus form a *stack* of these components, and we have to be continually aware that any given application is not necessarily the only consumer of those services—in other words, sometimes performance problems may be originating from other applications that are placing a load on some of the services *our* application depends upon. Figure 2.6 is a common type of logical application architecture diagram that illustrates an application stack.

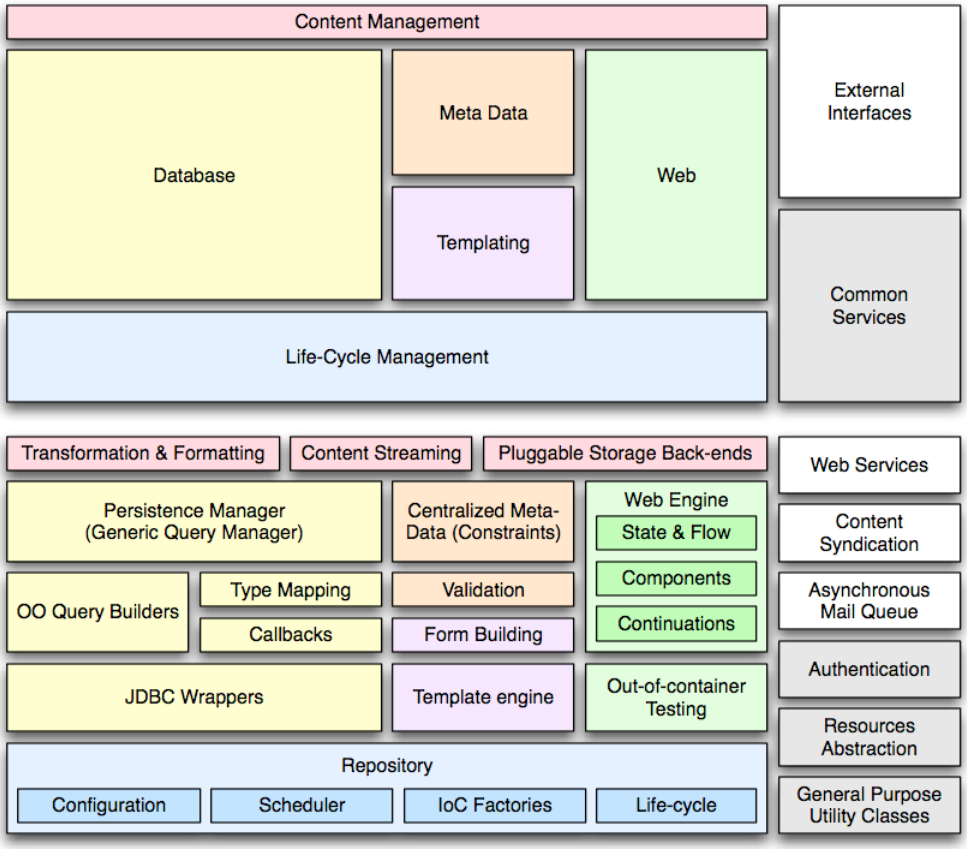

Figure 2.6: Logical application architecture.

These diagrams can become very complex and detailed, matching the complexity of the application itself. In some cases, the application may even rely on external components and services, such as a Web service that performs currency exchange rate queries. The basic technique of application performance troubleshooting—looking at each component's standalone performance—still holds true; however, for applications this complex, it's virtually impossible to do that without specialized tools, and without understanding how user transactions move across this stack.

How User Transactions Traverse the Stack

One reason that these complex applications are so difficult to monitor and troubleshoot is that it can be difficult to ascertain exactly which bits are involved in any given user transaction. For example, a user logging on to a Web site may engage dozens of individual components, each of which contributes a performance impact to the overall EUE. Figure 2.7 illustrates how the various components may interact to complete a single user transaction.

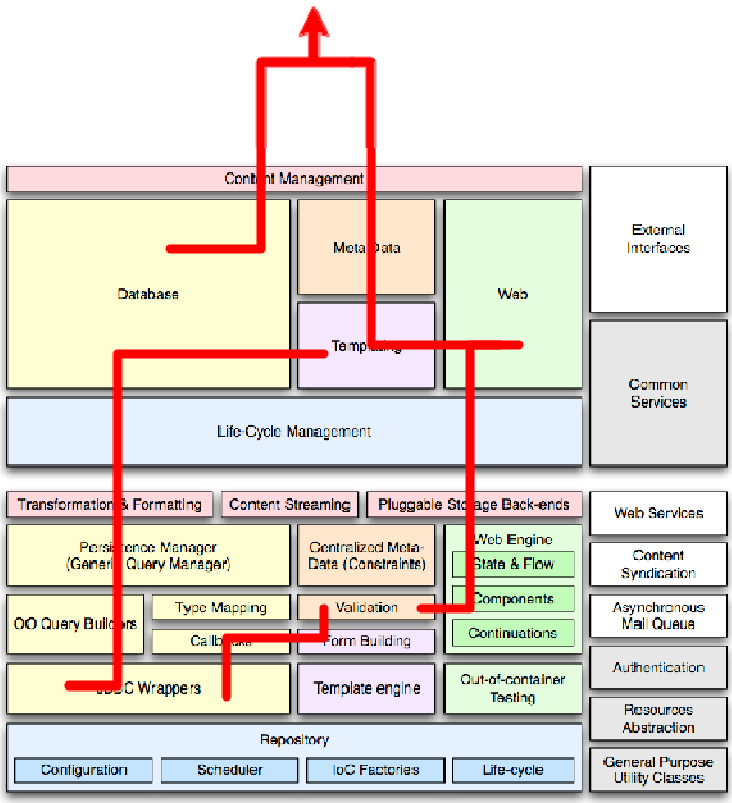

Figure 2.7: Various components interact for each user transaction.

The point here is that it can be incredibly tough—if not entirely impractical—to *manually* chart the components that are engaged for a single given user transaction; doing so for a set of commonly-performed transactions is often impossible. Yet without understanding exactly which components are involved, it becomes completely impossible to test their individual performance to see which one may be contributing to an unacceptable EUE. In order to even begin troubleshooting, you have to *know*, not merely guess, *exactly* which bits of the application are involved with a given operation.

Old-School Monitoring, and Why It Doesn't Work

Traditional performance monitoring tends to focus on macro levels: We monitor the database server, the Web server, the network, and perhaps the overall performance of an application server. We tend to look at that performance information in an abstract fashion. It's very common to see harried administrators staring at a console like the one shown in Figure 2.8, which illustrates individual performance characteristics of a particular application element, such as an application server or database server.

Figure 2.8: Monitoring individual performance characteristics.

The problem is that this macro-level monitoring completely ignores *what's happening throughout the application*, especially as performance relates to particular transactions. In other words, we can't see how a given shopping cart checkout, or logon, or whatever other user transaction we're examining, contributes to specific performance conditions. We can see that a server is busy or not busy, sure, but that doesn't tell us why a particular user transaction is too slow. Using these traditional, macro-focused performance monitoring techniques, we can't really find the root cause behind an unacceptable EUE.

The 5D Approach's Goals for Monitoring User Transactions

The 5D approach doesn't rely on these traditional, macro-level performance characteristics because those *don't* help drive to the root cause of an unacceptable EUE. I've already written about the EUE being our ultimate top-level application performance metric, and I've explained that the EUE is—in a simple explanation—a measurement of how long it takes an application to process a given *user transaction.* Emphasis on user transaction: We're measuring things that our end users actually do with our application. That being the case, the trick for solving performance problems isn't to start looking at technology components; the trick is to continue following that user transaction through the application's stack, measuring at each step how long application components are taking to complete their portion of that user transaction.

In other words, rather than starting at the EUE and then jumping to the server level, we should monitor that same EUE at a more detailed level. If the EUE is a measurement of how long the entire transaction takes to complete, the next level down should be monitoring the time it takes to complete each discrete portion of that transaction. In other words, as suggested in Figure 2.9, we have to *follow the user transaction* as it hits each element of our application, and measure the time the transaction spends within each element—indicated by the blue blocks in the figure.

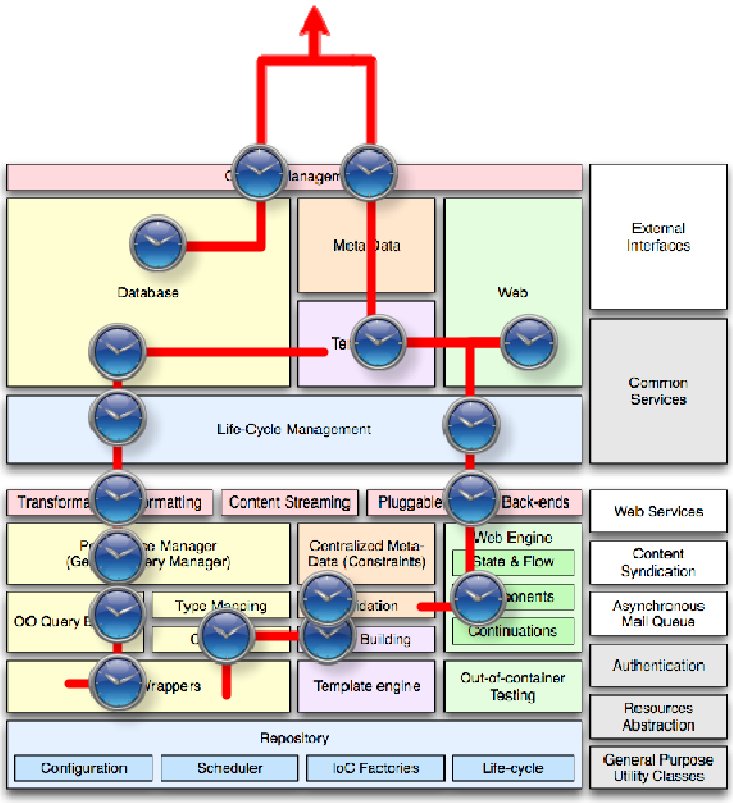

Figure 2.9: Measuring the user transaction in each application element.

We don't worry about technological details like processor or memory utilization—not yet, at least. In fact, we ignore the infrastructure somewhat and focus on the *services* that infrastructure is providing in the form of application elements. The infrastructure still has a play here, because we're obviously also concerned about infrastructure-level services like network communication, but the idea is to trace the user transaction through each active component and figure out how much time each one is contributing to the EUE. We're still focused on the EUE, just at a more detailed level. So how do we actually do it?

Techniques for Monitoring User Transactions

You're definitely not going to be monitoring user transactions manually; you're either going to have to build or—far more likely—buy tools that can do it for you. Currently, these tools rely on two broad techniques, and two broad implementations, to perform user transaction monitoring.

As an example, let's work with a more simplistic user transaction: downloading a single image from a Web server. Understand that this isn't a real-world example of a common business user transaction, but it will serve as an easy-to-discuss example. As Figure 2.10 shows, this transaction consists of multiple technological steps, just as a more realistic business user transaction would.

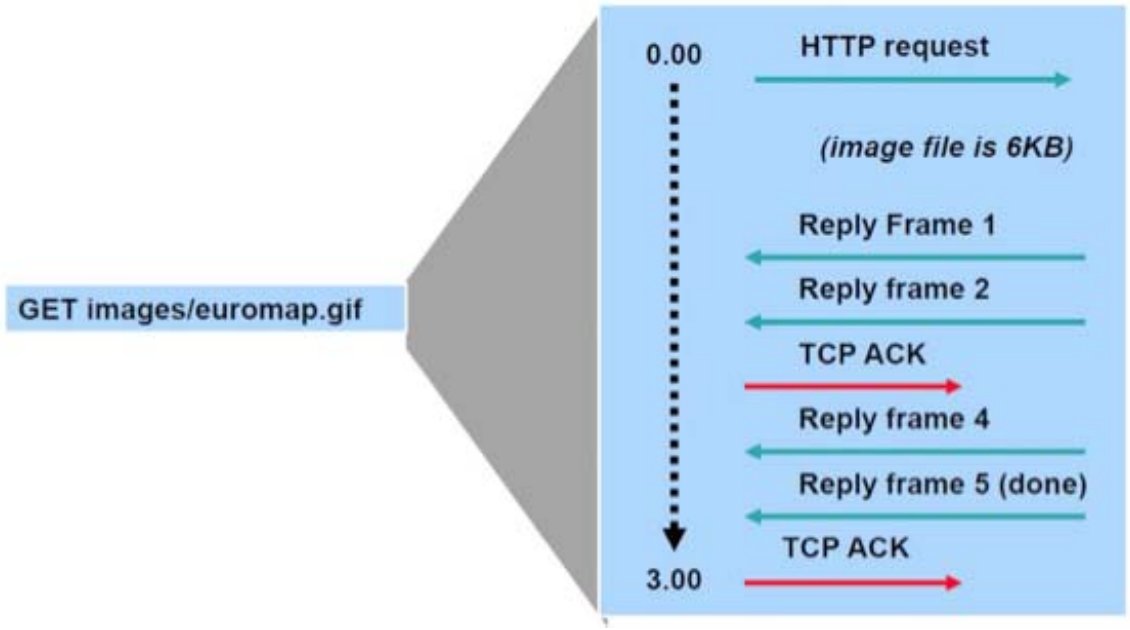

Figure 2.10: A simple user transaction—downloading an image from a Web server.

> **Note**
> This is definitely a simplistic example: Downloading a single image is less complicated than a more real-world user transaction, like completing a shopping cart, which would involve multiple steps on the user's part. However, this example serves to illustrate that any given user transaction consists of multiple technological steps—such as the conversations between Web server and browser—necessary to complete whatever the user was doing. This example *does* focus on a *user transaction*: It is something a user would do, and it is something that a user would have performance opinions about.

The EUE in this example is the time it takes for the entire download to complete—say, 3 seconds as illustrated here. Once that EUE starts to pass our threshold for "good performance" (3 seconds seems like a long time for a single image), we have to dive a little deeper and start tracing that transaction.

High-Level Techniques

There are two main techniques used to monitor performance at a user transaction level: event correlation and analysis, and transaction tagging. Some tools will use one or the other, while other tools may rely on both for different aspects of their job.

Transaction-Centric Event Correlation and Analysis

Essentially, correlation involves a sort of meta-monitor that monitors your entire application as a single user transaction is being performed, and correlates the performance that it sees from various components. This is more than just gathering performance data and centralizing it into a single view; there's also a time factor, with the idea being that all of the performance *at a particular moment in time* will correspond to one or more user transactions that are being conducted at that moment in time. In other words, if there are *x* user transactions in progress, the total observed performance at that same moment in time is the performance attributable to those transactions. Rather than just looking at performance in an abstract fashion, we're looking at performance that relates to a specific set of tasks that the application is attempting to complete.

Just that high-level description may sound complex, but the reality is much more involved. For example, a tool may be able to tell you that, "the middle-tier application server only exceeds threshold *x* when the database server's response time exceeds *y,* and when the transaction involved includes more than *z* items in an order." This is useful because you're not *just* looking at, say, the application server, nor are you looking at performance in an abstract sense that is disconnected from the application's workload. Instead, you're looking at cross-application performance for a given workload.

Following our image-downloading example, event correlation might look at separate performance measurements like the time it takes a database server to query and retrieve the image from a database, the processing power required of the Web server to process the image download request, and so on.

Transaction Tagging

The other broad technique for tracing user transactions is called *transaction tagging.* The basic idea here is to mark a particular user transaction so that it can be readily identified as it moves through the application stack, and to then intercept each portion of the application and measure the time it takes to complete that particular user transaction.

Without getting overly technical about how this is done, you can imagine using a tool that creates a "synthetic" transaction with some specific, say, transaction ID number. The tool then analyzes the amount of time each component requires to complete that transaction. Sometimes, this is accomplished by using agent-based instrumentation that runs on the same computers where your application's components are executing; in other instances, you may use network probes to capture inter-component traffic, allowing you to identify transactions as they move from machine to machine, and to measure the time it takes each machine to send the transaction on to the next one. In some cases, adding instrumentation to an application component may actually require developers to insert specific "hooks;" in other cases—especially with code running in managed code frameworks like Microsoft .NET or J2EE—agents may be able to hook into the framework's runtime without any modifications to your application code.

Continuing our image-download example, transaction tagging might ask for a very specific image—one that a normal user wouldn't normally request. Doing so would allow that particular request to be traced granularly through the application stack. In fact, the technique is not entirely different from the network trace that a network administrator might perform, except that this trace is focused on application components rather than infrastructure elements. The result might be displayed as a kind of thread analysis, such as the one shown in Figure 2.11, that illustrates the amount of time needed to complete each discrete step of the transaction.

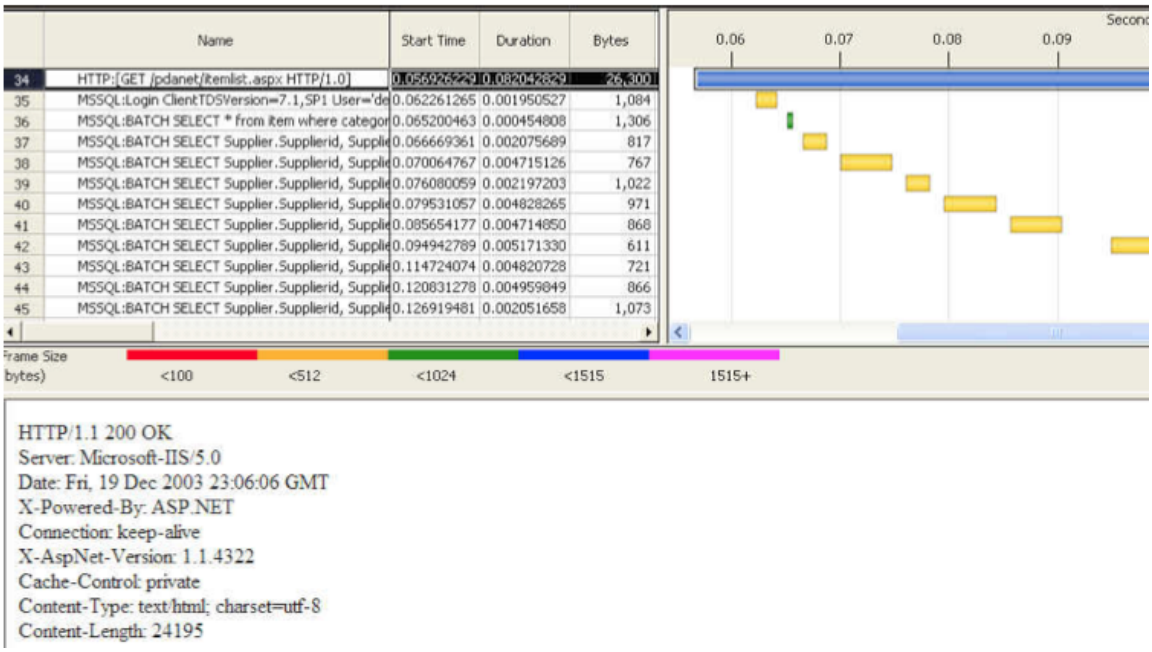

Figure 2.11: Examining the performance of each discrete step in a transaction.

Implementation Details

Both correlation and tagging can be implemented in a couple of ways: agentless and agent-based. Neither is inherently good or bad, and both deliver slightly different advantages.

Agent-Based

My definition of *agent* is some piece of software that is installed directly on the same machines that execute your application code. Because of their location, agent-based systems can often collect more detailed data. For example, an agent may be able to directly hook a managed code runtime engine, such as a Java virtual machine (JVM) or the .NET Framework's Common Language Runtime (CLR). Agents can also be installed to monitor lower-level operating system (OS) performance, such as monitoring individual application threads, or sockets, or system reads and writes. This data is then transmitted to some central monitoring application, which correlates the data. The trick is to have agents that can natively "talk" to each major component of your application. As Figure 2.12 shows, that can require a lot of variety on the part of the application monitoring solution's vendor, but it can give you an extraordinary amount of detailed insight.

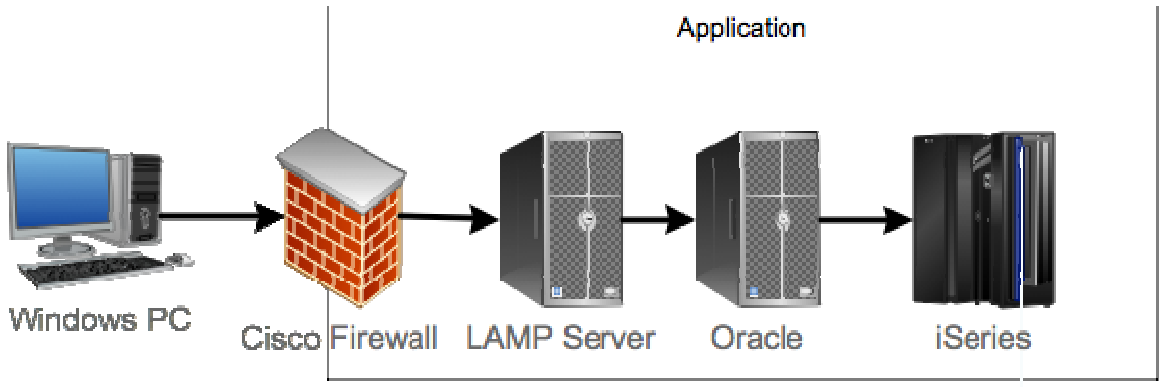

Figure 2.12: The major components of an application where agents might reside.

> **Note**
>
> Many of today's enterprise-class platforms *provide* the hooks needed to obtain performance information, which makes APM solution vendors' jobs a bit easier. They just need to write agents capable of using those hooks and transmit the resulting data back to home base for correlation and analysis.

A downside of agents, of course, is that they have to be installed and maintained, and it's not impossible or unheard of for agents to impose their own performance overhead, which means it may not be feasible to use them all the time. If you're considering a solution that relies on agents, talk to the vendor and some of their other customers to get a feel for whether the agent-based approach seems to work well for them. Well-written agents shouldn't diminish application performance.

Agentless

Agentless systems don't require anything to be installed on your servers; instead, they observe from without. These may take the form of a centralized monitoring application that remotely observes server performance or a network appliance that monitors all traffic passing between your application servers.

The obvious upside is that agentless systems are non-invasive; the downside is that they can be expensive to deploy broadly, and they may never be able to capture the same detailed level of information as an agent-based system.

> **Note**
>
> As I've written, some vendors may use a hybrid approach that offers agent-based monitors for some elements of an application while relying on agentless monitors for other elements.

The downside of agentless systems is their lack of tight integration. For example, you might have an application where the database server isn't performing well. An agentless system might be able to tell you that transactions are taking a long time to come out of the server, but it won't necessarily be able to tell you *why*. An agent that hooked into the database server's software, however, might reveal a high number of deadlocks or other database-specific issues. The domain-specific knowledge available to an agent is simply higher than the knowledge available to an external, generic observer.

Again for clarity, my definition of *agentless* is some monitoring component that does not have to reside directly on the machines that execute your application code. It might well be that an "agentless" system is comprised of agent software that is installed onto a *standalone* machine or is even an appliance of some kind. Network probes are perhaps one of the more common examples of agentless systems. These are often installed in-line on the network, allowing them to "see" all network traffic—perhaps within the data center, or between the data center and end users—and to report back to a central monitoring station with details. Those details might include observations of tagged transactions, for example.

Still, these kinds of probes can often provide performance information that is otherwise unavailable. For example, Figure 2.13 shows an application that relies on external components, such as a payment processing service or a shipping rate service. As these systems aren't yours to control, a probe—which sits in-line and reports on all traffic flowing to and from those systems—can be the only practical way of capturing performance information about those services.

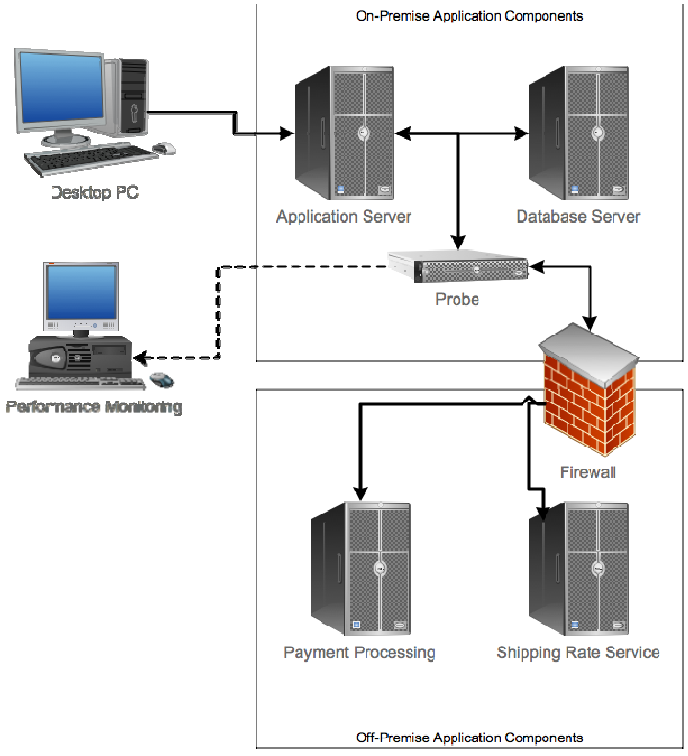

Figure 2.13: Using probes to collect in-line data from external components.

An Example of User Transaction Monitoring

Let's walk through a more robust example of how user transaction monitoring can help monitor, manage, and improve application performance. We'll start with a situation where the EUE for a given task—say, looking up a customer order in the system—is taking an unacceptably long period of time. We don't know this because a user told us; we know it because we've observed the performance using an automated EUE monitoring tool. That tool has thrown up a warning and alerted us to a problem with the EUE (see Figure 2.14).

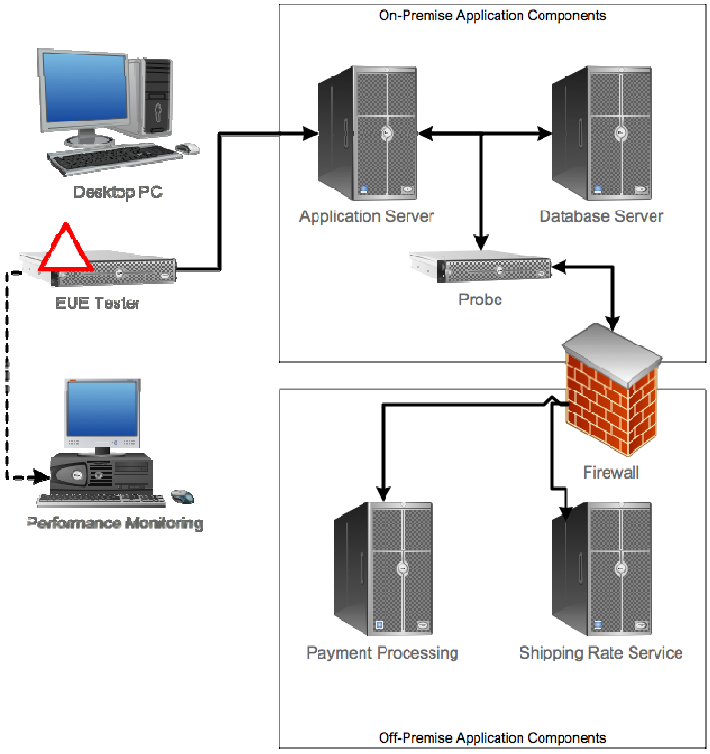

Figure 2.14: Alerted to an EUE problem by a testing tool.

We immediately spring into action. As Figure 2.15 shows, we start looking at the data being collected by agents installed on our applications' servers and by a network probe that is monitoring traffic to external services. We might even use our EUE testing tool to inject synthetic test transactions into the application so that we can trace those specific transactions and measure the component-level performance of the application.

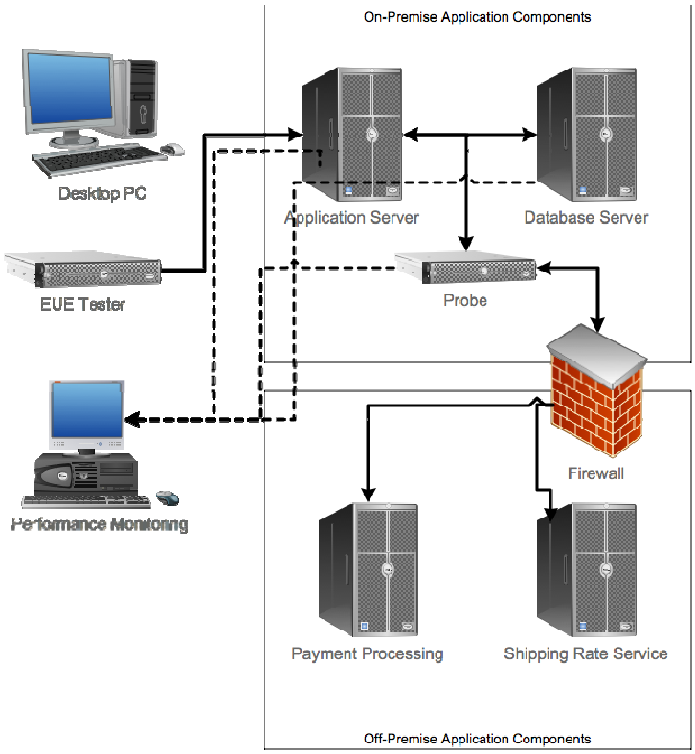

Figure 2.15: Gathering information about the user transaction.

That collected performance data can be extremely granular. By trying a variety of permutations in our synthetic test transactions, we might discover that the database server passes its normal performance thresholds whenever we inject a transaction that involves a specific category of products. Synthetic transactions and automated tools can discover this fact very quickly because they're able to try different transaction permutations more rapidly than a real human being, and can gather information on multiple transactions being conducted in parallel. By alerting us to the specific location and circumstances of the problem, as Figure 2.1 shows6, we can begin the human-level task of figuring out what the problem is.

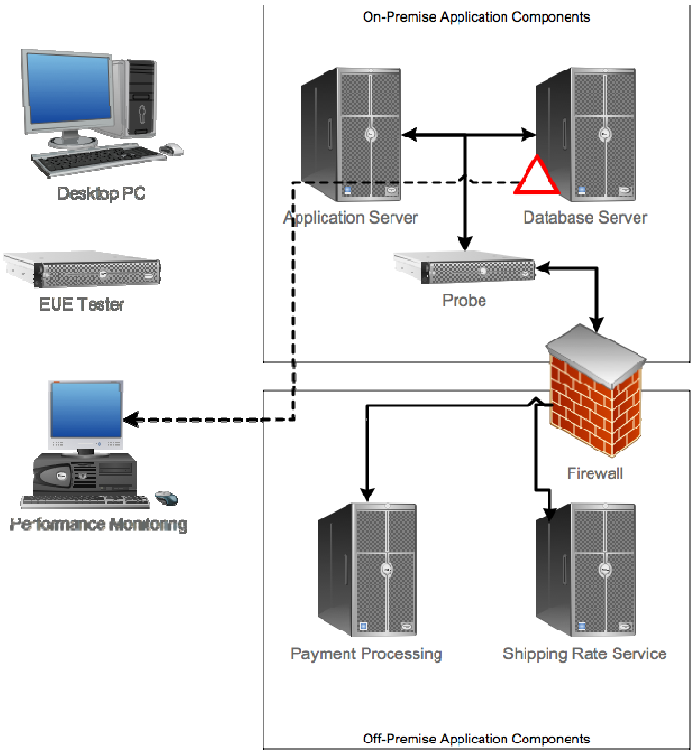

Figure 2.16: Detecting the cause of the performance problem.

Our tools might even help us find the problem more quickly. For example, suppose that this particular category of products was extremely heavy. Our tools might not realize *that* fact, but they could tell us that the database server performance problem correlates to an increased number of queries from our external shipping rate service (see Figure 2.17), which is perhaps generating error records because the products we're trying to get a shipping rate for exceed the shipping carrier's limits. That would actually indicate a bug in our code because we're not handling that error record correctly. But by seeing the correlation in poor performance between these various components, developers can start trying the problematic user transaction on their own, analyzing the detailed data, and finding their error.

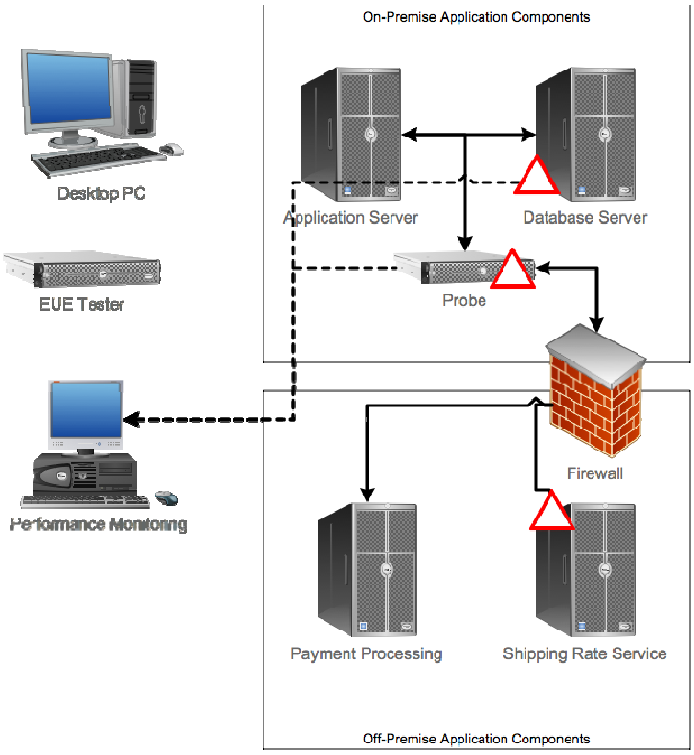

Figure 2.17: Correlating problems between multiple application components.

Without this holistic view of the application, and without the ability to trace user transactions, we could have spent a lot more time trying to solve the problem. For example, the database itself might look like the problem, but there wouldn't actually be any problem with our *data,* so we might wonder why the database is slow. It's not until we involve a tool that can correlate actions across the entire application that we realize the database's problem is actually a cascade from an external service.

Coming Up Next…

It's all well and good to talk about tracking the user transaction—but how, exactly, will you do it? Your toolset first needs to understand what your application looks like, what components are involved, and what processes are taking place. To achieve that understanding, you'll need to engage in *discovering and modeling*, a process designed to produce a model, or map, of what your application looks like. That's what we'll cover in the next chapter: modern tools and techniques for creating a model of your application.

Download Additional Books from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this book to be informative, we encourage you to download more of our industry-leading technology books and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.