Realtime publishers

The Five Essential Elements of Application Performance Monitoring

Don Jones

sponsored by

QUEST SOFTWARE®

Copyright Statement

Chapter 3: Discovering and Modeling Application Components

We're moving on to the third dimension of application performance monitoring (APM). At this point, we've learned to monitor the end-user experience (EUE) as our primary, top-level performance metric. That will let us know whether our application is performing to our business needs, and comprises the first dimension of APM. The second dimension, discussed in the previous chapter, involves tracking individual user transactions through our application, which provides us with high-level insight into how individual elements of the application are performing. At that level, we should be able to tell which major component of our application is contributing to poor EUE metrics. For example, we might be able to narrow the problem to a Web server, database server, or some other high-level component. Once we've done so, we need to dive deeper into those components to find the root cause of the performance problem.

Before we can do that, we need to help our monitoring tools understand how our application is built—and we need to do so at a fairly granular level. So before we can begin the component-level deep-dive, and indeed before we can even start seriously evaluating the user transaction tracking, we have to create a *model* of our application, which is the third dimension of APM's five-dimensional, or 5D, model.

Gartner defines this dimension as the "…technologies [that] discover what software and hardware components are exercised as user-defined transactions are executed, and how those components are related to one another, [insofar] as they support user-defined transaction execution paths." Gartner also notes that this is the one dimension in the model that is still maturing; in today's solutions, you can expect a combination of three techniques:

- IT service dependency mapping tools, which are typically technologies that discover how different types of traffic flows through different types of physical and virtual infrastructure elements
- Transaction profile snapshots, which are built from the output of transaction tracking from the model's second dimension
- Service-Oriented Architecture (SOA) topology maps, when available

In this chapter, we'll focus on these techniques as well as how to use the application model, or map, that they can help create.

Defining the Application Stack: You'll Always Miss Something

One of the biggest challenges in creating an application model is *remembering* everything. For example, ask your developers to create a map of your Web application, and you might get something like Figure 3.1.

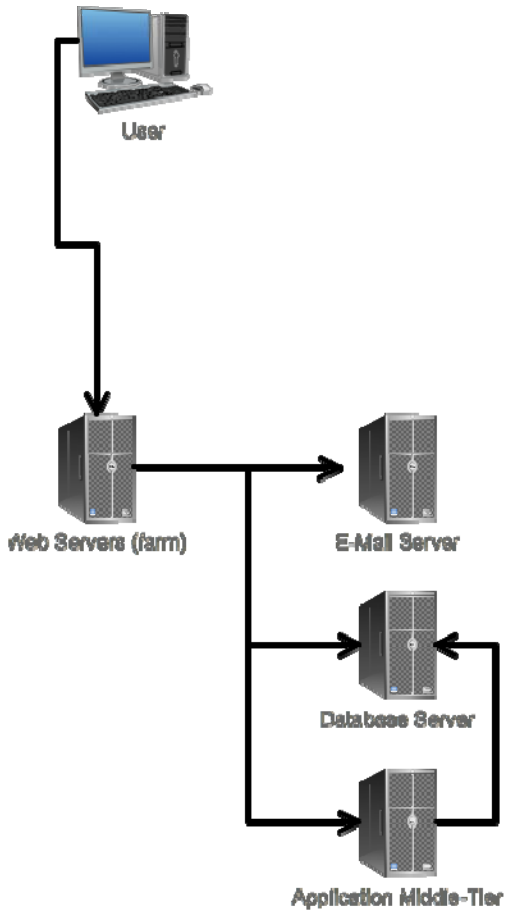

Figure 3.1: Basic application model.

But there are definitely key components missing. For example, this model doesn't reflect the hardware infrastructure that connects all of the components—and those are certainly important elements of the application. Handing the map to a network engineer might reveal additional information about the application, as Figure 3.2 shows.

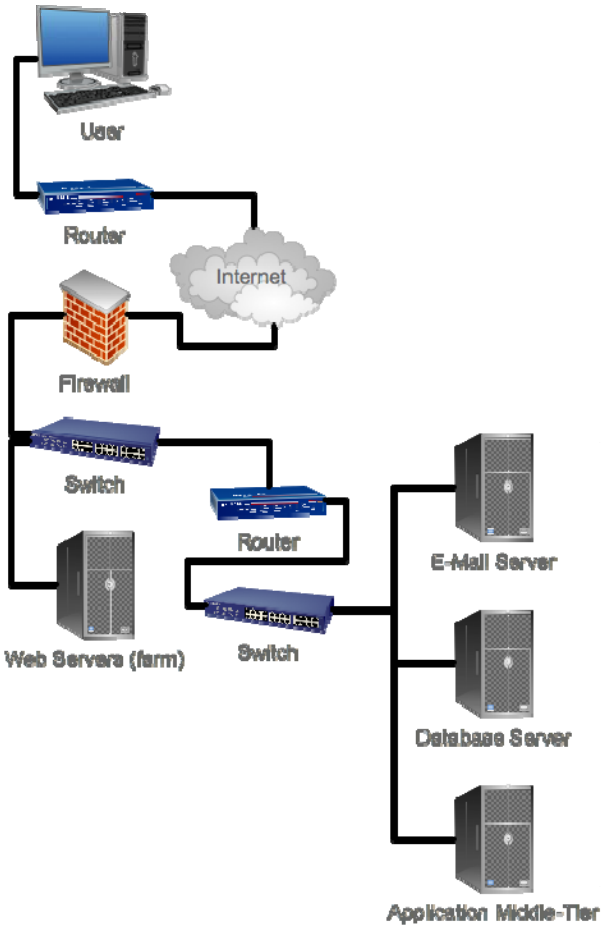

Figure 3.2: Extended application model.

Now, the physical infrastructure is starting to come into play. But even this model doesn't necessarily reflect all of the application's dependencies. The email server, for example, will need access to a DNS server in order to send email. Incoming connections to the Web server may actually pass through a load-balancing device. In order to really get an accurate picture of the application's performance, we need to consider all of those components, as shown in Figure 3.3.

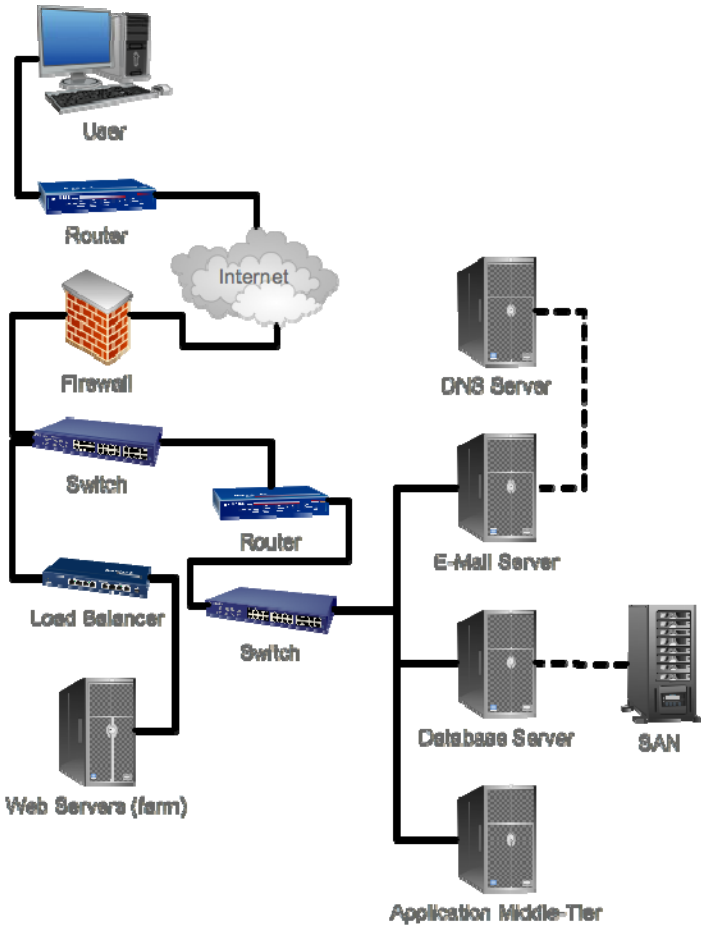

Figure 3.3: Further completion of the application model.

But even this model isn't complete. It focuses primarily on physical assets—servers, routers, and so forth—with no knowledge of the component-level breakdown of the application's software. For example, as Figure 3.4 shows, the Web server might consist of Microsoft's IIS Web server, running PHP Web scripts, which in turn instantiate Java EE components that use a custom driver to access back-end resources. The other servers contain their own discrete software components; I've "dimmed" those in Figure 3.4 to indicate that the focus is on the Web server software. All of these software components are important, all of them contribute to, or detract from, the overall application's performance, and all of them need to be included in the application model.

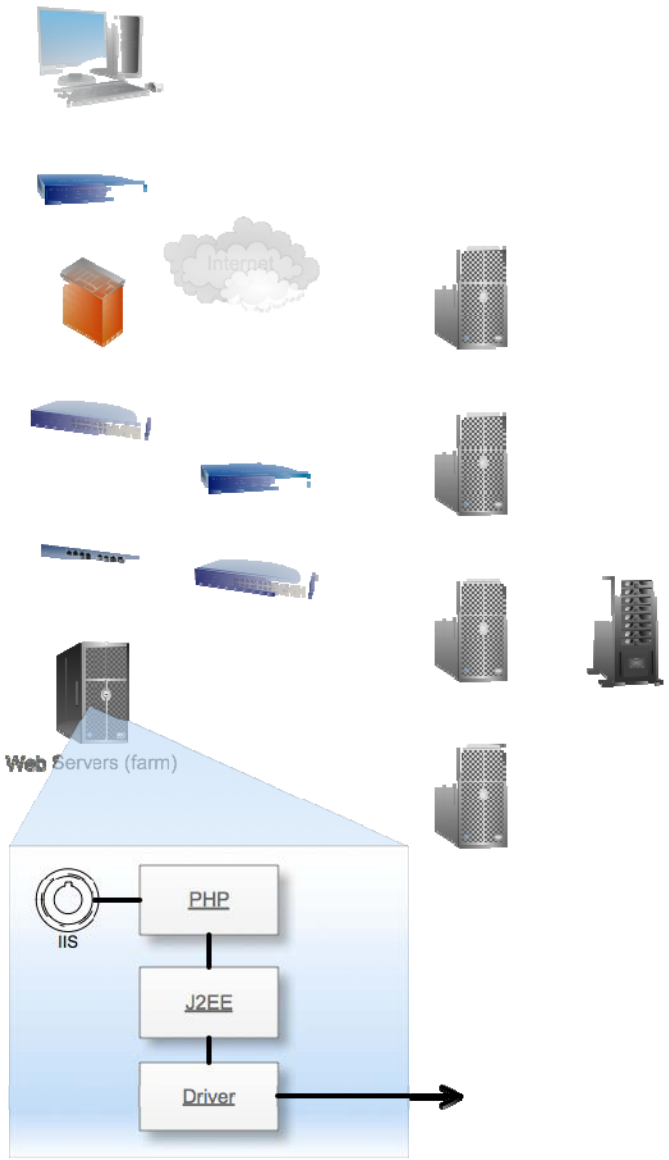

Figure 3.4: Adding software to the application model.

The point of all this is to illustrate how difficult it can be to manually create a complete application map or model, and how easy it is to overlook individual elements. Without a complete and accurate model, however, it can be very difficult to troubleshoot performance problems. Suppose, for example, that your model only includes "the Web server," without recognizing the breakdown in software components that comprise that portion of the application. When a performance problem arises, you might be able to track the transaction and determine that it was stalling "in the Web server," but unless you could narrow the problem further than that, you'd still be left with a lot of moving parts to try and fix. Without understanding the email server's dependency on a DNS server, you might be fooled into thinking that "the mail server is slow," sending your troubleshooting efforts off in a particular direction—when in fact the problem might simply be that the mail server is *waiting* on the DNS server, which might be closer to the root cause of the problem.

An accurate and complete model is, therefore, absolutely critical. That's one reason Gartner identified application discovery and modeling as a standalone dimension in their five-dimensional model: This dimension is critical to the success of your APM efforts.

Techniques for Discovering and Mapping Application Components

Many of today's APM solutions that implement this third dimension capability do so through automated discovery mechanisms. In other words, the APM solution actively and passively seeks out application components, which may include switches and routers, servers and operating systems (OSs), software, configuration files, log files, business applications, and more—including dependencies of these. Some solutions may require agents to be installed throughout your application infrastructure, while others operate "agent-free" and collect information through a variety of "sensors," or by using standard protocols such as Secure Shell (SSH), Windows Management Instrumentation (WMI), Simple Network Management Protocol (SNMP), and more.

The actual techniques used by different vendors can vary significantly, and the techniques a vendor uses should be a major factor in your evaluation criteria. It's impossible for me to document every technique used by every vendor, but I can provide some examples to help in your solution evaluations.

Passive IP Mapping

Passive IP mapping is the least-intrusive technique used by most vendors. It doesn't require you to provide any kind of administrative credentials on your network. Instead, it does basic IP address discovery to map out the major hardware devices, including servers, in your network. In a heavily-secured or -firewalled network, this technique's efficacy might be limited. In some cases, the discovery might be able to determine server OS versions. This will produce a fairly high-level map, perhaps with as much detail as shown in Figure 3.5.

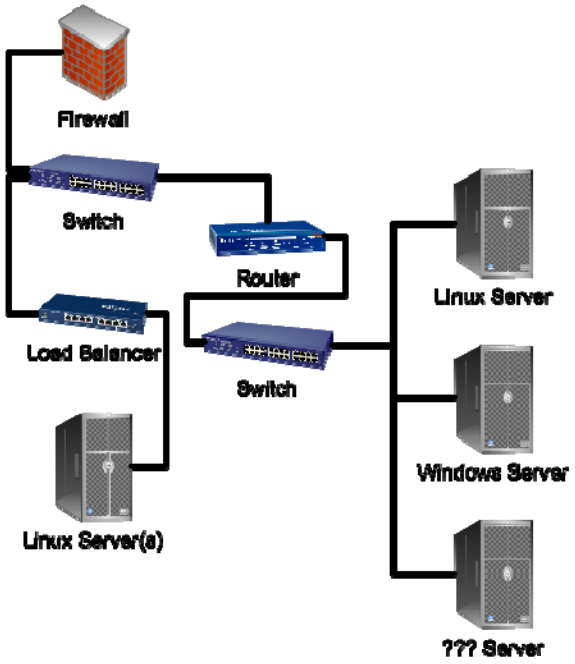

Figure 3.5: IP device map.

Shallow Authenticated Discovery

The shallow authenticated discovery technique is more active, and typically requires that you provide administrative credentials to the discovery software. Given the IP-based map it has already generated, the software will attempt to log into each device, validate the OS version on each device (this may include OS information for intelligent network devices, such as routers), and in most cases attempt to discover the processes running on each server. You probably won't get a detailed look at application-level information, but by relying on a library of known OSs and processes, you can get a great deal of information about the overall environment. Figure 3.6 shows the level of additional detail that you might see—primarily what each server is being used for.

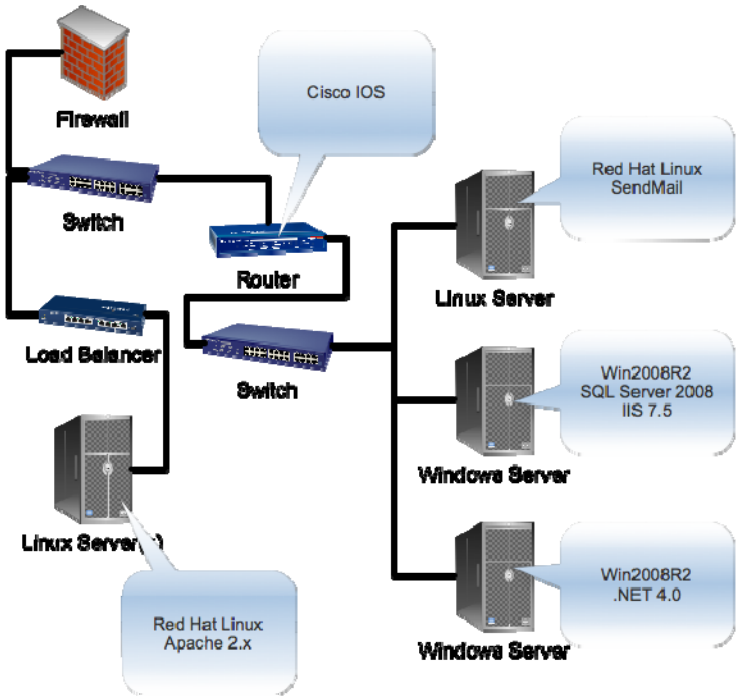

Figure 3.6: Discovering more detailed application model information.

Detailed Application Discovery

This level of discovery is much more detailed, and often relies on the discovery software's specific knowledge of specific application stacks. For example, the software might analyze each process running on a server in an attempt to discover whether it is written in Java or .NET, and from there might trace specific dependencies. The solution may be able to discover which databases an application is accessing, what network segments exist within the application model, and so forth. When finished, the solution may be able to present a more complete, visual application map or model, such as the one shown in Figure 3.7.

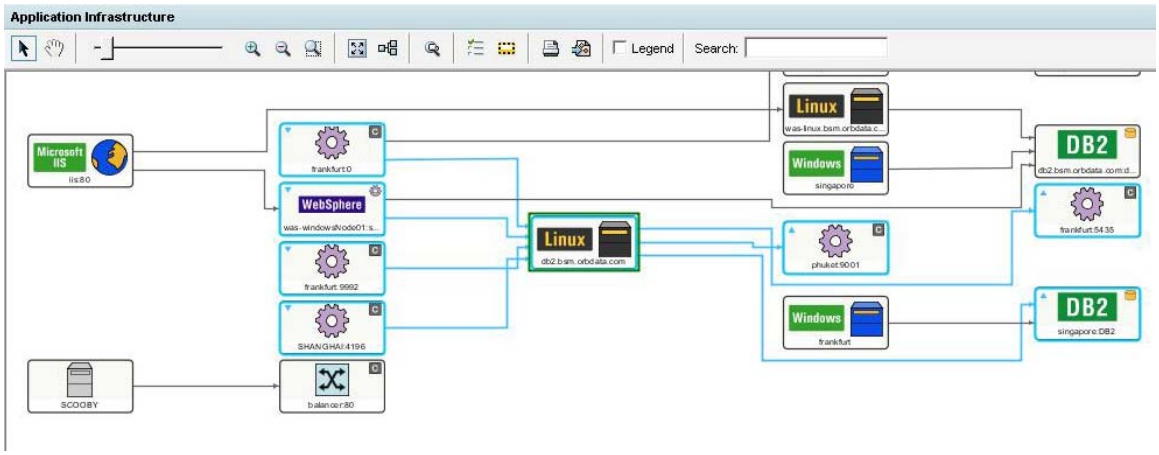

Figure 3.7: Completed application model.

As Figure 3.8 illustrates, these visual models can become quite complex. As a result, these discovery solutions will often include tools for automatically arranging elements of the model into an orderly diagram, zooming in on particular elements, and so forth.

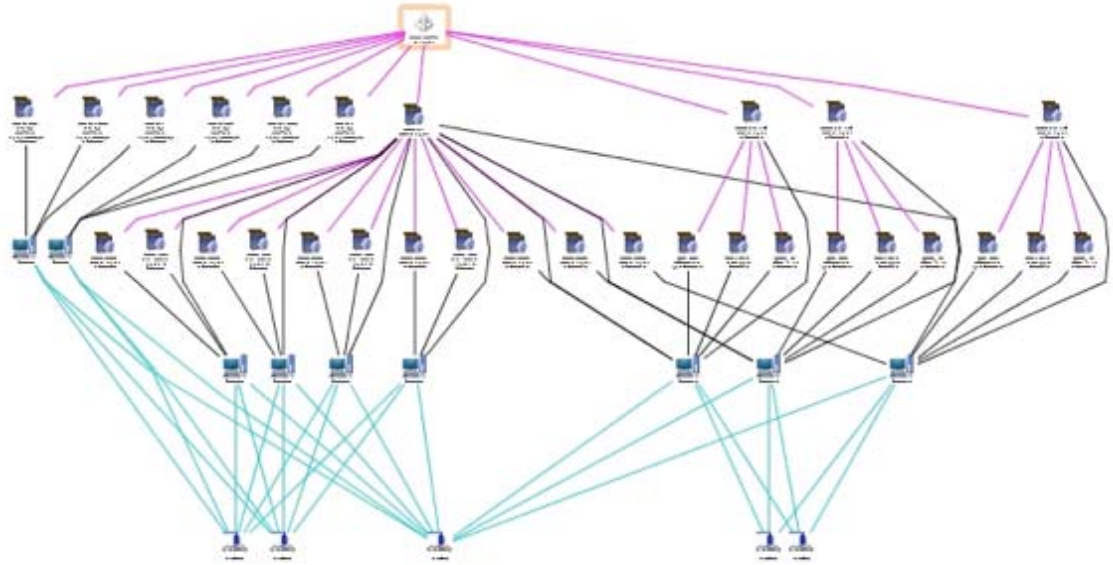

Figure 3.8: Complex application model.

It's important to note that this model doesn't just exist as a diagram. Most APM solutions actually store this information in a configuration management database (CMDB, sometimes also called a *performance* management database, or PMDB), which is an integral part of the overall APM solution. The diagram is actually generated from that database. The database is what enables the APM solution to keep track of each application component, and is what enables the other elements of the APM solution to measure, monitor, and report on each application component individually.

Adding Common Sense: Making the Best Application Map

The automatic discovery features in many APM solutions can often only serve as a starting point. For example, consider Figure 3.9, which is a zoomed-in view of an automatically-generated communications model between numerous servers. Near the middle-right of this diagram, you'll notice some "super connected" servers, which were observed, by the discovery tool, to engage in communications with a great many other nodes in the network.

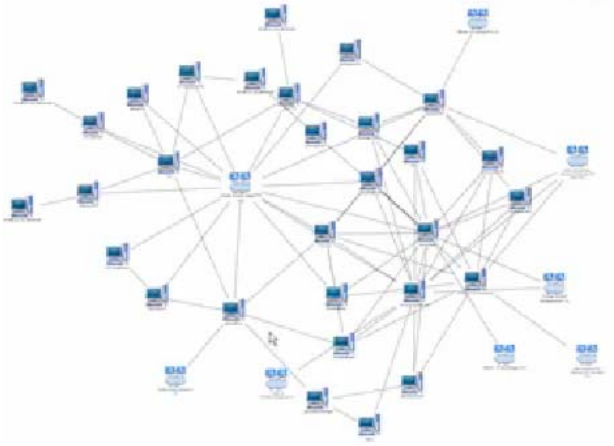

Figure 3.9: Reviewing the automatically-generated model.

This is where your own common sense and knowledge of your application come into play. You may be able to determine that much of this observed traffic has nothing to do with your application. For example, if an email server forms a component of your application (perhaps for sending emails to customers), the discovery solution may also observe a great deal of email traffic that is *not* related to your application. Such traffic might influence the model that the discovery tool creates, and you might need to go in and manually modify it to remove unrelated connections. As Figure 3.10 shows, you might then manually identify the servers that are, in fact, a part of your application, and instruct the APM solution to ignore the other connections that it discovered automatically.

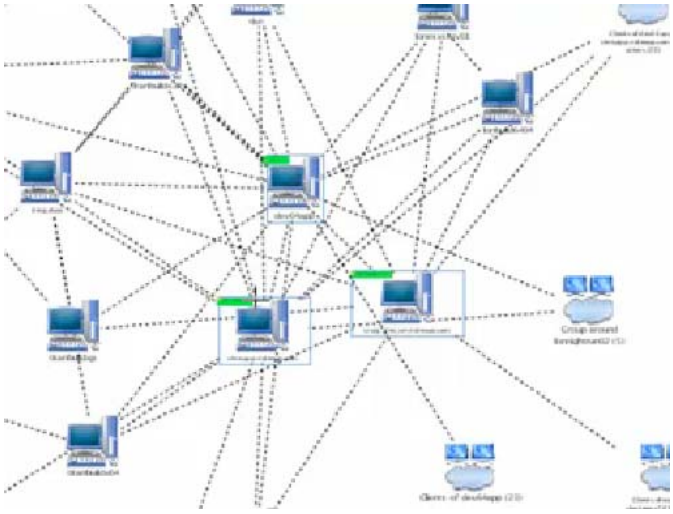

Figure 3.10: Manually modifying the application model.

Applying your own knowledge of the application and its infrastructure is an important final step in the discovery and modeling process because automatic discovery cannot always be 100% accurate. In most cases, the automated discovery will create a good starting point for you. It will often "over-discover," meaning it will often identify components unrelated to your application. That's actually preferred over "under-discovering," which would lead it to *miss* critical elements; you can always go into an "over-discovered" model and manually tweak it to more accurately represent your application.

> **Terminology**
>
> I'm accustomed to referring to this kind of diagram as an *application map,* although the term *application model* is probably more accurate because the APM solution is actually defining a model in the database and then generating the visual map from there. You'll also see the term *service model* used by some vendors.

It's also possible for automated discovery techniques to *miss* components, especially in a tightly-secured network or in an application that relies on external components such as Software as a Service (SaaS) solutions, cloud-based elements, or other remote components. Again, your knowledge and common sense are an invaluable part of the modeling process because you can identify those missing components and manually add them to the model.

Identifying Key Component Metrics

Once you have identified all of the components on your network, you need to start identifying the individual performance metrics for each. This will obviously differ based on the component. For example:

- For a server, you might look at metrics such as processor utilization, memory utilization, disk throughput, and network throughput.
- For Web server software, you might look at request queue depth, requests per second, faults per second, and so on.
- For a Java EE software component, you might look at processor and memory utilization for its process as well as metrics like transactions per second, network throughput, and so forth. You might also look at Java-specific details, such as garbage collection.
- Hardware components like routers might be measured on overall network throughput, dropped packets, overall utilization, and so on.
- Database server software might suggest metrics such as the number of locks, the number of transactions per second, various memory cache utilization rates, processor utilization, and so on.

You need to be a bit careful about choosing metrics. For one, you don't want to have so many that you wind up staring at a huge mass of data that you can't make any sense of. In general, I like to choose metrics that meet these criteria:

- **Easily measurable.** I don't like metrics that require a great deal of runaround in order to collect.
- **Low-impact.** Some metrics can create their own load on the measured component, which is often counterproductive. I prefer metrics that can be remotely gathered with minimal impact on the monitored system.
- **Actionable.** I like metrics that can point to a specific root performance cause. For example, measuring CPU utilization isn't always useful because it isn't always something I can do anything about. Sure, it tells me that the CPU is overloaded, but it doesn't tell me *what* is overloading it. If a particular metric doesn't lead me to a particular corrective or further-troubleshooting action, that metric is automatically less useful to me.

The good news is that many APM solutions contain a great deal of built-in intelligence about what should be monitored. For example, consider Microsoft's System Center Operations Manager product (which isn't an APM solution but does share some general high-level characteristics with one). It's designed for monitoring servers, and not applications, but the point is that it can be extended with "Management Packs" that tell the product *what* to monitor for specific types of servers, such as Microsoft's database server, messaging server, and so on. This "intelligence" or "knowledge" helps identify desirable metrics for you so that you don't have to do so manually. A good APM solution can accomplish the same thing for an application: identifying important metrics for various components so that you know what to monitor without having to sit down and figure it out for yourself.

That built-in knowledge can be *especially* useful when your application includes entire third-party applications, such as Oracle's E-Business Suite, Siebel applications, PeopleSoft applications, and so on. Monitoring those can be an entire science unto itself; by having an APM solution that understands those and already knows what metrics to look at, you can save yourself a considerable amount of time, trial and error, research, and effort—and start monitoring your overall application more effectively right from the start.

Assigning Component Thresholds

Once you—or your APM solution—have identified the metrics to measure, you need to start defining thresholds for them. Think of these as mini-SLAs; each metric should have values that tell you "everything is OK," "things are starting to look bad," and "things are definitely not good." When you start to see poor EUE metrics, diving into the individual component-level metrics can help you quickly spot the root cause, or at least identify components that require further, more detailed troubleshooting.

Typically, an APM solution will "roll up" lower-level metrics into a top-level component indicator. For example, let's take the example of a VMware vSphere or ESX server infrastructure. Figure 3.11 shows what the top-level view might look like.

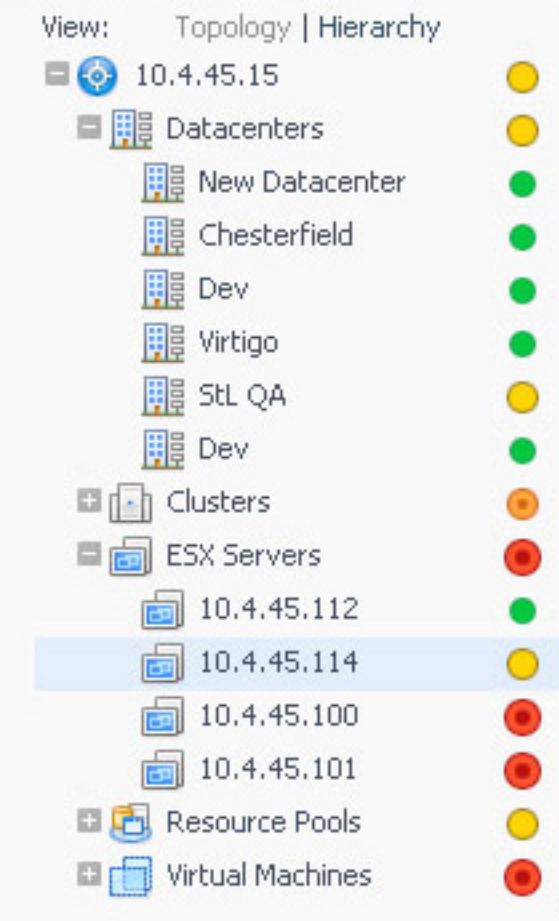

Figure 3.11: Top-level view of the virtual infrastructure.

Here, we're given green-yellow-red indicators that help quickly draw our attention to problem ESX servers; we're also given an alternate view that focuses on each of our physical data centers. When we see a yellow or red indicator in this level, we can dive slightly deeper, as shown in Figure 3.12.

In this deeper-view dashboard, we can monitor a variety of key metrics, and quickly see which ones are leading to the yellow or red indication for the server as a whole. In this example, we're seeing some swap measurements that are above our "everything's okay" threshold.

Figure 3.12: Monitoring an individual ESX server.

An APM solution might offer an even more detailed drill-down, such as the one in Figure 3.13. This view breaks down each component of the server, helping us spot problems with (in this case) CPU utilization, processor queue length, number of processes, disk utilization, disk throughput, and so forth.

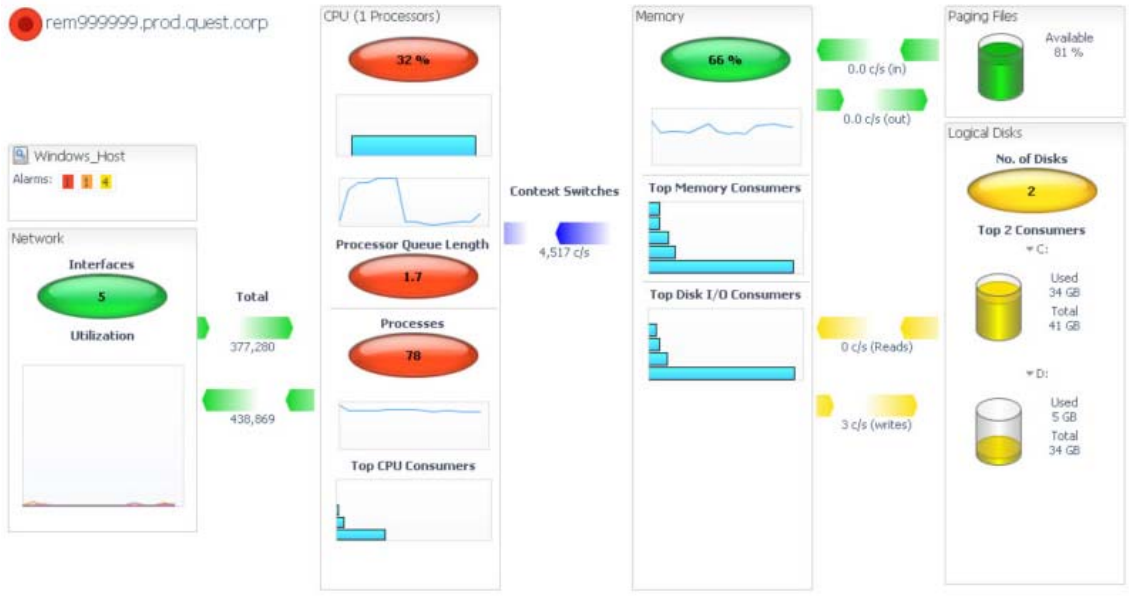

Figure 3.13: Server-level performance drill-down.

Keep in mind that this example is specific to a server; ideally, an APM solution can provide this kind of detailed monitoring for a variety of application components, which might include network hardware, applications such as PeopleSoft, Web server and database platforms, line-of-business applications written in Java or .NET Framework, and so on.

In addition to helping you identify the metrics to measure, a good APM solution can save you time and effort by helping to define threshold values for those metrics. In other words, if the solution already "knows" that specific performance levels equate to good, declining, and bad performance, you can start accurately monitoring your application that much more quickly and effectively.

Create a Response Plan and Assign Component Responsibility

APM solutions can only point you toward the cause of the problem; they can't generally fix it. The next step in your application modeling, then, is to assign responsibility for each component to a specific person or group within your IT team, and to define a response plan for when performance is declining or has gone bad.

The following example highlights how this might work in an ideal environment, starting from the first dimension, the EUE:

> Charlie receives an email alert from the APM solution, indicating the EUE metrics for a key type of transaction may have fallen below the acceptable SLA. In other words, customers are starting to see poor application performance, and something needs to be done.
>
> Charlie fires up his APM console, and sure enough, the application has a yellow alert, meaning performance is degraded but hasn't yet missed the SLA. He double-clicks to drill deeper, and is presented with an application map that displays the various elements of the application. A database server is yellow-flagged, indicating that it is starting to fall below its performance threshold.
>
> Rather than immediately notifying a server administrator or a DBA, Charlie drills one level deeper by clicking on the server. The database server software's metrics are all in the yellow—it's handling significantly fewer transactions per second than it normally does. The server hardware looks fine except for disk throughput: that metric is also falling. Charlie decides to notify both a server administrator *and* the DBA because he can't determine exactly what the cause is.
>
> The DBA looks at the same APM console, and with his domain expertise is able to determine that queries are taking too long to complete due to unusually poor disk response times. He consults with the server administrator, who sees that the Storage Area Network (SAN) backing up the database server has much lower throughput than normal. Further investigation reveals that disks in the SAN have failed, and that the system is operating at reduced performance by relying on redundant disks. The administrator replaces the failed disks, and notifies Charlie that application performance will be degraded until the redundant disk array has rebuilt itself.

That's the kind of scenario you want to see when performance fails: a quick drill-down into the root cause component, and quick notification of the domain expert or experts who can deal with that specific component. The key is to know *who* is responsible for each component, and for those responsible to have a plan for what to look at and how to troubleshoot specific problems. That is, when metric "A" is out of whack, troubleshooters should have an action plan to further troubleshoot that specific problem or to fix the problem.

The Advantages of Application Modeling

That kind of rapid, directed response is a direct result of having a good application model driving your APM solution. Here's the opposite:

> Charlie receives an email from someone in customer service, who explains that customers are having a hard time completing their online checkout. Apparently, the system seems to hang and time out, or at best is very slow.
>
> Charlie notifies the entire application team: server administrators, Web developers, DBAs, network engineers, messaging administrators, and so forth, informing them of the problem.
>
> Everyone goes into "fire-fighting" mode and starts troubleshooting their specific piece of the application. The entire team is disrupted.

That kind of shotgun approach is not productive. First, by not having the first dimension EUE metric, we had to wait for someone to tell us that the application seemed slow. By not having the second-dimension ability to track individual user transactions, we weren't able to pin the problem down to a general piece of the application—instead, it was the entire, multi-component application that needed troubleshooting. By not having the third dimension's application model, we couldn't pin the problem to a specific component and alert *just* the experts who could troubleshoot and fix that component.

The application model, by giving our APM solution an idea of what the application *looks like,* can help drive a very directed, measured response to performance problems. It can do so *before* performance problems become noticeable to customers and end users, and it can do so in response to a problem with the EUE—again, before actual end users may perceive a significant performance problem.

Cautions

As you begin looking at tools that can provide APM capabilities, you need to dig pretty deeply into how they're built. There are a few main things you want to be careful of:

- **External dependencies.** Some vendors build their tools based on *other* vendors' tools, and that can be especially true for application discovery and modeling. Having an external dependency isn't necessarily a problem, but it is a risk. For example, if the vendor who supplies that external dependency is purchased and their part of the solution is no longer available, you could find yourself with a less-functional APM solution. Your preference should be to purchase an APM solution from a vendor who can provide all five dimensions in a single solution set.

- **Lack of breadth.** You typically want to avoid solutions that are built entirely for a single kind of application stack, such as Java or .NET Framework. Ideally, your APM solution will be cross-platform, supporting multiple common stacks rather than focusing on a single one.
- **Lack of intelligence.** The more work you have to do to manually model your application, define metrics, and set thresholds, the longer your APM deployment will take and the less effective it will be. Look for APM solutions that take as much work off your shoulders as possible; there will still be plenty of work for you to do, but the more that comes "built-in," the faster you'll be able to take advantage of the solution's capabilities.

Coming Up Next…

So now you have an accurate electronic model, or map, of your application. What do you do with it? Generally speaking, *dive deeper.* In other words, when a problem occurs with your application's performance, you'll want to start diving into individual components to find out where the problem is beginning so that you can start implementing a fix. That's what the next chapter is all about: the fourth dimension of APM.

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.