Jon's Performance Musings: CYA

As one moves toward Application Modernization and/or Service Oriented Architectures (SOA) the service-aware software professional needs to remember CYA: Cover Your Aggregation. The service your application provides is directly dependent on many factors, not the least of which is the service received from other services. It can be said that the service that you're providing is based on the aggregation of the services that you call. If you pay attention to that, you can save yourself lots of aggravation.

Greater Than The Sum Of The Parts

Modernization and SOA tend to break up an application into lots of parts. These parts may be spread around the Enterprise, out to external partners, or around the Internet. The good news is that interfacing to these services may be quite easy. Enablers like SOAP, CGI, scripting languages, WSDL and other interface definitions, etc. hide complexity and layers of software and communications. They may be so easy to use that the designer and developer forget that each instance contributes to the service time and overhead of an application.

Trust, But Verify

Application modernization and SOA is becoming a fact of life in enterprises. There is no turning back. To remain competitive one has to embrace the agility and functionality of these techniques. Yet, those of us in the high-availability, high-throughput, high-visibility transaction arena recognize that quality of service must remain our mantra. To that end, we must be able to understand and quantify the contribution that called services make to the service we're providing. We need to measure the service we're receiving, and confirm that those external services are meeting their SLA commitment to us.

Clock In, Clock Out

This is something the payments industry has always been on top of. With money at stake, and with lots of external partners helping process transactions, it is mandatory that a well-run ATM/POS system or switch quantify the service it's receiving. Timers have always been part of the application anyway, since there are strict rules about when transactions time out. A well-designed application records those timers as part of each transaction's record. Usually several times are recorded:

- Overall time, from when the transaction is received to when it completes processing and is returned to the originator;
- Authorizer time, from when the transaction is sent to the back end for processing and when it returns;
- Encryption time, if required;
- Queue time, if the application has to queue the transaction;

- Database time, if the application uses a DBMS.

Time The Enterprise

SOA environments will have lots of services, and also major DBMS systems. While I'm sure that these are well run in your organization, the nature of IT guarantees that there will be periods of peak demand which may, repeat may, result in slower response. As a service provider and a customer of other services you owe it to yourself to understand and measure everything you are depending upon. In SOA, this could easily mean lots of services: Authentication, CRM, app-specific services like order creation, inventory management, finance, credit, shipping, etc. Measure each of them.

Now That You've Got It, Review It

Collecting data without using it is a meaningless exercise. You have two choices: Use the data for "post-mortem" analysis, which is useful but a waste of potential. Or, use the data to anticipate problems and discover worrisome trends and tune the services so that no one suffers.

Canary In The Coal Mine

Like the canary in the coal mine, the service times you've collected can help predict a problem before it hurts your transaction, or before your boss calls you. But it requires that you *look* at the data periodically. I've found that most of the time the Heinrich Ratio holds true: There are always a set of events which are precursors to major problems or outages. If anyone had been looking they could have clearly seen the oncoming crisis. So look!

Use Response Distributions

Figure 1 shows you a very effective technique for presenting response times, and what they may show. It shows an 18-month history, by day. This requires reporting only one summary row each day, and charting it. The key here is that the data shows both the transactions by day, and the response distribution for that day. NOT the average response time. The response distribution in this case is:

- Red line: Percentage (right scale) of transactions completing in less than 2 seconds;
- Green line: between 2 and 5 seconds;
- Blue line: between 5 and 10 seconds;
- Purple line: Over 10 seconds.

So this chart shows that something changed in early January. The 2 second line dropped from over 95% to less than 95%, and the 5 second line jumped to almost 10%. Why? What has changed in the environment? Is it our system, or is it a service or back-end that we are calling?

The Buckets

When you write the report program to summarize the response times, I suggest that you select your buckets for the response distribution as follows:

- Ideal response
- Good enough response
- On the verge of broken
- Broken

As you plot this data each month, you will want to focus on the latter categories. Are they changing? Why are they changing? Is it a steadily worsening situation? Will it eventually start to break transactions? And for the Enterprise SOA service, "Who else is affected?"

The Detail

Figure 2 shows the service or back-end level response detail. This is the second level of plot you will need to do of the data which you've been collecting, if you're tracing a problem. This shows you by time of day what kind of response you are receiving from the services you are calling. This particular chart shows an extreme problem, since the mythical "bank1" is giving us terrible response every weekday around 9 a.m.

In an ideal system, a system with zero bottlenecks and zero queuing, every line on this chart would be flat. Ideal systems don't exist. To state it another way, there is always the "next bottleneck" in a system. You may simply have not hit it yet.

CYA: Covering Your Aggregation Will Save Aggravation

A properly-designed application, mindful of the services that it depends upon, can provide sufficient information for its management to predict problems before they become expensive. Doing it right will CYA.

Jon E. Schmidt
Transaction Design, Inc.
San Rafael, CA, 94901, USA
1.415.256.8369
inform@banbottlenecks.com

Jon is the founder of Transaction Design, Inc. (TDI), a consulting firm located in the San Francisco Area which specializes in quality of service studies with clients worldwide. He is the creator of the Ban Bottlenecks® service and has an extensive background in the implementation, testing, and tuning of high-availability systems.