lessons

DANIEL ROGERS, MICROSOFT

the January monthly service quality meeting started normally—around the table were representatives from development, operations, marketing, and product management, and the agenda focused on the prior month's performance. As usual, customer-impacting incidents and quality of service were key topics, and I was armed with the numbers showing the average uptime for the part of the service that I represent: MSN, the Microsoft family of services that includes e-mail, Instant Messenger, news, weather and sports, etc.

Running a service of this size, with thousands of servers behind the scenes, makes it hard to boil performance down to simple averages. Managing a service that provides news, weather, stock quotes, instant messaging, blogs, and hundreds of millions of e-mail accounts by aggregating uptime percentages has proven to be less than effective because a lot of detail is hidden behind the numbers. The customer service calls we handled during the month represented just over .004 percent (that's right, four one-thousandths of a percent) of the overall service provided—so is it a problem or not?

The better question is, "Percent of what?" Month-over-month comparisons were not enough. We needed to drill into the numbers to get a better understanding of exactly what it means to manage and measure a large-scale Internet service.

PERFORMANCE MEASUREMENT CHALLENGES

The MSN service generally consists of three server roles: front-end Web servers, middle-tier transaction servers, and back-end storage servers. Different components of MSN may use all or only some of these roles. In the case of the three-tiered services, the Web server talks to the middle-tier servers using a variety of protocols depending on the needs of the service. Hotmail, for example, uses an e-mail-oriented message protocol with each message defining an atomic unit of work against the e-mail stores. Load balancers handle the extremes of user traffic. A typical Web page refresh results in a number of sequenced requests from the Web servers through the middle-tier servers. In general, we try to keep no session affinity between a Web server that is performing a given page paint and the middle-tier servers that manage the transactions against the underlying data stores.

In this arrangement, reducing availability to a single number that represents quality of service gets difficult. The solution that generates performance as averages involves creating test passport accounts and using automated agents to simulate logging in and exercising the features of the service component. Each of the components that make up the larger MSN (such as MSN Messenger or Hotmail) has dedicated agents for measuring service quality. The first numbers problem, therefore, is how to manage the sheer number of test accounts

The manufacturing industry

can teach us a lot about

measuring performance in

large-scale Internet services.

from the floor

lessons from the floor

required to cover the many parts of the MSN service adequately. Since agents need to catch issues on each of the possible stores that manages user-specific data, a good practice is to have a test account on each partition. That adds up to thousands of test accounts. Getting a feel for how well the various parts of the MSN service are running means scaling the number of test user accounts and increasing the number of simulation sessions to cover all parts of the service.

Over time we discovered that a measurement infrastructure based on simulated sessions and test accounts becomes a scalability problem all to itself. For a quick look at why this is the case, consider that each partition holding user data (for example, a disk for holding a user's e-mail, messenger buddy list, stocks, or even color preferences) means hundreds more test accounts distributed on each of the stores involved. Why is this difficult? Here are a few reasons:

- Test accounts need their passwords expired/cycled monthly.
- Session simulators need to be updated when new stores and test accounts are added.
- More and more simulation servers need to be added to handle the simulation session load.
- As the service grows, false negatives become a problem when failures in the simulation infrastructure trigger problem reports against the service (you have to monitor the monitors).
- Finally, simulation doesn't provide visibility into the effect that any failure in any particular part of the MSN service layer has on the end customer experience, nor does it indicate if any specific transaction types are more problematic than others.

We concluded that the session simulators are valuable for troubleshooting when something isn't working, but as the service scales up, they are not the best way to measure service availability. Determining the aggregate quality of the service being provided requires more in-depth measurements at the unit transaction level.

BORROWING FROM THE MANUFACTURING INDUSTRY
The manufacturing industry faces quality-sampling challenges for complex products that are similar to ours. In Web services, we have the advantage of being able to simulate a customer interaction simply and cheaply (on

a small scale, at least). When it comes to measuring the quality of a product that has many facets, however, trying to define one aggregate measure to represent quality fails as an approach. In Hotmail, we looked at large-scale manufacturing operations for inspiration—and came away with several useful techniques that better matched our business goals.

Consider the manufacture of any mass-scale complex engineered product, such as a car, motorcycle, or small appliance. Customer satisfaction—the gauge of quality—is multifaceted. Manufacturers need ways of quantifying the ways that customers make judgment calls. A customer may say, “This toaster was a disappointment.” Was the most important facet causing that disappointment the consistency of the color of the paint on the outside of the toaster? Was fit-and-finish (the way the product feels when used) a factor? Or was the ability to make more than one piece of toast, with uniform quality, breakfast after breakfast, the determining factor? The take-away message is that complex products have complex quality goals. For a Web-based e-mail service such as Hotmail, we need ways of determining the priority that customers place on a number of service quality measurements.

In Hotmail, for example, fit-and-finish translates to look and feel. Consistency in execution (the toast) translates to how long it takes to log in, find your new e-mail, delete any unwanted messages, read messages, and log out. If it takes too long, or behaves inconsistently, the customer gets frustrated. If customers experience erratic behavior from their browsers—for example, if opening e-mail is fast sometimes but slow at other times, or if one operation is markedly slower than others—the customers' perception of service quality takes a nosedive. Customers will go to another service if they have too many bad experiences. MSN marketing studies suggest that long response times, as well as inconsistent responsiveness, are key contributors that customers identified as leading them to be dissatisfied with their overall MSN experience.

For services that consist of several layers, such as those based on three-tier architectures, measuring and achieving consistency isn't straightforward. The programming teams that are responsible for a given tier, for example, can control only how well their code behaves. The user experience, however, is a result of all of the layers working together. Thus, for the team that manages the servers

that provide access to user-specific data (such as e-mail or buddy lists), the timing of individual transactions and the consistency of that timing are important to providing a predictable user experience. The team that runs the Web servers traditionally took on trying to manage user experience, but layers lower down in the call stack, as well as components such as load balancers and network latency, all have an impact. Measuring the impact that a given layer has on overall service levels, therefore, requires partitioning the measurements across all of the layers.

STATISTICAL SERVICE MANAGEMENT

Let's look at one of MSN's larger services as an example. In Hotmail, the first cut at gathering statistical measurements involved identifying the transactions that affect customer experience. We identified 32 important transaction types on the initial pass. We wanted to distinguish measurements that differentiate a good experience from a bad one at the transaction level, and avoid introducing uncertainty in the measurement. After some discussion we realized that the difference between good and bad happens in gradations, not in a binary yes/no manner. Given the complexity of the system, we expected some degree of randomness (different accounts, different stores, different levels of traffic at any point in time) to be a part of the normal measurements.

By studying a technique known as SPC (statistical process control) used in manufacturing, we learned that the distribution of good to bad would be normal (a bell curve). Some results would be very, very good; others, hopefully few, would be out of tolerance, or bad. The majority, however, would fall into a normal range. To get the level of granularity required, we started looking at how to capture the measurements. Since the primary user experience indicator we focused on was consistency in timing, we started with measurements of transaction time—and ran into our first challenge: handling the volume of timing data.

For a service such as Hotmail, which can easily exceed a billion user-driven transactions a day, measuring timing data for every

transaction quickly adds up to a very large data management problem. The challenges that we faced with simulated transactions quickly became apparent once more. The main questions were:

- How do we capture, sort, classify, and present the data from so many transaction data points? What kind of processing is required to reduce the raw data to a meaningful form?
- If we resort to sampling, how few samples do we use and how do we avoid introducing too much uncertainty as a result of our sampling?
- Would sampling be done randomly? Or on a round-robin basis? Would sampling be driven by simulated user transactions?

In the end we decided that the best solution would measure every transaction, but to do so we had to store the data in a directly meaningful form. That meant making the data available for collection.

After looking at performance counters in Windows NT—a good first place to look—we realized that they weren't suitable for exposing graduated measurements. To represent five gradations from great to awful, we needed to create five counters for each measurement we wanted to expose. That is a lot of added complexity. Further, our experience with custom performance counters revealed that they need to be re-created every time the system is rebooted. Using performance counters to expose this data means that services need startup routines. For implementations based in NT Services (the equivalent of Unix daemons in Windows NT), pre-service startup routines add complexity. This added complexity makes them

fragile and error-prone for exposing measurements: If a problem occurs in the startup scripts that create the performance counters, the service cannot be measured and may actually stop unexpectedly because its required performance counters are not present. These two factors made NT performance counters a bad choice. The solution we ended up with is something we call performance curves.

A performance curve is the result of accepting that for any given transac-

**We realized that
the difference between
good and bad happens
in gradations, not in
a binary yes/no manner.**

lessons from the floor

tion type, for any given period of time, there will be a predictable distribution of behaviors. In most cases, this will be a normal distribution. Certain caching strategies can introduce bimodal distributions as well (such as caching search results—cache hits will show up as one peak, and cache misses as a second peak in the measurement). Individual performance measurements aren't interesting by themselves because of the inherent randomness in any complex system. When viewed alone, you can't tell whether the individual measure represents a problem or not. If we define ranges of values that represent different categories of behavior—such as extremely good, very good, good, poor, and very poor—and then count how many individual transactions fall into each range over a fixed period of time, then the result is a small data sample that directly measures the behavior of a transaction type.

Figure 1 shows an example of a performance curve representing the aggregate transaction timings of a transaction type called GetData for a five-minute period called a measurement interval. The graphical representation in figure 1 is a plot of the data that is captured from the software. Each plot is actually a set of measurements that count the number of transactions completed within each range over a fixed measurement interval. Each such set of related measurements represents a virtual plot that is self-consistent and represents the performance measurements for that transaction type for a period of time.

The individual data points represent counts that fall into a range of values—in this case, response timings.

As labeled in figure 1, one set of counts represents OK performance, whereas others represent good, poor, etc. To manifest these, the software that collects the data needs to know the timing boundaries associated with each range. These boundaries can be unique to each transaction type, and in our implementation they are based on settings in a configuration file that is loaded when the measurement library is loaded. This combination of collecting the counts within a time interval and then making different ranges that represent different performance categories results in a substantial compression of data, as compared with our test-account-based sampling.

By capturing one row of data consisting of the transaction type, the time interval, and the five performance ranges (buckets), one row of measurement data (five long integers and some key data) represents what would otherwise be a measurement per transaction. In the example shown in figure 1, therefore, one curve measurement represents the equivalent of 223,750 individual timing data measurements—which is the number of calls to the GetData transaction in a five-minute interval. That level of data compression makes managing a full sample strategy (100 percent of transactions measured) feasible. In some cases, we break down the data further so that we can identify the instance of a server that is gathering data. The data compression for a service component that handles more than 1 billion GetData transactions a day gets close to 85,000 to 1 at peak hour—a substantial compression.

When comparing the number of sample rows required to manage the MSN service with the amount of data from our traditional session simulators (one test account per logical store partition, with a 15-second traditional playback interval for session simulators), we found that the amount of data to be managed with performance curves compared favorably. When we factored in not needing to manage test accounts and adding more simulators as we added more stores, the decision was very simple: performance curves provided better data with better granularity for less cost.

SPEED—MAKING THE DATA AVAILABLE

One aspect to consider in deciding to measure every transaction is the overhead associated with the measurement. To be acceptable, the cost of driving the data to a

centralized store must not have a substantial performance impact on the application being measured. At MSN, we decided that there should not be a blocking behavior related to the storage of the measurement. This affected how we captured the curve measurement data in an instrumented application component; in the case of any failure or delay in the curve data infrastructure, we would rather lose the measurement than impact the caller.

In our 1.x production implementation, we decided that each of the collecting server types would cache several intervals' worth of data, and then periodically a background thread would write out the data to the collection store. This approach separated the storage of measurements from the collection of measurements. The storage thread then made a single nonblocking Web service call using one-way SOAP, completing the "no waiting" decision made earlier.

Figure 2 shows the logical collection pipeline. Individual measurements get bucketed (i.e., placed into the appropriate buckets). At the end of a measurement interval, the values associated with a data point are flushed to the measurement cache and then reset. A separate thread periodically flushes all collected measurements to the store via a one-way SOAP call and empties the cache of all of the measurements that were sent to the store. This is done on a flush-interval boundary, a time period that can also be configured.

By configuring the measurement interval and the flush interval, we can control how many rows of data (one per transaction type by collecting server) are generated overall, as well as how often we make the call to the Web service. This configurable combination allows us to scale up the solution nearly without limit, at the cost of granularity.

MAKING THE CONFIGURATION MANAGEABLE

After prototyping the approach, we noticed that since each of the measurement buckets represents a range of values, it is possible to mistakenly configure gaps or

overlaps into the configuration files. In the ideal case, each bucket represents a discrete range of numbers, and when you look across all of the buckets, every possible value falls into one—and only one—bucket in a contiguous fashion.

To simplify the configuration and make it impossible for value gaps to occur, we used an implementation of what we called "the egg sorter algorithm." This terminology goes back to a time when chicken eggs were sorted by rolling them down a chute with holes of different sizes along the way, arranged smallest to largest. Each egg would either fall through a hole along the way (thus sorting by size range) or slide off the end because it was bigger than any hole.

In software, given five buckets, we need four values to represent a contiguous set of ranges. Each number represents the upper boundary for the bucket such that any value less than the boundary value would fall into the bucket with that boundary. After four checks, the fifth bucket is incremented automatically. This approach not only simplifies coding by minimizing the number of comparisons, but also simplifies managing the configuration of boundaries for a large number of transactions by not having to worry about gaps and overlaps in the bucket configurations.

USING VARIANCE TO MAKE DECISIONS

The aim of putting this kind of measurement system in place was to increase the accuracy and visibility of the way the system works. A measurement system of this complexity enables us to make decisions on the data. The primary question when we set out in this direction was how to use this kind of data.

The interesting aspect of arranging data in discrete sets is that any given row of data, on its own, represents a complete piece of information. Compared with earlier techniques, such as automated session simulators that sampled by generating test transactions, the technique of collecting curve data based on every transaction handled

yields far more actionable and realistic data. Where we once had session simulators (which measured averages of the sampled transactions) and performance counters before them (which measured single values without correlation to time), we now had comparative data that

lessons from the floor

indicated the behavior of individual scale unit components relative to themselves. This feature of being self-contained is useful in determining the mix of behaviors that a given server farm is exhibiting—at one point in time, over a small unit of time, or even over larger periods of time.

In one implementation, we immediately noticed that transaction types that we broke out by message size were not behaving the same. In our implementation, we made it possible to measure the timing for small, medium, and large messages separately. This allowed us to address the different thresholds (good to bad) that different message sizes would naturally incur. The thinking is that a message with a 10-MB attachment should take longer to return to the requesting user than a 2-KB message with no attachments. This thinking drove us to add logic to make the size of the data being managed a factor in the bucketing step. Large-size messages are tracked separately from small ones, and different configuration thresholds are possible. The logic was generalized so that we could add size break-out with only a configuration change (no recompiles, please!).

With these capabilities, we are now able to analyze a day as a time-series for each measured curve/transaction type. The difference between the definitions of the term *transaction* and the term *curve* is a matter of the degree related, whether there is a size breakdown or not. Decisions can be made by analyzing the degree of variability in the data. If all is working well, the variance should be steady. The variance—a statistical measure that is the square of the standard deviation of the distribution for each curve—can be used to locate aberrant behaviors. A well-behaved system is predictable and, hopefully, achieves linear scale.

The assumption is that transactions on the same-scale unit should behave in similar ways near the same time. Thus, areas of interest pop out when you start to look at different spreads (more failures in one type of transaction than another) and when you look at the variability of the data over time. A stable system that behaves predictably may slow down under load, but a system that is showing wavering timings, or timings that display different variances by message size (normalized by well-chosen boundaries), can indicate optimization opportunities, or even identify slow or buggy code.

MEASUREMENTS IN CONTROL: A HAPPY ENDING

Since we implemented performance curves in several MSN service components, the monthly QoS (quality of service) meeting is no longer a game of guessing the meaning of averages. When a manager asks for a breakdown of the figures for our overall transaction component of QoS, we can show the breakdown in detail. Each slice of data is self-consistent, so that you can see the relative contributions of any of the dimensions in the performance curve data. Whether it is by scale unit, service component, transaction, or message size, we can show details that measure how well every transaction that passed through the system performed. As a result, our grasp on QoS from a customer perspective has improved.

Further, as in the case where we were able to identify the unexpected behaviors related to message size highlighted earlier, the added visibility gives us a better understanding of the impact of decisions related to both software and hardware architectures, and gives us data on which to base our optimization decisions.

Finally, from the perspective of the return on investment for adding the instrumentation to the service components, the implementations can take several months to put in place. Planning which parts of the system to measure, achieving buy-in from the development teams and business managers, and then getting a first implementation up and gathering data takes substantial investment. Making that data visible and coming up with the boundaries that you will use to determine your levels of service quality require a burn-in period. The decision about whether to aggregate the curve data for reporting or to load it all into a data warehouse is also going to be situation-specific—the more you do, the better your visibility, but the higher the cost. Right now, I'd submit that the approach is paying off for MSN, but the costs were substantial. ☺

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DANIEL ROGERS is a program manager working in the Windows division of Microsoft. He has spent his career of more than 20 years focused on large-scale distributed applications. He joined Microsoft more than seven years ago.

© 2005 ACM 1542-7730/05/1200 \$5.00