RELIABILITY BEFORE YOU SHIP

Wayne Roseberry

wayner@microsoft.com

Principle SDET, Microsoft Corporation

ABSTRACT

Reliability is one of the most difficult areas to establish confidence before shipping the product. There is always that nagging question, "Will it be reliable enough for real world load and demand? Did what we build this time get better than what we had before?"

There are two parts to the solution:

1. Choose the right set of metrics and consistently measure against those metrics in every deployment. By carefully defining what you mean by availability and reliability metrics you can better assess if your product is headed toward success or miserable failure.

2. Build a toolset and methodology that joins the metrics to investigation and data collection

This paper will describe how the Microsoft SharePoint 2010 team used reliability and monitoring tools in lab and real-world environments to substantially improve service availability and performance. The presentation will discuss what our key definitions were for availability, failure and performance targets, and show how we used those to establish confidence in reliability before the product shipped

BIOGRAPHY

Wayne Roseberry is a Principal Design Engineer in Test at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Micorosoft Commercial Internet Services, Site Server and all versions of SharePoint. Previous to testing, Wayne also worked in Microsoft Product Support Services, assisting customers of Microsoft Office.

Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.

In his spare time, Wayne writes, illustrates and self-publishes children's literature.

Copies may not be made or distributed for commercial use

1 BACKGROUND

1.1 SHAREPOINT

Microsoft SharePoint Server is a web-based collaboration and content publishing application. Customers buy it to satisfy a broad range of business requirements that include the following:

- team communication and issue tracking,
- document management
- simple workflow processing
- enterprise search and information portals
- business application aggregation
- content management and publishing

SharePoint has been in the market since 2001 and has become a strategic part of the Microsoft product offering, with growth rates that outrun any server product in Microsoft's history. Sales of SharePoint to date have been largely enterprise oriented, strongly attached to Microsoft Office client sales. Microsoft likewise offers cloud-based hosting services for SharePoint.

1.2 PERFORMANCE AND RELIABILITY CHALLENGES IN 2007

SharePoint 2007 released with rapid success in the market. At six months, around the point of the first service pack delivery to channel, the SharePoint team started receiving troubling escalations from early adopters. The general complaint was regarding performance. Customer SharePoint deployments were either experiencing high latency rates, or were experiencing high volume of request failures. The anecdotal evidence of product performance issues were inconsistent with our own performance measurements.

After addressing a few customer deployments one at a time, we drew the following conclusions:

1. The lack of previous feedback from the market regarding performance problems were due to the long deployment and adoption cycles common with server products. Our early adopters, who were only just now getting past their server rollouts were an early indication of where the first real wave of customers would be at about 1 year from ship

2. We would not be able to handle the flood of issues by responding to them as maintenance requests one at a time via the hotfix channel. We would instead need to find as many key performance problems as we could by hunting for them and releasing a new drop to market, much like a typical release cycle.

3. Our performance problems were actually reliability problems. SharePoint was fast enough delivering the typical request, but there were enough bugs in the product that would cause moments of high latency or server timeout to give the customer an unacceptably slow, or failed, experience.

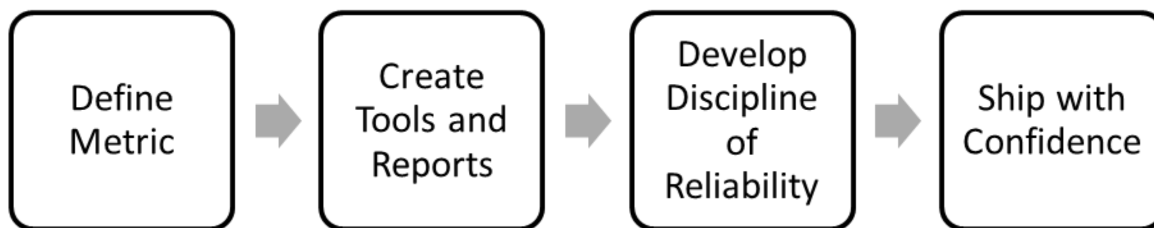
4. Our previous means of assessing reliability were inadequate and we needed to fix them

This led to a large and expensive update of SharePoint 2007. The update was necessary to preserve product credibility and keep customers happy. The project deferred a lot of time and energy away from the SharePoint 2010 release. However, it was during this time that we learned many things about how to

Copies may not be made or distributed for commercial use

approach performance and reliability engineering, which had a substantial impact on the SharePoint 2010 release.

2 WHAT WE AIMED TO ACHIEVE



2.1 DEFINE A METRIC AND MEANS TO TRACK RELIABILITY

In order to track and work toward progress, we needed something to measure. Discovering software flaws that would cause system failures and performance problems was not enough. We needed the measurement to determine the impact that those flaws were creating, as well as a means to motivate discovery of more problems. We needed to know how in-house production systems were actually doing so that we could assess on a regular basis if they were running more, or less reliably. The mechanisms in use by operations staff at the time were ad hoc. One means was to produce a visual graph of performance counter metrics for page delivery time, and then visually assess whether or not the data backing the graph had changed at all. This approach led to a great deal of inaccuracy, as the visual model was represented too densely, and the data points were not chosen well enough to draw any interpretation at all. We knew that we needed a quantitative, mathematical model based on well selected criteria. We knew that we needed tools to generate this model for us.

2.2 CREATE TOOLS TO IDENTIFY ROOT CAUSE

During the SharePoint 2007 release, we had a very difficult time linking evidence of failure (memory leak, server crash, growing latencies) with the root cause. There was almost no instrumentation in the product, and almost no tools available that assisted in the expensive task of examining log files for more information. We knew that the product itself needed more instrumentation, and we knew we needed a platform of tools for examining the data to find issues and draw correlations.

2.3 DEVELOP A DISCIPLINE FOR TRACKING RELIABILITY

Copies may not be made or distributed for commercial use

Simply creating tools and reports does not solve the problem. We knew that we needed to make a regular practice out of reviewing metrics and discovering root cause for every instance of the product failing to meet goals. We knew we had to put together a team that saw chasing down reliability issues as their primary duty. We knew that we had to be able to account for every measurement that was off goal with a list of product flaws or problematic events which caused them.

2.4 SHIP WITH CONFIDENCE

Part of the problem with the SharePoint 2007 release was that we had no quantitative assessment of reliability. We didn't know what sorts of numbers we were getting, and we didn't know what any such numbers would have meant. This had caused us to ship not so much with a false sense of security as a vague feeling of uncertainty. That uncertainty manifested itself as a near disaster in market that was mitigated at a very high cost.

We knew that we needed to establish at least some measurement which quantified reliability in a way we understood. This would at least allow us to establish what we did know, and did not know about product reliability. That way, at ship, we would know at least if we were within the goals we were able to define.

The rest of this document describes how we met these goals.

3 DEFINE A METRIC FOR RELIABILITY

3.1 RELIABILITY BACKGROUND

The primary inspiration for our methodology came from Musa's (Musa, 1998). For those not familiar with Musa's approach, it can be summarized as follows:

1. Reliability goals are set based on customer perceived failure tolerances. The customer can tolerate failures up to a certain level. The product needs to meet that tolerance or better.

2. Reliability is generally either operation based, or time based. Examples of operations might be something like "percentage of jobs sent through the printer that print correctly without jam or failure" or "percentage of orders processed correctly without abort". Examples of time based might be "Hours available to service requests over total hours". The rule is that you have to pick which type of reliability you are going to measure, but you cannot mix both in the same metric.

3. To deal with the varying severity of failure, you group failures into classes of similar severity and weight them. Less important failures have less impact on the calculation than more important failures.

4. For values of reliability up to 95% there is a big complicated calculus formula. For values above 95%, the values drive toward a simple percentage calculation like "Attempted Operations/Total Operations" or "Available Time/Total Time".

There is a lot more to Musa's work than the above. He talks about how to weight operations in a load test, which failure levels to abort a run on, and how to use reliability test lab results to predict trends toward ship goals. For our sake, though, we were largely interested in how to build metrics around failure and reliability. To that end, we further simplified Musa's conclusions with our own:

1. Failure rates below 99% are so abysmally bad that accuracy on reliability becomes less important – so we were willing to tolerate simple percentages in our metrics, even for values of failure rate larger than 5%. We didn't consider ourselves even close to the ballpark until we crossed 99% (every 9 is an order of magnitude jump)

Copies may not be made or distributed for commercial use

2. We had no idea how to weight our failures relative to each other. We didn't know if an "HTTP 500 server encountered an error" result was the same, more, or less severe than a page timeout. We skipped weighting different classes of failure and considered everything we captured equal

3. We realized that we wanted to know operational based reliability (# successful requests / # total requests) and time-based reliability (available time / total time). We also realized we wanted to reflect variance and reliability in request latency. We opted to measure all three, but as separate metrics.

The following section discusses how we defined and calculated those metrics.

3.2 OUR METRICS

As stated above, we chose three metrics to track, Failure Rate, Availability and Latency. All goals are stated as sustained in a real-world production environment for a period of 1 week.

3.2.1 FAILURE RATE: GOAL < .01%

Selecting a failure rate of .01% allowed us to target a success rate of 99.99%. We chose failure rate as an operations-based metric because single failures spread far enough apart have no impact on an availability calculation, but can still be frequent enough to cause substantial customer dissatisfaction.

1. Failure Rate: operations failed/operations attempted
2. We used the following definitions for failure:
    a. Timeout: Any request that takes longer than 15 seconds was considered a timeout and failed request
    b. HTTP error code: Any request returning an unexpected HTTP return code between 500 and 599 (the failure code range in the HTTP specification), 404 (not found), 403 (access denied)
    c. Error in body text: SharePoint pages will sometimes return with a successful HTTP response code (e.g. 200), but may have an error message in the body of the page. For these cases, we looked for simple indications of failure, such as the string "error" or "exception" in the HTML body of the response.

3.2.2 AVAILABILITY: GOAL 99.9%

While failure rate gave us a sense of how many requests were failing, most customers and systems operators talking about server and service reliability in terms of availability or uptime. We took the same data we were receiving to calculate the failure rate and used it to calculate an availability metric as follows:

    d. Availability: (Total Time – Downtime)/Total Time

    e. Downtime: We examined the timing of the requests for clusters of failures. If failures were packed close together in time, we counted that time block as being down. We defined a sliding window as follows:

        i. Window of time 10 seconds long

        ii. More than 10% failures occurring within a 10 second span defines a downtime window

3.2.3 LATENCY: GOAL 95TH PERCENTILE LESS THAN 500 MILLISECONDS

One lesson we had learned from customer issue in the previous release was that many of the reported performance problems manifested as intermittent moments of high latency. Most of the time the server

Copies may not be made or distributed for commercial use

would deliver requests well below the time needed to render the page fully in 1 second or less, but periodically requests would start taking 10, 15, 20 seconds or more before even leaving the server. We decided we needed a reliability metric that was exclusively about page latencies, defined as time between the request being received from the client the time for the server to finish constructing the requested content and deliver it to the client.

We also knew that we could not take a simple aggregate of performance. Average latencies, even on systems where customers were reporting performance problems, were looking very good. The problem is that it takes very few outlier moments to create substantial customer problems. We decided we needed to target a distribution of latencies. We did this via the following method

1. Collect the time to last byte on client in milliseconds for a known set of pages
2. Sort all requests ascending by latency
3. Report the latency value at 25%, 50%, 75% and 95% of requests

By doing this we could tell whether the system was having performance problems with every request (at which point the values between 25-75% would be higher), or if the performance problems were isolated to outlier requests (at which point the values at the 95% would be higher). It also kept us from missing a performance problem that was hidden inside an overall average that was doing well.

3.3 THE MEASUREMENT METHODOLOGY

To collect these measurements, we had to build a few tools and make a few choices.

3.3.1 BUILD A RELIABILITY MONITOR

The first tool we built was an external monitor. This program executes a set of HTTP requests to the server. The program submits the requests repeatedly. It records the time between request submission and time to last byte. It records all error codes, timeouts, as well as any request that contains known failure messages in the content. This recorded is then aggregated into the reliability metrics stated above.

These metrics are collected and for every internal server that running SharePoint. We built a website that delivered daily updates on the metrics. The following is an excerpt from the report running against the Office team website, March of this year:

Copies may not be made or distributed for commercial use

Reliability Metrics

	RTM	2011-03-31	2011-03-30
		Actual	Actual
Availability (%)	99.99	99.93	99.77
Failure Rate (%)	0.01	0.07	0.23
Latency: 25th Percentile (sec)	0.2	0.02	0.05
Latency: 50th Percentile (sec)	0.3	0.06	0.10
Latency: 75th Percentile (sec)	0.4	0.12	0.21
Latency: 95th Percentile (sec)	0.5	0.36	0.94
Avg APP Memory Used (GB)	6	0.00	0.00
Max APP Memory Used (GB)	8	0.00	0.00
Avg WFE Memory Used (GB)	6	0.00	0.00
Max WFE Memory Used (GB)	8	0.00	0.00
Time in GC counter (%)	5.0	1.06	0.94
WFE Crashes	0	0	9

3.3.2 PICKING THE OPERATIONS

3.3.2.1 WELL KNOWN PAGES

One of the first decisions we had to make was which operations to monitor. The problem with a rich, complicated server product that responds to user requests is defining a standard set of goals for all operations. We knew we wanted pages to fully render within 1 second on the client, leaving us about half a second (500 milliseconds) to finish processing the page on the server. We knew we didn't want to see failure code against user operations. Our debate was whether to base our measurement on a sample of requests or against all requests.

The argument for measuring all requests is simple. We didn't want to miss anything. This is a classic test dilemma, and can throw a test team into paralysis by analysis as you try to craft the test methodology which will catch all issues. The argument for measuring only certain requests is more complicated.

Not all operations are created equal. Some requests are incredibly fast, with delivery in single milliseconds, while some requests are known and expected by the end user to take a multiple seconds to

Copies may not be made or distributed for commercial use

render. Our workload was dominated by lots and lots of small, fast requests, and we didn't want those to skew the distribution to the point of obscuring other requests. In the same regard, we didn't want to investigate latencies only to discover that higher latencies were acceptable for that particular request.

Even error conditions are difficult to equalize. Some errors should not happen ever (server crash, an exception thrown in the middle of a page body), but some errors are a legitimate response to an invalid end user request, such as "illegal operation", "forbidden operation", "invalid request" or "access denied". We knew we didn't want such errors for well-formed requests, but our measurement was not about determining how often the end user sent a bad request. Our measurement was about how reliable the server was under typical workloads – bad requests or not.

We opted to only measure a specific set of known requests. These requests were ones that should never violate the metrics we established – they should never return "access denied", or take longer than 500 milliseconds to render on the server. We decided that while this would miss failure on some operations, we would still be catching overall server health and reliability. We were betting that failure across the sample operations would highly correlate with failure on other operations.

3.3.2.2 PAGES CUT ACROSS RESOURCES, TIERS AND FEATURES

Once we knew we were going to work with a known set of operations, we knew we had to pick them carefully. We used the following criteria:

1. We would have a request representing each feature in the SharePoint product: Features in SharePoint are sometimes impacted by separate resources or components of the system, each capable of their own bottlenecks and failure points. Requests for documents from document libraries come out of a different SQL database than requests for a user's profile page. Requests to service end user search queries come from a completely different set of servers, files and databases than requests to deliver site pages.

2. The request would be one that was expected to be complete within 500 milliseconds or less: There are requests against database reports, or complex queries against large sets of data, which take a little bit longer. We decided to eliminate those to keep everything within the under 500 millisecond range.

3. The request would involve the complexity and cost of rendering a page body: Most pages viewed by end users have multiple HTTP requests, the bulk of which are tiny graphic object for rendering bullets, highlights and other page entities. Some requests for files are accompanied by frequent inexpensive polls on status checks. We decided to eliminate all of these smaller requests, particularly if they occurred in a background fashion where the end user might not notice them.

4. The request had to be relatively non-impactful on the system to execute: All requests put load on the system. We didn't want the measurement process to introduce a problem, so we intentionally picked requests that if delivered under a load necessary to get a good measurement would generally have little measurable difference to the system's reliability.

3.3.3 TUNING THE SAMPLES

Artificial load based monitoring systems require tuning. The problem is to balance accuracy of measurement against impacting the system. These two variables sometimes impose each other.

We wanted to be able to measure reliability all the way to 99.99% This meant that the sample intervals had to be as small as 1/10,000th of the entire time period. Any larger would make it impossible measure all the way to 99.99%. This works out mathematically as follows:

Seconds per day: 86400

Copies may not be made or distributed for commercial use

Largest interval that could still measure to 99.99% accuracy: 8.6 seconds

In other words, out of an entire day, a single block of downtime lasting 8.6 seconds is the difference between 100% and 99.99%. We didn't want the occurrence of a single failed request to have that much impact. This meant we needed to send requests to the server at a higher frequency than once per 8.6 seconds, and that our failure window in the downtime calculation needed to be smaller than 8 seconds.

For request rate, we selected 2 requests per second. The product system at peak easily ran from 100-300 requests per second, so we felt fairly certain this would fit safely within our parameters.

We hit problems with these choices. One that came up right was search requests. For the initial production system, realistic requests per second from end users was running around 1 request every few seconds. At 2 requests per second, we were nearly quintupling the request rate above normal user load. To make matters worse, the search system cached recent results. The monitoring sample was forcing an artificial load pattern into the search cache, skewing distribution of data and pushing out real-world driven cache data, thus slowing down the end user experience for search users.We adjusted the search based requests to monitor at a lower rate, although by doing that we lost the ability to report on search reliability granularity to 99.99%.

4 PRODUCT CHANGES & TOOLS TO IDENTIFY ROOT CAUSE

Once we had actual measurements of production system reliability, we needed tools to aid in diagnosis of the causes of downtime, high latency spikes and operation failure. Some of these tools were built into SharePoint as features, others remained external. A brief listing of those feature changes are included below:

Unique Request Identification	Every end-user request was assigned a unique identifier that followed was recorded at every step inside the server via debug logs, error reports and database queries.
Detailed Debug Logging	A detailed logging system with varying levels of verbosity was tracked on ever server. In addition, a usage tracking system was introduced to allow identifying usage patterns and trends via a database query.
Monitored Code Blocks	The base class inside SharePoint was built such that it wrapped all requests with a special block that allowed for instrumentation and timing statistics. Data as captured in the debug logs, and any request that violated certain thresholds, such as length of time processing, was flagged as an exception in the debug log.
SQL Statistics Capture	SQL statistics, such as too much time spent executing a query, or queries that consume a great deal of IO and other resources on the database, were recorded inside the usage database.
Memory Usage Capture	Monitored code blocks tracked memory objects allocated and released. At end of the monitored code blocks the number of released objects did not match the number of allocated objects, an exception was recorded in the debug log.

Copies may not be made or distributed for commercial use

Developer Dashboard	A special control was placed on all web pages which if enabled would deliver diagnostic information on the page containing time spent on each step of page rendering, all database and service calls executed on the page and a variety of other useful information to aid in performance diagnosis. The tool ships with the product and is available for customers as well (Microsoft, 2010).
SharePoint Diagnostic Tool	A special tool that aggregated usage database and debug log reports to help spot trends and isolate problematic requests. This started as an internal ad hoc tool and eventually shipped for customers to use as well (Microsoft, 2011).

5 HOW WE USED THE TOOLS

We had all the pieces of the above tools in place about halfway into the 2010 release cycle.

We introduced the tools into both test lab runs and real-world production systems. In the PNSQC 2010 conference I published a paper about testing simulated real-world load patterns in SharePoint 2010 (Roseberry, W.). We used the tools described in this paper to measure reliability in those test runs. This gave us parity with the production systems. We knew that we were measuring reliability on the same terms and definitions, the only difference being the load pattern. Any difference between lab results and production system results were either from the load pattern, the data set under load, or configuration of the system itself.

In fact, the simulated load-pattern tests were effectively finding performance and reliability bugs until approximately nine months before product release (around the time of the second beta release for Office 2010). At that point, load test results were almost always delivering 100% availability with very little evidence of latency problems, while production systems were returning approximately 90% availability and frequent spikes in end user latency. We made a conscious decision at that point to focus more of our time analyzing the production system failures.

We had the product in deployment in three internal systems:

- HTTP://Office The Office team departmental portal: this is where the Office product group shares information with each other, the rest of the company, and collaborates on projects and specifications

- HTTP://MSW The company-wide corporate portal. This site is used for searching across all content inside Microsoft's internal network, as well as finding key resources like legal, human resources, IT, facilities, etc.

- HTTP://SharePoint A company-wide hosting site for teams to have their own SharePoint site. Very similar to the http://office site, it mostly serves as a team collaboration and communication solution.

- HTTP://MY A company-wide personalization site, providing personal profile information for all employees in the company.

The hard work that remained was determining the best way to investigate reliability failures on these product systems when they happened.

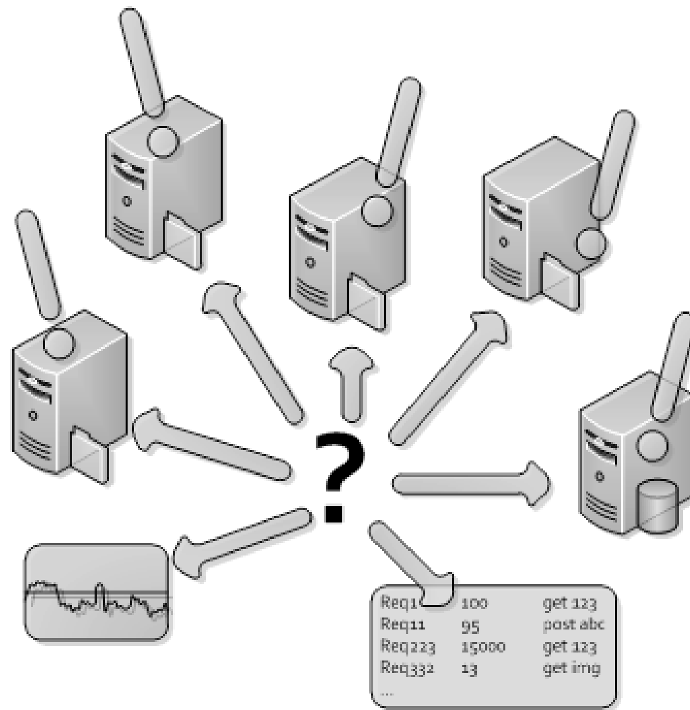
Copies may not be made or distributed for commercial use

6 INVESTIGATION

6.1 COMMIT THE RESOURCES AND BUILD THE TEAM

It is popular to say “Reliability is everybody’s job”, and while that it is true, it should not be mistaken as an argument against dedicating resources to specifically investigating performance and reliability problems. We found that properly addressing reliability a committed team of individuals actively investigating issues and hunting down root cause of problems. During the nine months of the SharePoint Server 2010 project, there were roughly a dozen or so people doing nothing but investigating performance issues on our internal deployments.

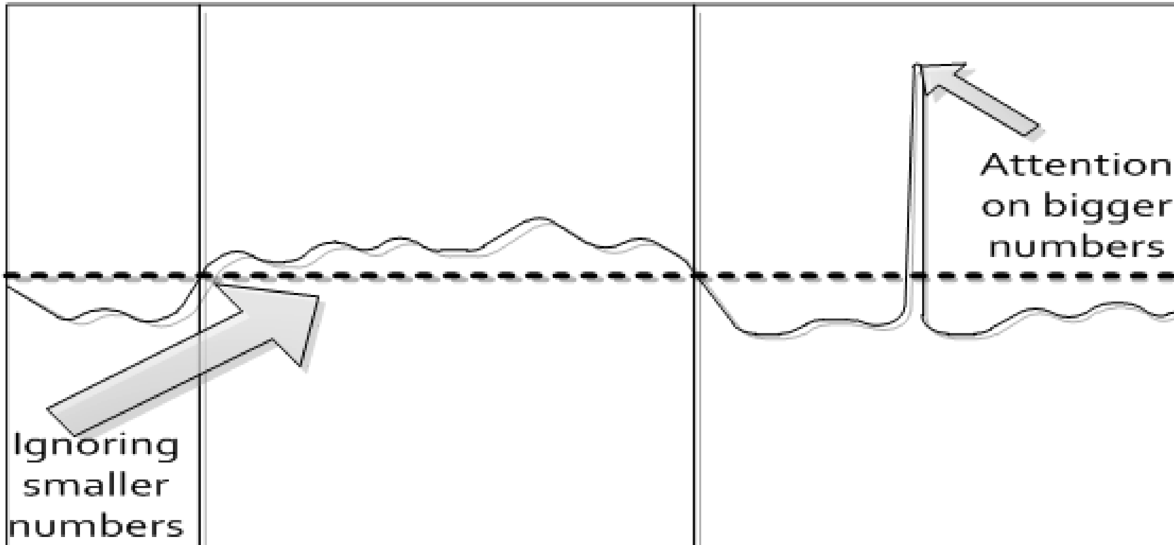
6.2 WHERE TO START?



Even with a set of tools at our disposal, our initial investigations were informal and random. The sheer volume of data in the system presented almost as much difficult to navigate as complete lack of information. Engineers tended to investigate problems based on what they were familiar with. If an engineer was really good at troubleshooting memory, then they spent a lot of time taking memory dumps. If an engineer had a lot of database experience, they would analyze query plans and disk utilization on the SQL server.

Another problem that occurred was how engineers chose which events in the reliability report to investigate. It was not uncommon for them to select the largest number available in terms of latency, mostly because such numbers stood out more readily in the report. The problem here was that there were sometimes hundreds of data points of this nature, and often the largest number was not the biggest problem.

Copies may not be made or distributed for commercial use



The common aspect of the randomness was that engineers were examining failures at the individual level. Rather than look at the metrics defining availability and overall latency, they were looking to the lower level constructs that might indicate a problem. For example, a specific query on the database server might have been more expensive than others, so the engineer would begin examining that query to determine the reason why. While these led to many worthwhile discoveries, the process was slow, and often did not correlate to the most important problems that were showing up on the reliability measurements. Further, even when bugs were reported, it was not uncommon for the problem to be resolved as “Won’t Fix”, given that there was no evidence of how much of a problem the bug was causing.

We needed a methodology to guide engineers to properly investigate the most important problems.

6.3 FINDING PRIORITY IN STORMS AND SPIKES

The solution to random investigation was to stop investigating individual requests and instead focus on periods in time. The idea was to treat reliability and performance problems as an event that had a weight, much like hurricanes are given a name and a weighting. This would aggregate multiple requests together into the same event, allowing us to prioritize our investigation by the event representing the biggest problem.

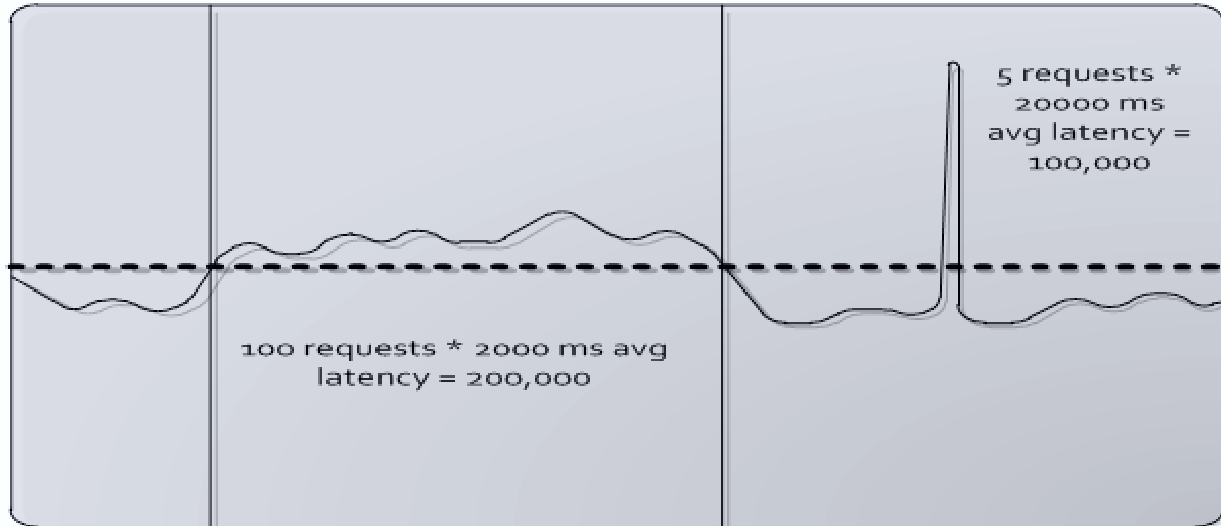
We created a construct called a storm. A storm was defined as a series of contiguous requests that failed to meet our criteria for success. A storm had two properties, average latency of the requests in the storm, and the number of requests in the storm. These were multiplied together to calculate what we called the storm intensity.

For example, imagine two blocks of requests, Storm 1 and Storm 2, each having the following properties:

Storm 1: 100 contiguous requests, average latency 2000 milliseconds

Storm 2: 5 contiguous requests, average latency 20000 milliseconds

Copies may not be made or distributed for commercial use



Storm 1 has an intensity of 100 * 2000 = 200,000. Storm 2 has an intensity of 5 * 20000 = 100,000. While Storm2 has much longer requests, Storm1, with its longer time span of shorter requests actually accounts for an end user experience that is twice as bad. Both events were important, but Storm 1 rises to the top of the priority stack.

Even more important is the effect storm analysis has in regard to filtering out single requests that have little impact on overall reliability. Imagine that in the same time period there was a single request lasting 25000 milliseconds. While it is interesting, it is not nearly as important as the contiguous span of 5 requests in storm 2 all lasting 20,000 milliseconds.

6.4 INVESTIGATE EFFICIENTLY

The next technique we had to master was figuring out root cause of the problem as efficiently as possible. This is another place where the sheer volume of information overwhelmed us at first.

To put the volume in perspective, our production systems were typically running at 300-500 HTTP requests per second during the peak hour. Each of those requests would create around 1-2 dozen entries in the debug logs, and at least one entry in the usage log. The typical request would require 1 to 4 queries to the database, maybe more. This meant that even for a period of only five minutes of some sort of problem there were likely 3 million or more data points to examine for a possible cause. Some of these data points were difficult to investigate. An expensive database query, for example, may take days or hours of analysis before determining if it indeed contributed to the problem. With this many data points, and the cost of investigation being so high, we had to streamline our technique.

We came up with our approach by having all the engineers come together in a room, and say how they approached their investigations. Everybody laid their tricks out on the table, compared notes, and shared how they might do something differently. Doing this, we finally came up with a strategy that made investigation fast and efficient as possible, and more likely to find results that contributed to the problems customers were experiencing.

Here is the process we defined:

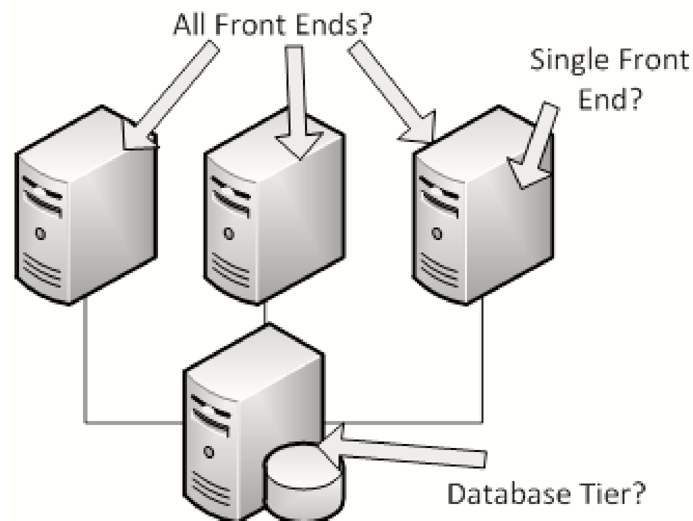
1. **Start from the reliability report numbers**: This guaranteed that the investigation would always be relevant to the metrics.

Copies may not be made or distributed for commercial use

2. **Pick the time period to investigate based on storm analysis**: Pick the times which had the highest intensity. This guaranteed that the first things found would contribute the most to improving the numbers in the report.

3. **Isolate the requests that are within that timespan – focusing on the moment the problem started**: This allowed the number of requests to investigate to narrow even more, substantially reducing the amount of data required. Further, we noticed that whatever had caused the problem had to be happening at the moment the storm started. This allowed us to focus not just on the entire time span, but only those requests that were actively processing at the start of the time span, reducing the required requests from hundreds of thousands for a five minute storm to just a couple dozen.

4. **Isolate the tier having the problem:** This particular step is specific to systems like SharePoint, where components, services, and applications are distributed across many machines and resources. There were certain signals in the data, such as CPU performance counters, memory usage, length of time spent making external service and database requests, that allowed us to isolate very quickly whether a problem was occurring at a front end machine, at a backend service or database, or possibly across every machine in the system at once. This was so effective a means of reducing the investigation surface area that we sometimes went right to this step before even examining the request involved in the storm.



5. **Investigate the types of problem that occur at that tier specifically:** If we could see that all slow requests were coming from a single front end machine, then we knew that it was unlikely our problems would be in a shared resource, such as disk utilization on a shared database. This meant we would begin investigating CPU utilization on that machine, memory dumps, IO contention or serialized lock contention at the front end. If, however, we noticed that database queries were running long across all requests on all machines, then we would start looking at the database for expensive queries, IO intensive operations, disk queues, memory and CPU utilization.

Copies may not be made or distributed for commercial use

6. **Correlate suspects to problem:** Once we had a list of possible causes, expensive database queries, serialized access for IO ports, memory consumptive operations, garbage collection, CPU heavy requests, we had to establish how well those requests correlated to the problems in the report. Were they the actual cause of the problem or not. To do this, we would look for other occurrences of the same problem and see if they lined up well with problems in the report. For example, if there was an expensive database query that occurred 100 times in the span of a day, and only two or three of those

7 SHIPPING WITH CONFIDENCE

Fortunately, the story here ends well.

First, we met all of our performance goals for our internal deployments. By the time we were shipping, all of the internal systems were running at < .01% failure rate, 99.9% availability and request latencies were easily less than 500 milliseconds at the 95% of requests. For the last two to three months of the cycle all availability incidences had been identified as caused by hardware, network or environmental issues outside the control of the product.

The mood and confidence at ship was a great deal different than with the previous release. We weren't naïve enough to believe that there would be no performance problems coming from the field, but we had solid numbers from production systems and we knew what they meant.

Even more important, though, was the response from support. We were monitoring the support requests, but we knew from previous releases that there would be a lag of at least six months or longer before there would be enough large scale deployments to really know if we had succeeded. We kept checking, and we kept hearing the same thing. Performance appeared to be okay. All evidence showed that our investment had paid off.

We knew we had a pattern and a methodology. We moved on to the next release, taking everything we learned with us. Indeed the methods, tools and learning we used shipping SharePoint Server 2010 would serve as the backbone of our next big investments in our hosted offerings in Office 365, running SharePoint in the cloud for millions of users worldwide.

8 CONCLUSION

The lessons learned by our team, and conveyed in this article, are not new. If anything, they were just the application of old principles and techniques. It is reassuring to learn, though, that taking a methodical, measured approach to establishing product reliability before ship is possible. The things you should do:

a. Pick your metrics and measure
b. Build tools, both external and inside the product, to make root cause investigation fast and efficient
c. Invest in time and people to investigate, using them to build better tools, methodology and find the root cause of problems

If you commit to similar kinds of approach to solving the problem as we did you should be able to ship confident that your product is meeting yours, and most importantly, your customers' expectations for reliability.

REFERENCES

Musa, J. *Software Reliability Engineering*. Osborne/McGraw-Hill

Copies may not be made or distributed for commercial use

Roseberry, W. *“Simulating Real-world Load Patterns When Playback Just won't Cut It”.* Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon, October 2010.

Microsoft. “SharePoint Diagnostic Studio 2010 (SPDiag 3.0) (SharePoint Server 2010).” Last modified April 28, 2011. http://technet.microsoft.com/en-us/library/hh144782.aspx.

Microsoft. “Using the Developer Dashboard.” Last modified May 2010. http://msdn.microsoft.com/en-us/library/ff512745.aspx.

Copies may not be made or distributed for commercial use