Monitoring, at Your Service

Automated monitoring can increase the reliability and scalability of today's online software services.

BILL HOFFMAN, MICROSOFT

Internet services are becoming more and more a part of our daily lives. We derive value from them, depend on them, and are now beginning to assume their ubiquity as we do the phone system and electricity grid. The implementation of Internet services, though, is an unsolved problem, and Internet services remain far from fulfilling their potential in our world.

Internet services are implemented as large, distributed computer systems. Large computer systems require human operators. Hardware fails, software fails, facilities require management. Humans act as the white blood cells that keep the systems working. Humans diagnose problems. Humans replace broken hardware. Humans remove damaged nodes from systems. Humans tune systems to the changing demands of large user bases, optimizing as new features are added and new access patterns emerge. Humans respond to external abuse of services.

Monitoring, at Your Service

It turns out that humans are also expensive and prone to making mistakes. For example, an article in *Scientific American* in June 2003 surveyed three Internet sites and found that operators caused 51 percent of the failures.¹ At Microsoft, we have seen human error cause problems as well. We have seen issues with software bugs, improper handling of hard drives, and misconfiguring of hardware and software, to name a few. We have also seen human error trend downward. As systems become larger and larger, hardware failures tend to dominate.

Focusing on the core goals of an Internet service—providing a reliable service to users and keeping costs low—one finds that human involvement gets in the way of success. To make progress toward next-generation scenarios, Internet services need to focus on developing software that provides for cost-effective and reliable deployment of computer systems. In particular, we need to pay attention to the challenges involved in systems consisting of thousands of computers.

A discussion on monitoring in Internet services provides a good entry point into the larger problem.

MONITORING

Internet services find different reasons to monitor their systems:

- Business planners need to understand the user base: How do users access the service? What is the overall availability of the service?
- Operations planners need to undertake capacity planning: How many more computers do we need to purchase to handle the new users we are attracting?
- Engineers need to understand what the system is doing at any point in time: Are there failed components? Is a particular component slow? What data do we have to help diagnose problems that customers report?

In large-scale Internet services, built on top of thousands of individual components, what is the role of monitoring? Traditional approaches involve collecting data, rendering visualizations of the data, and ultimately relying on human analysis to arrive at decisions. The beauty of monitoring data is that it can be processed by computers much faster than it can by the human brain. At Microsoft, we are asking ourselves how systems can benefit by directly consuming monitoring data, program-

matically, and making automatic decisions as a function of the data. We specifically focus on monitoring data as it can be used programmatically by a system itself.

The Communications Services Platform group at Microsoft has been influenced by the ROC (Recovery-Oriented Computing) project led by David Patterson of the University of California, Berkeley, and Armando Fox of Stanford University. (For more information on ROC, see <http://roc.cs.berkeley.edu>.) ROC holds that we should embrace the inevitability of system failures and human error in Internet services by optimizing a system's ability to recover from failures.

Take a hypothetical service that has a single type of failure: it happens once a month, and each time it happens, it takes three hours to recover. This results in a total of 36 hours of downtime over the course of a year, for an availability score of 99.59 percent. When addressing this issue, the service's engineers can attempt to improve availability by reducing either the frequency of the failure or the recovery time of the failure.

Note that reducing the recovery time has a direct impact on availability. For example, reducing the recovery time from three hours to 45 minutes will reduce the amount of downtime over the course of a year, resulting in nine hours of downtime for an availability score of 99.9 percent.

Availability	Amount of downtime allowed per year
99%	~3 days
99.9%	~9 hours
99.99%	~1 hour
99.999%	~5 minutes
99.9999%	~30 seconds

The reliability of the system is constant, the failure rate is the same, but the recovery time has been optimized. This is an important example because, in practice, we have found it more fruitful to optimize the recovery times than to reduce the occurrences of failures. Failure rates tend to be fixed, stemming from fundamental properties of hardware.

We have found ROC to be the correct approach in practice. For example, we almost never worry about trying to optimize a server to respond faster by a few more

milliseconds, but we are always focusing on optimizing recovery procedures. Further, a mature and technologically advanced approach to monitoring is the most important factor in the ability to recover quickly from failures. Without the ability to detect failures, there is no chance to recover efficiently from them.

At a high level, we need to understand what is happening across thousands of nodes at any point in time. When a problem occurs, a system must be able to fail-over and continue operating successfully without a human fixing the problem. All important improvements in the cost of both hardware and daily operations start with establishing a solid monitoring system.

PROCESS CONTROL

An Internet service is a 24/7 process that involves many interacting parts. As in manufacturing, Internet services strive for consistent, repeatable results. Although there is no manufactured product, the service itself should exhibit consistent reliability and responsiveness. When adding capacity to a data center, planners desire predictability—recently added systems should function as well as the existing systems.

We borrow from the field of SPC (statistical process control) to achieve consistent results. SPC is used to ensure the quality and stability of manufacturing processes. The goal of SPC is to use statistical methods to measure variance in processes, allowing one to identify problems and stabilize processes over time. For example, it would not be economical for a manufacturing plant that produces tires to use an unpredictable amount of rubber for each tire produced. Using SPC, the manufacturer sets a target for the amount of rubber that should be used in each tire. Further, the manufacturer sets a tolerance for the amount of acceptable variance around the target. SPC methods then measure the rubber usage on every tire, helping the manufacturer to understand if the manufacturing process is stable and predictable with respect to rubber usage. If the process becomes ineffective, or unpredictable, then the SPC methods help to identify the source of the problem.

This approach has benefits for Internet services. Service managers need to understand and define what the system looks like when it is working well. They need to be able to detect any variance or trending away from that state. When engineers make changes to systems, service managers need to be able to measure the effects of the changes. We use SPC measurements to understand if we are using our resources properly. For example, if all of a service's transactions are being performed well under the

target response time, then the service is probably over-built.

In addition to using SPC for managing a service, we can use it in software to achieve self-management. A service's software implementation itself should actively maintain a strict process control. Service software should be factored and designed from the ground up with process control in mind. Software should borrow and build from the concepts of closed control loops, process control, and self-stabilizing systems. Think of control systems as responding continuously to stimuli. Software, generally, does not fit this model, but Internet services fit it perfectly. Internet services run 24 hours a day, 7 days a week, constantly responding to requests from millions of users. Although, at the fundamental level, Internet services are not realtime systems and are not continuous in a strict sense, they are in aggregate constantly responding to stimuli. When a change is made, the effect can be seen relatively quickly in systems under heavy load from millions of users.

Control systems can be classified as open loop or closed loop. An open-loop system can be monitored and controlled, but the monitoring information is not used directly to control the system. A closed-loop system, however, feeds monitoring data directly back to controllers, allowing it to compensate for variations in operating conditions.² An example would be an air conditioning unit, shown in figure 1. An open-loop system would allow the operator to control how much work the air conditioner does. A closed-loop system would measure

Monitoring, at Your Service

the temperature of the air, feeding the result back to the controller, thus controlling the air conditioner's output and maintaining a constant temperature.

Each service or process in a system can be thought of as exposing a controller that implements a standardized and generalized interface. Similarly, each service or process in a system can be thought of as exposing a standardized set of sensors, through a generalized software interface, that provides a picture of the performance of the component. The component can monitor itself, feeding back to its own controller, to maintain a desired state. External software agents may also monitor components, interfacing with exposed controllers to repair or maintain the desired state for the wider system.

DEFINING THE STATE OF A SERVICE

Internet services must be self-stabilizing systems. These are systems that, when perturbed, are guaranteed to return to a legitimate state in a finite number of steps.³ When a software system does not satisfy this property, human operators must step in to place the system in a legitimate state.

A good example of this involves feedback loops, a problem we have seen come up in practice many times. Consider a situation where a specific server in a system begins responding slowly and failing intermittently. There is a temporary problem on the machine. Clients, receiving failures, time out and retry their operations. On the server, the failed operations are still attempting to complete. The server, having problems, is not yet aware that clients are timing out and retrying. Each client retry exacerbates the problem by taxing the server's already strained resources, initiating new operations that the server cannot complete, to the point that the server cannot respond at all and cannot recover. If clients had simply stopped using the server for a period of time, it would have been able to recover, with the intermittent failure going away, and the server returning to a good state. In this example, the system, when placed in an illegitimate state, is not taking steps to move back to a legitimate state.

How do we apply the concept of self-stabilizing systems to a software service? We have to start by defining legitimate state in terms of monitoring data.

We can reason about the state of an Internet service in the following areas:

- Responsiveness—how quick is the site at responding to users?
- Throughput—how much work is the site doing?
- Resource usage—how much capacity is being used?
- Defects—how many failures have occurred?
- Recovery—how long does it take to recover from failures?
- Management—how much human involvement is there?

Each node in a system must take careful measurements in each of these areas. Problems in any of the areas could indicate that the service is in an illegitimate state. Monitoring software, given raw measurements in these areas, should pay attention to variance in each of the areas as well. Inconsistent results can be as much a sign of an underlying problem as straightforward bad results can.

In practice, we have found that choosing the right measurements makes a big difference. For example, when measuring responsiveness, does this mean measuring the end-to-end response time for a user? Does it mean measuring the response time of each internal transaction? If all measurements are low-level measurements, understanding exactly what a user experiences is extremely difficult. The ability to measure a user's experience, independent of the implementation, including all end-to-end functionality, is extremely important. On the other hand, high-level measurements do not allow software or service managers to understand exactly where problems are. Thus, we have found that it is also important to have many specific measurements. Correlating low-level implementation measurements to high-level user experience measurements is a big challenge.

FAILURE DETECTION AND ISOLATION

Every engineer dreads being paged in the middle of the night with the message, "The site is down!" We all have live-site stories to tell, ranging from "Can you believe this simple problem caused such a big deal?" to "We spent an entire week debugging the site and ended up re-architecting the network in one weekend!"

Internet services, with many interconnected components and with changing and emerging user access patterns, will fail in nontrivial ways. Simple problems end

up cascading into site-wide issues that impact customers. At some point in the process, someone notices—either an engineer or a customer—and the problems escalate.

Escalations will typically arrive pointing to side effects of the root problem. For example, a set of slow or failed hard drives can result in customer complaints. When customer complaints escalate, problems tend to be stated as a function of customer experience. Since different customers describe their problems differently, engineers may feel as though they are seeing multiple real problems. After careful debugging, engineers will find the root problem and repair it. The overall approach, however, is inefficient.

To tackle a problem such as this, services must detect and isolate failed components. First, a service must detect failed components so that they do not silently cascade failures into other parts of the system. The best way to find a failed component is to directly monitor the component, rather than to debug for hours after an escalation. Then the service must isolate the failed component so that it cannot impact the availability of other, working systems.

Failure detection is a murky area. Components can fail in an absolute sense. For example, if a computer's CPU stops working, we can say that it has absolutely failed, knowing that it will not work unless physically repaired. The computer is down, and all requests to the computer will fail.

Components can also fail in a relative sense. For example, if one node in a system were functioning at 10 percent of the performance of similar nodes in the system, we would want to consider it failed, even though it is technically functioning. Failures can also be intermittent. A server that sporadically fails, working fine for long periods of time but entering a failed state for an hour every now and then, may be a failed server.

Given a stream of monitoring data, how can we accurately determine if a component is failed? A simple binary approach is not sufficient. It helps to enumerate the states in which a component may exist:

- Working—no errors have been reported.
- Failing—errors have been reported.
- Probation—we think the component is recovered, but it requires additional monitoring to confirm it.
- Failed—the component has been taken out of rotation.

State machines are very applicable to the problem. Given a state machine to reason with, we can intelligently decide how a component transitions between the various states. We have found that simple thresholds on error counts work well. For example, if there are 1,000

straight failures against a component, we consider the component failed. This technique is simple, but is deficient in some areas. For example, a failed component that is rarely accessed will take a long time to reach its failure threshold.

More advanced techniques take failure rates into account and are generally more adaptive. We have found that, in practice, simpler techniques tend to prove sufficient. Most real-world failures happen regularly, are well

In an Internet service, each node needs to know the state of all the other nodes it may communicate with.

understood, and are easy to detect with simple thresholds. Since failure criteria do not change rapidly, there is little to gain from adaptive techniques.

In an Internet service, each node needs to know the state of all the other nodes it may communicate with. This helps prevent cascading failures by allowing “fast failing”—returning instant errors when attempting to access known failed components instead of tying up resources while timing out. This also facilitates failure isolation and prevents negative feedback loops.

Although most components can retain the full state of all other components in memory, persistent-state storage is a requirement. As machines go down and are brought back up, we want them to know all the currently

Monitoring, at Your Service

failed components without having to re-detect the states through real operations.

LOAD BALANCING

Internet services tend to solve very parallel problems. For example, among millions of users, individuals usually introduce work items that can be independently satisfied. Replication techniques allow common data structures to be queried across many machines in parallel. To the extent that a successful large-scale Internet service must be parallelized to scale well, service managers must be able to measure how well their service is actually parallelized.

Monitoring systems help determine how well parallelized a system is. Measuring the balance of a system is relatively straightforward, to the extent that taking measurements across thousands of computers is straightforward. You can directly measure the amount of storage capacity, memory, network bandwidth, and I/O bandwidth used. You can compare these measurements across the nodes in the system to determine if it is uniformly balanced or not.

Where imbalance is detected, you can bring the system back to equilibrium by moving data and user accounts around, and generally by reallocating resources in the data center such that balance is achieved. But how do you intelligently reallocate resources?

Simple techniques for resource reallocation can succeed in bringing a site to a balanced state, but they do not minimize the amount of resources that must be explicitly moved. For example, in the load balancing of a large-scale distributed storage service, with petabytes worth of data, where data is accessed in a nonuniform manner, one simple technique would involve moving data around at random. Although simple and sure to guarantee balance, this technique is prohibitively expensive since it involves constantly randomly moving data around across all nodes.

Research into methods for minimizing the amount of change that must be introduced to achieve balance would be fruitful. We can statistically correlate certain user behaviors with the amount of work that they introduce for the system. By having monitoring systems measure these behaviors, we can compute strategies that choose

specific data to move while still achieving balance, minimizing the amount of resources that must be reallocated to bring a system to a balanced state.

In practice, we have seen user behaviors change drastically. As new technologies are developed, and services offer new features to users, we find completely new access patterns. Digital photos provide a good example. Internet mailboxes used to be relatively small, filled with simple text messages. Now, Internet users routinely swap digital photos and have many multiple-megabyte messages in their mailboxes. The greatest difficulty we have found is in adapting balancing algorithms to changing user patterns. On the monitoring side, however, we do not need to change the way we monitor balance to keep up with changing user patterns.

DEPLOYMENT CORRECTNESS

Internet services push the software industry further toward the “software as a service” vision. More and more applications are available without the need for a local installation of software on the user’s PC. Software now spans many computers and devices, and is generally available to users wherever they can find Internet access.

The developers of Internet services embrace this new world. Innovations and features can be delivered to users on any given day, simply through an upgrade of service software, with no requirement of a download on the user’s part. Further, Internet services have adopted advertising-based business models, requiring no purchase on the part of a user. This compels service developers to ship innovations more frequently because there is no barrier for user adoption.

Given the continuous innovation approach, and the short development cycles that come with it, we arrive at an imposing problem: How does one release new software to a running Internet service, deploy it across thousands of nodes, and still avoid downtime?

We need predictable and repeatable software rollouts, and we specifically need to monitor the correctness of the actual software installation—independent of the correctness of the functionality and quality of the performance. Human error is a common theme in Internet service downtime. In our experience, human errors are common in the deployment and configuration of software across

thousands of computers. The daunting task of software rollout cries out to be completely automated and monitored.

Internet services tend to have many moving parts that, like an engine, need to be tuned for performance. Software engineers and operators exercise control through the configuration of the site. We refer colloquially to the configuration parameters as *knobs* that we can *tune*, evoking images of a scientific apparatus or an audio engineer's mixing board. Examples include the amount of memory that services should allocate for caching, the number of threads to use in a given process, maximum queue depths, etc.—any parameter that may impact the classic computer science speed-versus-space trade-off tends to show up as a knob that can be tuned.

Some implementers configure their systems through configuration files that are stored locally on each node. Others configure their systems through centralized configuration databases. The results are functionally equivalent: a single intended configuration exists, persisted, that should be applied consistently across all the nodes in the system.

As user bases grow, new features are added and access patterns change, requiring regular changes to the configuration to keep a service running optimally. How do you make a change to the configuration of a service while it runs, and correctly execute that change across thousands of nodes? What happens if a configuration change across thousands of nodes succeeds on all but four of the nodes?

Because failures in deployment across thousands of nodes are likely, services must monitor the exact configuration of every node. A master configuration specifies the intended configuration, and machines are constantly and automatically inspected to ensure the proper configuration.

ALARMING

When running a data center, things consistently go wrong, and humans are required to keep the data center operating smoothly. Given hardware and software failure rates, and multiplying by the large numbers of nodes that run in today's data centers, the math shows that failures will easily happen every day.

Consider the following example: Hard-disk MTTF (mean time to failure) estimates tend to be on the order of 1 million hours. Suppose a data center has 10,000 computers, each with 10 hard drives, for a total of 100,000 hard drives. Using a simplified model, assuming that hard disks all have uniform failure rates, and that new and older hard disks have the same failure rate (not the case

in practice), you would expect to have a hard drive fail every 10 hours. At Microsoft, our day-to-day operations do prove this math—it's not uncommon to have hard-disk failures on a daily basis in our large data centers.

Services want to make efficient use of humans, paying careful attention to the ways in which humans interact with the data center. As an Internet service grows, and its data center scales, businesses do not want to hire more humans as they add capacity to their data center. The scalability of a data center, in human management terms, is a direct function of the interaction points that humans

have with the systems. Since human involvement tends to be unnecessary when a computer is working, we find that the most common interactions occur when a system is failing. Internet services, therefore, focus on streamlining failure modes and recovery scenarios.

A first principle in alarming is that humans are easily overwhelmed with large amounts of data. A data center cannot escalate every single failure it sees to a human. Monitoring software needs to process data, aggregate it, and generally map what it sees into higher-level metadata about the system. As with most problems in computer science, an extra layer of indirection always helps—metadata is more useful than raw monitoring data in making automatic decisions.

We have seen situations where a single problem was

Monitoring, at Your Service

over-escalated, generating hundreds of independent escalations. In cases such as these, engineers are always overwhelmed, spending hours sifting through the data to understand root causes, filtering out false positives and looking for an underlying pattern. Our most streamlined situations are those where a problem caused just the right amount of escalations.

When stepping back to watch the day-to-day operations for an Internet service, one finds a fair amount of “procedure following.” Internet services bias themselves toward using well-known and mature hardware and software platforms. Over time, engineering teams gain experience running the system and tune it to the point where there are a small number of well-known failure modes, each of which repeat with relative consistency. Engineering teams assign a standard operating procedure to each of the known failure modes.

The motivation is to take guesswork out of the routine. If something goes wrong in the middle of the night, you

want to follow a clearly defined procedure—no analysis, no debugging, just fix the problem the way you have always fixed it in the past. Given this tendency, the problem cries out to be solved in software. Human operators are essentially expert systems whose brains are able to map experience in debugging the service and understanding of standard procedures into actions that bring a system back into a legitimate state.

The first step in applying monitoring to self-healing is to be able to enumerate and detect a service’s failure modes. If failure modes are ad hoc, not understood, or ever-changing, then there is no way for the system to heal itself without the aid of human brains. In practice, many failures require the same response to repair. The key is to identify all the necessary repair responses and then map them to detectable failures.

We have found that, in practice, there are specific failures that require specific responses. We have also found that there are groups of failures that can be handled with

a generic response. We bias ourselves toward systems that have a small and fixed set of access patterns, and thus a small and fixed set of failure modes. We bias ourselves toward keeping failure modes simple so that recovery is simple. We then map each of the failures to a specific recovery response. It is very difficult to build a system that is completely generic, but we have found that it is practical to have a generalized notion of mapping failures to recovery actions. Over time, we enhance our understanding of and ability to detect failures, and improve on our recovery actions.

The incident frequencies chart in figure 2

illustrates this example for Hotmail. Here, an *incident* is considered something that requires a recovery action of some form. Incidents are not necessarily outages or user-impacting—merely things that happen behind the scenes. The x-axis tracks 38 different types of incidents. The y-axis shows how many times the incident happened over a month. We can see that a small number of incidents happen frequently and that the majority of the incidents do not happen often.

Optimization proceeds on two dimensions. We focus on reducing the frequency and impact of the top incidents. We also focus on reducing the number of incident types—in this case, many incidents happen only once or twice, but in sum they contribute a significant number of incidents.

SELF-MANAGING DATA CENTERS

The goal for any large Internet service is to produce a self-managing system—one that “just works” and that is managed efficiently by a small team of engineers. In the limit, this reduces to a problem in artificial intelligence. Can we build a system that requires no human involvement?

The easiest place to remove human involvement is in the recovery from well-understood failure modes. Indeed, most successful Internet services have made determined progress in this area already. How do we make progress in the more dynamic areas? How do we improve the ability of a data center to adapt to changing user demands? How can we have software dynamically reconfigure the systems in a data center to provide optimal performance as users adopt newly released features?

Internet services with large user bases enjoy the support of the law of large numbers. With millions of users accessing the services, sampling techniques can extract usage patterns and detect changes in those patterns. We can further correlate changes to a service’s configuration with either positive or negative effects on the overall performance of the system.

Complexity is a real-world problem. To implement large-scale Internet services, development teams must strive ardently for simplicity. Dynamic and adaptive approaches can push the complexity of a system beyond reasonable means, to the point that it is impossible to predict results. With continued improvement in underlying software paradigms and infrastructure, however, we will be able to develop more dynamic systems without suffering from complexity.

CONCLUSION

Monitoring systems help to reduce human involvement

in Internet services. Monitoring software frameworks must cover a wide variety of problems stemming from measurement collection across thousands of nodes to software actuators that automatically stabilize systems. By using monitoring data programmatically, services can maintain process control, detect and isolate failures, maintain balanced resource utilization, and shield humans from the noise that is inherent in a complex system.

As Internet services mature, and large-scale systems used heavily by millions of people become better understood, we will make large gains in economies of scale. Businesses will be able to deploy thousands of cheap, commodity components, tie them together in distributed systems, and make massive amounts of computing power available to Internet users anywhere, anytime, and on any device. We will begin to see today’s relatively small number of Internet services grow in number. And we will find satisfaction as Internet services become as ubiquitous and taken for granted as today’s telephone network and electricity grid. Q

REFERENCES

1. Fox, A., and Patterson, D. 2003. Self-repairing computers. *Scientific American* (June); <http://www.sciam.com/article.cfm?articleID=000DAA41-3B4E-1EB7-BDC0809EC588EEDE>.
2. Shaw, M. 1995. Beyond objects: A software design paradigm based on process control. *ACM Software Engineering Notes* 20(11): 27-38; <http://www-2.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/ByndObj.pdf>.
3. Dijkstra, E.W. 1974. Self-stabilizing systems in spite of distributed control. *Communications of the ACM* 17(11): 643-644; <http://www-csag.ucsd.edu/teaching/cse291s03/Readings/p643-Dijkstra.pdf>.

LOVE IT, HATE IT? LET US KNOW

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

BILL HOFFMAN is a development manager in Microsoft’s Communications Services Platform group. At Microsoft, he has focused on software development in Internet services and systems of scale, from Web applications to Web services to storage systems to overall management of software in data centers. Prior to joining Microsoft, he was a software developer with Jump Networks, an Internet startup that was acquired by Microsoft in 1999. He holds a degree in computer science from Cornell University and resides in Berkeley, California.

© 2005 ACM 1542-7730/05/1200 \$5.00