Realtime publishers

Creating Unified IT Monitoring and Management in Your Environment

Don Jones

sponsored by

Copyright Statement

Chapter 5: Turning Problems into Solutions

The satirical news outlet *The Onion* recently ran a story related to the economy. In it, the publication claimed that a special kind of scientist called a *historian* was advancing the novel idea of *looking at the past.* "Sometimes," one pseudo-historian was quoted, "we can look at how people tried to solve problems which are similar to those problems we are having today. We can look and see how their solutions worked, and that can give us an idea of whether or not the same solution will work for us." Hah!

Although targeted at politicians who seem to keep making the same mistakes over and over, *The Onion's* jibe is pretty applicable to IT as well. "Look, if this same problem happened 3 months ago, and we solved it then, perhaps we can solve it more quickly now. What, exactly, did we do last time? Maybe doing the same thing again will have the same effect that it did then!"

I'll put it another way: Perhaps you have children, or at least know someone who does. Ever tell a kid not to touch the hot pot that's on the stove? Sure. Did they touch it? Of course. How many times? Usually just once. That's because human beings are designed to learn primarily by making mistakes. Provided we *remember* the mistake, and that we *remember* how to avoid it or solve it, we can do so in the future very quickly. Memory becomes the key factor, and as we get older, stop touching hot pots and start playing with computers at work, it sometimes gets harder to remember. This chapter is all about the final aspect of unified management: Taking problems that we've solved, and turning those into solutions for the future.

Closing the Loop: Connecting the Service Desk to Monitoring

Before we dive into the memory aspect of solving problems, we've first got to close the operational loop in our unified monitoring toolset. Earlier in this book, we discussed that one aspect of a unified management system is the ability to monitor devices and services, such as a database server. When a problem condition is monitored, the monitoring system creates an alert, which is typically shown on a console, and may involve notifying someone via email or text message. A truly unified system may also open a problem ticket in the organization's IT ticket-tracking system. The ticket enables management to track the problem and its time to resolution. It also allows the ticket to be passed to different personnel who collaborate to solve the problem. The ticket can even be pre-populated with information germane to the case, helping the person working the problem to get going more quickly. Figure 5.1 illustrates this first step: The alert showing up in the console, and the ticket being created from that.

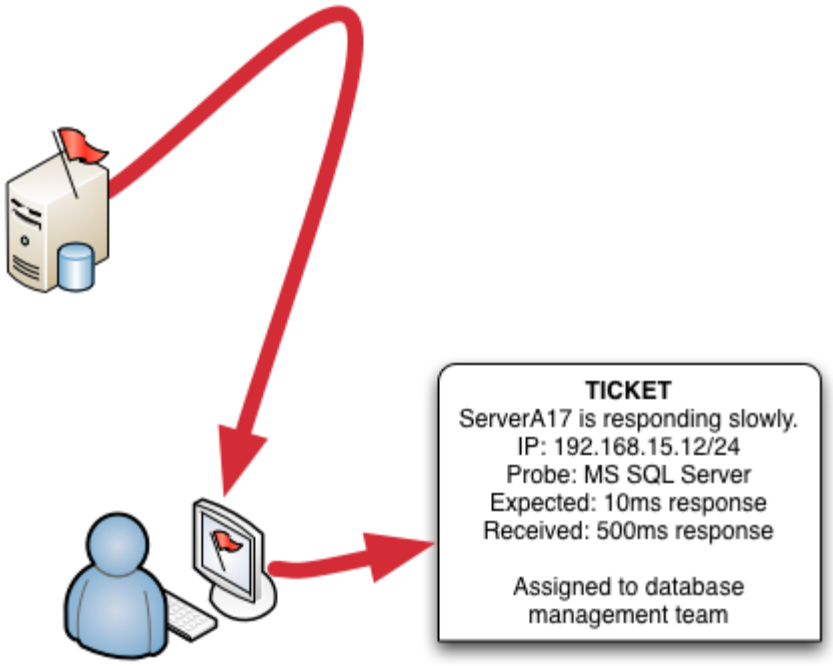

Figure 5.1: Getting an alert and opening a ticket.

Eventually, one hopes, the problem will be corrected. At that point, it's common for the person who completed it to close the ticket, marking it as completed.

But what about the alert?

Of course, the real-time monitoring component of the system will realize that the problem no longer exists—but that doesn't necessarily get rid of the *alert.* Typically, you want alerts left in-place until the problem is *confirmed* as being handled, which means that in addition to closing the ticket, you've also got to go in and clear the alert. This is actually pretty common in organizations that don't have a unified management system: Close the ticket in one system, then log into the monitoring system and mark the alert as handled. In a truly unified system, however, it would make sense for the closed ticket to also clear the associated alert because the alert is what created the ticket in the first place. Figure 5.2 shows how this loop can be closed within a single system.

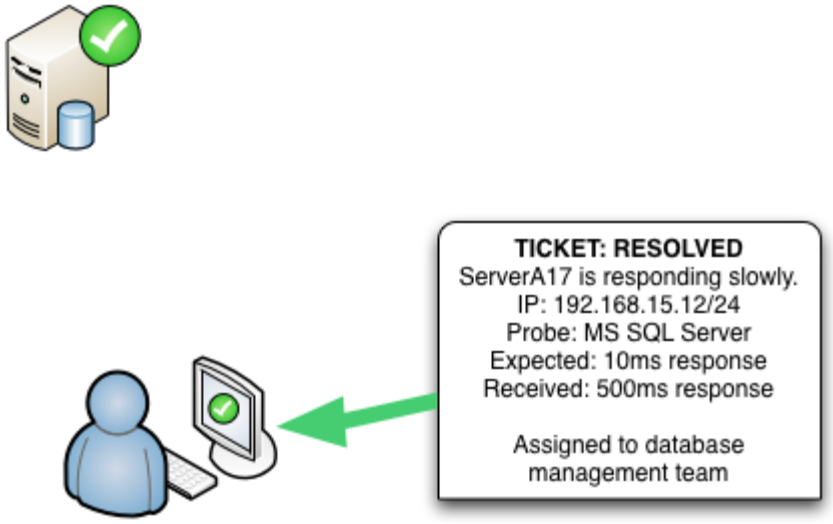

Figure 5.2: Closing a ticket clears the original alert.

There's a perfectly good reason to have alerts and tickets remain separate from each other. A ticket tends to be an internal-use-only type of thing. It contains technical information, intended for use in solving a problem and for reporting on that resolution process. An alert, however, is consumable by a wider range of people. An alert might surface in a companywide dashboard, for example, showing users that a given system is indeed impacted. You don't necessarily clear the alert just because the monitoring system is no longer seeing a problem, because temporary relief from a problem doesn't necessarily indicate a *resolved* problem. You might *want* the alert to remain in place as a high-level indicator that, "we know it's not working perfectly right now." But at some point you'll want to clear the alert and return the affected system to "operating normally" status; having that happen automatically as part of closing the ticket can be a convenient way to keep the two different audiences updated more easily.

Retaining Knowledge Means Faster Future Resolution

Once a problem is solved, it doesn't go away. At least, you *hope* it doesn't go away. As I pointed out in the beginning of this chapter, every problem solved is a potential turbo-boost for solving problems in the future—both that exact same problem as well as related ones. In other words, you want to retain information about the problem, as well as its solution, so that it can become useful in the future.

Knowledge Bases

Probably the oldest formal means of retaining this information is the *knowledge base.* Originally, these were separate databases, consisting of articles about how to solve problems. When you have a problem, you *first* search the knowledge base to see if any clues exist to help solve the problem.

One of the earliest knowledge bases to be widely distributed was Microsoft's, who made it available in the early 1990s on CD. Today, it's a massive collection of online articles—so massive, in fact, that there's actually a knowledge base article on how to query the knowledge base (shown in Figure 5.3, just in case you don't believe me).

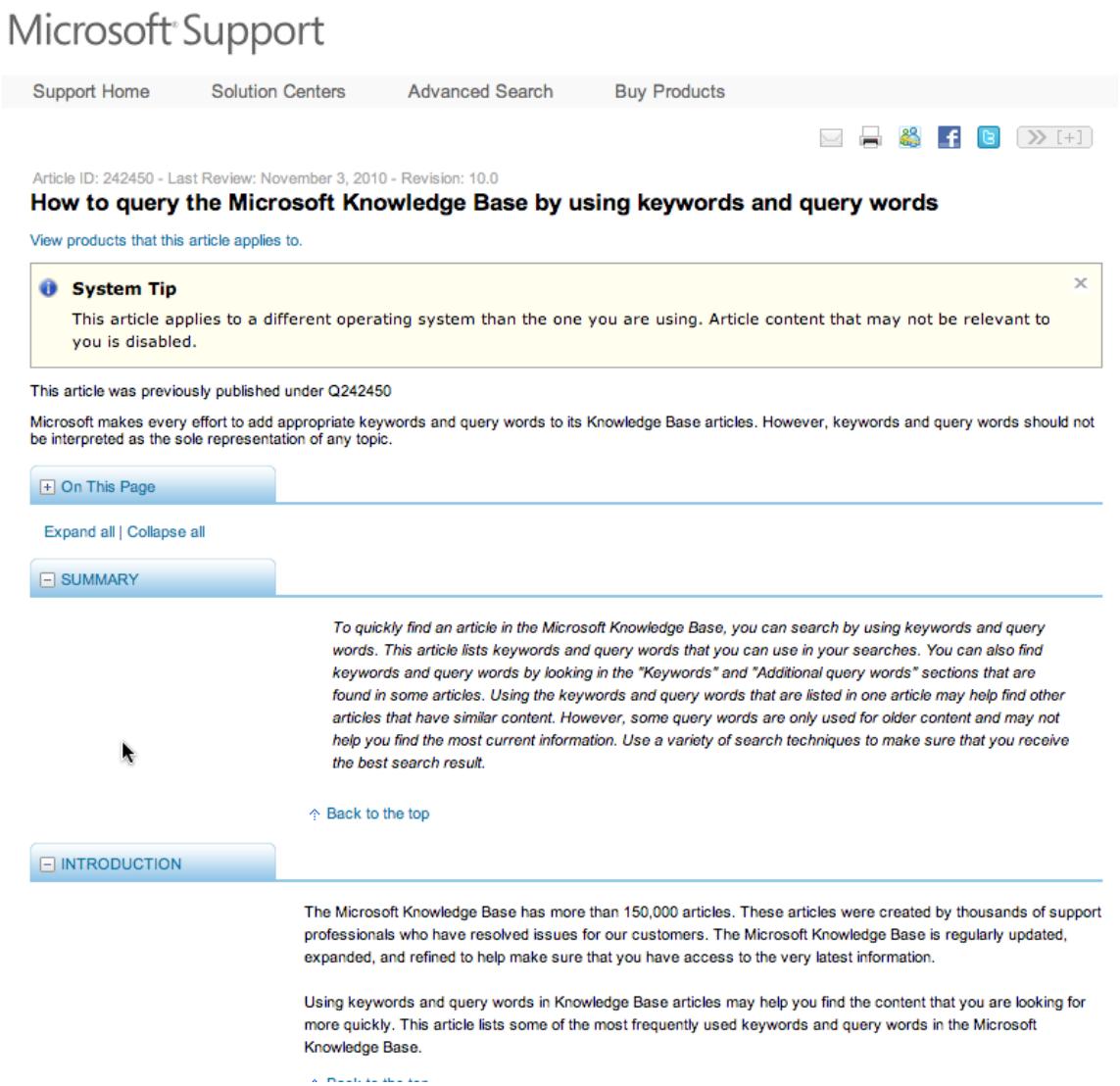

Figure 5.3: Microsoft Knowledge Base article.

This illustrates just one of the problems with a knowledge base: People have to learn to use it, and have to remember to use it. Unfortunately, IT professionals aren't necessarily the audience most likely to reach for the manual—or a knowledge base—when a problem crops up. They're a lot more likely to just dive in and try to use their own knowledge to solve the problem. Using the knowledge base—"search the KB," in the vernacular—usually only happens after they've exhausted their internal knowledge. Part of this attitude comes from their professional competency, part from the poor usability of most knowledge bases, and part from the fact that knowledge bases can get outdated pretty quickly.

Which illustrates another major problem with a knowledge base: The task of keeping it updated. Unless you're careful at the outset to tag articles with things like product versions and so forth, it can get really easy for the knowledge base to become a repository of *misinformation.* Consider a line of business application, version 1.5, that has a particular problem. You document that in a KB article, then rely upon that knowledge to fix the problem whenever it arises. Finally, your developers correct the problem in v1.6. Does anyone go back and update the KB article? No. Even if the KB article indicates that it applies to v1.5, it doesn't provide guidance beyond that. Was the problem fixed in v1.6? Will the same fix procedure work in v1.5? If you're using v1.6 and the problem occurs again, should you follow the v1.5 procedure or report the problem as a new one—since the developers thought they'd fixed it?

All of this presumes, of course, that you've addressed the *major* problem of knowledge bases: Getting articles into them in the first place. Vendors like Microsoft spend millions of dollars per year on the salaries of people who do little more than write documentation and contribute articles to knowledge bases. Are you willing to make that kind of investment? I've seen *many* companies set up a knowledge base, use it enthusiastically for a few months, and then let it slide and fall into disuse.

Tickets as Knowledge Base Articles

The first solution to many of the knowledge base's inherent problems was to simply discard the separate knowledge base and instead use closed tickets as a form of knowledge base. This is pretty much what every major ticket-tracking system these days offers.

This approach solves the primary knowledge base problem of how to get content into the system, because it simply re-purposes content that's already in the system: tickets. With a good ticketing system, it can also help solve the problem of "what this applies to," because your tickets are typically categorized with a specific product or service. So you'll at least know when you're reading an old ticket, what product and version it applied to—although you won't necessarily know if it *still* applies to the current version of that product or device.

Using Help desk tickets as a knowledge base doesn't solve the problem of getting people to actually search them for answers. In fact, using Help desk tickets *can* make the problem *worse.* Think about it: Every time a problem arises, a new ticket is created. So when you search the knowledge base (for example, the old tickets) using a keyword or by just selecting a product or device, you're going to get a lot more search results because you're going to be looking at *every* ticket that matches your criteria.

Help desk tickets don't always make a great source of self-service documentation, either. Not all IT folks are the best writers in the world, and tickets have a way of gathering… let's just call it "informal" language, which you might not want to surface to your end users. For example, a user who logs onto your self-service knowledge base, trying to solve a problem on their own rather than bugging your Help desk, might not be encouraged if the result they found said something like, "Rebooted stupid user's computer." Technicians might also provide very little in the way of detail. For example, it's not unusual to see "fixed" as the resolution to a ticket—not very useful for future reference. But using Help desk tickets as a knowledge base source isn't far off from the real solution.

Unifying the Knowledge Base

There are two things you can do to turn more Help desk tickets into useful knowledge base articles. First, you need some automation. Whenever a new ticket is created, the ticketing system should *automatically* go looking for related past tickets, presenting them as candidates to whatever technician is working the problem. One great example of this technique is used by the StackOverflow.com site—itself a kind of ticket/knowledge base combination—when you ask a new question. It automatically searches past questions and presents them to you in a visually interruptive fashion: They're inserted below your question, but *above* where you'd type the details of your question, as Figure 5.4 shows. It essentially forces you to review those suggestions so that you can quickly see if your question has, perhaps, already been answered.

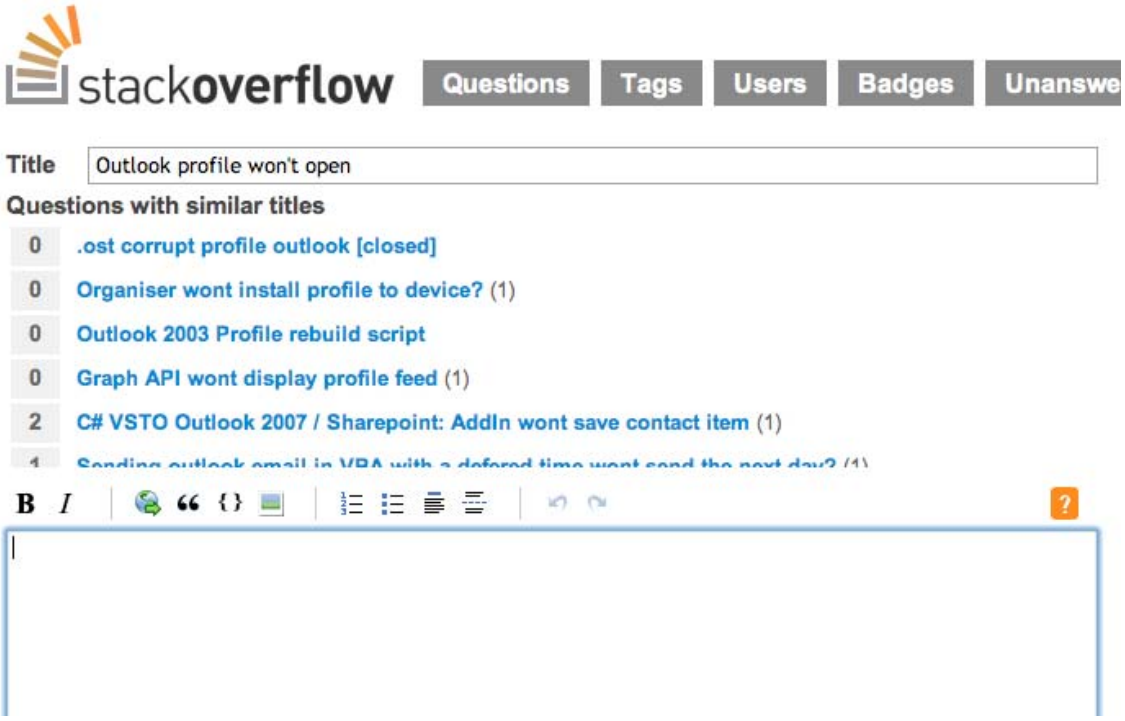

Figure 5.4: Suggested answers to a question.

As you begin to type your question's details, additional suggestions are shown off to the side, again helping you *use* the database of past answers rather than requiring you to explicitly search it in an extra step.

So a unified system can help take that extra step (see Figure 5.5). By including potentially-relevant tickets, or links to them, in with a newly-created ticket, the system can give technicians a jump start on solving the problem by calling their attention to similar situations in the past—along with the solutions to those situations.

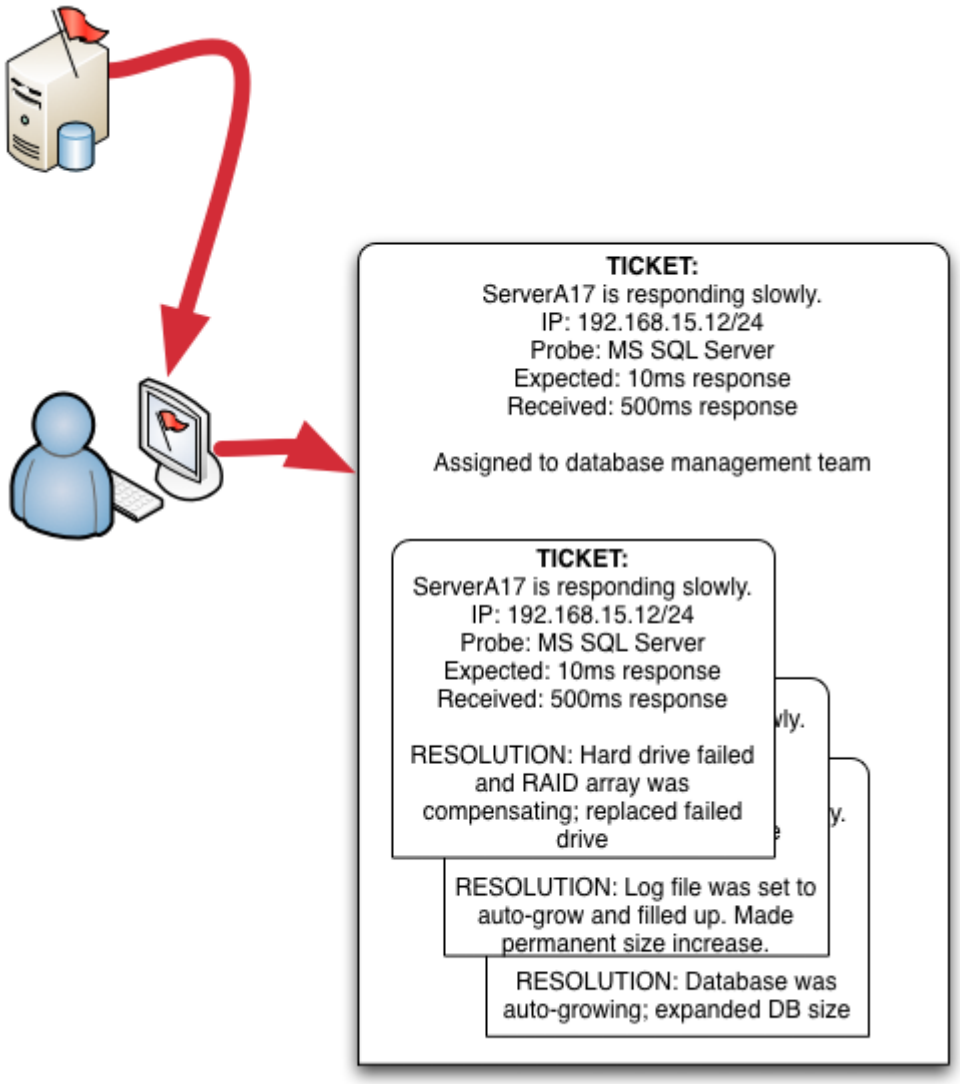

Figure 5.5: Using old tickets to solve new problems.

In fact, for tickets generated automatically from an alert, the ticketing system can potentially do a much better job of finding older tickets that actually relate to the problem. Because the system doesn't mind taking the extra steps, it can include more detailed search criteria, such as the nature of the problem, the device or service affected, and so forth. A technician might not think to include all the detail, which would net them a large set of search results, which is often what discourages them from searching in the first place. By getting a narrow result set to begin with, the automatically-referenced tickets are more likely to relate to the problem at hand.

Such a system can be made even better if the Help desk system includes a couple of check boxes in its tickets. When closing a ticket, a technician should be able to independently indicate:

- Whether this ticket contains a bona fide resolution to the problem. For example, sometimes the technician might solve the problem by looking at an older ticket, meaning the *current* ticket might not contain a lot of detail on how the problem was fixed. But if the technician has fixed the problem, and has filled in the details of what was done, then the current ticket can be marked as a "solution" ticket, making it bubble to the top of future search results.
- Whether this ticket contains an end-user consumable, self-service solution. Many ticket-tracking systems these days include both "public" and "private" notes fields, helping to ensure that end users don't see anything that they might find upsetting or insulting, while accepting the fact that technicians will put that stuff into a ticket sometimes. By having a specific indication of which tickets are consumable outside the IT team, and which ones specifically contain user-implementable solutions, you can build a self-service knowledge base that actually works.

Figure 5.6 shows how a system might implement this—in this case, rather than a checkbox, the system uses a "visibility" drop-down to change a ticket from being held to being published.

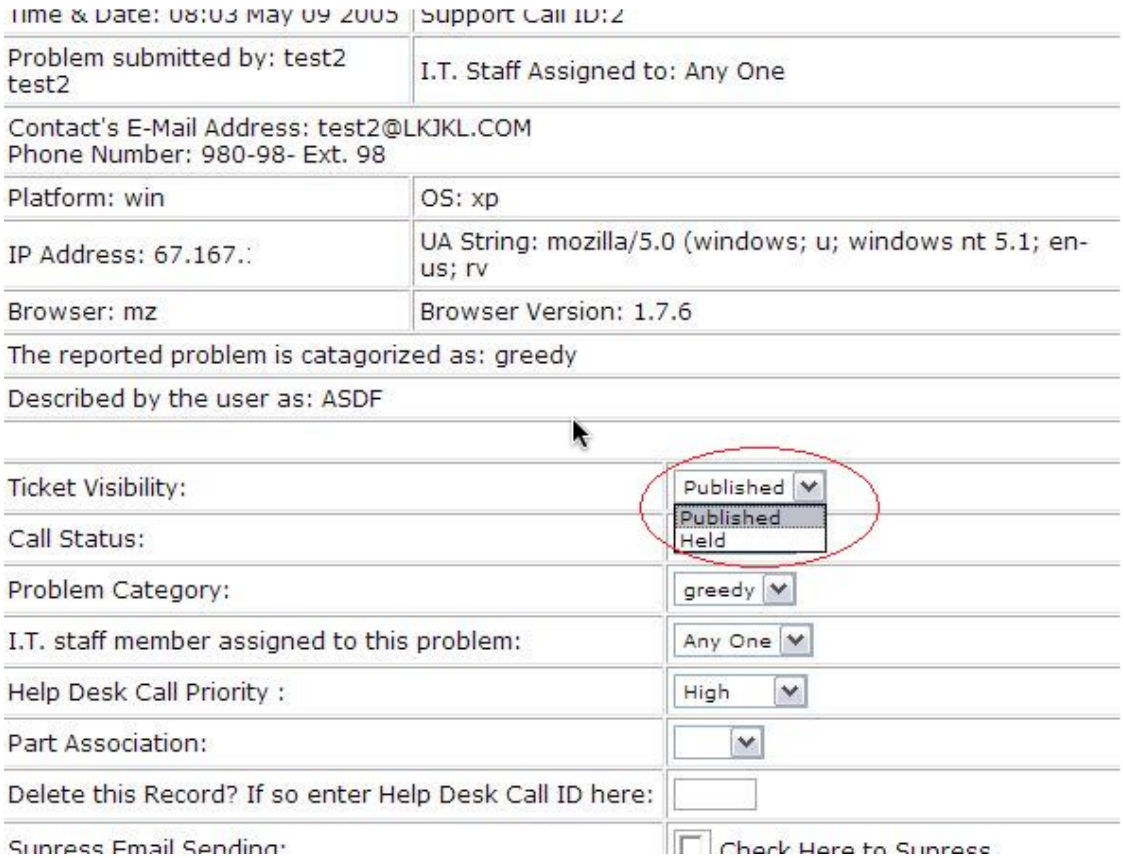

Figure 5.6: Controlling ticket visibility.

Simply the presence of these check boxes (or other indicators) can help to remind technicians that documented solutions are desirable. From a management perspective, organizations might set quotas for technicians: At least 75% of the tickets you close must either contain a detailed solution or refer to the ticket that does contain the solution. Metrics like that are manageable through ticket system reports, and can help ensure that tickets really do begin to serve as a basis for knowledge retention.

Making Tickets an Asset

The overall idea is to take your tickets from being a way of tracking problems and work to being a complete life cycle for problem-solving. In order to be effective, tickets-as-a-solution has to overcome some of the common human behaviors and implementation issues that have often been hurdles in the past:

- Technicians don't always search the ticket database—so that should happen automatically to some degree, with tickets being offered as potential solutions.
- Technicians' search skills aren't always that great—so a unified system should, using the information it already has at its disposal, make a first attempt at finding relevant tickets.
- Technicians' writing skills aren't always a priority—so the system should emphasize the need for complete solutions, management should focus on that as a metric, and technicians should be able to offer both "internal" and "externally-consumable" versions of a solution, when appropriate.

With the right system—particularly one connected to your monitoring system, making a truly unified environment—solutions to problems can be just a click away.

Past Performance *Is* an Indication of Future Results

Another way to use historical data is in developing service level expectations. I'm deliberately avoiding the phrase "service level *agreement*" because an SLA is a formal document that often includes some element of an organization's politics. A service level *expectation,* however, is the level of service that, based upon past performance, you can realistically expect to achieve in the future. An SLA will ideally be based upon those real-world expectations—*if* you can provide them.

One issue with many organizations' SLAs is that they're not actually based in reality. Someone will either make up an ambitious goal to "look good," like promising 99.999% availability and then boldly stating that they will just "manage to that number." Other times, someone will take an overly-cautious approach when establishing service levels, forcing the organization to expect a lesser level of service than they realistically could.

At fault are the tools we use. This gets all the way back to the first chapter of this book, when I wrote about the silos that IT tends to work in, and the varying domain-specific tools we rely on to troubleshoot and solve problems. Those same domain-specific tools are what we use to measure our existing performance levels. Because those tools don't all speak a common language or use a common set of metrics, it's actually really tough to figure out what our actual service levels are.

The bottom line is this: You have an existing environment. All political and internal issues aside, your existing infrastructure is capable of delivering some level of technically-measurable performance. You just need to discover what that is, using a common and easily-communicated set of metrics, based upon your infrastructure's current capability. You can't really do that by using a hodgepodge of domain-specific tools, though—and you really can't do it when your infrastructure starts to contain outsourced elements. Start bringing in cloud computing platforms, collocated servers, software as a service (SaaS) elements, and so on, and you'll find that your domain-specific tools just can't provide enough information. So how can you establish a good service level expectation?

This drives right back to the previous chapters in this book. Say you've got a complex set of services and applications—who doesn't these days? Figure 5.7 shows an infrastructure offering a lot of different elements, some inside the data center, some out.

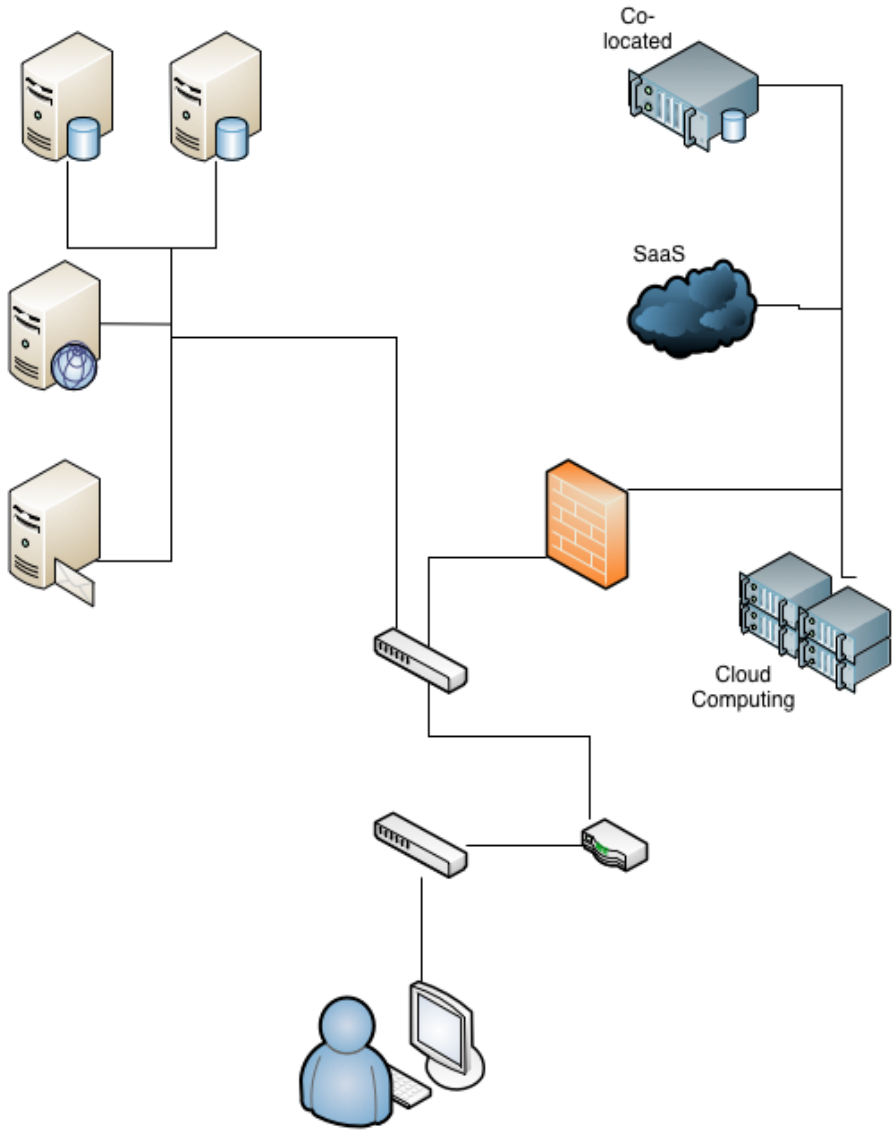

Figure 5.7: Modern environments include many different components.

You start your measuring at the one place it matters most: *the end user.* Put some probes, agents, synthetic transactions, or whatever else you need in place to figure out *what your users are actually seeing, today,* in terms of performance. Monitor that over several days that represent real, normal workload—no fair picking holidays as your day to monitor—and you'll know what your infrastructure is actually providing. It stands to reason that you can't expect any better but that you also shouldn't put up with anything worse. If that service level *expectation* isn't as good as your SLA—well, that's fine. You can start looking for areas where you can improve, bringing things up to that SLA level.

You'll also want to capture individual performance from each component—and this is where things can get tricky. It's crucial that, at this level of monitoring, you get everything onto a single console, in a single language, using a single set of metrics. What you're looking for is a performance *range* for each component that represents a normal workday. Provided each component operates within that observed range, you *should* be delivering the end-user experience that you measured. Those ranges provide the basis for your monitoring thresholds: Anything outside those ranges is something you need to be alerted to.

With that service level expectation established, you can start measuring different workload levels. See how things look on an especially busy day, for example, and what they look like on a light day (this is where it's okay to pick a holiday, for example). You'll start to get a feel for how your end user experience differs under those different workloads, and how the elements of your infrastructure change under different workloads.

Making sure that any outsourced elements are included in all of this is, of course, absolutely crucial. As I've pointed out in earlier chapters, monitoring those is a bit different than monitoring things that live inside your own data center. You'll either need a unified monitoring solution that's truly capable of hybrid monitoring, or you'll need a special set of tools to gather performance information on those outsourced pieces.

Notice that I've laid out two sets of metrics for you to monitor: performance and workload. Too often, I see SLAs that don't take the workload into account. "We will provide a 100ms response time." Okay—under what workload, specifically? Because maybe I can give you a 100ms response time under what I consider to be a normal workload, but if you start loading on additional users and functions, then that response time is obviously going to fall off. Again, monitoring solutions can help with this by not only measuring performance of things like processor, memory, disk, and so forth, but also *workload,* like the number of transactions being processed, the number of network packets being routed, and so on. It's important that your performance *expectations* include that workload context so that you can begin to make better service level *agreements* in the future.

It's the Performance *Database*

All of this performance data needs to not only be captured but also *stored.* That's where a lot of monitoring solutions miss the point: They're monitoring in real-time, and they're alerting to problems, but they're not always *saving* the data they see. Let's expand the example application to include that performance database—it's in Figure 5.8.

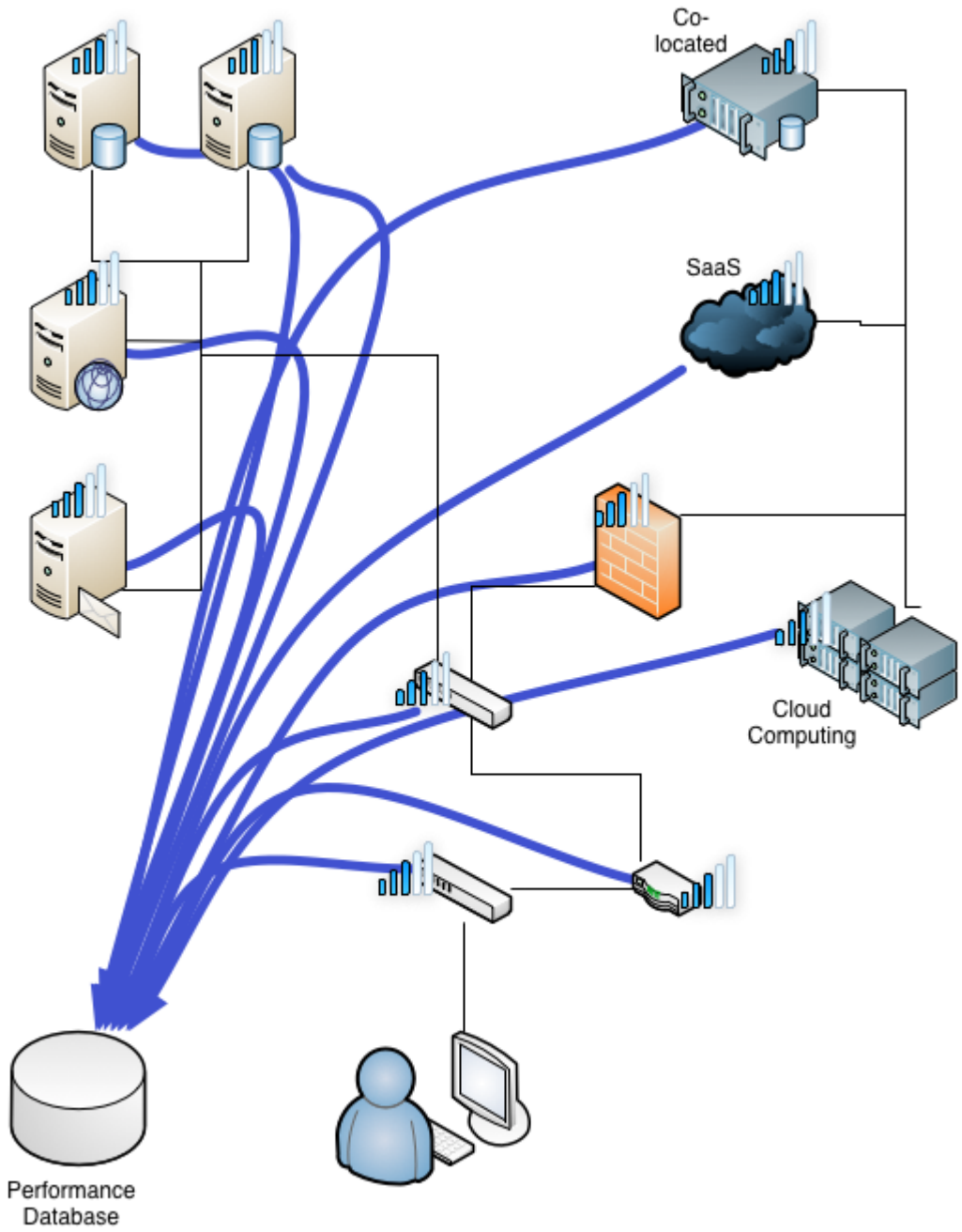

Figure 5.8: Adding a performance database to the environment.

The point of this figure is simply that you need to get performance data from *every component*—even the outsourced ones—into that database. Why? Two reasons:

- This database is what's going to show you what your performance really looks like on what you consider to be a normal day. This is where your performance *expectations* will come from, and it's hopefully what you'll use to derive more realistic and meaningful SLAs.
- This database is what's going to tell you when your performance is trending *away* from previously-established norms. I'm not referring here to a situation where one component's performance goes wonky due to a problem—live monitoring and alerting will take care of that. The database is there to spot the long-term trends: "Hey, did you know that performance is down 1% from last month, which was down .75% from the month before? At this rate, you'll be unable to meet your SLAs in 6 months."

And frankly, a good monitoring solution shouldn't even show you the prototypical "trend line" in performance as its first step. The first step should be a simple dashboard: "You're meeting your SLA, and based on current trends, will continue to do so for the foreseeable future." Or, "You're meeting your SLA—but barely, and based on trends, you aren't going to be able to meet your SLA for more than a month or two."

From *there*, you can drill down into graphs and charts that give you more detail so that you can find the component or components that look like the current bottleneck, and start making plans to get more capacity in place *before* you miss your SLA.

Summary

Embracing the past to make a better future—that's been the theme of this chapter. Whether you're gathering ticket-resolution information so that it can be used to solve future problems more quickly, or gathering performance information so that you can establish expectations and predict capacity, it's all about keeping historical data and leveraging it to put the organization on a better footing for tomorrow.

Coming Up Next…

In the last chapter of this book, we're going to step all the way back to the beginning and look at unified management from a case study perspective. I'll use my consulting and field experience to construct a composite case study, drawing elements of unified management together to show you what a modern, truly-unified environment can look like. I'll share specific problems from each environment, and explain how unified management helped solve those problems more quickly and effectively.

Download Additional Books from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this book to be informative, we encourage you to download more of our industry-leading technology books and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.