Hidden in Plain Sight: Practical Tips for Detecting and Fixing the Underlying Causes of Common Application Performance Problems

Applications have a direct impact on the business. When critical applications underperform, the business suffers.

Performance issues today are compounded by the ever-increasingly complex infrastructures that IT teams must contend with when delivering applications. Infrastructures that contain a combination of virtualized, hosted, and cloud environments. Applications that are increasingly distributed, multi-tiered, and consist of disparate components supported by a growing number of suppliers.

Making matters more difficult is the nature of today’s hybrid enterprises, where data and resources are often outside of a troubleshooter’s immediate control. The question of whether a problem is related to the network or the application itself can now extend to a SaaS, cloud, Internet, or other third-party provider’s environment.

As a result of these architectural complexities, detecting and fixing application performance problems has never been more difficult. Sluggish end-user transactions may present themselves as being slow due to the code. However, that slowness is often not the root cause, but rather a symptom of an underlying infrastructural issue hidden from view.

This field guide examines common, yet elusive application performance problems that reveal themselves only when you look at them from the right vantage point, and with tools that can capture all of the data, not just some of it. This guide is based on the real-world experiences of Jon Hodgson, a Riverbed APM subject-matter expert who has helped hundreds of organizations worldwide optimize their mission-critical applications.


Part 2, *Obliterating Haystacks*, shows how a big data approach can help you quickly pinpoint the needle in a haystack by removing the haystack from the equation.

Part 3, *The Power of Correlation Analysis*, explores a particularly vexing issue: seemingly random, intermittent slowness moving from one part of an app to another.

Part 4, *The Performance Trinity*, shows that while response time and throughput get all the attention, understanding load is the key to avoiding misdiagnosis and solving many issues.

Part 5, *Eliminating Leaks*, provides an overview of memory leaks and similar behaviors, and introduces some common approaches to troubleshoot leak-induced problems.

Part 6, *Troubleshooting Leak-like Behavior*, expands on concepts from the previous section, discussing how to troubleshoot other types of leak-like behavior.
Hidden in Plain Sight, Part 1: The Flaw of Averages

Quite often, examining CPU usage is considered a quick way to reveal application performance problems. But if you see usage at an all-too-perfect percentage, such as 25% or 50%, you likely have a problem. In the chaotic world that is computing, nothing runs naturally at exactly 25%. Something artificial must be happening there — and that’s a red flag.

Simple math explains why. Let’s say your server has four CPU cores. Something behind the scenes goes haywire. One core pegs at 100%, while the other three have low loads varying from 0% or greater. In your monitoring tool, you will see a variety of samples, but all with a too-perfect floor of 25%, meaning the values never dip below that value. Find the cause of the spiking utilization, and you’ll likely find a cause of slow applications.

Hiding in plain sight and the flaw of averages

CPU usage is a perfect example of a problem that’s hidden in plain sight by the flaw of averages. Professor Sam Savage first explained the concept in his October 8, 2000, article in the San Jose Mercury News. In that article he says: “The Flaw of Averages states that: Plans based on the assumption that average conditions will occur are usually wrong.” Figure 1 shows a humorous example involving the statistician who drowned while fording a river that was, on average, only three feet deep.

For application performance troubleshooting, why shouldn’t you rely on averages? Because there is always a significant variety in what you’re monitoring.

The fact is, the vast majority of transactions for most applications are likely OK. When you take that huge number of OK transactions and roll the poor-performing outliers into the statistics, everything will still look OK — the outliers are averaged out and hidden from sight.

Tools that collect only a sampling of data compound the flaw of averages problem. Combat this by taking a big data approach — collecting all the data with no sampling — to enable application teams to look at the actual distribution for every transaction for every user.

For instance, let’s say you’re measuring CPU load with Riverbed® SteelCentral™ AppInternals, which collects data every second. As Figure 2 shows, you’re intermittently hitting 100% CPU usage while other times exhibit heavy utilization. These spikes indicate where slowness will occur.
It’s a different story when you look at the exact same data using a tool with only 15-second granularity, as seen in Figure 3. No problem here, right? In both scenarios, the problem exists, but at 15-second granularity, you’re not aware of it. The information averages out and simply slips through the cracks.

Customer example: Forgotten freeware claims 10,000 CPUs

In a classic example from the trenches, Jon Hodgson was visiting a customer and immediately spotted a 16-core machine where CPU usage never dipped below a 6% floor. The company’s IT team thought the machine was just fine, and instead kept looking elsewhere for performance problems.

But, as you now know, that 6% floor should have been a red-flag example of the too-perfect number discussed above. (Quick math: 100% usage for the entire server, divided by 16 cores, is 6.25% per core.) Using SteelCentral AppInternals, Hodgson quickly discovered that a little freeware system administration utility, which hadn’t been updated since 2004, was single-handedly devouring one entire CPU of the 16 available.

Compounding the problem was the fact that the offending freeware utility was part of the default build for more than 10,000 company servers. The freeware utility was locking up a core on every one of those 10,000 servers, affecting thousands of applications and countless end-user transactions.

No one knew about it because it was hidden in plain sight, wasting resources and killing performance. But by looking at it with the right tool, SteelCentral AppInternals, the customer recovered processing time equivalent to 10,000 CPUs, and several unexplained problems immediately disappeared — at little cost or effort.

A final thought: Fixing problems reveals other problems

Fixing overarching performance problems can reveal other problems in two important ways. First, users change their behavior by starting to use the now better-performing service more frequently or in different ways. That puts new strains on different resources.

And second, troubleshooters will notice other problem areas previously hidden from view by the terrible effects of the first problem. To illustrate, picture an assembly line in which Joe takes 40 seconds to complete a task. When you fix Joe’s performance, you suddenly realize that Bob, the next person in the chain, wasn’t performing either, but he was previously masked by Joe’s inefficiency. This is often referred to as “the problem moving downstream.”
Hidden in Plain Sight, Part 2: Obliterating Haystacks

When it comes time to find that elusive needle in the haystack, the easiest method is to simply remove the haystack. Or, to relate that example to application performance management (APM), you need to remove the noise to reveal what matters.

Decoding hidden messages

To illustrate this point, Figure 4 shows a seemingly undecipherable jumble of letters. But when you use the right tool, in this case 3D glasses, a pattern emerges (Figure 5).

The haystack: Transaction noise

Here is a technical example. Most teams start by analyzing the slowest transactions, and then determine that the root cause is a slow piece of code. Figure 6 shows more than 2,000 transactions over an eight-minute timeframe. User complaints are pouring in, so the APM team zeros in on the transactions at 10:17 a.m. that take between seven and nine seconds.

But if the team fixes those slow transactions, will the end-user complaints stop? Keep in mind that just because certain transactions are the slowest does not mean they are the culprits affecting users most. The old logic rings true: Correlation does not imply causation.

The fact is, the data set in Figure 6 is a mix of many transaction types, each with their own typical and acceptable performance range. That blend of behaviors masks issues lurking just below the surface. To get at root-cause problems, you need to look deeper by removing the haystack to reveal the needles hidden in plain sight. And for that, you need a big data approach that captures all transactions.
So how does this work in practice? Figure 7 shows the same transactions, but now you can distinguish between different transaction types. The blue transactions normally take about four seconds, but for a minute, some take about two times longer. These were the transactions the APM team focused on.

The faster red transactions, in contrast, normally take about 250ms or less. But every minute or so, some take about 11 times longer — a much more severe problem than the original spike the APM team zeroed in on. Why? Three reasons:

1. It’s a much larger change in behavior.
2. It affects more transactions.
3. It occurs chronically.

In this case, the guilty party is an overloaded database — a completely different issue than the initial spike. Fixing that initial spike would not have cured the performance hit from the overloaded database. But solving the database problem would give you the biggest performance boost.

To slice and dice, you need all the data

To remove the haystack, you need a toolset that lets you slice and dice your dataset to reveal patterns not visible in the aggregate. If you capture only a subset of your transactions, you will solve only a subset of your problems. Therefore, it’s critical that your APM solution capture all transactions, all the time, rather than relying on an incomplete sampling approach.

“*If you capture only a subset of your transactions, you will solve only a subset of your problems.*”
– Jon Hodgson
Riverbed APM Subject Matter Expert

Customer example: The needle – a sawtooth pattern

Let’s demonstrate this point with a real example from Jon Hodgson’s archives. He was working with a financial organization that does banking, stock trading, and portfolio management. Regulatory demands required the APM team to ensure that the company’s applications could handle surges in traffic. In the past, trading microbursts had caused systems to lock up, which can have cascading effects on the stock market.
To see what the applications would do under severe stress, the team ramped up traffic on a production-parallel platform comprising hundreds of servers to 3X the peak load on the highest trading day. Under this test, you’d expect to see throughput increase proportionately, until it plateaus when some resource like CPUs or a database becomes saturated (see the red line in Figure 8). Instead, the IT team saw thrashing behavior, which indicated something completely different.

So the IT team wondered: What’s causing the stalling, then the drop in throughput, then a surge, culminating with thrashing? By the way, this was a death spiral from which the environment could not recover, which is an even worse situation.

To get to the root cause, Hodgson installed SteelCentral AppInternals, which was able to record all transactions, even at thousands of hits per second. He then compared the throughput chart from the load generator with the response-time chart from SteelCentral AppInternals, shown in Figure 9. Notice the distinct sawtooth pattern, and remember that rigid patterns like this indicate something artificial is happening.

Closer inspection also revealed that the sawtooth pattern preceded every burst of traffic. There’s a dip, a surge, a dip, a surge, and on and on. So the key was to figure out what caused the sawtooth pattern.

Next, Hodgson looked at a handful of slow transactions and noticed that many experienced delays when calling a remote web service named GetQuotes.jws — a stock-ticker application managed by a different team.

Remember, it’s important to be cautious when making assumptions. When you fix the slowest few transactions, you can’t assume you’ll also fix millions of other transactions. Just because the top-five slowest transactions are due to reason X, it doesn’t mean transactions six through 10,000 are also slow because of X.
But since it’s not practical to analyze 10,000 transactions individually, the better practice is to analyze them as an aggregate collection. Then, if the overall root cause is in fact reason X, you’ll prove that fixing it will correct all 10,000 transactions.

To test this theory, Hodgson focused on only those transactions that made remote calls to the GetQuotes.jws web service. The sawtooth remained, but many other transactions were filtered out, further confirming the initial hypothesis.

Although this information was compelling on its own, Hodgson wanted to be absolutely certain before making recommendations that would ultimately lead to a lot of engineering work and potentially cause political turmoil. So as a final confirmation, he tested this theory’s inverse – show only the transactions that do not call GetQuotes.jws:

Eureka! The sawtooth pattern and the trashing behavior completely disappeared. That confirmed the theory beyond a shadow of a doubt. GetQuotes.jws, a shared downstream web service, was the culprit for millions of slow transactions.

This example removed the haystack to reveal all the needles, allowing the APM team to figure out what the needles all had in common and identify the single component that needed to be fixed. The team then used SteelCentral AppInternals to determine that this issue affected hundreds of different transaction types in dozens of seemingly unrelated applications, which gave the business clear justification to fix it.

The customer’s own engineers had spent months investigating the code and the servers, but they couldn’t pinpoint the issue. But in a couple of hours with SteelCentral AppInternals, Riverbed located the problem, quantified its effects, and armed the customer with the evidence necessary to take action. This is just another way a big data approach, enabled by solutions such as SteelCentral AppInternals, can help reveal performance problems that are hidden in plain sight.
Hidden in Plain Sight, Part 3: The Power of Correlation Analysis

A common issue in performance troubleshooting is seemingly random, intermittent slowness moving from one part of your application to another. Without the proper tools and methodologies, you might not be able to identify the phantom’s root cause. This section highlights how to identify some of the most common (and uncommon) causes so you can expel these specters from your applications for good.

Don’t jump to conclusions

Consider this hypothetical key transaction shown in Figure 12, which normally takes 1.5 seconds to execute.

In real life, this transaction might contain thousands of method calls, but for the purposes of this example, it’s been simplified down into three main sub-steps of a half-second each. If an APM team has a service-level agreement (SLA) of 2 seconds for this transaction, when it’s exceeded, team members will receive an alert on their dashboard prompting them to drill down into the slow transaction to investigate further.

Comparing the slow instance to a typically fast one, it’s clear that the “FormatDetails” code is what got slower:

It’s easy to assume you now understand the problem. But be careful not to make that leap prematurely, lest you burden developers with fix requests that don’t correct the underlying issue(s).
To validate the initial theory, let’s investigate further by looking at multiple instances of the transaction:

It’s clear it would have been wrong to attribute the problem to just the “FormatDetails” method. But will attempting to optimize each of the three, sometimes-slow methods actually solve the problem?

Probably not – it may not be the code itself, or the code’s slowness may be a side effect of something else. You must figure out the common root cause that’s impacting the performance of all three components.

**Timing is everything**

In cases like this, you can’t analyze each problematic transaction independently. You need to consider each in the context of the rest. One key piece of context is understanding when each transaction ran relative to the others.

Here, the delays line up with each other, indicating that there’s likely some common root cause that took place during this time slice. Some shared resource like CPU may be saturated, or the disk might be queuing, or garbage collection may have paused the JVM/CLR.
Here’s an abstraction of what is likely happening:

Regardless of the specific reason, whatever unfortunate application code is running at that time will stall or slow down until it can access the resources it needs to finish executing.

Remember that code runs in a JVM/CLR, which in turn runs on the OS, which may be a VM running on a hypervisor, so the code is dependent on all the resources provided lower in the stack (Figure 17).

**Correlation, my dear Watson**

To determine why code is slow, you must also monitor the behavior and resource utilization of the entire stack. AppInternals typically monitors 5,000 metrics every second for each server in an application. You need a way to determine which of those metrics are related to the response-time symptom you’re seeing in the app. Luckily, AppInternals offers an easy way to achieve this: correlation analysis.
Start by picking a metric that is representative of your symptom. Overall response time for all transactions is usually a good one to use since it will factor in many different types of transactions, which may all be affected by a common root cause.

SteelCentral AppInternals dynamically analyzes all of the metrics to determine their typical range of behavior, depicted above as green samples. Orange samples outside of this range are treated as “deviations” and receive a score based on severity. Correlation analysis uses this information to scan through tens of thousands of metrics to find the handful of metrics that most closely resemble the symptom. If you can look at the response-time spike and figure out which metrics correlate, you can figure out what the ultimate root cause is. Some correlated metrics may be symptoms, others may be side effects, but the rest will be the root cause(s) you’re looking for.

The search will yield results you wouldn’t have thought of, or known, to look for. You’re no longer limited by your own expertise. A database admin can find issues in Java code and a developer can find issues deep in the operating system.
Put another way: think of the 150,000 metrics collected from an application as a pile of jigsaw puzzle pieces from 3,000 different puzzles. Correlation analysis is functionally a magnet that goes in and pulls out 50 related pieces. It says, “Here are the 50 pieces related to your problem, the rest is noise.”

Now all you need to do is assemble that simpler puzzle. Even if the pieces are outside of your area of expertise, you’ll still know exactly which subject-matter experts to enlist to help, and provide them with the actionable information they need to rapidly resolve it.

**Customer example: Analyze applications in the context of their environment**

A while back, Jon Hodgson encountered a similar problem at a customer’s site. The customer was monitoring two key applications on two different servers with another APM tool. The customer was experiencing intermittent slowness on their .NET application, but couldn’t determine root cause by using their existing tools.

An important tip before proceeding further into the example: be careful not to restrict yourself when troubleshooting. Humans have bias, and often make assumptions about root cause. You may have limited knowledge of the environment and application architecture, so don’t be too myopic on what you think might be related. Just as it is important to analyze transactions in the context of others, you need to analyze apps in the greater context of the environment in which they run.

Guided by this principle, Hodgson installed SteelCentral AppInternals to monitor both applications. He decided to take the response-time spike in the .NET application and correlate it across the board with all the metrics for both applications. This practice revealed something unexpected: there was a clear relationship between the .NET application and the seemingly unrelated Java application. Whenever response-time spikes occurred in the .NET application, they correlated to CPU spikes and garbage collection in the Java application.
Under provisioned and overcommitted

AppInternals revealed something else with its correlated VMware metrics. Contrary to the customer’s initial belief, both application servers were VMs running on the same hypervisor (Figure 22).

Instead of thinking about the servers as independent peers running side by side, think of them as part of the same stack since they share common resources. The Java application was doing garbage collection, consuming CPU and demanding more physical resources. The shared hypervisor was overcommitted so it had to rob Peter to pay Paul — stealing CPU cycles from the .NET application’s VM, thereby pausing the .NET application’s code execution until the hypervisor gave the resources back (Figure 23).

Figure 22

Although the symptom was in the response time of the .NET app, the root cause was in a seemingly separate Java app. However, the two apps were related because of shared physical infrastructure. Correlation analysis made this unknown relationship clear.

Figure 23
Visibility and trust issues are common with virtualized machines, since you don’t always know exactly where an app is running and if it’s actually been given the resources it was promised. VMware servers are commonly overcommitted since they’re often provisioned for the average resource needs of the guest VMs. But then the flaw of averages strikes when multiple guests demand the same resource in an unexpectedly small timeframe. With the wrong tools, these infrastructural issues may look like code issues, and you’ll spend an eternity tracking a problem you can never catch.

There are dozens of other usual suspects like garbage collection, saturated thread pools, and disk queuing, and hundreds of unusual suspects that could have just as easily been the root cause. That’s why it’s critical to have comprehensive monitoring of the code and its dependencies (JVM, OS, hypervisor, etc.) so that no matter what the root cause is, correlation analysis will be able to find it for you.

Next time you’re troubleshooting an issue and can’t find a consistent root cause, you’re probably looking in the wrong place. Take a step back, and use whatever tools you have at your disposal to look deeper into the underlying infrastructure.
Hidden in Plain Sight, Part 4: The Performance Trinity

What does “performance” mean to you?

Most people will respond to this with an answer revolving around response time, such as, “Our website pages should load in less than 2 seconds,” or, “We need to ensure that our SLA of 5 seconds is met.” These answers are fair, since one of the primary goals of APM is to keep customers happy, and the primary aspect of APM that is visible to end users is how long applications take to respond. That being said, response time is actually more of a side effect of performance. It is a symptom, not a root cause.

A smaller number of people will give an answer related to throughput, such as, “Our app needs to be able to handle 1,000 checkouts per second.” Throughput is a measure of how much work is being done by an application in terms of operations per unit of time, and is a great indicator of performance. However, it can be difficult to interpret throughput without greater context.

There are many textbook definitions for performance, but here’s the most accurate one in the context of APM: “The capabilities of a machine, vehicle, or product, especially when observed under particular conditions.”

Take special note of the last part: “…especially when observed under particular conditions.” Knowing the precipitating conditions is critical to being able to interpret the effects that you see. Unfortunately, few people take this into account when responding to the initial question posed above, and overlook the most important component of the answer: load.

The Performance Trinity

In order to adequately troubleshoot performance issues, you need to understand the relationship between load, throughput, and response time.

Load is where it all begins. It is the demand for work to be done. Without it, you have no work and, therefore, no response time. This is also what causes performance to ultimately degrade. At some point, there is a greater demand for work than the application is capable of delivering, which is when bottlenecks occur.

Throughput is the execution rate of the work being demanded. The relationship between load and throughput is predictable. As load increases, throughput will also increase until some resource gets saturated, after which throughput will “plateau.”

Many textbook definitions exist for performance, but the most accurate one in the context of APM is:

“The capabilities of a machine, vehicle, or product, especially when observed under particular conditions.”
When throughput plateaus, it’s an indicator your application no longer scales.

Response time is the side effect of throughput. While throughput is increasing proportionately to load, response time will increase negligibly, but once the throughput plateau is reached, response time will increase exponentially with the telltale “hockey stick” curve as queuing occurs:

**Response time isn’t everything**

From a technical perspective, response time is not actually an indicator of performance bottlenecks. Response-time thresholds and SLAs are human constructs to indicate when something is “too slow” for the end user. In the following example, the SLA is 3 seconds, which maps to 109 users; however, the saturation point maps to only 80 users.

While it is true to say that the application does not meet its SLA of 3 seconds for more than 109 users, that is only a business metric. Technically speaking, it is incorrect to say that the application doesn’t scale beyond 109 users, since it actually stops scaling at 80 users. The importance of this is not semantics, as it dictates how you should troubleshoot the issue and whether or not you will identify the root cause.

You need to determine why performance bottlenecked at 80 users (the technical breaking point). If you identify and solve that, you can solve the SLA problem at 109 users (the perceptual breaking point).
**Compare and contrast**

The relationship between load and throughput becomes increasingly important in modern, multi-tier applications. When a throughput plateau occurs, it may be visible across multiple tiers simultaneously (Figure 27).

Application teams have the natural tendency to blame downstream items in situations like this, because poor performance downstream usually bubbles upstream. While this is a true assessment for response time, it does not always apply to throughput.

By comparing throughput to load at each tier, you can determine what the root cause is. Here’s one possible scenario corresponding to the throughput above, where the load scales linearly at each tier (Figure 28).

In this case, since the load did increase continually at the database tier, one can determine that the bottleneck is indeed on the downstream database tier.

Another possible scenario is where the app server load increases linearly, but does not propagate to the database tier (Figure 29).

In this case, the bottleneck is actually in the application server, which is not passing the load down to the database server. Remember that load is the demand for work, and at some point the database is not being asked for anything additional. Without additional load, you won’t have additional throughput.

These two scenarios illustrate that you shouldn’t jump to conclusions based on an initial, limited dataset, but rather cross-reference your theory from other perspectives to ensure that it’s accurate. This will ensure you don’t waste valuable time trying to solve the wrong (perceived) problem. The woodworking adage of “measure twice, cut once” is equally applicable to APM.
Customer example: It’s not always the application’s fault

Here’s a real-world scenario where load was a critical factor to troubleshooting an issue. Jon Hodgson was working with a new customer who purchased SteelCentral AppInternals to help them troubleshoot a performance issue they’d been unable to solve previously, despite months of effort.

The customer would routinely run a standard ramp-load test against their Java application, and it would always fail because of massive response-time increases after a modest increase of load.

Figure 30 shows what the customer saw from the perspective of their load generator: load increasing linearly as they expected, and response time increasing gradually for a while until the “hockey stick” response-time spike occurred.

Hodgson then installed the SteelCentral AppInternals Java monitoring agents on the five front-end application servers, and the customer re-ran their load test, which again exhibited the same increase in response time. This time, however, they were able to see inside each of the five servers separately, which revealed a different issue (see Figure 31 on the following page).
Something quite unexpected appeared in the upper-right diagram from AppInternals. In the first half of the test, the load was distributed evenly among all five servers, but then a dramatic change occurred where almost all load was sent to a single server, at which point the response time on that one server increased 12x from 5 seconds to 1 minute. During that time, however, the other four servers had extremely fast response times.

The problem wasn’t an issue with application server performance at all, but rather load balancing. The load balancer was set to a simple “round-robin” mode, which is notorious for causing this type of issue (though one would think it would distribute load evenly). If one of the servers in a round-robin cluster is slightly less powerful or slightly more utilized, then the transactions that are blindly fed to it may start to pile up, resulting in the imbalance and overload shown here.

The solution was simple. The customer changed the load balancing policy to “least connections,” which ensured that in cases where one server fell behind, the others would pick up the slack. The next time the load test was performed it ran far better. The customer was able to generate about 4x the load before performance started to degrade, and they didn’t have to optimize a single piece of code or add any new hardware to experience those improvements.

The customer was previously so focused on the application servers and their downstream dependencies that they missed the root cause, which was one hop upstream. This is another great example where the right tools and the right methodology revealed the root cause of an issue hiding in plain sight with just a couple hours of effort.
If you’ve had any experience with application performance analysis, you’ve likely encountered leaks more times than you even realize. If your application needs to be routinely restarted to recover from degraded performance or complete failures after a certain amount of runtime, you likely have a leak.

Most people are familiar with memory leaks, but many different types of resources can exhibit leak-like behavior, requiring you to have a variety of approaches to troubleshoot them all.

What is a leak?

A leak occurs whenever an application uses a resource and then doesn’t give it back when it’s done. Possible resources include memory, file handles, database connections, and many other things. Even resources like CPU and I/O can leak if the calling code encounters an unexpected condition that causes a loop it can’t break out of, and then processing accumulates over time as more instances of that code stack up.

One critical concept to understand is that all leaks occur when some problematic block of code is running. That code can be triggered by an end-user web request, a background timer, application events, and many other things. The more frequently that code runs, the quicker the leak will accumulate.

A simple way to think about this is a person dealing cards onto a stack one by one. Although each card is very thin, the stack eventually grows to a significant height. The faster the cards are dealt, the faster the stack grows. If the dealer stops to chat for a moment, the stack stops growing until they resume dealing.

Do you have a leak?

Here are some signs that you most likely have a leak, or some leak-like behavior, in your application:

- Your app gets progressively slower over time, requiring routine restarts to resolve.
- Your app gets progressively slower over time, but restarts don’t help.
- Your app runs fine for a few days or weeks and then suddenly starts failing, requiring a restart.
- You see some resource’s utilization growing over time, requiring a restart to resolve the issue.
- You haven’t made any changes to the app’s code or environment, yet its behavior changes over time.

These are all serious issues that can negatively impact your application’s availability, stability, and performance. They may often cause many other phantom problems, which you could waste countless hours trying to solve unsuccessfully.

Why are leaks bad?

Leaks should be the number one item on your list to obliterate from your application. Different resource leaks have different effects on application behavior. Leaks for resources like connection pools and file handles have no negative impact on the application until they are fully starved, at which point the application will fail or be seriously delayed as it waits for non-leaking instances to become available.

Other leaking resources, such as Java virtual machine (JVM) heap, will have a slight impact on application performance as they become increasingly utilized, and the application will completely fail once they are starved.

When runaway threads consume CPU resources, there are fewer cycles available for other code, which will
become slower whenever those cycles are needed. This is also true of disk and network I/O.

In order to keep your application performing as efficiently as possible, you need to ensure it consistently has the resources it needs, whenever it needs them. Leaks present a moving target for application troubleshooters, since performance issues may simply be due to the cumulative effects of an existing leak, or an entirely new, unrelated issue.

It is critical to identify and eliminate leaks so they don’t cast a shadow of doubt over new issues that arise. A healthy application should experience no difference in performance between the hundredth and millionth execution of the same transaction. If it does, then there’s some sort of leak-like behavior occurring.

**Java and .NET memory leaks**

Due to the increasing prevalence of Java and .NET applications, memory leaks associated with these application types are the most frequently encountered. So it’s important to have a basic understanding of how Java and .NET manage memory.

For simplicity, we’ll focus on Java as an example, but .NET functionally behaves the same way.

In older languages like C++, developers directly access native OS memory, and manually allocate/de-allocate memory as needed. Java still uses native OS memory, but a large portion is reserved for managed “heap,” which is automatically managed by the JVM. Developers can then worry less about memory management and spend more time on code functionality. While this makes life easier for developers, it can abstract issues when troubleshooting.

If you look at your JVM memory usage with a traditional process monitor like *task manager* or *top*, the memory usage listed consists of the unmanaged memory used by the JVM itself and any native libraries, plus the entire managed heap reserved for Java code, which could be 1% utilized or 99% utilized. This ambiguity makes these tools ineffective at solving Java memory leaks. The only way to see the internal heap usage is to use a specialized tool that can see inside the JVM itself.

---

One critical concept to understand is that all leaks occur when some problematic block of code is running. That code can be triggered by an end-user web request, a background timer, application events, and many other things. The more frequently that code runs, the quicker the leak will accumulate.
Most Java monitoring tools will show you the percentage heap utilization. Different tools will report this data in different ways depending on how it’s collected, but it’s important to verify that, over time, you have a pattern that is either flat or increases and then decreases in a repeated cycle. If you ever see heap utilization generally trending upward, you most likely have a leak, and your application will fail once it hits 100%.

Knowing that you have a leak is only a tenth of the battle. If you want to determine the root cause of the leak, you’ll need an advanced tool that can see which objects are occupying the heap and leaking over time.

Many tools rely solely on heap “dumps” to determine the contents of the heap at a moment in time. These dumps require the application to be paused for several seconds, sometimes several minutes, which can result in outages for your users if you don’t have the ability to route them to other systems that are capable of continuing their existing session.

Dumps will typically reveal which objects are using the most heap, as shown in Figure 35.

This might lead you to believe that the newsfeed.story object is leaking since it’s using the most heap. However, there’s no way to know that it’s leaking without another dump to compare it to, so these tools will typically require you to take another dump (with another potential user outage) after an hour or so of runtime to determine what’s changed the most during that time.

Figure 36 show the comparison of two heap dumps that suggests that the newsfeed.story was likely a red herring, but the other two results don’t appear to be the root cause either. Although order.shippingDetails is the second-largest consumer of heap, its change was negligible. While health.pingResult changed the most, it was a relatively small consumer of heap.
In order to truly understand the heap usage patterns of this application, you’d need to take many snapshots over the course of a day or week and compare them to identify the trends. This is a laborious process that may result in difficult-to-interpret data and further user outages.

A better option is to employ a more advanced APM product that can monitor the heap utilization in real time and trend it historically. SteelCentral AppInternals can safely do this in production and reveal a much clearer story.

This chart shows the stacked history of the top-three consumers of heap since the application started 24 hours earlier. Newsfeed.story (red) appears to be some sort of cache. It grows to a finite size and then never increases after that. So even though it is the largest consumer of heap, it is an intentional behavior of the app and neither an issue nor a leak.

Order.shippingDetails (blue) is clearly the primary leak. The reason it didn’t appear significant in the dump comparison is because the leak is minimal overnight when the dumps were taken, but during business hours it grows more rapidly as users execute the code more frequently. The developers must correct the leak or the app will run out of memory every one to two days.

Health.pingResult (green) didn’t seem like an issue in the dump comparison, but here we can see that it’s steadily leaking throughout the day. Once the order.shippingDetails leak is fixed and the app is able to run for weeks at a time, the Health.pingResult leak will become more significant, causing a complete starvation of memory after just 15 days. This should be fixed at the same time as the primary leak.

As you can see, with the right tool, you can solve not only the current issue, but also address yet-to-be-discovered issues rapidly and with minimal effort.
Hidden in Plain Sight, Part 6: Troubleshooting Leak-Like Behavior

As stated earlier, many things other than Java and .NET heap memory can leak. This final section continues that topic and discusses how to troubleshoot other types of leak-like behavior, which traditional leak analysis tools and methodologies cannot address.

A deeper look into leaks

Tools that troubleshoot Java and .NET memory leaks do most of the heavy lifting for you, so you don’t need to have a deep understanding of how leaks behave in order to solve them. However, other types of leaks are less traditional, so you’ll need to have some fundamental knowledge in order to devise and employ innovative techniques to hunt them down.

As you’ll recall, leaks occur when some problematic block of code runs. This code may run when a particular URL is hit in a certain way or accessed by a certain user. The leaking code may be shared by multiple functions across multiple sub-applications, so there may be more than one precipitating cause. It may also run at scheduled times or in response to an event. Basically, there is some code, somewhere, that adds to the leak when it runs in a particular way.

Figure 38 shows a block of code with two states, either running or idle.

Typically, the code will use some resource (RAM, connections, file handles, threads, CPU, etc.) while it’s running, and then give that resource back when it completes, as shown in Figure 39.

However, when that code has a leak, the used resource isn’t given back, so it accumulates with each subsequent execution of that code (Figure 40).

Finding leaks and leak-like behavior

The best way to ferret out leaks in QA is to run a constant load of the same set of operations against your application for several hours or days. You can then compare the performance of each operation, as well as the application and server resource utilization between the first hour and the last hour, to see if any have significantly changed. While this is conceptually simple, there are hundreds of application and server resources that could leak, and thousands of pieces of code that could cause the leak. Sifting through all of this manually would be a painful and impractical task.

SteelCentral AppInternals provides powerful analytics to automatically identify which of the thousands of metrics changed the most between the beginning and end of your load test. This will allow you to rapidly identify which transactions and operations experience degraded performance, as well as what resources on which servers exhibit leaking behavior.

At the start of your load test, AppInternals observes the typical range of thousands of metrics to determine a “baseline” for
each of them. You then simply perform a “deviation” analysis at the end of your load test, and AppInternals will quickly reveal which metrics on which servers changed the most.

![Graph showing DB Open Connections, Home Page Resp. Time, and Java Heap Utilization](image)

Here, we can see that the home page’s response time is continually getting slower, and that corresponds to the number of database connections, which are continually increasing. In the real world, you’d likely see other deviations, which would fill in more pieces of the puzzle, such as runaway database queries that are causing the increase in open connections and are ultimately responsible for a slowdown in the database.

Also, notice that Java Heap Utilization shows no deviation, quickly proving this issue is not due to a Java memory leak. Ruling out components is equally important when troubleshooting issues, as it saves you countless hours trying to find issues where they don’t exist. In the real world, AppInternals would immediately rule out thousands of potential issues that are unrelated to the leak, leaving only tens of elements to investigate further.

Having solutions like this in your toolbox can save you hours or days of effort and make it possible to identify sources of leaks you wouldn’t think to look for or knew existed.

**Drips vs. floods**

The previous workflow is great at finding all leaks, but it will only find root causes if they exhibit the same sloped behavior. Sometimes, root causes have different patterns based on how they’re tabulated. The trick to finding these lies in the subtle changes of the leak’s behavior. This is a particularly challenging problem, so it warrants a deeper look to understand it.

Leaks don’t always grow at a constant pace, especially when they occur in production. If you trend a leak over time, you’ll often see periods where it grows more rapidly, or doesn’t grow at all (Figure 42).
Leaks grow relative to the rate of execution of the problematic code that causes them. The more rapidly the code runs, the quicker the leak will accumulate (Figure 43).

This relationship can initially be confusing, because the leak is typically measured as a “count,” but the root cause is often measured as a “rate.” Understanding this difference is critical to solving trickier leaks.

A “count” is the instantaneous value at a moment in time, such as “the website has 200 active requests,” or “the DB connection pool has 50 active connections.”

A “rate” represents the change in a value over a range of time, such as “the website gets 10 requests per second,” or “the DB gets 10 new connections per second.”

Rates and counts are often calculated from the same underlying data, so they’re just different sides of the same coin, but they admittedly look wildly different. Although the leak looks like a slope, its root cause often will not, so don’t limit yourself to just looking for the former if you hope to identify what’s causing your leak.

By noticing the changes in the pattern of the leak and exploiting them, you can identify these trickier, rate-based root causes with ease.

Rather than specifying a particular period of time to baseline, just leverage SteelCentral AppInternals’ default behavior to continually tune its baseline based on recent history. Then, all you need to do is target your deviation analysis at the time right after the leak’s pattern changes.

By using this targeted approach, the contributing factor(s) whose change in behavior caused the change in the leak’s behavior will become immediately evident. Although this is a very complex problem, SteelCentral AppInternals makes the solution quite easy (again, pulling the needles out of the haystack).
Troubleshooting your leaks

Now that you have a good understanding of what leaks are and are equipped with some ways to find them, let’s expand upon each of the common signs you have a leak or leak-like behavior in your application.

Your app gets progressively slower over time, requiring routine restarts to resolve.

In this type of leak, as it worsens, the impact to the application’s performance or behavior worsens as there is competition for the leaking resource. This may be due to transactions or operations that remain running in the background after you think they’ve completed. For example, an unhandled exception might throw the transaction into an unending loop, so even though the end user may see an error and think the request ended, some code actually keeps running behind the scenes.

For this type of leak, you’ll typically see a gradual increase in the concurrent execution of particular transactions, or sub-parts of the transaction, such as specific downstream web service calls or database queries. You’ll also see an increase in utilization of a key resource like CPU or disk I/O, which is ultimately where the contention is occurring. To solve this issue, you’ll need to enhance your code to handle the exception better or detect runaway transactions and terminate them after a timeout occurs.

Your app runs fine for a few days or weeks and then all of a sudden starts failing, requiring a restart to resolve the problem.

This one can catch you off guard if you’re not looking for it, since the leak doesn’t negatively impact the performance or behavior of the application until the resource is almost (or completely) saturated. This type of leak often holds on to resources like web server threads, DB connection pool connections, file handles, etc. When needed, the resource is opened, but after completion that resource isn’t closed — sometimes because an unhandled connection or unexpected code path prevents a graceful close. These resources are not being actively used, so there’s no effect on response time or CPU utilization, but once the resource is saturated, transactions that request that resource will stall or fail. To solve this issue, you’ll need to enhance your code to better handle the exceptions or unexpected code paths to ensure unused resources are returned to their pool. Another option is to have background housekeeping routines that hunt for these leaks and return them back to their pools.

You see some resource’s utilization growing over time, requiring a restart to resolve the issue.

This could be either of the previous leaks, but identified based on the leaking resource itself rather than the end-user effects of the leak. You’d troubleshoot them the same ways.

You haven’t made any changes to the app’s code or environment, yet its behavior changes over time.

This is a high-level abstraction of the effects of any type of leak or leak-like behavior. Your end users likely won’t report that they think the app has a leak, but they’ll say things like “it seems a lot slower lately,” or “it fails much more than it used to.” Listen for hints like this that indicate there’s probably a leak you need to hunt down.

Your app gets progressively slower over time, but restarts don’t help.

This type of issue is the reason the phrase “leak-like behavior” exists. While not technically a leak, they can have a similar effect on performance. The typical reason for this is the growth of a data store over time, but a lack of an index for subsequent lookups against that store.

The most common example is a database table that is continually growing, so queries against that table get progressively slower as “table scans” occur and take longer due to a lack of an index. This will often reveal itself in long load tests at a constant load, where both inserts and reads are occurring on the same table. It’s easy enough to fix by adding the required index.
A rarer example is when new files keep getting added to same folder, such as a large PDF or image repository that grows daily. Some file systems don’t have an index of the files in a folder, so they have to perform the equivalent of a database table scan to find the one you’re looking for. This is normally an instantaneous operation, but when there are tens of thousands of files in a single folder, this can cause seconds of delay, especially with concurrent lookups. Rectify this by creating a nested directory structure that divides the files up into hundreds or thousands of sub-folders. Conceptually, you’d store the file “Bob’s Report.pdf” in the \reports\b\bobs\bobsr\ folder. Although in the real world, the directory names would be based on a hash algorithm, so you’d know exactly where to look for it without having to scan through thousands of files.

Identifying leaks, eliminating problems, and delivering high-performing apps

Leaks are incredibly common and can be very difficult to hunt down due to a wide variety of reasons and the almost infinite number of places they can occur. Adding a solution like SteelCentral AppInternals to your arsenal will allow you to go beyond the limitations of traditional Java and .NET heap analysis and identify any leak-like behavior in your application so that you can ensure consistent performance and availability for your users.
Conclusion

The scenarios covered in this guide may sound like esoteric corner cases, but in reality they’re actually quite common. Just because you haven’t seen something yet doesn’t mean it doesn’t exist; hence the title of this series, *Hidden in Plain Sight*. To ensure you don’t overlook or misdiagnose issues, follow these key takeaways:

- Always be mindful of “the flaw of averages,” which can mask problems. Ensure that you have adequate monitoring resolution to catch the issue, and that your troubleshooting workflows effectively leverage that data.

- The reason why “the slowest thing is slow” may not be the reason the application is slow overall. Use a big data approach to APM and analyze your transactions holistically to determine overarching causes of delay and target those areas for optimization first.

- Code that runs slowly may not be that code’s fault. Be cognizant of the environment your code runs in and remember it’s dependent on the JVM/CLR, OS, and hypervisor it runs on. If any of those dependencies become saturated, your code will suffer.

- Learn to differentiate symptoms from root causes. Don’t jump to conclusions based on initial findings. Take a step back, look at the problem from a different angle, and confirm your theory to minimize wasted time and effort chasing phantoms.

- Don’t make assumptions. Verify everything.

- Make sure you’re getting the load you expect. Verify that the load is being distributed the way you assume it should and that it’s propagating down the tiers as designed. You can’t interpret your output until you’ve verified your input. A stall on the way to the data source is a different issue than a delay incurred on the way back.

- You need to capture the details of all transactions, 24x7, to identify overarching reasons for delay and not be mislead by supposedly “representative” sampling approaches. If you capture only a subset of your transactions, you will solve only a subset of your problems.
Take Control of Application Performance with SteelCentral AppInternals

Continuously monitor your applications. Trace all transactions, across all users, all the time to get the most comprehensive end-to-end application visibility and analytics for rapid problem resolution and continuous improvement. Deliver high-performing applications around-the-clock.

STAY AHEAD OF PERFORMANCE PROBLEMS

You’re only two clicks from an answer to virtually any performance problem. Monitor user experience, applications, infrastructure, and key business transactions from an interactive web-based dashboard.

RECONSTRUCT & DIAGNOSE ALL INCIDENTS

Trace every transaction from the user device or browser to the application backend, and capture second-by-second system metrics. Reconstruct and analyze incidents in great detail to fix code, SQL, infrastructure, or remote calls.

PROACTIVELY IMPROVE PERFORMANCE

Analyze billions of metrics using simple queries to expose bugs, to discover anomalies, to derive business insights or to plan for capacity. Explore the dataset and continuously improve performance.

COLLABORATE ACROSS THE APP LIFECYCLE

Measure impact and share performance insights across development, QA, and production. Detailed diagnostics help development and support teams fix problems without having to reproduce them.

SEE THE COMPLETE PICTURE

Integrate with Riverbed SteelCentral Portal to get end-to-end visibility and analytics across applications, networks, and infrastructure. Get the complete picture that other point APM tools cannot deliver.

TRY APPINTERNALS FOR FREE!

Go to www.appinternals.com to sign up to try AppInternals for free. To reach us by phone, call 1-87-RIVERBED.
Authored by:

Jon C. Hodgson is an APM subject matter expert for Riverbed Technology. For over a decade, Jon has helped hundreds of organizations around the world optimize the reliability and performance of their mission-critical applications. With a background in Systems Administration, Networking, Programming and Application Architecture, Jon’s multidisciplinary approach to troubleshooting has enabled him to analyze and solve a wide variety of problems ranging from typical code/sql inefficiencies to more complex issues related to resource saturation, network effects and low-level OS/virtualization bottlenecks. When he’s not obsessing about how to make things perform faster he enjoys digging things up with his tractor at his home in Missouri.

Field guide compiled by:

Mason Coffman is the content marketing manager for Riverbed’s Platform Marketing team. In this role, he develops thought-provoking content that provides Riverbed’s customers with valuable and consistent information as they plan for and complete strategic IT and business initiatives. With nearly a decade of B2B marketing experience, Mason has covered a variety of topics, including digital learning content, enterprise mobility, cloud computing, application and network performance management, and more.

About Riverbed
Riverbed, at more than $1 billion in annual revenue, is the leader in application performance infrastructure, delivering the most complete platform for the hybrid enterprise to ensure applications perform as expected, data is always available when needed, and performance issues can be proactively detected and resolved before impacting business performance. Riverbed enables hybrid enterprises to transform application performance into a competitive advantage by maximizing employee productivity and leveraging IT to create new forms of operational agility. Riverbed’s 26,000+ customers include 97% of the Fortune 100 and 98% of the Forbes Global 100. Learn more at www.riverbed.com/appinternals