Top 5 Java performance metrics, tips & tricks

APPDYNAMICS

Chapter 1: Getting started with APM 4

Chapter 2: Challenges in implementing an APM strategy 9

Chapter 3: Top 5 performance metrics to capture in enterprise Java applications 14

Chapter 4: AppDynamics approach to APM 20

Chapter 5: APM tips and tricks 25

Chapter 1

Getting started with APM

Chapter 1: Getting started with APM

Application Performance Management, or APM, is the monitoring and management of the availability and performance of software applications. Different people can interpret this definition differently so this article attempts to qualify what APM is, what it includes, and why it is important to your business. If you are going to take control of the performance of your applications, then it is important that you understand what you want to measure and how you want to interpret it in the context of your business.

What is Application Performance Management (APM)?

As applications have evolved from stand-alone applications to client-server applications to distributed applications and ultimately to cloud-based elastic applications, application performance management has evolved to follow suit. When we refer to APM we refer to managing the performance of applications such that we can determine when they are behaving normally and when they are behaving abnormally. Furthermore, when someone goes wrong and an application is behaving abnormally, we need to identify the root cause of the problem quickly so that we can remedy it.

We might observe things like:

- The physical hardware upon which the application is running
- The virtual machines in which the application is running
- The JVM that is hosting the application environment
- The container (application server or web container) in which the application is running
- The behavior of the application itself
- Supporting infrastructure, such as network communications, databases, caches, external web services, and legacy systems

Once we have captured performance metrics from all of these sources, we need to interpret and correlate them with respect to the impact on your business transactions. This is where the magic of APM really kicks in. APM vendors employ experts in different technologies so that they can understand, at a deep level, what performance metrics mean in each individual system and then aggregate those metrics into a holistic view of your application.

The next step is to analyze this holistic view your application performance against what constitutes normalcy. For example, if key business transactions typically respond in less than 4 seconds on Friday mornings at 9am but they are responding in 8 seconds on this particular Friday morning at 9am then the question is why? An APM solution needs to identify the paths through your application for those business transactions, including external dependencies and environmental infrastructure, to determine where they are deviating from normal. It then needs to bundle all of that information together into a digestible format and alert you to the problem. You can then view that information, identify the root cause of the performance anomaly, and respond accordingly.

Finally, depending on your application and deployment environment, there may be things that you can tell the APM solution to do to automatically remediate the problem. For example, if your application is running in a cloud-based environment and your application has been architected in an elastic manner, you can configure rules to add additional servers to your infrastructure under certain conditions.

Thus we can refine our definition of APM to include the following activities:

- The collection of performance metrics across an entire application environment
- The interpretation of those metrics in the light of your business applications
- The analysis of those metrics against what constitutes normalcy
- The capture of relevant contextual information when abnormalities are detected
- Alerts informing you about abnormal behavior
- Rules that define how to react and adapt your application environment to remediate performance problems

Why is APM important?

It probably seems obvious to you that APM is important, but you will likely need to answer the question of APM importance to someone like your boss or the company CFO that wants to know why she must pay for it. In order to qualify the importance of APM, let's consider the alternatives to adopting an APM solution and assess the impact in terms of resolution effort and elapsed down time.

First let's consider how we detect problems. An APM solution alerts you to the abnormal application behavior, but if you don't have an APM solution then you have a few options:

- Build synthetic transactions
- Manual instrumentation
- Wait for your users to call customer support!?

A synthetic transaction is a transaction that you execute against your application and with which you measure performance. Depending on the complexity of your application, it is not difficult to build a small program that calls a service and validates the response. But what do you do with that program? If it runs on your machine then what happens when you're out of the office? Furthermore, if you do detect a functional or performance issue, what do you do with that information? Do you connect to an email server and send alerts? How do you know if this is a real problem or a normal slowdown for your application at this hour and day of the week? Finally, detecting the problem is one thing, how do you find the root cause of the problem?

The next option is manually instrumenting your application, which means that you add performance monitoring code directly to your application and record it somewhere like a database or a file system. Some challenges in manual instrumentation include: what parts of my code do I instrument, how do I analyze it, how do I determine normalcy, how do I propagate those problems up to someone to analyze, what contextual information is important, and so forth. Plus you have introduced a new problem: you have introduced performance monitoring code into your application that you need to maintain. Furthermore, can you dynamically turn it on and off so that your performance monitoring code does not negatively affect the performance of your application? If you learn more about your application and identify additional metrics you want to capture, do you need to rebuild your application and redeploy it to production? What if your performance monitoring code has bugs?

There are other technical options, but what I find most often is that companies are alerted to performance problems when their custom service organization receives complaints from users. I don't think I need to go into details about why this is a bad idea!

Next let's consider how we identify the root cause of a performance problem without an APM solution. Most often I have seen companies do one of two things:

- Review runtime logs
- Attempt to reproduce the problem in a development / test environment

Log files are great sources of information and many times they can identify functional defects in your application (by capturing exception stack traces), but when experiencing performance issues that do not raise exceptions, they typically only introduce additional confusion. You may have heard of, or been directly involved in, a production war room. These war rooms are characterized by finger pointing and attempts to indemnify one's own components so that the pressure to resolve the issue falls on someone else. The bottom line is that these meetings are not fun and not productive.

Alternatively, and usually in parallel, the development team is tasked with reproducing the problem in a test environment. The challenge here is that you usually do not have enough context for these attempts to be fruitful. Furthermore, if you are able to reproduce the problem in a test environment, that is only the first step, now you need to identify the root cause of the problem and resolve it!

So to summarize, APM is important to you so that you can understand the behavior of your application, detect problems before your users are impacted, and rapidly resolve those issues. In business terms, an APM solution is important because it reduces your Mean Time To Resolution (MTTR), which means that performance issues are resolved quicker and more efficiently so that the impact to your business bottom line is reduced.

Evolution of APM

The APM market has evolved substantially over the years, mostly in an attempt to adapt to changing application technologies and deployments. When we had very simple applications that directly accessed a database then APM was not much more than a performance analyzer for a database. But as applications moved to the web and we saw the first wave of application servers then APM solutions really came into their own. At the time we were very concerned with the performance and behavior of individual moving parts, such as:

- Physical servers and the operating system hosting our applications
- JVM
- Application server behavior
- Application response time

We captured metrics from all of these sources and stitched them together into a holistic story. We were deeply interested in garbage collection behavior, thread and connection pools, operating system reads and writes, and so forth. Not to mention, we raised fatal alerts whenever a server went down. Advanced implementations even introduced the ability to trace a request from the web server that received it across tiers to any backend system, such as a database. These were powerful solutions, but then something happened to rock our world: the cloud.

The cloud changed our view of the world because no longer did we take a system-level view of the behavior of our applications, but rather we took an application-centric view of the behavior of our applications. The infrastructure upon which an application runs is still important, but what is more important is whether or not an application is able to execute its business transactions in a normal fashion. If a server goes down, we do not need to worry as long as the application business transactions are still satisfied. As a matter of fact, cloud-based applications are elastic, which means that we should expect the deployment environment to expand and contract on a regular basis. For example, if you know that your business experiences significant load on Fridays from 5pm-10pm then you might want to start up additional virtual servers to support that additional load at 4pm and shut them down at 11pm. The former APM monitoring model of raising alerts when servers go down would drive you nuts.

Furthermore, by expanding and contracting your environment, you may find that single server instances only live for a matter of a few hours. I have heard of one large cloud-based application that uses a very large amount of RAM in its JVMs, but its recycling strategy ensures that those servers are shut down before garbage collection ever has a chance to run. This might be an extreme example, but it illustrates that what was once one of the most impactful performance issues has been rendered a non-issue by a creative deployment model.

You may still find some APM solutions from the old world, but the modern APM vendors have seen these changes in the industry and have designed APM solutions to focus on your application behavior and have placed a far greater importance on the performance and availability of business transactions than on the underlying systems that support them.

Buy versus build

This article has covered a lot of ground and now you're faced with a choice: do you evaluate APM solutions and choose the one that best fits your needs or do you try to roll your own. I really think this comes down to the same questions that you need to ask yourself in any buy versus build decision: what is your core business and is it financially worth building your own solution?

If your core business is selling widgets then it probably does not make a lot of sense to build your own performance management system. If, on the other hand, your core business is building technology infrastructure and middleware for your clients then it might make sense (but see the answer to question two below). You also have to ask yourself where your expertise lies. If you are a rock star at building an eCommerce site but have not invested the years that APM vendors have in analyzing the underlying technologies to understand how to interpret performance metrics then you run the risk of leaving your domain of expertise and missing something vital.

The next question is: is it financially worth building your own solution? This depends on how complex your applications are and how downtime or performance problems affect your business. If your applications leverage a lot of different technologies (e.g. Java, .NET, PHP, web services, databases, NoSQL data stores) then it is going to be a large undertaking to develop performance management code for all of these environments. But if you have a simple servlet that calls a database then it might not be insurmountable.

Finally, ask yourself about the impact of downtime or performance issues on your business. If your company makes its livelihood by selling its products online then downtime can be disastrous. And in a modern competitive online sales world, performance issues can impact you more than you might expect. Consider how the average person completes a purchase: she typically researches the item online to choose the one she wants. She'll have a set of trusted vendors (and hopefully you're in that honored set) and she'll choose the one with the lowest price. If the site is slow then she'll just move on to the next vendor in her list, which means you just lost the sale. Additionally, customers place a lot of value on their impression of your web presence. This is a hard metric to quantify, but if your web site is slow then it may damage customer impressions of your company and hence lead to a loss in confidence and sales.

All of this is to say that if you have a complex environment and performance issues or downtime are costly to your business then you are far better off buying an APM solution that allows you to focus on your core business and not on building the infrastructure to support your core business.

Chapter 1: Getting started with APM (cont'd)

Conclusion

Application Performance Management involves measuring the performance of your applications, capturing performance metrics from the individual systems that support your applications, and then correlating them into a holistic view. The APM solution observes your application to determine normalcy and, when it detects abnormal behavior, it captures contextual information about the abnormal behavior and notifies you of the problem. Advanced implementations even allow you to react to abnormal behavior by changing your deployment, such as by adding new virtual servers to your application tier that is under stress. An APM solution is important to your business because it can help you reduce your mean time to resolution (MTTR) and lessen the impact of performance issues on your bottom line. If you have a complex application and performance or downtime issues can negatively affect your business then it is in your best interested to evaluate APM solutions and choose the best one for your applications.

This article reviewed APM and helped outline when you should adopt an APM solution. In the next article, we'll review the challenges in implementing an APM strategy and dive much deeper into the features of APM solutions so that you can better understand what it means to capture, analyze, and react to performance problems as they arise.

Chapter 2

Challenges in implementing an APM strategy

Chapter 2: Challenges in implementing an APM strategy

The last article presented an overview of Application Performance Management (APM), described high-level strategies and requirements for implementing APM, presented an overview of the evolution of APM over the past several years, and provided you with some advice about whether you should buy an APM solution or build your own. This article expands upon that foundation by presenting the challenges to effectively implementing an APM strategy. Specifically this article presents the challenges in:

- Capturing performance data from disparate systems
- Analyzing that performance data
- Automatically, or programmatically, responding to performance problems

Capturing performance data

Most applications of substance leverage a plethora of technologies. For example, you may have an application server or a web container, a SQL database, one or more NoSQL databases, a caching solution, web services running on alternate platforms, and so forth. Furthermore, we're finding that certain technologies are better at solving certain problems than others, which means that we're adding more technologies into the mix.

In order to effectively manage the performance of your environment, you need to gather performance statistics from each component with which your application interacts. We can categories these metrics into two raw buckets:

- Business Transaction Components
- Container or Engine Components

Measuring business transaction performance

The previous article emphasized the importance of measuring business transactions as an indicator of the performance of your application because business transactions identify real-user behavior. If your users are able to complete their business transactions in the expected amount of time then we can say that the performance of the application is acceptable. But if business transactions are unable to complete or are performing poorly then there is a problem that needs to be addressed.

Business Transactions can be triggered by any significant interaction with your application, whether that is a web request, a web service request, or a message that arrives on a message queue. Business Transactions are composed of various components, or segments, that run on tiers: as a request passes from one system to another, such as a by executing a web service call or executing a database query, we add the performance of that tier to the holistic business transaction.

Therefore, an APM strategy that effectively captures business transactions not only needs to measure the performance of the business transaction as a whole, but also needs to measure the performances of its constituent parts. Practically this means that you need to define a global business transaction identifier (token) for each request, find creative ways to pass that token to other services, and then access that token on the those servers to associate this segment of the business transaction with the holistic business transaction on an APM server. Fortunately most communication protocols support mechanisms for passing tokens between machines, such as using custom HTTP headers in web requests or custom JMS headers/properties in asynchronous messaging. The point is that this presents a challenge because you need to account for all of these communication pathways in your APM strategy.

Once you have captured the performance of a business transaction and its constituent tiers, the fun begins. The next section describes analysis in more depth, but assuming that you have identified a performance issue, the next step is to capture a snapshot of the performance trace of the entire business transaction, along with any other relevant contextual information. There are different strategies for capturing performance snapshots, but the most common are byte-code instrumentation (BCI) and thread polling.

Java source code is compiled into byte-code, which is similar to assembly or machine code, and then the Java Virtual Machine interprets the byte-code in real-time. Byte-code instrumentation involves modifying the byte-code of a running application, typically by hooking into the JVM's class loader, to inject performance-monitoring code. For example, we might create a new method that wraps a method call with code that captures the response time and identifies exceptions. BCI is complex and not for the weary hearted, but it is a well-understood science at this point. The big caveat to be aware of is that you need to capture performance information without negatively impacting the performance of the business transaction itself. Stated another way, don't make the problem (too much) worse!

BCI provides a real-user view of the behavior of the business transaction, but it can be a heavyweight solution that can slow down the overall performance of the business transaction. An alternative is to identify the thread that is executing the business transaction and poll for a stack trace of that thread on a regular interval, such as every 10 or 50 milliseconds. From these stack traces you can infer how long each method took to execute, at the granularity of your polling interval. Thread polling adds constant overhead to the JVM while it is running so it does not get in the way any individual business transaction execution. The key, as with BCI, is to perform thread polling sparingly and intelligently to reduce the overhead that you're adding to already overburdened machine.

Measuring container performance

In addition to capturing the performance of business transactions, you are going to want to measure the performance of the containers in which your application is running. Unfortunately the benefits that Java brings us through the notion of "write once, run everywhere" translates to challenges in container monitoring. Figure 1 attempts to illustrate this complexity.

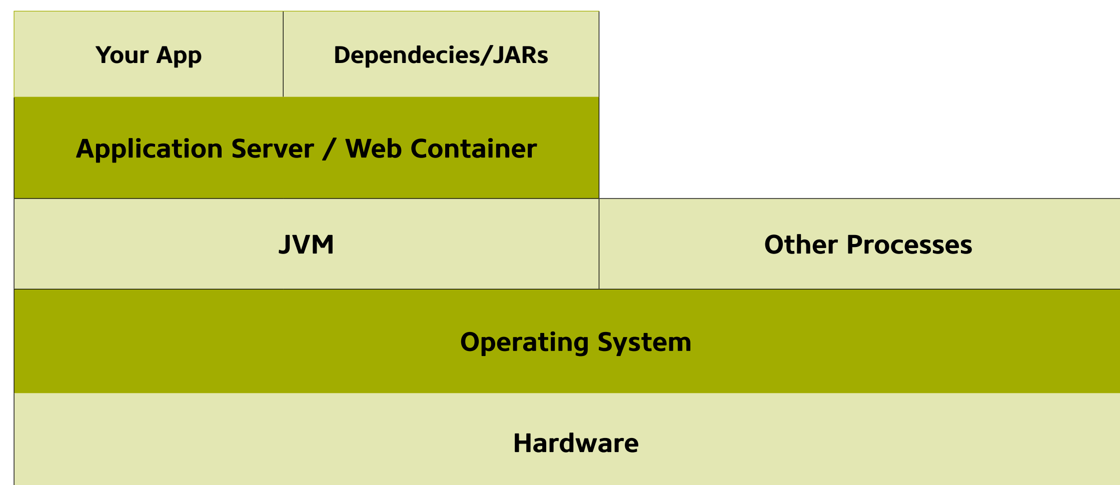


Figure 1 A JVM's Layered Architecture

An application runs in an application server or web container that runs in a JVM that runs on an operating system that runs on physical (or virtual) hardware. If there is a performance issue in the application, there is a fair amount of infrastructure that could be contributing to the problem. And in a modern virtualized or cloud-based deployment, the problem is more complex because you have introduced an additional layer of infrastructure between the JVM and the underlying hardware.

In order to effectively manage the performance of your application, you need to gather container metrics such as the following:

- Application Server / Web Container: thread pool usage, resource pool usage (e.g. connection pools), cache hit-counts and miss-counts, queued requests
- JVM: memory usage, garbage collection, JVM threads
- Operating Systems: network usage, I/O rates, system threads
- Hardware: CPU utilization, system memory, network packets

These are just a few of the relevant metrics, but you need to gather information at this level of granularity in order to assess the health of the environment in which your application is running. And as we introduce additional technologies in the technical stack, such as databases, NoSQL databases, .NET services, distributed caches, key/value stores, and so forth, they each have their own set of metrics that need to be captured and analyzed. Building readers that capture these metrics and then properly interpreting them can be challenging.

Analyzing performance data

Now that we have captured the response times of business transactions, both holistically and at the tiered level, and we have collected a hoard of container metrics, the next step is to combine these data points and derive business value from them. As you might surmise, this is a non-trivial effort.

Let's start with business transactions. We already said that we needed to generate a unique token for each business transaction and then pass that token from tier to tier, tracking the response time for each tier and associating it with that token. We need a central server to which we send these constituent "segments" that will be combined into an overall view the performance of the business transaction. Figure 2 shows this visually.

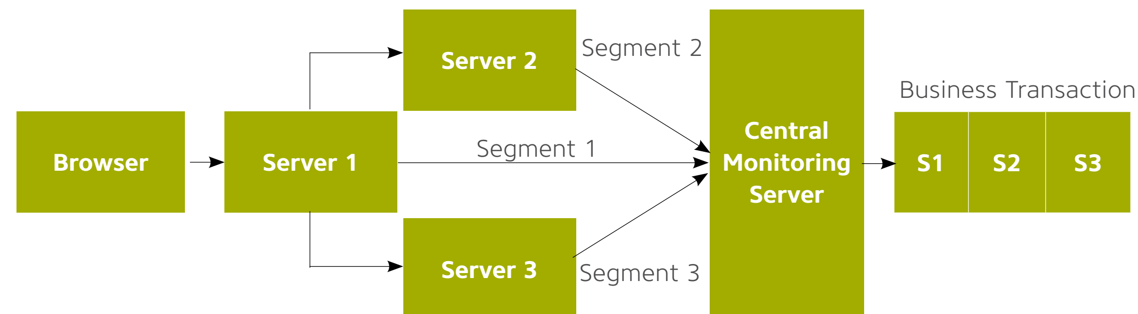


Figure 2 Assembling Segments into a Business Transaction

Analyzing the performance of a business transaction might sound easy on the surface: compare its response time to a service-level agreement (SLA) and if it is slower than the SLA then raise an alert. Unfortunately, in practice it is not that easy. In the years that I spent delivering performance tuning engagements to dozens of companies I can count on one hand the number of companies that had formally defined SLAs. In practice we want to instead determine what constitutes "normal" and identify when behavior deviates from "normal".

We need to capture the response times of individual business transactions, as a whole, as well as the response times of each of those business transactions' tiers or segments. For example, we might find that the "Search" business transaction typically responds in 3 seconds, with 2 seconds spent on the database tier and 1 second spent in a web service call. But this introduces the question of what constitutes "typical" behavior in the context of your application?

Different businesses have different usage patterns so the normal performance of a business transaction for an application at 8am on a Friday might not be normal for another application. In short, we need to identify a baseline of the performance of a business transaction and analyze its performance against that baseline. Baselines can come in the following patterns:

- The average response time for the business transaction, at the granularity of an hour, over some period of time, such as the last 30 days.
- The average response time for the business transaction, at the granularity of an hour, based on the hour of day. For example, we might compare the current response time at 8:15am with the average response time for every day from 8am-9am for the past 30 days.
- The average response time for the business transaction, at the granularity of an hour, based on the hour of the day and the day of the week. In this pattern we compare the response time of a business transaction on Monday at 8:15am with the average response time of the business transaction from 8am-9am on Mondays for the past two months. This pattern works well for applications with hourly variability, such as ecommerce sites that see increased load on the weekends and at certain hours of the day.
- The average response time for the business transaction, at the granularity of an hour, based on the hour of day and the day of the month. In this pattern we compare the response time of a business transaction on the 15th of the month at 8:15am with the average response time of the business transaction from 8am-9am on the 15th of the month for the past 6 months. This pattern works well for applications with date based variability, such as banking applications in which users deposit their checks on the 15th and 30th of each month.

In addition to analyzing the performance of business transactions, we also need to analyze the performance of the container and infrastructure in which the application runs. There are abhorrent conditions that can negatively impact all business transactions running in an individual environment. For example, if your application server runs out of threads then requests will back up, if the JVM runs a major/full garbage collection then all threads in the JVM will freeze, if the OS runs a backup process with heavy I/O then the machine will slow down, and so forth. It is important to correlate business transaction behavior with container behavior to identify false-positives: the application may be fine, but the environment in which it is running is under duress.

Finally, container metrics can be key indicators that trigger automated responses that dynamically change the environment, which we explore in the next section.

Chapter 2: Challenges in implementing an APM strategy (cont'd)

Automatically Responding to Performance Issues

Traditional applications that ran on very large monolithic machines, such as mainframes or even high-end servers, suffered from the problem that they were very static: adding new servers could be measured in days, weeks, and sometimes months. With the advent of the cloud, these static problems went away as application environments became elastic. Elasticity means that application environments can be dynamically changed at run-time: more virtual servers can be added during peak times and removed during slow times. It is important to note that elastic applications require different architectural patterns than their traditional counterparts, so it is not as simple as deploying your traditional application to a cloud-based environment, but I'll save that discussion for a future article.

One of the key benefits that elastic applications enable is the ability to automatically scale. When we detect a performance problem, we can respond in two ways:

- Raise an alert so that a human can intervene and resolve the problem
- Change the deployment environment to mitigate the problem

There are certain problems that cannot be resolved by adding more servers. For those cases we need to raise an alert so that someone can analyze performance metrics and snapshots to identify the root cause of the problem. But there are other problems that can be mitigated without human intervention. For example, if the CPU usage on the majority of the servers in a tier is over 80% then we might be able to resolve the issue by adding more servers to that tier.

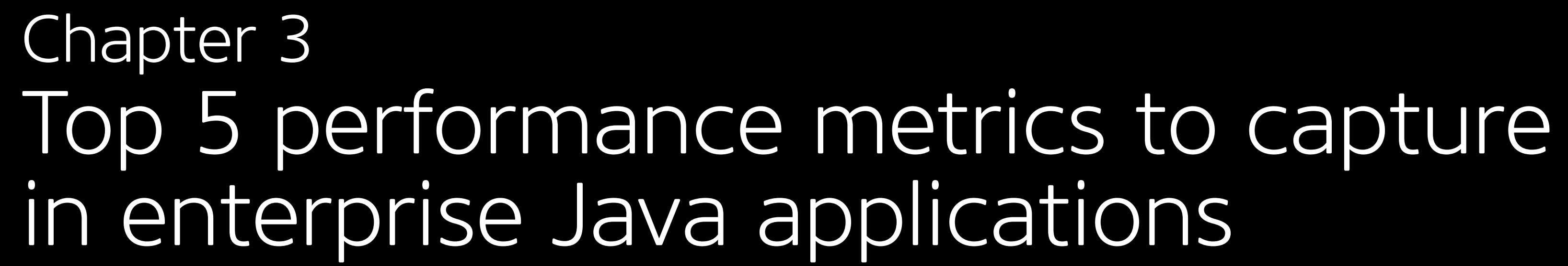
Business transaction baselines should include a count of the number of business transactions executed for that hour. If we detect that load is significantly higher than "normal" then we can define rules for how to change the environment to support the load. Furthermore, regardless of business transaction load, if we detect container-based performance issues across a tier, adding servers to that tier might be able mitigate the issue.

Smart rules that alter the topology of an application at runtime can save you money with your cloud-based hosting provider and can automatically mitigate performance issues before they affect your users.

Conclusion

This article reviewed some of the challenges in implementing an APM strategy. A proper APM strategy requires that you capture the response time of business transactions and their constituent tiers, using techniques like byte-code instrumentation and thread polling, and that you capture container metrics across your entire application ecosystem. Next, you need to correlate business transaction segments in a management server, identify the baseline that best meets your business needs, and compare current response times to your baselines. Finally, you need to determine whether you can automatically change your environment to mitigate the problem or raise an alert so that someone can analyze and resolve the problem.

In the next article we'll look at the top-5 performance metrics to measure in an enterprise Java application and how to interpret them.


Chapter 3
Top 5 performance metrics to capture
in enterprise Java applications

Chapter 3: Top 5 performance metrics to capture in enterprise Java applications

The last couple articles presented an introduction to Application Performance Management (APM) and identified the challenges in effectively implementing an APM strategy. This article builds on these topics by reviewing five of the top performance metrics to capture to assess the health of your enterprise Java application.

Specifically this article reviews the following:
- Business Transactions
- External Dependencies
- Caching Strategy
- Garbage Collection
- Application Topology

Business Transactions

Business Transactions provide insight into real-user behavior: they capture real-time performance that real users are experiencing as they interact with your application. As mentioned in the previous article, measuring the performance of a business transaction involves capturing the response time of a business transaction holistically as well as measuring the response times of its constituent tiers. These response times can then be compared with the baseline that best meets your business needs to determine normalcy.

If you were to measure only a single aspect of your application I would encourage you to measure the behavior of your business transactions. While container metrics can provide a wealth of information and can help you determine when to auto-scale your environment, your business transactions determine the performance of your application. Instead of asking for the thread pool usage in your application server you should be asking whether or not your users are able to complete their business transactions and if those business transactions are behaving normally.

As a little background, business transactions are identified by their entry-point, which is the interaction with your application that starts the business transaction. A business transaction entry-point can be defined by interactions like a web request, a web service call, or a message on a message queue. Alternatively, you may choose to define multiple entry-points for the same web request based on a URL parameter or for a service call based on the contents of its body. The point is that the business transaction needs to be related to a function that means something to your business.

Once a business transaction is identified then its performance is measured across your entire application ecosystem. The performance of each individual business transaction is evaluated against its baseline to assess normalcy. For example, we might determine that if the response time of the business transaction is slower than two standard deviations from the average response time for this baseline that it is behaving abnormally, as shown in figure 1.

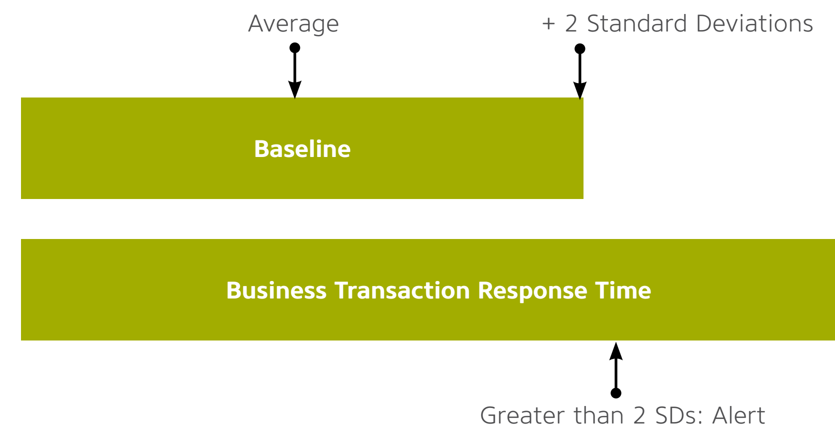


Figure 1 Evaluating BT Response Time Against its Baseline

The baseline used to evaluate the business transaction is consistent for the hour in which the business transaction is running, but the business transaction is being refined by each business transaction execution. For example, if you have chosen a baseline that compares business transactions against the average response time for the hour of day and the day of the week, after the current hour is over, all business transactions executed in that hour will be incorporated into the baseline for next week. Through this mechanism an application can evolve over time without requiring the original baseline to be thrown away and rebuilt; you can consider it as a window moving over time.

In summary, business transactions are the most reflective measurement of the user experience so they are the most important metric to capture.

External Dependencies

External dependencies can come in various forms: dependent web services, legacy systems, or databases; external dependencies are systems with which your application interacts. We do not necessarily have control over the code running inside external dependencies, but we often have control over the configuration of those external dependencies, so it is important to know when they are running well and when they are not. Furthermore, we need to be able to differentiate between problems in our application and problems in dependencies.

From a business transaction perspective, we can identify and measure external dependencies as being in their own tiers. Sometimes we need to configure the monitoring solution to identify methods that really wrap external service calls, but for common protocols, such as HTTP and JDBC, external dependencies can be automatically detected. For example, when I worked at an insurance company, we had an AS/400 and we used a proprietary protocol to communicate with it. We identified that method call as an external dependency and attributed its execution to the AS/400. But we also had web service calls that could be automatically identified for us. And similar to business transactions and their constituent application tiers, external dependency behavior should be baselined and response times evaluated against those baselines.

Business transactions provide you with the best holistic view of the performance of your application and can help you triage performance issues, but external dependencies can significantly affect your applications in unexpected ways unless you are watching them

Caching Strategy

It is always faster to serve an object from memory than it is to make a network call to retrieve the object from a system like a database; caches provide a mechanism for storing object instances locally to avoid this network round trip. But caches can present their own performance challenges if they are not properly configured. Common caching problems include:

- Loading too much data into the cache
- Not properly sizing the cache

I work with a group of people that do not appreciate Object-Relational Mapping (ORM) tools in general and Level-2 caches in particular. The consensus is that ORM tools are too liberal in determining what data to load into memory and in order to retrieve a single object, the tool needs to load a huge graph of related data into memory. Their concern with these tools is mostly unfounded when the tools are configured properly, but the problem they have identified is real. In short, they dislike loading large amounts of interrelated data into memory when the application only needs a small subset of that data.

When measuring the performance of a cache, you need to identify the number of objects loaded into the cache and then track the percentage of those objects that are being used. The key metrics to look at are the cache hit ratio and the number of objects that are being ejected from the cache. The cache hit count, or hit ratio, reports the number of object requests that are served from cache rather than requiring a network trip to retrieve the object. If the cache is huge, the hit ratio is tiny (under 10% or 20%), and you are not seeing many objects ejected from the cache then this is an indicator that you are loading too much data into the cache. In other words, your cache is large enough that it is not thrashing (see below) and contains a lot of data that is not being used.

The other aspect to consider when measuring cache performance is the cache size. Is the cache too large, as in the previous example? Is the cache too small? Or is the cache sized appropriately?

A common problem when sizing a cache is not properly anticipating user behavior and how the cache will be used. Let's consider a cache configured to host 100 objects, but that the application needs 300 objects at any given time. The first 100 calls will load the initial set of objects into the cache, but subsequent calls will fail to find the objects they are looking for. As a result, the cache will need to select an object to remove from the cache to make room for the newly requested object, such as by using a least-recently-used (LRU) algorithm. The request will need to execute a query across the network to retrieve the object and then store it in the cache. The result is that we're spending more time managing the cache rather than serving objects: in this scenario the cache is actually getting in the way rather than improving performance. To further exacerbate problems, because of the nature of Java and how it manages garbage collection, this constant adding and removing of objects from cache will actually increase the frequency of garbage collection (see below).

When you size a cache too small and the aforementioned behavior occurs, we say that the cache is thrashing and in this scenario it is almost better to have no cache than a thrashing cache. Figure 2 attempts to show this graphically

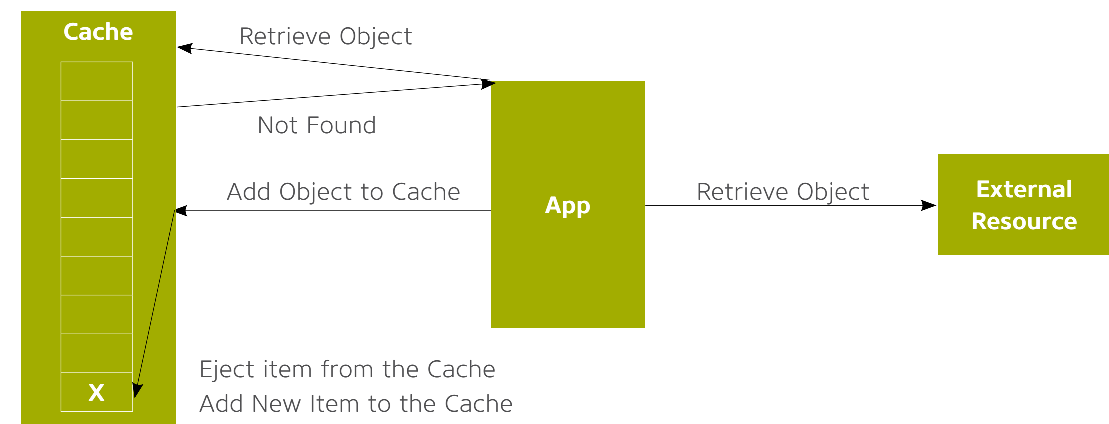


Figure 2 Cache Thrashing

Chapter 3: Top 5 performance metrics to capture in enterprise Java applications
(cont'd)

In this situation, the application requests an object from the cache, but the object is not found. It then queries the external resource across the network for the object and adds it to the cache. Finally, the cache is full so it needs to choose an object to eject from the cache to make room for the new object and then add the new object to the cache.

Garbage Collection

One of the core features that Java provided, dating back to its initial release, was garbage collection, which has been both a blessing and a curse. Garbage collection relieves us from the responsibility of manually managing memory: when we finish using an object, we simply delete the reference to that object and garbage collection will automatically free it for us. If you come from a language that requires manually memory management, like C or C++, you'll appreciate that this alleviates the headache of allocating and freeing memory. Furthermore, because the garbage collector automatically frees memory when there are no references to that memory, it eliminates traditional memory leaks that occur when memory is allocated and the reference to that memory is deleted before the memory is freed. Sounds like a panacea, doesn't it?

While garbage collection accomplished its goal of removing manual memory management and freeing us from traditional memory leaks, it did so at the cost of sometimes-cumbersome garbage collection processes. There are several garbage collection strategies, based on the JVM you are using, and it is beyond the scope of this article to dive into each one, but it suffices to say that you need to understand how your garbage collector works and the best way to configure it.

The biggest enemy of garbage collection is known as the major, or full, garbage collection. With the exception of the Azul JVM, all JVMs suffer from major garbage collections. Garbage collections come in a two general forms:

- Minor
- Major

Minor garbage collections occur relatively frequently with the goal of freeing short-lived objects. They do not freeze JVM threads as they run and they are not typically significantly impactful.

Major garbage collections, on the other hand, are sometimes referred to as "Stop The World" (STW) garbage collections because they freeze every thread in the JVM while they run. In order to illustrate how this happens, I've included a few figures from my book, Pro Java EE 5 Performance Management and Optimization.

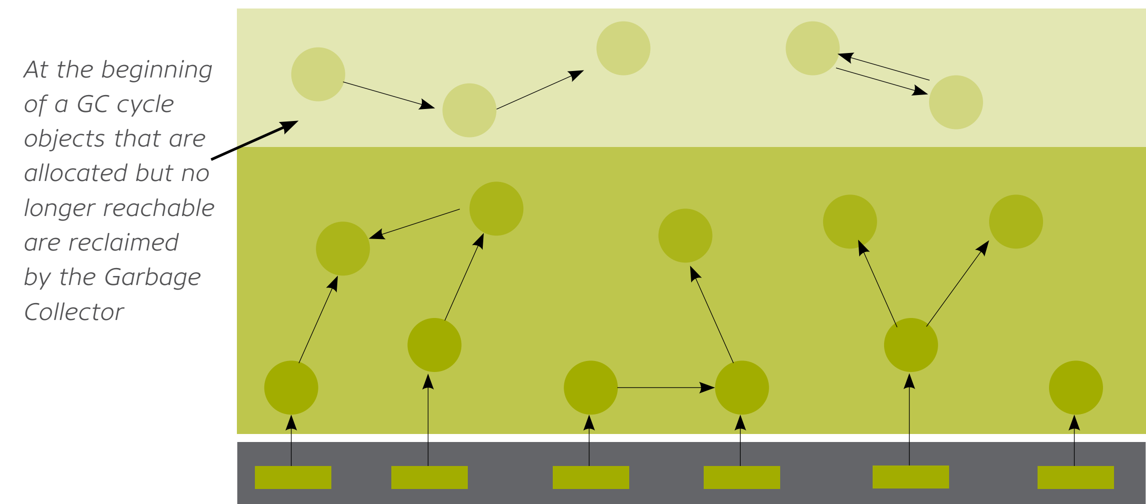


Figure 3 Reachability Test

When garbage collection runs, it performs an activity called the reachability test, shown in figure 3. It constructs a "root set" of objects that include all objects directly visible by every running thread. It then walks across each object referenced by objects in the root set, and objects referenced by those objects, and so on, until all objects have been referenced. While it is doing this it "marks" memory locations that are being used by live objects and then it "sweeps" away all memory that is not being used. Stated more appropriately, it frees all memory to which there is not an object reference path from the root set. Finally, it compacts, or defragments, the memory so that new objects can be allocated.

Chapter 3: Top 5 performance metrics to capture in enterprise Java applications
(cont'd)

Minor and major collections vary depending on your JVM, but figures 4 and 5 show how minor and major collections operate on a Sun JVM.

Figure 4 Minor Collection

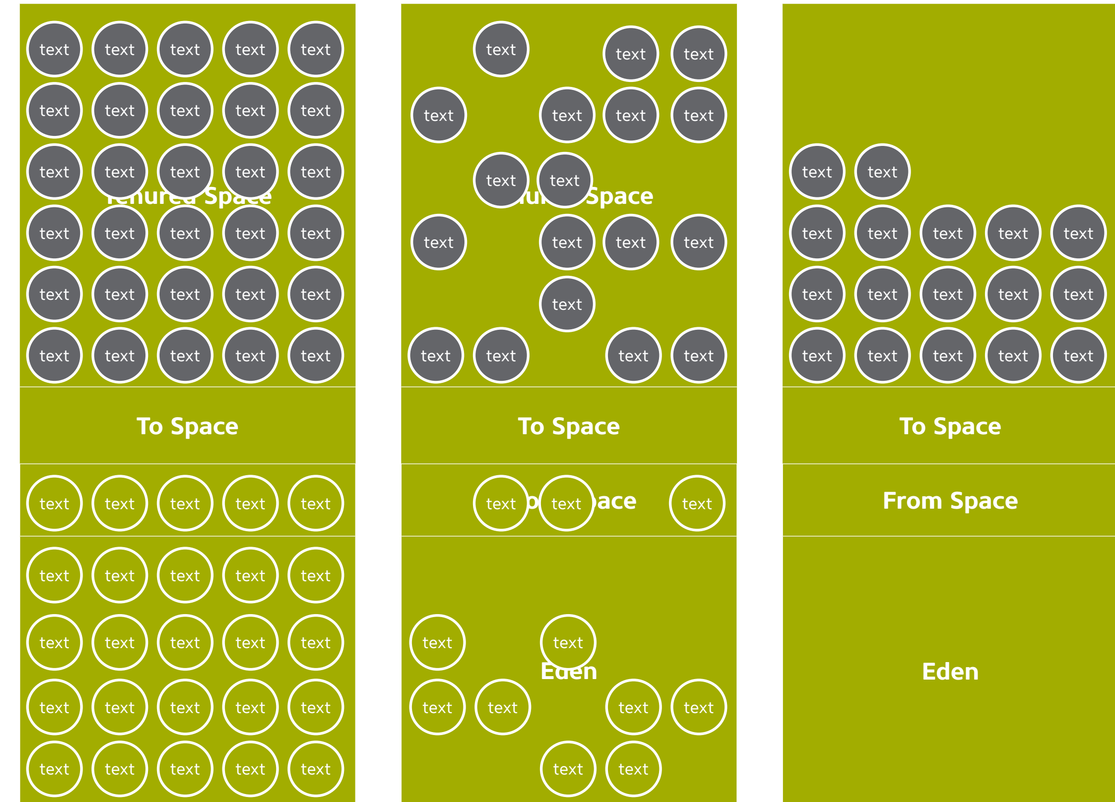


Figure 5 Major Collection

In a minor collection, memory is allocated in the Eden space until the Eden space is full. It performs a "copy" collector that copies live objects (reachability test) from Eden to one of the two survivor spaces (to space and from space). Objects left in Eden can then be swept away. If the survivor space fills up and we still have live objects then those live objects will be moved to the tenured space, where only a major collection can free them.

Eventually the tenured space will fill up and a minor collection will run, but it will not have any space in the tenured space to copy live objects that do not fit in the survivor space. When this occurs, the JVM freezes all threads in the JVM, performs the reachability test, clears out the young generation (Eden and the two survivor spaces), and compacts the tenured space. We call this a major collection.

As you might expect, the larger your heap, the less frequently major collections run, but when the do run they take much longer than smaller heaps. Therefore it is important to tune your heap size and garbage collection strategy to meet your application behavior.

Application Topology

The final performance component to measure in this top-5 list is your application topology. Because of the advent of the cloud, applications can now be elastic in nature: your application environment can grow and shrink to meet your user demand. Therefore, it is important to take an inventory of your application topology to determine whether or not your environment is sized optimally. If you have too many virtual server instances then your cloud-hosting cost is going to go up, but if you do not have enough then your business transactions are going to suffer.

It is important to measure two metrics during this assessment:

- Business Transaction Load
- Container Performance

Business transactions should be baselined and you should know at any given time the number of servers needed to satisfy your baseline. If your business transaction load increases unexpectedly, such as to more than two times the standard deviation of normal load then you may want to add additional servers to satisfy those users.

The other metric to measure is the performance of your containers. Specifically you want to determine if any tiers of servers are under duress and, if they are, you may want to add additional servers to that tier. It is important to look at the servers across a tier because an individual server may be under duress due to factors like garbage collection, but if a large percentage of servers in a tier are under duress then it may indicate that the tier cannot support the load it is receiving.

Because your application components can scale individually, it is important to analyze the performance of each application component and adjust your topology accordingly.

Conclusion

This article presented a top-5 list of metrics that you might want to measure when assessing the health of your application. In summary, those top-5 items were:

- Business Transactions
- External Dependencies
- Caching Strategy
- Garbage Collection
- Application Topology

In the next article we're going to pull all of the topics in this series together to present the approach that AppDynamics took to implementing its APM strategy. This is not a marketing article, but rather an explanation of why certain decisions and optimizations were made and how they can provide you with a powerful view of the health of a virtual or cloud-based application.

Chapter 4

AppDynamics approach to APM

Chapter 4: AppDynamics approach to APM

This article series has presented an overview of application performance management (APM), identified the challenges in implementing an APM strategy, and proposed a top-5 list of important metrics to measure to assess the health of an enterprise Java application. This article pulls all of these topics together to describe an approach to implementing an APM strategy, and specifically it describes the approach that AppDynamics chose when designing its APM solution.

Business Transaction Centric

Because of the dynamic nature of modern applications, AppDynamics chose to design its solution to be Business Transaction Centris. In other words, business transactions are at the forefront of every monitoring rule and decision. Rather than only answering questions like: what is the behavior of the execution queue thread pool on this server, it instead first answers the questions of whether not users are able to execute their business transactions and if those business transactions are behaving normally. The solution also captures container information, but really with the focus of supporting the analysis of business transactions.

Business transactions begin with entry-points, which are the triggers that start a business transaction. AppDynamics automatically identifies common entry-points, such as servlet executions, Struts Actions, Spring MVC service calls, and so forth, but also allows you to manually configure them. The goal is to identify and name all common business transactions out-of-the-box, but to provide the flexibility to meet your business needs. For example, you might define a SOAP-based web service that contains a single operation that satisfies multiple business functions. In this case, AppDynamics will automatically identify the business transaction, but will allow you to define the criteria that splits the business transaction based on your business needs, such as by fields in the payload.

Once a business transaction has been defined and named, that business transaction will be followed across all tiers that your application needs to satisfy it. This includes both synchronous calls like web service and database calls, as well as asynchronous calls like JMS messages. AppDynamics adds custom headers, properties, and other protocol-specific elements so that it can assemble all tier segments into a holistic view of the business transaction. It collects the business transaction on its management server for analysis.

Figure 1 shows how a business transaction might be assembled across multiple tiers.

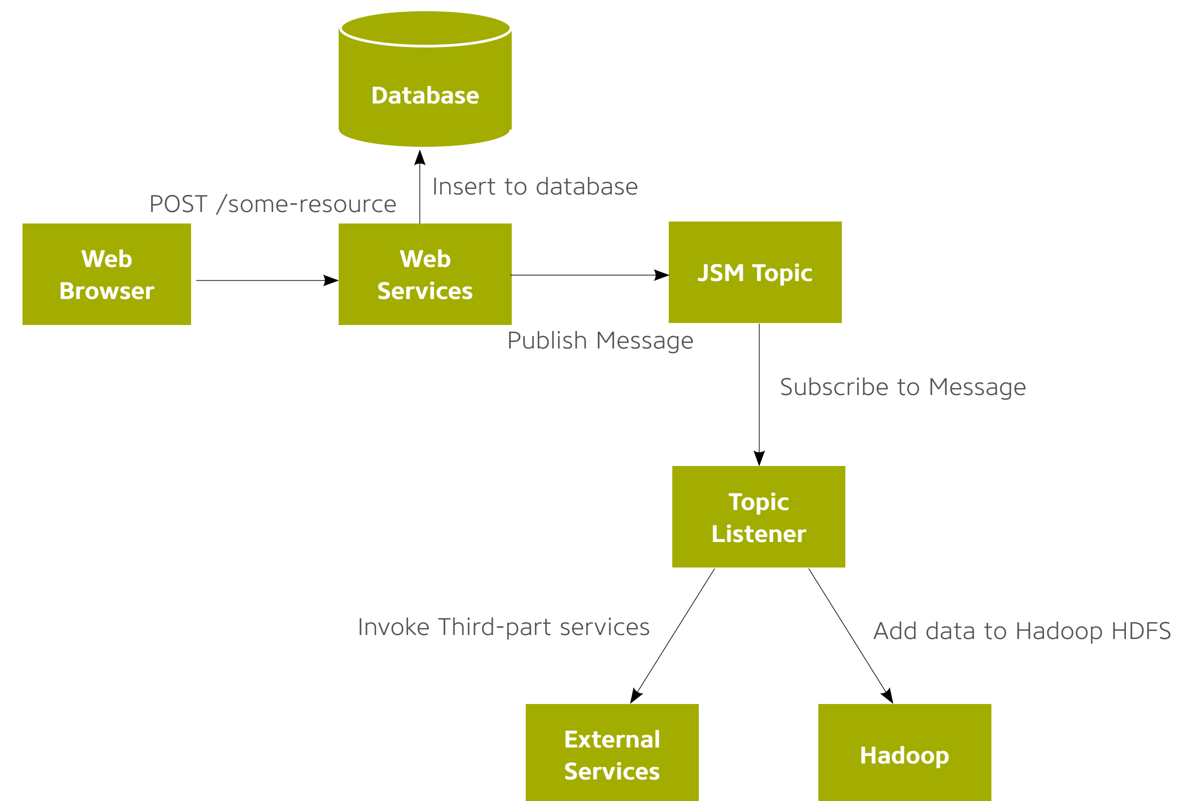


Figure 1 Defining a Business Transaction

Chapter 4: AppDynamics approach to APM (cont'd)

AppDynamics identified that defining static SLAs are difficult to manage over time, as an application evolves, so, while it allows you to define static SLAs, it primarily relies on baselines for its analysis. It captures raw business transaction data, saves it in its database, and allows you to choose the best baseline against which to analyze incoming business transactions. Baselines can be defining in the following ways:

- Average response time over a period of time
- Hour of day over a period of time
- Hour of day and day of week over a period of time
- Hour of day and day of month over a period of time

Recall from the previous articles in this series that baselines are selected based on the behavior of your application users. If your application is used consistently over time then you can choose to analyze business transactions against a rolling average over a period time. With this baseline, you might analyze the response time against the average response time for every execution of that business transaction over the past 30 days.

If your user behavior varies depending on the hour of day, such as with an internal intranet application in which users log in at 8:00am and log out at 5:00pm, then you can opt to analyze the login business transaction based on the hour of day. In this case we want to analyze a login at 8:15am against the average response time for all logins between 8:00am and 9:00am over the past 30 days.

If your user behavior varies depending on the day of the week, such as with an ecommerce application that experiences more load on Fridays and Saturdays, then you can opt to analyze business transaction performance against the hour of day and the day of the week. For example, we might want to analyze an add-to-cart business transaction executed on Friday at 5:15pm against the average add-to-cart response time on Fridays from 5:00pm-6:00pm for the past 90 days.

Finally, if your user behavior varies depending on the day of month, such as a banking or ecommerce application that experiences varying load on the 15th and 30th of the month, when people get paid, then you can opt to analyze business transactions based on the hour of day on that day of the month. For example, you might analyze a deposit business execution executed at 4:15pm on the 15th of the month against the average deposit response time from 4:00pm-5:00pm on the 15th of the past six months.

All of this is to say that once you have identified the behavior of your users, AppDynamics provides you with the flexibility to define how to interpret that data. Furthermore, because it maintains the raw data for those business transactions, you can select your baseline strategy dynamically, without needing to wait a couple months for it to recalibrate itself.

Figure 2 shows an example of how we might evaluate a business transaction against its baseline.

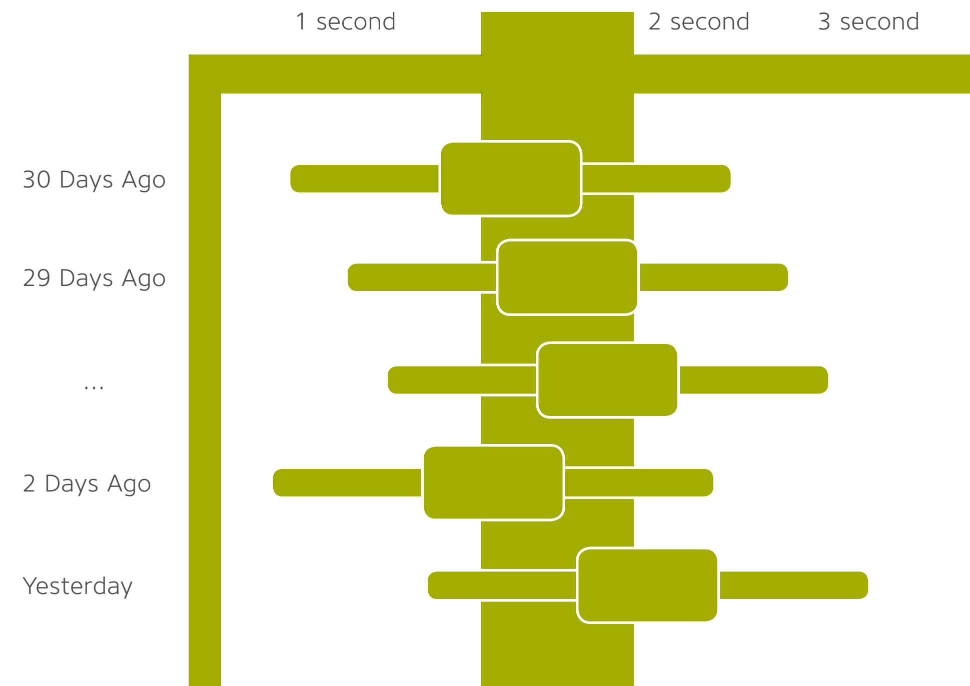


Figure 2 Sample Business Transaction Baseline Evaluation

Finally, baselines are not only captured at the holistic business transaction level, but also at the business transaction's constituent segment level. This means that although a business transaction might not wholly violate its performance criterion, but may be degrading at a specific tier, such as the database, that you will be alerted to the problem. This early warning system can help you detect and resolve problems before they impact your users.

Production Monitoring First

Defining a Business Transaction centric approach was an important first step in designing a production monitoring solution, but AppDynamics expanded its solution by realizing that an APM solution must not impact the application it is monitoring, to every extent possible. For example, if an APM solution adds an additional 20% overhead to the application it is monitoring then that valuable information is gathered at too high of a price.

AppDynamics implemented quite a few optimizations that make it well suited to be a production monitoring solution, but two of those powerful are:

- Sparing use of byte-code instrumentation
- Intelligently capture performance snapshots

Byte-code instrumentation involves injecting byte-code into your application classes at runtime, as classes are loaded into memory by the JVM class loader. In terms of performance monitoring, this instrumentation adds performance counters and tracing information that can be used to capture performance snapshots. Byte-Code Instrumentation is a powerful tool, but if it is used too liberally it can add significant overhead to your application. For example, if it is used to measure the response time of every method in a business transaction and that business transaction needs 50 methods to complete, then that is a lot of additional overhead. Most monitoring solutions mitigate this by running in a passive mode (a single Boolean check on each instrumented method tells the instrumentation code if it should be capturing response times). But even this passive monitoring comes at a slightly elevated overhead (one Boolean check for each method in the business transaction.)

AppDynamics takes a different approach: it only uses byte-code instrumentation where it needs to. It leverages byte-code instrumentation for all entry-points, which start business transactions, and for exit-points, which are the points in which a business transaction leaves the current virtual machine by making an external call. Entry-points are important to instrument because they setup and name a business transaction, exit-points are important to instrument because business transaction contextual information needs to be sent to the next tier in the business transaction, but for all other methods, byte-code instrumentation is not necessary.

Instead, AppDynamics monitors the performance of live running business transactions and, if it detects a problem, then it incorporates a thread stack trace polling strategy. Thread stack trace polling can be performed external to live business transactions: it adds additional constant overhead to the JVM, but it does not impact individual business transactions. It does come with its own cost because the amount of time spent in any individual method is inferred, which means that it is not exact and its granularity is limited to the polling interval. Out-of-the-box, AppDynamics polls at an interval of 10 milliseconds, but it allows you to tune this to balance monitoring granularity with overhead. This is a concession that AppDynamics made when defining itself as a production-monitoring tool because its goal is to capture rich information, but while keeping its overhead to a minimum. And the rationale is that if a method really takes less than 10 milliseconds to execute and you're searching for the root cause of a performance problem, do you really care about it?

This strategy leads to the next strategy: intelligently capturing performance snapshots. Some monitoring solutions capture every running transaction because they want to give you access to the one that violated its performance criterion. But as with almost every aspect of performance monitoring, this comes with a price: significant overhead for every transaction that you do not care about.

Because AppDynamics was designed from its inception to monitor production applications, they took a different approach. AppDynamics captures two types of snapshots:

- In-flight partial snapshots
- Real-time full snapshots used to diagnose a problem

First, if AppDynamics identifies that a business transaction is running slower than it should then it starts tracing from that point until the end of the transaction in an attempt to show you what was happening when it started slowing down. Depending on the nature of the problem this may or may not provide you with insight to the root cause of the problem (it may have already passed the slow portion of transaction), but many times it can identify systemic problems while keeping overhead at a minimum.

Once it has identified a slow running business transaction then it starts capturing full snapshots that can be used to diagnose the root cause of the performance problem. Rather than capturing as many transaction executions as it can, it instead attempts to capture relevant examples of the problem while reducing its overhead. Out-of-the-box it is configured to capture up-to 5 examples of the problem per minute for 5 minutes, but stopping at a maximum of 10 traces. This means that for every minute it will capture up to 5 examples of the problem: if the first 5 traces exhibit the problem then it stops tracing for the rest of the minute and picks up the next minute, but if it captures 10 traces and does not yet have 5 examples of the problem then it quits for the minute and waits for the next minute. After 5 minutes it stops and provides you with all of examples that show the problem.

This is a tradeoff between relevant data and the overhead required the capture the relevant data. The rationale, however, is that if the performance problem is systemic then 50 attempts (10 tries for 5 minutes) should be more than enough to identify the problem; if 50 attempts do not illustrate the problem then chances are that it is not systemic. Furthermore, how many examples do you need to examine to identify the root cause? If the solution captures 500 examples, are you really going to go through all of them? The assertion is that a representative sample illustrating the problem is enough information to allow you to diagnose the root cause of the problem.

Finally, if it is a systemic problem, rather than starting this 5 minute sampling every 5 minutes, AppDynamics is smart enough to give you a 30 minute breather in between the samples. This decision was made to reduce overhead on an already struggling application.

Auto-Resolving Problems though Automation

In addition to building a Business Transaction centric solution that was designed from its inception for production use, AppDynamics also observed the trend in environment virtualization and cloud computing. As a result, it has been refining its automation engine over the years to enable you to automatically change the topology of your application to meet your business needs.

It is capturing all of the performance information that you need to make your decision:

- Business Transaction performance and baselines
- Business Transaction execution counts and baselines
- Container performance

And then it adds to that a rules engine that can execute actions under the circumstances that you define. Actions can be general shell scripts that you can write to do whatever you need to, such as to start 10 new instances and update your load balancer to include the new instances in load distribution, or they can be specific prebuilt actions defined in AppDyamics repository to do things like start or stop Amazon Web Service AMI instances or add servers to a Heroku environment.

In short, the AppDynamics automation engine provides you with all of the performance data you need to determine whether or not you need to modify your application topology as well as the tools to make automatic changes.

Monitoring Your Business Application

It is hard to choose the most valuable subset of features that AppDynamics provides in its arsenal, but one additional feature that sets it apart is its ability extract business value from your application and visualize and define rules against those business values. It allows you to capture the value of specific parameters passed to methods in your execution stack. For example, if you have a method that accepts the amount charged by a credit card purchase, you can tell AppDynamics to instrument that method, capture the value passed to that parameter, and store it just as it would any performance metric. This means that you report on credit card average purchases, minimum, maximum, standard deviation, and so forth. Furthermore, it means that you setup alerts when the average credit card purchase exceeds some limit, which can alert you to potential fraud. In short, AppDynamics allows you to integrate the performance of your application with business-specific metrics, side-by-side.

Conclusion

Application Performance Management (APM) is a challenge that balances the richness of data and the ability to diagnose the root cause of performance problems with the overhead to capture that data. This article presented several of the facets that AppDynamics used when defining its APM solution:

- Business Transaction Focus
- Production First Monitoring
- Automatically Solving Problems through Automation
- Integrating Performance with Business Metrics

In the next, and final, article, we'll review tips-and-tricks that can help you develop an effective APM strategy.

Chapter 5
APM tips and tricks

Chapter 5: APM tips and tricks

This article series has covered a lot of ground: it presented an overview of application performance management (APM), it identified the challenges in implementing an APM strategy, it proposed a top-5 list of important metrics to measure to assess the health of an enterprise Java application, and it presented AppDynamics' approach to building an APM solution. In this final installment this article provides some tips-and-tricks to help you implement an optimal APM strategy. Specifically, this article addresses the following topics:

- Business Transaction Optimization
- Snapshot Tuning
- Threshold Tuning
- Tier Management
- Capturing Contextual Information
- Intelligent Re-Cycling Virtual Machines

Business Transaction Optimization

Over and over throughout this article series I have been emphasizing the importance of business transactions to your monitoring solution. To get the most out of your business transaction monitoring, however, you need to do a few things:

- Properly name your business transactions to match your business functions
- Properly identify your business transactions
- Reduce noise by excluding business transactions that you do not care about

AppDynamics will automatically identify business transactions for you and try to name them the best that it can, but depending on how your application is written, these names may or may not be reflective of the business transactions themselves. For example, you may have a business transaction identified as "POST /payment" that equates to your checkout flow. In this case, it is going to be easier for your operations staff, as well as when generating reports that you might share with executives, if business transactions names reflect their business function. So consider renaming this business transaction to "Checkout".

Next, if you have multiple business transactions that are identified by a single entry-point, take the time to break those into individual business transactions. There are several examples where this might happen, which include the following:

- Business Transactions that determine their function based on their payload
- Business Transactions that determine their function based on a query parameter
- Complex URI paths

If a single entry-point corresponds to multiple business functions then configure the business transactions based on the differentiating criteria. For example, if the body of an HTTP POST has an "operation" element that identifies the operation to perform then break the transaction based on that operation. Or if there is an "execute" servlet that accepts a "command" query parameter, then break the transaction based on the "command" parameter. Finally, URI patterns can vary from application to application, so it is important for you to choose the one that best matches your application. For example, AppDynamics automatically defines business transactions for URIs based on two segments, such as /one/two. If your application uses one segment or if it uses four segments, then you need to define your business transactions based on your naming convention.

Naming and identifying business transactions is important to ensuring that you're capturing the correct business functionality, but it is equally important to exclude as much noise as possible. Do you have any business transactions that you really do not care about? For example, is there a web game that checks high scores every couple minutes? Or is there a batch process that runs every night, takes a long time, but because it is offline you do not care? If so then exclude these transactions so that they do not add noise to your analysis.

Chapter 5: APM tips and tricks (cont'd)

Snapshot Tuning

As mentioned in the previous article, AppDynamics intelligently captures performance snapshots by both sampling thread executions at a specified interval instead of leveraging byte-code instrumentation for all snapshot elements, and by limiting the number of snapshots captured in a performance session. Because both of these values can be tuned, it can benefit you to tune them.

Out-of-the-box, AppDynamics captures stack trace samples every 10 milliseconds, which balances the granularity of data captured with the overhead required to capture that data. If you are only interested in "big" performance problems then you may not require granularity as fine as 10 milliseconds. If you were to reduce this polling interval to 50 milliseconds, you will lose granularity, but you will also reduce the performance overhead of the polling mechanism. If you are finely tuning your application then you may want 10-millisecond granularity, but if you have no intention of tuning methods that execute in under 50 milliseconds, then why do you need that level of granularity? The point is that you should analyze your requirements and tune accordingly.

Next, observe your production troubleshooting patterns and determine whether or not the number of snapshots that AppDynamics captures is appropriate for your situation. If you find that, while capturing up to 5 samples every minute for 5 minutes is resulting in 20 or more snapshots, but you only ever review 2 of those samples then do not bother capturing 20. Try configuring AppDynamics to capture up to 1 snapshot every minute for 5 minutes. And if you're only interested in systemic problems then you can turn down the maximum number of attempts to 5. This will significantly reduce that constant overhead of thread stack trace sampling, but at the cost of possibly not capturing a representative snapshot.

Threshold Tuning

AppDynamics has designed a generic monitoring solution and, as such, it defaults to alerting to business transactions that are slower than two standard deviations from normal. This works well in most circumstances, but you need to identify how volatile your application response times are to determine whether or not this is the best configuration for your business needs.

AppDynamics defines three types of thresholds against which business transactions are evaluated with their baselines:

- Standard Deviation: compares the response time of a business transaction against a number of standard deviations from its baseline
- Percentage: compares the response time of a business transaction against a percentage of difference from baseline
- Static SLAs: compares the response time of a business transaction against a static value, such as 2 seconds

If your application response times are very volatile then the default threshold of two standard deviations might result in too many false alerts. In this case you might want to increase this to more standard deviations or even switch to another strategy. If your application response times have very low volatility then you might want to decrease your thresholds to alert you to problems sooner. Furthermore, if you have services or APIs that you provide to users that have specific SLAs then you should setup a static SLA value for that business transaction. AppDynamics provides you with the flexibility of defining alerting rules generally or on individual business transactions.

You need to analyze your application behavior and configure the alerting engine accordingly.

Chapter 5: APM tips and tricks (cont'd)

Tier Management

I've described how AppDynamics captures baselines for business transactions, but it also captures baselines for business transactions across tiers. For example, if your business transaction calls a rules engine service tier then AppDynamics will capture the number of calls and the average response time for that tier as a contributor to the business transaction baseline. Therefore, you want to ensure that all of your tiers are clearly identified.

Out of the box, AppDynamics identifies tiers across common protocols, such as HTTP, JMS, JDBC, and so forth. For example, if it sees you make a database call then it assumes that there is a database and allocates the time spent in the JDBC call to the database. This is important because you don't want to think that you have a very slow "save" method in a DAO class, instead you want to know how long it takes to persist your object to the database and attribute that time to the database.

AppDynamics does a good job of identifying tiers that follow common protocols, but there are times when you're communication with a back-end system does not use a common protocol. For example, I was working at an insurance company that used an AS/400 for quoting. We leveraged a library that used a proprietary socket protocol to make a connection to the server. Obviously AppDynamics would know nothing about that socket connection and how it was being used, so the answer to our problem was to identify the method call that makes the connection to the AS/400 and identify it as a custom back-end resource. When you do this, AppDynamics treats that method call as a tier and counts the number of calls and captures the average response time of that method execution.
You might be able to use the out of the box functionality, but if you have special requirements then AppDynamics provides a mechanism that allows you to manually define your application tiers.

Capturing Contextual Information

When performance problems occur, they are sometimes limited to a specific browser or mobile device, or they may only occur based on input associated with a request. If the problem is not systemic (across all of your servers), then how do you identify the subset of requests that are causing the problem?

The answer is that you need to capture context-specific information in your snapshots so that you can look for commonalities. These might include:

- HTTP headers, such as browser type (user-agent), cookies, or referrer
- JMS properties
- HTTP query parameter values
- Method parameter values

Think about all of the pieces of information that you might need to troubleshoot and isolate a subset of poor performing business transactions. For example, if you capture the User-Agent HTTP header then you can know the browser that the user was using to execute your business transaction. If your HTTP request accepts query parameters, such as a search string, then you might want to see the value of one or more of those parameters, e.g. what was the user searching for? Additionally, if you have code-level understanding about how your application works, you might want to see the values of specific method parameters.

AppDynamics can be configured to capture contextual information and add it to snapshots, which can include all of the aforementioned types of values. The process can be summarized as follow:

1. AppDynamics observes that a business transaction is running slow
2. It triggers the capture of a session of snapshots
3. On each snapshot, it captures the contextual information that you requested and associates it with the snapshot

The result is that when you find a snapshot illustrating the problem, you can review this contextual information to see if it provides you with more diagnostic information.

The only warning is that this comes at a small price: AppDynamics uses byte-code instrumentation to capture the values of methods parameters. In other words, use this functionality where you need to, but use it sparingly.

Intelligent Re-Cycling Virtual Machines

The final recommendation that I have for you is to recycle your virtual machines intelligently. I've discussed the nature of cloud-based applications and how they are intended to be elastic: they are meant to scale up to satisfy increased user load and scale down when user load decreases to save on cost. But how do you determine what servers to decommission when you need to scale down?

The best strategy is to keep track of the servers that you have started and the order in which you started them, and then decommission servers in the reverse order. In other words, remove the servers that have been running the longest, see Figure 1.

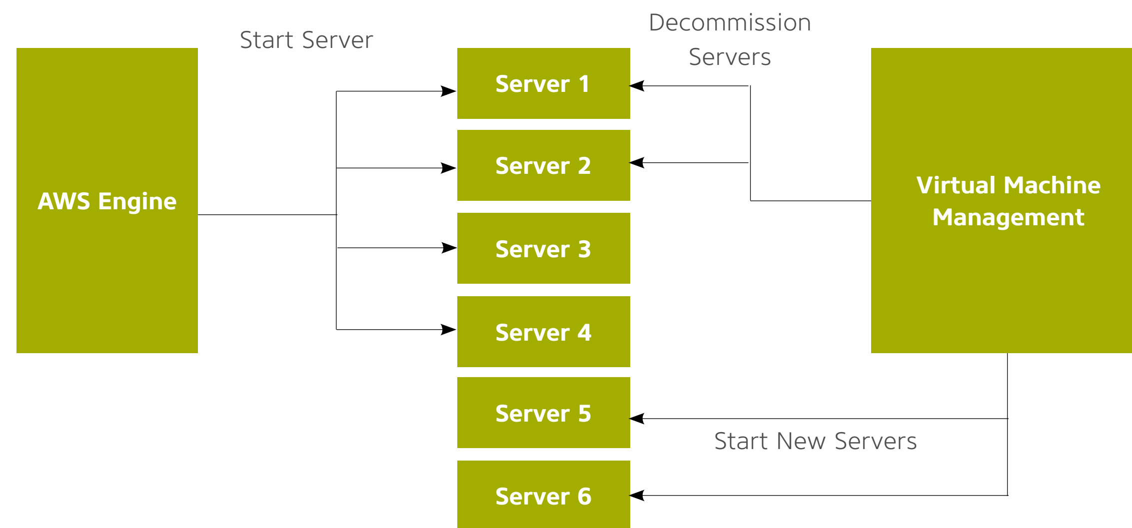


Figure 1 Recycling Virtual Machines

Why does this matter? To illustrate this, I spoke with someone responsible for the performance of one of the largest cloud-based environments in Amazon and he told me that they ran JVMs with 30 gigabyte heaps and he opted to use a parallel mark-sweep garbage collection strategy. This strategy suffers from one big problem: while it postpones major (or full) garbage collections, when they do run, they are very impactful and long running. I asked him how he handles this problem and his answer surprised me: he decommissions virtual machines before they ever have a chance to run major garbage collections! In other words, he redefined the garbage collection problem altogether, to the point where it becomes a non-issue. It is a powerful strategy that enables them to avoid garbage collection pauses altogether. But the key to his strategy is cycling virtual machines in a very prescribed and precise order.

Furthermore, even when you do not need to scale up or scale down to meet user demands, recycling servers on a regular basis can have a profound affect on the performance of your environment.

Conclusion

Application Performance Management (APM) is a challenge that balances the richness of data and the ability to diagnose the root cause of performance problems with the overhead required to capture that data. There are configuration options and tuning capabilities that you can employ to provide you with the information you need while minimizing the amount of overhead on your application. This article reviewed a few core tips and tricks that anyone implementing an APM strategy should consider. Specifically it presented recommendations about the following:

- Business Transaction Optimization
- Snapshot Tuning
- Threshold Tuning
- Tier Management
- Capturing Contextual Information
- Intelligent Re-Cycling Virtual Machines

APM is not easy, but tools like AppDynamics make it easy for you to capture the information you need while reducing the impact to your production applications.

APPDYNAMICS

appdynamics.com

© 2015 Copyright AppDynamics