APPDYNAMICS

WHITE PAPER

APM tips and tricks

In this paper you'll be provided with 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.

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.

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.

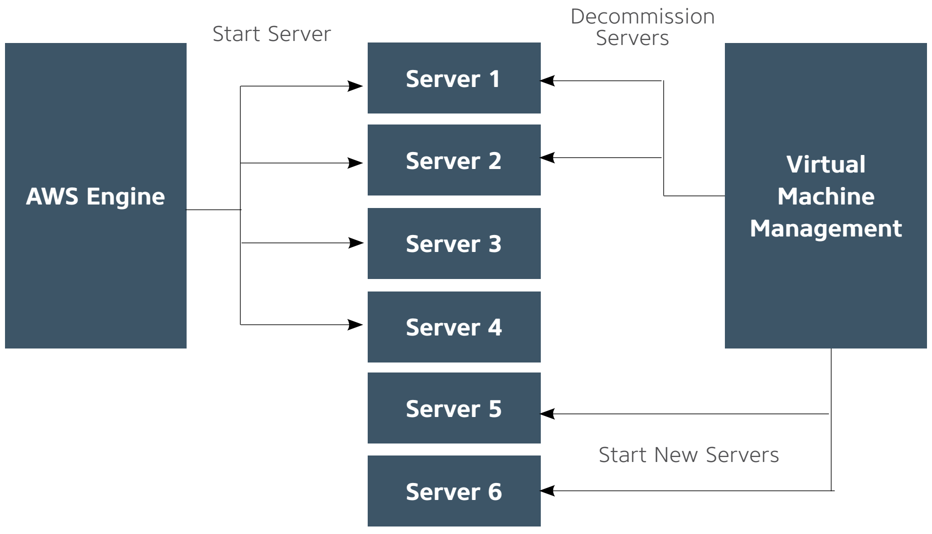

Figure 1.

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.

Try it FREE at appdynamics.com