Top 5 Java Performance Considerations

APPDYNAMICS

Contents

01

A Culture of Java Performance

Perhaps more than any programming language, Java continues to have a profound impact on how people navigate today's world. Java's functionality is responsible for setting a great deal of what users expect in terms of performance from their internet-accessible devices.

The history of Java is more than two decades long and the language continues to grow and adapt in response to evolving consumer and business expectations. Throughout all of these changes, however, the performance of Java applications remains a paramount concern for developers.

A Brief History of Java

Java was originally created in the 1990s at Sun Microsystems by James Gosling, Michael Sheridan, and Patrick Naughton, who intended to use it for next-generation smart televisions. Because the language would be used in consumer appliances, its developers had five guiding principles for Java's performance, security, and functionality. These principles declared that Java would be:

- Secure and robust
- High performance
- Portable and architecture-neutral, able to run on any software or hardware
- Threaded, interpreted, and dynamic
- Object-oriented

Although using Java (then known as Oak) for interactive television failed to pan out, the language was repurposed for use with the World Wide Web. In 1995, Sun released Java 1.0, promising that programmers could "Write Once, Run Anywhere" by developing code on any device and running it on any device with a Java Virtual Machine.

Due to Java's many strengths, the language has not only survived but thrived up to the present day. Over the course of its two-decade history, Java has been used for an incredible variety of purposes, from embedded systems and web applets to desktop and mobile applications.

Java Today

With an estimated 9 million developers worldwide, the Java community is a very robust one. One of the most important uses of Java today is building applications for mobile devices using the Android operating system. In addition, Java remains incredibly popular for enterprise applications and client-server web applications, and is also a very common language of instruction in software development courses.

According to application security company Veracode, Java's popularity among web applications has slightly declined over the past few years. In 2011, Veracode's customers used Java to write 52% of their web applications, a figure which has since decreased to 43% in 2016 as .NET applications have grown in popularity.

However, Java has still maintained its dominant position in the world of software development thanks to advantages such as its flexibility, portability, and ease of use. According to the April 2017 TIOBE index, Java remains by far the most popular programming language for developers around the world.

Why Is Java Performance Important?

Software developers and architects love to solve problems — after all, it's part of the job description. However, it's not always true that the most elegant, efficient, or obvious solution is also the best-performing solution.

Car analogies are rampant in the world of software development, but here's another one. Imagine that you're an automobile engineer in charge of building a racecar. Would you start by building a family-friendly sedan and then make changes to the engine and the chassis, or would you build a racecar from the very beginning?

Of course, you'd choose to build it from scratch, because turning a sedan into a racecar would involve much more effort than building a racecar in the first place. The same philosophy should apply to your Java applications. If you don't design them for performance from the outset, then you'll spend a lot more time and effort upgrading their performance once they're built.

Why Is APM Important?

Application performance management (APM) is the monitoring and management of your applications in order to learn how well they perform. Of course, this is a very broad definition that can apply to many different activities with various kinds of software and hardware.

To be a little more specific, APM can monitor — and correlate — factors such as:

- The physical hardware atop which your application runs
- The virtual machines in which your application runs
- The JVM hosting the application environment
- The container in which your application runs
- The application's behavior
- The supporting infrastructure, including networks, databases, and caches

Once you've captured performance metrics from these sources, you need to interpret them and determine what impact they'll have on your business. APM experts are able to understand what these performance metrics mean for an individual system, and whether they indicate abnormal behavior for your application.

In addition, depending on your application and deployment environment, your APM solution may be able to take corrective action automatically when it detects an issue. For example, if you have an elastic application running in a cloud environment, you can have your APM solution add additional services to your infrastructure when your application is experiencing high demand.

APM is important because it helps you determine when your applications are behaving abnormally so you can identify the root cause of the problem. APM solutions can resolve your application's performance issues more quickly and efficiently than other options, such as manual instrumentation or even having your end users inform you about problems.

Business Transactions

Business Transactions are how users directly interact with and experience your business. This kind of interaction can take many forms, from online purchases, watching a video, tracking an order, or transferring funds. Think of Business Transactions as discrete tasks or objectives that people want to achieve when they use your applications.

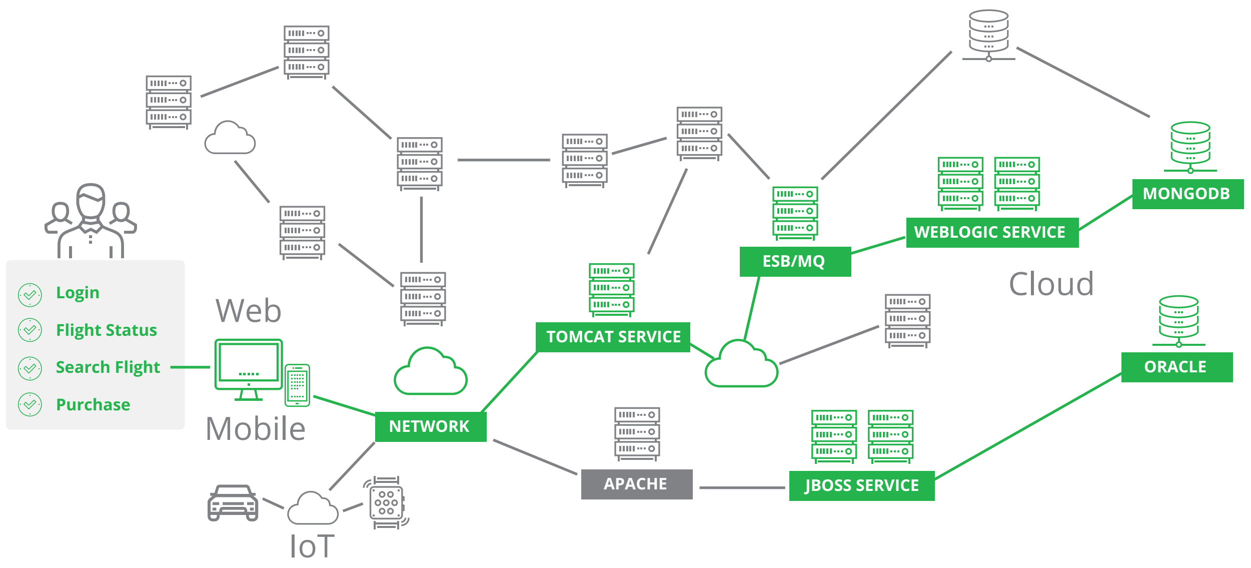
A well-defined Business Transaction is how non-technical users think of your software. That means a non-technical user won't say, "CPU utilization is at 98 percent," but would definitely say something like, "When I try to check the weather the application is slow." By monitoring Business Transactions, we're also monitoring the functions of the software as the users see them, which helps ensure that we're monitoring the important stuff.

Of course, there's a lot of abstraction built into your web applications. This means that there's a lot of separation between any given Business Transaction (e.g., checking the weather) and the software functions and components that go into executing that transaction.

Being able to break a Business Transaction down into these constituent parts is immensely valuable. By doing so, you can troubleshoot poor performance from your application and identify code-level issues in order to isolate the root causes of your performance issues.

As a software developer, Business Transactions should be one of the core fundamental metrics that you pay attention to. In order to properly assess your application's performance, you need to keep track of how well your users can execute the Business Transactions they set out to accomplish.

Capturing metrics about your Business Transactions helps you understand how your users experience your application's behavior. In order to extract the most valuable information, analyze what the minimum, maximum, and average response times are for your Business Transactions, as well as the standard deviation in order to assess the impact of any outliers.



A Search Flight business transaction

Organizational Inertia

Unfortunately, Java's incredible persistence as a programming language has meant that it has become ossified within many organizations. Companies that have been using Java for a long time have likely seen a great deal of success with it. Of course, this makes them reluctant to adapt their software development practices and workflow. After all, why change what's working for them?

Many companies see the cost of upgrading their applications as too high, causing them to rely on older, inefficient APIs and solutions. Some of the most common outdated technologies in Java applications are:

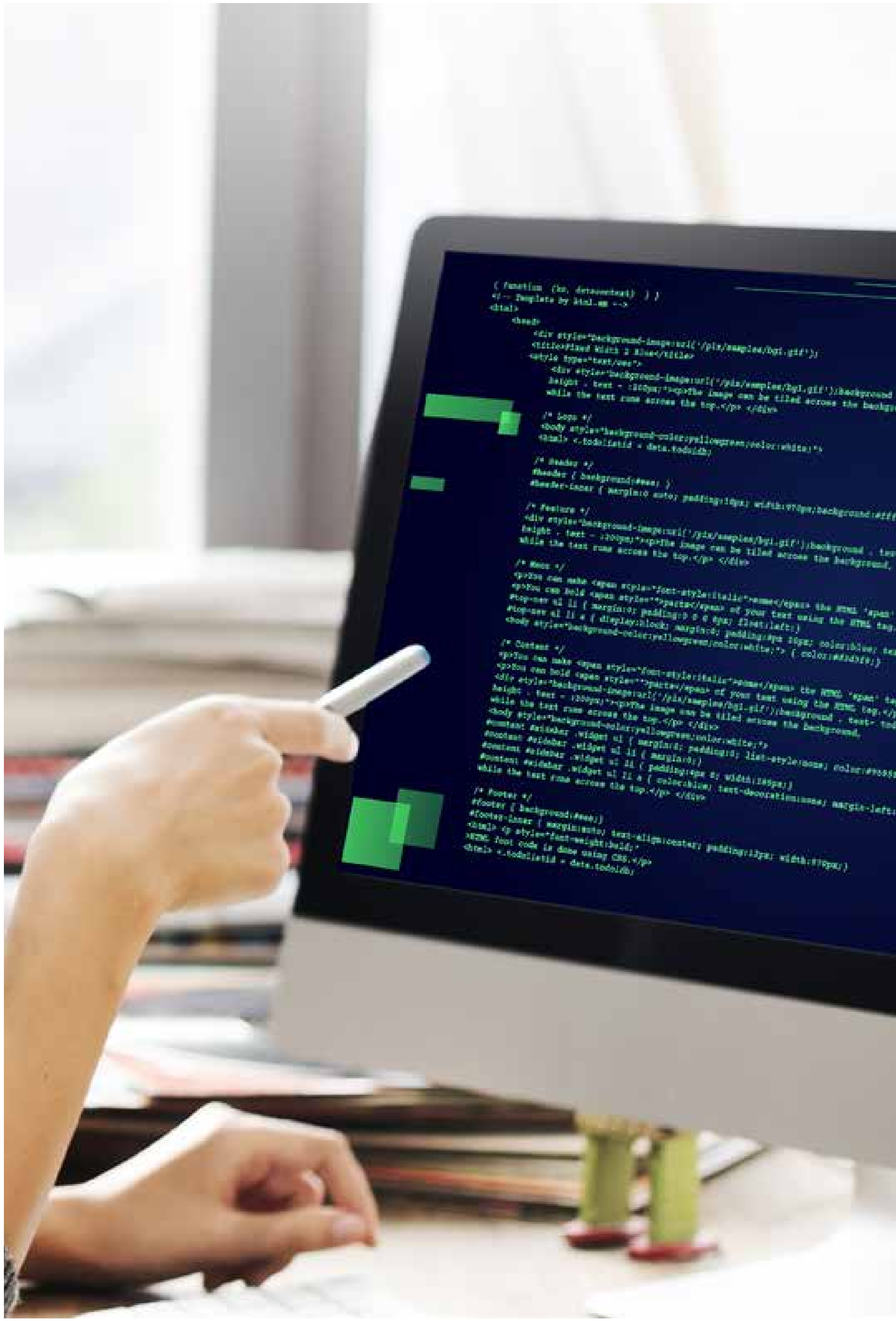
- **Synchronous HTTP:** Current best practices are to almost always use asynchronous HTTP requests. Synchronous HTTP requests are disfavored because they block the client until the operation is complete, wasting valuable computation time.
- **Java Messaging System (JMS):** Developers now prefer to work with modern, high-performance messaging solutions such as Kafka or ZeroMQ.
- **Java Database Connectivity (JDBC):** JDBC is oriented for use with relational database management systems (RDBMS). However, many developers currently favor NoSQL for their web applications.
- **J2EE containers:** Nowadays, developers tend to use modern asynchronous HTTP engines.

This unwillingness to stay current is dangerous and potentially fatal for organizations. Using outdated enterprise applications may feel like slipping into a comfortable pair of shoes, but it puts you at risk for unexpected behavior and events. For one, older technology — especially incredibly popular technology such as Java — is more vulnerable to security risks and cyber attacks that can bring your organization to a standstill. Naturally, there's also the simple fact that older technology makes you less competitive against your rivals in the constantly changing business landscape. It becomes harder to hire technical people who understand how the older software works together. Meanwhile, it's easier to hire talent willing and able to bring applications up to date with modern technologies.

To avoid this fate for your organization, it's your obligation to keep abreast of the most crucial and common Java performance issues. By doing so, you'll be able to minimize their impact and even prevent them from happening in your application.

02

Using the Express Lane: Code Optimizations



In terms of performance, Java has come a long way since the 1990s. Over the years, the language has acquired a reputation of being slow — partly deserved and partly not. At its inception, Java was an interpreted language, not compiled, which made it sluggish to execute. What's more, after just-in-time (JIT) compilation was introduced, it took some time to refine, growing more and more efficient with each version.

Today, well-designed Java applications perform extremely well in production environments. In many instances, Java code is able to match or even outperform code in "fast" languages such as C/C++. Java isn't the ideal choice for every type of application, and many Java applications take a long time to start up, which may contribute to the lingering reputation for slowness.

Today, the responsibility for slow Java applications lies solely with the developers. Unfortunately, too many Java developers are either ignorant of the code issues that are harming their application's performance, or they don't care enough to fix them.

This negligent attitude is a shame, because many problems with Java code boil down to a few core issues that are fairly straightforward to understand. Here are some of the most egregious errors you might be committing when writing Java code — and the ways you can avoid them in your own applications.

Remote Calls

Java Remote Method Invocation (RMI) is a Java API that, as its name implies, allows objects in a distributed system to access the data and invoke the methods of other non-local objects. Essentially, Java RMI makes it possible for applications running on one Java Virtual Machine (JVM) to call the methods of remote Java objects on other JVMs.

More specifically, Java RMI works by using "stubs" and "skeletons", objects created to ensure reliable communication during the method call. The calling object delegates a method request to the stub, which converts the caller's arguments into a byte stream representation and passes them to the remote skeleton. The skeleton "reinflates" this byte stream back into the original arguments and invokes the desired method of the called object. Once the skeleton receives a return value, it converts this response into a byte stream and returns it to the stub.

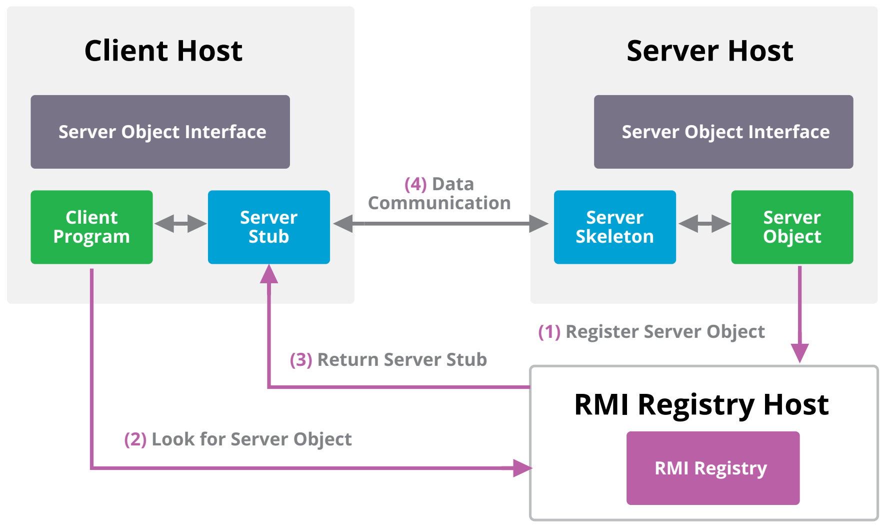
Why It's Bad

Unfortunately, as you might have guessed from the description above, remote calls are resource-intensive. For starters, the application needs to create the stub and skeleton objects. You'll also need to spend additional time and effort to convert and revert the method arguments, send them over the network, and wait for a response.

Although there are still valid reasons to use Java RMI, using remote calls inefficiently or designing your remote interfaces poorly can put a serious ding in your application's performance. For example, you may be using web services specified at a level that's too granular, requiring dozens of calls in order to complete the functionality for a single request.

What to Do

To lessen the drawbacks of using Java RMI, try to make as few round trips as possible by making it easy to retrieve multiple items within a single remote invocation. Keep your method arguments as simple as you can so that you don't have to convert and revert complex objects and pass them over the network.



Java RMI Client/Server Model
Source: Oracle.com

Exceptions as Control Flow

Using exceptions to control the flow of your program is one of the most well-known "anti-patterns" in Java development, and yet programmers still do it all the time. So, why is it so tempting, and why is it so bad for your application?

In Java, exceptions are objects created when an error or other unusual event happens during program execution, disrupting the normal flow of the program. The exception is given or "thrown" to the Java runtime system, which is responsible for figuring out what to do with it and what to do next.

Like their name implies, exceptions are intended for exceptional circumstances — times when your application screeches to a halt — and to help figure out what's gone wrong. They're not to be used to save a few lines of code or craft an elegant function, and they're not what compilers expect to find on a regular basis.

Why It's Bad

Exceptions are akin to GOTO statements, which transfer control to another line of code elsewhere in the program. Of course, GOTO statements have been falling out of favor ever since Edsger Dijkstra's renowned 1968 letter "Go To Statement Considered Harmful." GOTO statements violate standard principles about what your code should look like — and tend to make your program more confusing for other developers in the process.

Furthermore, using exceptions for control flow is generally frowned upon because exceptions are expensive. Creating a Java exception is a very slow operation. After all, exceptions are supposed to be rare events, so compilers have less reason to optimize their performance.

This sluggishness is generally because it takes a long time to fill in the exception thread stack via the Throwable.fillInStackTrace() function. This function fills in the execution stack trace, providing information about the state of the stack frames at the time the exception occurred, and places it inside the new Throwable object.

What to Do

Despite these performance costs, you shouldn't be dissuaded from using exceptions in your Java application. Just make sure that you save them for truly exceptional events that require the intervention of the Java runtime system.

Data Transformations

One of the convenient things about Java from a development perspective is the ease with which you can convert data between different representations, such as XML or JSON. This transformation is known as "serialization" when converting an object or data structure into a storable format, or "marshalling" when this data will be used in a remote call. Appropriately enough, the opposite process, inflating a representation of an object into the executable object itself, is called "deserialization" or "unmarshalling".

Why It's Bad

Java's in-house implementation of serialization leaves a lot to be desired. For one, Java's serialization algorithm relies on reflection to discover information about the object instance that it's serializing, which tends to be slow. In addition, the serialization algorithm is recursive, meaning that any objects reachable from the current object also must be serialized.

What to Do

If you're struggling with slow performance from your serialized objects, know that there's an alternative — the Externalizable interface. Unlike the Serializable interface, Externalizable requires you to implement the serialization and deserialization methods yourself. This requires more work upfront, but pays off later with control over these algorithms, eliminating some of the inefficiencies of the default serialization algorithm.

When done correctly, switching your objects to implement the Externalizable interface instead can deliver significant performance improvements for your application.

Excessive Logging

Logging messages are incredibly useful for your work as a software developer — until you realize that they're hurting your performance. At first glance, it's hard to see how this could be so. After all, more information about your application could never be harmful, and logging as much data as possible could be helpful for bug fixing and troubleshooting in the future.

Why It's Bad

Logging usually means your code base will be longer by at least one line for every event logged. If you're logging everything that you possibly can, your code will be cluttered up, obscuring the actual functions your application is carrying out.

Excessive logging doesn't just clog up your code, it also creates crowded log files. Remember that the logging messages you're creating are ultimately for human consumption, and the longer these log files are, the more draining it will be to analyze them.

However, the real drawback of excessive logging is that it will have a measurable impact on your application, slowing it down to unacceptable levels. String construction and concatenation in Java are not negligible operations, and when performed thousands or millions of times in succession, they will use up a great deal of resources. In addition, excessive logging requires your application to perform disk I/O operations more frequently, which can create problems with CPU wait time.

What to Do

Ultimately, only you can say whether the amount of logging in your application is truly excessive or if you actually need it to do your work. If you're struggling to figure out how much logging you need to do, you're probably doing too much. Finding that sweet spot between logging and performance will require some degree of experimentation.

If you're using a logging utility such as Log4j in your application, make full use of the different levels: DEBUG, INFO, WARN, ERROR, and FATAL, in increasing order of severity. Correctly placing your logging events within this hierarchy will make sure that you see the right levels of severity exactly when you want to see them. Also, avoid creating strings or any other objects by guarding the logging calls inside of an "if" block checking whether the appropriate log level is active.

03

Building Foundations: Application Infrastructure

Your Java applications are much more than the code that they've been created with. Modern non-trivial web applications require support from a number of infrastructural elements — including databases, servers, and networks — in order to function properly.

But, introducing this external infrastructure also introduces a variety of new concerns and performance issues that you'll have to address. Here's a look at some of the biggest problems that Java developers face when dealing with their application's infrastructure.

Database

The backbone of any modern web application is its data. Since their introduction in the mid-1990s, web applications have gone from being static programs with little user interaction to incredibly complex beasts that perform a variety of functions.

Of course, "With great power comes great responsibility." The database is both the most essential part of many Java applications and the greatest source of performance issues. Problems might crop up in many places. Your application code may access the database inefficiently, the database connection pool may be improperly sized, or the database itself may be missing indices or otherwise in need of tuning.

The N+1 Problem

One of the biggest database problems for Java applications is known as "Death by a thousand cuts," also called the "N+1 problem". Suppose that you want to retrieve the last 100 entries from your "Order" table. To begin with, you'll need to execute a query to find the primary keys of each item:

```
SELECT id FROM Order WHERE …
```

Next, you'll need to execute one query for each record:

```
SELECT * FROM Order WHERE id = ?
```

Doing this requires one query for each record, plus the initial query to find the primary keys. In other words, it takes 101 queries to retrieve 100 records, hence the term "N+1 problem". Fortunately, fixing issues of this sort is usually fairly straightforward — simply increase your database's capacity.

Database Metrics

The N+1 problem is just one example of a Java database issue that reveals the importance of understanding your program's performance and behavior. It's essential to manage the overall performance of your application and measure the performance of your database metrics. A few important database metrics are described below:

- **EXPLAIN PLAN:** SQL statements that display how the Oracle optimizer plans to perform an operation.
- **Wait States:** Delays when a processor needs to access external memory or devices.
- **Disk I/O:** The number of read-and-write operations to and from disk, which may cause delays if excessive.
- **Network I/O:** The number of read-and-write operations over your network, which also may cause delays if excessive.
- **CPU:** The percentage of database call time that is spent on the CPU. Although there is no "correct" figure for this metric, significant changes may indicate changes in how the application or database operates.

SQL

Using SQL in your application means that you need to be mindful of a number of software and hardware performance issues. First, make sure that your queries are well-designed and efficient. Poor query design is one of the biggest causes of degrading database performance. Some of the issues surrounding SQL query design are:

- Selecting more data than necessary, often with the use of the SELECT * statement.
- Inefficient join operations between tables with large amounts of data.
- Too few or too many indexes, causing slow performance.
- Too many literals in your SQL statements and functions, creating parse-related contention.

Another issue that can strike SQL databases is capacity, which can come in many forms:

- **CPUs:** You may not have enough CPUs, or the speed of your current CPUs may be too slow.
- **Disks:** Your disks may be full, misconfigured, or too slow, without a sufficient number of input/output operations per second (IOPS).
- **Memory:** You may have insufficient memory for the operations that you want to execute.

Other issues can be caused by the configuration of the SQL Server, such as a buffer cache that is too small or the failure to cache previous queries.

Cache

When you use it correctly, caching can provide your application with huge savings in time and memory. Done properly, the majority of your requests should be hitting the cache, rather than requiring a trip to your database.

If your application isn't using the cache properly, however, the pressure on your database will increase as the pressure on your application increases. This means that your database will require more CPU power and may need to read from and write to the disk more, which degrades the performance of applications that interact with the database.

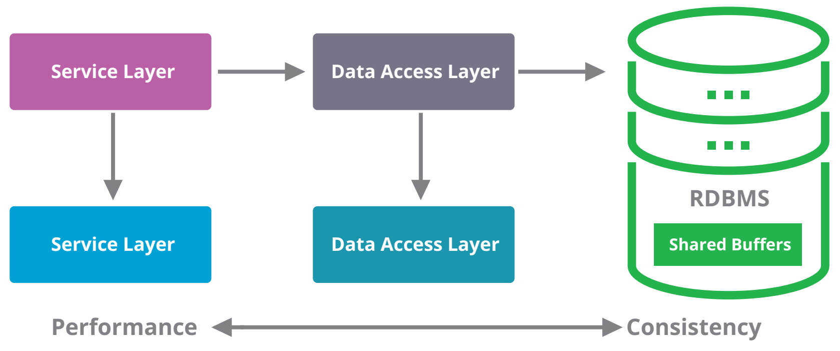
No Caching

The first mistake that some developers make is not using a cache at all. Of course, serving content from memory is much faster than having to execute a query in order to retrieve your data from a database. Unless you have specific reasons to do so in your application, not caching is a critical error in judgment.

Cache Configuration

Another performance issue occurs when you fail to configure your cache properly. Caches hold stateful objects, which represent specific object instances that are distinct from other objects of the same type. Because caches are stateful, you need to configure them to a finite size so that you don't run out of memory.

When the cache is full, it will take action based on how you've configured it. For example, it might decide to remove the objects that have been accessed the least, or those that have not been accessed for the longest period of time. If your application searches for an object in the cache that's no longer present, then it results in a miss that usually requires a database lookup. Monitor your cache's hit ratio — the percentage of object retrievals from the cache that were successful — in order to determine if you need to adjust your caching strategy.



Anatomy of Cache Layers

Distributed Caching

One final issue with caching comes when you have multiple servers each writing to their own caches. Without configuring these caches to be distributed, they won't stay in sync with each other, which means that your results may vary depending on which server your application uses.

Most modern caches support a distributed paradigm where updates to one cache propagate to other members in the cache. However, this can be an expensive operation, depending on the number and size of the caches and the data consistency that your application requires. You may be able to tolerate caches that are "eventually consistent" after a short period of time where they differ. However, it may be just as true that you need all of the caches to be consistent before any update can be considered complete.

Whatever the case, figure out what your business requires from your application, and adopt the loosest distributed caching strategy that matches those requirements.

Server

If you're running your own servers locally, it's essential to know your machine metrics in order to accurately judge their performance. Some of the most important server metrics are:

- **Response Times:** The amount of time that your application requires to generate a response to a request. You should pay attention to both average response times over a given monitoring period as well as peak response times, which measure the longest gap between request and response.
- **CPU Utilization:** The amount of CPU time that your application takes to service a request. Percentages near 100 percent usually indicate a problem with the application or a need for additional computing power.
- **Memory Utilization:** The amount of memory that your application uses while servicing a request. This metric is usually calculated as a percentage by determining the ratio of your application's resident set size to the total physical memory of the server.
- **Error Rate:** The percentage of server requests that result in an HTTP status code indicating an error.
- **Uptime:** The time that a server has been available and working. Both the absolute amount of time and the percentage of actual uptime are important metrics. If your server has been available less than roughly 99% of the time, you should investigate possible causes and fixes.

If you're running in the cloud with a provider such as Microsoft Azure or Amazon Web Services EC2, it will still help to understand the behavior of the machine, even if it's virtual.

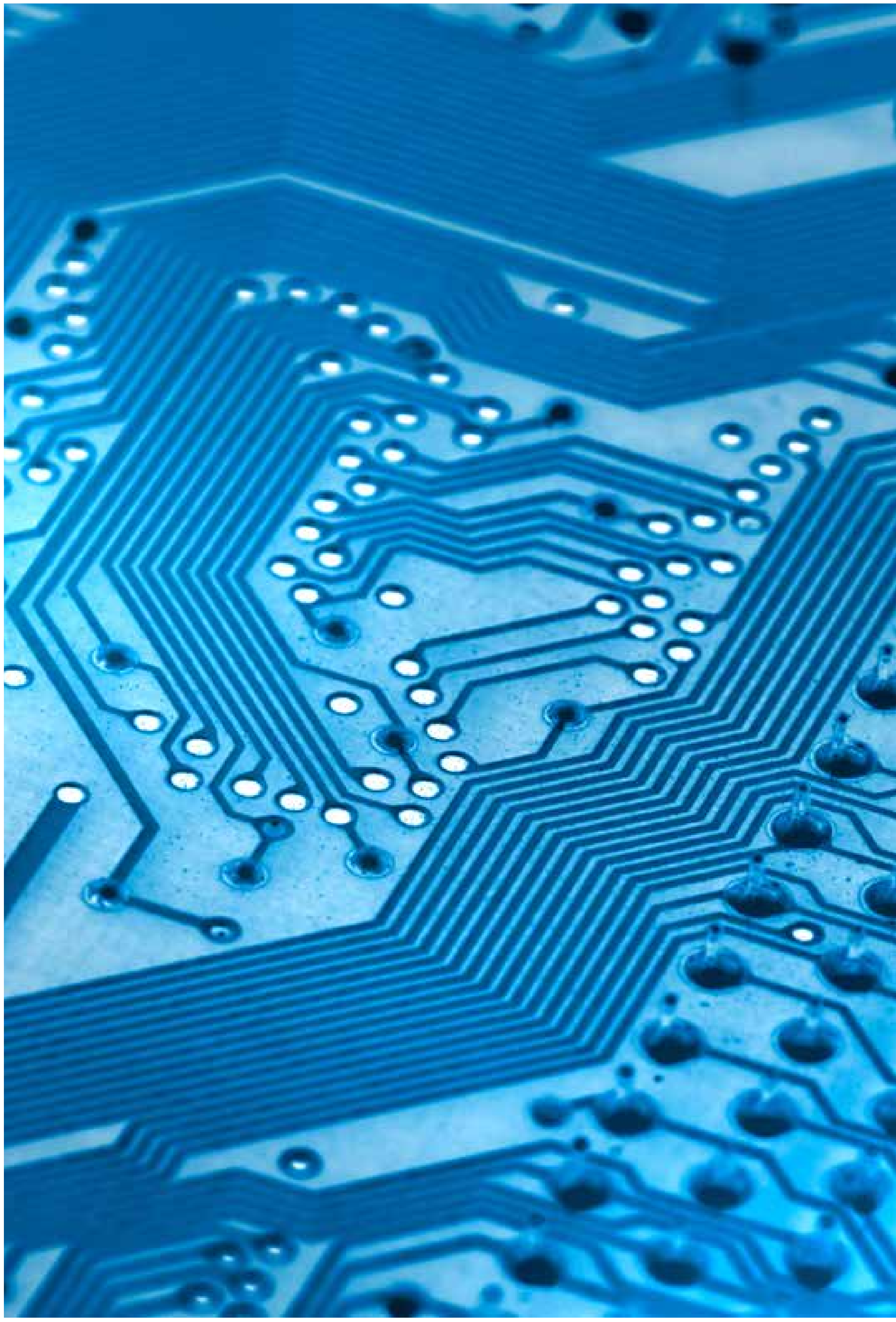
Network

Network visibility is currently a black box for many APM solutions. At AppSphere 2016, we showed how our customers can now shine a light into the black box of network performance and begin measuring throughput of all their systems.

The AppDynamics network visibility dashboard doesn't rely on switched port analyzer (SPAN) or test access point (TAP) methods. Instead, we monitor traffic in and out of the network by assembling five data points into a tuple: the source and destination IP addresses, the source and destination ports, and the protocol. Once you toggle the network dashboard, you'll see an overlay of the underlying network traffic flow data, as well as key performance indicators such as bandwidth, packet loss, and throughput.

04

Software Nostalgia: Memory Management

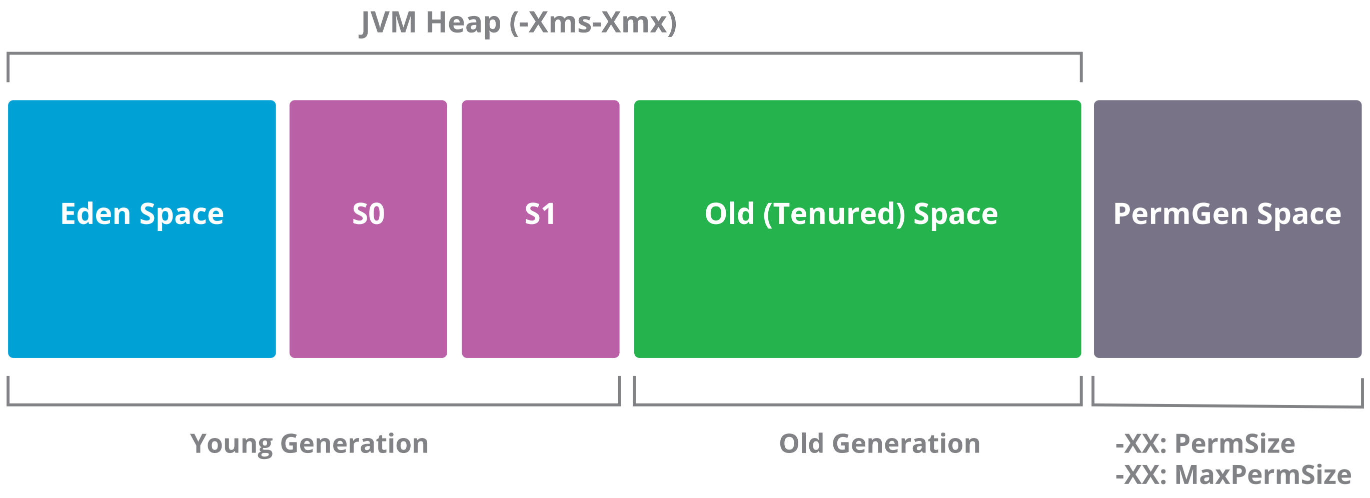


How Memory Management Works in Java

In Java, memory management is the responsibility of the Java Virtual Machine (JVM). The JVM's memory-related duties include allocating memory for the application from the operating system, managing the application's heap and stack, and removing unneeded objects ("garbage collecting").

Unlike in C/C++, there is no explicit allocation or deallocation of memory in Java. Instead, when a new object is created in the course of running an application, the JVM allocates a contiguous block of memory on the heap in order to store it. Objects that are no longer referenced are considered to be "garbage" and are cleared from the heap by the JVM's garbage collector, a program that reclaims unused memory for use at a later time.

The specifics of memory management and garbage collection vary depending on the JVM implementation. For example, the Sun JVM uses a "generational" heap that is divided into two primary components: the "young generation" and the "old generation". The young generation is further subdivided into three spaces: Eden space, where objects are created, and two Survivor spaces. When the Eden space is full, objects that survive garbage collection are moved to one of the Survivor spaces. After a major garbage collection event, objects within the young generation are moved to the old generation for preservation.



Java Garbage Collection

Primary Symptoms of Java Memory Leaks

The garbage collector in Java helps remove many of the mundane concerns surrounding memory management, and memory issues with Java are much less of a concern than with C/C++. However, despite all the hype around Java's automatic memory management, it's still very possible to experience memory leaks in a Java application.

Java memory leaks occur when your application continues to maintain a reference to an object that is no longer necessary. Thus, the object remains in memory far past its "expiration date", taking up space and putting pressure on the application's performance. Because the garbage collector has no insight into the developers' thought process, it will assume that you want to use an object reference that continues to be maintained, even if you have no intention of doing so.

In Java, memory leaks can only occur inside containers that support unbounded growth. The most common example of these are the Collection classes: lists, maps, and sets that can be used to build data structures such as arrays, trees, hash tables, and linked lists. In many cases, memory leaks occur when the application adds an object to a Collection but does not remove it from the Collection when it's no longer needed.

If you're coming from a C/C++ background, the symptoms of a memory leak in Java will be all too familiar to you. When a memory leak occurs, the amount of memory that the application uses continually increases until the heap eventually runs out of memory. Often, this means that your application will work well at first, but slow down over time or as you work with larger datasets. In addition, your application might crash unexpectedly.

When a memory leak causes your application's heap to run out of memory, Java will often throw an OutOfMemoryError, which usually requires you to restart the JVM. However, it's important to note that not all OutOfMemoryErrors are due to a memory leak, and not all memory leaks will cause an OutOfMemoryError. If memory leaks are a disease affecting your application, then OutOfMemoryErrors are a symptom that isn't present in all patients.

The Benefits of Memory Monitoring

In order to monitor the memory that your application is using, you need to track the memory inside of the JVM, not the memory used by the JVM process. This means that you'll have to use a specialized tool for looking inside the JVM process, rather than a system monitor such as Task Manager for Windows, or Activity Monitor for Mac.

Garbage collection in Java isn't a constant activity, but is run only after certain conditions are met. This means that the memory you see the JVM using at any given point in time is likely more than your application requires, unless you measure it immediately after the garbage collector has run.

Therefore, in order to get an accurate picture of your application's memory usage, you need to measure the average amount of memory used over an extended period of time. Noticing that your application has recently started using a little more memory isn't cause for concern, but if its memory usage consistently increases while running, then there may be a memory leak somewhere in your code.

If your application is consistently running out of memory and you suspect that it contains a memory leak, you can test your hypothesis by simply allocating more memory to the application. Examine the time your application takes to crash after providing this additional memory. If the time between crashes has now increased, there's likely a memory leak somewhere in the application.

The Garbage Collector

The JVM garbage collector works by performing a "mark-and-sweep" test to determine which objects are reachable. In this test, the garbage collector first traverses the heap, marking those objects that it was able to reach during its journey. Next, all objects that were not marked during the traversal are swept away by the garbage collector, reclaiming this unused memory.

In certain cases, garbage collection can itself cause performance issues for your application. As described above, the Sun JVM divides the heap into two spaces, the young generation and the old generation. When a major garbage collection event occurs, objects in the young generation that are still reachable are transferred to and compacted within the old generation. These major events are also known as "stop-the-world" (STW) collections.

STW garbage collections are extremely effective for your application, but they also cause serious performance issues. This is because they live up to their name — when STW collection occurs, all of the threads in the JVM are frozen while the garbage collector is carrying out its business. In other words, your application has to stop and wait while the JVM does some interior redecorating. The length of the pause depends directly on the size of your heap. If your heap is only a few gigabytes, you might wait 3 to 5 seconds, but a heap 10 times that size could require half a minute or more.

There are a number of possible solutions to minimize the impact of STW collections on your application's performance. However, none of them will completely remove the need for STW collection, since it's a normal process that only presents a problem when it takes too long or occurs too often.

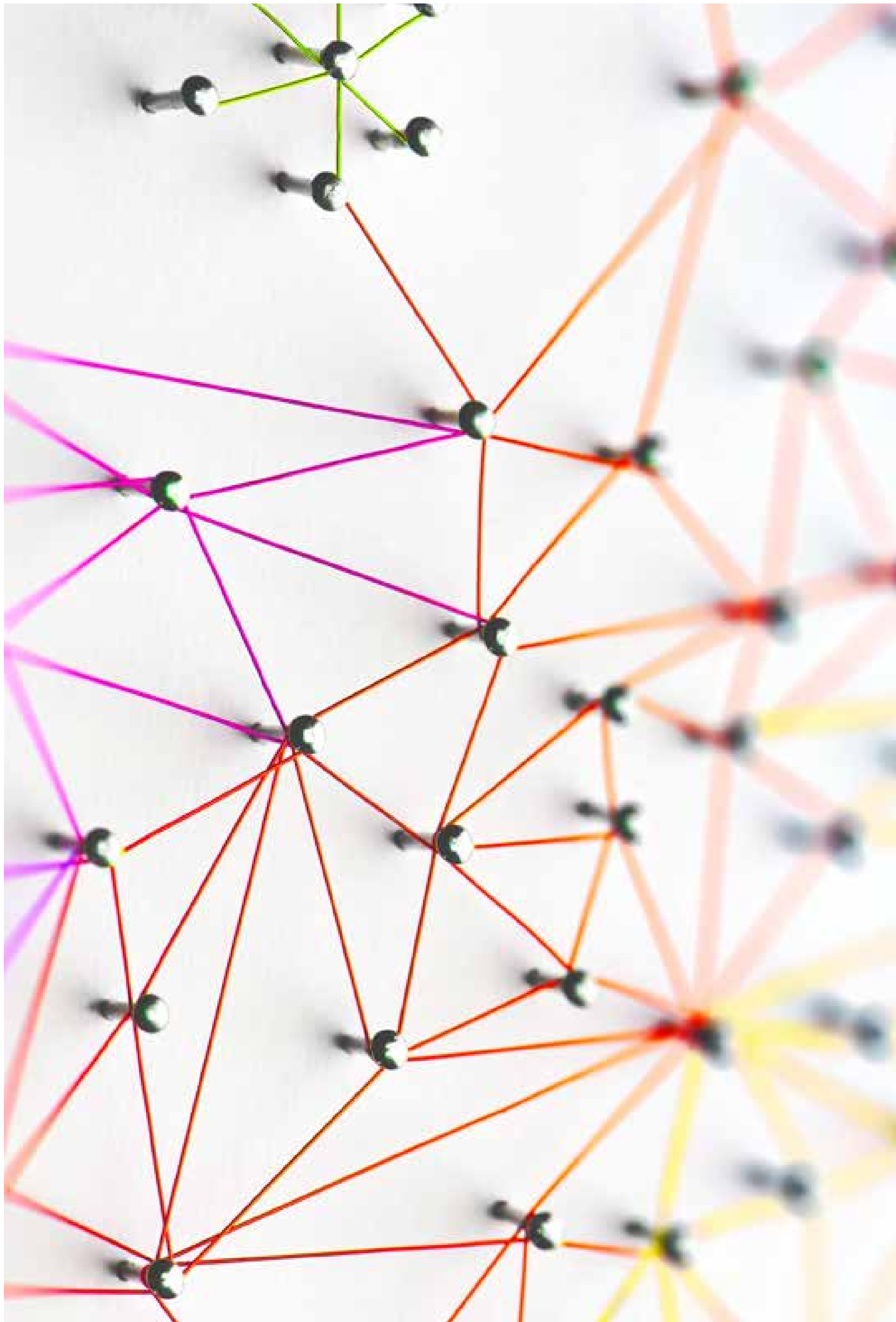
To begin with, you can try fine-tuning the performance of your garbage collector using command-line options. The JVM provides you with a number of options related to garbage collection that you can set at runtime. For example, you can adjust the initial and maximum sizes of the heap, as well as the comparative sizes of the young and old generations and the Eden and Survivor spaces. By playing around with these options, you can see which ones result in the best performance.

Java also allows you to set the type of garbage collector that you use in your application. The JVM actually includes four different garbage collectors — serial, parallel (the default), concurrent-mark-sweep (CMS), and garbage first (G1). These last two modes are the most interesting for experimentation.

The CMS garbage collector uses an additional thread to constantly mark and sweep objects, in exchange for using more CPU resources. This extra thread helps to reduce the number of times that your application needs to perform a STW collection. Meanwhile, the G1 garbage collector, which was introduced in Java Development Kit (JDK) 7, is intended for heaps that will grow larger than 4 gigabytes. The G1 collector uses multiple threads to scan through different regions of the heap, focusing on those regions that contain the most garbage objects.

05

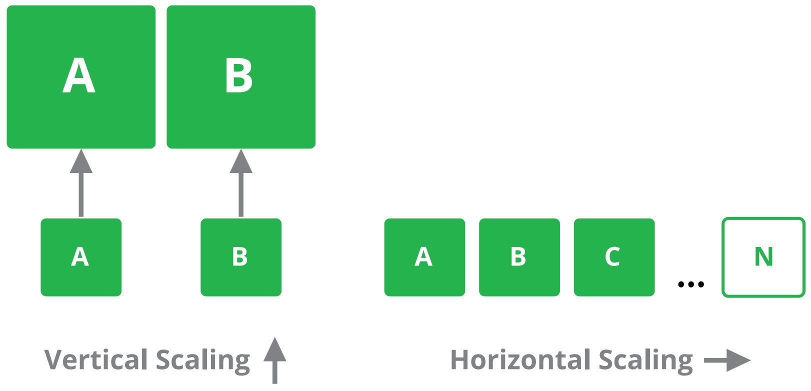
Embracing Your Growth: Scale



You can scale an application in one of two ways — horizontally or vertically. Horizontal scaling involves adding more machines or nodes to your current system, usually to reduce the burden on the existing nodes. Vertical scaling, on the other hand, involves adding resources to an existing machine, such as memory or processing power. Horizontal scaling has no physical limits, whereas vertical scaling does, since you can't add infinite memory or processors. In addition, you can even scale your application down during times of decreased demand in order to conserve resources.

With the advent of cloud computing, scalability has become easier than ever, as the resources available for your application become near-infinite. These days, scaling your application is less a question of whether you can do so, and more a question of how you can do so efficiently.

Many applications rely on concurrency in order to achieve scalability. But what exactly is concurrency, and how does it help you use your available hardware to the fullest extent?



Vertical versus Horizontal Scaling

An Introduction to Concurrency

Users expect that modern enterprise applications will be able to do all sorts of things at once, from performing complex mathematical calculations to communicating with distant servers. In other words, enterprise applications need to be concurrent. In the context of software development, an application that is concurrent is one that's able to execute several streams of computations at the same time.

Concurrent applications tend to be more complex than sequential programs because each stream of operations can interact and interfere with other streams. Objects that can be shared between these different streams — known as threads — should not change while several computations are ongoing. Otherwise, multiple threads might each make changes to an object, unaware of the other threads' behaviors, and this will cause an unexpected (and likely incorrect) result.

The Java language deals with thread concurrency using a technique called synchronization. Each object in Java has a lock that can only be given to one thread at a time. If a thread wants to have an object's lock but finds that it's unavailable, then it must wait for the other thread to release the lock before executing its own synchronized code.

As great as synchronization is, of course, it also introduces added complexity into your Java application — and the possibility of additional performance issues.

Deadlocks

Imagine that you're in kindergarten and you want to do some finger painting with your best friend. You both go to grab the supplies. You get the paints from your backpack while your friend finds some paper. Now both of you have one material that you need to do the painting, but not the other, and neither one of you wants to give up what you have.

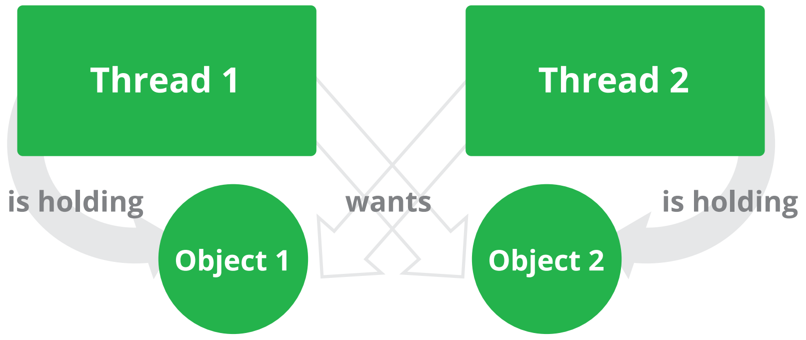
Of course, in real life the situation would be resolved somehow. This kind of scenario is exactly what happens when multiple threads need access to objects that the other thread is currently using. However, unlike real life, there's no easy way to resolve the situation in your application — the two threads are deadlocked.

Diagnosis

Deadlocks are very difficult to reproduce in a development environment because they're the result of race conditions, when simultaneous operations are not executed in the order that you expected they would be. However, if you believe that your application is suffering from deadlock, watch your metrics to see if the CPU remains consistently underutilized while the application does less and less. To confirm your suspicions, you can request a thread dump that will show you reports of deadlocked threads.

Avoiding Deadlocks

In order to avoid deadlocks, make your application and resources as immutable as you can, and try to avoid using too much thread synchronization. As we'll see in the next section, that creates its own issues.

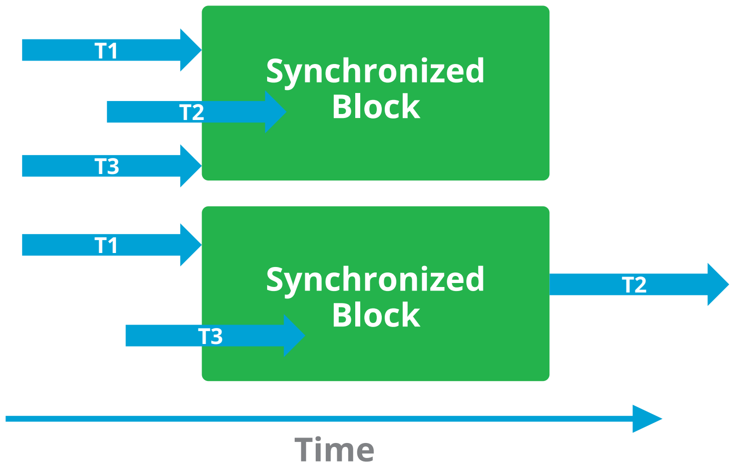


Thread Deadlock

Gridlocks

If you've ever driven on a highway, you've almost certainly been the victim of a traffic jam that seems to come out of nowhere. After waiting for an excruciating amount of minutes or hours, you finally get past the bottleneck, only to see that it was caused by a car accident or a new construction project.

In these cases, restricting cars to one or two lanes is necessary because of the unexpected block on the highway. Similarly, it's sometimes necessary to use synchronization in your application in order to avoid unexpected behavior. If your application is over-synchronized, however, then you're essentially merging all of the "lanes" on your highway for no good reason. Thread synchronization is a powerful tool, but it should be used with caution to ensure that you don't unintentionally cripple your application.



Gridlock as result of thread over-synchronization

Diagnosis

One major symptom of over-synchronization is that your application will have slow response times and low CPU utilization, because only one thread will be active at a time. In order to diagnose over-synchronization in your application, however, you'll have to figure out where your code is waiting and why each thread has been stalled. This means that you may need to use thread dumps or examine your Business Transactions from a method-level view. With AppDynamics, you can simply use the new thread contention feature to mitigate.

Avoiding Gridlocks

To remove gridlocks and deadlocks from your application, the same advice is generally applicable — use synchronization only when absolutely necessary. Try to use non-blocking constructs such as Java's built-in collections from the java.util.concurrent package.

You should be especially clear about when to use a StringBuilder and when to use a StringBuffer. Unlike a StringBuilder, the StringBuffer class is thread-safe, and all of its public methods are synchronized. However, StringBuilders perform better than StringBuffers under most circumstances, and you should generally prefer them.

Issues such as gridlocks and deadlocks are usually themselves a symptom of a greater infrastructural problem. You should architect your application in a manner that will avoid thread synchronization issues in the first place.

Worker Sizing Problems

Whether it’s in line at the supermarket or on hold on the phone, everyone has had the unpleasant experience of waiting too long to get customer service. Fortunately, this sort of incident taught you the exact frustrations that your users have when there are gridlocks in your application’s thread pool.

Applications that run in an application server or web container will have a thread pool that controls how many requests they can handle at once. Arriving requests are placed in a queue. When a thread from the pool is available to do work, this worker thread claims the first request in the queue, processes it, and returns to the pool to claim another job.

Getting the size of your application’s thread pool right can make you feel like Goldilocks hunting for the right bed to sleep in — it can’t be too big or too small. If you don’t have enough threads in your thread pool, then your requests will linger a long time before being serviced. On the other hand, if you have too many threads in the pool, then they’ll all execute at the same time, soaking up all of your computing resources. It’s even possible that some of your threads will “starve” because they have to wait indefinitely for CPU time while other threads use it up.

Diagnosis

Thread pools that are too small are relatively easy to diagnose by examining your metrics. If this is the case for your application, you should see a thread pool utilization of nearly 100 % while the CPU is being underutilized. This is a fairly clear sign that your machine is able to process additional requests.

Conversely, if your thread pool is too large, you’ll see the opposite results — a very high rate of CPU utilization, but a thread pool that has too many idle threads and few requests waiting for service.

Avoiding Worker Sizing Problems

The ideal size of your thread pool is highly dependent on your application and your business needs. Of course, the best way to figure this out is configuration through experimentation. Your thread pool should be large enough to take full advantage of your processing resources, while remaining small enough to avoid trying to do too much at the same time.

Configuring the size of your thread pool grows more complicated if your application needs to access external resources such as a database. In this case, your thread pool also needs to be the right size to use these resources effectively without flooding them with data.

Figuring out the right size of your thread pool for working with external resources is still as much of an art as it is a science. However, you can use a performance tuning strategy called “wait-based tuning” for a little assistance. First, determine the maximum capacity of your external resources, and then regulate your application so that it will send an appropriate volume of requests without exceeding this capacity. From there, you can configure your application backwards to determine the optimum size of the thread pool.

Conclusion

Performance is something that needs to be baked into your code from the very beginning. If you only start measuring performance once all the functionality has been completed, you'll likely spend extra hours or days fixing the issues that you uncover. Even worse, if you don't have true visibility into application performance, you'll have trouble finding the root causes of your performance and scalability issues, which are often architectural in nature.

As your little Java web application gets bigger and bigger, you'll likely have to adjust it to make it more scalable. Perhaps you've hit it big and now have 10 times the number of users, or the size of your database is growing by leaps and bounds. Nonetheless, your technical debt will catch up so augmenting your software development life cycle (SDLC) now with an APM solution will help you reach maximum velocity and peak performance moving forward.

Whatever your situation, your application needs to expand in order to satisfy your business needs and understanding the most critical performance metrics will help you reach rapid scale.

Todd Rader has been in the software field for over 20 years doing applications, system software, and for the last 10 years, APM. He has held positions in development, leadership, and sales engineering, and has been in the trenches as APM has moved up the stack from monitoring low-level metrics of a three-tier app to monitoring the health of a software-defined business, in companies as diverse as Sun Microsystems, Wily Technology, CA, VMware, and currently, AppDynamics. As a Java developer in a startup implementing a multi-tenant SaaS offering during the dot-com boom, Todd was sold on the promise of DevOps before it even had a name.

START A FREE TRIAL AT APPDYNAMICS.COM

APPDYNAMICS

appdynamics.com