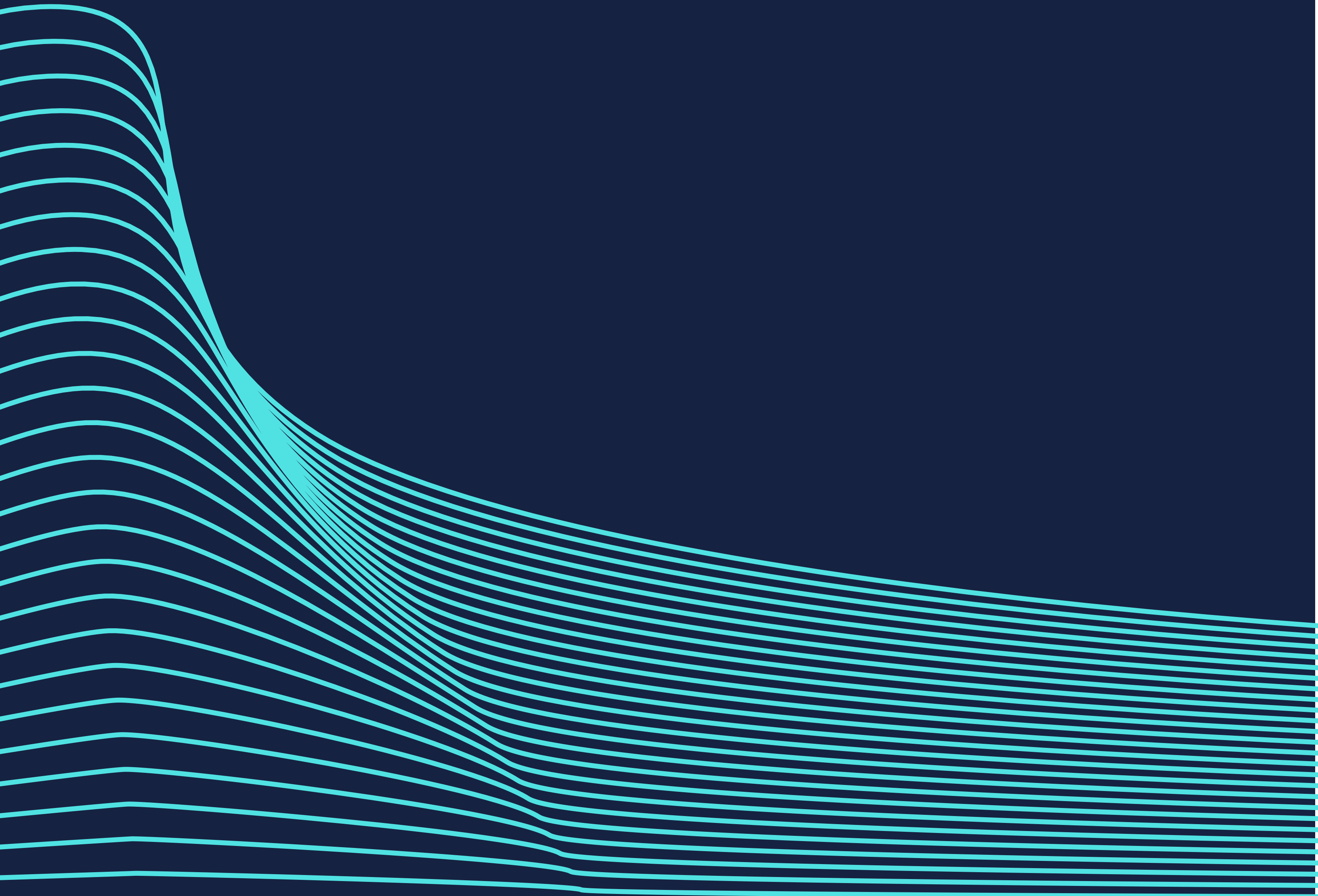
Technology Whitepaper

azul

High-Performance Microservices Using Java



A Brief History of Microservices

Although microservices as a term and a set of technologies are relatively new, its roots can be traced all the way back to the development of distributed systems in the 1960s and 70s. The concept of distributed computing is to divide an application into a number of separate, concurrent processes that communicate by message passing. Although this sounds simple, there are many challenges that make it hard to implement such systems in a reliable way using a clean design.

Many projects, technologies and standards have been developed over the last fifty years whose goals have been to solve the problem of simplifying development of distributed systems. Many of the core concepts of these technologies form the basis of Microservices.

To help understand Microservices a review of the most well-known standards and technologies is useful.

RPC: Remote Procedure Calls. RPC implements the most basic form of distributed computing, allowing an application to execute a procedure in a different address space. The coding is the same as for a local procedure without the developer coding the details of the remote interaction. RPC allows for calls to be made on the same machine or across a network.

CORBA: The Common Object Request Broker Architecture. With a shift to the use of Object Oriented (OO) programming languages, Remote Method Invocation (RMI) replaced RPC, since in OO programming methods associate behavior with objects. CORBA was designed to facilitate communication between systems that were deployed on diverse platforms. It enabled collaboration between objects running on different operating systems and hardware as well as those written in different languages. Because CORBA had such great flexibility its use was somewhat more complex, requiring the generation of stub and skeleton code as well as interaction through an Object Request Broker (ORB)

DCOM: Distributed Common Object Model is a Microsoft proprietary technology and was a direct competitor to CORBA. Both CORBA and DCOM suffered from difficulties related to the requirement of specific firewall configuration, which partially drove the development of web services.

Java EE: Servlets and Enterprise Java Beans (EJBs). Java EE was designed to separate the deployment details of an enterprise application from the implementation (making it platform neutral). This extends to being able to locate components (servlets and EJBs) at runtime on different machines through the use of the Java Naming and Directory Interface (JNDI). Resources like database, transaction and security details are provided at runtime through dependency injection.

XML-RPC: This is RPC using XML (eXtensible Markup Language) as an encoding scheme for messages and uses HTTP to transfer the messages. XML is an encoding scheme that is both human and machine-readable. Although this solved some of the problems of systems like DCOM and CORBA, it is inefficient in terms of how much data needs to be transmitted using approximately four times as many bytes as plain XML, which is also more verbose than more modern encoding schemes such as JavaScript Object Notation (JSON).

SOAP: The Simple Object Access Protocol. As more business level functionality was added to XML-RPC, this

developed into SOAP, which is one of the core protocols for what we know as Web Services. Other related protocols like the Web Service Definition Language (WSDL) allowed developers to connect to new services programmatically without the need to write specific code. These type of web services were popular initially but have been superseded by the Representational State Transfer (REST) approach. One of the biggest advantages of web services is the use of HTTP, which allows all communications to happen via IP port 80 thus eliminating many of the problems associated with distributed systems needing additional firewall configuration.

SOA: Service Oriented Architecture. This is the most recent approach to providing an architecture for the development of distributed systems using reusable components. The basic ideas are similar to previous approaches but with more of a business focus. Services logically represent a business activity with a specified outcome. They are self-contained appearing as a black-box to consumers, possibly encapsulating other services. SOA relies on standardized service contracts to define how services are called and the format of the data produced in response. There is also the concept of a service registry or repository which clients can use to locate services.

Why Have Microservices Risen in Popularity?

Given that there have been many attempts to simplify the development of distributed systems let's look at why Microservices have risen so quickly in popularity. Essentially there are four areas of software development that have created the *perfect storm* for Microservices to become successful:

1. The Cloud: The last few years have seen a massive move away from deploying applications in corporate data centres to hosting those applications in a commercial provider of computing resources such as Microsoft's Azure or Amazon Web Services (AWS). The most significant advantage of using the cloud is the flexibility it provides. Enterprises only need pay for the resources they use, as peaks and troughs of demand arise resources can be provisioned or reclaimed by the cloud provider as required. A traditional data center requires significant initial investment (Capex) followed by continued costs for running and maintenance (Opex). Often a datacenter must be designed to deal with a high peak load meaning many machines sit idle during normal load levels that are much lower. Designing software to run in the cloud requires a different, more flexible approach due to the need to adapt quickly to changing loads.

2. Agile Design Methodologies: The process of software development has long used what is referred to as the waterfall model. This is a sequential, and most importantly, non-iterative design process. Agile software development describes a set of values and principles under which requirements and solutions evolve through collaborative effort. It advocates adaptive planning, evolutionary development, early delivery, and continuous improvement, and it encourages rapid and flexible response to change. With a shift to deployment in the cloud an agile approach to development enables the benefits of deployment to the cloud to be leveraged effectively.

3. DevOps: Traditionally, the process of developing software has been distinct from that of deploying the software to production systems. This separation leads to a lack of understanding by the groups responsible for each area about the issues affecting the other. DevOps is a software development and delivery process that emphasizes communication and collaboration between product management, software development and operations professionals, requiring close alignment with business objectives. The merging of roles between development and administration is a logical progression when moving to deployment in the cloud and the use of agile software development.

4. Continuous Integration and Continuous Deployment: To support deployment in the cloud when using agile software development and the DevOps process requires a different approach to the delivery of software and new tools to support it. Continuous integration involves merging all developer working versions to a shared central copy several times a day, which enables rapid inclusion of modifications and enhancements. Continuous deployment is an approach where development teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently. This approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery, which is where open-source tools like Jenkins play an important role.

The combination of these four concepts has helped drive developers to use Microservices.

The Fundamentals of Microservices

The previous section described the waterfall software development methodology, which is relatively inflexible and does not adapt to changing requirements. The end products of this type of approach are *monolithic* applications.

A monolithic application is self-contained and independent from other applications. The design philosophy assumes that an application is responsible not just for a particular step of a task, but can perform every step needed to complete that task. The advantage of a monolithic application is its independence, meaning it can be deployed as a single unit. However, this independence is often more of a hindrance than a help regarding flexibility. Even the smallest change to the application requires the whole application to be rebuilt and redeployed.

The *Microservice* architecture takes the approach of splitting a monolithic application into numerous component tasks. The name, Microservices, can be a bit misleading as the use of the prefix micro (from the Greek word for small) would indicate that each service is, by definition, small. This is not necessarily the case. The fundamental idea of a Microservice is about doing one thing and doing that one thing well. This is based on the same philosophy used in the design of the UNIX operating system, which has numerous commands that can be linked together using pipes to transfer the output of one command to the input of another. Cassandra is a NoSQL database written in many thousands of lines of code but can be considered as a Microservice when composing an application because it only deals with one thing: persisting data.

The micro-services architecture is inherently distributed, which can lead to problems not exhibited in a monolithic application. Individual services are orchestrated to handle requests. When a service is invoked synchronously by another, the reality is that the service being invoked is unavailable or is exhibiting such high latency that it is essentially unusable. Limited resources, such as threads, might be consumed by the caller while waiting for the invoked service to respond. This could lead to resource exhaustion, which would make the calling service unable to handle other requests. When one service fails, it can potentially cascade to other services throughout the application.

To alleviate this problem, the circuit-breaker pattern is commonly used.

When one service invokes another service it does so via a proxy rather than directly. The proxy works in a similar way to an electrical circuit breaker. When the number of consecutive invocation failures exceeds a

threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout has expired, the circuit breaker allows a limited number of requests to pass through. If those requests succeed the circuit breaker resumes normal operation. If failures continue, the timeout period is restarted.

For the rest of this document we will shorten Microservice to just service unless making a specific point about Microservices.

Breaking a monolithic application into discreet services requires well-defined interfaces between the individual services. Using Microservices deliver a number of distinct advantages:

Each service can be developed by an independent team. Domain experts can be recruited to work on specific services, either in isolation or working co-operatively with other teams. This also leads to the ability to select the best technology to address the needs of that service rather than considering the application as a whole.

The implementation of a service can be changed independently of the complete application. Although the agreed interface cannot be changed without breaking existing applications, it can be extended to enable new clients to take advantage of new functionality. This could extend to completely rewriting the service or using new libraries and frameworks to improve its performance.

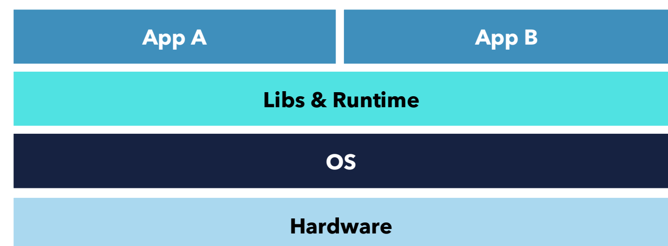
The Microservice architecture is easy to deploy into the cloud. Because the services are independent, it does not matter where they are deployed, meaning they do not need to be collocated on the same server. They can easily be distributed across numerous machines, which is very much in keeping with the ideas of distributed computing discussed at the start of the document.

Varying loads are easy to cope with. A particular service will only be able to cope with a certain transaction load. When this load is exceeded, rather than needing to move the service to a bigger, more powerful server, more instances of the service can be started and clients redirected to these new instances as required. Load balancing and clustering are built into the Microservices architecture.

How complete applications are constructed from individual services and how services are orchestrated is beyond the scope of this document, but there are popular technologies like Kubernetes and Mesos that address these requirements.

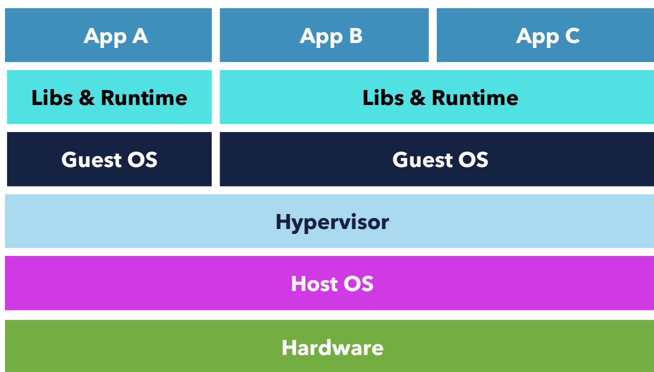
Microservices and Virtualization

To understand Microservices we need to understand how they integrate with virtualization. Without using any virtualization, an application runs on what is sometimes referred to as a bare metal server. **As the diagram below shows, the application, its libraries and possible runtime (like the JVM) are separated from the hardware by the operating system (OS).**



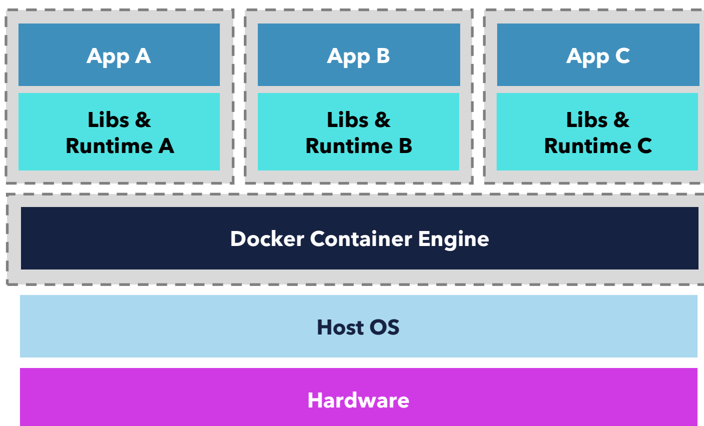
The operating system's role is to abstract away the complexities of how to interact with specific hardware at the lowest level. This takes the form of system calls, which provide an API that allows developers to perform functions like opening a file (which may provide access to a device rather than data), reading and writing data to that file as well as more complex calls like those used to control specific features (this is called an ioctl call, short for input-output control).

Today's server-class machines typically have multiple sockets for CPUs and multiple cores in those CPUs as well as many gigabytes (or even terabytes) of RAM. They also frequently have multiple network interfaces and connect to external disk arrays for storage. To utilize all these resources efficiently, it is often desirable to run multiple OS instances on the same physical server. This enables better control of resources, and more importantly, isolation of each OS instance from all others. If a problem occurs in one OS instance, it does not affect the others. **The diagram following shows how a hypervisor fits into the other parts of the system.**

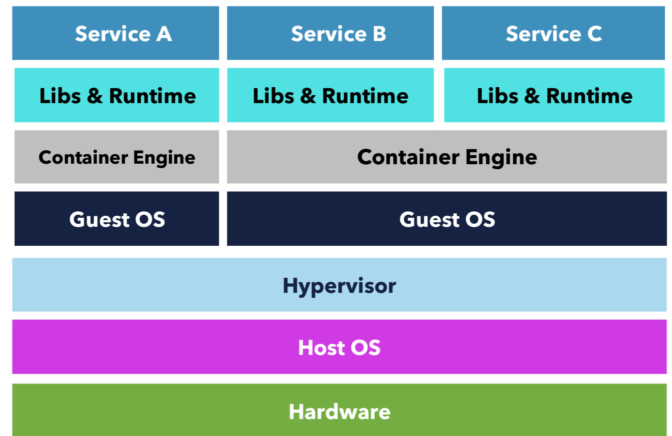


The server can either run the hypervisor directly on the hardware or it can use a host OS to support the hypervisor. The hypervisor provides an interface that, to the guest operating systems, make it appear that they are interacting directly with physical hardware. The hypervisor and host OS can impose resource restrictions (so one guest OS does not degrade the performance of others by using all the available resources) and impose the necessary isolation between guest operating systems.

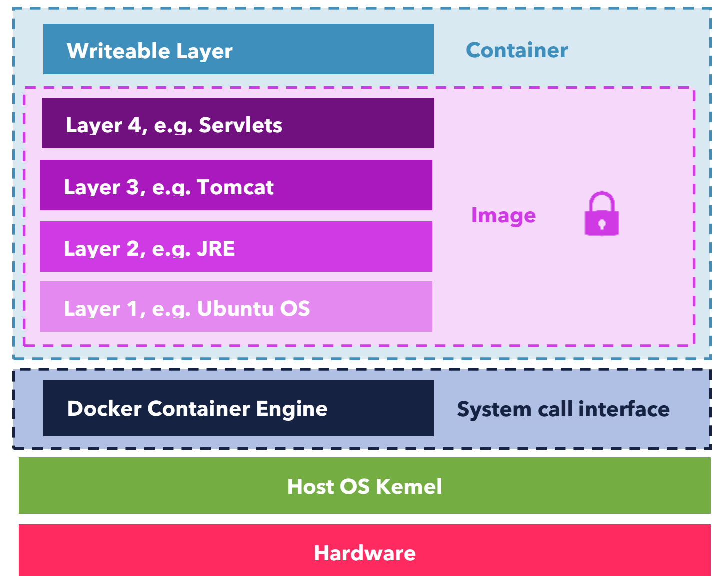
The diagram below shows how a Microservice container fits into the picture of virtualization using the popular Docker container engine.



The container looks like it provides the same role as a hypervisor but without any additional guest operating systems. The key aspect of the container is that it does not virtualize an operating system but isolates a process on the host OS from other processes, giving it many of the same advantages without the need to install a new OS. **Containers can be used in conjunction with virtualization, as shown:**



The diagram below shows how Docker images relate to a container



At the bottom, there is the host operating system kernel, and the container relies on the system call interface being consistent across all platforms that support these containers. In the case of Docker, this uses a set of Linux system calls, which will work across all modern distributions. Mac OS X is built on a UNIX style operating system (BSD), so some additional code is required to make it compatible. On Windows, which is not at all UNIX-like, a complete emulation system is required. The container interacts with the operating system directly (via system calls) and with the Docker container engine to provide lifecycle management, administration and so on.

The container itself consists of several layers that build up the image in which the application is going to run. In this example, there is an Ubuntu layer to provide a consistent set of OS services above those of the system calls. This will be things like device files, daemons, etc. Above that there is a Java Runtime Environment layer (JRE). This demonstrates one of the big advantages of containers. We may well have microservices that have been developed using different versions of Java.

By putting the JRE inside the container, this immediately eliminates the issue of whether the correct version of the JRE is installed on the machine the application is being deployed to. Above the JRE there is a Tomcat layer. Again, this is included in the container so that the need to have a specific version of Tomcat installed on the target machine is not necessary. The final layer of the Docker image is the servlets that will provide the functionality of the service.

A key point here is that all the layers in the image are read-only. Nothing can be changed in these layers. This provides a significant advantage when multiple services are running on the same machine. Rather than each service needing to have its own copy of the Ubuntu, JRE, Tomcat and Servlet layer they can all share a single copy.

The final layer in the container, which is not part of the image, is the Writeable layer. If any part of a layer in the image needs to be modified a copy of that part is taken and placed in the Writeable layer. The container maps read and write operations to the Writeable layer giving the illusion that the lower layers can be modified.

The Docker container relies on two features of the Linux kernel to enable it to isolate containers from one another:

Namespaces are a feature that isolates and virtualizes system resources used by a collection of processes. Currently, six namespaces are supported: pid (processes), net (networking), mnt (mounted filesystems), uts (hostname), ipc (inter-process communication) and user (user and group ids). A container is bound to a specific namespace so that it is isolated from the namespaces used for other containers. By doing this, the service running in the container will only see the resources configured for that namespace.

The service will only be able to view parts of the file system, a subset of OS processes, certain network interfaces and so on.

Cgroups (Control Groups:) is a feature that limits, accounts for, and isolates the resources usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. A container can be run within a cgroup so that it only has access to a portion of the available resources of the physical machine. When running multiple containers on a machine, this can be very useful for dividing the available resources in a fair and equitable way. If one container develops a fault that would normally cause it to starve other processes of resources, the cgroup implementation prevents the from happening.

As can be seen, containers provide a very useful way to deploy precisely defined services isolated from one another with strict control over their resource utilization when in operation.

Java-Based Microservices

Java is very complementary to the Microservice architecture. There are three ways that Java can be used to provide Microservices:

1. Container-less Services: This approach provides the minimum level of isolation between services. The service is delivered as a single JAR file that contains all the code of that particular service. The service is run using a single, system-wide JVM, which means that all services must be aligned to the same version of the JVM. Similarly, all services typically make use of system-wide copies of libraries and frameworks they require with the same implication of alignment on

specific versions. It is possible (although more complex for deployment and management) to use different versions of libraries and frameworks.

2. Self-Contained Services: These services are similar to container-less services. These services will also use a system-wide JVM requiring alignment on a specific version. However, any libraries that are required by the service are included in the JAR eliminating issues of version mismatch. However, this increases the size of JAR files since each service must include all the libraries it uses regardless of whether these are shared with other services. There are several technologies available to enable self-contained services such as Spring Boot, DropWizard and WildFly Swarm; all of which simplify the process of developing and delivering these types of services.

3. In-Container Services: This is the most flexible approach to delivering Java-based Microservices. Containers eliminate the problems of how to manage multiple versions of the Java runtime on a single machine and ensure that each service uses the version of the runtime and libraries required by that service. To the OS, the JVM is just another process and is not treated differently in respect of namespaces and cgroups. Deploying services in Docker images greatly simplifies the issues of JVM and library versions. By using read-only layers for these components minimizes storage by enabling sharing between different Microservices.

The use of containers is the most popular way to deploy Java-based Microservices because of the advantages it provides.

JVM Challenges With Microservices

To deploy Java-based Microservices requires some careful consideration of how the JVM works in this type of environment. Three specific things need to be addressed:

1. Responsiveness under load: Ideally as the load on the service increases there should be no degradation in performance of that service. That is, after all, one of the benefits of using a Microservice architecture and deploying into the cloud. New service instances can be started, ensuring that existing instances deliver consistent performance as load. As the load on an instance of a service increases, this will typically result in more object creation and an increased load on the garbage collector, which must be accounted for in the service deployment configuration.

2. Supporting multiple JVM instances on the same physical hardware. *This is not the issue of how to* manage multiple versions of the Java runtime, but how to deal with having multiple JVM processes running on the same machine. The ability to do this in a way that optimizes the use of resources is critical to being able to scale services on a particular server.

3. How to add ("spin up") new service instances quickly and efficiently. A key feature of Microservices is the elasticity they provide for handling varying levels of workload. The need to start up new instances of services as the load increases require those services to be available and performing at their optimum level as quickly as possible. A traditional JVM takes time to warm-up as it analyses the bytecodes being used and compiles frequently used methods into native instructions.

To effectively deploy Java-based Microservices it is essential to address all of the points above. Azul have developed Azul Platform Prime, centered upon a JVM that provides enhanced performance for many categories of applications, one of which is Microservices running inside containers.

Azul Platform Prime for Microservices

Azul is a company whose focus is purely on developing JVMs. In this section, we'll explain how Azul Platform Prime is ideally suited to deliver better performance and reduced cost for the deployment of Java-based Microservices.

Azul Platform Prime is based on the open-source OpenJDK source code, which is the reference implementation for the Java SE standard. To deliver better performance than traditional JVMs, parts of the core JVM are replaced with alternative implementations. Specifically, this includes the memory management system and part of the Just in Time (JIT) compilation system.

JVM Memory Management and C4

When a Java application is executed, memory management is handled automatically by the JVM. The JVM allocates space on the heap when a new object is instantiated and reclaims that space when the application no longer has any references to it. This is the process of garbage collection (GC). The GC also manages the heap, optimizing the availability of space for new objects by periodically compacting the heap, moving objects to make live data contiguous, eliminating fragmentation. GC provides significant advantages over languages like C and C++, which rely on the programmer to explicitly deallocate space used by the application. This explicit deallocation can be the source of many application bugs, particularly in the form of memory leaks or abrupt application termination when the programmer tries to deallocate an invalid or incorrect address.

However, in a traditional JVM, GC, while very useful, also requires application threads to be paused to avoid corruption of data as objects are moved during compaction. This effect is two-fold: firstly, it introduces pauses to an application while the GC performs its work. The length of these pauses can be very long, ranging from milliseconds to hours and the length is directly proportional to the amount of memory allocated to the heap, not how much data is in the heap. Secondly, when these pauses occur cannot be accurately predicted. This introduces non-deterministic behavior to an application. In the context of a Microservice, this could lead to clients of a service incorrectly assuming that the service is no longer available even though its JVM is just working on GC. This may cause new service instances to be started, further impacting performance when this is not necessary.

Azul Platform Prime uses a different GC algorithm, the Continuous Concurrent Compacting Collector (C4). Unlike other commercial GC algorithms, application threads can continue to operate whilst the GC is performing its work (hence the concurrent part of the name). Although other algorithms like Concurrent Mark Sweep (CMS) and Garbage First (G1) perform part of their work while application threads are running other parts still require application threads to be paused. Both these algorithms will fall back to a full compacting collection cycle if they reach a point where they are unable to meet the memory needs of the application. This full compaction will require application pause times proportional to the heap size. C4 will not, under any circumstances, perform a full compaction since it is never necessary. With C4 collection happens as a background task to the application work resulting in an extremely low impact on the application performance.

Unlike traditional collectors that require numerous command line options that are difficult to configure correctly C4 only needs the heap size to be set. C4 scales from 1Gb to 8Tb of heap space without increasing application pause times.

JIT Compilation

Java source code is compiled to bytecodes. Unlike statically compiled languages like C and C++, these bytecodes are not the instructions of the specific platform on which the application will run. Bytecodes are instructions for a virtual machine, in this case, the Java Virtual Machine (JVM). When the JVM starts an application, it has to interpret each bytecode, converting it to the necessary native instructions and operating system calls. This interpretation incurs overhead, and the code will run at a much slower rate than statically compiled code.

To eliminate this problem, the JVM profiles the code as it interprets the bytecodes, keeping a count of how many times groups of bytecodes are used and how many times methods are called. This profiling data is used by the JVM to identify hot spots in the code, which can be compiled to native instructions using a just in time (JIT) compiler. Over time the majority of code executed by the application will be compiled but the time it takes to get to this point is referred to as the warm-up time of the application.

A traditional JVM employs two JIT compilers called C1 and C2 (sometimes referred to as client and server). Each JIT has different characteristics, which govern application performance. C1 is designed to compile bytecodes quickly but with a lower level of code optimization. This is well suited to short-lived applications that need to warm-up quickly. C2, conversely, takes longer to compile the code but applies more optimizations in the code that it generates. This is better suited to longer running (typically server) type of applications where a warm-up phase that may take several minutes is not an issue. Since JDK 7 both JITs can be used for the same application through tiered compilation. C1 is used initially; C2 is used as the application runs for longer to provide increased performance.

In Azul Platform Prime the C2 JIT has been replaced with a new,improved version called Falcon. Falcon is based on the open-source LLVM compiler project, which is supported by numerous corporations and individuals including Intel, NVidia, Apple and Sony. This enables Falcon to take advantage of a much larger range of contributions than from just Azul's engineers.

Falcon is modular in its design allowing new optimizations to be added easily, as they become available. As an example, Falcon can optimise much more complex code than C2 and apply vector-processing optimizations to it. Through the use of single instruction, multiple data (SIMD) instructions in the most recent Intel processors (such as AVX 2 and AVX512) code generated by Falcon can dramatically improve the performance of certain loops. Overall the effect is to deliver a higher level of application performance.

ReadyNow!

The issue of application warm-up is exacerbated by the fact that each time an application is started the JVM has no knowledge of what might have happened during previous executions. Each time the application starts the JVM must go through the same profiling phase, performing the same analysis and, most likely, compiling the same sections of code. Clearly, this is not the most efficient approach for an application that needs to be frequently restarted.

Azul has addressed this problem through a technology called ReadyNow! Using this, an application is started in a production environment and allowed to run until it is fully warmed up. At this point, a profile of the application's JIT status is recorded.

An obvious way of solving this problem would be to take a snapshot of the compiled code when the application has reached a steady state and when all necessary code has been compiled. When the application is restarted this snapshot could be reloaded, and the application could continue as if it had not stopped (from a compilation point of view).

Unfortunately, things are not as simple as this (the implementation of this would be far from simple, even though the description makes it sound that way). The definition of the JVM places restrictions on what can happen when it starts, specifically in the area of class loading and initialization. Several other issues make this approach impractical.

Azul has addressed this problem through a technology called ReadyNow! Using this, an application is started in a production environment and allowed to run until it is fully warmed up. At this point, a profile of the application's JIT status is recorded. This profile records details of the classes that are currently loaded, classes that are initialized, instruction profiling data (similar to the data used by the JVM to decide which sections of code to compile) and speculative optimization failures. When the application is started again, the profile can be used to have the JVM and JIT perform almost all the work it would normally do during the warm-up phase of the application; in this case, this can all happen before the application starts executing code in the main() method. This all but eliminates the warm-up phase so the application can start running at very nearly full speed when it starts.

As you can see, Azul Platform Prime offers numerous advantages over a traditional JVM for optimizing performance. Looking specifically at Microservices, these features address all of the issues highlighted in the previous section:

1. Responsiveness under load: The C4 collector will not resort to a full stop-the-world compacting collection so, even under heavy load; the performance of the JVM memory management remains the same. Where services become memory constrained, it is simple to increase the allocated heap space to solve potential out of memory problems without affecting performance.

The significant impact of the reduced and consistent latency produced by the C4 algorithm applies to circuit-breakers. There is a substantial reduction in how often circuit-breakers need to be activated in a system and the long timeouts before they reset.

2. Supporting multiple JVM instances on the same machine: Azul Platform Prime works in conjunction with the Linux operating system to optimize the memory management sub-system. The Azul Platform Prime System Tools will reserve an area of physical memory (the size of which is configurable), which is shared between all instances of the Azul Platform Prime VM running on that machine. The Azul Platform Prime System Tools also provides greater performance for Azul Platform Prime by reusing memory pages that are already in cache (hot pages), unlike a traditional JVM that frequently gets cold pages from the OS as it allocates new objects on the heap.

3. Spinning up new server instances quickly and efficiently: The ReadyNow! feature of Azul Platform Prime is ideally suited to enabling new service instances to start quickly and provide a full level of performance straight away.

Conclusions

It is clear that the direction of software development is primarily moving to the cloud and using Microservices as a way to do this in a flexible, scalable way.

Java, as the most popular programming language on the planet, is an obvious choice for developing Microservices. Java Microservices also provide several advantages over those developed in other languages. However, using a traditional JVM, several deployment issues must be carefully considered when choosing an application architecture.

Azul Platform Prime includes a modern JVM that uses different internal algorithms for Garbage Collection and part of the JIT compilation system. When combined with the Ready-Now! technology to effectively eliminate the warm-up phase of Microservice performance, Azul Platform Prime becomes a clear choice for use in modern Java application development on microservices.

Azul Platform Prime can be tried free for 30 days and is available at www.azul.com/zingtrial

Contact Azul

To discover how Azul Platform Prime can improve scalability, consistency and performance of all your Java deployments, contact:

385 Moffett Park Drive, Suite 115

Sunnyvale, CA 94089 USA

+1.650.230.6500

www.azul.com
info@azul.com