**Literature Review: Migrating to a Container-Based Microservices Architecture with Docker, by Camron Khan**

November 3, 2017

INTRODUCTION

Many major companies – such as Microsoft, IBM, Netflix, and BBC – have begun implementing a microservice architecture paired with DevOps practices in recent years to improve their flexibility and scalability [1]. Although there are many competing definitions of a microservice, a microservice can be characterized as an application that (1) fulfills a distinct business requirement, (2) communicates with other microservices via a programming language neutral API (application programming interface), (3) is loosely coupled from other microservices with which it interacts, and (4) can be deployed independently of such microservices [2]. A microservice architecture implemented using DevOps practices offers a number of advantages such as the ability to deploy related but independent services at varying intervals, the flexibility to develop different services with different programming languages, and the capacity to horizontally scale out services to meet changing user request levels or maintain client SLA (service level agreement) requirements for high availability [1].

Currently, the trend in implementing this strategy is moving from a virtualization-based approach to a containerization-based approach, specifically with the use of Docker as the containerization framework [3]. Although the tide seems to be shifting towards the latter, using containerization technology in a production-grade, distributed environment comes with its own pitfalls. Therefore, the following will examine what containerization means, its strengths and weaknesses, and how it compares to current virtualization-based approaches in the context of performing common DevOps tasks.

IMPORTANT IDEAS

A container is an abstraction over an operating system (OS) that contains the system resources, dependencies, and environment configurations necessary to run an autonomous application [4]. Resource containers are defined by a fine-grained control over a system’s resource consumption [4] and the independence and reproducibility of application execution environments [3]. Although similar to a virtual machine (VM) in many ways, a container does not contain its own independent OS kernel. Instead, all containers running on a system share that system’s kernel. As a result, a container does not store the files related to running the OS, but rather, simply the files required to run the application(s) within the container [6].

Docker is arguably the most popular containerization framework currently in use. More specifically, it is a collection of open-source containerization tools which are designed to assist development teams quickly deliver deployable applications via resource virtualization and container-based workflows [2]. It includes a client-server application, image registry, and additional applications for container configuration and orchestration [6]. The client-server application (Docker Engine) is comprised of a client and daemon that can communicate via a REST (Representational State Transfer) API, UNIX sockets, or a network interface. The client can send requests to the daemon to build and run images, which are read-only instructions for creating Docker containers. Users can save images in registries (Docker Hub or Docker Cloud) for reuse and sharing, which can be downloaded at any time by the daemon. Multiple copies of an image can be run at once to scale out an application [7].

Docker – and containerization more generally – have a number of strengths in the context of deploying microservices and following DevOps practices, such as code portability, application environment independence, lifecycle automation and management, and resource utilization. The following will look at each in turn.

Containerization proponents cite code portability as one of the most significant gains by switching to a container-based architecture, which is exemplified in the mantra, “develop once, deploy everywhere”. The use of read-only images that can be pushed up to a registry and reused over and over allows developers to not only reuse code but also reuse system configurations and dependencies as easily as code itself. This feature is most pronounced on development teams where traditionally each developer must set up her own development environment prior to running an application. Docker would allow the same developer to simply pull the image containing the application from Docker Hub, and run the application in the same environment as the colleague who developed it. This also reduces the chances that code running on one’s machine cannot run on another’s machine [8].

Another touted feature of containers is application independence – or loose coupling – which results in two major benefits: (1) dependency isolation and (2) language-neutrality. Because an application’s environment is isolated from other applications’ environments, two applications with different versions of the same package or library can easily be developed and executed simultaneously. Developers do not need to worry if such dependencies will break current code in other parts of the ecosystem if they are deployed in separate containers [2].

Deploying applications in their own independent environments also means that applications working together in the same microservice ecosystem can be developed with incompatible programming languages. If a microservice has its own independent runtime environment and communicates with other applications via a language-neutral interface (i.e., REST APIs), then the programming language in which that microservice was written has no bearing on any other service. This has the advantage of allowing developers to use the right tool for the job – whether due to developer proficiency or application performance requirements [2].

It is also argued that application lifecycle automation and management becomes easier with a container-based solution. For example, Docker containers can be created and deployed via scripts, which allows DevOps teams to automate the process of deploying applications from a development environment to a production environment [2]. Once deployed in production, applications can also be easily rolled back to previous versions if bugs are identified. Rolling back a Docker image simply entails deploying the previous version of the image. Once the development team has built a new image with the bugfix, a new Docker image can be redeployed to production [8].

The final major benefit examined here is resource utilization. Because containers use compute and memory resources more efficiently than virtual machines, more containers can be run on a single host [8]. As a result, they have a smaller resource footprint due to each container sharing the underlying host OS’s resources with the other containers [2]. Finally, because resource consumption restrictions can be implemented at the container level, each container can be fine-tuned to only use the resources necessary to run its application(s) [6].

Although containerization has many benefits, there are also challenges associated with implementing a container-based design for a microservice architecture, such as managing dependencies among services and persisting application state. Each consideration will be reviewed in turn.

A microservice architecture with containerization moves the dependency problem from the application level to the service level. Rather than multiple applications relying on the same environment and libraries, a microservice will likely be dependent on other microservices. This dependency means that adding or removing (via manual intervention or application crash) can have rippling effects on other services in the ecosystem [8]. Although this problem exists for microservices generally, there are currently not many off the shelf solutions that are straightforward to implement and are production tested.

Persisting application state is another shaky area for containerization and Docker specifically. Given that Docker containers are supposed to run independently of the host OS, theoretically there should be no way to save application state data other than building a new image version with the data-to-be-persisted. There are a number of workarounds, each with individual security and performance tradeoffs, such as shared storage on the host OS and disk volume creation. However, there is no straightforward way to perform in-memory application state replication across containers [8]. This means that if a container goes down, all the application state data will be lost.

Now that the conceptual foundation has been laid and the strengths and weaknesses discussed regarding containerization, it is worth comparing containerization-based solutions against virtualization-based solutions on performance metrics relating to DevOps practices. Research has demonstrated that the former outperforms the latter in three critical areas: (1) scaling out nodes, (2) performing rolling upgrades, and (3) executing failover and recovery processes. Compared to VMs, additional containers can be deployed in nearly half the time. VMs usually utilize more of a host’s system resources when being initialized, and as a result, this can lead to bottlenecks during the scale out period and in turn lead to longer deployment times. A container’s ability to outperform a VM in scaling out applications also allows it to outperform a VM in performing rolling upgrades. Since a rolling upgrade, in the context of containers and VMs, simply involves spinning up instances of the upgraded node and then switching over traffic to that instance, the time required to scale out a node also determines the time required to upgrade a node. Finally, containers have been shown to outperform VMs in regards to the overall recovery time to recreate a failed node [8].

CONCLUSION

Docker, and container-based approaches to designing microservice architectures more generally, is an exciting area of current development. With such benefits as improved code portability, application environment independence, increased automation in deployment and maintenance, and better resource utilization, it is expected to see more organizations adopt this technology. Furthermore, as these tools mature in production environments, it can be expected that challenges such as managing service dependencies and persisting container applications states will be actively addressed by the developer community. As a result, development, DevOps, and infrastructure teams may experience a major shift in their day-to-day workflows in coming years.

REFERENCES

1. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture", IEEE Software, vol. 33, no. 3, pp. 42-52, 2016.
2. D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology", SoutheastCon 2016, Norfolk, VA, USA, March 30-April 3, 2016, IEEE, 2016, pp. 1-5.
3. A. Sill, "The design and architecture of microservices", IEEE Cloud Computing, vol. 3, no. 5, pp. 76-80, 2016.
4. G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems", in the Proc. of the 3rd Symposium on Operating Systems Design and Implementation, OSDI 1999, New Orleans, LA, USA, February, 1999, pp. 45-58.
5. S. Julian, M. Shuey, and S. Cook, "Containers in research: Initial experiences with lightweight infrastructure", in Proc. of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, XSEDE16, Miami, FL, USA, July 17-21, 2016, New York: ACM, 2016.
6. M. Fokaefs, C. Barna, R. Veleda, M. Litoiu, J. Wigglesworth, and R. Mateescu, "Enabling devops for containerized data-intensive applications: An exploratory study", in Proc. of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON '16, Toronto, Ontario, Canada, October 31-November 02, 2016, Riverton: ACM, 2016, pp. 138-148.
7. "Docker Overview", docker.com. [Online]. Available: https://docs.docker.com/engine/docker-overview/. [Accessed Oct. 31, 2017].
8. H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops", 2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016, IEEE, 2016, pp. 202-211.