Troubleshooting Comparison of VMs and Containers

Bassam Gamal
Eric Söderberg

May 31, 2022

1 Introduction

1.1 Relevance

A core part of software engineering is troubleshooting issues, and it is especially central for DevOps. Engineers usually spend a significant portion of their day trying to understand problems, reproduce scenarios, debug, and fix issues. Some of these issues might cost the company millions of dollars, so finding and fixing them needs to be a very fast process. To find them, engineers need to know technical details about the issue, the environment, the time, what application or even the line of code triggered it, and many other things. It must also be reproducible, and testable.

1.2 Troubleshooting Problems

As the industry has moved increasingly from VMs to containerized applications, with the relevant pieces of troubleshooting information possibly spanning a complex and ephemeral cloud-based architecture, the need for new tools to tackle troubleshooting under these circumstances arose.

In this essay, we compare troubleshooting in VMs and containers. Their respective requirements and difficulties are accounted for, and different tools to solve the different problems are presented. The solutions include state-of-the-art concepts to deal with logging, aggregation, and troubleshooting and what importance they play for VMs and containers. Although the focus is on concepts and their solutions, examples of a few appropriate tools are also presented. Finally, the pros & cons during the troubleshooting process of the respective paradigm are explained and discussed.

2 Background

2.1 Virtual Machine (VM)

A virtual machine, *VM* for short, is an emulated isolated machine running its own ("guest") operating system inside of another host. The virtual machine's state can be saved to create a snapshot, and easily duplicated. These features mean that they are quick to deploy, easy to reason about and provide both isolation and some degree of reproducibility, in comparison to deploying an application directly on a machine.

2.2 Containers

A container utilizes virtualization to run one or more isolated processes. Commonly this is done to provide very granular and reproducible control of the processes' dependencies and environment. Unlike in a virtual machine, a container commonly shares parts of the operating system kernel with its host and other containers. This allows it to have a lighter footprint, as the entire operating system does not need to be replicated for each container - which lets it scale much more efficiently. They are often designed to be pseudo-stateless, starting from a preset state with each new startup.

2.2.1 Docker

Docker is today one of the most popular tools for running containerized applications, to the extent where Docker is even sometimes used synonymously with containers. As can be seen in fig. 1, it continues to grow in popularity every year. A Docker image is a recipe that, through a so-called dockerfile, describes the steps to be performed to run an application with a very specific environment, and a docker container is an instance of such an image. Docker also utilizes the kernel-sharing techniques mentioned in section 2.2.

2.2.2 Orchestrator

As the use of horizontal scaling has increased, so has the need for a means of controlling the large number of containers spread across multiple machines. An orchestrator, such as Kubernetes, is responsible for deploying, running, and managing containers. It provides benefits such as scalability, high availability, service discovery, and much more out of the box. Notable features of its scalability

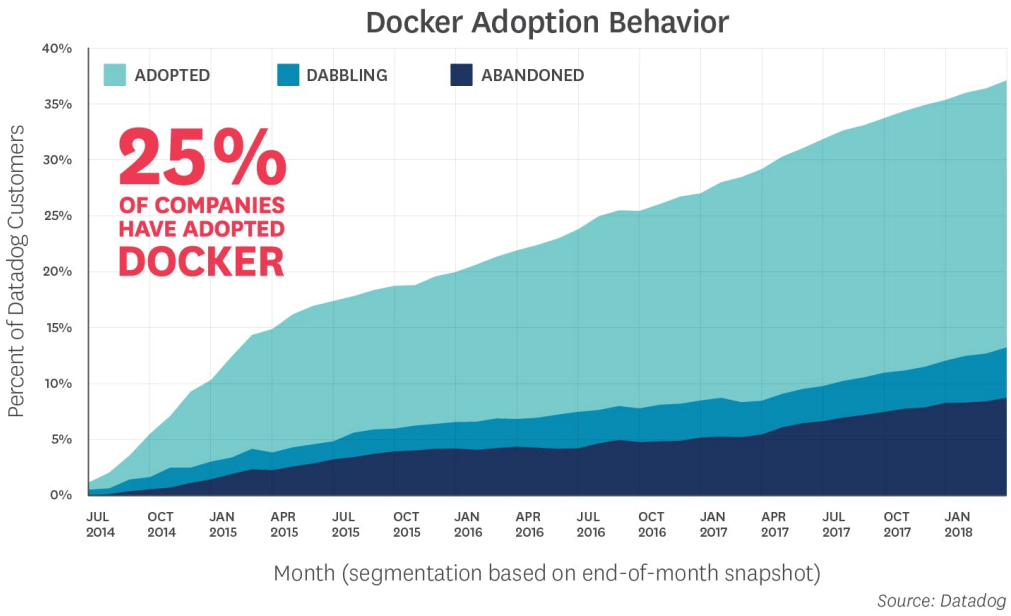

Figure 1: Docker adoption over time, as of 2018 [4].

include the ability to distribute containers automatically across available resources (such as different machines), and even provision more resources if deployed in a cloud context.

2.3 Microservice

A microservice is a small self-contained application or service with a focused set of either domain or business functionalities. Though they do not per definition need to be built and run as containers, they often are due to their lightweight properties described in section **??**. The design philosophy behind microservices tends to include properties such as being lightweight and independent of other parts of the system it is to be used in, and easily deployable in different environments.

2.4 Log Aggregation

Log aggregation refers to the practice of collecting logs from many sources into one centralized location. Two common and highly scalable tools for this purpose are ELK [1] and Kafka [2]. ELK gets its name from the three projects it encompasses; Elasticsearch, Logstash, and Kibana. The ELK stacks' components work in tandem to allow logs to be aggregated and analyzed and provide various analytics and metrics, and other forms of troubleshooting and monitoring aids. Kafka serves a similar purpose, facilitating high-performance data collection pipelines.

3 Current State of the Art

3.1 Virtual Machines

Usually, engineers run applications on different VMs. Each isolated cluster of connected VMs represents an environment. Managing the VMs, replicating, and backing them up is the job of system engineers. Any incorrect configurations between VMs will cause random bugs to appear.

Depending on the transaction scale, applications log their information to different destinations. If the scale is small, then it is sufficient to log to files. If the scale is medium, databases are used instead. Finally, if the scale is huge, then tools like Kafka, ELK stack, and many others are used, as they easily process and store these huge amounts of logs.

A typical troubleshooting process would start by checking the logs. If the issue is pinpointed to a specific app/API, then the engineer will solve the issue, write more tests, and go through a deployment cycle. If the logs didn't have enough details, then the engineer would have to add more logs. Sometimes even that doesn't help, in which case engineers would check operating system logs, as it could be something from the environment causing the error.

3.2 Containers & Orchestration

The move to containers has enabled systems to grow huge and become distributed. It also means that reproducibility can be ensured on a level not previously possible. The software's deployment environment can be ensured to have core characteristics and underlying structures identical to that of the engineer. Containers largely eliminate issues stemming from a difference in operating systems or package versions, in many cases freeing up the engineer to focus on troubleshooting issues in code and deployment rather than the environment.

Ideas like microservices have become feasible and horizontal scaling has turned effortless. Any app can have several replicas running with a simple setup. Engineers can also easily deploy new versions to various stages of deployment, enabling rapid debugging, testing, and eventually deployment to production.

However, these advances didn't come without downsides; troubleshooting problems in a heterogeneous distributed system is far more complicated than in a monolithic system. Logging on the filesystem isn't an option. Many replicas can run simultaneously, and containers are usually ephemeral. Storing logs in a transactional database isn't feasible, as the volume of logs increases drastically, leading to high latency costs. As an example, Netflix logs hundreds of billions of events *every day*, or an excess of 2 million per second [8].

The need for more complicated tools arose so that engineers could quickly identify strange code behaviors and isolate them in such systems. Engineers have therefore started to embrace DevSecOps culture, and use new troubleshooting tools to make sure their systems are up and running. New strategies to gather logs were used, like Sidecar containers that ship logs to Kafka, or ELK stack in a loosely coupled way that doesn't affect the performance of the applications. New monitoring tools are used to monitor app performance and metrics. Questions like "why does the CPU go up at 1:00 am?" are part of what engineers troubleshoot now. Observability is a new term that means engineers understand their systems more and can ask questions that add value to the business - which is a step away from just monitoring graphs and numbers.

Typical troubleshooting in containers is to search for the unique id that links all related transactions. Then go through the list of requests, and responses. If engineers have found that an application or API caused the issue, then a code fix can be worked on to resolve the issue. If the issue appears to stem from the infrastructure, then a more thorough investigation is done to identify the correct fix. After that, the engineer can simply go through a testing and deployment cycle.

4 Comparison

4.1 Scaling & Ephemeral Life-cycle

While virtual machines are not effortlessly scaled horizontally, containers generally are. The applications running on a virtual machine are therefore usually more monolithic, compared to the more microservice-favored approach of containers.

This leads to an important difference in how troubleshooting must be approached for containers. Containers may only exist for a short amount of time, or (by design) lose their data upon a shutdown or crash. The error may manifest itself only in a subset of containers, or rely on the interaction with each other or other services. Information about each container's state and other possible affecting variables must be aggregated in a central location, as the troubleshooting engineer cannot often gather any of this information after the error has been detected.

An engineer who wishes to learn more about the state of a virtual machine, however, may simply be able to access said machine directly, through SSH or similar remote access tools. Its logs *may* still be aggregated centrally for efficiency's sake - but an engineer can easily poke around inside of the virtual machine or a clone of it. Once inside, the engineer can identify and collect information of diagnostical

interest. Applications, even those consisting of many layers, all reside in one singular place if running inside of a VM. This is however not at all a scalable solution. Conversely, all of the information that can be thought to be relevant must often be logged pre-emptively in the case of the containers.

4.2 Resource Requirements

Since information about a container's environment cannot be trusted to still be accessible by the time the engineer is notified that there is an issue, all relevant information must have been logged prior and stored outside of the container. What constitutes "relevant" information immediately becomes a point of contention; as this can be very difficult to predict, and thus logging must err on the side of caution and rather aim to log too much than too little.

An application relying on a multitude of microservices which all have to log information about their environment, state, and events, will in total generate a massive amount of logs that must be stored in a central location.

For containers, this trivially leads to significantly increased storage requirements, but also processing requirements to be able to process logs and flag or predict errors as they occur. Even for virtual machines which send and aggregate their logs centrally, the engineer can generally rely on artifacts of the failing virtual environment still existing for inspection and analysis purposes, reducing the amount of data that must be logged.

4.3 Speed of Troubleshooting

The complexity of potentially cloud-based architectures spanning many machines and even more containers yet, and what it means for troubleshooting, should not be underestimated. What led to a single virtual machine's failure may, with appropriate logging, be deduced with the help of logs and an investigation of the offending environment. Even with all aggregated logs and proper monitoring configured, the process is all but guaranteed to be more complex in the case of containers. As containers tend more toward horizontal than vertical scaling, it will be much more time-consuming to pinpoint the actual *origin* of the error, even if the container upon which it manifested can easier be found.

On the flip side, engineers will be helped greatly by the stronger reproducibility of containers over VMs. While it is easier to investigate the state of a virtual machine if you know what you're looking for, the task can quickly become incredibly time-consuming once an engineer has to look into issues stemming from differences in the VM's environment and/or its configuration, or simply ensuring it is deployed in a properly reproduced state for testing. The containers on the other hand follow a more rigorous standard of reproducible deployment. This means that finding some bugs will be faster in a VM, but others, such as issues stemming from failing reproducibility upon deployment, can be much, much slower to troubleshoot.

Finally, as containers are much faster to deploy than virtual machines, and integrate very nicely in CI/CD pipelines, it allows much quicker testing to ensure that the bug is fixed - as well as facilitates a quicker deployment once the troubleshooting process is completed.

4.4 Complexity

The container troubleshooting process requires a higher degree of knowledge and time to implement. Any programmer can install a tool on a machine, or write to a file on a disk as is the case in the more primitive cases of VM logging. Even advanced single-machine logging tools typically require less configuration as they by default require no information about other services or otherwise communicate with other machines or orchestrators. In contrast, the more advanced state-of-the-art tools for container logging, especially in a cloud context, require knowledge outside the scope of traditional programming and systems knowledge. The underlying concepts and important theory may be hard to grasp, such as important performance characteristics when picking the correct product.

Once set up properly, maintenance of the system remains a complex moving piece that requires an ongoing investment in knowledgeable maintainers.

5 Conclusions & Reflection

From the authors' professional experience, the conclusion is that troubleshooting, whether done in VMs or containers, can today benefit from a large array of logging and monitoring tools to quickly help engineers identify the cause of the issue. While some may be used for either paradigm, it is clear that engineers must take some of the differences outlined herein into account when choosing their troubleshooting or even deployment strategy.

A complex cloud-based logging system may be appropriate for a large company, and less so for a small hobby project. In general, it's a balancing act between front-loaded work plus heavier maintenance - at the benefit of aiding scalability, troubleshooting, and deployment - and opting for simpler solutions in less mission-critical systems with no existing base for advanced concepts like log aggregation and monitoring.

We believe that it is part of the engineer's job to be able to identify the right tools for the job, and take troubleshooting into account when designing systems and choosing architectures. It thus also falls upon the engineer to perform an analysis of the expected gain over more or less advanced solutions, both in the short- and long term of the product.

References

[1] Amazon. The elk stack. https://aws.amazon.com/opensearch-service/the-elk-stack/.

[2] Apache. Kafka. https://kafka.apache.org/.

[3] Boettiger, C. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev. 49*, 1 (jan 2015), 71–79.

[4] Datadog. 8 surprising facts about real Docker adoption. https://www.datadoghq.com/docker-adoption/, June 2018.

[5] Davoudian, et al. Big data systems: A software engineering perspective. *ACM Computing Surveys (CSUR) 53*, 5 (2020), 1–39.

[6] Dinh-Tuan, et al. Development frameworks for microservice-based applications: Evaluation and comparison. In *Proceedings of the 2020 European Symposium on Software Engineering* (2020), pp. 12–20.

[7] Pourmajidi, et al. On challenges of cloud monitoring. *arXiv preprint arXiv:1806.05914* (2018).

[8] Sykes, B. How netflix uses druid for real-time insights to ensure a high-quality experience, mar 2020. https://netflixtechblog.com/19e1e8568d06,retrieved on 2022-05-31.

[9] Yussupov, et al. Faasten your decisions: A classification framework and technology review of function-as-a-service platforms. *Journal of Systems and Software 175* (2021), 110906.

[10] Łukasz Kufel. Tools for distributed systems monitoring. *Foundations of Computing and Decision Sciences 41*, 4 (2016), 237–260.