Microservices und Sicherheit

Seminar Komponenten, Agenten und Workflows in verteilten Systemen

Felix Ortmann und Konstantin Möllers
{0ortmann,1kmoelle}@informatik.uni-hamburg.de

Universität Hamburg
Fachbereich Informatik
Vogt-Kölln-Straße 30
22527 Hamburg

Zusammenfassung.

1 Einführung

2 Microservice-Architekturen und Sicherheit

Ein Microservice-basiertes System ist ein verteiltes System, das in der Regel auf mehreren Maschinen läuft. Auf jeder dieser Maschine laufen ein oder mehrere Microservices. Viele Teile des Systems müssen kommunizieren, entweder es werden Nachrichten bzw. Tasks entgegen genommen oder Nachrichten verschickt (Newman, 2015). Ein breiter Angriffsvektor entsteht.

Dieses Kapitel beschreibt zunächst gängige Technologien zum Bau von Microservices. Anschließend werden Möglichkeiten betrachtet, koplexe Systeme auf Microservice-Basis zu ermöglichen. Die dabei entstehenden Herausforderungen werden hervorgehoben. Mögliche Ansätze zum Schutz eines Microservice-basierten Systems werden diskutiert.

2.1 Technologien zum Bau von Microservices

Ein Microservice sollte replizierbar (und somit skalierbar), sowie einfach zu deployen und sicher sein (Newman, 2015; Richardson, o. J.). Um Skalierbarkeit und einfaches Deployment zu erreichen, bietet es sich wiederum an Microservices in ähnlichen Ausführungsumgebungen laufen zu lassen. Das jedoch steht im Wiederspruch zu dem Konzept der Service-Unabhängigkeit und Autonomie.

Virtuelle Maschinen Mit herkömmlichen Virtualisierungstechnologien, wie etwa VMWare lässt sich die gewünschte Abstraktion schaffen. Services werden als Images für die virtuelle Maschine (VM) spezifiziert und können somit repliziert und portiert werden. Auch das Deployment des Services ist dann einfacher, weil nur noch das Service Image in der entsprechenden virtuellen Maschine gestartet werden muss.

Wenn mehrere Microservices den gleichen Host oder die gleiche VM teilen, sind sie nicht unabhängig (Richardson, o. J.). Bei böswilliger Manipulation oder Infiltrierung eines Services könnten auch andere Services im gleichen System beeinflusst werden. Folgt man dem Konzept so wäre es korrekt und sicher(-er) einen Service pro Host oder VM zu deployen. Das resultiert allerdings in Resourcen-Overhead der benötigt wird, um die (möglicherweise vielen) virtuellen Maschinen bzw. Hosts zu betreiben (Newman, 2015). Die Grafik 1 zeigt die Resourcen auf, die bei einem Setup mit virtuellen Maschinen benötigt werden und zeigt vergleichsweise den Resourcen-Verbrauch eines Container-basierten Setups.

Der Ansatz, ein Microservice-basiertes System mit virtuellen Maschinen zu realisieren bringt jedoch auch Sicherheitsvorteile mit sich. Sofern wirklich nur ein Service pro virtueller Maschine läuft, sind alle Services des Systems vollständig unabhängig voneinander. Somit ist das Gesamtystem weniger Anfällig für Ausfälle oder bösartige Manipulation einzelner Services.

Container Ein Container ist ein schlanker „Wrapper“ um einen Linux Prozess. Wir betrachten hier Docker (`https://www.docker.com`) als Container Engine, es sei aber noch LXC (Linux Containers) erwähnt.

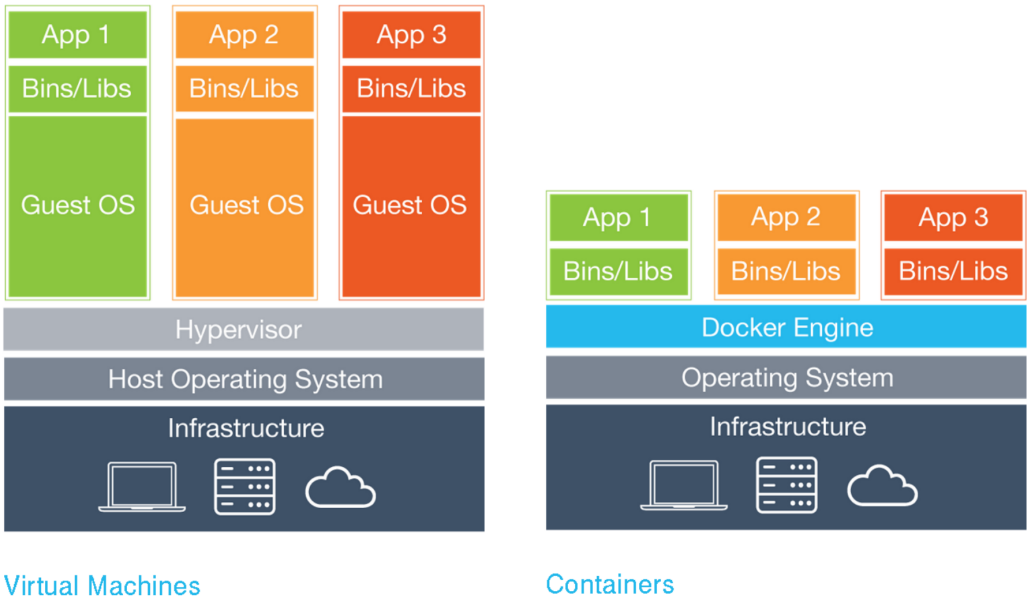

Abb. 1. Vergleich von standard Virtualisierung und Docker Conainern, entnommen von `http://www.docker.com/what-docker`

Docker und Docker-Container bieten ähnliche Funktionen wie eine virtuelle Maschine und zugehörige VM Images. Allerdings sind Container um ein Vielfaches leichtgewichtiger als VMs. Während eine VM ein Gast-Betriebssystem und eigene Libs enthält, enthalten Container nur eine Anwendung und deren benötigte Abhängigkeiten. Container teilen sich den Kernel des ausführenden Host-Betriebssystems und laufen als einzelner Prozess im Userland (Newman, 2015). Die Grafik 1 zeigt deutlich, auf welche Weise Container leichtgewichtiger sind als virtuelle Maschinen.

Zusätzlich werden Container durch Container Images spezifiert – genauso wie bei VMs – was die Vervielfältigung ungeheuer einfach macht. Allerdings ist die Modifikation von Dockerimages einfacher als das Ändern eines VM Images. Docker ermöglicht mithilfe eines sogenannten *Dockerfile*s sehr einfache Manipulation von Images.

Auch Docker bietet hervorragende Sicherheitsaspekte. Durch die intelligente Verwendung von *Namespaces*, *Network Interfaces* und *Cgroups* werden in Containern laufende Prozesse vollständig von anderen Prozessen des ausführenden Systems isoliert (Newman, 2015; Marmol, Jnagal & Hockin, 2015). Laufende Docker Container haben einen eigenen Namespace. Für Prozesse innerhalb des Containers sind jegliche andere Prozesse nicht sichtbar und nicht beeinflussbar – ob auf dem ausführenden Host-Betriebssystem oder in anderen Containern. Jeder Container hat einen eigenen Network-Stack, wodurch jeder Container nur Einfluss auf seinen eigenen Traffic nehmen kann. Zuletzt wird der Resourcenverbrauch reguliert durch Control Groups, ebenfalls ein Linux Kernel Feature. Es wird garantiert, dass ein Container genügend Resourcen bekommt, um zu arbeiten; es kann auch eingeschränkt werden, dass ein Container nicht zu viele Resourcen des Host-Betriebssystems verbraucht[1].

Die Charakteristika von Docker-Containern legen nahe, wie gut diese Technologie geeignet ist um die Anforderungen abzudecken, die eine Microservice-Architektur mit sich bringt. Durch das Design von Docker sind in Containern laufende Prozesse vollständig unabhängig voneinander. Services lassen sich in Form von Images einfach replizieren und portieren. Das ausführende System ist unwichtig, solange es ein Linux System ist und Docker installiert ist. Zum Thema Microservices, die mit Hilfe von Docker realisiert werden finden sich viele aktuelle Softwareprojekte.

2.2 Managing Microservices / „Orchestration“

Es gibt verschiedene Möglichkeiten, Microservices mit einander interagieren zu lassen um daraus ein komplexeres System entstehen zu lassen. Generell gilt für Microservices, dass sie *klein*, auf *eine Aufgabe* spezialisiert und *autonom* sind (Newman, 2015). Das bedeutet insbesondere auch, dass Services eines

[1] Detaillierte Informationen in der Docker-Dokumentation: https://docs.docker.com/engine/articles/security/

Systems einander meist nicht kennen. Das gängigste Problem in Microservice Architekturen ist die *Service Discovery*. Um miteinander zu kommunizieren, müssen die Services sich zunächst in irgendeiner Form bekannt werden. Außerdem müssen Healthchecks vorhanden sein, um das Ausfallen einzelner Services beobachten zu können. Microservice Systeme, die produktiv genutzt werden sollen müssen auch Aufgaben wie Skalierung, Loadbalancing und natürlich Sicherheit im Sinne von *Security* übernehmen.

Es gibt viel Bewegung im Feld um Microservices und Docker. Viele großangelegte (Opensource-)Projekte, getrieben durch führende Software-Konzerne[2] nehmen maßgeblich Einfluss auf die Softwareentwicklung in diesem Gebiet.

Im Folgenden geben wir einen Kurzüberblick über Google Kubernetes.

3 Kommunikation und Sicherheit

4 Schutzziele

5 Zusammenfassung

Literatur

Marmol, V., Jnagal, R. & Hockin, T. (2015, Feb). Networking in containers and container clusters.

Newman, S. (2015). *Building microservices.* O'Reilly Media, Incorporated. Zugriff auf `https://books.google.de/books?id=1uUDoQEACAAJ`

Richardson, C. (o. J.). *A pattern language for microservices.* `http://microservices.io/patterns/`.

[2] `http://kubernetes.io`, `http://mesos.apache.org`, `http://azure.microsoft.com`, `https://aws.amazon.com/de/ecs/`