Microservices und Sicherheit

Aufsatz zum Seminar Komponenten, Agenten und Workflows in verteilten Systemen

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

12. Dezember 2015

Zusammenfassung

Mit der stetig zunehmenden Diversifikation von Informationssystemen im Internet ist ein Bedarf für eine verteilte und skalierbare Architektur aufgekommen. So entstand etwa der neue Trend zur Microservice-Infrastruktur. Allerdings birgt diese Risiken in Bezug auf Sicherheit, da sowohl Architektur als auch Kommunikation eines Microservice-Netzwerks Sicherheitsprobleme aufwerfen können. In diesem Aufsatz gehen wir daher auf diese Probleme ein und zeigen für drei Schutzziele Lösungsmöglichkeiten auf. Es wird gezeigt, dass es bereits Möglichkeiten und neuartige Technologie gibt um sich vor Angriffen zu schützen.

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. Dabei dürfen Services sich während des Ausführungszeit nicht gegenseitig beeinflussen. 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 und geht auf Sicherheit im Sinne von Isolation ein. Anschließend werden Möglichkeiten betrachtet, koplexe Systeme auf Microservice-Basis zu ermöglichen. Die dabei entstehenden architekturellen Herausforderungen sowie auch mögliche Sicherheitsprobleme 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 Ausführungs-

umgebungen laufen zu lassen. Das jedoch steht im Wiederspruch zu dem Konzept der Service-Unabhängigkeit und Autonomie.

2.1.1 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). Abbildung 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.

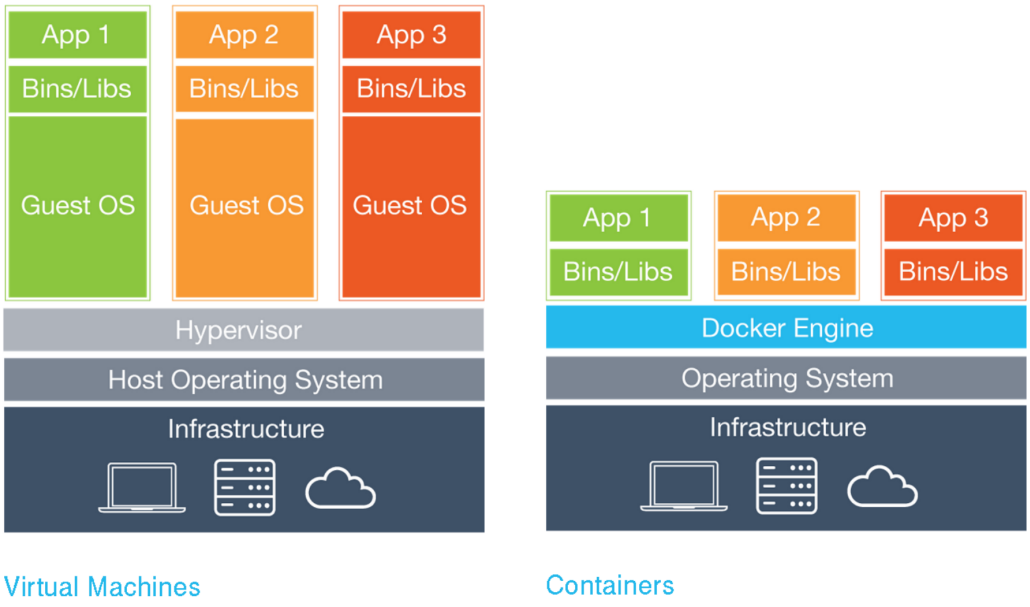

Abbildung 1: Vergleich von standard Virtualisierung und Docker Conainern, entnommen von http://www.docker.com/what-docker

2.1.2 Container

Ein Container ist ein schlanker „Wrapper“ um einen Linux Prozess. Wir betrachten hier *Docker*[1] als Container Engine, es sei aber noch LXC (Linux Containers) erwähnt.

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 Bibliotheken 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 spezifiziert – 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. Docker bietet hervorragende Sicherheit im Sinne von Isolation. 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[2].

Dockers Charakteristika 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 / Orchestrierung

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 Systems einander meist nicht kennen. Im Folgenden werden einige Herausforderungen diskutiert, die beim Bau eines Microservice-basierten Systems entstehen. Es werden Software Lösungen und Frameworks vorgestellt, die viele der Problemstellungen intelligent lösen. Sicherheitskritische Bereiche werden aufgezeigt und Maßnahmen zu deren Schutz erläutert. Anschließend wird das opensource Project

[1] https://www.docker.com

[2] Detaillierte Informationen zu Docker Security: https://docs.docker.com/engine/articles/security/

Kubernetes von Google vorgestellt.

2.2.1 Systemanforderungen

Bei einem System aus autonomen Microservices muss auch das Deployment der Services autonom sein. Services zu starten/stoppen, zu verändern oder zu verfielfätigen sollte vom Gesamtsystem ermöglicht werden, während andere Services davon nicht beeinflusst werden. Das Gesamtsystem muss den Ausfall einzelner Services verkraften. Für Entwickler oder Administratoren muss ersichtlich sein, welche Komponenten des Systems korrekt oder inkorrekt arbeiten (Newman, 2015). Aus diesen Anforderungen und aus der Spezifikation von Microservices (siehe Kapitel 1) ergeben sich einige problematische Fragestellungen, die gelöst werden müssen.

Wenn Services autonom agieren und einzeln an- und abschaltbar sind, wie können sie dann miteinander kommunizieren? Microservices haben in der Regel keine feste IP Adresse, unter der sie immer erreichbar sind. Dieses Problem wird als *Service Discovery* bezeichnet. Services müssen einander im System finden, um Nachrichten austauschen zu können. Technisch lässt sich das Problem beispielsweise durch den Einsatz einer DNS (Domain-Name-System) lösen. Auch *etcd*[3] ließe sich dafür verwenden.

Wie lässt sich das Verhalten einzelner Services nachvollziehen? Dazu kann zentralisiertes Logging verwendet werden; alle Microservices schicken ihre Logs zu einem Service, der allein dafür zuständig ist, Lognachrichten zu akkumulieren. Ein sehr häufiges Setup ist der sogenannte *ELK-Stack*[4]. Dabei wird *Logstash* als zentralisierter Log-Akkumulator verwendet, die Logs werden anschließend in einer *Elasticsearch* gespeichert und mit *Kibana* visualisiert. Ebenfalls populär, insbesondere im Container Umfeld sind *InfluxDB*[5] und *Grafana*[6] womit sich Zeitbasiert Metriken gut visualisieren lassen.

Das Sichten von Lognachrichten erfordert in der Regel menschliche Interaktion und nur mit Logging lässt sich nicht gezielt abfragen, ob ein Service überhaupt arbeitet oder nicht. Daher werden *Healthchecks* für jeden einzelnen Microservice benötigt. Ein Healthcheck gibt Auskunft darüber, ob ein Service läuft und arbeitsfähig ist. Für jeden Microservice kann ein Healthcheck unterschiedlich sein, beispielsweise ein einfacher Ping für einen Rest-Service oder ein Select-Statement für eine Datenbank.

Wie lässt sich das System skalieren? Da die Microservices des Systems einfach replizierbar und portierbar sind, kann bei hoher Last auf einem Service einfach weitere Services hinzugefügt werden, die die selbe Aufgabe erfüllen. Man spricht von horizontaler Skalierung. Damit dies jedoch fehlerfrei funktionieren kann, muss Service Discovery (s.o) und auch Loadbalancing im System vorhanden sein.

Wenn mehrere Services die gleiche Aufgabe übernehmen, muss das Gesamtsystem entscheiden, welcher Service welche Aufgaben erhält. Das nennt man Loadbalancing. Gleiche Arbeit wird von replizierten Services übernommen, aber wie kann entschieden werden, welcher Service gerade bereit für neue Last ist? Hier können simple Verfahren wie etwa Round-Robin eingesetzt werden; häufig werden jedoch erprobte Systeme wie etwa

[3] https://coreos.com/etcd/docs/latest/
[4] https://www.elastic.co/webinars/introduction-elk-stack
[5] https://influxdb.com/
[6] http://grafana.org/

HAProxy[7] oder *Nginx*[8] verwendet.

Zuletzt bleibt noch die Frage nach Sicherheit. Zum einen muss die Kommunikation zwischen den Services sicher sein, Nachrichten dürfen nicht manipuliert werden. Klassische Angriffsszenarien wie beipielsweise ein Man-In-The-Middle Angriff zwischen Services sollte nicht möglich sein. Zum anderen muss auch das gesamte System gesichert sein. Die Ausführung, das Deployment oder Balancing von Services darf für Unbefugte nicht manipulierbar sein.

Im Folgenden stellen wir kurz das Open-Source-Projekt *Kubernetes* vor. Es wird diskutiert, wie das System in seiner Gesamtheit vor unbefugtem Zugriff geschützt wird. Auf die „Servicekommunikation und Sicherheit“ gehen wir dann im darauf folgenden Kapitel detaillierter ein.

2.2.2 Kubernetes

Es gibt viel Bewegung im Feld um Microservices und Docker. Viele großangelegte Projekte, getrieben durch führende Software-Konzerne[9] nehmen maßgeblich Einfluss auf die Softwareentwicklung in diesem Gebiet. Kubernetes, eines dieser Projekte ist eine open-source Software, gestartet von Google und wird von einer wachsenden Community weiter vorangebracht.

> *„Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts.“(*Kubernetes Documentation*, o. J.)*

Kubernetes bietet eine Menge von grundlegenden Abstraktionen und Konzepten, die bei der Entwicklung von Microservice-Systemen hilfreich sind. Kubernetes lässt sich ebenfalls hervorragend dafür verwenden, *PaaS*-Systeme (Platform-as-a-Service) zu realisieren; diesen Anwendungsfall wollen wir hier nicht näher betrachten.

Kubernetes erstreckt sich als transparent verteiltes System über eine Gruppe von physikalischen Maschinen (*Nodes*). Sämtliche Resourcen, so wie etwa CPU und Memory werden transparent für das Gesamtsystem nutzbar. Die nun folgenden Begriffserklärungen stützen sich alle auf die Kubernetes Dokumentation, siehe (*Kubernetes Documentation*, o. J.).

Container werden in Kubernetes als Gruppen bestehend aus mindestens einem Container betrachtet; diese Abstraktion nennt sich *Pod*. Ein Pod abstrahiert einen logischen „Host“ für Container. Alle Container innerhalb eines Pods laufen auf der gleichen Node und teilen das gleiche Schicksal (*Kubernetes Documentation*, o. J.). Pods werden niemals bewegt. Wenn eine Neu-Lokation stattfinden muss, etwa weil eine Node aus dem Gesamtsystem ausscheidet, werden alle Pods (und darin lebende Container) die auf dieser Node liefen zerstört. Neue Pods mit Containern, die die gleiche Aufgabe übernehmen wie die zerstörten werden auf einer neuen, noch verfügbaren Node gestartet.

Pods können repliziert werden. In einem einfachen Beispiel enthält ein Pod genau einen Container, der eine einfache Aufgabe erledigt (etwa Requests entegennehmen). Sollte nun der eine Pod nicht mehr in der Lage sein, alle Requests in zufriedenstellender Zeit

[7] http://www.haproxy.org/

[8] http://nginx.org/

[9] Kubernetes//Google: http://kubernetes.io, Mesos//Apache: http://mesos.apache.org, Azure//Microsoft: http://azure.microsoft.com, ECS//Amazon: https://aws.amazon.com/de/ecs/

zu beantworten, so wird es nötig diesen Dienst zu beschleunigen. In Kubernetes kann dazu mithilfe eines *Replication Controller*s spezifiziert werden, wie viele Pods mit diesem Dienst gestartet werden sollen. So müssten z.B. zwei Dienste mit der gleichen Aufgabe dann exakt halb so viel arbeiten wie nur ein Dienst. Replication Controller bilden für den Microservice-Einsatz-Kontext das Konzept der horizontalen Skalierbarkeit ab. Replication Controller bieten darüber hinaus aber noch mehr. Sie garantieren, dass von einem Pod zu jeder Zeit exakt so viele Replica existieren, wie spezifiziert. Wenn es zu viele oder zu wenig gibt, werden neue gestartet oder überflüssige Pods terminiert. Dies geschieht völlig dynamisch. Bei einer Spezifikationsänderung werden nur so viele hinzugefügt bzw. entfernt wie nötig. Es wird garantiert, dass mindestens immer ein Dienst läuft. *„Think of it similarly to a process supervisor, only it supervises multiple pods across multiple nodes instead of individual processes on a single node“* (*Kubernetes Documentation*, o. J.).

Da Pods jeweils eine eigene IP im Kubernetes System haben, müssen sie entsprechend einzeln angesprochen werden. Das wird jedoch durch die Ersetzbarkeit und dynamische Erzeugung und Termination erschwert, weil neue Pods neue IPs bekommen. Wie findet man eine Gruppe von replizierten Pods und wie spricht man sie an? Dafür gibt es in Kubernetes das Konzept des *Service*. Ein Service in Kubernetes kapselt eine Gruppe von Pod-Replica und bietet ein festes Interface, um die Pod-Gruppe anzusprechen. Der Service übernimmt außerdem das Loadbalancing zwischen den einzelnen Pods.

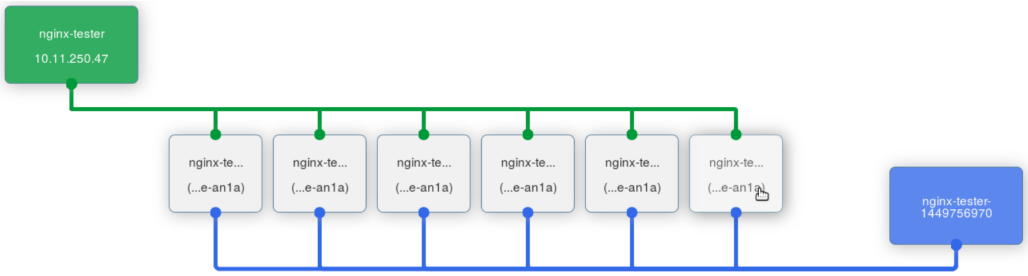

Abbildung 2: 6 replizierte Pods (weiß), 1 Service (grün), 1 Replication Controller (blau)

Abbildung[10] 2 zeigt das Zusammenspiel von Pods, Services und Replication Controllern. Alles beginnt mit dem Container *„nginx-tester“*, der eine nginx beherbergt, die wiederum http-Requests entgegen nimmt. Dieser Container wird durch einen Pod gekapselt. Damit bietet der Pod, ansprechbar unter einer kurzlebigen IP diesen Dienst an. Der Pod wiederum wurde sechs mal repliziert, was durch den blau gefärbten Replication Controller dargestellt wird. Alle sechs Pods bieten unter sechs verschiedenen IP Adressen nun den Dienst „nginx-tester“ an. Um diese Gruppe einheitlich ansprechen zu können, gibt es einen Service – dargestellt in grün. Der Service hat eine feste IP im Kubernetes System, sichtbar in der Grafik unterhalb des Service Namens in der grünen Box. Möchte man diesen Service ansprechen, so erreicht man ihn unter seiner festen IP. Der Service leitet den Request an einen der sechs möglichen „nginx-tester“ Pods weiter. Werden Pods gelöscht / erzeugt, so wird der umgebende Service dies transparent, also ohne dass es einem Nutzer auffallen würde, handhaben. Dies ist angedeutet in Abbildung 3. Der dortige Service wird ankommende Requests transparent auf einen der vier verfügbaren Pods (weiß) balancen, zwei Pods sind gerade nicht ansprechbar (gelb).

[10]Grafik erzeugt mit k8s-visualizer, https://github.com/0ortmann/k8s-visualizer

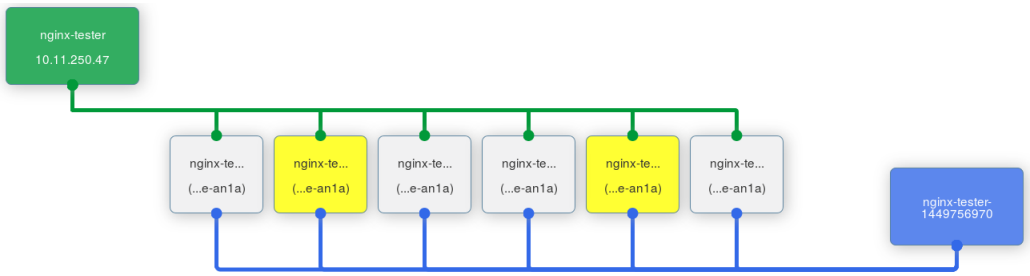

Abbildung 3: 6 replizierte Pods, davon 4 laufend (weiß) und zwei startend (gelb), 1 Service (grün), 1 Replication Controller (blau)

Durch den Einsatz eines Systems wie Kubernetes werden viele (nicht alle) sicherheitskritische Fragen für den Anwender gelöst. Kubernetes baut für die interne Resourcen-Koordination eigene Subnetze auf; es werden interne IP Adressen vergeben, die nach außen hin nicht existieren. Es ist unmöglich einen Man-in-the-middle Angriff zu Praktizieren, solange man außerhalb des Subnetzes ist. Erst wenn ein potenzieller Angreifer innerhalb des Gesamtsystems Zugriff auf die einzelnen Subnetze hat, ist Abhören oder Modifizieren von Traffic möglich.

Das böswillige Manipulieren von Anwendungen ist noch schwieriger und nur möglich, wenn der umgebende Container einer Anwendung manipuliert werden kann. Um einen Container zu manipulieren muss zunächst der entsprechende Pod lokalisiert und inflitriert werden; wenn ein Angreifer die Container-Ebene erreicht hat, müssen immer noch die Isolationsmechanismen von Docker überwunden werden.

Die Konfiguration und Manipulation von Kubernetes Resourcen ist geschützt durch Authentifikation via OAuth und SSH. Ferner ist es zu empfehlen, den Zugriff auf Schnittstellen zu Kubernetes durch eine Firewall zu schützen, sodass nur vertrauenswürdige Personen am System arbeiten dürfen. In einem Produktionssystem könnte man beispielsweise den Zugriff auf ein Kubernetes Cluster mit einer Firewall schützen, die nur die IP Adressen des Entwickler-Büros durchlässt. Jeder Entwickler wiederum muss sich per OAuth oder SSH authentizieren.

2.3 Zusammenfassung

In Kapitel 2 wurden Technologien präsentiert, um Microservices umzusetzen und zu kapseln. Dabei wurden virtuelle Maschinen und Container vorgestellt und verglichen. Beide Konzepte bieten Mechanismen zur Isolation von Anwendungen. Weiterhin wurde eine Anforderungsanalyse an ein Microservice-basiertes System aufgestellt. Aufgrund der Aktualität wurde Kubernetes als mögliche Abstraktion eines solchen Systems ausgewählt und näher erläutert. Konzepte zur Replizierung, Skalierung und Balancing von und zwischen Containern wurden diskutiert. Sicherheitsfragen wurden formuliert und beantwortet. Dabei lag der Fokus im Speziellen auf dem Schutz des Gesamtsystems.

Im folgenden Kapitel betrachten wir ein verallgemeinertes Microservice-System und heben die Spezifikation und die Schwachpunkte der Kommunikationswege zwischen Microservices in der Vordergrund.

3 Kommunikation und Sicherheit

Wie in Kapitel 2 beschrieben, besteht ein Microservice-basiertes System aus einer Menge von mindestens einem physikalischen Host und beliebig vielen Microservices. Um überhaupt von einem System sprechen zu können, müssen die einzelnen Komponenten noch vernetzt werden. Wir benutzen im Folgenden den Ausdurck *Service-to-Service-Kommunikation* – StS-Kommunikation. Services kommunizieren miteinander, tauschen Nachrichten aus, fordern Dinge an die von einem anderen Service berechnet werden müssen etc (Newman, 2015). Diese Kommunikation kann angegriffen werden, insbesondere wenn die Kommunikation über unsichere Kanäle stattfindet wie etwa das World-Wide-Web.

Im Folgenden wollen wir verschiedene Möglichkeiten von Sts-Kommunikation erläutern, Vor- und Nachteile diskutieren und auf Sicherheitsaspekte eingehen.

3.1 Kommunikationswege

Kommunikation zwischen Microservices findet fast immer event basiert statt. Die Eventverarbeitung kann synchron oder asynchron stattfinden (Newman, 2015). Dadurch, dass Microservices immer genau eine logische Aufgabe erfüllen, werden sie meist von extern dazu angestoßen etwas zu verarbeiten (*receive-event*). Wenn ein Microservice etwas verarbeitet hat, dann gibt er entweder das Ergebnis zurück oder verschickt eine Nachricht, dass nun ein Ergebnis vorliegt (*send-event*). „Applications built from microservices aim to be as decoupled and as cohesive as possible [...] receiving a request, applying logic as appropriate and producing a response“ (Fowler & Lewis, 2014). Wir werden dies an den zwei populärsten Verfahren zur Sts-Kommunikation erläutern: *Message Queues* und *REST* (Fowler & Lewis, 2014; Newman, 2015). Dabei werden Aufgabengebiete herausgestellt, für die die jeweilge Kommunikationsart im Mircoservice Kontext am geeignesten erscheinen.

3.1.1 REST

REST ist die Kurzschreibweise für „Representational State Transfer“, eine zustandslose, protokollbasierte Frage-Antwort-Kommunikationsform. Darüber hinaus beschreibt REST umgangssprachlich auch einen Architektur-Ansatz, um verteilte Anwendungen zu entwerfen (Webber, Parastatidis & Robinson, 2010). Für detaillierte Informationen zum Thema REST empfehlen wir *REST in Practice, O'Reilly.*

Im Microservice Kontext lässt sich REST hervorragend für Sts-Kommunikation verwenden (Fowler & Lewis, 2014). Dabei wird in der Regel das HTTP Protokoll verwendet. Um über REST Schnittstellen kommunizieren zu können, müssen die jeweiligen Microservices in der Lage sein, einander anzusprechen. Es müssen also die IP-Adressen der einzelnen Komponenten bekannt sein, etwa durch Service-Discovery (siehe 2.2.1).

HTTP Kommunikation kann belauscht und unterbrochen werden, außerdem ist sie anfällig für Man-in-the-Middle Angriffe und vergleichbare Szenarien. Um ein wenig mehr Sicherheit zu schaffen, kann zur Sts-Kommunikation auch HTTP/s verwendet werden. HTTP/s für sich allein bringt nach (Newman, 2015) nur einen geringen Vorteil. Zwar ist dann der Traffic verschlüsselt, aber in der Regel sind die APIs der Services frei ansprechbar. Daher wird zusätlich empfohlen, dass die einzelnen Microservices sich bei anderen authentizieren. Ein solider und einfacher Ansatz wäre etwa Basic-Auth über HTTP/s.

Die Übetragung von Username und Passwort erfolgt verschlüsselt; es kann sichergestellt werden, dass nur berechtigte Services anderen Services Nachrichten schicken dürfen. Außerdem kann zu späteren Zeitpunkten anhand von Logs und Usernamen nachgewiesen werden, wann und zwischen welchen Services eine Kommunikation stattgefunden hat. Im folgenden Kapitel werden wir Verfahren zum Schutz und zur Authentizierung detaillierter behandeln.

REST eignet sich für Microservice-basierte Systeme insbesondere, wenn der Bedarf nach synchroner Kommunikation gegeben ist. Nachdem ein Service mit einem Request angesprochen wurde, erwartet der Anfragende eine entsprechende Antwort. Als Übertragungsformat kommen meist leichtgewichtige und ausdrucksstarke Text-Formate zum Einsatz wie JSON und XML. Von Vorteil ist auch, dass es eine breite Unterstüzung in Form von Frameworks und HTTP Clients gibt, sodass meist nur minimaler Overhead entsteht, um einer Software-Komponente zu ermöglichen via REST zu kommunizieren.

Da die Kommunikation via REST synchron ist, ergeben sich Probleme wenn ein Service neu gestartet, gelöscht oder ersetzt wird. Bestehende Verbindungen werden unterbrochen und erwartete Anworten bleiben aus. Der Anfrangende Service muss selbst eine Fehlerbehandlung implementieren, wenn benötigte Antworten nicht kommen.

3.1.2 Message Queues

Das Konzept einer Queue werden wir nicht näher erläutern. Einen guten Überblick liefert (Hohpe & Woolf, 2003). Kommunikation über Queues ist immer asynchron. Wenn etwas in einer Queue abgelegt wurde, verweilt es dort bis es wieder herausgenommen wird. Sender und Emfänger müssen einander nicht kennen. Es kann beliebig viele Sender und beliebig viele Empfänger geben. Queues lassen sich auch als Puffer verwenden.

Im Microservice Kontext lassen sich Queues insbesondere gut dafür verwenden, asynchrone, horizontal skalierbare Worker-Chains zu realisieren (Newman, 2015). Dabei wird eine große logische Verarbeitungsaufgabe in viele kleine Einzelschritte zerlegt. Ein Microservice bildet die Arbeitsoperation eines Teilschrittes ab. Dabei lauscht ein Microservice auf einer Queue, sobald etwas verfügbar ist wird es abgearbeitet und das Ergebnis auf eine andere Queue geschrieben. Wenn ein Verarbeitungsschritt absehbar mehr Zeit kostet als andere, so können einfach weitere Microservices gestartet werden, die an der gleichen Queue lauschen, sodass dort mehr Durchsatz geschaffen wird.

Ein gewaltiger Vorteil von Queueing ist, dass diese Kommunikationsarchitektur das Problem der Service-Discovery etwas abschwächt. Worker-Services müssen nur die entsprechende Queue kennen, auf der sie lauschen bzw. schreiben. Services müssen sich gegenseitig nicht kennen, um zu kommunizieren. So wird das Skalieren, Stoppen oder Ersetzen von Services völlig transparent für das Gesamtsystem.

Probelematisch ist allerdings hier die Integrität von Nachrichten. Services sollten sich auf ein dediziertes Format einigen und jede angenommene Nachricht zunächst validieren und verifizieren. Da Sender und Empfänger sich nicht kennen, ist es schwierig den Absender eindeutig festzustellen. Allerdings kann durch ein geteiltes Nachrichtenformat, etwa durch ein JSON-Schema oder Ähnliches sichergestellt werden, dass eingehende Nachrichten überhaupt verarbeitet werden können.

Queues implementiert man in der Regel nicht selbst, sondern greift auf bestehende Lösungen zurück. Es gibt zwei sehr aktuelle Message Queues, die für Microservice-Systeme

verwendet werden. Das ist zum einen *RabbitMQ*[11], zum anderen *ZeroMQ*[12]. Beim Einsatz jedweder Lösung sollte man stets darauf achten, dass Nachrichten vor Modifikation durch unbefugte geschützt sind. So könnte etwa bei jeder Nachricht in der Queue noch ein entsprechender Hash mitgeschickt werden, der zusätzlich zum Nachrichtenformat zur Validierung der Nachricht verwendet wird.

3.2 Zusammenfassung

In diesem Kapitel haben wir verschiedene Möglichkeiten zur Sts-Kommunikation aufgezeigt. Dabei wurden insbesondere REST und Message Queues betrachtet. Mögliche Schwachstellen bei der Sts-Kommunikation wurden herausgestellt; der Bedarf nach applikationsseitig implementiertem Schutz wurde deutlich gemacht. Aufgrund der Heterogenität eines Microservice-basierten Systems muss jede Komponente selbst in der Lage sein, ankommende Nachrichten zu validieren und zu verifizieren. Im Folgenden werden wir Schutzziele definieren und auf Microservice-Systeme sowie auf Kommunikation übertragen. Es werden einige konkrete Verfahren und Bibliotheken vorgestellt, mit denen sich diese Schutzziele umsetzen lassen.

4 Schutzziele

Nachdem wir bisher auf die Architektur und die Kommunikation von Microservices eingegangen sind, betrachten wir im folgenden Abschnitt Angriffspunkte einer solchen Infrastruktur. Explizit werden drei Schutzziele der Informationssicherheit betrachtet, welche eine besondere Relevanz in verteilten Systemen spielen. Zunächst gehen wir auf das der Vertraulichkeit in Unterabschnitt 4.1 ein. Weitergehend setzt sich Unterabschnitt 4.2 mit der Verfügbarkeit und zuletzt Unterabschnitt 4.3 mit der Integrität von Informationen innerhalb eines Microservice-Netzwerks auseinander.

4.1 Vertraulichkeit

„Informationsvertraulichkeit ist bei einem IT-System gewährleistet, wenn die darin enthaltenen Informationen nur Befugten zugänglich sind. Dies bedeutet, dass die sicherheitsrelevanten Elemente nur einem definierten Personenkreis bekannt werden. Dazu sind Maßnahmen zur Festlegung sowie zur Kontrolle zulässiger Informationsflüsse zwischen den Subjekten des Systems nötig (Zugriffsschutz und Zugriffsrechte), sodass ausgeschlossen werden kann, dass Informationen zu unautorisierten Subjekten ‚durchsickern'." (Bedner & Ackermann, 2010)

4.1.1 Handshake-basierte Authentifikation

4.1.2 Token-basierte Authentifikation

4.2 Verfügbarkeit

„Die Verfügbarkeit betrifft sowohl informationstechnische Systeme als auch die darin verarbeiteten Daten und bedeutet, dass die Systeme jederzeit betriebsbereit sind und auf die Daten wie vorgesehen zugegriffen werden kann. Zum einen muss die Datenverarbeitung

[11] https://www.rabbitmq.com/
[12] http://zeromq.org/

inhaltlich korrekt sein und zum anderen müssen alle Informationen und Daten zeitgerecht zur Verfügung stehen und ordnungsgemäß verarbeitet werden.“ (Bedner & Ackermann, 2010)

4.2.1 Redundanz und Lastverteilung

4.2.2 Client-seitige Lastverteilung

4.2.3 Kontinuierliches und unabhängiges Deployment

4.3 Integrität

„Integrität oder Unversehrtheit bedeutet zweierlei, nämlich die Vollständigkeit und Korrektheit der Daten (Datenintegrität) und die korrekte Funktionsweise des Systems (Systemintegrität). Vollständig bedeutet, dass alle Teile der Information verfügbar sind. Korrekt sind Daten, wenn sie den bezeichneten Sachverhalt unverfälscht wiedergeben. Die Integrität bedeutet, dass Daten im Laufe der Verarbeitung oder Übertragung mittels des Systems nicht beschädigt oder durch Nichtberechtigte unbefugt verändert werden können. Beschädigungs- oder Veränderungsmöglichkeiten sind das Ersetzen, Einfügen und Löschen von Daten oder Teilen davon.“ (Bedner & Ackermann, 2010)

4.3.1 Verschlüsselungsverfahren

5 Zusammenfassung

Literatur

Bedner, M. & Ackermann, T. (2010). Schutzziele der IT-Sicherheit. *Datenschutz und Datensicherheit*, *5*, 323 - 328.

Fowler, M. & Lewis, J. (2014, März). Microservices. A definition of this new architectural term.

Hohpe, G. & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kubernetes documentation. (o. J.). http://kubernetes.io/v1.1/docs/.

Marmol, V., Jnagal, R. & Hockin, T. (2015, Feb). Networking in containers and container clusters. Netdev 0.1, Ottawa, Canada.

Newman, S. (2015). *Building microservices.* O'Reilly Media, Incorporated.

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

Webber, J., Parastatidis, S. & Robinson, I. (2010). *Rest in practice: Hypermedia and systems architecture.* O'Reilly Media.