Prof. Dr. René Peinl
(rene.peinl@iisys.de)
ist Hochschulprofessor an der Hochschule Hof und Forschungsgruppenleiter am angeschlossenen Institut für Informationssysteme (iisys). Er unterrichtet Web-Architekturen und -Technologien und forscht zu Systemintegration mit Open-Source-Werkzeugen in den Umfeldern Team-Zusammenarbeit und Internet der Dinge.

Überblick über Docker-Cluster-Technologien – Tools und Trends

Der Hype um das Container-Werkzeug Docker ist auch drei Jahre nach seiner ersten Veröffentlichung ungebrochen. Docker adressiert sowohl Entwickler, die einen einfachen Weg suchen, um ihre Anwendungen zu testen und anschließend in die Produktivumgebung zu bringen, als auch Cloud-Betreiber, die durch die Verwendung von Containern statt Virtuellen Maschinen (VMs) eine höhere Packungsdichte und Performance realisieren können. Während die Handhabung für Entwickler mit einzelnen Docker-Containern recht einfach ist, setzen die „Googles" und „Netflixe" dieser Welt häufig selbstentwickelte Werkzeugketten ein, um die Komplexität in großen Umgebungen durch Automatisierung beherrschbar zu machen. Dazwischen stehen mittelständische Cloud-Anbieter oder Rechenzentren größerer Unternehmen, die sich keine Entwicklerschar für Docker-Tools leisten können oder wollen, andererseits aber die Komplexität manuell nicht mehr handhaben können. Für diese gibt es eine schier unüberschaubare Anzahl an Open-Source-Werkzeugen, die das eine oder andere Problem mildern, jedoch kaum eine umfassende Lösung aus einem Guss. Der vorliegende Artikel zeigt, welche Aufgaben zu erfüllen sind, welche Werkzeuge welche Aspekte davon abdecken und welche Kombinationen von Werkzeugen miteinander harmonieren, um eine umfassende Lösung zu erreichen.

Wozu ein Docker-Cluster?

Cloud Computing hat den Umgang mit Software nachhaltig beeinflusst. Nicht nur die Nutzung von zunehmend mehr Software aus der Cloud, auch die Art wie Rechenzentren betrieben werden hat sich stark gewandelt. Wer den Begriff private Cloud ernst nimmt, der muss dieselbe Agilität, Selbstbedienungsphilosophie und den Automatisierungsgrad erreichen, wie das bei öffentlichen Angeboten der Fall ist.

Dafür ist es bei weitem nicht ausreichend alle Anwendungen in virtuelle Maschinen zu packen und bei Bedarf eine weitere zu starten. Containerlösungen wie Docker sind angetreten, um die vorhandenen Ressourcen bestmöglich auszulasten und gleichzeitig ein möglichst reibungsfreies Nebeneinander von verschiedenen Anwendungen zu gewährleisten und wurden damit quasi über Nacht zum Liebling der Cloud-Provider.

Bei genauerer Betrachtung stellt sich jedoch heraus, dass Docker keineswegs die Lösung des Automatisierungsproblems ist, sondern den RZ-Betreiber schlagartig aus dem gut beherrschten Umfeld virtueller Maschinen (z. B. VMware und XenServer) in weitgehend unerforschtes Territorium bringt. Ein Update in einen Container einspielen? Daten der Anwendung einfach ins Dateisystem des Containers schreiben? Einen Container mal eben im laufenden Betrieb von einem Host auf einen anderen migrieren? Alles nicht vorgesehen. Man muss sich also von einigen eingeschliffenen Verhaltensweisen verabschieden und sich ganz auf die neuen Prinzipien einlassen, um sich neben erwünschten Vorteilen nicht auch eine Reihe neuer Probleme einzukaufen.

Eines dieser Probleme ist die Einschränkung der Kommunikation von Docker-Containern untereinander auf einen Host. Insbesondere wenn man die sinnvolle Möglichkeit der Verlinkung zwischen Containern benutzen will, ist das eine gewichtige Limitation. Docker Inc., das Unternehmen hinter Docker, ist sich dieser Einschränkungen natürlich bewusst und arbeitet beständig darauf hin, die Lücken zu schließen und hat mit den mittlerweile sehr brauchbaren Werkzeugen Docker *machine*, *swarm* und *compose* bereits viel bewegt [Roß15].

Auch die Mitte 2015 eingeführten offenen Schnittstellen zu Netzwerk- und Speicherlösungen (network / volume plug-ins [UEP]) haben viel Gutes bewirkt. Um aber

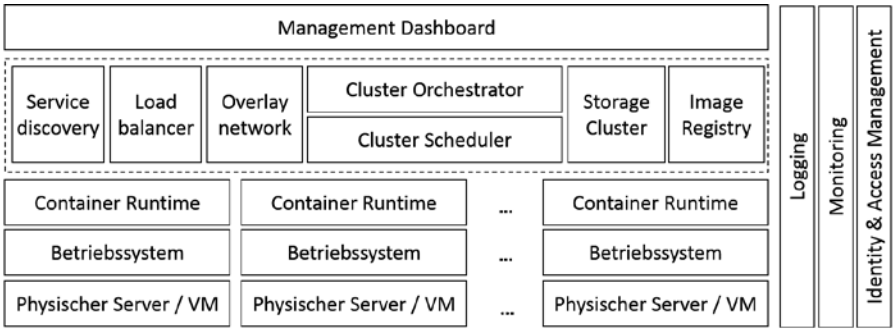

Abb.: Schematische Übersicht der Bestandteile eines Container-Clusters

die Automatisierungsansprüche einer Cloud-Umgebung erfüllen zu können, ist man immer noch auf Werkzeuge von Drittanbietern angewiesen. Nur durch ein komplexes Zusammenspiel einer ganzen Reihe von Werkzeugen erhält man einen gut verwaltbaren Cluster, der sowohl auf der Kommandozeile als auch über eine Weboberfläche Informationen über aktuelle und historische Messwerte und den Zustand seiner Anwendungen liefert. Daran ändert nach Einschätzung des Autors auch der neueste Vorstoß von Docker Inc. namens Docker Datacenter nur wenig [Sve16], da viele integrale Bestandteile so einer Clusterlösung nur als Schnittstellen vorhanden sind, nicht aber als fertige Dienste.

Cluster-Bestandteile – Pflicht

Im Prinzip kann man sich bei den nötigen Bausteinen von Container-Clustern an bestehenden IaaS- (Infrastructure as a Service) Lösungen einerseits und PaaS (Platform as a Service) andererseits orientieren (siehe Abbildung). Als Basis dient das **Betriebssystem** von einzelnen Servern, die mit einer Container-Laufzeitumgebung wie *Docker*, *LXD* (Canonical) oder *rkt* (CoreOS) ausgestattet zum Container-Host werden.

Dabei ist es aus Sicht des Containers egal, ob der Host eine physische oder eine virtuelle Maschine (VM) ist. In manchen Szenarien ist es trotz des Overheads durch VMs sinnvoll, diese zwischenzuschalten. Mit zunehmend besserer Abschottung der Container untereinander werden aber immer mehr RZ-Betreiber die Container-Engine direkt auf der physischen Hardware laufen lassen.

Obwohl im Prinzip jedes Linux-Betriebssystem in Frage kommt, spricht vieles dafür, ein auf Docker-Container spezialisiertes System wie *CoreOS*, *Project Atomic* (Red Hat) oder *Snappy Ubuntu core* (Canonical) zu verwenden. Das Vorbereiten eines neuen Servers für die Aufgabe als Container-Host kann man mit Tools wie *Terraform* oder *Docker machine* automatisieren. Im Falle von VMs kann man auch einfach entsprechende Images vorbereiten und gegebenenfalls mit Chef, Puppet oder Ansible dynamische Operationen durchführen.

Damit die Verwaltung der Container nicht mehr Host-spezifisch zu erledigen ist, benötigt man **Cluster Scheduler**. Sie sorgen dafür, dass man Container auf beliebigen Hosts des Clusters starten und stoppen kann und bieten idealerweise auch editierbare Platzierungsstrategien, sodass man z. B. zunächst Hosts verwendet, die schon aktiv sind, aber noch ausreichend freie Ressourcen bieten, bevor man noch unbenutzte Hosts belegt (bin packing), oder gezielt Container-Instanzen eines Typs auf verschiedenen Hosts, Racks oder Rechenzentren starten kann. Die verbreitetsten Vertreter dieser Gattung sind *CoreOS fleet*, *Apache Mesos* und *Docker swarm*.

Obwohl der Cluster-Scheduler für eine vereinheitlichte Sicht auf die Container-Hosts sorgt, kümmert er sich nicht darum, dass die Container auch Host-übergreifend miteinander sprechen können. Docker war nämlich lange Zeit auf einzelne Hosts beschränkt. Erst Docker v1.9 (Nov 2015) brachte eingebautes Multi-Host-Networking basierend auf VXLAN mit, welches aus der Übernahme von Socketplane hervorgegangen ist. Docker network wurde aber noch als experimentell bezeichnet und in v1.10 (Feb 2016) stark verbessert. Sogenannte **Overlay-Netzwerke** für Docker wie *Weave* und *CoreOS flannel* gab es schon früher. Seit der Neusortierung der Netzwerkfunktionen in Docker mit libnetwork schießen sie aber wie Pilze aus dem Boden. Sie sorgen dafür, dass die Container direkt ohne Network Address Translation miteinander sprechen können.

Canonical stellte mit *Fan* ein interessantes Konzept vor, dass eine vorhersagbare IP-Adresszuweisung für neue Container vorsieht. *Project Calico* wurde von Metaswitch Networks initiiert, dann laut Ankündigung von Cisco unterstützt, um jetzt scheinbar ohne Pressemeldung unter dem Namen contiv geforkt und von Cisco alleine weiterentwickelt zu werden. *Midonet* tritt an, auch gleich das vielgescholtene SDN (Software Defined Network) von OpenStack zu ersetzen. Es ist allerdings komplex zu konfigurieren und auf Basis der frei verfügbaren Dokumentation kaum einsetzbar.

Das eigentliche Anstoßen des Containerstarts erfolgt dann über einen **Orchestrator**, der auch die Abhängigkeiten zwischen den einzelnen Containern berücksichtigt. So macht es z. B. wenig Sinn den zu einem Container gehörigen Storage Container auf einen anderen Host zu legen, da dadurch nur unnötige Netzwerkkommunikation verursacht würde. Auch die Reihenfolge des Startens von Diensten, die aufeinander aufbauen, muss der Orchestrator beachten. Bekannte Vertreter dieser Kategorie sind *Kubernetes* (Google), *Apache Marathon* und *Docker Compose*. Kubernetes ist dabei am flexibelsten und kann außer auf Fleet als Cluster Scheduler auch auf Mesos oder Docker Swarm aufsetzen.

Die Container-Images werden dabei aus einer **Image-Registry** bezogen. Dafür kann man entweder auf das öffentliche Angebot *Docker Hub* zurückgreifen, wo sich bereits tausende vorgefertigter Images finden, ein alternatives Cloud-Angebot wie von Google, Amazon oder CoreOS (*quay.io*) nutzen oder eine eigene Registry betreiben.

Ein Management-Dashboard ist bei Kubernetes und Mesos/Marathon schon integriert. Bei Docker scheint im neuen Datacenter-Angebot hinter dem Titel *Universal Control Plane* ebenfalls eine Web-Oberfläche zu stecken [Sei16]. Man kann aber zu Lösungen wie *Panamax* oder *Cockpit* (Redhat) greifen, wenn man eine alternative Weboberfläche nachrüsten will. Diese arbeiten aber Host-, nicht Cluster-spezifisch.

... und Kür

Eine der kniffligsten Aufgaben im Container-Cluster ist es, die Verbindung zwischen den einzelnen Diensten in den Containern herzustellen. Da die IP-Adressen erst beim Starten des Containers vergeben

werden, braucht man einen Mechanismus, der die notwendigen Parameter wie IP-Adresse und gegebenenfalls Port dem abhängigen Container bekannt macht. Die naheliegende Antwort auf die Frage der IP-Adressauflösung lautet DNS und tatsächlich versuchen einige Lösungen auch genau diesen Weg zu gehen.

Sowohl Kubernetes als auch Mesos/Marathon bieten einen DNS-Service an. Mit *SkyDNS* und *WeaveDNS* gibt es auch Alternativen, die man selbst betreiben kann. Wichtig ist, dass der Dienst, der die IP-Adresse vergibt, auch den DNS-Eintrag aktualisiert bzw. erstellt. Bei genauerer Betrachtung ergibt sich aber ein Problem. DNS ist relativ träge. Üblicherweise hat der Client, also der abhängige Container einen DNS-Cache, um nicht den DNS-Server zum Flaschenhals zu machen. Ändert sich die IP-Adresse des Basiscontainers, z. B. durch Migration auf einen anderen Host, dann wird der abhängige Container erst einmal weiter mit der alten IP-Adresse arbeiten, weil er nicht weiß, dass sein DNS-Cache veraltet sein könnte.

Eine andere verbreitete Lösung ist es, einen verteilten Key-Value-Store als Speicher für Konfigurationsdaten zu verwenden. Dieser wird damit dann zur **Service-Registry**. Die gängigen Implementierungen heißen *etcd* (CoreOS), *Consul* (HashiCorp) und *Zookeeper*. Diese machen aber erstmal wenig besser als ein DNS-Server, da sie nur ein Datenspeicher sind. Was man braucht, ist **Service Discovery**. Die Einträge werden entweder vom Orchestrator in die Service-Registry geschrieben (z. B. Kubernetes mit etcd) oder von Zusatztools, die auf dem Host auf neue Container reagieren, z. B. *Registrator*.

Bleibt noch der letzte Aspekt die Daten zur Anwendung zu bringen. Wenn man nicht die Applikationen anpassen kann oder will, dann greift man zu Zusatzwerkzeugen wie *confd*. Das überwacht die Service-Registry und schreibt Änderungen in eine Textdatei innerhalb der abhängigen Container, oder wie bei Docker üblich in Umgebungsvariablen, die dann von der Anwendung ausgewertet werden können. Bleibt die Frage, ob die Anwendung überhaupt im laufenden Betrieb die Konfigurationsdateien bzw. Umgebungsvariablen auswertet oder diese nur einmalig beim Start liest und dann intern im Speicher hält.

Ist Letzteres der Fall, dann sucht man eventuell doch wieder nach einer Lösung auf Netzwerkebene. Man möchte meinen, wenn man eh schon ein Overlay-Netzwerk hat, dann kann es doch nicht so schwierig sein, die IP-Adresse eines Containers beim Umzug auf einen anderen Host mitzunehmen. In der Praxis ist das aber kaum möglich, weil die meisten Netzwerke hierarchisch aufgebaut sind und der Host als Gateway fungiert. Daher muss der Container eine Adresse innerhalb des Host-spezifischen Subnetzes haben (häufig ein Class-C Netz) und diese ändert sich dann eben beim Umzug.

Ein gangbarer Weg ist es, die Container nicht direkt anzusprechen, sondern den kleinen Umweg über einen **Load Balancer** zu gehen. In der Praxis sollte man ja sowieso immer mehrere Container vom gleichen Typ haben, die sich zu einem Lastverbund (active-active) oder zumindest zur Ausfallabsicherung (active-passive) vereinen.

Insofern ist der Load Balancer kein Overhead, sondern sowieso vonnöten. Dieser kann dann seine Konfiguration von der Service-Registry beziehen und somit veränderte IP-Adressen verzögerungsfrei berücksichtigen. Gängige Produkte sind hier *HAproxy* und *nginx*, die über Addons mit der jeweiligen Service-Registry verbunden werden. Spezialisierte Alternativen sind *hipache*, *pound* und *Vulcan*, die z. T. schon nativ mit etcd, Consul und Co umgehen können.

Persistente Daten

Hat man die Netzwerkseite der Containermigration in den Griff bekommen, bleibt das unbedeutende Problem der Daten. Es lässt sich in den meisten Anwendungen nicht vermeiden, dass Daten persistiert werden müssen. Das erledigt man entweder in einer Datenbank (die ja konsequenterweise auch im Container läuft) oder im Dateisystem. Man braucht also einen effizienten Weg, um die Daten zu speichern und bei einem Umzug des Containers auch auf dem neuen Host wieder verfügbar zu machen.

Ersteres ist leicht erledigt, weil man einfach einen Ordner aus dem Host als Speicherbereich in den Container durchreichen und damit das für Speicheroperationen ineffiziente Overlay-Dateisystem des Containers umgehen kann. Eine weitere Möglichkeit sind dedizierte Resource-Container, die über Docker-Links an den Container gehängt werden und nur für das Speichern da sind.

Der große Vorteil davon hat sich dem Autor aber nie erschlossen, da es keines der Probleme löst. Die treten nämlich wieder bei der Migration auf. Der Container wird auf dem neuen Host gestartet (meist als Kopie, nicht das „Original“) und braucht dann eben auch dort die Daten. Sind die an den vorherigen Host gebunden, muss man sie zeitraubend umkopieren, was in den meisten Fällen keine akzeptable Lösung sein kann. Eine Ausnahme stellt evtluell *Flocker* dar, das durch geschicktes Ausnutzen des ZFS-Dateisystems die Zeit zum Umkopieren minimieren kann.

Es bleiben zwei Wege mit dem Problem umzugehen. Entweder die Anwendung baut selber einen Speichercluster auf und repliziert die Daten auf den neuen Knoten, bevor der alte abgeschaltet wird, sodass man den Container nicht migrieren muss (Dovecot oder ein MySQL Galera Cluster können so konfiguriert werden), oder man hat ein Host-übergreifendes Speichersystem „darunter“, das die Daten ohne Kopieroperationen auch am neuen Host wieder zur Verfügung stellen kann. *Ceph* und *GlusterFS* sind Beispiele für solche **Storage-Cluster**. Diese bekommen mittlerweile Unterstützung von Docker durch die in Version 1.7 experimentell eingeführten Volume-Plugins, eine Art Treiber mit dem man direkt aus dem Container heraus und vorbei am Host solche zentralen Speichersysteme ansprechen kann. In der aktuellen Version 1.10 kann man das auch benutzen, wenn man weniger experimentierfreudig ist.

War ClusterHQ, die Firma hinter Flocker, lange Zeit die einzige, die sich dem Thema Datenmigration in Containern angenommen hatte, so sind seit Einführung dieser Schnittstelle auch hier die Plugins zahlreich geworden. Insbesondere EMC hat sich hier engagiert und mit *REX-Ray* ein Plugin geschaffen, um die firmeneigenen Speicherlösungen ScaleIO und XtremIO an Docker anzuschließen. Das erinnert stark an Cisco, die sich im Netzwerkbereich für Docker engagieren und interessanterweise mit der neuen Lösung *contiv* nicht nur ein Network-Plugin, sondern auch ein Ceph-kompatibles Volume-Plugin anbieten. Für Ceph gibt es einige weitere Alternativen, z. B. von *Yp engineering*.

Totale Überwachung

Bei aller Automatisierung will man natürlich auch die Möglichkeit haben, zu überwachen, was passiert, um gegebenenfalls schnell eingreifen zu können. Das gilt sowohl für Ressourcenverbrauch als auch für Fehlermeldungen, die in irgendwel-

Kategorie\Stack	Tectonic	Docker DC	SCHub	Mantl	Nomad
Orchestration	Kubernetes	Compose	Marathon	Marathon	Nomad
Scheduler	Fleet	Swarm	Mesos	Mesos	Nomad
Service discovery	Etcd	-	Consul Registrator	Consul	Consul
Overlay Netz	Flannel	Libnetwork	Weave	Contiv	?
Speicher-Anbindung	Kubernetes	siehe [UEP]	Ceph (Yp)	Contiv	?
Load balancer	Kube-proxy	-	Nginx	HAproxy	?
Monitoring	cAdvisor Heapster (Fluentd)	-	cAdvisor Prometheus Grafana	collectD ELK Stack	?
Management	Kubernetes Dashboard	Universal control plane	Eigenes Dashboard	Eigenes Dashboard	Nomad

Tab. 1: Vergleich der Bestandteile unterschiedlicher Lösungsstacks

chen Logs aufschlagen. Letzteres wird auch von Docker direkt unterstützt, zumindest wenn die Applikation keine eigenen Log-Dateien, sondern einfach nach stdout schreibt. Das wird dann vom Docker-Dämon aufgeschnappt und an einen konfigurierbaren **Logging-Dienst** weitergeleitet. *Splunk*, Amazon *Cloud Watch*, *Fluentd* und das Syslog des Hosts sind mögliche Ziele.

Wenn die Anwendung unkooperativ ist, muss man einen entsprechenden Logging Client in den Container integrieren, der dann die Änderungen in den Logdateien selbständig ohne Docker-Unterstützung an den zentralen Logging Server schickt. *Greylog 2* und *Logstash* sind gängige Varianten dafür. Letzteres stellt den mittleren Teil des bekannten ELK-Stacks dar, wobei *Elasticsearch* als Datenbank für die Log-Einträge dient und *Kibana* als webbasiertes Frontend für die Visualisierung der Auswertungen. Andere Tools verwenden statt dem für Volltextsuche optimierten Elasticsearch lieber Zeitreihendatenbanken wie *InfluxDB* oder *Prometheus*. Für die Visualisierung bietet sich dann z. B. *Grafana* an.

Die genannten Web-Frontends lassen sich auch für das **Monitoring des Ressourcenverbrauchs** einsetzen. Dafür muss man die Informationen darüber aus dem Docker-Dämon bzw. dem Host-Betriebssystem erst einmal herauskitzeln und in die Datenbank schreiben. Für Docker-Container gibt es da kaum eine Alternative zu *cAdvisor*, das von Google entwickelt wurde. Es bringt selbst ein Frontend für die Auswertung mit, hält Daten aber nur einen Tag vor.

Alternativ kann man es aber auch in InfluxDB oder Prometheus schreiben lassen. Mit *Heapster* gibt es für cAdvisor auch eine Integration in Kubernetes, sodass man direkt in der Kubernetes Web-UI die Daten auswerten kann. Hat man schon eine der Allzweck-Monitoringlösungen wie *Nagios*, *Icinga* oder *Sensu* im Einsatz, so kann diese natürlich mit entsprechenden Erweiterungen auch im Container-Umfeld weiter verwendet werden.

Zoomanager

Wer jetzt aufstöhnt und meint, er könne unmöglich selbst den ganzen Zoo von Werkzeugen installieren, konfigurieren und verwalten, dem sei ein versöhnlicher Ausblick gegeben. Das Problem wurde erkannt und es gibt zunehmend mehr Angebote, die eine vorkonfigurierte Zusammenstellung von aufeinander abgestimmten Werkzeugen in einem Gesamtpaket vereinen.

Das vielleicht prominenteste davon ist *Tectonic*, das kommerzielle Angebot von CoreOS und Google, welches die oben genannten Google-Tools wie *Kubernetes*, *cAdvisor* und *Heapster* mit den CoreOS-Tools wie *fleet*, *flannel* und *etcd* kombiniert. Dies erscheint aktuell das rundeste Gesamtpaket zu ergeben, da Google Kubernetes in Version 1.0 auch Unterstützung für zentrale Speichersysteme wie Ceph, GlusterFS, Amazon EBS und natürlich die Google-Cloud-Engine spendiert hat.

Als Alternative dazu hat Docker Inc. – wie eingangs erwähnt – kürzlich das Docker-Datacenter platziert. Es basiert auf machine, swarm und compose, lässt aber die schwierigen Fragen wie Service Discovery und Storage unbeantwortet. Scheinbar ist man der Meinung, es reiche eine Schnittstelle zu bieten. Interessanter sind die Alternativen, die auf Apache Mesos und Marathon setzen.

Die kommerzielle Variante von Mesos namens Data Center Operating System zeigt den hohen Anspruch, den man an die eigene Lösung hat. Dadurch, dass dort Docker nur ein Framework von vielen ist, das unterstützt wird, ist aber auch die Gesamtlösung aus Docker-Perspektive weniger umfassend. Dass man dennoch eine umfassende Docker-Cluster-Verwaltung auf Basis Mesos bauen kann, zeigt das Project „Social Collaboration Hub“ (SCHub) der Hochschule Hof (sc-hub.de).

Dort wurde mit ein wenig eigenem Code, Mesos, Marathon, Consul, Weave, Ceph und weiteren Tools (siehe Tabelle 1) eine Lösung geschaffen, die sogar Auto-Scaling basierend auf CPU und RAM Belegung beherrscht. Sehr ähnlich macht es Cisco mit seinem *Mantl* (siehe Tabelle 1).

Einen ganz eigenen Weg geht dagegen Nomad von HashiCorp, die Firma hinter Terraform und Consul. Es unterstützt ähnlich wie Mesos nicht nur Docker, sondern auch andere Frameworks (von Nomad Driver genannt). Das interessanteste ist dabei Java, womit JAR-Dateien direkt auf Basis von Nomad ausgeführt werden können. Obwohl einige Internas von HashiCorp veröffentlicht wurden, aus denen hervor geht, dass einige firmeneigene Tools wie z. B. Serf wiederverwendet werden, bleibt bei den meisten diskutierten Bausteinen unklar, ob und welche Lösung Nomad hierfür zur Verfügung stellt.

Fazit

Container im Docker-Stil sind ein junges und spannendes Thema. Nur drei Jahre nach dem ersten Docker-Release scheint es aus keiner Cloud-Umgebung mehr wegzudenken, egal ob Public oder Private. Auch im kleinen Stil eingesetzt als Werkzeug für die Entwickler oder zur besseren

Isolation zwischen Anwendungen, die sich einen Server oder eine VM teilen, bringt Docker unmittelbare Vorteile.

Will man es jedoch in größerem Stile einsetzen, so stößt man bald auf eine noch sehr junge und teilweise unausgereifte Toolchain. Wer nicht warten möchte, der greift momentan vermutlich am besten zu Tectonic. Das Angebot scheint dem Autor der „Platzhirsch" unter den Container-Cluster-Managern zu werden, ähnlich wie VMware für virtuelle Maschinen, da bereits zahlreiche andere Unternehmen Kubernetes mit den CoreOS-Tools in ihre Lösungen integrieren (z. B. IBM, HP und Redhat). Insofern kann man damit wohl am wenigsten falsch machen.

Wer über den Docker-Tellerrand hinausschaut, dem sei ein genauerer Blick auf die neuen Angebote Mantl und Nomad empfohlen. Mit Cisco und HashiCorp stecken hier Firmen mit viel Erfahrung und einiger Durchsetzungskraft am Markt dahinter. Das unter Mantl liegende Mesos bietet einiges Potenzial z. B. neben Docker gleich auch Apache Storm, Elasticsearch oder Cassandra auf eine Host-übergreifende Basis zu stellen. Bei Nomad erscheint in erster Linie das direkte ausführen von Java JARs interessant.

Beide Initiativen sind aber zum Zeitpunkt des Schreibens noch kein halbes Jahr alt und haben daher sicher noch die eine oder andere Kinderkrankheit. Wer spezielle Anforderungen und eine innovative Betriebsmannschaft hat, dem sei am Beispiel SCHub gezeigt, dass man auch selber mit vertretbarem Aufwand eine tragfähige Lösung auf der guten Grundlage bestehender Open-Source-Tools bauen kann.

Tabelle 2 fasst die im Text besprochenen Werkzeuge noch einmal zusammen. Der Überblick ist trotz der Fülle keineswegs vollständig, sondern nur ein Ausschnitt des kaum überschaubaren Docker-Ökosystems. ■

Kategorie		Beispiele für Vertreter	
Betriebssystem für Container		CoreOS, Project Atomic, Snappy Ubuntu core	
Container Runtime		Docker engine, LXD, rkt	
Cluster Scheduler		Docker swarm, Apache Mesos, fleet	
Cluster Orchestrator		Docker compose, Marathon, Kubernetes	
Overlay Network		Libnetwork, Weave, flannel, Project Calico, contiv, Fan, Midonet	
Load Balancer		HAproxy, Nginx, hipache, pound, Vulcan	
Service Discovery		Speichern: etcd, Consul, Zookeeper Eintragen: Registrator, confD	
Storage Cluster		Anbinden: REX-Ray, Flocker, contiv, Yp engineering Ceph Bereitstellen: Ceph, GlusterFS, ScaleIO, XtremIO, Google Cloud Engine	
Image Registry		Docker Hub, quay.io, Google Container Registry, Docker private registry	
Logging	Sammeln	Logstash, Greylog2, fluentd	Alles-in-einem: Splunk
	Speichern	Elasticsearch, InfluxDB, Prometheus	
	Auswerten	Kibana, Grafana, Greaylog2	
Monitoring	Sammeln	cAdvisor, Heapster	
	Speichern	InfluxDB, Prometheus	
	Auswerten	Kibana, Grafana	
Identity & Access Mgmt		Docker Data Center mit AD/LDAP Anbindung, Kubernetes AD Plug-In	
Management Dashboard		Cluster: Kubernetes, Mesos/Marathon, Docker UCP Host: Panamax, Cockpit	
Lösungsstacks		Tectonic, Docker Datacenter, Mantl, Nomad	

Tab. 2: Besprochene Werkzeugkategorien mit gängigen Vertretern als Beispiel

Ressourcen & Links

[Roß15] Peter Roßbach: „Docker-Container-Orchestrierung", ObjektSpektrum Themenspecial Innovation in und durch Architekturen, 2015.

[UEP] Understand Engine Plugins, https://docs.docker.com/engine/extend/plugins/

[Sve16] Yevgeniy Sverdlik: „Docker Makes Docker Easier for Data Center Managers", 24.02.16 auf datacenterknowledge.com, http://bit.ly/1STVEYD

[Sei16] Udo Seidel: „Build, Ship, Run - Das Docker-Universum im dritten Lebensjahr", iX 4/2016, S. 46.