HOCHSCHULE DER MEDIEN

Stuttgart Media University

Bachelorarbeit im Studiengang Medieninformatik

Konzeption und Implementierung eines Monitoring Systems für Docker Swarm Cluster

vorgelegt von

Oliver Fesseler

an der

Hochschule der Medien Stuttgart

am 28. Februar 2017

zur Erlangung des akademischen Grades

eines Bachelor of Science

Erstprüfer: Prof. Dr. Martin Goik

Zweitprüfer: Thomas Maier

Ehrenwörtliche Erklärung

„Hiermit versichere ich, Oliver Fesseler, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel: „Konzeption und Implementierung eines Monitoring Systems für Docker Swarm Cluster“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

Ort, Datum Unterschrift

Zusammenfassung

Skalierbarkeit und Ausfallsicherheit sind bei der Entwicklung von Verkaufsplattformen relevante Faktoren, um Verlässlichkeit gegenüber dem Kunden zu gewährleisten und dessen Vertrauen zu gewinnen. Clusterlösungen sind dazu prädestiniert diese Anforderungen zu erfüllen, da sie auf sich ändernde Bedingungen flexibel reagieren können und bei dem Ausfall von einzelnen Komponenten unter Umständen immer noch funktionsfähig bleiben. Im Folgenden wird ein Cluster vorgestellt, welches mit den Infrastrukturkomponenten Docker, Docker Swarm, Weave Net, Consul und GlusterFS betrieben wird. Zur Gewährleistung eines unterbrechungsfreien Betriebes muss ein Cluster überwacht werden. Dabei ist es wichtig einen Überblick über den aktuellen Status zu haben, damit im Fehlerfall entsprechend reagiert werden kann. Diese Bachelorarbeit beschäftigt sich mit der Evaluation von Metriken, welche den aktuellen Clusterstatus repräsentieren.

Abschließend wird unter Einbezug dieser die prototypische Implementierung eines Überwachungssystems für die Infrastruktur ausgearbeitet. Der besondere Fokus dieser Bachelorarbeit liegt dabei auf der Interpretation und Aggregation der aus den Komponenten gewonnenen Metriken.

Stichworte Monitoring, Cluster, Docker, Docker Swarm, Verteile Systeme, Prometheus, Consul

Abstract

In developing sales platforms, scalability and availability are important features to ensure reliability and therefore gain the trust of customers. Cluster solutions are qualified to reach these objectives, since they can react dynamically to changing conditions and enable service even in cases of failure of single components. The cluster consists of the following components: Docker, Docker Swarm, Weave Net, Consul and GlusterFS. To ensure high availability, clusters need to be monitored. In order to enable fast reactions to upcoming errors, the current system status needs to be represented in a suitable way. This thesis is about the evaluation of metrics representing the current cluster status. Subsequently, a prototypical implementation of a monitoring system, including the priorly discussed metrics, is developed. This work focuses on the interpretation and aggregation of the metrics that are derived from the single components within the cluster.

Inhaltsverzeichnis

1. Einleitung **2**

2. Grundlagen **4**
2.1. Microservices Architektur 4
2.2. Cluster 5
2.3. Monitoring 10
2.4. Sicherheit 11

3. Konzept **13**
3.1. Clusterstatus 13
3.2. Metriken 14
3.3. Implementierungsziele: Panopticon-Monitoring 19

4. Umsetzung **21**
4.1. Testcluster mit Docker Swarm, Consul und GlusterFS 21
4.2. Evaluation Metriken 23
4.3. Tool: Panopticon-Monitoring 26

5. Fazit und Ausblick **32**
5.1. Bewertung der Umsetzung im Vergleich zur Konzeption 32
5.2. Verwendete Software 32
5.3. Ausblick 33

Literatur **34**

Abbildungsverzeichnis **36**

Tabellenverzeichnis **37**

Listings **38**

Glossar **39**

Akronyme **41**

A. Anhang **42**
A.1. Beispielhafte Darstellung der Metriken eines Exporters anhand des Gluster Exporters 42

1. Einleitung

Die Leomedia GmbH entwickelt ein neues System zum Ticketverkauf. Das alte System war eine monolithische Anwendung mit einer über Jahre und verschiedene Anforderungen hinweg gewachsenen Codebase. Eine Neuentwicklung soll nun eine höhere Flexibilität sowie einfachere kundenspezifische Anpassungen ermöglichen. Allerdings ist die Skalierbarkeit der Anwendung eine große Herausforderung, da das alte Produkt für jeden Kunden spezifisch auf gemieteter oder eigener Infrastruktur installiert wurde. Um eine einfachere Wartung und schnellere Evolution zu ermöglichen, werden zukünftig alle Kunden auf einer gemeinsamen Infrastruktur in einer Installation bedient. Damit einher geht die Anforderung ein skalierbares Produkt zu entwickeln und das *Cluster* anpassbar zu machen. Nach Evaluation verschiedener Ansätze hat sich die Entwicklung der Leomedia GmbH für eine Microservice Architektur entschieden, mit welcher die Ziele eines anpassbaren, skalierenden und flexiblen Systems zu erreichen sind.

Aus der Neuentwicklung und der Änderung der Technologien ergibt sich die Aufgabe eines Neuaufbaus einer geeigneten Infrastruktur und damit einhergehend die Neuimplementierung eines Monitoringsystems. Dabei wird zwischen der Infrastruktur- und der Applikationsüberwachung differenziert. In dieser Bachelorarbeit wird ausschließlich die Überwachung der Infrastruktur behandelt, nicht die Überwachung der Microservices. Unabhängig davon können die Technologien ebenfalls für die Überwachung der Applikation eingesetzt werden. Die Herangehensweise an die Implementierung der Überwachung des Clusters wird im Folgenden anhand der aufgeführten Fragen bearbeitet:

- Wie kann ein *Cluster* überwacht werden?
- Welche Metriken sind für den Betrieb wichtig?
- Wie kann eine Übersicht der wichtigsten Metriken erstellt werden?
- Welche Sicherheitsaspekte müssen beachtet werden?
- Wie sieht eine beispielhafte Monitoring Lösung aus?

Um Antworten auf diese Fragen zu erhalten, wurde ein auf Docker Swarm aufbauendes Testcluster eingerichtet, welches mit dem Entwicklungscluster der Leomedia GmbH in der Auswahl der Komponenten und deren Versionen übereinstimmt. Dies wird mit dem Monitoringsystem Prometheus überwacht um aus den gewonnenen Daten den Clusterstatus abzuleiten.

In *Kapitel 2* wird auf die grundsätzliche Funktion der Komponenten und deren Rolle innerhalb des Clusters eingegangen. Das zentrale Kapitel wird *Kapitel 3*. In diesem werden

Abhängigkeiten dargestellt und erörtert, erklärt wie der Status des Clusters bestimmt werden kann und es findet eine Evaluation der Metriken statt. Die praktische Umsetzung wird in *Kapitel 4* beschrieben, aufgeteilt in die Bereiche: Installation des Testclusters (*Abschnitt 4.1*), Evaluation der Metriken (*Abschnitt 4.2*) und der Implementierung des Prototypen Panopticon-Monitoring (*Abschnitt 4.3*). Abschließend wird in *Kapitel 5* ein Fazit gezogen und ein Ausblick auf die Weiterentwicklung sowie potentielle zukünftige Entwicklungen gegeben.

2. Grundlagen

In diesem Kapitel wird der Aufbau des Testclusters, welches als Basis für die angestrebte Microservice Architektur dient, beschrieben. Dazu wird eine kurze Einführung in die Microservice Architektur und deren Vor- sowie Nachteile gegeben. Anschließend werden die einzelnen Komponenten des Clusters vorgestellt. Der Einsatzzweck der jeweiligen Komponente wird dargestellt, um die Auswahl nachvollziehbar zu machen. Dabei wird auf die einzelnen Softwarekomponenten eingegangen und deren Besonderheit und Funktion für dieses Testcluster erörtert. Dabei wird auf die Funktionsweise der Komponenten nur in Grundzügen eingegangen, da die Projekte sehr gut und aktuell dokumentiert sind.

2.1. Microservices Architektur

In den letzten Jahren hat der Begriff der Microservice Architektur zunehmend Auferksamkeit erhalten [2] [4]. Er beschreibt eine Architektur, in der unabhängige einzelne Services mit einfachen Mechaniken kommunizieren und ein System bilden. Eine eindeutige Definition des Begriffes gibt es nicht[1], jedoch geht damit oft ein automatisiertes Deployment und eine organisatorische Trennung der Services einher.

Wenn Microservice Architekturen beschrieben werden, tauchen verschiedene Prinzipien in der Literatur immer wieder auf [8]. In einer Microservice Architektur werden kleine Services erstellt, die vollständig erfassbar sind und in der Summe die Applikation darstellen. Die Programmiersprache kann dabei anhand des spezifischen Problems des Services gewählt werden. Zudem ist es einfach einen Service neu zu schreiben und mit dem Vorgänger auszutauschen. Mit Microservice Architektur geht oft ein Strukturwechsel innerhalbn einer Organisation einher, sodass die Verantwortlichkeit für das gesamte Produkt von einer organisatorischen Einheit, im weiteren als Team bezeichnet, übernommen wird. Somit teilt sich die Arbeit an einer Applikation häufig auf mehrere unabhänige Teams auf. Dementgegen steht eine Struktur, die sich nach den Verschiedenen Layern einer Architektur richtet, wie Frontend, Backend und Administration. Weiterhin besteht theoretisch die Möglichkeit einzelne Services zu skalieren, ohne dass die gesamte Anwendung skaliert werden muss, anders als dies bei monolithischen Architekturen der Fall ist.

Der Unterschied in der Skalierung zwischen einer monolithischen Applikation und einer Microservice Architektur wird in *Abbildung 2.1* dargestellt und soll im Folgenden kurz zusammengefasst werden. Bei einer monolithischen Applikation befindet sich die gesamte Funktionalität in einem Prozess. Daher wird diese im Gesamten repliziert und

[1] Etymologie der Microservices Architektur *https://martinfowler.com/articles/microservices.html#footnote-etymology*

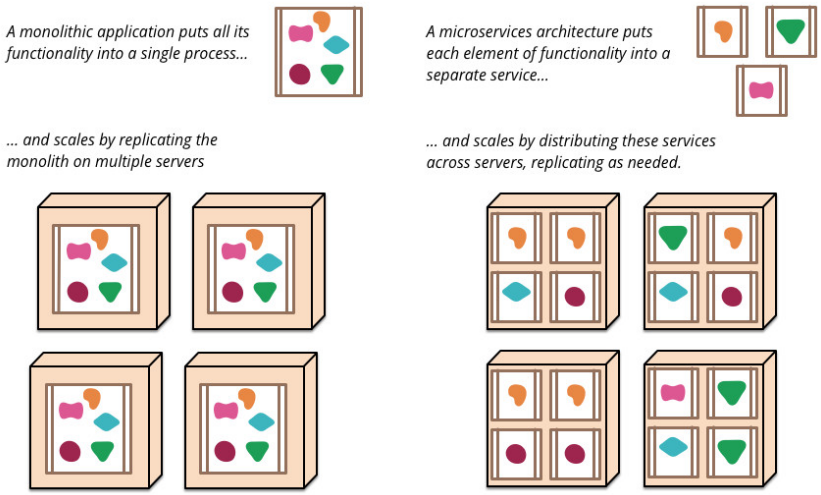


Abbildung 2.1.: Monoliths and Microservices ©Martin Fowler [2]

auf mehreren Servern betrieben. Im Gegensatz dazu steht die Microservice Architektur, hier wird jede Funktionalität in einen seperaten Service verpackt. Dadurch ist es möglich, die einzelnen Services nach Bedarf zu replizieren und über Servergrenzen hinweg zu verteilen.

Der Nachteil einer solchen Architektur besteht in der höheren Gesamtkomplexität des Systems. Mit der Entkopplung ist ein größerer Aufwand in der Kommunikation verbunden, wodurch wiederum mit Latenzen und Fehlern gerechnet werden muss. Durch eine Vielzahl an Services kann das Testen der gesamten Applikation komplexer werden. Zudem ist eine Migration von einer Monolithischen Applikation hin zu einer Microservice Architektur eine Änderung des Entwicklungsparadigmas, welches vom Team akzeptiert und getragen werden muss. Entgegen der oft üblichen Aufteilung der Teams nach Kompetenzen und Aufgaben wie Frontend, Backend und Infrastruktur, wird hier üblicherweise der Ansatz verfolgt, dass ein Team für den gesamten Service verantwortlich ist. Aufgrund der Vorteile einer Microservices Architektur, wie Flexibiltät und einfachere kundenspezifische Anpassbarkeit, hat sich die Leomedia GmbH dennoch für eine solche Architektur entschieden. Im Folgenden soll der Cluster und seine einzelnen Komponenten vorgestellt werden.

2.2. Cluster

Allgemein gesehen ist ein Computercluster ein Verbund von Computern, welche untereinander agieren und nach außen eine Einheit bilden. In dieser Bachelorarbeit wird unter dem Begriff *Cluster* eine bestimmte Auswahl an Komponenten verstanden. Diese ergeben sich aus der Konfiguration des Testclusters, welches von der Leomedia GmbH vorgegeben wurde.

Dabei kommen Docker, Docker Swarm, Consul, Weave Net und GlusterFS zu Einsatz. Im foldenden werden die einzelnen Komponenten und deren Aufgaben kurz erläutert.

2.2.1. Docker

Docker ist ein Open Source Projekt, welches ermöglicht Applikationen als leichtgewichtige Container zu erstellen, auszuliefern und zu betreiben [9]. Docker Container sind plattform - und hardwareunabhängig und können von einem Laptop bis zu einem großen Server betrieben werden. Die verwendete Programmiersprache oder das Framework sind unabhängig von Docker wählbar. Es können skalierbare Web Apps und Services geschrieben werden, die unabhängig von einem bestimmten Software Stack oder Betreiber sind. Ursprünglich wurde Docker als Open Source Implementierung der Deployment Engine von dotCloud[2] entwickelt.

Problematisch für Produktionsumgebungen ist, dass Docker noch relativ jung ist. Mit jeder Version kommen Neuerungen hinzu, Funktionalitäten fallen weg und *API* verändern sich. Das macht es sehr aufwändig ein Projekt auf dem aktuellen Stand zu halten um Sicherheit und Aktualität zu gewährleisten.

Docker eignet sich aufgrund der Plattformunabhängigkeit und Isolierung allerdings sehr gut um eine Microservice Architektur zu realisieren. Einzelne Container können ohne weiteres ausgetauscht, gelöscht oder neu hinzugefügt werden. Um ein modulares, fexibles und skalierbares System zu erhalten, sollten Container möglichst nur Funktionalität für ihr Hauptziel beinhalten [17]. Unterziele können von anderen Containern erfüllt werden. Dadurch ist eine hohe Austauschbarkeit und geringe Kopplung gegeben [14].

Docker besteht aus verschiedenen Komponenten, deren Aufgabe im Folgenden kurz erläutert wird [7].

Docker Daemon

Der zentrale, auf dem Hostsystem laufende Service einer Docker Installation. Dieser verwaltet die lokalen Docker Container und Images. Zudem werden noch Filesystem und Netzwerkkomponenten verwaltet.

Dockerfile

Ein Dockerfile stellt eine „Bauanleitung„ für ein Image dar. Darin werden u.a. die Abhängigkeiten, das zugrunde liegende Image und die Kommunikation nach außen beschrieben.

Docker Image

Das Image ist eine portable Abbildung eines Containers [18]. Dies enthält alle notwendigen Daten um die definierten Anwendungen zu betreiben. Zusätzlich enthält es einige Metadaten, wie das zugrunde liegende Image und die Applikation, die ausgeführt werden soll. Images können aufeinander aufbauen, so kann zum Beispiel ein existierendes Debian-Image um die gewünschte Applikation erweitert werden.

[2] urlhttps://github.com/docker/docker/blob/v1.13.1/README.md

Docker Container

Wird ein Container gestartet, so wird aus dem Image und einem eigenen Filesystem Layer eine virtuelle Umgebung erstellt. Daraufhin wird ein vom Hostsystem isolierter Prozess gestartet, der die im Image definierte Anwendung startet.

Docker Client

Der Docker Client ist ein Tool mit einem *CLI*, welches den Docker Daemon verwalten kann. Der Daemon wird mittels Docker *API* direkt über HTTP angesprochen.

Docker Registry

Die Registry ist der Ort an dem Docker Images gehalten werden. Die Docker Inc. stellt den Docker Hub, eine öffentlich verfügbare Docker Registry zur Verfügung. Daneben ist es ebenfalls möglich eine eigene Registry zu betreiben, dafür wird eigens ein Image auf Docker Hub bereitgestellt[3].

Docker Swarm

Docker Swarm ist eine systemeigene Clusterlösung für Docker. Es handhabt eine Ansammlung von Docker Hosts wie einen einzelnen, virtuellen Docker Host und kommuniziert mit diesem über die Docker *API*. Weil Docker Swarm die „Standard Docker *API*“ ausliefert, kann jedes Tool, das bereits mit einem Docker Daemon kommuniziert den Swarm benutzen, um transparent auf viele *Host* zu skalieren. Docker Swarm selbst speichert keine Daten, diese werden ausschließlich durch einen Key-Value Store verwaltet [10].

Abgrenzung: Swarm mode von Docker Docker Swarm ist ein Produkt von Docker Inc.[4] um einen Verbund von mehreren Docker Engines zu erstellen und zu verwalten. Bis zur Version 1.12 war dies nur mit Docker Swarm möglich, seit Version 1.12 ist der Swarm mode bereits in Docker integriert[5]. Der Swarm mode vereinfacht die Verwaltung von Services im Docker *Cluster* und bietet u.a. eine eigene Service Discovery.

Docker Network

In Docker sind drei Netzwerke voreingestellt: host, bridge, none. Diese können Containern zugewiesen werden. Docker bietet die Möglichkeit benutzerdefinierte Netzwerke zu erstellen, ein passender Treiber wird mitgeliefert. Der Nutzer kann sich entweder an diesen bedienen oder eine andere Implementierung wählen. In *Abbildung 2.2* ist ein virtuelles Container Netzwerk in Docker Swarm schematisch dargestellt. Bei Weave Net handelt es sich um ein virtuelles Container Netzwerk. Es wird in *Unterabschnitt 2.2.3* genauer erläutert.

[3]Docker Registry auf Docker Hub *https://hub.docker.com/_/registry/*
[4]Produkt Webseite *https://www.docker.com/products/docker-swarm*
[5]Stand 16.12.2012, Docker v1.12.5

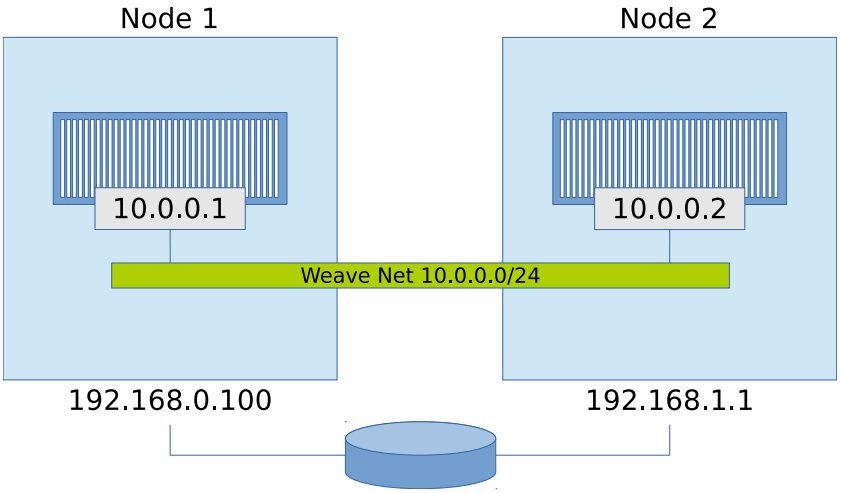


Abbildung 2.2.: Schematische Darstellung eines Weave Net Netzwerks in Docker Swarm

2.2.2. Consul

Consul ist ein verteilter Key-Value Store, eine Service Discovery und Failure Detection Software. In einem Docker Swarm *Cluster* stellt Consul eine Liste der registrierten Docker Swarm Instanzen bereit. Dabei wird ein health-Wert zu jeder Instanz hinzugefügt, der Aufschluss über den aktuellen Status gibt. Consul wählt mit Hilfe des Raft Konsensus Algorithmus [3] die Instanz aus, die als Koordinator des Verbundes dient, näheres ist in *Exkurs 1* beschrieben.

In der Testinstallation dient Consul als Key-Value Store und Service Discovery. Docker Swarm legt darin gemeinsam genutzte Daten ab und kann gemeinsam genutze Services auffinden. Dazu gehören wichtige Eigenschaften für das Cluster, wie z.B.: der *Cluster* Leader und die registrierten Nodes.

Exkurs 1: Raft Konsensus Algorithmus

Raft ist ein konsensus Algorithmus, der ähnlich dem Paxos Algorithmus [1] ist. Im Vergleich zu Paxos ist Raft einfacher zu verstehen und einfacher zu implementieren [3]. Dadurch versprechen sich die Entwickler geringere Fehleranfälligkeit und bessere Zuverlässigkeit als bei anderen konsensus Algorithmen. Ein Merkmal von Raft ist, dass es immer einen Leader gibt, über den alle Aktionen ausgeführt werden. Dieser wurde von einer Mehrheit der beteiligten Knoten im *Cluster* gewählt. Ist dies nicht der Fall, bleibt/ist Consul funktionsunfähig.

2.2.3. Weave Net

Weave Net ist ein virtuelles Container Netzwerk für Docker Container. Es verbindet Docker Container über *Host* hinweg und ermöglicht deren automatische Erkennung. Weave Net ermöglicht eine Kommunikation über Rechenzentrumsgrenzen hinweg, sodass

die Container miteinander kommunizieren können als wären diese mit einem Switch verbunden [12].

Ein wichtiges Feature ist die durch Weave Sleeve ermöglichte Verschlüsselung des Netzwerkverkehrs. Dies ist besonders wichtig, wenn eine Kommunikation über öffentliche Netzwerke notwendig ist. Weave nutzt zur Verschlüsselung und Authentifizierung der Nachrichen des Kontrollverkehrs und des eigentlichen Datenverkehrs die Verlüsselungsbibliothek NaCl[6].

Der von Weave Net bereitgestellte *DNS* Service weaveDNS verwaltet die Namensauflösung im Netzwerk. Load Balancing wird von weaveDNS ebenfalls unterstützt und es werden alle zu einem Domainnamen zugehörigen *IP*-Adressen in zufälliger Reihenfolge zurückgegeben. Dadurch wird eine zufallsbedingte Lastverteilung auf den Systemen erreicht.

2.2.4. GlusterFS

GlusterFS ist ein verteiltes Dateisystem, das auf *FUSE* aufbaut [5]. Es bietet die Möglichkeit Dateien zu replizieren, zu verteilen, einzelne Dateien über mehrere Hosts zu verteilen oder eine Kombinationen dessen [11]. Diese Modi werden „Distributed“, „Replicated“ und „Striped“ genannt. Im ersten Modus werden die Daten auf den Bricks - siehe unten - verteilt. Damit kann das Laufwerk skaliert werden ohne auf die Redundanz zu achten, diese ist Aufgabe von anderen Soft- und Hardwareschichten. Der zweite Modus repliziert die Daten auf alle Bricks im Volume, um Ausfallsicherheit und Verfügbarkeit mit Gluster zu erreichen. Im Striped-Modus werden die Dateien selbst auf Bricks verteilt, um einen höheren Schreibdurchsatz zu erlangen, was vor allem in hoch parallelen Umgebungen mit großen Dateien empfohlen wird. Im Testcluster werden die Daten ausschließlich repliziert, um Redundanz und Ausfallsicherheit zu gewährleisten.

Brick Ein Brick ist die Repräsentation der Daten eines Volumes auf einem *Host* im GlusterFS Verbund. Über Bricks hinweg verläuft die Synchronisierung und Verteilung der Daten eines Volumes.

Volume Ein Volume ist ein virtuelles GlusterFS Laufwerk bestehend aus Bricks, dass von einem Gluster Client gemountet werden kann.

Das Besondere an GlusterFS ist, dass es mit Consumer Hardware betrieben werden kann.

Damit ist die Beschreibung der Komponenten des Clusters abgeschlossen. Als nächstes wird auf das Monitoring des Clusters genauer eingegangen.

[6]Encryption and Weave Net *https://www.weave.works/docs/net/latest/how-it-works/encryption/*

2.3. Monitoring

Der Begriff Monitoring beschreibt die Protokollierung, Messung, Beobachtung oder Überwachung eines Vorgangs mit Hilfe von technischen Hilfsmitteln. Die regelmäßige Durchführung dieser ist ein zentrales Element um durch Vergleichen der Ergebnisse Schlussfolgerungen ziehen zu können, vgl. [13] .

Üblicherweise wird dabei auf vorhandene Software zurückgegriffen. Da beim Monitoring Daten mit Zeitbezug anfallen bieten sich Zeitserien- Datenbanken, auch *TSDB* genannt, an. Für verteilte dynamische Systeme bietet sich das in *Unterabschnitt 2.3.1* eingeführte Prometheus, welches Überwachungstool und TDSB in einem ist, an. Die in der TDSB gesammelten Daten sollten übersichtlich dargestellt werden, um schnelle Rückschlüsse zu ermöglichen. Um die Daten darzustellen empfehlen sich die folgenden Anhaltspunkte:

- Zentrale Übersichtsseite auf der die wichtigsten Daten abgebildet werden
- Zugunsten des Überblicks sollte eine Bildschirmseite nicht überschritten werden
- Detailliertere Informationen auf weitere Übersichtsseiten nach Thema oder Aufgabe sortiert

Als grobe Regel gilt, nicht mehr als 6 Graphen auf einem Dashboard und nicht mehr als 6 Plots in einem Graphen darzustellen. Damit ist laut Farcic [15] eine gute Erkennbarkeit und Übersicht gewahrt. Jedes Team sollte ein eigenes Dashboard haben, um den Zielen des Team gerecht zu werden, allerdings sollte nicht versucht werden jeder Person gerecht zu werden, damit die Teamziele gewahrt bleiben. Als Empfehlung gibt Farcic [16] an, nicht auf Anhieb zu versuchen das bestmögliche Dashboard zu erstellen, sondern es als iterativen Prozess zu sehen in dem es immer weiter verbessert wird.

2.3.1. Prometheus

Prometheus ist ein Überwachungstool und eine *TSDB* zur Überwachung von verteilten Systemen. Es wurde von Soundcloud entwickelt und erfreut sich einer wachsenden Benutzerbasis und sehr aktiven Community.

TSDB sind entworfen und optimiert um Zeitseriendaten zu speichern. Ein großer Vorteil ist, dass diese in der Lage sind, große Mengen an Daten sehr effektiv zu speichern. Ein Problem der meisten *TSDB* ist die fehlende Möglichkeit diese als verteilte Datenbank zu betreiben. Häufig ist es jedoch für Monitoring nicht relevant, dass diese Daten über einen langen Zeitraum in hoher Auflösung gespeichert werden. Durch regelmäßiges „Säubern“ der Datenbank kann diese in einer überschaubaren Größe gehalten werden. Zum anderen ist eine hohe Ausfallsicherheit nicht notwendig, da mit aggregierten Daten gearbeitet wird und deshalb der Verlust von Daten über einen kurzen Zeitraum verkraftet werden kann.

Viele Anwendungen besitzen eine *API*, um Monitoringdaten im Prometheus-Format abzufragen.

Ist dies nicht der Fall, so gibt es eine sehr große Auswahl an Prometheus Exportern. Diese sind eigenständige Programme, welche Metriken der zu überwachenden Software für Prometheus verfügbar machen. Der Terminus, den Prometheus dafür vorsieht, lautet Instrumenting. Die instrumentierten Daten werden per *HTTP* abgerufen und in die Datenbank geschrieben.

Damit die Daten nicht nur aufgeschrieben und ggf. betrachtet werden, gibt es den AlertManager. Dieser benachrichtigt über die konfigurierten Kanäle bei Abweichungen der geforderten Werte. Dieser ist ein Plugin für Prometheus, das beliebig konfiguriert und angepasst werden kann.

Ein Dashboard, wie in *Abschnitt 2.3* beschrieben stellt Grafana dar, das eine diekte Schnittstelle für Prometheus mitbringt. Darin können die aufgezeichneten Daten auf verscheidene Weise dargestellt und analysiert werden[7].

AlertManager ist ein Programm, das direkt mit Prometheus zusammenhängt und für die Benachrichtigung einer Grenzwertüberschreitung zuständig ist. Die Benachrichtigungskanäle sind sehr vielfältig und werden ständig erweitert[8]. Desweiteren beherrscht der AlertManager die Stummschaltung von Kanälen und eine intelligente Benachrichtigung um zu vermeiden, dass Administratoren möglichst ausschließlich wesentliche Nachrichten mitgeteilt werden.

Aufbau Abtastwert In *Listing 2.1* ist der schematische Aufbau eines Abtastwertes oder auch *Metrik* dargestellt. Dieser ergibt sich aus dem Namen, einem Label einem Zeitstempel und dem tatsächlich gemessenen Wert der als `float64` oder `int64` repräsentiert wird.

```
1 <metric_name>{<label_name>="<label_value>", ...} = [<timestamp>, <value>]
```

Listing 2.1: Schema eines Prometheus Abtastwerts

2.4. Sicherheit

Die vorgestellten Anwendungen kommunizieren alle über ein Netzwerk. Wenn die Docker Hosts im Internet sind, dann kommt die Fragen nach der sicheren Kommunikation innerhalb des Clusters auf.

Im Entwicklungs- und Testcluster wurde ein Ansatz gewählt der auf Standardkomponenten im Internet basiert, den SSL-Zertifikaten und den HTTP Verkehr zu verschlüsseln. Dabei wird eine eigene *CA* erstellt, die für alle beteiligten Parteien eine Zertifikat ausstellt. Durch dieses Vorgehen kann zum Beispiel der Webserver auf ein Client Zertifikat prüfen um sicherzustellen, dass der zugreifende Client von der *CA* ebenfalls ein Zertifikat ausgestellt bekommen hat. Dieses Vorgehen sichert Authenzität der Clients und dient als Basis für die Integrität und Verschlüsselung der Verbindung.

Die Container untereinander Kommunizieren über das Weave Net, der Verkehr innerhalb dessen ist ebenfalls Verschlüsselt siehe *Unterabschnitt 2.2.3*.

[7] Grafana *http://grafana.org/*

[8] AlertManager Benachrichtigungskanäle *https://prometheus.io/docs/alerting/configuration/*

Nachdem die Microservice Architektur eingeführt und alle speziellen Komponenten, welche in das *Cluster* involviert sind besprochen wurden, wird in das nächste Kapitel übergegangen. Dieses bietet eine Einführung in das Problem der Bewertung von Überwachungsmetriken, definiert den Clusterstatus und beschäfitgt sich eingehend mit der Beschreibung und Begründung der verwendeten Metriken, welche anhand der verwendeten Komponenten untergliedert und diskutiert werden. Das im Rahmen der Bachelorarbeit erstellte Tool Panopticon-Monitoring wird anschließend erläutert und im Anschluss wird mit den Zielen für das *Kapitel 4* abgeschlossen.

3. Konzept

Dieses Kapitel ist eine Hinführung zur Problematik eines aggregierten Clusterstatus. Mit Hilfe der Services Consul, GlusterFS, Weave Net, Docker und Docker Swarm werden einzelne Hosts zu einem *Cluster* zusammengeschlossen und bieten eine Plattform für eine auf Docker Container aufbauende Anwendung. Der Status der involvierten Services beeinflusst den Gesamtzustand des Clusters. Um einen unmittelbaren und direkt ableitbaren Status zu erhalten, wird in diesem Kapitel eine Klassifizierung der Services durchgeführt und die Abhängigkeiten der Services beleuchtet.

Im *Cluster* bauen die Komponenten aufeinander auf oder sind direkt oder indirekt voneinander abhängig, wie in *Abbildung 3.1* abgebildet. Um einen Service in einem Docker Container in dem Testcluster zu deployen, müssen die folgenden Komponenten verfügbar sein: Docker Swarm übernimmt die Verteilung und Konfiguration der Container auf den Hosts. Consul speichert den Zustand für Docker Swarm und hält Konfigurationsdaten vor. Weave Net stellt ein virtuelles Netzwerk für die Kommunikation der Container dar. GlusterFS übernimmt die Aufgabe des verteilten Dateisystems, dass die Daten synchron auf allen Hosts verwaltet.

Fällt nun eine dieser Komponenten vollständig aus, verliert da *Cluster* an Funktionsfähigkeit.

3.1. Clusterstatus

Um deutlich und schnell zu erkennen, welchen Status da *Cluster* hat, werden drei Farben (Rot, Orange und Grün) eingeführt. In *Tabelle 3.1* ist die Zuordung der Farbe mit dem Clusterstatus bzw. der Clustergesundheit dargestellt.

●	**healthy**	Cluster ist **voll** funktionsfähig.	*Keine Einschränkungen.*
●	**warning**	Cluster ist **teilweise** funktionsfähig.	*Ein Eingriff wird empfohlen.*
●	**critical**	Cluster ist **nicht** funktionsfähig.	*Ein Eingriff ist notwendig.*

Tabelle 3.1.: Je nach Funktionsfähigkeit wird der Status mit den Begriffen *healthy*, *warning* und *critical* beschrieben.

In *Abbildung 3.2* werden die verschiedenen Zustände als Zustandsautomat dargestellt und je nach Änderung der Funktionsfähigkeit, wird der Status geändert.

Fällt ein Service aus, der für die volle Funktion des Clusters benötigt wird, aber dennoch alle Operationen möglich sind, ändert sich der Status von *healthy* zu *warning*. Ein Eingriff ist notwendig, jedoch ist der Dienst noch verfügbar. Ist hingegen der Ausfall

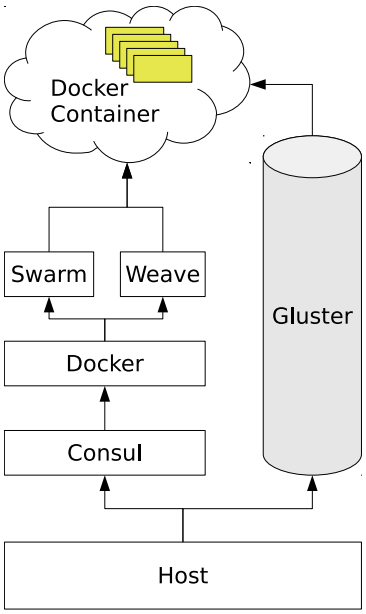


Abbildung 3.1.: Abhängigkeiten der Services im Docker Swarm Cluster

schwerwiegend und das System kann keine Operationen mehr ausführen, ist der Status *critical*. Um die Funktionsfähigkeit wieder herzustellen ist ein direkter Eingriff notwendig.

Lösen sich die Probleme, zum Beispiel duch einen Eingriff, ändert sich der Status entsprechend. Wird ein kritischer Fehler behoben und da *Cluster* ist wieder voll funktionsfähig ändert sich der Status in *healthy*. Sind noch andere Komponenten betroffen, die nur eingeschränkt funktionieren, ändert sich der Status zu *warning*.

3.2. Metriken

Jeder Service hat eigene Kennzahlen, anhand derer seine Funktionsfähigkeit abgefragt werden kann. Manche können von alleine regenerieren, andere benötigen einen manuellen Eingriff. Nachfolgend werden die Services in Bezug auf ihre Metriken untersucht und begründet, aus welchen ein Rückschluss zum Status des Clusters gezogen werden kann.

Die meisten eingesetzen Services bieten eine Vielzahl von Metriken, die im Folgenden nicht alle untersucht werden. Vielmehr wird auf die Metriken eingegangen, die sich im Laufe des Testbetriebes als verwendbar und sinnvoll herausgestellt haben. Diese Beurteilung hängt stark vom Anwendungsfall und der Größe des Clusters ab. In diesem Fall handelt es sich um eine Lösung die mit wenigen Hosts arbeitet (< 5). Der Schwerpunkt liegt auf der Bedeutung und Begründung der einzelnen Metriken.

3.2.1. Consul Metriken

Consul als verteilter Key-Value-Store hat immer einen Knoten, der als Leader fungiert. Nur dieser darf Einträge erstellen und verändern. Fällt dieser aus, wird ein neuer Leader

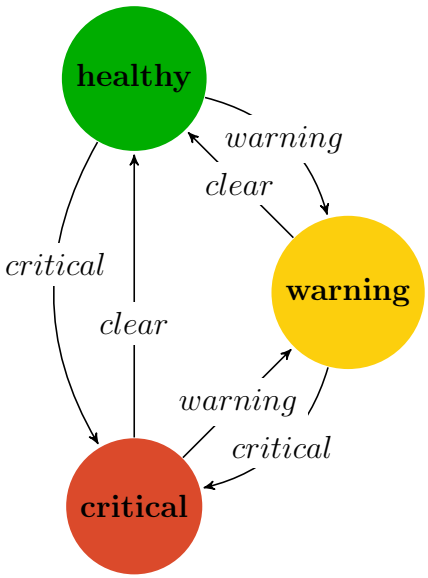


Abbildung 3.2.: Zustandsdiagramm mit Zustandsübergängen der verschiedenen Clusterstatus

gewählt. Ein Leader kann nur mit einem *Quorum*gewählt werden. Das bedeutet er benötigt eine eindeutige Mehrheit $(N_{active}/2) + 1$ der Stimmen der beteiligten Nodes N_{total}. Somit ist für Consul kritisch, dass es einen Leader gibt.

Fällt hingegen ein Knoten aus und es kann weiterhin eine Mehrheit gebildet werden, ist das System weiterhin funktionsfähig. In *Abbildung 3.3a* sind die Abhängigkeiten und Auswirkungen des Status *warning* anderer Services zu Consul dargestellt. Da ein Knoten ausgefallen ist, wird dieser Zustand als *warning* beurteilt. Es sollte etwas unternommen werden, Consul ist jedoch funktionsfähig. Ist Consul nicht betriebsbereit (siehe *Abbildung 3.3b*), kann Swarm nicht agieren und es sind keine administrativen Operationen auf de *Cluster* möglich.

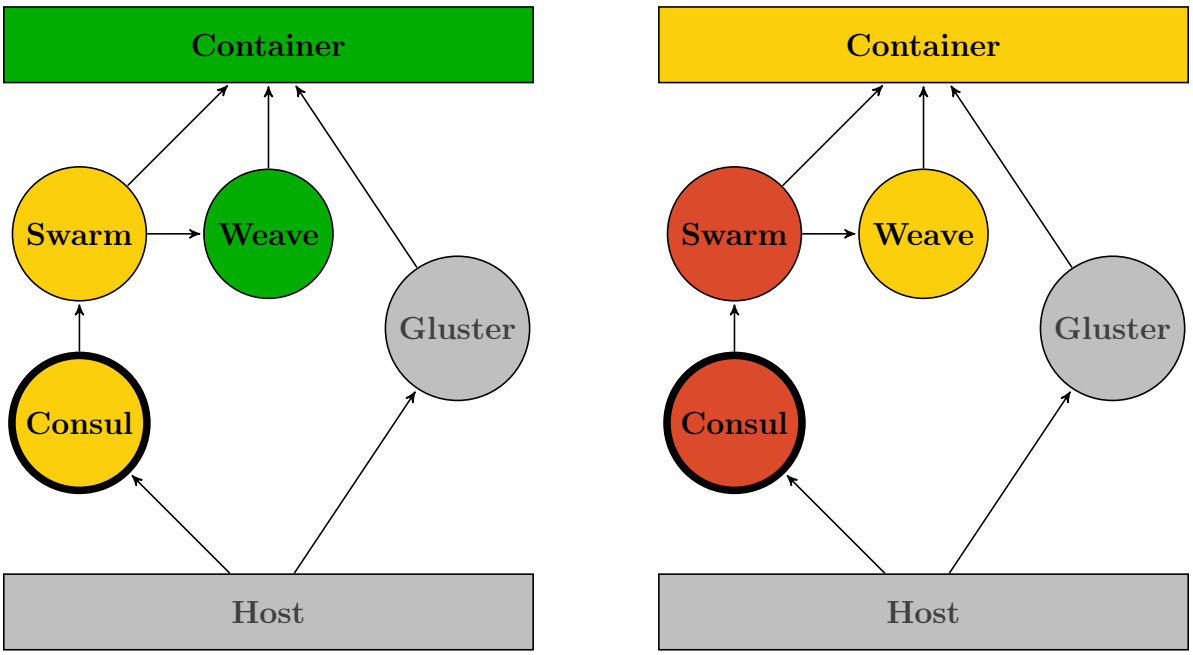
Die gestarteten Container sind grundsätzlich funktionsfähig, können aber weder skaliert, noch kann sichergestellt werden, dass deren Abhängigkeiten voll funktionsfähig sind. Daher ist dieser Zustand als *critical* einzuordnen.

3.2.2. GlusterFS Metriken

GlusterFS wird im Testcluster als synchron replizierendes Dateisystem verwendet. In dieser Konfiguration sind alle Knoten des Clusters gleichzeitig Server und Client. Dadurch wird sicher gestellt, dass zu jedem Zeitpunkt alle Daten auf allen Nodes verfügbar sind. Dieser Ansatz wird von GlusterFS für größer *Cluster* (> 3) ausdrücklich nicht empfohlen, da eine zu hohe Replizierung Dateioperation drastisch drosselt.

Beim Start von GlusterFS versucht dieses sich zu allen konfigurierten Peers zu verbinden. Ist dies nicht möglich, findet auf den nicht verbundenen Bricks keine Replizierung statt. Daher ist dies ein kritischer Wert.

Nach dem Start des Gluster Daemons wird versucht, das Gluster Volume in das



(a) **Consul: warning** Consul ist im Status warning, direkt davon sind Docker Swarm, Weave Net und die Container Applikationen betroffen. Da Consul seiner Funktion noch nachkommt, ist der Dienst der abhängigen Services nicht beeinträchtigt.

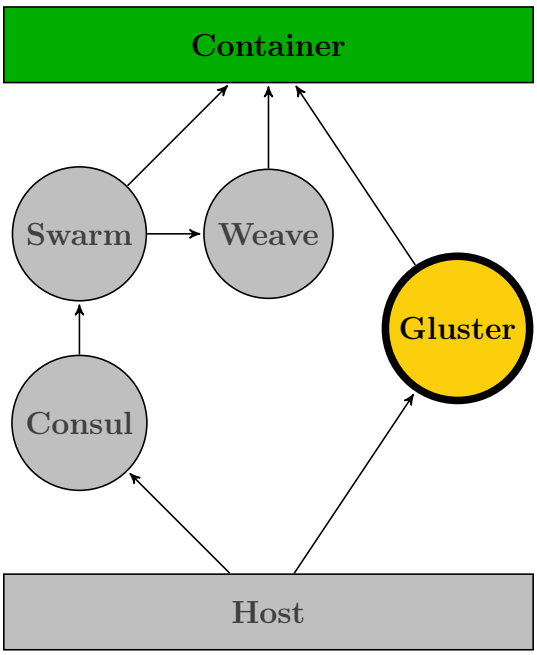
(b) **Consul: critical** Consul ist in den Status *critical* übergegangen, daher ist auch Swarm *critical*. Die direkt abhängigen Services Weave Net und die Container Applikation sind unter Umständen noch funktionsfähig.

Abbildung 3.3.: Änderung des Status mit Bezug auf Abhängigkeiten von Consul im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung von Consul.

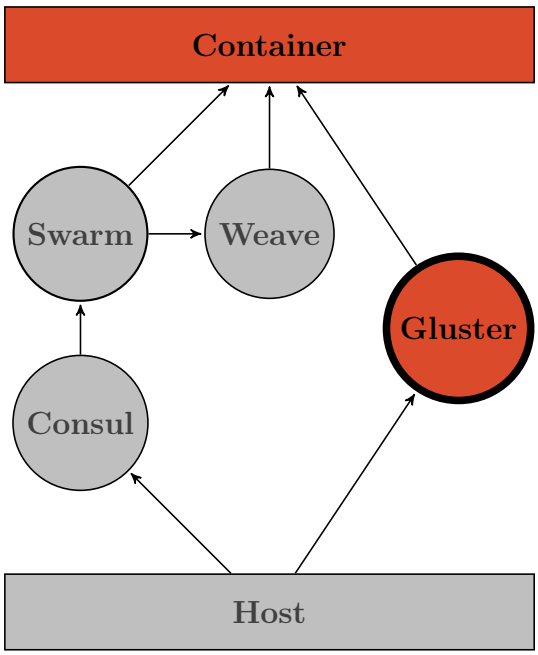
Dateisystem einzuhängen. Ob ein Volume ordentlich eingehängt wurde sollte durch eine Abfrage der Mounttabelle und durch einen Schreib- und Lesezugriff überprüft werden. Dies sind ebenfalls kritischere Werte, da ein nicht eingehängtes und beschreibbares Laufwerk zu einem nicht nutzbare *Cluster* führt.

GlusterFS führt eine Liste mit Dateien, die nicht synchronisiert werden konnten, um die Daten nach einem Ausfall replizieren zu können. Es hat sich bewährt, diese Liste zu überwachen, da zu erkennen ist, ob die Synchronisierung Fehlern unterliegt und ob ein ausgefallener Node ordentlich repliziert wurde. Sind solche nicht replizierten Dateien vorhanden deutet es auf eine fehlerhafte Synchronisierung hin. Jedoch fällt da *Cluster* nicht vollständig aus, daher wird im Falle des auftretens der Status von *healthy* auf *warning* gesetzt.

Im replizierten Modus von GlusterFS werden die Daten synchron geschrieben, somit beeinflussen Netzwerklatenzen und IO-Latenzen auf den jeweiligen Hosts beeinflussen die Performance.



(a) **Gluster: warning** Gluster funtioniert noch, jedoch müssen Abstriche bei der Verfügbarkeit oder Schreibgeschwindigkeit gemacht werden.



(b) **Gluster: critical** Auswirklungen von Gluster im kritischen Status. Gluster ist nicht mehr funktionsfähig, auch die Anwendungen in den Containern sind betroffen.

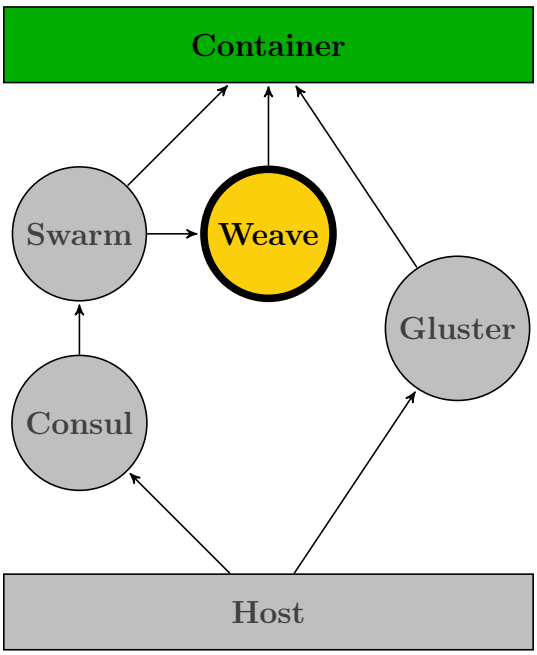
Abbildung 3.4.: Änderung des Status mit Bezug auf Abhängigkeiten von Gluster im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung von Gluster.

3.2.3. Weave Net Metriken

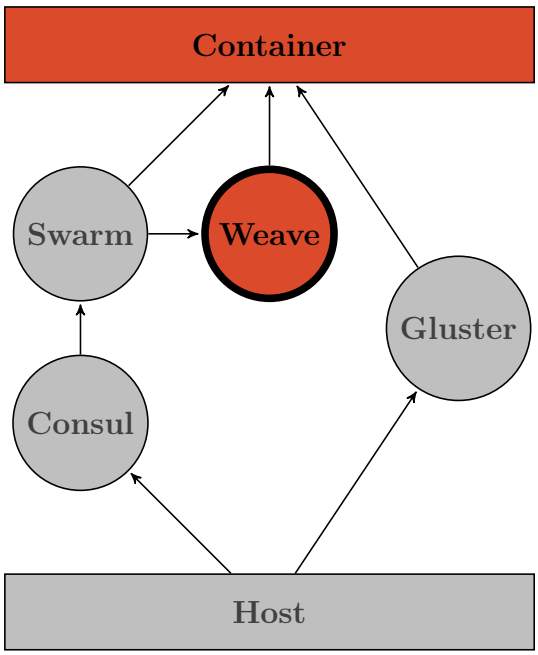
Weave Net als virtuelles Container Netzwerk ist dafür zuständig, dass die in Docker Swarm laufenden Container untereinander verschlüsselt kommunizieren können. Um dies zu gewährleisten sollten die Weave Container untereinander eine ständige Verbindung haben. Eine ständig abbrechende Verbindung ist ein kritischer Fehler, da dies die Kommuntikation der Container für den fehlschlagenden *Host* verhindert. Erfolgt ein (neuer) Verbindungsaufbau, ist dies kein kritischer aber auch kein gesunder Zustand.

3.2.4. Host Metriken

Die Metriken eines Hostsystems geben Auskunft darüber, in welchem Zustand es sich aktuell befindet. So kann zum Beispiel am Load abgelesen werden, wieviele Prozesse sich aktuell unverarbeitet in der Queue befinden. Davon ist unabhängig, ob gerade noch auf Rückmeldung gewartet wird oder ob der Prozessor vollständig ausgelastet ist. Ein kurzfristig hoher Load ist keine Seltenheit beim Start von Programmen. Auf den meisten Linux Systemen ist der Load in einem Mittel von 1, 5 und 15 Minuten über den Befehl



(a) **Weave: warning** bedeutet, dass Verbindungen nicht stabil sind, oder wieder verbunden wurde. Die Container können untereinander und nach außen kommunizieren.



(b) **Weave: critical**, die Docker Container können nicht mehr miteinander kommunizieren. Dadurch werden die meisten Services fehlschlagen. Je nach Konfiguration sind die Services von außerhalb des Weave Nets erreichbar.

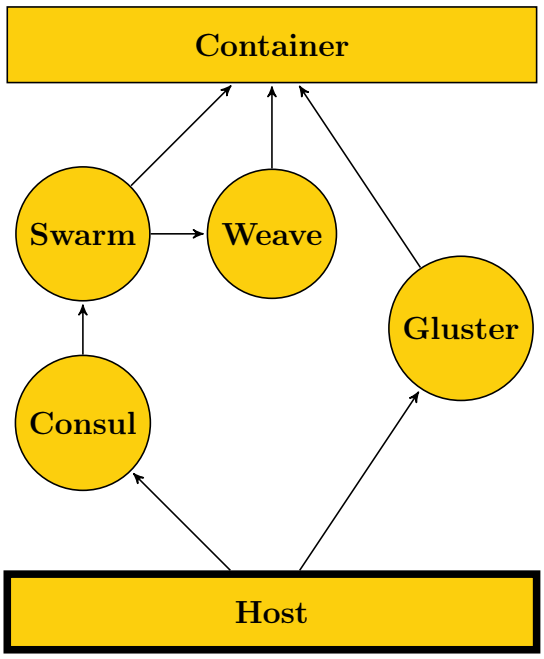
Abbildung 3.5.: Änderung des Status mit Bezug auf Abhängigkeiten von Weave im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung von Weave.

`cat /proc/loadavg` abfragbar. Eine allgemeine Beurteilung des Loads ist nicht möglich, da es sehr stark auf den Anwendungsfall des Systems ankommt. Ein System, das auf die möglichst effektive Abarbeitung einer Queue ausgelegt ist, wäre mit einem langfristigen Load kleiner 1 nicht optimal ausgelastet. Soll das System hingegen möglichst responsiv auf Nutzeranfragen reagieren, sollte es nicht ständig ausgelastet sein um auch bei erhöhtem Anfrageaufkommen eine schnelle Antwortzeit zu gewährleisten.

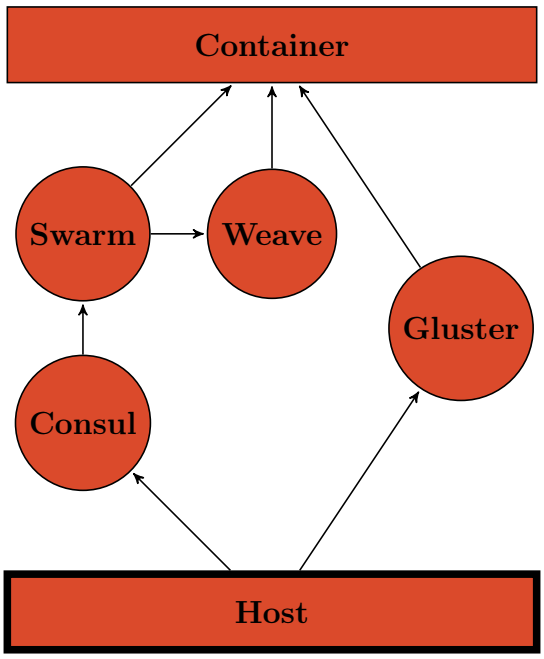
Die Verkaufsplattform für Veranstaltungstickets ist eine responsive Anwendung. In diesem Rahmen wurde bei der Leomedia GmbH ein Load über 15 Minuten von < 1 für den Zustand *healthy* definiert. Der Zustand für *warning* wurde auf < 2 definiert, ein höherer Load wird als *critical* gewertet.

Ein weiterer wichtiger Faktor ist der verfügbare Hauptspeicher. Ist dieser voll belegt beginnt das Betriebssystem zu swappen. Die Werte können aus `/proc/meminfo` ausgelesen werden. Die Dokumentation sind in der Manpage zu `free`[1] zu finden. Als Grenzwerte für verfügbaren Hauptspeicher hat sich in der Praxis ein Schwellwert für eine Warnung

[1] `free` ist ein Linux Programm das Informationen zum aktuellen Speicherverbrauch ausgibt.



(a) **Host: warning** alle Services sind betroffen.



(b) **Host: critical**, alle Services sind betroffen.

Abbildung 3.6.: Änderung des Status mit Bezug auf Abhängigkeiten des Hosts im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung des Hosts.

von $< 15\%$ bewährt. Sinkt dieser Wert auf 0 so werden Daten vom Hauptspeicher ununterbrochen auf die Festplatte ausgelagert, das hat eine hohe Auswirkung auf die Performance des Dateisystems und drosselt das System im Gesamten. Beim auslagern wird die Performance des gesamten Systems beeinträchtigt, daher hat sich bei dem Testcluster ein Wert von $< 5\%$ für einen krititschen Status bewährt.

3.3. Implementierungsziele: Panopticon-Monitoring

Ziel des nächsten Kapitels ist die Umsetzung eines Tools, dass das eben beschriebene Konzept verwirklicht. Panopticon-Monitoring ist ein Tool, mit dem der Clusterstatus betrachtet werden kann. Das besondere daran ist, dass mit einem Blick der Status des Clusters ersichtlich sein soll. Ähnlich einem Panopticon [2] soll es Panopticon-Monitoring ermöglichen mit einem Blick die gesamte Infrastruktur zu überwachen und eine Aussage darüber zu treffen, in welchem Zustand sich da *Cluster* befindet. Dafür wurden in *Abschnitt 3.1* die Signalfarben für die jeweiligen Zustände eingeführt.

Panopticon-Monitoring gewinnt die Metriken zur Bewertung aus Prometheus. Basierend

[2] Panopticon in der Wikipedia *https://de.wikipedia.org/w/index.php?title=Panopticon&oldid=162080055*

auf dem Testcluster wurde Panopticon-Monitoring als Prototyp für einen unmittelbar erkennbaren Clusterstatus entwickelt.

Nach Aufstellung des Konzeptes muss sich dieses auch in der Praxis beweisen.

- Bietet das Tool Panopticon-Monitoring eine schnelles Erkennen des Clusterstatus?
- Gibt es noch andere Metriken die interessant sind?

Die aufgeführten Ziele für die Umsetzung werden in *Kapitel 5* aufgegriffen und mit Blick auf die in *Kapitel 4* beschriebene Umsetzung betrachtet.

4. Umsetzung

4.1. Testcluster mit Docker Swarm, Consul und GlusterFS

Im Folgenden wird auf die Installation und Konfiguration des Testclusters eingegangen. Das *Cluster* umfasst vier Knoten, die durch physikalische Hosts repräsentiert werden. In *Tabelle 4.1* werden die technischen Daten aufgelistet.

Name	Lenovo ThinkCenter
Prozessor	Intel® Core 2 Duo®CPU E6550 @ 2.33GHz
RAM	2 GB
HDD	74.5 GB
Netzwerk	1 Gbit/s

Tabelle 4.1.: Hardwarekonfiguration der Knoten für das Testcluster

Debian Jessie[1] kommt als Betriebssystem zum Einsatz. Die weitere Software wurde sofern notwendig oder möglich als 3rd Party Quelle hinzugefügt. Docker ist in der Version 1.12.5 installiert. Um Docker im *Cluster* und mit Docker Swarm zu betreiben muss ein verteilter Key-Value Store verfügbar sein. Für das Testcluster wurde Consul ausgewählt. Docker kommuniziert über die Ports 2375 um mit der Docker Engine zu kommunizieren und 4000 um Docker Swarm zu administrieren. Für beide wird eine Authentifizierung via Client Zertifikaten durchgeführt. Consul ist in der Version 0.6.4 installiert und wurde als binary Release von HashiCorp heruntergeladen und für das Testcluster konfiguriert. Die Consul Nodes kommunizieren direkt miteinander, und wurden per IP auf der Firewall explizit für die Kommunikation freigegeben. Das Webinterface wurde so konfiguriert, dass es nur über einen Nginx Reverse Proxy auf Port 8501 zugänglich ist. Dieser ist nur für bestimmte vertrauenswürdige IP Adressen freigegeben. Zusätzlich muss sich ein Client mit einem TLS Zertifikat, welches von der Cluster-CA ausgestellt wird, authentifizieren. GlusterFS benutzt das Repository der Version 3.8, d.h. alle Aktualisierungen in dieser Version werden bei einer Systemaktualisierung ebenfalls mitberücksichtigt. Damit die Gluster Peers miteinander kommunizieren können wurden die Ports für die Kommunikation des Gluster Daemons (24007) und die synchronisation der Bricks (49152,49153) für jeden Knoten freigegeben sowie der Port 111 für das *Cluster* Port mapping. Weave Net ist in der Version 1.8.0 installiert. Um Kontrollnachrichten zu

[1]Release Webseite von Debian Jessie *https://www.debian.org/releases/jessie/*

senden verwendet es den TCP Port 6783 und die Daten des virtuellen Containernetzes werden über UDP auf den Ports 6783 und 6784 gesendet und empfangen.

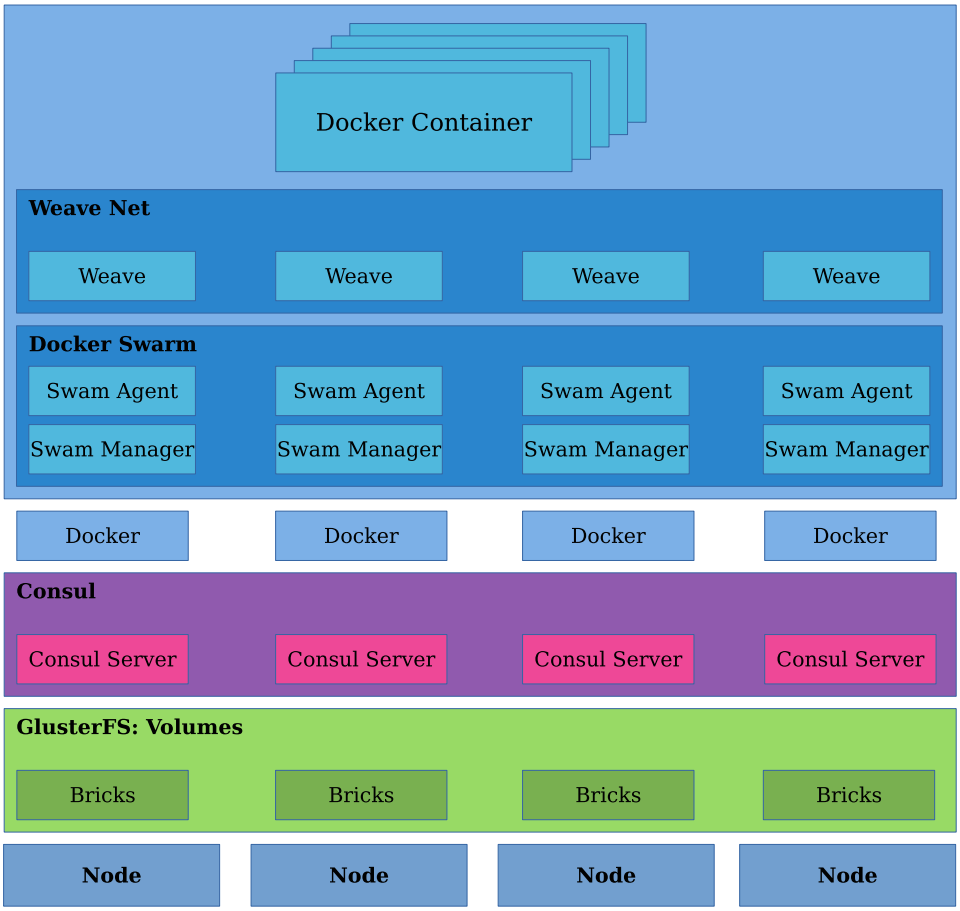


Abbildung 4.1.: Schematische Darstellung des Testclusters mit den Komponenten Docker, Docker Swarm, Weave Net, Consul, Gluster und den Hardware Nodes

Auf allen Knoten im Testcluster wurde der Gluster Daemon installiert, damit die Bricks auf alle Knoten verteilt sind. Die Volumes wurden überall eingehängt, um die Daten auf allen Hosts verfügbar zu haben. Consul wurde ebenfalls auf allen Knoten als Server installiert und Docker Swarm greift als Agent darauf zu.

Für alle Nodes im Testcluster wurden die Komponenten als Client und als Server installiert und betrieben, die geringe Größe des Clusters (drei bzw. vier Nodes) bedingt diese Entscheidung. Dadurch ist gewährleistet, das bei einem Ausfall eines Containers alle Daten auf allen Nodes verteilt sind und ein Container unabhängig der Nodes gestartet werden kann.

Wie in *Abschnitt 2.4* beschrieben, wurde für die Verschlüsselung der internen Verbindungen des Clusters HTTPS eingesetzt und mit Client Zertifikaten die Authorisierung geprüft. Um sicherzustellen, dass nur authorisierte Personen und Applikationen auf die Kommunikation von Consul und Docker zugreifen können wurde diese mit Hilfe von Client Zertifikaten abgesichert. Die Kommunikation im virtuellen Container Netzwerk

Weave Net wurde so konfiguriert, dass diese vollständig verschlüsselt abläuft.

Um Metriken auszulesen und diese im Prometheus-Format verfügbar zu machen, wurden *Exporter* eingesetzt. Verfügbare *Exporter* können in der Dokumentation zu Prometheus eingesehen werden[2]. Jeder *Exporter* hat einen zugewiesenen Port unter dem die Metriken via HTTP verfügbar gemacht werden. Damit es keine Überschneidungen untereinander gibt, ist eine Liste mit Ports (als Empfehlung) in der Prometheus Dokumentation[3] zu finden. Für Consul wurde auf den Consul Exporter[4] zurückgegriffen. Dieser bietet die in *Unterabschnitt 3.2.1* beschrieben Metriken um den Status von Consul auszuwerten. Die Metriken der Hosts selbst sind mittels Node Exporter[5] zugänglich gemacht worden, um Load und Memory, wie in *Unterabschnitt 3.2.4* beschrieben, zu überwachen. Ausschließlich für GlusterFS war kein *Exporter* verfügbar. Dieser wurde im Rahmen der Bachelorarbeit erstellt und ist nun als kleiner Beitrag für die Community unter der Apache 2.0 Lizenz auf Github verfügbar[6]. Im Anhang Abschnitt A.1 findet sich ein Beispiel zu den Metriken des Gluster Exporters. Weave implementiert die Instrumentierung der Metriken im Prometheus-Format.

4.2. Evaluation Metriken

Die in *Abschnitt 3.2* vorgestellten Metriken werden an dieser Stelle genauer erklärt. Ebenfalls wird auf die Abfragen aus Prometheus eingegangen.

In den jeweiligen Abschnitten werden die entsprechenden Metriken beschrieben inklusive eines Hinweises zu den Grenzwerten. Alle Metriken werden auf allen beteiligten Hosts abgefragt und in Prometheus gespeichert.

4.2.1. Allgemeine Metriken

Einige Kennzahlen sind für alle *Exporter* verfügbar und geben Aufschluss über die Funktion des jeweiligen Exporters.

`up` Zeigt an ob Prometheus den *Exporter* erreicht hat. Der Wert dieser *Metrik* ist 0 oder 1.

`*_up` Der * steht für den Namen des jeweiligen Exporters. Diese *Metrik* zeigt an, ob dieser eine Abfrage an die Komponente ausführen konnte und ob eine Rückmeldung erfolgte. Der Wert dieser *Metrik* ist 0 oder 1. Im Kontext des Testclustes bieten einzig der Consul Exporter und der Gluster Exporter diese *Metrik* an.

[2]Liste von Exportern
https://github.com/prometheus/docs/blob/master/content/docs/instrumenting/exporters.md
[3]Port Zuweisungen
https://github.com/prometheus/prometheus/wiki/Default-port-allocations
[4]Consul Exporter *https://github.com/prometheus/consul_exporter*
[5]Node Exporter *https://github.com/prometheus/node_exporter*
[6]Gluster Exporter *https://github.com/ofesseler/gluster_exporter*

4.2.2. Host mit Node Exporter und cAdvisor überwachen

Die in *Unterabschnitt 3.2.4* beschriebenen Metriken des Hostsystems werden durch den Node Exporter für Prometheus zugänglich gemacht. Erreichbar ist dieser jeweils unter der Url `https://[hostname]:9000/node`. Der Platzhalter `[hostname]` ist durch jeweiligen Hostnamen zu ersetzen.

`node_load15` ist der Mittelwert des Systemloads über die letzten 15 Minuten. Wie in *Unterabschnitt 3.2.4* beschrieben, kann die Last auf dem *Host* abgelesen werden. Der Wert ist ein Gleitkommawert (`float64`).

`machine_cpu_cores` ist eine *Metrik* aus cAdvisor und beinhaltet die Anzahl der auf dem System installierten Prozessorkerne. Dies ist für die Interpretation der Load-Metrik wichtig.

`node_memory_MemTotal` gibt die Größe des verfügbaren Hauptspeichers in Byte auf dem abgefragten Host wieder. Ausgegeben wid dies als Gleitkommawert ()`float`).

`node_memory_MemAvailable` verfügbarer Hauptspeicher der vom Betriebssystem für Anwendungen genutzt werden kann. Tendiert dieser gegen 0, wird das Betriebssystem zuerst versuchen aktuell unbenutzte Speicherbereiche auf den Swap auszulagern. Dies bedeutet gleichzeitig, dass das System an der oberen Auslastungsgrenze seiner Resourcen arbeitet.

4.2.3. Consul Exporter

Die Metriken aus Consul werden vom Consul Exporter über die Consul HTTP API abgerufen und im Prometheus-Format unter der Url `https://[hostname]:9000/node` für jeden *Host* verfügbar gemacht.

`consul_health_node_status` ist der von Consul bewertete Status über die Gesundheit des Serf Hosts. Serf ist das Gossip Network Protokoll über das Consul kommuniziert. Die Werte sind 0 oder 1 und sind vom Datentyp `int`.

`consul_raft_leader` sagt aus, ob es aus Sicht des angefragten Hosts einen Leader gibt. Zu Beginn der Bachelorarbeit wurde dieser Wert vom Consul Exporter nicht erfasst, aber als wichtig für den Clusterstatus erkannt, da ein nicht vorhandener Leader für das Testcluster *critical* ist. Diese *Metrik* wurde in einem Merge Request vom 15.11.2016 hinzugefügt[7].

`consul_raft_peers` stellt die Anzahl der im Raft *Cluster* registrierten Hosts dar. Eine Beschreibung zu Raft findet sich unter *Exkurs 1.*

[7]Merge Request auf Github *https://github.com/prometheus/consul_exporter/pull/36*

4.2.4. Weave Exporter

Für das virtuelle Container Netzwerk Weave Net ist es notwendig, dass immer eine Verbindung zwischen den Peers aufrechterhalten wird. Schlägt die Verbindung fehl, ist die Funktionsfähigkeit des Clusters nicht gegeben.

`weave_connections` stellt die Anzahl der Peer-to-Peer Verbindungen dar. Im Setup von der Leomedia GmbH mit einem *Cluster* von 4 im Weave Net betriebenen Hosts sollte die Anzahl der bestehenden Verbindungen stets 3 betragen. Diese *Metrik* ist durch das Label `state` unterteilt in die Bereiche: *connecting*, *established*, *failed*, *pending* und *retrying*. Diese sollten, mit Ausnahme von *established* immer den Wert 0 enthalten.

4.2.5. Gluster Exporter

GlusterFS hatte zu Beginn der Bachelorarbeit noch keine Instrumentierung für Prometheus. Daher wurde im Rahmen dieser Bachelorarbeit ein *Exporter* für GlusterFS implementiert[8]. Dieser ist in Go implementiert und instrumentiert die Informationen, die über das Gluster CLI abrufbar sind. Gluster bietet in dieser Version keine API an, über die Monitoring-Informationen abgerufen werden können, daher war es notwendig diese über das *CLI* auszulesen.

Ein Problem dabei ist, dass jede Anfrage an die Gluster *CLI* immer im gesamten Glusterverbund ausgeführt wird. Daher erhöhen sich die Latenzen im Fehlerfall bis zu einem internen Timeout. Dadurch kommt es zu sehr hohen Latenzen ($\approx 1 - 30$ sec.), die es notwendig machen in Prometheus eine hohe Timeout- und Abfragezeit anzugeben (> 40 sec.).

Prometheus liefert einige Bibliotheken mit, um eigene *Exporter* zu implementieren unter anderem Java, Python und Go. Einer der Vorteile von *Go* ist, dass beim Build eine Binary erstellt wird, die ohne weitere Abhängigkeiten auf dem Zielsystem deployed werden kann. Diese Eigenschaft wurde bei der Erörterung der Programmiersprache als entscheidender Vorteil gesehen und damit fiel die Wahl auf *Go*.

Die wichtigsten Metriken, die sich aus der Gluster *CLI* ableiten lassen werden im Folgenden beschrieben.

`gluster_peers_connected` Die mit dem abgefragten Peer verbundenen Peers werden in der *Metrik* `gluster_peers_connected` angegeben. Bei vier Gluster Hosts sind drei Verbindungen zu erwarten.

`gluster_mount_successful` Gibt an, ob das Einhängen des Gluster Volumes erfolgreich war. Geprüft wird dies indem der Befehl `mount -t fuse.glusterfs` ausgeführt wird. Dieser gibt alle eingehängten Laufwerke und Verzeichnisse nach dem Typ `fuse.glusterfs` gefiltert aus. Ist das Volume gemountet, ist der Wert 1, ansonten bleibt er 0.

[8] Github: Gluster Exporter *https://github.com/ofesseler/gluster_exporter*

`gluster_volume_writeable` Ist das Volume duch den Gluster Exporter beschreibbar, ist der Wert 1. Dies wird geprüft indem ein `os.Create(FILENAME)` mit anschließendem `os.Remove(FILENAME)` ausgeführt wird. Sind beide Kommandos erfolgreich, kann davon ausgegangen werden, dass das Volume beschreibbar ist.

`gluster_heal_info_file_count` Steigt dieser Wert über 0, dann funktioniert die Replikation zwischen den Bricks nicht mehr. Es ist ein sehr kritischer Wert, der durch Überlastung oder Netzwerkprobleme plötzlich steigen kann. Über den Self-Heal Mechanismus können die Daten wieder auf alle Hosts verteilt werden.

Damit ist die Beschreibung der Metriken abgeschlossen. Es folgt die Vorstellung des Tools Panopticon-Monitoring.

4.3. Tool: Panopticon-Monitoring

Panopticon-Monitoring ist ein Tool, dass die einzelnen Komponenten des Clusters bewertet und dessen aktuellen Status darstellt[9]. Es lassen sich verschiedene Dienste konfigurieren, die in die Bewertung mit aufgenommen werden sollen. Dargestellt wird dann ein aggregierter Status aus den konfigurierten Services, der entweder „grün“, „orange“ oder „rot“ sein kann.

Der erste wichtige Schritt bestand darin, geeignete Metriken zu finden um eine Aussage über den Status des Clusters und der Komponeten zu geben.

Es wurde ein iteratives Vorgehen gewählt, um schnellstmöglich ein funktionsfähiges Tool zu erhalten. Dieses Vorgehen ist in die folgenden Schritte unterteilt:

1. Abfragen der Prometheus *API* ✔
2. Auswertung der Services ✔
3. Aggregierung der Auswertung zu einem Gesamtergebnis ✔
4. Optimierungen ✗

In den folgenden Abschnitten werden die oben aufgelisteten Implementierungsziele diskutiert. Dabei konnte der letzte Punkt der Auflistung im Rahmen der Arbeit nicht umgesetzt werden. In *Unterabschnitt 4.3.2* werden aber die Ansätze für zukünftige Erweiterungen besprochen.

Die automatischen Tests wurden hauptsächlich auf Funktionen angewandt, die später den Clusterstatus prüfen. Dadurch kann sichergestellt werden, dass auch Änderungen an der Bewertung getestet werden und zum erwarteten Ergebnis führen.

[9] Der aktuelle Stand kann unter *https://github.com/ofesseler/panopticon-monitoring* abgerufen werden. Abgegeben wurde das Tool auf der beiliegenden CD im Ordner `panopticon-monitoring`.

4.3.1. Implementierung

Panopticon-Monitoring ist ein Tool zur Auswertung von Prometheus Metriken mit Bezug auf das Test- und Entwicklungscluster der Leomedia GmbH. Die Übersicht wird über den Browser aufgerufen, woraufhin die Daten von Prometheus angefragt und verarbeitet werden.

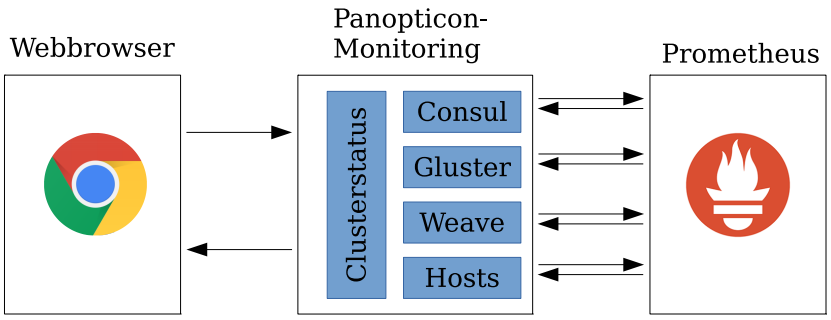


Abbildung 4.2.: Schematische Darstellung von Panopticon-Monitoring

In *Abbildung 4.2* ist ersichtlich, wie der Zugriff vom Webbrowser über Panopticon-Monitoring zu Prometheus vor sich geht. Die Anfrage wird vom Webbrowser/Nutzer gesteuert und wird direkt an Panopticon-Monitoring gesendet. Dieses erstellt eigene Anfragen an Prometheus um Daten zu Gluster, Consul, Weave und den Hostsystemen zu erlangen. Die einzelnen Daten werden, wie in *Abschnitt 3.1* eingeführt, zu einem Clusterstatus aggregiert.

Dadurch ist es möglich den Status des Clusters so darzustellen, dass dieser auf einen Blick zu erfassen ist.

Abfragen der Prometheus HTTP API

Um an die Informationen von Prometheus zu gelangen, muss dessen *API* abgefragt werden. Prometheus bietet eine HTTP API, die es ermöglicht einzelne Metriken, sowie Funktionen und Aggregationen dieser abzufragen. Für Panopticon-Monitoring wurden ausschließlich die einzelnen Metriken abgefragt, da Prometheus die Abhängigkeiten im *Cluster* nicht kennt.

Die Implementierung der Abfrage wurde mit Interfaces realisiert. Dadurch ist es sehr einfach möglich, Anpassungen durchzuführen, zum Beispiel Änderungen der Prometheus API. Ein weiterer Vorteil ist die einfache Testbarkeit der Funktionen und Methoden. Mittels Interface kann ein *Mock-Objekt* erstellt werden, das es ermöglicht Methoden zu testen, die externe Abhängigkeiten aufweisen. Im Modul `promapi` wurde das Interface `Fetcher` erstellt. Damit die Funktion zur Abfrage einfach ausgetauscht werden kann, wurde diese mittels Interfaces implementiert. Dadurch ergibt sich zusätzlich eine einfache Testbarkeit der aufrufenden Funktionen [6]. Die Implementierung eines solchen Interfaces, des `type` und der Funktion ist in *Listing 4.1* exemplarisch dargestellt.

```go
// Fetcher is interface for queries to Prometheus, mainly implemented for testing
type Fetcher interface {
  PromQuery(query string, host string) (StatusCheckReceived, error)
}

// PrometheusFetcher is implementation type for Fetcher
type PrometheusFetcher struct {
}

// PromQuery implementation with PrometheusFetcher to issue queries to Prometheus
  HTTP API
func (PrometheusFetcher) PromQuery(query, promHost string) (StatusCheckReceived,
  error) {
  // Anfrage an die Prometheus HTTP API
}
```

Listing 4.1: Fetcher Interface und die zugehörige Implementierung des `type` Prometheus-Fetcher in der Methode PromQuery

Um die Methode `PromQuery` als *Mock-Objekt* zu schreiben muss ein neuer `type` implementiert werden und eine Funktion, welche die gleiche Signatur besitzt, wie im Interface `Fetcher` beschrieben. Damit können alle Methoden, die das Interface `Fetcher` implementieren anstatt der eigentlichen Implementierung aufgerufen werden wie im Test für `PromQuery` in *Listing 4.2* aufgeführt.

```go
type ConsulTest struct {
  Total        int
  Failed       int
  SuccessValue string
  FailValue    string
}

func (f ConsulTest) PromQuery(query, promHost string) (StatusCheckReceived, error) {
 // Mock Implementierung der PromQuery Funktion
}
```

Listing 4.2: Implementierung der Mock Funktion PromQuery

In der Applikation wird `PromQuery` nur indirekt durch `Fetch*`-Funktionen aufgerufen, wobei der Stern in diesem Falle für mögliche Namen steht. Diese Funktionen geben ein `type PRomQR{}` zurück, welches aus den Metrikdaten befüllt wird. Dies beinhaltet die notwendigen Informationen, um den Clusterstatus zu erhalten. Der `type PRomQR{}` wurde wie in *Listing 4.3* implementiert.

```go
type PromQR struct {
  Name     string
  Job      string
  Node     string
  Value    int64
  Instance string
}
```

Listing 4.3: type PRomQR

Das Modul `promapi` ist eigenständig verwendbar, daher kann diese auch in anderen Projekten verwendet werden. Im Zuge der Bachelorarbeit wurde die Modularität dazu genutzt, um schnell Nebenprojekte und kleine Versuche mit der Prometheus API zu entwickeln.

Entscheidungsfindung und Aggregierung des Clusterstatus

Nach Abfragen der Prometheus HTTP API werden die Antworten verglichen und je nach abgefragter *Metrik* findet die Bewertung unterschiedlich statt. Am Beispiel der *Metrik* `consul_raft_peers` wird in *Listing 4.4* die Bewertung dargestellt. Dabei wird geprüft, ob es mit der Anzahl Peers noch möglich ist ein *Quorum* zu erhalten $((N/2)+1)$.

```
1  func computeCountersFromPromQRs(r Rate, reference int, promQRs []api.PromQR) api.
       ClusterStatus {
2    // vars ...
3    for _, promQR := range promQRs {
4      // Die Funktion Rater() prueft gegen eine Referenz, ob ein  \Gls{quorum}erlangt
       werden kann.
5      computedStatus, err := r.Rater(promQR, reference)
6      if err != nil {
7        log.Error(err)
8      }
9      switch computedStatus {
10     case api.HEALTHY:
11       hCounter++
12     case api.WARNING:
13       wCounter++
14     case api.CRITICAL:
15       cCounter++
16     }
17   }
18   if hCounter == reference {
19     status = api.HEALTHY
20   } else if wCounter >= (reference/2)+1 || hCounter >= (reference/2)+1 {
21     status = api.WARNING
22   } else if cCounter >= (reference / 2) {
23     status = api.CRITICAL
24   }
25   return status
26 }
```

Listing 4.4: Bewertung der Metrik `consul_raft_peers`

Die Entscheidung für alle anderen Metriken verläuft nach dem gleichen Prinzip. Sobald alle Auswertungen abgeschlossen sind, werden diese zu einem Gesamtzustand aggregiert. Schematisch ist dies im Entscheidungsbaum in *Abbildung 4.3* dargestellt. Somit ist in allen Zwischenebenen immer ein bewerteter Status vorhanden, welcher dazu verwendet werden kann, einen Fehler einzugrenzen.

4.3.2. Optimierungen

Die Funktionalität von Panopticon-Monitoring wurde im Rahmen dieser Arbeit implementiert. Weitere mögliche Schritte um das Tool zu verbessern werden in den folgenden Abschnitten vorgestellt. Zum Einen kann die Darstellung verbessert werden und zum Anderen würde die Verarbeitungsgeschwindigkeit durch paralleles Abfragen der Prometheus Zugriffe erhöht werden.

Darstellung der Zusammenhänge Für eine Weiterentwicklung wäre es denkbar, eine Darstellung der Zusammenhänge zu implementieren. Dabei bietet sich das Schema aus

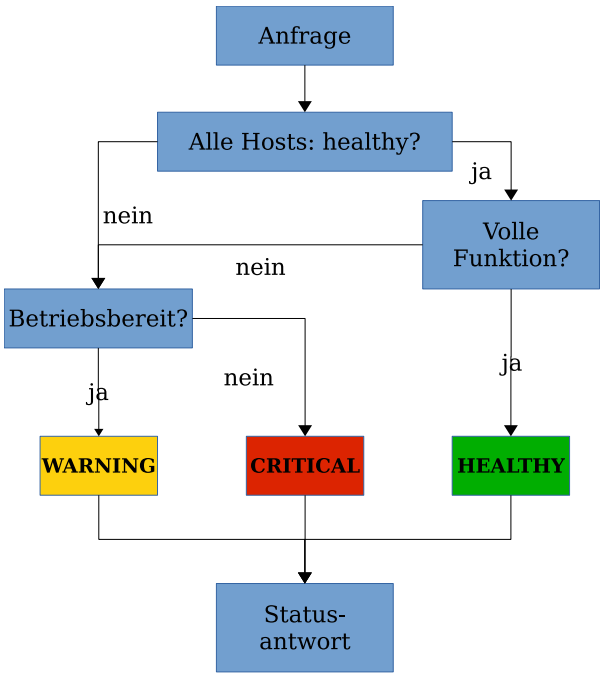


Abbildung 4.3.: Darstellung des Entscheidungsbaumes für den Clusterstatus

Abbildung 3.3a an. Dadurch kann eine Prognose für einen auftretenden Fehler vereinfacht werden.

Parallelisierung der Clientzugriffe auf Prometheus Für jede Abfrage des Clusterstatus werden aktuell zehn Anfragen an Prometheus gesendet um von den unterschiedlichen Metriken auf den Status zu schließen. Um eine schnellstmögliche Antwort durch Panopticon-Monitoring zu bekommen, ist eine Möglichkeit alle Client Zugriffe auf Prometheus zu parallelisieren. In diesem Fall bezeichnet Parallelisierung Nebenläufigkeit. Dabei werden einzelne Threads nicht parallel ausgeführt, sondern je nach Zuteilung der CPU Zeit nacheinander, sodass während eines IO-Wait eine andere Aufgabe abgearbeitet werden kann. Im Englischen hat sich der Begriff *Concurrency* für diese Art von Parallelisierung eingebürgert, er kann mit dem deutschen Wort *Nebenläufigkeit* sehr treffend übersetzt werden.

Go bietet mit *Goroutines*[10], *Channel Buffering*[11] und *Channel Synchronization*[12] kombinierbare Methoden zur einfachen Parallelisierung von Abläufen an. Somit kann mit einer *Goroutine* jede Funktion nebenläufig ausgeführt werden. Das Problem, dass nebenläufig ausgeführte Routinen ihren Rückgabewert an einen gemeinsam genutzen Speicher geben, löst Go mit den sogenannten Channels. Ein Channel ist eine Art Kanal in den Rückgabewerte aus Goroutines geschrieben werden können. Diese warten mit der weiteren Verarbeitung so lange, bis der Wert geschrieben wurde.

[10]Goroutines *https://gobyexample.com/goroutines*
[11]Channel Buffering *https://gobyexample.com/channel-buffering*
[12]Channel Synchronisazion *https://gobyexample.com/channel-synchronization*

Somit lassen sich mit relativ geringen Aufwand Fork-Join Operationen implementieren.

Jedoch birgt die Parallelisierung in Go trotz ihrer Einfachheit auch ein hohes Fehlerpotential in sich. Dies liegt vor allem an den durch die Parallelisierung auftretenden Nebeneffekten.

Daher ist im Rahmen der Implementierung vorerst darauf verzichtet worden die Zugriffe auf Prometheus zu parallelisieren.

5. Fazit und Ausblick

5.1. Bewertung der Umsetzung im Vergleich zur Konzeption

In *Abschnitt 3.3* wurden verschiedene Ziele für die Umsetzung aufgestellt. Im Folgenden wird bewertet, inwiefern diese erreicht werden konnten.

Alle in der Konzeption aufgeführten Metriken konnten für den Clusterstatus überwacht werden. Manche Parameter, wie der Load (*Unterabschnitt 3.2.4*) wurden aufgrund praktischer Erfahrungen festgesetzt und müssen weiterhin beobachtet und ggf. angepasst werden. Der in *Abschnitt 3.1* angedachte Zustandsautomat wurde nicht implementiert, da für jede Abfrage eine neue Beurteilung anfällt und kein Status persistiert wird. Vielmehr wird für jeden Teilstatus ein Entscheidungsbaum durchlaufen, wie in *Unterunterabschnitt 4.3.1* erläutert.

Das Tool Panopticon-Monitoring muss seinen Mehrwert in der Praxis erst noch unter Beweis stellen. Einen einfacheren Zugang zur aktuellen Gesundheit des Clusters bietet es in jedem Fall, da alle Werte zu einem Status aggregiert werden. Dadurch ist es nicht nötig verschiedene Werte zu betrachten um einen Fehler oder eine Ungereimtheit zu entdecken. Allerdings ist fraglich, ob die Form der Darstellung nicht noch weiter optimiert werden könnte. Eventuell wäre eine Integration in den AlertManager sinnvoll, damit im Falle einer Statusänderung unmittelbar alamiert wird. Aktuell ist immer ein Blick auf die Statusseite vonnöten.

Vor allem im Bereich der Performancemessung gibt es einige interessante Größen, welche für eine weitere Evaluation in Betracht gezogen werden könnten. Weiterhin wäre denkbar, die Monitoringdaten aus der Infrastruktur mit den Monitoringdaten der Applikation zu verbinden um ganz im Sinne von DevOps eine gesamtheitliche Sicht auf das *Cluster* zu ermöglichen.

Zusammenfassend lässt sich sagen, dass das wichtigste Ziel der Bachelorarbeit, die Auswahl, Beschreibung und Implementierung der Metriken erreicht werden konnte, auch wenn etwa in Bezug auf das Tool interessante Änderungen denkbar wären.

5.2. Verwendete Software

Im Rahmen der Bachelorarbeit ist bei der Recherche immer deutlicher geworden, dass Docker und Docker Swarm in den den Anfängen stecken. Das System entwickelt sich in rasanter Geschwindigkeit. Dadurch ist es sehr schwer einschätzbar, welche Features bereit für den produktiven Betrieb sind und bei welchen noch abgewartet werden sollte. Aus

vielen Blogbeiträgen[1] geht hervor, dass sich in kurzen Abständen größere Änderungen ergeben. Die neueste Version Docker 1.13 bringt zum Beispiel einige Veränderungen mit sich, die manche von den benutzten Applikationen überflüssig machen. So ist etwa Docker Swarm als Swarm mode seit 1.12 in Docker integriert und die Funktionalitäten von Consul werden mit 1.13 ebenfalls in Docker integriert. In Bezug auf die im Cluster verwendete Software lässt sich also vor allem feststellen, dass vor der Verwendung von Docker und Docker Swarm die hohe Änderungsrate bedacht werden sollte.

5.3. Ausblick

Das Tool in seinem derzeitigen Zustand ermöglicht einen schnellen Überblick über den Clusterstatus, allerdings kann die Lokalisierung von Fehlern aufwändig sein. Um beim Sammeln von Metriken mit Prometheus nicht nur Zahlenwerte zu protokollieren, wäre es sinnvoll, mit dem *ELK-Stack*[2] Logdaten an einem Ort zu sammeln. Damit sind diese von einem zentralen Ort aus zugänglich und können als Volltext durchsucht werden. Davon verspricht man sich eine schnellere Lokalisierung der Fehler.

Abschließend lässt sich zusammenfassen, dass die Leomedia GmbH mit ihrer Entscheidung, zu einer Microservices Architektur zu wechseln, einen zukunftsweisenden Schritt getan hat und dass das Monitoring sich in der zu Beginn angedachten Weise als angemessen erwiesen hat.

[1] Serverfault - Docker Swarm and Consul production setup recommendation *http://serverfault.com/questions/748040/docker-swarm-and-consul-production-setup-recommendation*

[2] Complete Guid ELK-Stack *http://logz.io/learn/complete-guide-elk-stack/*

Literatur

[1] Leslie Lamport. „Paxos Made Simple“. In: *SIGACT News* 32.4 (Dez. 2001), S. 51–58. ISSN: 0163-5700. DOI: *10.1145/568425.568433*. URL: *http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf*.

[2] Martin Fowler. *Microservices - A Definition of This New Architectural Term*. 25. März 2014. URL: *https://martinfowler.com/articles/microservices.html* (besucht am 26. 01. 2017).

[3] Diego Ongaro und John Ousterhout. „In Search of an Understandable Consensus Algorithm“. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, S. 305–319. URL: *https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro* (besucht am 10. 11. 2016).

[4] Martin Fowler und James Lewis. „Microservices: Nur Ein Weiteres Konzept in Der Softwarearchitektur Oder Mehr“. In: *Objektspektrum* 1.2015 (2015), S. 14–20. URL: *https://www.sigs-datacom.de/uploads/tx_dmjournals/fowler_lewis_OTS_Architekturen_15.pdf* (besucht am 28. 01. 2017).

[5] kernel.org. *FUSE - Userspace Filesystem - Kernel Documentation*. 29. Mai 2015. URL: *https://www.kernel.org/doc/Documentation/filesystems/fuse.txt* (besucht am 30. 01. 2017).

[6] Nathan LeClaire. *Interfaces and Composition for Effective Unit Testing in Golang*. 10. Okt. 2015. URL: *https://nathanleclaire.com/blog/2015/10/10/interfaces-and-composition-for-effective-unit-testing-in-golang/* (besucht am 24. 02. 2017).

[7] Jerry Preissler und Oliver Tigges. *Docker - Perfekte Verpackung von Microservices*. 9. Nov. 2015. URL: *https://www.innoq.com/de/articles/2015/11/docker-perfekte-verpackung-fuer-micro-services/* (besucht am 30. 01. 2017).

[8] Guido Steinacker. *Monolithen Und Microservices*. 2. Juni 2015. URL: *https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html* (besucht am 26. 01. 2017).

[9] Docker Inc. *Docker - the Open-Source Application Container Engine*. 2016. URL: *https://github.com/docker/docker/tree/v1.13.0* (besucht am 30. 01. 2017).

[10] Docker Inc. *Swarm: A Docker-Native Clustering System*. 8. Juli 2016. URL: *https://github.com/docker/swarm/tree/v1.2.6* (besucht am 30. 01. 2017).

[11] *GlusterFS Architecture*. 2016. URL: *https://github.com/gluster/glusterdocs/tree/8232306c91* (besucht am 12. 12. 2016).

[12] weaveworks. *Weave - Simple, Resilient Multi-Host Docker Networking and More*. 4. Nov. 2016. URL: *https://github.com/weaveworks/weave* (besucht am 30. 01. 2017).

[13] Wikipedia. *Monitoring Begriffserklärung*. In: *Wikipedia*. Page Version ID: 160566642. 12. Dez. 2016. URL: *https://de.wikipedia.org/w/index.php?title=Monitoring&oldid=160566642* (besucht am 23. 01. 2017).

[14] Viktor Farcic. In: *The DevOps 2.1 Toolkit: Docker Swarm*. LeanPub, 1. Nov. 2017. Kap. Collecting Metrics and Monitoring The Cluster. URL: *http://leanpub.com/the-devops-2-1-toolkit* (besucht am 13. 01. 2017).

[15] Viktor Farcic. In: *The DevOps 2.1 Toolkit: Docker Swarm*. LeanPub, 1. Nov. 2017. Kap. Monitoring Best Practices. URL: *http://leanpub.com/the-devops-2-1-toolkit* (besucht am 13. 01. 2017).

[16] Viktor Farcic. In: *The DevOps 2.1 Toolkit: Docker Swarm*. LeanPub, 1. Nov. 2017. Kap. What Now? URL: *http://leanpub.com/the-devops-2-1-toolkit* (besucht am 13. 01. 2017).

[17] Viktor Farcic. *The DevOps 2.1 Toolkit: Docker Swarm*. LeanPub, 1. Nov. 2017. 385 S. URL: *http://leanpub.com/the-devops-2-1-toolkit* (besucht am 13. 01. 2017).

[18] Wikipedia. „Docker (Software) — Wikipedia, Die Freie Enzyklopädie“. In: (2017). [Online; Stand 30. Januar 2017]. URL: *https://de.wikipedia.org/w/index.php?title=Docker_(Software)&oldid=162125187*.

Abbildungsverzeichnis

2.1. Monoliths and Microservices ©Martin Fowler [2] 5
2.2. Schematische Darstellung eines Weave Net Netzwerks in Docker Swarm . 8

3.1. Abhängigkeiten der Services im Docker Swarm Cluster 14
3.2. Zustandsdiagramm mit Zustandsübergängen der verschiedenen Clusterstatus 15
3.3. Änderung des Status mit Bezug auf Abhängigkeiten von Consul im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung von Consul. 16
3.4. Änderung des Status mit Bezug auf Abhängigkeiten von Gluster im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung von Gluster. 17
3.5. Änderung des Status mit Bezug auf Abhängigkeiten von Weave im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung von Weave. 18
3.6. Änderung des Status mit Bezug auf Abhängigkeiten des Hosts im Testcluster. Die eingefärbten Services sind abhängig von einer Statusänderung des Hosts. 19

4.1. Schematische Darstellung des Testclusters mit den Komponenten Docker, Docker Swarm, Weave Net, Consul, Gluster und den Hardware Nodes . . 22
4.2. Schematische Darstellung von Panopticon-Monitoring 27
4.3. Darstellung des Entscheidungsbaumes für den Clusterstatus 30

Tabellenverzeichnis

3.1. Je nach Funktionsfähigkeit wird der Status mit den Begriffen *healthy*, *warning* und *critical* beschrieben. 13

4.1. Hardwarekonfiguration der Knoten für das Testcluster 21

Listings

2.1. Schema eines Prometheus Abtastwerts 11

4.1. Fetcher Interface und die zugehörige Implementierung des `type` PrometheusFetcher in der Methode PromQuery 28
4.2. Implementierung der Mock Funktion PromQuery 28
4.3. type PRomQR 28
4.4. Bewertung der Metrik `consul_raft_peers` 29

A.1. Antwort des Gluster Exporters im Prometheus Format 42
A.2. Antwort des Gluster Exporters im Prometheus Format 43

Glossar

Certificate Authority ist eine Zertifizierungsstelle für digitale Zertifikate, wie in diesem Fall Transport Layer Security. Zum einen gibt es wirtschafts Unternehmen die solche Zertifikate ausstellen oder man kann selbst zur *CA* werden und Zertifikiate ausstellen. 41

Cluster ist eine Menge von Computern die miteinander agieren und sich als Gemeinschaft verstehen. Eine genauere Beschreibung befindet sich in *Abschnitt 2.2*. 2, 5, 7, 8, 12–16, 19, 21, 24, 25, 27, 32

ELK-Stack ist eine Zusammenstellung von Anwendungen bestehend aus der Suchmaschine Elasticsearch, der serverseitigen Datenverarbeitungspipeline Logstash und dem Elasticsearch Plugin für visualisierung Kibana[1]. 33

Exporter ist eine Applikation die Monitoringdaten einer anderen Applikation abfragt und diese im Prometheus-Format über HTTP verfügbar macht. 23, 25

Go ist eine statisch typisierte, freie open source Programmiersprache, die das einfache Erstellen von zuverlässiger und stabiler Software ermöglichen soll[2]. In den letzten Jahren gewinnt Go immer mehr an Aufmerksamkeit und ist in die Top 15 des Tiobe Indexes aufgestiegen[3]. 25

Host ist ein aus dem Englischen stammender Begriff (übers.: Wirt), der einen Computer oder Server, welcher mit zugehörigem Betriebssystem in einem Netzwerk eingebunden ist. *Host* wird oft als Abkürzung für Hostrechner/Hostcomputer benutzt[4]. 7–9, 17, 24, 39

Metrik ist ein Wert, der eine Eigenschaft in Form eines Zahlenwertes darstellt[5]. 11, 23–25, 29

Mock-Objekt ist in der Softwareentwicklung ein Platzhalter für ein echtes Objekt. Häufig werden so genannte Mock-Objekte in Unit-Tests eingesetzt um externe Abhängigkeiten zu modellieren, ohne diese verfügbar zu machen. 27, 28

[1] Überblick und Installation des ELK-Stack *http://logz.io/learn/complete-guide-elk-stack/*
[2] Golang Homepage *https://golang.org/*
[3] Tiobe Index *http://www.tiobe.com/tiobe-index/go/*
[4] *https://de.wikipedia.org/w/index.php?title=Hostrechner&oldid=162072939*
[5] *https://de.wikipedia.org/w/index.php?title=Softwaremetrik&oldid=160402025*

Quorum ist die minimale Anzahl an Stimmen, die eine verteilte Transaktion benötigt um erfolgreich ausgeführt zu werden. Hier: Sind genügend Nodes funktionsfähig um operabel zu sein. 15, 29

Akronyme

API Application Programming Interface. 6, 7, 10, 26, 27

CA Certificate Authority . 11, 39

CLI Command Line Interface. 7, 25

DNS Domain Name System. 9

FUSE Filesystem in Userspace. 9

HTTP Hypertext Transfer Protocol. 11

IP Internet Protokoll. 9

TSDB Time Series Database. 10

———————- Anhang ———————-

A. Anhang

A.1. Beispielhafte Darstellung der Metriken eines Exporters anhand des Gluster Exporters

Die Metriken werden in der Bachelorarbeit erklärt. An dieser Stelle werden zwei beispielhafte Zustände erörtert und anschließend ein Status für Gluster abgeleitet.

```
# HELP gluster_brick_count Number of bricks at last query.
# TYPE gluster_brick_count gauge
gluster_brick_count{volume="gv_data"} 3
# HELP gluster_exporter_build_info A metric with a constant '1' value labeled by version, revision, branch, and goversion from which gluster_exporter was built.
# TYPE gluster_exporter_build_info gauge
gluster_exporter_build_info{branch="dev",goversion="go1.7.4",revision="6745a7e9ed428ffb04991630bce935cd849c1096",version="0.2.5"} 1
# HELP gluster_peers_connected Is peer connected to gluster cluster.
# TYPE gluster_peers_connected gauge
gluster_peers_connected 2
# HELP gluster_up Was the last query of Gluster successful.
# TYPE gluster_up gauge
gluster_up 1
# HELP gluster_volume_status Status code of requested volume.
# TYPE gluster_volume_status gauge
gluster_volume_status{volume="gv_data"} 1
# HELP gluster_volumes_count How many volumes were up at the last query.
# TYPE gluster_volumes_count gauge
gluster_volumes_count 1
# HELP gluster_mount_successful Checks if mountpoint exists, returns 0 or 1
# TYPE gluster_mount_successful gauge
gluster_mount_successful{mountpoint="/mnt/data",volume="localhost:data"} 1
# HELP gluster_volume_writeable Writes and deletes file in Volume and checks if it is writeable
# TYPE gluster_volume_writeable gauge
gluster_volume_writeable{mountpoint="/mnt/data",volume="localhost:data"} 1
# HELP gluster_heal_info_files_count File count of files out of sync, when calling "gluster v heal VOLNAME info".
# TYPE gluster_heal_info_files_count gauge
gluster_heal_info_files_count{volume="localhost:data"} 0
```

Listing A.1: Antwort des Gluster Exporters im Prometheus Format

Die Anfrage an den Gluster Exporter gibt eine Auflistung der Metriken in Plaintext zurück. Exemplarisch wird nur die Ausgabe von einem Exporter dargestellt, um die Übersicht zu wahren. In der tatsächlichen Umgebung, werden alle Peers von Prometheus abgefragt.

In diesem Fall sind alle Metriken in einem sehr guten Zustand. Aus den in Listing A.1 dargestellten Metriken kann folgendes abgelesen werden:

- Die Bricks sind auf 3 Peers verteilt

- Es besteht eine Verbindung zu 2 Peers
- Die Gluster CLI kann erfolgreich abgefragt werden
- Gluster gibt an, dass der Status des Volumes `gv_data` 1 ist
- Es gibt ein Volume in diesem GlusterFS
- Der Mountpoint taucht in der Liste der gemounteten Dateisysteme auf.
- Das Volume ist beschreibbar
- Es liegen keine Synchronisationsfehler vor.

Nachfolgend wird ein Zustand der als *critical* gewertet wird dargestellt.

```
# HELP gluster_brick_count Number of bricks at last query.
# TYPE gluster_brick_count gauge
gluster_brick_count{volume="gv_data"} 3
# HELP gluster_exporter_build_info A metric with a constant '1' value labeled by version, revision, branch, and goversion from which gluster_exporter was built.
# TYPE gluster_exporter_build_info gauge
gluster_exporter_build_info{branch="dev",goversion="go1.7.4",revision="6745a7e9ed428ffb04991630bce935cd849c1096",version="0.2.5"} 1
# HELP gluster_peers_connected Is peer connected to gluster cluster.
# TYPE gluster_peers_connected gauge
gluster_peers_connected 2
# HELP gluster_up Was the last query of Gluster successful.
# TYPE gluster_up gauge
gluster_up 1
# HELP gluster_volume_status Status code of requested volume.
# TYPE gluster_volume_status gauge
gluster_volume_status{volume="gv_data"} 1
# HELP gluster_volumes_count How many volumes were up at the last query.
# TYPE gluster_volumes_count gauge
gluster_volumes_count 1
# HELP gluster_mount_successful Checks if mountpoint exists, returns 0 or 1
# TYPE gluster_mount_successful gauge
gluster_mount_successful{mountpoint="/mnt/data",volume="localhost:data"} 1
# HELP gluster_volume_writeable Writes and deletes file in Volume and checks if it is writeable
# TYPE gluster_volume_writeable gauge
gluster_volume_writeable{mountpoint="/mnt/data",volume="localhost:data"} 0
# HELP gluster_heal_info_files_count File count of files out of sync, when calling "gluster v heal VOLNAME info".
# TYPE gluster_heal_info_files_count gauge
gluster_heal_info_files_count{volume="localhost:data"} 0
```

Listing A.2: Antwort des Gluster Exporters im Prometheus Format

Die einzige Änderung in Listing A.2 ist, dass das Volume nicht beschrieben werden kann. Dadurch schlägt die Überprüfung für Gluster fehl und der Clusterstatus ist *critical*