Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Lokales Alerting in Monitoringsystemen

Alexander Diehm

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Ing. habil. B. Mitschang
Betreuer/in:	Dipl.-Inf. Mathias Mormul
Beginn am:	23. November 2018
Beendet am:	23. Mai 2019

Kurzfassung

In der Arbeit wird ein Konzept für lokales Alerting in agentenbasierten Monitoringsystemen vorgestellt. Dieses ermöglicht es, die Auswertung von zentral definierten Alerts, auf die überwachten Hosts zu verteilen. Dadurch kann die Netzwerkauslastung des Monitoringsystems gesenkt werden. Um eine Verteilung und loakle Auswertung der Alerts zu ermöglichen, werden leichtgewchtige Alerting-Systeme auf den Hosts eingesetzt. Da eine manuelle Verteilung nicht praktikabel ist, beinhaltet das Konzept eine Mapper-Komponente, die die Verteilung der Alerts auf die Alerting-Systeme der Hosts automatisiert. Die Mapper-Komponente ist modular strukturiert, wodurch sie sich mit unterschiedlichen Monitoringsystemen einsetzen lässt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

AS Alerting System. 17

CAM Central Alerting Module. 34

CAS Central Alerting System. 23

CASC Central Alerting System Connector. 49

CEP Complex Event Processing. 28

EPL Event Processing Language. 45

IoT Internet of Things. 17

LAM Local Alerting Module. 34

LAS Local Alerting System. 23

LASC Local Alerting System Connector. 49

MA Monitoring Agent. 29

MAC Monitoring Agent Connector. 49

MAM Monitoring Agent Module. 34

MS Monitoring-Server. 19

SPOF Single Point of Failure. 20

VM virtuelle Maschine. 19

1. Einleitung

IT-Systeme sind durch die fortschreitende Digitalisierung unterschiedlicher Unternehmensbereiche komplexer und dynamischer geworden. Dies liegt beispielsweise daran, dass sich immer mehr Komponenten in IT-Systeme einbinden lassen, wie es z.B. beim Internet of Things (IoT) oder in der Industrie 4.0 der Fall ist. Zusätzlich hat die Verbreitung des Cloud-Computing [MG11] dazu geführt, dass sich IT-Systeme dynamischer gestalten lassen. Das Verhalten solcher IT-Systeme lässt sich mit Hilfe von Monitoringsystemen überwachen. Diese ermöglichen es, Informationen über alle Komponenten eines IT-Systems zu sammeln und zu verarbeiten. Dadurch kann ein fehlerhaftes Verhalten einzelner Komponenten schnell erkannt und auf dieses reagiert werden. Um die Prozesse der Problemerkennung und der Reaktion im Problemfall automatisieren zu können, nutzen viele von ihnen eine Form des Alerting. Alerting Systems (ASs) ermöglichen zu definieren, unter welchen Bedingungen Probleme vorliegen und wie auf diese reagiert soll.

Da diese Bedingungen meist in einer zentralen Komponente des Monitoringsystems definiert werden, müssen die Informationen aller überwachten Komponenten zu dieser gesendet werden. In großen Systeme kann es dazu kommen, dass der durch das Monitoring verursachte Netzwerk-Traffic die überwachten Systeme beeinflusst. Die Grundidee dieser Arbeit ist es, die Menge der Nachrichten durch den Einsatz von lokalem Alerting zu reduzieren. Dafür sollen die Bedingungen nahe der Informationsquellen ausgewertet werden. Wird bei der lokalen Auswertung ein Problem festgestellt, wird das zentrale AS benachrichtigt und führt die konfigurierte Reaktion aus. Der Hauptbestandteil der Arbeit ist die Entwicklung eines Konzepts, das die zentral definierten Bedingungen automatisch verteilen kann.

Die Arbeit ist wie folgt aufgebaut:

Den Einstieg bildet eine Einführung in die Grundlagen der Monitoringsysteme und des Alertings in Kapitel 2. Darauf folgt die Beschreibung eines Lösungsansatzes für das oben beschriebene Problem in Kapitel 3. In Kapitel 4 werden zwei aktuelle Monitoringsysteme und deren Alerting vorgestellt. Außerdem werden Möglichkeiten aufgezeigt, wie man Alerts zerlegen und verteilen kann. Der Hauptteil der Arbeit befasst sich mit dem in Kapitel 5 vorgestellten Konzept, welches eine Verteilung von Alerts in unterschiedlichen Monitoringsystemen ermöglichen soll. Eine Implementierung dieses Konzepts wird in Kapitel 6 beschrieben. Die mit dieser erzielten Ergebnisse werden in Kapitel 7 erläutert. Den Abschluss der Arbeit, in Kapitel 8, bilden die Zusammenfassung dieser und ein Ausblick über mögliche Weiterentwicklungen des Konzepts.

2. Grundlagen

IT-Systeme sind in den letzten Jahren stark gewachsen und durch die Verbreitung von Cloud-Computing immer dynamischer geworden. Dadurch wird es schwieriger die Komponenten moderner Systeme zu überwachen. Dies ist notwendig, um Fehler der Systeme zu erkennen oder die Effizienz dieser zu ermitteln. Hier können Monitoringsysteme Abhilfe schaffen. Sie ermöglichen es den Aufwand für die Überwachung zu verringern und machen diese oftmals erst möglich. Ihr Aufbau und ihre Funktionsweise werden in Abschnitt 2.1 beschrieben. Um auf Fehler möglichst schnell reagieren zu können, wird Alerting eingesetzt. Dieses erweitert die Funktionalität eines Monitoringsystems dadurch, dass für Fehlerfälle Aktionen konfiguriert werden können, die automatisch ausgeführt werden, sobald dieser Fehler eintritt. In Abschnitt 2.2 werden die Grundlagen des Alerting erklärt.

2.1. Monitoringsysteme

Monitoringsysteme dienen dazu, Informationen über IT-Systeme zu sammeln und für eine Auswertung zugänglich zu machen. Anhand der Informationen sollen Fehler oder Engpässe des Systems frühzeitig erkannt werden. Im Cloud-Computing gibt es noch weitreichendere Gründe für Monitoring. Während Nutzer von Cloud-Computing daran interessiert sind zu überwachen, ob mit dem Betreiber vereinbarte, Service Level Agreements eingehalten werden, ist dieser an einer möglichst effizienten Auslastung seiner Hardware interessiert [WLY+10].

Meist sind Monitoringsysteme so aufgebaut, dass Informationen über unterschiedliche Systemkomponenten, auch Metriken genannt, an einem zentralen Punkt zusammenlaufen. Dort werden diese den Nutzern, meist in Form von Dashboards, präsentiert. Dashboards bieten die Möglichkeit Metriken grafisch darzustellen und ermöglichen Nutzern mit diesen zu interagieren. Sie erlauben es einen schnellen Überblick über das System zu erlangen. Die zu überwachenden Komponenten können unterschiedlichster Natur sein. Von Hardware, über virtuelle Maschinen (VMs) und Applikation, können auch einzelne Sensoren überwacht werden. Die einzige Voraussetzung dabei ist, dass die Komponenten Metriken erzeugen die für das Monitoringsystem erfassbar sind. Zu überwachende Metriken sind beispielsweise die CPU Auslastung oder der freie Speicher eines Hosts, aber auch der Netzwerkverkehr oder die Verfügbarkeit einer API.

Monitoringsysteme sammeln die Metriken unterschiedlicher Produzenten auf ihren Monitoring-Servern (MSs). Produzenten können z.B. eine VM, die ihre Auslastung in Logs schreibt, ein Sensor, der simple Sensorwerte erzeugt, oder ein Agent, der Daten aus verschiedenen Quellen sammelt, sein. Monitoringsysteme lassen sich anhand verschiedener Eigenschaften, basierend auf der Klassifizierung von Seyd et al. [SGA+17], unterscheiden.

Es gibt zwei unterschiedliche Methoden, wie Metriken von den Produzenten zu den MSs gelangen. Man unterscheidet zwischen Pull- und Push-Kommunikation, deren Funktionsweise in Abbildung 2.1 zu sehen ist. Bei ersterem wird der MS aktiv und sammelt die Metriken der Produzenten

ein. Darunter fallen z.B. Methoden, bei denen der MS die Logs eines Produzenten ausliest oder eine Schnittstelle eines Produzenten anspricht, die Metriken an den MS sendet. Hierbei bestimmt der MS, wie häufig er die Metriken abfragt. Dahingehen sendet bei der Push-Kommunikation der Produzent die Metriken entweder in definierten Abständen oder wenn diese erzeugt werden zum MS. Dafür stellt der MS eine Schnittstelle für den Produzenten zur Verfügung , an die dieser seine Metriken senden kann. Je nach zu überwachendem Produzenten und Metrik, eignet sich eine der beiden Methoden besser, daher lassen sich in den meisten Monitoringsystemen beide Varianten konfigurieren. Ein Vorteil der Push-Kommunikation ist, dass der MS nicht alle Produzenten kennen muss und keinen Zugriff auf deren Metriken benötigt, was ein Sicherheitsrisiko darstellen könnte. Weiterhin entfällt das Polling durch den MS, sodass weniger Nachrichten übertragen werden müssen. Allerdings bestimmt der Produzent die Häufigkeit der Metriken.

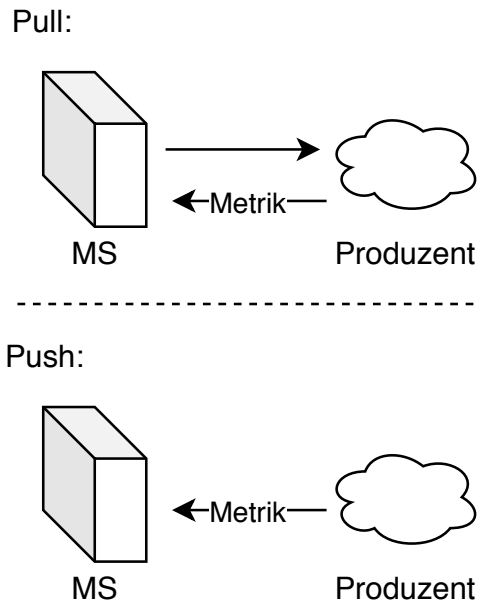

Abbildung 2.1.: Pull/Push Kommunikation

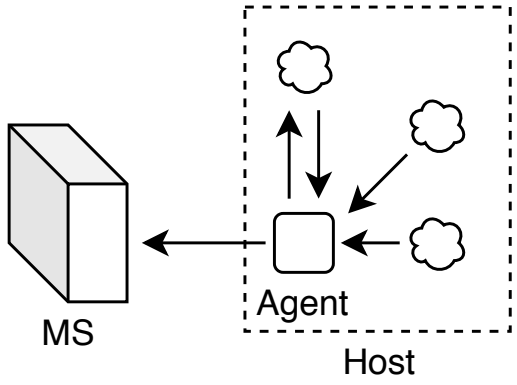

Abbildung 2.2.: Agentenbasiertes Monitoring

Viele Monitoringsysteme können Agenten einsetzen, dies nennt man agentenbasiertes Monitoring. Ein Agent, ist eine Softwarekomponente, die auf einem Host ausgeführt wird und Metriken von Produzenten sammelt. Mit Hosts sind in der Arbeit die VMs oder Rechnersysteme gemeint, die Agenten beherbergen. Abbildung 2.2 zeigt die Kommunikation zwischen einem MS, einem Agent und drei lokalen Produzenten. Der Agent erhält die Metriken der Produzenten über Push- oder Pull-Kommunikation. Beim Sammeln der Metriken ist er, in den meisten Fällen, nicht auf Metriken lokaler Produzenten beschränkt, sondern kann auch Entfernte miteinbeziehen. Er ist eine Art Zwischenstation zwischen Produzenten und dem MS, kann aber auch selbst als Produzent auftreten. Viele Agenten lassen sich flexibel konfigurieren und sind durch Skripts erweiterbar. Dies ermöglicht ihnen Metriken zu bearbeiten, beispielsweise zu aggregieren, um die Datenmenge, die an den MS gesendet wird, zu reduzieren. Allerdings benötigen sie dafür zusätzliche Ressourcen des Hosts. Je nach Umfang der, dem Agenten übertragenen, Aufgaben könnten dadurch andere Komponenten auf dem Host zu stark beeinflusst werden. Daher muss abgewogen werden, welche Aufgaben dem Agenten übergeben werden können. Ein Agent kann seine gesammelten Metriken dem MS über Push- oder Pull-Kommunikation zur Verfügung stellen. Bei Monitoringsystemen lassen sich zwei

Architekturen unterscheiden. Die am weiten verbreitetste Architektur ist eine Hierarchische, welche in Abbildung 2.3 zu sehen ist. Hierbei sind die MS und Produzenten in einem Baum angeordnet, bei dem die MS die inneren Knoten und die Produzenten die Blätter sind. In dieser Topologie fließen die Metriken von den Produzenten über die einzelnen Monitoring Server, in denen sie evtl. vorverarbeitet werden, in Richtung der Wurzel. Alle Metriken fließen in einem zentralen MS zusammen. Dadurch ist es leicht möglich eine Gesamtübersicht über das zu überwachende System zu erhalten. Besteht die Wurzel nicht nur aus einem MS, so spricht man von einer verteilten Architektur. Diese eliminiert den zentralen MS als Single Point of Failure (SPOF).

Eine andere Form der Architektur ist die dezentrale Variante, welche in Abbildung 2.4 zu sehen ist. Hierbei gibt es keinen zentralen MS, der als einziger die Gesamtübersicht des Systems liefert, denn alle MS haben die Aufgabe eine solche bereitzustellen. Dadurch entsteht ein hoher Aufwand bei der Verteilung der Metriken, die benötigt werden um die Sichten aller MS auf das System konsistent zu halten. Allerdings besitzt das System gegenüber der hierarchischen Varianten keinen SPOF und bleibt beim Ausfall eines MS weiterhin funktionstüchtig.

Für die meisten Anwendungsfälle ist die hierarchische Architektur besser geeignet, da sich diese einfacher entwickeln und verwalten lässt. Dezentrale Monitoringsysteme sind für Cloud-Computing interessant, da hier eine große Anzahl an unterschiedlichen Nutzern, Monitoring betreiben will. Diese benötigen alle unterschiedliche Sichten auf ihre Ressourcen, wobei sie selbst keinen Zugriff auf die Hardware besitzen. In diesem Fall kann eine dezentrale Lösung, bei der MS unterschiedlich Sichten liefern, besser skalieren [PLL+13].

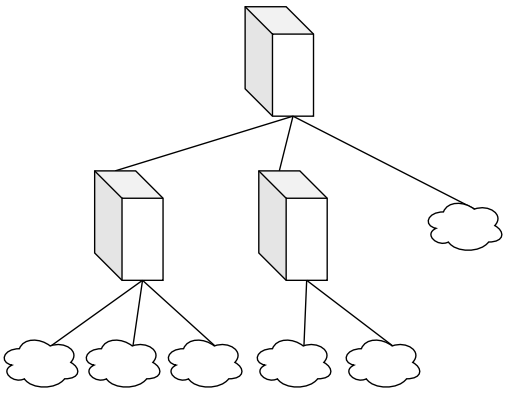

Abbildung 2.3.: Hierarchische Architektur

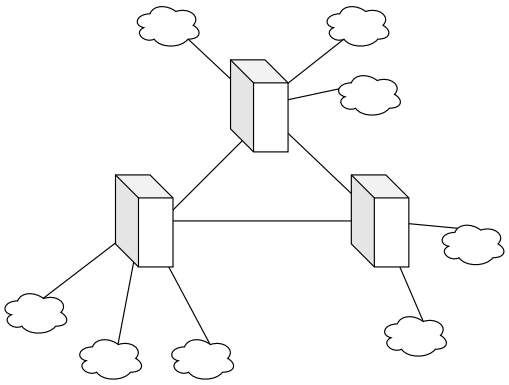

Abbildung 2.4.: Dezentrale Architektur

2.2. Alerting

Um Probleme in den Metriken eines Montoringsystems erkennen und auf diese reagieren zu können, müssen diese überwacht werden. Dies kann beispielsweise durch Nutzer geschehen, die die Dashboards der Systeme überwachen. Allerdings ist dies eine sehr mühsame Aufgabe und kann für große Systeme nicht händisch erledigt werden. Mit Hilfe von ASs kann dies automatisiert werden. Ein AS kann Teil eines Montoringsystems sein, oder dieses nur erweitern. Es ermöglicht es automatisiert auf außergewöhnliche Werte der Metriken zu reagieren. Dies geschieht durch die Definition von Alerts, die auf dem Strom an Metriken, des Monitoringsystems, ausgewertet werden.

Sie bestehen aus zwei Teilen, den Bedingungen und einer Aktion. Bedingungen legen fest wann, der Alert eintritt und dessen Aktion ausgeführt wird, indem sie Wertebereiche für Metriken festlegen, die diese einhalten sollen. Von festen Schranken für Werte, die nicht unter- oder überschritten werden dürfen, über Dynamische, die sich aus vorhergegangen Werten der Metriken ergeben, bis hin zu kompletten Profilen, die eine Metrik einhalten soll, ist vieles möglich. Sie lassen sich durch boolesche Operatoren kombinieren, sodass beliebig verschachtelte Bedingungen für Alerts erstellt werden können. Viele AS erlauben es, temporale Abhängigkeiten zwischen Metriken in Bedingungen zu definieren. So kann beispielsweise festgelegt werden, dass ein Alert nur dann eintritt, wenn Metriken über einen bestimmten Zeitraum ihre Bedingungen nicht erfüllen. Je komplexer Bedingungen werden, desto aufwendiger wird auch deren Auswertung.

Der zweite Teil eines Alerts ist seine Aktion. Diese wird ausgeführt, wenn alle Bedingungen des Alerts erfüllt sind. ASs bieten eine große Bandbreite an möglichen Aktionen an. Vom Ausführen von Skripts, die Probleme automatisch lösen sollen, bis hin zum Benachrichtigen, der für die Problemlösung zuständigen Personen, kann vieles konfiguriert werden.

Abbildung 2.5 zeigt einen MS der ein AS und einen Speicher für Metriken besitzt. Zwei Produzenten senden ihre CPU Metriken an den MS, die dann an das AS weitergeleitet werden können. In diesem sind mehrere Alerts definiert. Einer der Alerts besitzt eine Bedingung für die CPU Metriken von Produzent 1 und Produzent 2. Diese löst aus, wenn die CPU Auslastungen beider Produzenten 80% überschreitet. Ist dies der Fall, wird die Aktion ausgeführt und eine Nachricht an den Admin versandt. ASs müssen sicherstellen, dass möglichst wenig Fehlalarme erzeugt und unnötige Aktionen

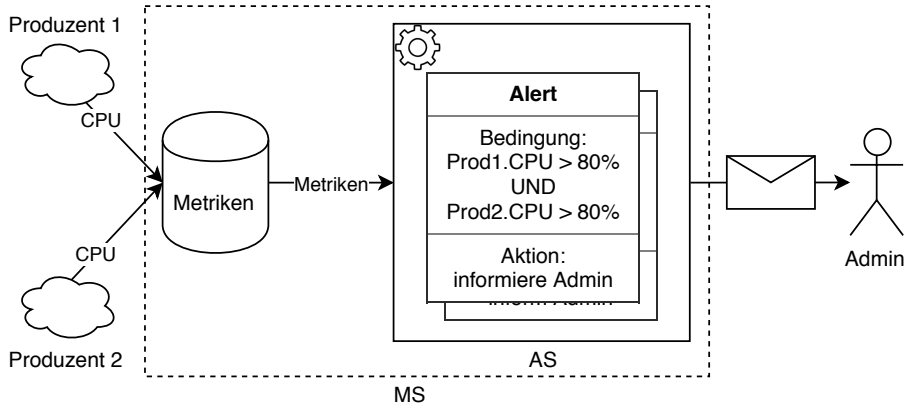

Abbildung 2.5.: Alerting System

vermieden werden. Zu viele unnötige Alerts verunsichern Nutzer und führen dazu, dass sie dem AS nicht mehr vertrauen. Daher müssen die Bedingungen und Aktionen der Alerts mit Bedacht gewählt werden.

In großen Monitoringsystemen ist es denkbar, dass die Menge an Metriken, die an einen zentralen MS und dessen AS gesendet werden, das Netzwerk zwischen den einzelnen Komponenten stark beansprucht. Dies kann dazu führen, dass das Monitoring eines Systems dessen Leistungsfähigkeit einschränkt, was verhindert werden muss. Ein Weg, die Menge der Metriken, die zum einem zentralen Server geschickt werden müssen, zu reduzieren, wird im nächsten Kapitel vorgestellt.

3. Lösungsansatz

Wird Alerting in Monitoringsystemen eingesetzt, so findet die Auswertung der Alerts meist in einem AS auf einem zentralen MS des System statt. Dort laufen alle Metriken der Produzenten zusammen. Während dies bei kleinen Systemen unproblematisch ist, sorgt es bei großen Systemen dafür, dass die Netzwerkauslastung ansteigt und dadurch das überwachte System beeinflusst wird. Für den Lösungsansatz werden nur agentenbasierte Monitoringsysteme betrachtet. Um die Anzahl der versandten Metriken zu reduzieren, kann es eine Lösung sein, die Auswertung der Alerts möglichst nahe an den Produzenten durchzuführen. Dies könnte umgesetzt werden, indem auf den Hosts der Agenten ebenfalls ASs eingesetzt werden, auf denen Alerts platziert werden können. Zur besseren Unterscheidung werden diese Local Alerting Systems (LASs) genannt. Sie sollten möglichst leichtgewichtig sein, um die Hosts nicht stark zu belasten. Die ASs der MSs werden Central Alerting System (CAS) genannt. Neben der Auswertung ganzer Alerts auf den LASs ist es auch denkbar, nur Teile einer Bedingung dort auszuwerten. Sollte ein Alert oder eine Bedingung auf einem LAS eintreten, so sendet dieses eine Alert-Metrik an das CAS, um diesem den Zustand des Alerts mitzuteilen.

Abbildung 3.1 stellt den Ist-Zustand vieler agentenbasierter Monitoringsysteme an einem vereinfachte Beispiel dar. Das Monitoringsystem besteht aus einem zentralen MS und den zwei Agenten auf Host 1 und Host 2. Der Agent von Host 1 sendet CPU- und RAM-Metriken an den MS, während der Agent des anderen Hosts, dessen CPU-Metriken versendet. Dieser besitzt eine CAS die die vier Alerts A1,A2,A3 und A4 verwaltet. Deren Bedingungen sind Schranken für die Metriken der beiden Hosts. Alert A4 unterscheidet sich von den drei anderen darin, dass dieser mehrere Bedingungen besitzt, welche Metriken von beiden Hosts betreffen. Will man nun die Anzahl der versendeten Metriken reduzieren, kann man das System, wie in Abbildung 3.2 zu sehen, anpassen. Dafür wurde jeder Host um ein LAS ergänzt und die Agenten schicken ihre Metriken nun an dieses. Da die Bedingungen der Alerts A1, A2 und A3 nur Metriken von jeweils einem Host betreffen, können sie auf den entsprechenden LASs platziert werden. Die LASs werten die Alerts auf dem Strom der Metriken aus und leiten Alert-Metriken oder Metriken, für die sie keine Bedingungen besitzen, an den Agenten weiter. Dieser schickt sie an das CAS. Die Bedingungen des CAS werden so angepasst, dass diese die Aktion der Alerts bei Eintreffen der Alert-Metriken auslösen. Dadurch, dass die Aktionen weiterhin auf dem CAS ausgeführt werden, können die LAS einfacher gehalten werden.

Für die Auswertung der Alerts A1, A2 und A3 werden nun nur noch Alert-Metriken zwischen Hosts und dem CAS versandt, was die Netzwerkauslastung reduziert. In der Abbildung wurde A4 nicht lokal platziert, da seine Bedingungen mehrere Host betreffen. Es ist jedoch denkbar A4 auf einem der beiden Host zu platzieren, und die fehlenden Metriken der anderen VM zu dieser zu outen. Dieses Vorgehen wird in der Arbeit Clustering genannt, da man die Metriken nur innerhalb der Hosts versendet und nur ein Host mit dem CAS kommuniziert. Clustering kann vor allem dann sinnvoll sein, wenn die Kommunikation zwischen den Hosts günstiger ist als die Kommunikation zwischen

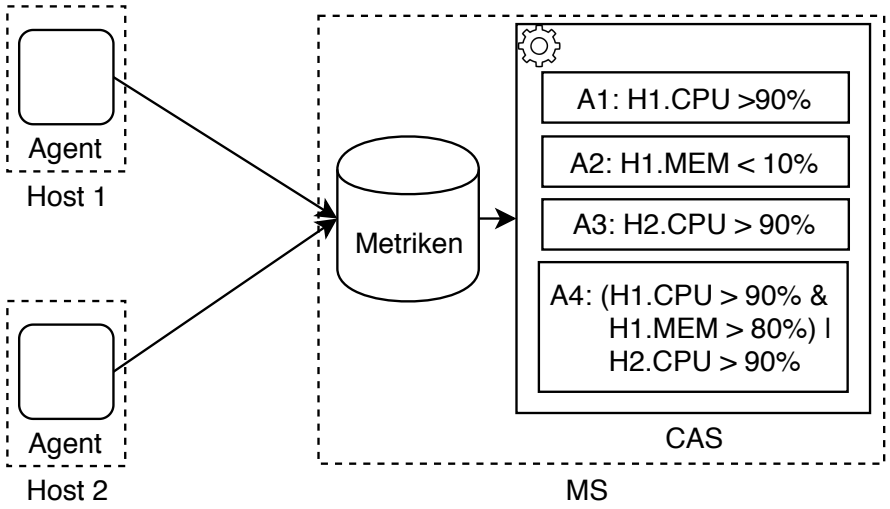

Abbildung 3.1.: Ist-Zustand

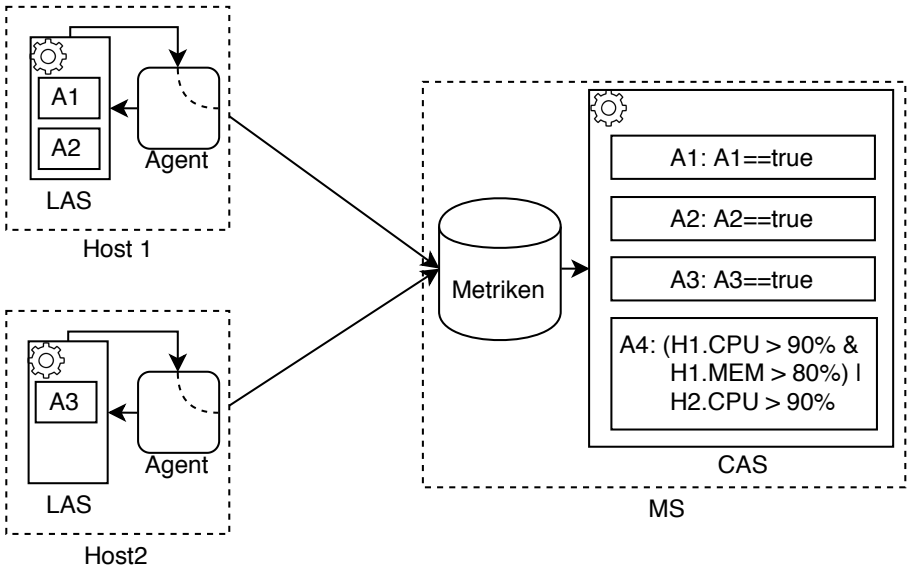

Abbildung 3.2.: Lösungsansatz

Host und CAS. Dies könnte z.B. dann der Fall sein, wenn sich die Hosts im gleichen Rechenzentrum befinden und das CAS sich nicht in diesem befindet. Bei komplizierteren Bedingungen kann es außerdem sinnvoll sein diese, in ihre Teilbedingungen zu zerlegen und diese einzeln zu platzieren.

Damit man auf dem zentralen MS weiterhin einen Überblick über die Metriken der Hosts erhalten kann, ist es denkbar, aggregierte Metriken an diesen zu schicken. Die Aggregation müsste dabei durch den Agenten oder das LAS durchgeführt werden.

Die Verteilung von Alerts kann dazu beitragen, die Netzwerkauslastung zu senken. Allerdings ist sie mit Konfigurationsaufwand verbunden. Bei der Verteilung eines Alerts müssen alle betroffenen Agenten und LASs, sowie das CAS angepasst werden. Dies von Hand zu erledigen ist fehleranfällig und mühsam. Daher wird in Abschnitt 5 ein Konzept vorgestellt, welches die automatisierte Verteilung von Alerts in Monitoringsystemen ermöglichen soll.

4. State of the Art

Monitoringsysteme sind heutzutage vielfältig [SGA+17][WB14][ABDP13][FEH+14]. Abschnitt 4.1 beschreibt, wie Monitoringsysteme die Netzwerkauslastung reduzieren und wie Alerting in Nagios [Nag] konfiguriert wird. Abschnitt 4.2 befasst sich mit der Verteilung von Alerts auf die unterschiedlichen Knoten eines Systems.

4.1. Monitoringsysteme

In vielen Monitoringsystemen kann die Menge der Daten die an einen zentralen MS gesendet werden durch einfache Aggregation im Agenten reduziert werden [Infb][Nag][MSM+13][MCC04]. Ebenso bieten dies einige Agenten an, die in Monitoringsysteme eingebunden werden können [col][sta]. Allerdings kann es beim Alerting auf aggregierten Metriken dazu kommen, dass wichtige Ereignisse nicht erkannt werden. Eine andere Möglichkeit die Menge der Daten zu reduzieren, besteht darin, aus der Ressourcenauslastung von Hosts Profile zu generieren und diese für das Alerting zu nutzen [HW18]. Dabei muss daraf geachtet werden das diese genug Aussagekraft besitzen. Moogsoft [Mooa] besitzt mit Observe [Moob], einen Agenten im Betastadium, der lokales Alerting ermöglicht und Benachrichtigungen versenden kann. Alle dieser Methoden haben den Nachteil, dass dass die Konfiguration der Agenten einzeln angepasst werden muss und dies nicht automatisch geschieht.

4.1.1. Nagios

Nagios ist wahrscheinlich das am weiten verbreitetste Monitroingsystem und gilt als Industriestandard. Der Kern von Nagios beinhaltet nur die Monitoringlogik, samt Alerting. Um einen Strom von Metriken zu erzeugen nutzt Nagios Plugins. Diese Plugins führen aktive oder passive Checks durch, um den Zustand von zu überwachenden Objekten, z.B VMs, zu erfahren. Objekte haben einen der vier Zustände: OK, WARNING, UNKNOWN oder CRITICAL. Bei aktiven Checks versucht das Plugin aktiv den Zustand eines Objekts zu erfahren, es betreibt Polling, während bei passiven Checks die Ergebnisse von entfernten Checks, beispielsweise durch Agenten, an das Plugin gesendet werden. Agenten können wiederum aktive oder passive Checks durchführen. Aufgrund der Plugin-Architektur ist Nagios fast beliebig erweiterbar, was jedoch mit einem hohen Konfigurationsaufwand einhergeht.

Für das Alerting bietet Nagios die Möglichkeit Event-Handler und Notifications zu konfigurieren. Mit Hilfe von Event-Handlern können Aktionen festgelegt werden, die bei einem Übergang, eines Objekts, von einem Zustand in einen anderen ausgeführt werden. Aktionen können beliebige Skripts, z.B. zum Neu starten einer VM, sein. In Nagios können Zustände HARD oder SOFT sein. Ein Zustand ist zunächst SOFT und wird HARD, wenn ein Check n-mal hintereinander den selben

Zustand zurückgeliefert hat, wobei n für jeden Check separat konfiguriert werden kann. Finden Zustandsübergänge zwischen, oder in harte Zustände statt, so können Notifications an Gruppen versandt werden.

Durch den Einsatz von aktiven Checks auf den Agenten könnte ein Art des lokalen Alertings umgesetzt werden. Allerdings müssten die Agenten einzeln lokal angepasst werden, da Nagios eine automatisierte Verteilung von Checks nicht unterstützt. Ohne die Verwendung spezieller Plugins ist eine solche Konfiguration nicht einsetzbar. Der Nagios Fork Icinga [Ici] stellt mit dem Director ein Modul bereit, welches es ermöglicht, aktive Checks zentral zu verwalten und automatisiert auf die Agneten zu verteilen. Ein ähnliches Plugin existiert mit Mod-Gearman [Mod] für den Fork Naemon [Nae].

4.2. Operator Placement

Um die Frage zu beantworten, wie sich Alerts möglichst effizient verteilt auswerten lassen, lohnt es sich einen Blick auf den Bereich des Complex Event Processing (CEP) zu werfen, da Alerting eine Form des CEP darstellt. Beim CEP können Queries für Eventströme definiert werden, ähnlich den Alerts für Metriken im Alerting. Queries lösen ebenfalls Aktionen, z.B. das Versenden von komplexen Events, aus, wenn ihre Bedingungen erfüllt sind. Auch beim CEP stellt sich die Frage wie Queries aufgeteilt und auf Hosts verteilt werden können. Die folgenden Arbeiten bieten interessante Ansätze.

Schultz-Møller et. al [SMP09] beschreiben ihr NEXT CEP System, welches es ermöglicht Queries zu optimieren und zu verteilen. Dabei nutzen sie einen automatenbasierten Ansatz zur Eventerkennung. Dieser beruht darauf, dass sich alle Queries in sechs grundlegende Automaten, den Operatoren des Systems, zerlegen lassen. Queries werden in einer eigens entwickelten Querysprache verfasst. Für deren Optimierung von Queries nutzt das System die erwarteten Eventraten der Events. Daraus schätzt es die Eventraten der Operatoren der Queries ab und strukturiert diese so um, dass die Summe der Eventraten aller Operatoren minimal ist. Bei der Umstrukturierung muss darauf geachtet werden, dass diese noch immer die gleichen Events der Ausgangsquery erzeugt. Während der Verteilung auf die einzelnen Knoten, nutzt das System die Möglichkeit für die Operatoren mehrere Outputs zu definieren. Dadurch können Operatoren alter Queries wiederverwendet werden. Können Operatoren nicht wiederverwendet werden, werden sie ”greedyïm System verteilt. Das NEXT CEP System bietet interessante Ansätze, die sich aber nicht ohne Weiteres in bestehenden Monitoringsystemen umsetzen lassen.

Das T-Rex CEP System [CM12] von Cugola und Margara besteht aus einem Netz von Prozessor-Knoten, die alle in der Lage sind, Queries zu verarbeiten. Eventquellen, bzw. Produzenten, und Zielsysteme sind mit diesen verbunden. Queries werden in der Eventsprache TESLA [CM10] definiert. Für die Verteilung von Queries haben sie einen verteilten Algorithmus eingesetzt, der versucht die Queries möglichst nahe an den Eventquellen auszuwerten. [CM13] Dieser versucht, beginnend bei der Wurzel des eines Prozessorbaumes, dem mit dem Zielsystem verbundenen Knoten, die komplette Query möglichst tief im Baum zu platzieren. Ist dies an einem Prozessor nicht weiter möglich, wird die Query in Teilqueries zerlegt und diese werden wiederum tiefer im Baum platziert. Der Algorihmus endet, wenn sich die Queries nicht weiter zerlegen lassen. Die Tests der Autoren haben ergeben, dass die Verteilung von Queries die Netzwerklast gegenüber einer zentralen Auswertung deutlich reduzieren kann.

5. Konzept

Um die in Abschnitt 3 beschriebenen Probleme bei lokalem Alerting in Monitoringsystemen zu lösen, wurde das nachfolgende Konzept entwickelt. Es soll lokales Alerting in zentralen Monitoring Systemen ermöglichen oder vereinfachen. Dafür wurde eine neue Komponente entwickelt, die die Verteilung der Alerts und deren Verwaltung automatisiert. Sie liest Alerts aus dem CAS des Monitoringsystems aus, generiert eine Verteilung anhand deren Bedingungen und verteilt sie auf die Hosts. Damit dies möglich ist, müssen sich auf den Hosts LASs befinden, die die lokale Auswertung der Alert-Bedingungen übernehmen. Die bisherigen Arbeitsabläufe der Nutzer für das Erstellen von Alerts sollen sich nicht verändern. Deshalb werden Alerts weiterhin über das CAS konfiguriert, welches weiterhin für die Ausführung der Aktionen der Alerts zuständig ist.

Abbildung 5.1 zeigt die Architektur des Systems. Es erweitert die für lokales Alerting notwendige Architektur aus Kapitel 3, bestehend aus MS mit CAS und einem Speicher für Metriken und einem Host mit Monitoring Agent (MA) und LAS, um die Mapper-Komponente. Sie ist der Mittelpunkt des Konzepts und verteilt zentral definierte Alerts automatisch auf einzelne Hosts bzw. deren LASs. Dafür muss sie die Konfigurationen der CASs, LASs und MAs anpassen. Bei mehreren Hosts mit MAs besitzt jeder Host ein LAS, während der Mapper nur einmal im System vorkommt. LASs empfangen Metriken von den MAs und senden etwaige Alert-Metriken an den lokalen MA, der diese wiederum an den MS oder ein weiters LAS sendet.

Wie die einzelnen Komponenten funktionieren und welche Anforderungen an sie gestellt werden, wird in Abschnitt 5.2 im Detail beschrieben. Abschnitt 5.3 beschreibt, wie die Komponenten interagieren, um die in Abschnitt 5.1 an das Gesamtsystem gestellten Anforderungen zu erfüllen.

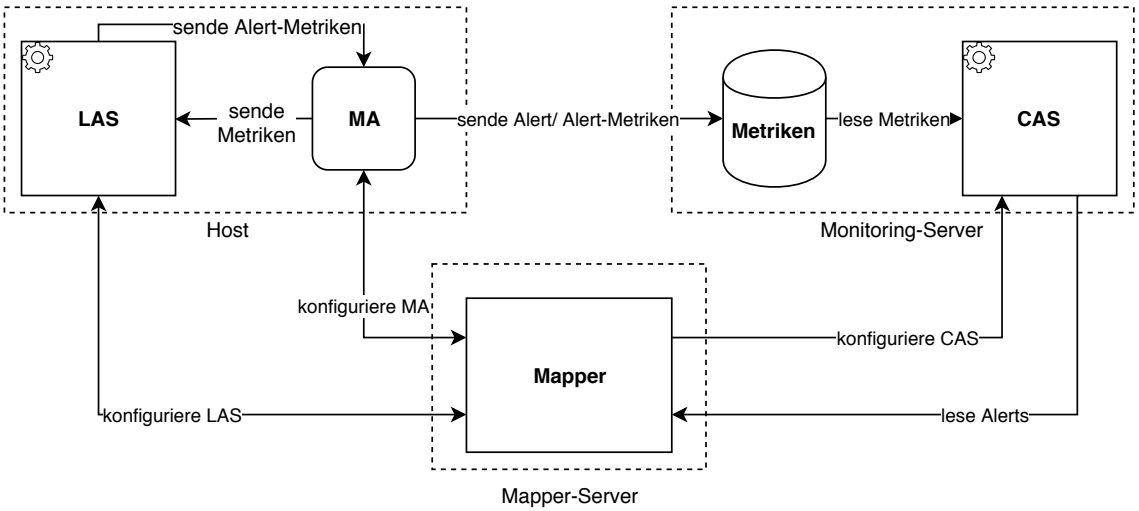

Abbildung 5.1.: Einfache Architektur des Systems

5.1. Anforderungen

Die Anforderungen an das Konzept lassen sich in die folgenden funktionalen und nicht-funktionalen Anforderungen unterteilen.

5.1.1. Funktionale Anforderungen

- **Zentrale Definition von Alerts**

  Alerts sollen weiterhin im CAS zentral definiert werden, da eine verteilte Definition von Alerts den Verwaltungsaufwand erhöht und für den Nutzer unübersichtlicher ist. Dadurch können die Arbeitsabläufe der Nutzer beibehalten und Aktionen der Alerts weiterhin zentral verwaltet und ausgeführt werden. Dies ermöglicht es wiederum die LASs einfacher zu gestalten, da sie im Gegensatz zum CAS nur für die Auswertung der Bedingungen zuständig sind.

- **Automatische Verteilung von Alerts**

  Die Verteilung der Alerts auf LASs soll automatisch und ohne Eingriffe eines Nutzers erfolgen. Alle notwendigen Konfigurationsschritte werden vom Mapper übernommen. Dabei muss sichergestellt sein, dass das Ergebnis einer verteilten Auswertung der Alerts stets dem einer zentralen Auswertung entspricht. Ohne die automatische Verteilung der Alerts ist das lokale Alerting, mit verteilter Auswertung eines Alerts, nicht praktikabel. Dies liegt zum einen daran, dass der für die Verteilung notwendige Konfigurationsaufwand hoch ist und zum anderen daran, dass die Konfiguration mehrerer Komponenten fehleranfällig ist.

- **Reduzierung des Netzwerk-Traffics**

  Alerts sollen durch das System so verteilt werden, dass der Netzwerk-Traffic zwischen den einzelnen Komponenten reduziert wird. Dies ist die Grundidee hinter dem lokalem Alerting und bildet die Grundlage für das vorgestellte System.

- **Einfluss auf die Verteilung über eine Schnittstelle**

  Das System soll eine Schnittstelle bieten, über die externe Systeme oder Nutzer Einfluss auf die Verteilung der Alerts nehmen können. Dadurch können neben der Reduktion des Netzwerk-Traffics andere Kriterien für die Verteilung angewandt werden.

- **Verwendbarkeit mit bestehenden Monitoringsystemen**

  Das vorgestellte System soll sich in bestehende Monitoringsysteme integrieren lassen. Dazu müssen die Schnittstellen zur Konfiguration der CASs, LAS und MAs generisch gestaltet werden. Dies ist vor allem in großen IT-Systemen notwendig, da hier oftmals mehrere Monitoringsysteme simultan eingesetzt werden.

- **Aggregation von Metriken verteilter Alerts**

  Durch das Umlenken der Metriken in die LAS können Nutzer den Verlauf deren Werte nicht mehr auf dem MS verfolgen. Eine Möglichkeit wäre es die Metriken zusätzlich an den MS zu senden. Allerdings findet dann keine Reduktion des Netzwerk-Traffics mehr statt. Um dieses Problem zu umgehen, soll es das System ermöglichen aggregierte Metriken an den MS zu senden, wenn eine Metrik in ein LAS umgeleitet wird.

- **Rückgängigmachen der Verteilung**

  Das System muss in der Lage sein, die Verteilung von Alerts rückgängig zu machen. Nur dadurch kann das Alerting konsistent bleiben, wenn der Mapper abgeschalten wird.

5.1.2. Nicht funktionale Anforderungen

- **Transparenz der Verteilung**

  Es soll für den Nutzer sichtbar sein, wie ein Alert verteilt wurde, bzw. wo seine Teilbäume ausgewertet werden. Dies ist notwendig, um Probleme der Auswertung von Alerts erkennen und beheben zu können.

- **LASs müssen leichtgewichtig sein**

  Der Einsatz von LASs darf die Ressourcenauslastung der Hosts nicht wesentlich erhöhen, da dies sonst negative Effekte auf andere Anwendungen des Hosts haben könnte. Dies kann dazu führen, dass die Metriken des Host ihre Aussagekraft verlieren.

5.2. Komponenten

Wie in Abbildung 5.1 zu sehen besteht das Konzept aus den vier Komponenten CAS, MA, LAS und Mapper. Deren Funktionsweisen und Aufbau werden in den folgenden Abschnitten genauer betrachtet. Zudem werden die Anforderungen, die sich aus den Anforderungen des Gesamtsystems ergeben, erläutert.

5.2.1. Monitoring Agent

Die MAs entsprechen den Agenten eines bereits verwendeten Monitoringsystems. Im Idealfall ermöglicht der Agent die Konfiguration der vier Bereiche Input, Output, Processing und Routing. Inputs können z.B. unterschiedliche Produzenten von Metriken oder auch Schnittstellen zur Weiterleitung von Alert-Metriken sein. Die Konfiguration unterschiedlicher Outputs erlaubt es Metriken an unterschiedliche Komponenten zu senden. Sollte der MA eine Verarbeitung von Metriken ermöglichen, wird das Processing konfiguriert. Das Routing von Metriken, innerhalb eines MAs, erlaubt es Inputs und Outputs miteinander zu verknüpfen. Ist diese feingranulare Konfiguration nicht möglich, reicht es aus, wenn der MA seine Metriken an ein LAS sendet und dieses die Aufgaben übernimmt.

Abbildung 5.2 zeigt die Konfiguration des Agenten von Host 1 aus Abbildung 3.1. Der Agent hat zwei Inputs, die die Metriken CPU und MEM erzeugen, und einen Ouput, der es ermöglicht, Daten an das CAS zu senden. Dies ist die Ausgangskonfiguration an der das System ansetzt.

Sollen die Metriken, wie in Abbildung 3.2, durch das LAS verarbeitet werden, muss die Konfiguration des MA verändert werden. Dies kann z.B. durch eine Anpassung einer Konfigurationsdatei und anschließendem Neustart des Agenten passieren. Abbildung 5.3 zeigt die Konfiguration des MA nach den notwendigen Änderungen. Es wurde ein neuer Output hinzugefügt, welcher es ermöglicht Metriken an das lokale LAS zu senden. Zudem wurde ein Input hinzugefügt, der Metriken des LAS

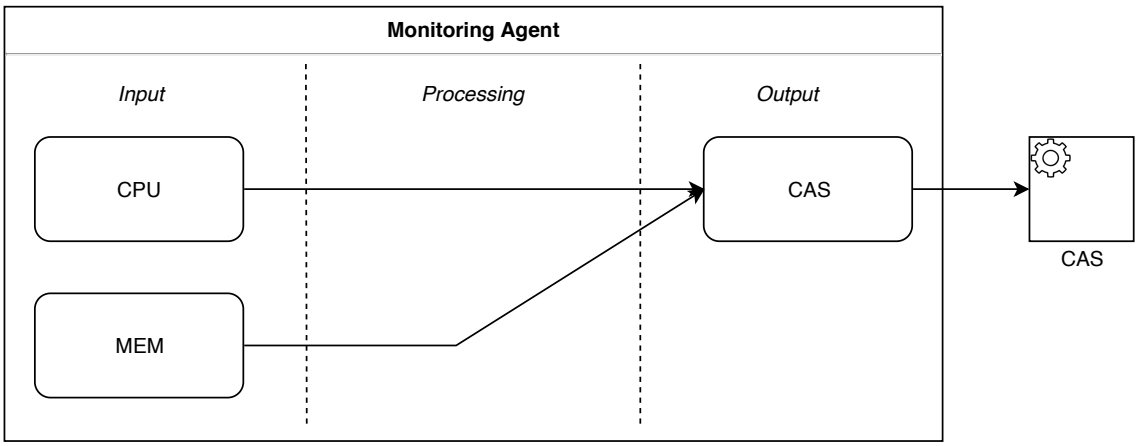

Abbildung 5.2.: Agent im Ausgangszustand

entgegen nehmen kann. Da beide Metriken nun durch das LAS verarbeitet werden sollen, wurde deren Routing entsprechend angepasst. Zusätzlich werden sie zum Aggregator geroutet, welcher ihre Werte aggregiert und an das CAS sendet. Sollten die Bedingungen für Alert A1 oder A2, aus Abbildung 3.1, erfüllt sein, sendet das LAS eine Alert-Metrik an den Alert-Backloop, welcher selbige an das CAS weiterleitet.

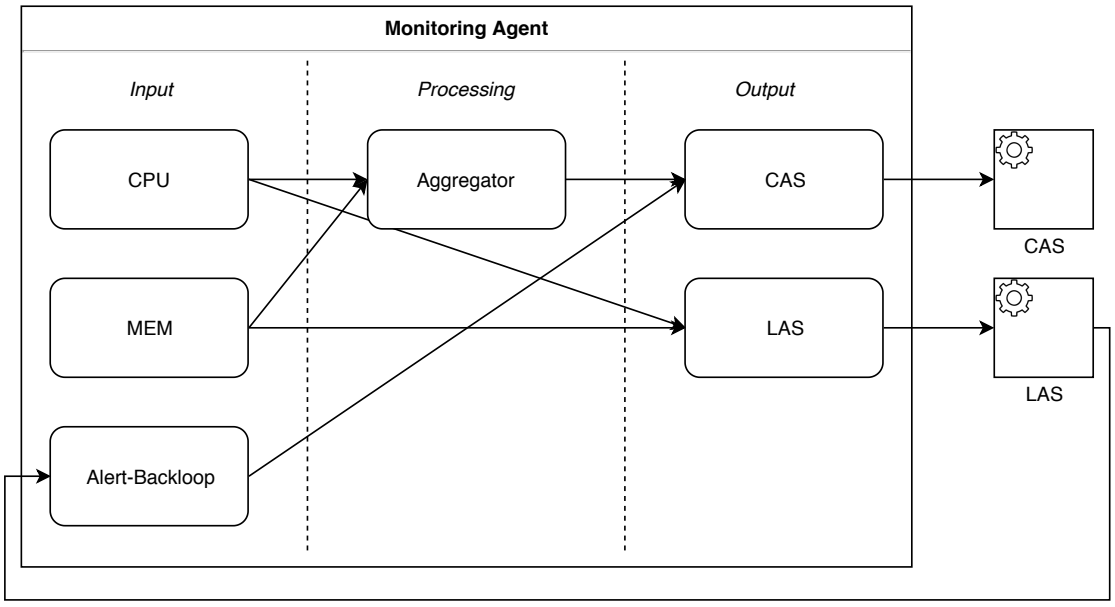

Abbildung 5.3.: Agent nach der Konfiguration für lokales Alerting

Liegen lokale Alert-Bedingungen für mehrere Metriken vor, so muss nur das Routing dieser Metriken verändert werden. Es müssen keine weiteren Inputs oder Outputs konfiguriert werden. Um die Änderungen für das lokale Alerting einer Metrik rückgängig zu machen, muss nur deren Routing wieder in den ursprünglichen Zustand zurückgesetzt werden. Nach der oben beschrieben Konfigura-

tion des Agenten kann dieser für das vorgestellte Konzept des lokalen Alertings genutzt werden. Die nötigen Schritte für die Konfiguration unterscheiden sich je nach eingesetztem Agenten stark. Sie werden im System durch den Mapper vorgenommen.

5.2.2. Local Alerting System

Das LAS ist dafür zuständig, die ihm durch den Mapper übergebenen Alert-Bedingungen auszuwerten. Es befindet sich auf dem selben Host wie der MA der sie zugeordnet ist. Als LAS können beispielsweise leichtgewichtige CEP Engines zum Einsatz kommen. Wie in Abbildung 5.4 zu sehen, können LASs Metriken von unterschiedlichen Hosts empfangen. So empfängt das LAS von Host 1 Metriken von MAs auf Host 1 und Host 2. Falls diese Metriken die Bedingungen im LAS erfüllen, so sendet dieses eine Alert-Metrik an ihren lokalen MA, welcher sich um deren Weiterleitung kümmert.

Die Komponente stellt eine Schnittstelle zur Konfiguration zur Verfügung, die vom Mapper genutzt werden kann. Sie erlaubt es, Alert-Bedingungen zum LAS hinzufügen oder von diesem entfernen zu können. Dies ermöglicht eine dynamische Verteilung der Alerts. Erst durch die Konfiguration des MA und des LAS durch den Mapper kann lokales Alerting auf dem Host stattfinden. Damit die Alert-Metriken des LAS verarbeitet werden können, fehlt nun noch die Anpassung des CAS, welche im nächsten Abschnitt beschrieben wird.

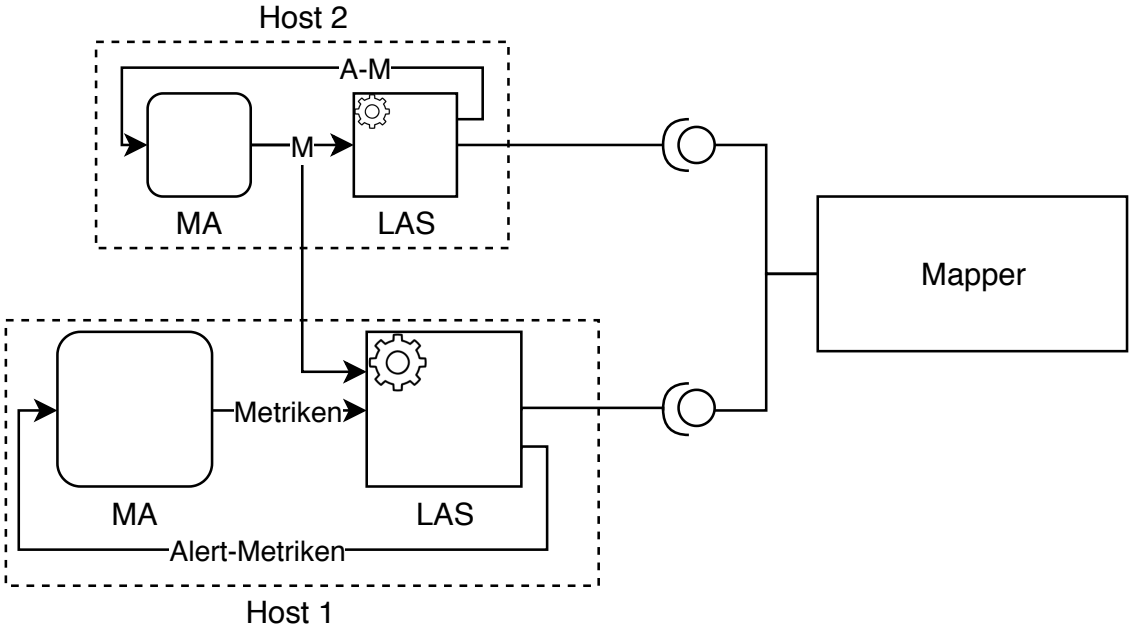

Abbildung 5.4.: Local Alerting System

5.2.3. Central Alerting System

Das CAS ist weiterhin die zentrale Komponente des Systems, in der Metriken und Alert-Metriken zusammenlaufen. Hier können Nutzer Alerts zentral definieren und die einzelnen Metriken des Systems einsehen. Zudem werden hier die Aktionen aller Alerts verwaltet und ausgeführt. Zusätzlich zu der normalen Funktionsweise des CAS muss dieses dem Mapper eine Schnittstelle bereitstellen, über welche Alerts, bzw. deren Bedingungen, ausgelesen und verändert werden können.

5.2.4. Mapper

Der Mapper ist der Mittelpunkt des vorgestellten Konzepts. Er übernimmt die Verteilung der Alerts auf die Hosts und verwaltet diese. Abbildung 5.5 zeigt den internen Aufbau der Komponente. Sie besteht aus den vier unterschiedlichen Modultypen Central Alerting Module (CAM), Core, Monitoring Agent Module (MAM) und Local Alerting Module (LAM). Der Core ist das zentrale Modul des Mappers und für die Verteilung der Alerts und deren Optimierung zuständig. Um diesen möglichst generisch zu halten, verwendet er eine interne Datenstruktur für Alerts, welche in Abschnitt 5.2.5 beschrieben ist. Zudem stellt er eine externe Schnittstelle zur Verfügung, über die die Verteilung der Alerts, mit Hilfe von Alert-Policys, beeinflusst werden kann. Alert-Policys können beispielsweise festlegen, ob Bedingungen eines Alerts lokal ausgewertet werden dürfen, oder ob und wie eine Aggregation der Metriken stattfinden soll. Für die bei der Verteilung der Alerts notwendigen Änderungen der Konfigurationen, der restlichen Komponenten, nutzt der Core die Konfigurationsmodule vom Typ CAM, MAM und LAM. Diese übernehmen die Konfiguration der Komponenten und stellen ihm einfache Schnittstellen zur Verfügung. Während der Mapper Core nur einmal pro Mapper vorkommt, hängt die Anzahl der Konfigurationsmodule von der Anzahl der unterschiedlichen CASs, LASs und MAs ab, die unterstützt werden sollen. Durch die Aufteilung des Mappers in Module ist dieser flexibel in bestehende Monitoringsysteme integrierbar.

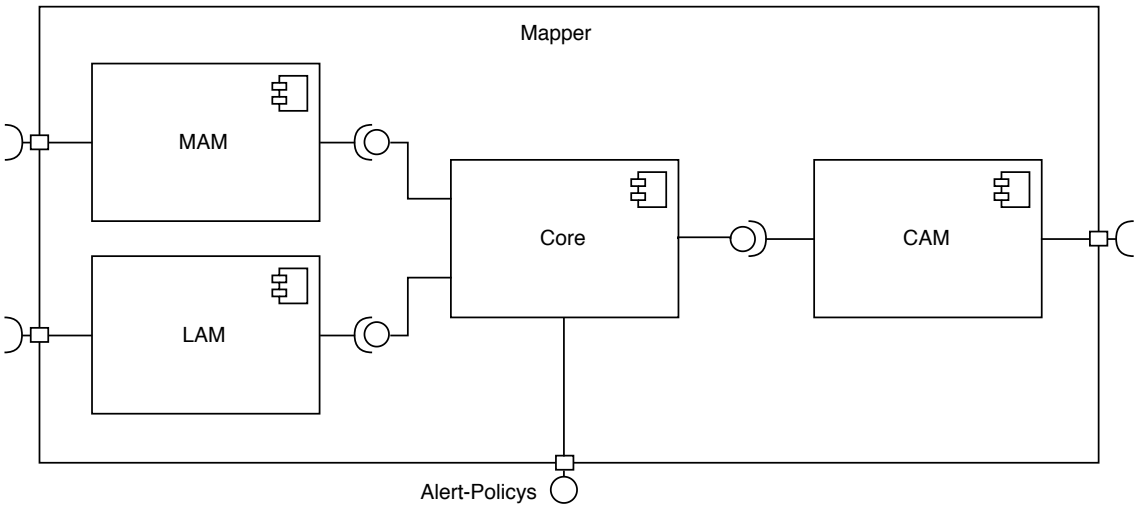

Abbildung 5.5.: Aufbau der Mapper-Komponente

Konfigurationsmodule

Konfigurationsmodule ermöglichen es dem Mapper in unterschiedlichen Monitoringsystemen eingesetzt zu werden. Sie sind für die Kommunikation mit den Komponenten außerhalb des Mappers und deren Konfiguration zuständig. Um für den Core nutzbar zu sein, stellen sie diesem eine Schnittstelle zur Verfügung, über die dieser die Konfigurationen der Komponenten anpassen kann. Damit der Mapper Alerts verteilen kann, muss von jedem Typ der Konfigurationsmodule mindestens eines vorhanden sein. Die Module haben je nach Typ die folgenden Aufgaben:

1. CAM
    - Implementiert eine Schnittstelle zu einem CAS. Diese soll es ermöglichen, die Alerts des CAS auszulesen und zu verändern. Außerdem kann sie genutzt werden, um die Konfiguration des CAS anzupassen.
    - Implementierung der zur Anpassung der Konfiguration notwendigen Logik.
    - Umwandlung der Alert-Definitionen des CAS in die interne Datenstruktur des Cores und umgekehrt.
    - Stellt dem Core Methoden zur Verfügung, mit Hilfe derer er Änderungen der Konfiguration anstoßen oder Alerts in der internen Datenstruktur erhalten kann.
2. MAM
    - Implementiert die Schnittstelle zu einem MA. Über diese kann die Konfiguration der MAs geändert werden.
    - Implementierung der zur Änderung der Konfiguration notwendigen Schritte. Dies kann z.B. die Anpassung der Konfigurationsdatei des MA sein, welche durch einen Neustart des Agenten aktiviert wird.
    - Stellt dem Core Methoden zur Verfügung, durch die er die Änderung des Routing anstoßen kann.
3. LAM
    - Implementiert die Schnittstelle zu einem LAS. Über diese lassen sich Bedingungen auf einem LAS platzieren und auch wieder von diesem entfernen.
    - Wandelt die Bedingungen der internen Datenstruktur des Cores in eine für das LAS nutzbare um, beispielsweise Regeln, wenn eine CEP-Engine als LAS eingesetzt wird.
    - Stellt dem Core Methoden zur Verfügung, mit Hilfe derer er Bedingungen auf LASs platzieren kann.

5.2.5. Alert Datenstruktur

Durch die Verwendung einer internen Datenstruktur für Alerts kann der Core eine Verteilung von Alerts unabhängig von den Alert Formaten der unterschiedlichen CASs und LASs durchführen. Für die Umwandlung der externen Formate in das interne sind die entsprechenden Module zuständig. Durch Nutzung eines internen Formats wird die Anzahl der benötigten Formatkonvertierungen auf ein Minimum reduziert. Die interne Datenstruktur ist wie folgt aufgebaut.

Ein Alert besitzt eine für den Mapper eindeutige ID, die Adresse des CAS aus dem dieser entstammt und einen Bedingungsbaum. Der Bedingungsbaum beinhaltet die Bedingungen die dazu führen, dass eine Alert ausgelöst wird. Innere Knoten sind boolsche Operatoren, während Blätter einzelnen Bedingungen entsprechen. Eine Bedingung besitzt eine Quelle und eine Schranke für eine Metrik der Quelle. Abbildung 5.6 zeigt den Bedingungsbaum von Alert 4 aus Abschnitt 3. Dessen Bedingungen legen Schranken für die Metriken MEM und CPU fest, die aus den Hosts, Host 1 und Host 2, stammen. Der Alerts löst eine Aktion aus, wenn die CPU Auslastung von Host 2 90% übersteigt oder die CPU und Speicherauslastung von Host 1 90% bzw. 80% übersteigt. Die Aktionen der Alerts werden für die Verteilung nicht benötigt und werden nicht in der internen Datenstruktur gespeichert, sie sind nur im entsprechenden CAS vorhanden.

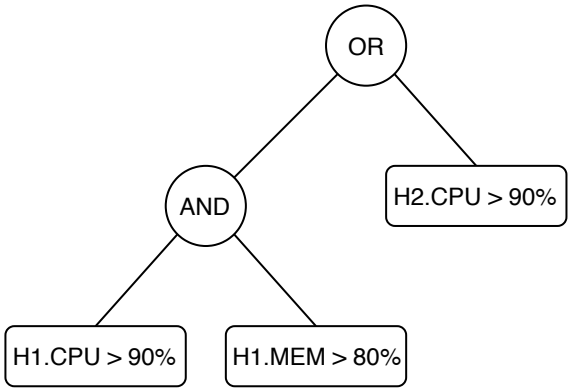

Abbildung 5.6.: Bedingungsbaum des Alerts A4

Core

Der Core ist das Hauptmodul des Mappers. Es besitzt eine externe Schnittstelle, über die sich Alert-Policys für Alerts und deren Bedingungsbäume definieren lassen. Diese Policys können beispielsweise festlegen, ob Bedingungen lokal ausgewertet werden dürfen oder wie die Metriken dieser aggregiert werden sollen. Wird der Mapper gestartet, initialisiert der Core alle notwendigen Datenstrukturen und Module und lädt mit Hilfe der CAMs alle Alerts der CASs. In regelmäßigen Abständen überprüft er, ob neue Alerts zum System hinzugekommen sind oder entfernt wurden und ob sich die Alert-Policys verändert haben. Wird dabei eine Änderung des Systems festgestellt, so führt der Core die entsprechenden Maßnahmen aus.

Kommt ein neuer Alert hinzu, so bestimmt der Core die Verteilung dieses anhand des Bedingungsbaumes und den Alert-Policys. Dabei versucht er möglichst große Teilbäume des Bedingungsbaumes nahe der Quellen der Metriken zu platzieren. Hierfür wird der Bedingungsbaum Bottom-up durch-

laufen und für jeden Knoten wird überprüft, ob sich dieser auf dem Host seiner Kinder platzieren lässt. Bedingungen besitzen den Host des MA, der die betroffene Metrik an das CAS sendet, als Host. Besitzen die Kinder unterschiedliche Hosts, so wird der Elternknoten nicht lokal platziert. Kann ein Konten nicht lokal platziert werden, sind seine Kinder Wurzeln von Teilbäumen, die lokal platziert werden können. Alle generierten Teilbäume können lokal ausgewertet werden und reduzieren die Auslastung des Netzwerks, da Metriken früh in Alert-Metriken aggregiert werden. Mit Hilfe der Alert-Policys kann auf diesen Vorgang Einfluss genommen werden, z.B. dadurch, dass für einzelne Knoten festgelegt wird, dass diese nicht lokal platziert werden dürfen. Nach der Bestimmung der Verteilung, nutzt der Core die entsprechenden CAMs, LAMs und MAMs zur Verteilung des Alerts. Dies wird im Detail in Abschnitt 5.3 erläutert. Ist ein Alert nicht mehr vorhanden, so entfernt der Core diesen von allen Komponenten. Wurde die Policy eines Alerts verändert, so wird dessen Verteilung neu berechnet und angepasst. Eine Änderung eines Alerts im CAS entspricht einem Entfernen des ursprünglichen Alerts und dem Hinzufügen eines neuen Alerts. Durch die Änderung der Policys kann die Verteilung von Alerts manuell rückgängig gemacht werden. Der Mapper kümmert sich einzig und allein um die Verteilung der Auswertung der Alerts. Die Definition und Ausführung der Aktionen findet weiterhin, wie in den Anforderungen beschrieben, im CAS statt.

5.3. Funktionsweise

Die folgenden Abschnitte zeigen, wie die einzelnen Komponenten zusammenspielen, um lokales Alerting und die Verteilung der Alerts zu ermöglichen. Dafür wird die Verteilung eines Alerts und die Rückgängigmachung einer Verteilung betrachtet.

5.3.1. Lokale Platzierung eines Alerts

Nachdem der Mapper die Verteilung eines Alert festgelegt hat, kann er diese umsetzen. Dafür geht der Mapper wie in Abbildung 5.7 zu sehen vor. Für jeden Teilbaum, der platziert werden soll, werden die folgenden Schritte ausgeführt.

Zunächst generiert ein LAM die Regel für das entsprechende LAS aus dem Teilbaum und fügt diese dem LAS hinzu. Danach ist das LAS bereit, den Teilbaum lokal auszuwerten. Jedoch fehlt diesem noch der Strom an Metriken. Um diesen zu erzeugen, müssen die Konfigurationen aller Agenten angepasst werden, deren Metriken im Teilbaum vorkommen. Die Anpassung findet dabei, wie in Abschnitt 5.2.1 beschrieben, durch die MAMs statt. Ab diesem Zeitpunkt findet eine lokale Auswertung des Baumes statt. Im letzten Schritt muss nun noch die Konfiguration des CAS mit Hilfe eines CAM angepasst werden, damit dieses die aggregierten Metriken und die Alert-Metriken verarbeiten kann. Dafür müssen die Bedingungen der Alert-Defnition im CAS verändert werden. Damit diese Änderungen leicht rückgängig gemacht werden können, speichert das CAM die Bedingungen. Für jeden verteilten Teilbaum werden alle in ihm enthaltenen Bedingungen aus der Definiton entfernt und eine neue hinzugefügt. Diese löst dann aus, wenn das CAS die zugehörige Alert-Metrik empfängt. Die Reihenfolge der einzelnen Schritte ist wichtig, da es sonst zu Inkonsistenzen bei der Auswertung von Alerts kommen kann.

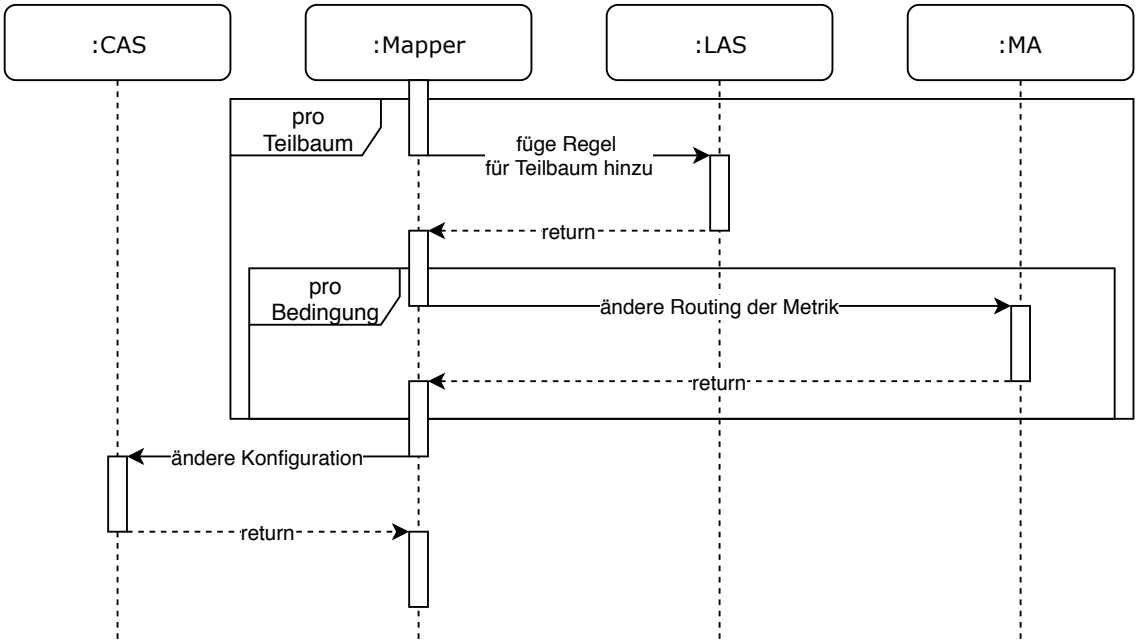

Abbildung 5.7.: Lokale Platzierung eines Alerts

5.3.2. Entfernen eines lokalen Alerts

Um einen verteilten Alert wieder zu zentralisieren, wird wie in Abbildung 5.8 vorgegangen. Für jeden Teilbaum, der zuvor platziert wurde, werden folgenden Änderungen vorgenommen. Zunächst wird das Routing der Metriken auf den MAs, die bei der Platzierung verändert wurden, durch die MAMs angepasst. Da nun der Datenstrom zum LAS nicht mehr vorhanden ist, kann die entsprechende Regel durch das LAM entfernt werden. Zuletzt wird das CAS angepasst, da es von nun an nicht mehr mit den aggregierten Metriken oder Alert-Metriken arbeiten muss. Dazu werden die veränderten Alert-Bedingungen durch die im CAM zwischengespeicherten ersetzt. Das CAS ist nun wieder vollständig für die Auswertung des Alerts verantwortlich.

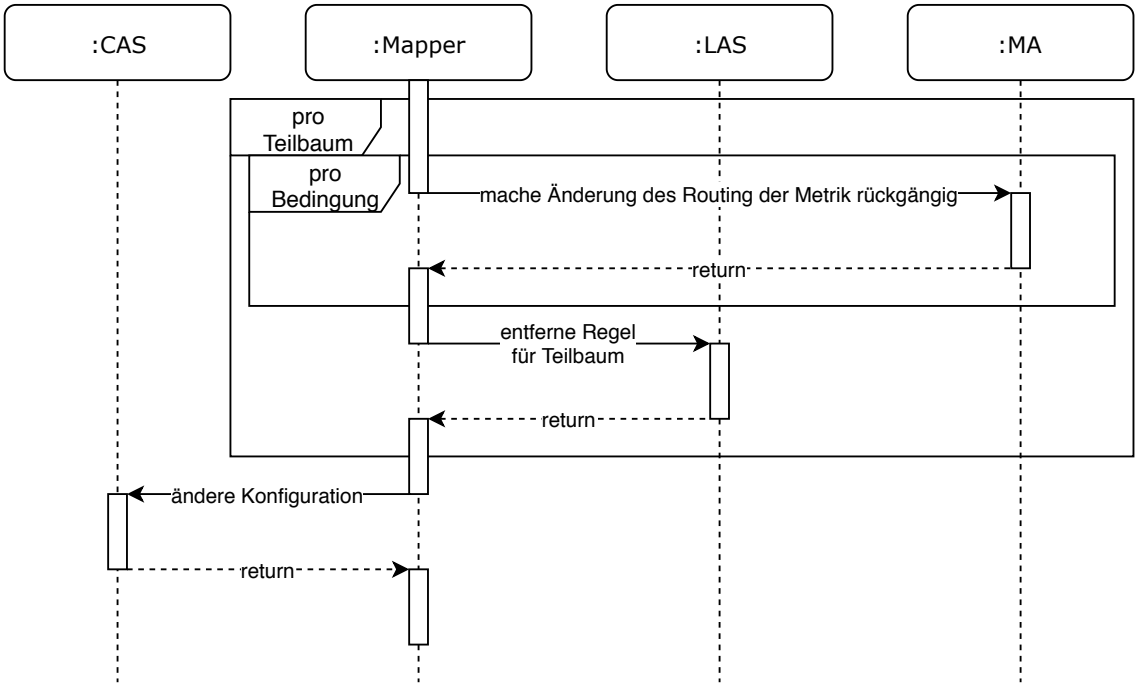

Abbildung 5.8.: Entfernen eines lokalen Alerts

6. Implementierung

Um die Anwendbarkeit des oben beschriebenen Konzepts zu zeigen, wurde dieses für ausgewählte Komponenten umgesetzt. Die Aufbau des implementierten Systems ist in Abbildung 6.1 zu sehen. Dabei wird Telegraf [Infc] als MA eingesetzt, da dieser leichtgewichtig und ausgiebig konfigurierbar ist. Als CAS wurde Grafana [Gra] ausgewählt, welches auf InfluxDB [Infa] als Speicher für Metriken zugreift. Bei dem eingesetzen LAS handelt es sich um eine ESPER-CEP-Engine [Esp], welche um ein HTTP Schnittstelle erweitert wurde. Der Mapper und dessen Module wurden in Java 8 komplett neu entwickelt. Dadurch lässt sich dieser einfach auf unterschiedlichen Systemen einsetzen.

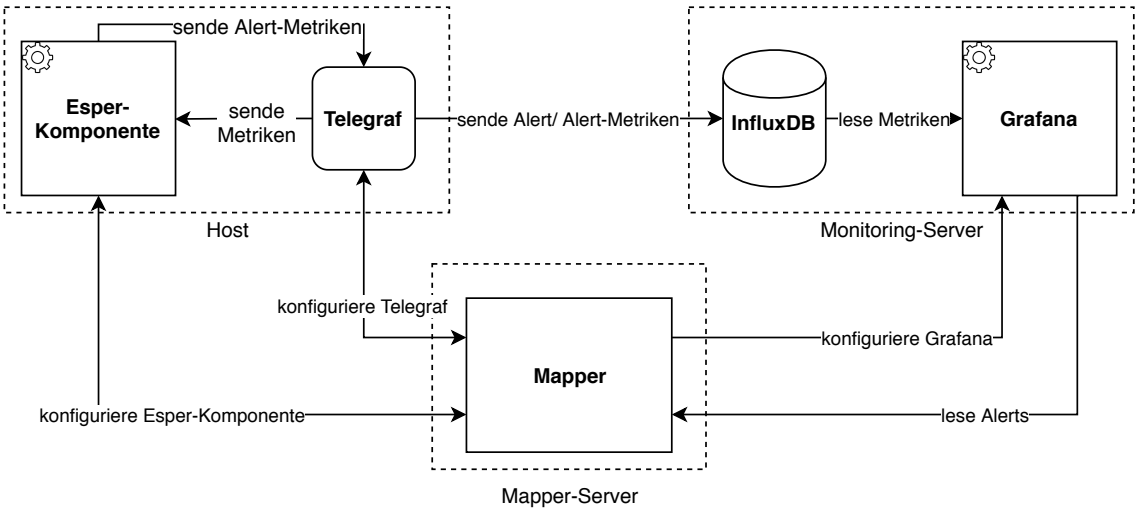

Abbildung 6.1.: System der Implementierung

Die Auswahl der Komponenten beschränkt den Anspruch des Konzepts, eine möglichst generische Lösung zu bieten, nicht, da andere Komponenten durch die Entwicklung der entsprechenden Konfigurationsmodule eingebunden werden können. Die Funktionsweise der einzelnen Komponenten und wie diese im Konzept eingesetzt werden, wird in den folgenden Abschnitten detailliert beschrieben.

6.1. InfluxDB

InfluxDB ist die Timeseries Datenbank des TICK Stack [Infb] und dient in der Implementierung als Metric Data Store. Der eingesetzte MA, Telegraf, entstammt ebenfalls diesem Technologie-Stack. Deshalb wird davon ausgegangen, dass Metriken in der Implementierung immer in dem von InfluxDB vorgegebenen Format vorliegen. Metriken bestehen aus ihrem Measurement, einer beliebigen Anzahl von Tags, mindestens einem Feld und einem optionalen Zeitstempel. Tags und

Felder sind Key/Value Paare. Die Werte von Tags sind auf Strings beschränkt, während sich bei Feldern auch Zahlenformate und Booleans nutzen lassen. Aus diesem Format ergibt wiederum das InfluxDB Line Protocol, welches zur Übertragung von Metriken zwischen den Komponenten eingesetzt wird. Soll eine Metrik mit dem Measurement *cpu*, dem Tag *host* , einem Feld *usage* und einem Zeitstempel mit Wert 0 übertragen werden, so wird folgende Zeile versendet:

cpu,host=”127.0.0.1” usage=90 0

Um Metriken von Telegraf empfangen zu können stellt InfluxDB eine HTTP Schnittstelle bereit, an welche die MAs ihre Metriken schicken können. Dieselbe Schnittstelle wird von Grafana genutzt um die Metriken auszulesen.

6.2. Telegraf

Telegraf ist der pluginbasierte MA des TICK-Stack und lässt sich mittels einer Konfigurationsdatei im TOML [Pre] Format konfigurieren. Er besteht aus mehreren Plugins, welche sich in die vier Plugin-Typen Input-Plugin, Processor-Plugin, Aggregator-Plugin und Output-Plugin gliedern lassen. Input-Plugins bestimmen welche Metriken der Agent verarbeitet, dabei kann ein Input-Plugin mehrere Metriken liefern. Metriken können aus unterschiedlichen Quellen, wie z.B. Datenbanken, Logdateien oder auch HTTP-Schnittstellen, stammen. Die Processor-Plugins ermöglichen es diese zu Filtern oder deren Struktur zu verändern. Mit Hilfe der Aggregator-Plugins lassen sich Metriken auf unterschiedliche Arten aggregieren. Dabei können die ursprünglichen Metriken verworfen oder behalten werden. Durch die Verwendung verschiedener Output-Plugins können die Metriken an unterschiedliche Ziele gesendet werden. Telegraf bietet bereits eine große Auswahl an unterschiedlichen Plugins und lässt sich durch eigene Plugins beliebig erweitern. Ohne ein Routing der Metriken innerhalb des Agenten, würden diese zunächst alle Processor-Plugins und Aggregator-Plugins durchlaufen, um anschließend von allen konfigurierten Output-Plugins an alle Ziele versendet zu werden. Daher muss ein Routing stattfinden, um dem Konzept zu entsprechen.

Um ein Routing von Metriken umzusetzen, muss man die zur Verfügung stehenden Metrik-Filter verwenden. Diese lassen sich für jedes Plugin individuell konfigurieren und ermöglichen es so, den Fluss einer Metrik zu bestimmen. Folgende Filter werden benötigt:

namepass: Nur Metriken mit entsprechenden Namen werden durch das Plugin verarbeitet.

tagpass: Nur Metriken bei denen die Werte der Tags mit den Definierten übereinstimmten, werden durch das Plugin verarbeitet. Sind mehrere tagpass-Filter für ein Plugin konfiguriert, so sind diese durch ein logisches Oder verknüpft.

tagpdrop: Funktioniert umgekehrt wie der tagpass-Filter. Metriken mit Übereinstimmungen werden nicht vom Plugin verarbeitet.

tagexclude: Entfernt die entsprechenden Tags von einer Metrik.

Zusätzlich bieten einige Plugins die Möglichkeit, Tags zu Metriken hinzuzufügen. Mit Hilfe der Filter und dem hinzufügen von Tags, lässt sich ein Routing durchführen. Wie dies im Detail funktioniert erklären die folgenden Abschnitte.

In Abbildung 6.2 ist der Aufbau und Informationsfluss eines einfachen Telegraf-Agenten zu sehen. Dieser besteht aus einem Mem Input-Pluin, welches in der Metrik **mem**, Werte über die Speicherauslastung des Hosts erfasst, und einem InfluxDB Output-Plugin, welches Metriken an eine InfluxDB Instanz sendet. Rechts daneben befindet sich Listing 6.1, welches einen Auszug aus der Konfigurationsdatei, eines Telegraf-Agenten mit entsprechendem Informationsfluss, zeigt. Der Agent sendet Metriken alle 10 Sekunden an die angegebene Adresse der InfluxDB Instanz. Damit die Quelle der Metrik leichter identifiziert werden kann, fügt er jedem Datenpunkt den Tag ”host_address”mit seiner Adresse hinzu. Will man nun für die Metrik **mem** lokales Alerting nutzen,

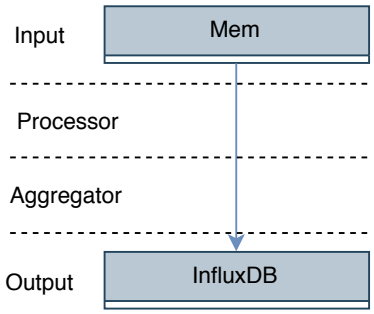

Abbildung 6.2.: Telegraf-Agent

```
[global_tags]
  host_address = "127.0.0.1"

[agent]
  interval = "10s"

[[inputs.mem]]

[[outputs.influxdb]]
  urls = ["http://127.0.0.1:8086"]
```

Listing 6.1: Konfiguration eines Telegraf-Agenten

so muss man die Konfiguration des Telegraf-Agenten, ähnlich wie in Abschnitt 5.2.1 beschrieben, anpassen. Dafür müssen Plugins zur Konfiguration hinzugefügt und bestehende angepasst werden. Zusätzlich muss ein Routing zwischen den Plugins eingeführt werden. Abbildung 6.3 zeigt den Informationsfluss des Agenten nach den notwendigen Anpassungen. Die grau gefärbten Rechtecke entsprechen Plugins und deren Zeilen beschreiben, wie diese die Tags der Metriken verändern, die von ihnen verarbeitet wurden. Es sind nur die für das Routing notwendigen Tags aufgeführt. Ein **+** in einer Zeile bedeutet, dass ein Plugin der Metrik den Tag mit dem angegebenen Wert hinzufügt und ein **-** entspricht dem **tagexclude**-Filter, welcher einen Tag, ohne Berücksichtigung des Wertes, entfernt. Ist kein Zeichen vorhanden, ändert das Plugin den Wert des Tags. Die beiden neu hinzugefügten Plugins, HTTP-Listener-Input-Plugin und HTTP-Output-Plugin, entsprechen dem Alert-Backloop bzw. dem LAS Output aus Abbildung 5.3. Durch die Anbindung der LASs mittels HTTP wird eine flexiblere Verteilung von Alerts ermöglicht, da auch LASs auf anderen Hosts als Ziel von Metriken verwendet werden können. Das neue Override-Processor-Plugin wird für das Routing benötigt, während das Basicstats-Aggregator-Plugin für das Aggregieren der Metrik zuständig ist. In dieser Konfiguration bildet das Basicstats-Plugin alle 30 Sekunden den Mittelwert der Felder der verarbeiteten Metriken. Neben dem Basicstats-Plugin könnten hier auch beliebige andere Aggregator-Plugins eingesetzt werden.

Rechtecke mit blauer Färbung sind Metrik-Filter vom Typ **tagpass**,**tagdrop** oder **namepass**. Ihre Zeilen enthalten die Filterkriterien, die Metriken erfüllen müssen, um vom nachfolgenden Plugin verarbeitet zu werden. Die farbigen Pfeile zeigen die unterschiedlichen Routen der Mertiken durch den Agenten. Rote Pfeile entsprechen dem Fluss der Alert-Metriken, die vom LAS wieder an den Agenten gesendet werden, um von diesem weitergeleitet zu werden. Sie erhalten vom Input-Plugin den Tag *to_cep=false*, weshalb von dem Override- und Basicstats-Plugins ausgefiltert werden und nur an die InfluxDB Instanz gesendet werden. Die blauen Pfeile entsprechen der Route der Metrik

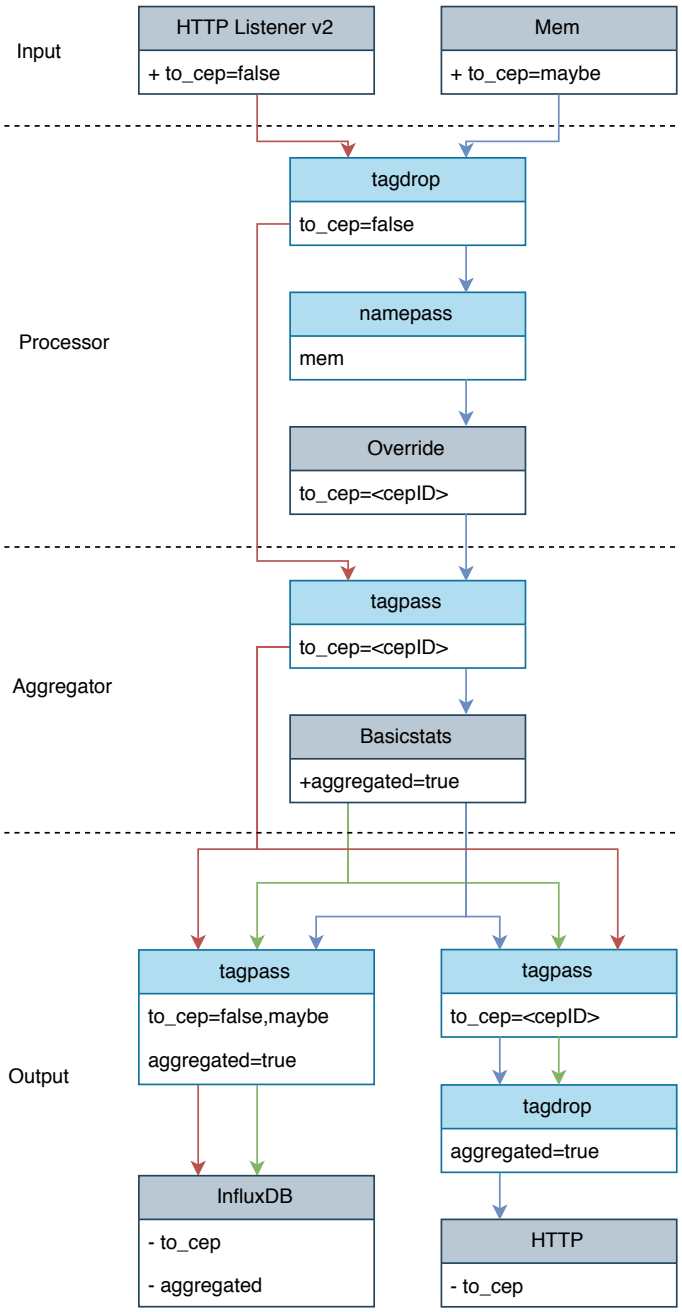

Abbildung 6.3.: Telegraf-Agent Routing

```
[[inputs.mem]]
  [inputs.mem.tags]
    to_cep = "maybe"

[[inputs.http_listener_v2]]
  service_address = ":9999"
  [inputs.http_listener_v2.tags]
    to_cep = "false"

[[processors.override]]
  namepass = ["mem"]
  [processors.override.tags]
    to_cep = "cepID"
  [processors.override.tagdrop]
    to_cep = ["false"]

[[aggregators.basicstats]]
  period = "30s"
  drop_original = false
  stats = ["mean"]
  [aggregators.basicstats.tags]
    aggregated = "true"
  [aggregators.basicstats.tagpass]
    to_cep = ["cepID"]

[[outputs.influxdb]]
  urls = ["http://127.0.0.1:8086"]
  tagexclude = ["to_cep","aggregated"
]
  [outputs.influxdb.tagpass]
    to_cep = ["false","maybe"]
    aggregated = ["true"]

[[outputs.http]]
  url = "http://127.0.0.1:10101/event
/influxdb"
  tagexclude = ["to_cep"]
  [outputs.http.tagpass]
    to_cep = ["cepID"]
  [outputs.http.tagdrop]
    aggregated = ["true"]
```

Listing 6.2: Konfiguration Routing

mem. Ihr wird zu Beginn der Tag *to_cep=maybe* hinzugefügt. Im Input-Plugin kann keine direkte Zuweisung einer Mertik zu einem LAS stattfinden, da ein Input-Plugin unterschiedliche Metriken erzeugen kann. Daher findet die Zuordnung erst im Override-Plugin und nach der Anwendung dessen **namepass**-Filters statt. Grüne Pfeile zeigen den Fluss der Aggregate des Basicstats-Plugins. Da Aggregate alle Tags ihrer ursprünglichen Metriken enthalten wird der Tag *aggregated=true* benötigt um diese voneinander zu unterscheiden. Dies ist auch der Grund, wieso der **tagdrop**-Filter vor dem HTTP-Output-Plugin verwendet werden muss.

Rechts neben der Abbildung befindet sich in Listing 6.2 die für das Routing benötigte Konfigurationsdatei. Diese Art der Konfiguration kann für mehrere Metriken verwendet werden. Hierfür müssen lediglich einige Filter angepasst und evtl. Plugins hinzugefügt werden. Damit der Telegraf die angepasste Konfiguration verwendet muss er neu gestartet werden.

6.3. Esper-Komponente

Die Esper-Komponente ist eine Umsetzung des in Abschnitt 5.2.2 beschriebenen LAS. Sie wurde in Java entwickelt und besteht im Kern aus der Esper CEP-Engine, welche um eine HTTP-Schnittstelle erweitert wurde. Esper besitzt eine Event Processing Language (EPL), welche den SQL Standard erweitert. Mit ihr lassen sich Esper-Queries für Eventströme schreiben. Metriken, die von den MAs an die LASs geschickt werden, sind ein solcher Eventstrom. Die EPL kann genutzt werden um Queries zu erstellen, die den Bedingungsbäumen der Alerts entsprechen. Um das Erstellen von Queries für bestimmte Eventströme zu vereinfachen, lassen sich diese in Esper durch Schemata beschreiben. Dies kann z.B. in Form von Java-Klassen geschehen. Da in der Implementierung alle Metriken im InfluxDB-Format verarbeitet werden, wurde ein Schema für diesen Eventstrom als Java Klasse erstellt. Diese trägt den Namen InfluxDBEvent und besitzt Variablen für alle Bestandteile des Formats. Sobald Esper dieses Schema besitzt, können InfluxDBEvent-Objekte an Esper übergeben und von dieser verarbeitet werden. Hat man CPU-Metriken, wie die aus dem Beispiel in Abschnitt 6.1, und eine Bedingung, die eintritt sobald *usage* den Wert 90 übersteigt, so sieht eine entsprechende Query auf den Strom von InfluxDBEvents folgendermaßen aus.

$$
\begin{aligned}
&SELECT * \\
&FROM\ InfluxDBEvent(measurement = 'cpu') \\
&WHERE\ cast(fields('usage'), BigDecimal) > 90
\end{aligned}
$$

Mit '*' wird das gesammte Event selektiert, das die Bedingungen erfüllt. Die Umwandlung des *usage* Felds ist nötig, da InfluxDBEvent-Objekte für Feldwerte den Datentyp Object vorsehen. Dies hängt damit zusammen, dass Felder im InfluxDB-Format unterschiedliche Typen besitzen können. Mehrere solcher Queries lassen sich in einem Esper-Statement zusammenfassen, welchem man einen Namen geben kann. Damit Statements von der CEP-Engine verarbeitet werden können, werden sie vom Esper-Compiler in Bytecode umgewandelt und anschließend von der Esper-Runtime ausgeführt. Um Alert-Metriken zu generieren erlaubt es Esper, Subscriber, in Form von Java-Objekten, für Queries zu definieren. Diese müssen eine *update* Methode besitzen, welche aufgerufen wird, sobald ein Event von einer Query erfasst wird. Dieser können je nach Spezifikation die Ergebnisse der Query übergeben werden. Dadurch ist es möglich unterschiedliche Alert-Metriken für unterschiedliche Alert-Bäume zu generieren und an den lokalen MA zurückzusenden.

Die hinzugefügte HTTP-Schnittstelle erlaubt es der Komponente, Metriken von unterschiedlichen Agenten zu empfangen. Zudem kann sie genutzt werden um neue Esper-Statements, bzw. Queries, in Esper zu platzieren oder zu entfernen. Die von der Schnittstelle zur Verfügung gestellten Pfade sind in Tabelle 6.1 aufgeführt. Unter */event/influxdb* können Agenten Metriken im Body einer HTTP-Nachricht an Esper senden. Die Metriken müssen im InfluxDB Line Protocol vorliegen. Um mehrere Metriken auf einmal zu senden, müssen diese durch einen Zeilenumbruch voneinander getrennt sein. Wurde eine Metrik empfangen, so wird diese in ein InfluxDBEvent-Objekt umgewandelt und an

Esper weitergereicht. Soll eine Alert-Bedingung lokal platziert werden, so muss ein entsprechendes Esper Statement an */statement* gesendet werden. Zur Identifikation des Statements, muss es einen eindeutigen Namen besitzen. Nach dem Empfang des Statements wandelt der Esper-Compiler dieses um und übergibt es der Runtime. Gleichzeitig fügt die Komponente der Query einen passenden Subscriber hinzu. Die erzeugte Alert-Metrik trägt die ID des Alerts, der den Bedingungsbaum der Query enthält, als Namen und hat ein Feld, das die ID des Bedingungsbaumens als Namen hat. Dieses Feld hat den Wert 1, um anzuzeigen, dass alle Bedingungen des Bedingungsbaumes eingetreten sind. Zum Löschen einer Query genügt es, den Namen des zugrundeliegenden Statements zu kennen und mit diesem */statement/{name}* aufzurufen.

Methode	Pfad	Beschreibung
POST	/event/influxdb	Senden von Metriken an Esper
POST	/statement	Hinzufügen eines Esper-Statements
DELETE	/statement/{name}	Entfernen eines Esper-Statements

Tabelle 6.1.: HTTP-Schnittstelle der Esper-Komponente

6.4. Grafana

Als CAS wird Grafana eingesetzt. Grafana ist eine Open Source Analyseplattform die Alerting unterstützt. Sie erlaubt es Nutzern, Metriken aus unterschiedlichen Quellen auf Dashboards darzustellen und Alerts für diese zu definieren. Jedes Dashboard besitzt Panels, welche es ermöglichen, Metirken zu visualisieren. Abbilung 6.4 zeigt ein Panel eines Grafana-Dashboards, das den Verlauf einer CPU-Metrik abbildet. Die rote Linie ist die Schranke eines Alerts. Panels können unterschiedliche Visualisierungen bieten. Beispielweise Kurvendiagramme oder Tabellen. Grafana unterstützt von Haus aus mehrere Typen von Datenquellen, aus denen Metriken ausgelesen werden können, darunter Graphite, MySQL und InfluxDB. Queries legen fest, welche Metriken aus den Datenquellen extrahiert und dargestellt werden sollen. Panels besitzen mindestens eine Query. Durch die Erweiterung mit Plugins lässt sich Grafana an neue Quellen anpassen. Alerts sind in Grafana für einzelne Panels definiert. Diese bestehen aus mehreren Bedingungen und einer Nachricht. Diese werden über konfigurierte Nachrichtenwege verschickt wird, falls alle Bedingungen eines Alerts erfüllt sind. Nachrichtenwege können z.B. E-Mail oder Slack-Benachrichtigungen sein. Bedingungen legen Schranken für die Werte der Metriken fest, die durch die Queries aus den Datenquellen extrahiert wurden. Sie können durch die boolschen Operatoren OR und AND sequentiell verknüft werden.

Grafana verwendet, in der Implementierung, InfluxDB als einzige Datenquelle. Um die Konfiguration einer Grafana-Instanz aus der Ferne zu ändern und Alerts auslesen zu können, bietet sie eine HTTP-Schnittstelle, die JSON als Datenformat verwendet, an. Alle für die Implementierung notwendigen Pfade der API sind in Tabelle 6.2 aufgelistet. Grafana besitzt keine Schnittstelle über die man Alerts auslesen und ändern kann, daher müssen diese aus den vorhandenen Dashboards, bzw. deren Panels, extrahiert werden. Dafür können die JSON-Repräsentationen der Dashboards genutzt werden, die über die Schnittstelle ausgelesen werden können. Eine vereinfachte Version, mit allen für die Implementierung wichtigen Bestandteilen, einer JSON-Repäsentation ist in Listing A.2 zu sehen. Wie diese vom Mapper genutzt wird, ist in Abschnitt 6.5.2 beschrieben. Über den Pfad */api/dashboards/uid/{uid}* erhält man die JSON-Repräsentation eines Dashboards anhand

Abbildung 6.4.: Panel eines Grafana-Dashboards

dessen eindeutiger ID (UID). Die UID ist für eine Grafana-Instanz eindeutig. Durch Aufrufen von */api/search/* mit Parametern kann eine Grafana-Instanz durchsucht werden. Allerdings benötigt die Implementierung nur die Suche ohne Parameter, um die UIDs aller Dashboards zu erhalten. Wurde die JSON-Repräsentation eines Dashboards vom Mapper angepasst und soll die alte überschrieben, so muss diese an */api/dashboards/db* gesendet werden. Wie und wann sie angepasst werden müssen ist in Abschnitt 6.5.2 beschrieben. Wenn sich die Quellen der Metriken von Alert-Bedingungen nicht direkt aus den Grafana-Queries ablesen lassen, dann besteht die Möglichkeit diese aus einer InfluxDB-Instanz auszulesen. Dafür können die Zugangsdaten der InfluxDB-Instanz über */api/datasources* ermittelt werden.

Methode	Pfad	Beschreibung
GET	/api/search/	Liefert Informationen zu allen Dashboards
POST	/api/dashboards/db	Erstellen oder Überschreiben eines Dashboards
GET	/api/dashboards/uid/{uid}	Liefert das Dashboard mit uid
GET	/api/datasources	Liefert Beschreibungen aller Datenquellen

Tabelle 6.2.: Verwendete Methoden der Grafana-HTTP-API

6.5. Mapper

Für die Implementierung des Mappers und dessen Modulen wurde Java gewählt. Die Struktur der Implementierung ist in Abbildung 6.5 zu sehen und folgt der des Konzepts. Der Core ist auch in der Implementierung der Mittelpunkt des Mappers und für die Verteilung der Alerts zuständig. Das Telegraf-Modul ist die Implementierung eines MAM und dient zur Konfiguration der oben vorgestellten Telegraf-Agenten. Ebenso sind das Esper- und das Grafana-Modul die Implementierungen eines LAM und CAM. Sie ermöglichen die Konfiguration der Esper-Komponenten und Grafana-Instanzen. Alle Konfigurationsmodule sind über die Mapper-API aus Abschnitt 6.5.1 mit

dem Core verbunden und nutzen die zuvor beschriebenen Schnittstellen der externen Komponenten. Ihre Funktionsweisen werden in den folgenden Abschnitten beschrieben. Für die Umsetzung der Modularität wurde auf das Plugin Framework for Java (PF4J) [PF4] zurückgegriffen. Dieses ermöglicht es, die Module und deren Schnittstellen dynamisch zu managen. Dadurch kann der Core mit den unterschiedlichsten Komponenten verwendet werden, solange diese die in der Mapper-API definierten Interfaces zur Verfügung stellen. Die Mapper API fasst alle für die Kommunikation zwischen Core und Konfigurationmodulen notwendigen Methoden zusammen. Das interne Datenmodell eines Alerts im Mapper ist ein Java-Objekt, dessen Struktur einer Erweiterung der in Abschnitt 5.2.5 vorgestellten Datenstruktur entspricht. Für die Implementierung wird davon ausgegangen, dass jeder Host bereits eine Esper-Komponente besitzt und der Mapper sich nicht um deren Deployment kümmern muss.

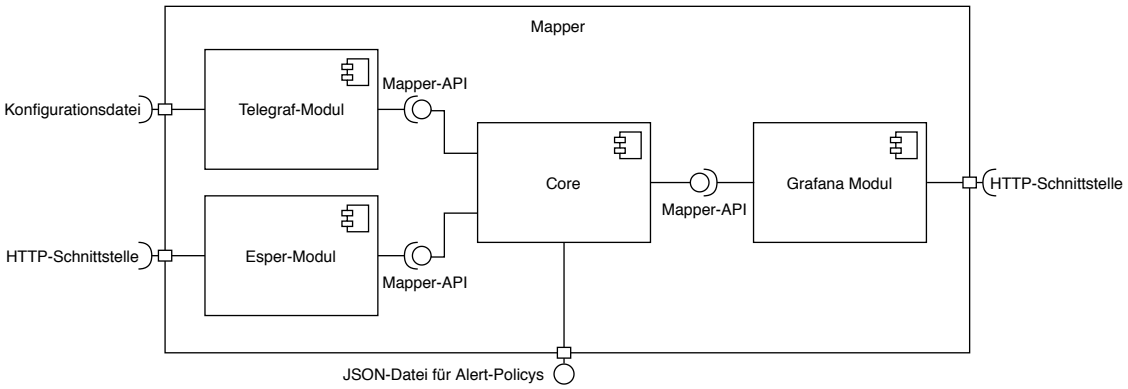

Abbildung 6.5.: Aufbau der Mapper Implementierung

Für Alert-Policys bietet der Mapper eine Schnittstelle durch eine JSON Datei auf die der Core zugreift. Diese kann dafür genutzt werden, den Knoten des Bedingungsbaumes einen Host zuzuweisen, oder die Art der Aggregation der Metriken festzulegen. Dabei kann z.B. bestimmt werden, dass Teilbäume auf bestimmten Hosts platziert werden müssen, oder dass keine Verteilung des Alerts erwünscht ist. Listing A.1 zeigt die Policys für einen einfachen Alert. Dessen Bedingungsbaum besteht aus zwei Bedingungen, die über einen Knoten miteinander verbunden sind. Für die Metriken beider Bedingungen soll, im Falle einer Aggregation, der Mittelwert alle 30 Sekunden berechnet werden. Beiden Bedingungen, bzw. Blättern des Baumes, ist die Adresse eines Hosts zugeordnet, während für den Elternknoten keine Platzierung vorgesehen ist. Anhand der Versionsnummer einer Policy kann leicht zwischen alten und neuen Policys unterschieden werden. Wurde eine Policy verändert, muss sich ihre Versionsnummer erhöhen.

6.5.1. Mapper-API

Die Mapper-API ist eine Sammlung von Java-Interfaces und lässt sich in zwei Teilbereiche unterteilen. Der erste Bereiche umfasst die ModuleToCoreConnector Interfaces, die definieren welche Methoden die CAMs, LAMs und MAMs zur Verfügung stellen müssen. Sie müssen von den Modulen implementiert werden, damit PF4J diese während der Laufzeit dynamisch laden und starten kann. Die in ihnen definierten Methoden werden benötigt, um die einzelnen Module nach ihrem Start zu konfigurieren. Ihre Hauptaufgabe ist es, dem Core Objekte zur Verfügung zu stellen, die die eingesetzten CASs, LAS und MAs abbilden. Der zweite Teil der API besteht aus den

Connector-Interfaces, die diese Objekte implementieren müssen. Sie bestimmen wie der Core mit den Instanzen der Komponenten interagieren kann. Für CASs, LAS und MAs ist jeweils ein Interface vorgesehen:

Central Alerting System Connector (CASC):

liefereAlerts(): Liefert alle Alerts des CAS zurück.

konfiguriereFürVerteiltenAlert(Alert): Konfiguriert ein CAS so, dass es mit der verteilten Version des Alerts arbeiten kann.

konfiguriereFürZentralenAlert(Alert): Ändert die Konfiguration wieder in den ursprünglichen Zustand vor der Verteilung des Alerts

Local Alerting System Connector (LASC):

platziere(Bedingungsbaum): Konfiguriert ein LAS so, dass dieses die Auswertung des Bedingungsbaumes übernimmt und bei dessen Eintreten eine Alert-Metrik an seinen lokalen MA sendet.

entferne(Bedingungsbaum): Entfernt die Konfiguration für den Bedingungsbaum.

Monitoring Agent Connector (MAC):

routeMetrikZuLAS(Bedingung,LAS): Passt die Konfiguration eines Agenten so an, dass er das Routing für eine Metrik so ändert, dass diese an ein LAS gesendet wird, während aggregierte Werte an das CAS gesendet werden.

entferneRouteZuLAS(Bedingung,LAS): Macht das Routing für die Metrik rückgängig.

Wie die Konfigurationsmodule die Methoden implementieren hängt dabei stark von den eingesetzten Komponenten ab. Durch die Verwendung der Interfaces, kann der Core mit den unterschiedlichsten Komponenten zusammenarbeiten. Für die Implementierung wurden Konfigurationsmodule für Grafana, Telegraf und die Esper-Komponente entwickelt. Diese werden in den folgenden Abschnitten beschrieben.

6.5.2. Grafana-Modul

Das Grafana-Modul ist die Implementierung eines CAMs. Es stellt dem Core Objekte vom Typ GrafanaConnector zur Verfügung, die wiederum Implementierungen des CASC Interfaces sind. Diese kann der Core nutzen um Grafana-Instanzen zu konfigurieren. Dafür nutzen sie die HTTP-Schnittelle von Grafana, die in Abschnitt 6.4 beschrieben wurde. Um auf die Schnittstelle zugreifen zu können, muss eine Grafana Instanz einem GrafanaConnector Access-Tokens mit den nötigen Berechtigungen bereitstellen. Die Tokens nutzt ein GrafanaConnector um sich gegenüber der Grafana-Instanz zu identifizieren. Sollen Alerts aus Grafana ausgelesen werden, müssen diese aus den vorhandenen Dashboards extrahiert werden. Dafür ruft ein GrafanaConnector zunächst den Pfad */api/search* der HTTP-Schnittstelle auf. Dadurch erhält sie eine Liste mit Informationen über alle Dashboards der Instanz. Aus dieser können die UIDs der Dashboards extrahiert werden. Mit Hilfe der UIDs können die JSON-Repräsentationen der Dashboards, über den Pfad */api/dashboards/uid/{uid}*, einzeln abgerufen werden. Dies ist in Abbildung 6.6 dargestellt.

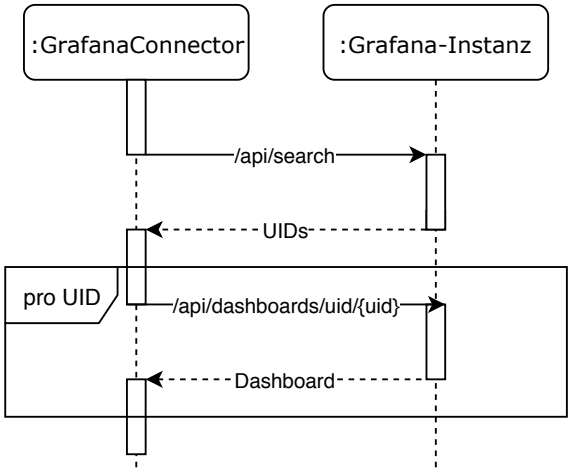

Abbildung 6.6.: Kommunikation des GrafanaConnector

Für die einfachere Handhabung, werden die JSON-Repräsentationen der Dashboards in geschachtelte Java-Objekte umgewandelt, welche im GrafanaConnector verwaltet werden. Aus diesen wiederum können die Alerts extrahiert und in internen Datenstruktur für Alerts überführt werden. Dazu wird über alle Panels eines Dashboards iteriert, für die Alerts definiert sind. Um die Alerts in die interne Datenstruktur zu überführen, müssen unterschiedliche Teile der JSON-Repräsentationen verknüpft werden. Das Listing A.2 zeigt eine vereinfachte Version der Repräsentation. Eine Bedingung entsteht durch die Verknüpfung einer *condition* mit einem *target*, anhand der *refID*, die alle *conditions* und *targets* besitzen. Die *targets* entsprechen den Queries, die Grafana an eine Datenquelle sendet um deren Mertiken zu extrahieren. Wenn sich die Quellen der von Bedingungen überwachten Metriken, nicht direkt aus den *targets* ablesen lassen, können diese aus der InfluxDB ausgelesen werden. Die Zugangsdaten der betroffenen InfluxDB werden über den Pfad */api/datasources* ausgelesen. Da Grafana bisher nur eine sequentielle Auswertung von Bedingungen erlaubt, können nur simple Bedingungsbäume der Form $((((Bed_1 OP_2 Bed_2)\, OP_3 Bed_3)\, OP_4 Bed_4)\, ...)$ entstehen. Die Operatoren der inneren Knoten können den *Conditions* entnommen werden. Damit die Alerts einfacher zu unterscheiden sind, wird eine ID aus der UID des Dashboards und der ID des Panels generiert. Sie ist auf Grund der Eindeutigkeit der UIDs ebenfalls eindeutig. Ebenso wird für jeden Knoten des Bedingungsbaumes eine eindeutige ID anhand der Operatoren und Bedingungen des Teilbaumes, für den der Knoten die Wurzel darstellt, generiert.

Wurde ein Alert eines Dashboardes durch den Mapper verteilt, müssen die *Conditions* und *Targets* des betroffenen Dashboards angepasst werden. Dies geschieht in der Java-Objekt-Repräsentation des Dashboards. Für alle Metriken von Bedingungen, die verteilt wurden, wird eine neue Query für die aggregierten Metriken hinzugefügt. Zusätzlich wird für jeden verteilten Teilbaum des Bedingungsbaumes eine Query hinzugefügt, wenn dieser eine Alert-Metrik für Grafana erzeugt. Außerdem werden für jeden Teilbaum neue Conditions hinzugefügt, die auswerten, ob sich ein Teilbaum im Alerting Zustand befindet. Dies geschieht anhand der Alert-Metriken die bei der Auswertung der Teilbäume generiert werden. Die Operatoren zur Verknüpfung der neuen Conditions können aus dem vollständingen Bedingungsbaum des Alerts entnommen werden. Alte Conditions werden von der GrafanaInstanz zwischengespeichert, um eine Wiederherstellung des Ausgangszustandes zu ermöglichen. Um die alte Version des Dashboards in der Grafana Komponente überschreiben zu können, wird der *override* Flag in den Metadaten des Dashboards auf true gesetzt. Nach die-

sen Änderungen wird das Java-Objekt in seine JSON-Repräsentation umgewandelt und an den Pfad */api/dashboards/db* der Grafana Komponente gesendet. Soll ein Dashboard wieder in seinen Ursprungszustand zurück versetzt werden, genügt es, die zusätzlichen Queries zu entfernen, die aktuellen Conditions durch die gespeicherten zu ersetzten und diese Version an die HTTP-API zu senden. Die oben beschriebenen Abläufe sind die Implementierungen der *liefereAlerts()*-, *konfiguriereFürVerteiltenAlert()*- und *konfiguriereFürZentralenAlert()*-Methoden des CASC Interfaces.

6.5.3. Telegraf-Modul

Das Telegraf-Modul ist die Implementierung eines MAM. Es stellt dem Core für jeden Telegraf-Agenten ein Objekt vom Typ TelegrafConnector als Impementierung des MAC Interfaces zur Verfügung. Soll die Konfiguration eines Agenten angepasst werden sind die in Abbildung 6.7 zu sehenden Schritte erforderlich. Zunächst wird eine Kopie der Konfigurationsdatei vom Host des Agenten geladen, damit diese lokal bearbeitet werden kann. Soll eine Metrik zu einem LAS geroutet werden, wird die Konfiguration wie in Abschnitt 6.2 beschrieben ergänzt. Beim Entfernen des Bedingungsbaumes werden die entsprechenden Schritte rückgängig gemacht. Bei beiden Bearbeitungsschritten muss darauf geachtet werden, dass die Konfiguration für sonstige Metriken weiterhin bestehen bleibt. Ist die Anpassung abgeschlossen, wird die originale Konfiguration auf dem Host überschrieben und der Agent muss neu gestartet werden. Dies sind die Implentierungen der *routeMetrikZuLAS()*- und *entferneRouteZuLAS()*-Metoden des MAC Interfaces.

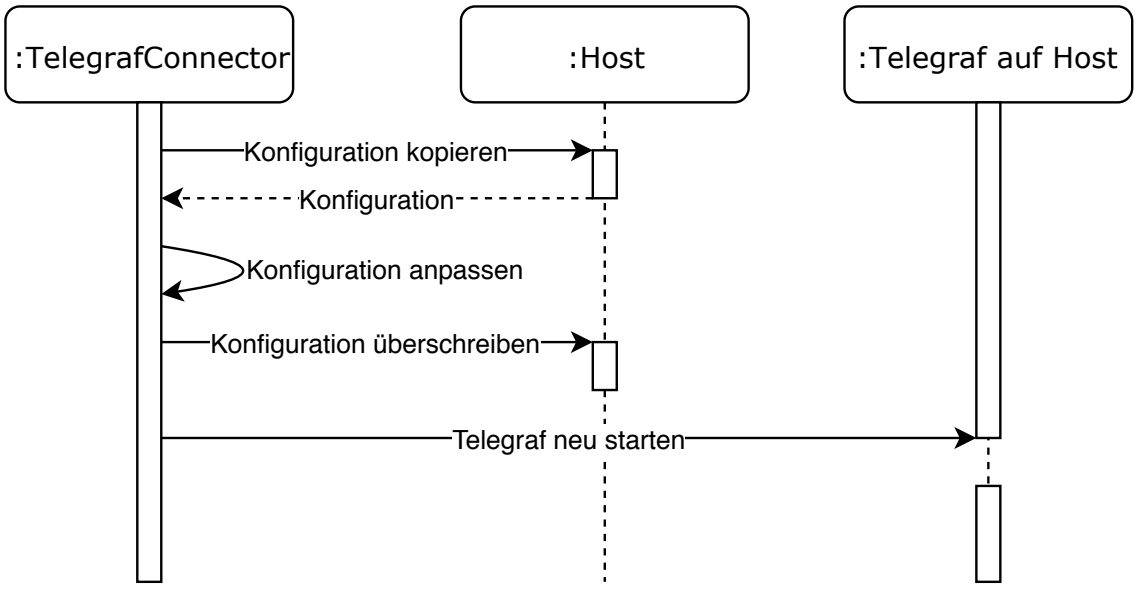

Abbildung 6.7.: Ändern einer Telegraf-Konfiguration

6.5.4. Esper-Modul

Das Esper-Modul ist die Implementierung eines LAM. Es stellt dem Core für jede Esper-Komponente ein Objekt vom Typ EsperConnector als Implementierung des LASC Interface zur Verfügung. Dieses besitzt die Methoden *platziere(Bedingungsbaum)* und *entferne(Bedingungsbaum)* um Bedingungs-

bäume auf den Komponenten zu platzieren oder diese Platzierung Rückgängig zu machen. Um einen Bedingungsbaum zu platzieren generiert das Modul eine Esper-Query die diesen abbildet. Dafür wird der Bedingungsbaum durchlaufen und für alle Blätter werden Subqueries erzeugt die mit den Operatoren der Knoten verknüpft werden. Diese Query wird mit dem Namen des Baumes versehen und an die in Abschnitt 6.3 beschriebene HTTP-API gesendet. Soll diese Platzierung rückgängig gemacht werden, so genügt es, den Namen des Bedingungsbaumes an die HTTP-API der Esper-Komponente zu senden.

6.5.5. Core

Der Core ist das zentrale Modul des Mappers. Er ist für die Verteilung der Alerts, bzw deren Bedingungsbäume, zuständig und nutzt die Connector-Objekte, die er von den oben beschriebenen Konfigurationsmodulen erhält. Der Ausgangspunkt für die Verteilung der Alerts sind die CAS für die er eine Verteilung vornehmen soll. Diese kann er nicht selbst ermitteln, weshalb sie zum Start des Mappers konfiguriert sein müssen. Dafür können die Adressen und Zugangsdaten unterschiedlicher CASs in einer Konfigurationsdatei im TOML-Format hinterlegt werden. Zusätzlich kann hier konfiguriert werden, wie häufig der Mapper die Verteilung überprüft.

Wird der Mapper gestartet, durchläuft der Core zunächst eine Initialisierung die in Algorithmus 6.1 zu sehen ist. Zunächst werden alle Module die sich im */plugins*-Verzeichnis des Mappers befinden geladen und gestartet. Danach werden die CASCs anhand der in der Konfigurationsdatei spezifizierten CASs, durch entsprechende Module, initialisiert und deren Alerts ausgelesen. Da die Bedingungen der Alerts die Adressen der Quellen, der Metriken, enthalten und diese mit denen der Hosts übereinstimmen, die MAs und LAS beherbergen, können sie dafür genutzt werden, die entsprechenden Objekte zu initialisieren. Abschließend werden Standard Policys für alle Alerts generiert und in eine Datei geschrieben. Dafür werden die Bedingungsbäume von oben nach unten durchlaufen und für alle inneren Knoten keine *deployOn* Adresse festgelegt. Für Blätter, bzw. Bedingungen, werden für *deployOn* die Adressen der Quellen, der betroffenen Metrik, eingetragen. Zusätzlich wird das Intervall für die Aggregation auf 30 Sekunden gesetzt und der Mittelwert als Methode festgelegt.

Algorithmus 6.1 Initialisierung des Mappers

```
ladeUndStarteModule()
Konfiguration ← ladeKonfigurationAusDatei()
CASCs ← initialisiereCASCs(Konfiguration)
Alerts ← leer
for all CASC ∈ CASCs do
    Alerts.add(CASC.Alerts)
end for
MACs ← initialisiereMACs(Alerts)
LASCs ← initialisiereLASCs(Alerts)
AlertPolicys ← generiereDefaultAlertPolicys(Alerts)
schreibeAlertPolicysInDatei(AlertPolicys)
```

Hat der Core die Initialisierung abgeschlossen, startet er die Programmschleife aus Algorithmus 6.2. Diese wird solange wiederholt, bis der Mapper gestoppt wird. Zunächst wird ermittelt, für welche Alerts eine Anpassung der Verteilung stattfinden muss. Dafür werden die aktuellen Alerts aus den

CASCs ausgelesen. Diese werden dazu verwendet, zu ermitteln, welche Alerts neu hinzugekommen sind und welche gelöscht wurden. Um herauszufinden, ob die Verteilung von bereits im System vorhandenen Alerts geändert werden muss, werden die aktuellen Policys aus der Datei geladen. Für jeden bestehenden Alert wird überprüft, ob sich seine Policys geändert haben. Dies ist dann der Fall, wenn sich die Versionsnummer der Policys eines Alerts erhöht hat. Danach kann die Verteilung aller gelöschten, neuen und geänderten Alerts angepasst werden. Für neue Alerts müssen dabei möglicherweise neue MACs und LASC initialisiert werden. Die eigentliche Anpassung der Verteilung findet in den Methoden *VerteileAlert* und *MacheAlertVerteilungRückgängig* statt, die in Algorithmus 6.3 und 6.4 zu sehen sind. Zum Abschluss der Schleife werden die aktuellen Policys in die Datei zurückgeschrieben und es wird für die in der Konfiguration festgelegte Zeit gewartet.

Algorithmus 6.2 Programmschleife des Mappers

```
while Core nicht gestoppt do
    AlertsAktuell ← leseAlerts(CASCs)
    AlertsNeu ← AlertsAktuell\Alerts
    AlertsGelöscht ← Alerts\AlertsAktuell
    AlertPolicysAktuell ← ladeAlertPolicysAusDatei()
    AlertsGeändert ← leer
    for all Alert ∈ Alerts do
        if AlertPolicys(Alert).Version < AlertPolicysAktuell(Alert).Version then
            AlertsGeändert.add(Alert)
        end if
    end for
    Alerts ← AlertsAktuell
    AlertPolicys ← AlertPolicysAktuell
    for all Alert ∈ AlertsGelöscht do
        MacheVerteilungRückgängig(Alert)
        AlertPolicyes.entfernePolicy(Alert)
    end for
    for all Alert ∈ AlertsNeu do
        AlertPolicys.generiereDefaultPolicyFür(Alert)
        if Neue MACs für Alert benötigt then
            MACs.add(initialsiereMASCs(Alert))
        end if
        if Neue LASCs für Alert benötigt then
            LASCs.add(initialsiereLASCs(Alert))
        end if
        VerteileAlert(Alert)
    end for
    for all Alert ∈ AlertsGeändert do
        MacheVerteilungRückgängig(Alert)
        VerteileAlert(Alert)
    end for
    schreibeAlertPolicysInDatei(AlertPolicys)
    warte()
end while
```

Beide Methoden zur Verteilung entsprechen in ihrer Funktionsweise der des Konzepts aus Abschnitt 5.3. Für die Verteilung werden zunächst die Teilbäume aus dem Bedingungsbaum generiert, die auf die Hosts verteilt werden können. Dafür werden die *deployOn*-Einträge aus den Policys auf die Knoten des Bedingungsbaumes übertragen und versucht für jeden inneren Knoten, der noch keinen Bestimmungsort hat, diesen aus den *deployOn*-Einträge seiner Kindknoten zu generieren. Dabei wird der Baum von den Blättern zur Wurzel durchlaufen. Wie dies für einen Bedingungsbaum ohne Policy-Einträge funktioniert, ist in Abbildung 6.8 zu sehen. Links ist die Ausgangslage dargestellt, bei der nur den drei Bedingungen ein*deployOn*-Wert zugeordnet ist. Dieser entspricht der Adresse der Quelle der jeweiligen Bedingung. Rechts ist das Ergebnis nach der Generierung der Einträge für innere Knoten zu sehen. Auf den Kanten sind die Werte der Kindknoten eingetragen. Der Elternknoten von Bed_1 und Bed_2 kann auf Host A platziert werden, da beide Kindknoten auf diesem platziert werden sollen. Für den Wurzelknoten kann kein *deployOn*-Wert generiert werden, da seine Kinder unterschiedliche Werte aufweisen.

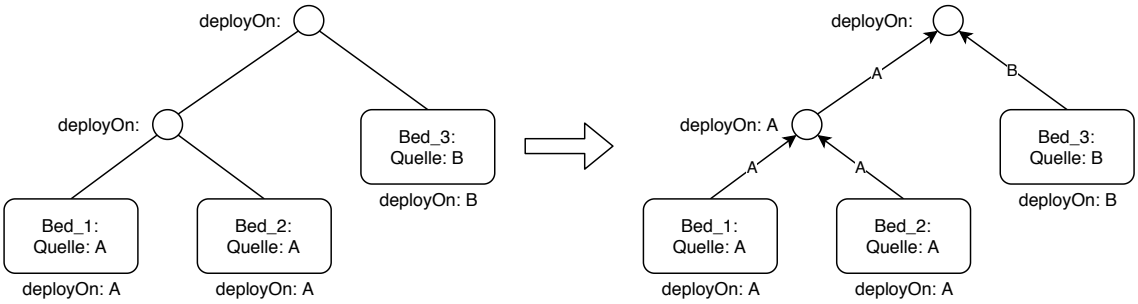

Abbildung 6.8.: Verbreitung der Policy

Nachdem die Policy auf den Baum angewendet wurde, können die zu platzierenden Teilbäume generiert werden. Dafür wird der Baum beginnend bei der Wurzel durchlaufen. Wird ein Knoten erreicht der einen *deployOn*-Wert besitzt ist dieser die Wurzel eines zu platzierenden Teilbaumes. Dieser erstreckt sich über alle Kindknoten die auf dem selben Host platziert werden sollen. Besitzt ein Kindknoten einen anderen *deployOn*-Wert, wird dieser durch eine Proxybedingung ersetzt, die dann eintritt, wenn der Baum des Kindknotens erfüllt wird. Der Kindknoten ist die Wurzel eines weiteren Teilbaumes, bei dem gleich vorgegangen wird. Dies wird in Abbildung 6.9 dargestellt. Auf der linken Seite ist der Bedingungsbaum nach der Anwendung der Policy zu sehen. Alle Knoten sind einem Host zugewiesen. Rechts sind die zwei Teilbäume zu sehen, die nach der oben beschriebenen Methode generiert wurden. Der linke der beiden kann auf Host A und der rechte auf Host B platziert werden. Damit dies möglich ist, musste dem rechten Baum eine Proxybedingung hinzugefügt werden, die dann erfüllt wird, wenn der linke Baum Alert-Metriken produziert. Sind die Teilbäume

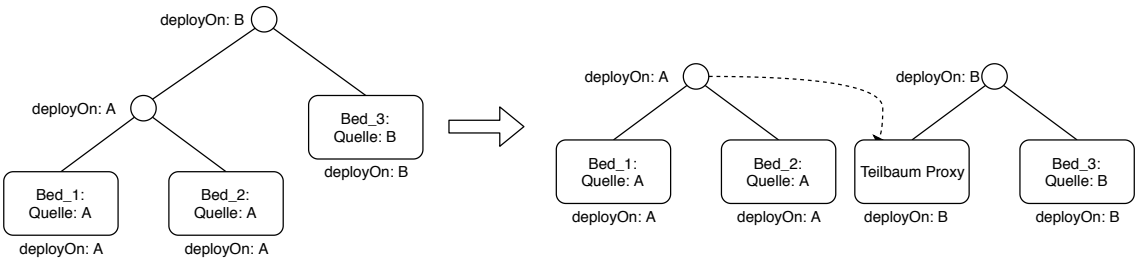

Abbildung 6.9.: Generierung von Teilbäumen mit Proxys

generiert, werden sie auf die LASs mit Hilfe der Methode der LASCs verteilt. Für jede Bedingung eines Teilbaumes wird das Routing des entsprechenden MAs, durch einen MACs, angepasst. Die Reihenfolge ist wichtig, da so das Routing des MA erst geändert wird, wenn die entsprechende Query auf dem LAS läuft. Abschließend wird das CAS des Alerts mit einem CASC angepasst.

Soll die Platzierung eines Alerts aufgehoben werden, müssen alle zuvor getroffenen Änderungen rückgängig gemacht werden. Für jeden Teilbaum wird die entsprechende Query von einem LAS entfernt und für jede Bedingung, wird das Routing der MAs angepasst. Abschließend wird auch hier das CAS angepasst. Wird der Core gestoppt, muss vorher die Verteilung aller Alerts rückgängig gemacht werden, da es sonst zu Inkonsistenzen kommen kann.

Algorithmus 6.3 Verteilung eines Alerts

```
procedure VerteileAlert(Alert)
    Teilbäume ← generiereZuPlatzierendeTeilbäume(Alert.Bedingungsbaum)
    for all Teilbaum ∈ Teilbäume do
        LASCs(Teilbaum.deployOn).platziere(Teilbaum)
        Bedingungen ← extrahiereBedingungenAus(Teilbaum)
        for all Bedingung ∈ Bedingungen do
            MACs(Bedingung.Quelle).routeMetrikZuLAS(Bedingung)
        end for
    end for
    CASCs(Alert).konfiguriereFürVerteiltenAlert(Alert)
end procedure
```

Algorithmus 6.4 Verteilung eines Alerts Rückgängig machen

```
procedure MacheAlertVerteilungRückgängig(Alert)
    Teilbäume ← generiereZuPlatzierendeTeilbäume(Alert.Bedingungsbaum)
    for all Teilbaum ∈ Teilbäume do
        Bedingungen ← extrahiereBedingungenAus(Teilbaum)
        for all Bedingung ∈ Bedingungen do
            MACs(Bedingung.Quelle).entferneRouteZuLAS(Bedingung)
        end for
        LASCs(Teilbaum.deployOn).entferneTeilbaum(Teilbaum)
    end for
    CASCs(Alert).konfiguriereFürZentralenAlert(Alert)
end procedure
```

7. Evaluation

Um zu zeigen, dass die Implementierung funktioniert und die Anzahl der an Grafana gesendeten Metriken durch die automatische Verteilung der Alerts reduziert werden kann, wurden drei unterschiedliche Verteilungen eines einfachen Alerts A5 getestet. Der Bedingungsbaum des Alerts A5 ist in Abbildung 7.1 zu sehen. Er besteht aus den zwei Bedingungen die über einen And-Operator miteinander verknüpft sind. Bedingung B1 setzt eine Schranke für die CPU-Metrik von Host 1 und B2 setzt die gleiche Schranke für die CPU-Metrik von Host 2. Beide Bedingungen besitzen ein Zeitfenster und treten ein, wenn die CPU-Auslastung der Hosts in der letzen Minute mindestens einmal höher als 90% war. Das Testsystem besteht aus den zwei VMs, Host1 und Host2, auf denen sich jeweils ein MA und ein LAS befinden. Das CAS befindet sich auf einer weiteren VM, auf dem sich auch der Mapper befindet. Abbildung 7.2 zeigt dieses System und die Erste der drei getesteten Verteilungen. Sie entspricht dem Ausgangszustand, bei der der Alert komplett im CAS ausgewertet wird. Die MAs senden ihre Metriken direket an dieses.

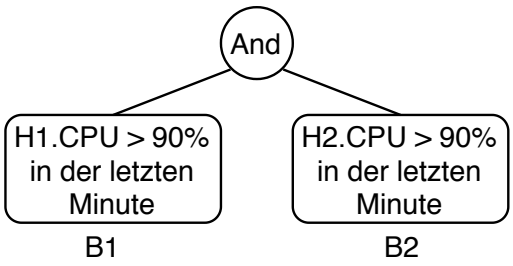

Abbildung 7.1.: Alert A5

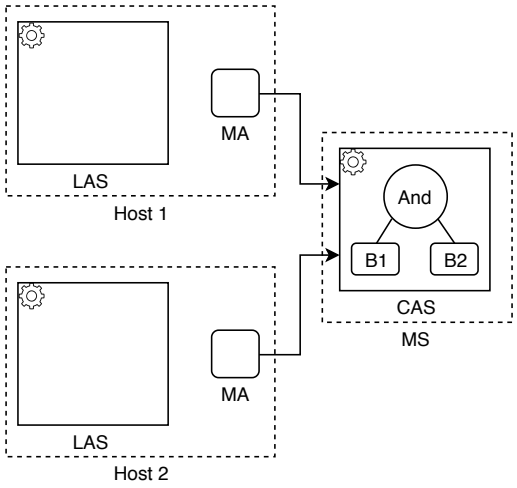

Abbildung 7.2.: Zentrale Auswertung

Abbildung 7.3 und 7.4 zeigen zwei Verteilungen, die durch den Mapper vorgenommen wurden. Hierbei senden die MAs ihre Metriken zunächst an ein LAS. Die linke Abbildung 7.3 zeigt die vom Mapper standardmäßig durchgeführte Verteilung. Hierbei werden die beiden Bedingungen lokal, durch das LAS des jeweiligen Hosts, ausgewertet, während die Verküpfung der entstehenden Alert-Metriken im CAS stattfindet. B1* und B2* stehen für die Bedingungen, die eintreten, sobald durch B1 und B2 erzeugte Alert-Mertiken das CAS erreichen. Die Verteilung in der rechten Abbildung 7.4 kann durch Anpassung der Alert-Policys erzeugt werden. Hier findet die Auswertung des gesamten Bedingungsbaumes außerhalb des CAS statt. Der MA von Host 1 sendet aggregierte Metriken an das CAS und Alert-Mertiken an das LAS auf Host 2. Die Bedingung A5* löst die Aktion des Alerts A5 aus, sobald die Alert-Mertiken für dessen Bedingungsbaum das CAS erreichen.

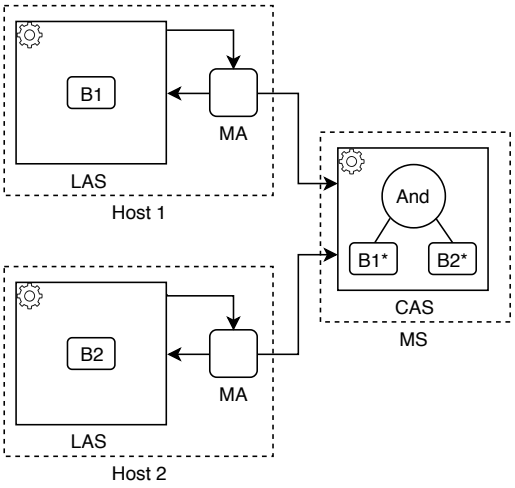

Abbildung 7.3.: Lokale Auswertung

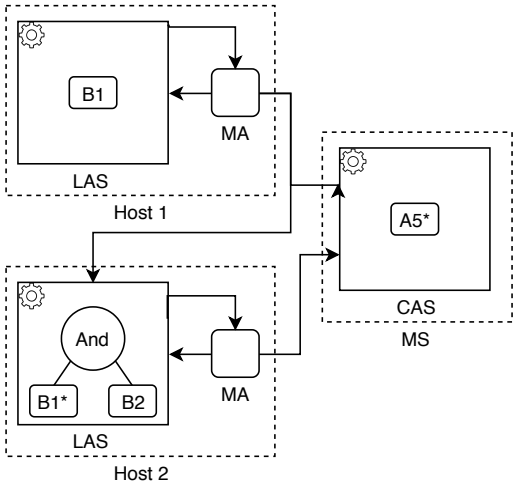

Abbildung 7.4.: Cluster Auswertung

Alle VMs nutzen Lubuntu 18.04 LTS als Betriebssystem und besitzen 1 GB Hauptspeicher. Um eine Aussage über den unterschiedlichen Netzwerktraffic der Verteilungen treffen zu können, wurde die Anzahl der Bytes gemessen, die von den MAs an andere VMs im Testzeitraum versendet wurden. Die Anzahl der versendeten Bytes wurden mit dem Kommandozeilentool *nethogs*[1] gemessen. Dieses ermöglicht die Überwachung des Netzwerk-Traffics einzelner Prozesse. CPU-Auslastung und der Speicherbedarf der MAs und LASs wurden mittels *ps* ermittelt. Die Messungen wurden über einen Zeitraum von 5 Minuten durchgeführt und fanden lokal auf den VMs statt. Sie wurden 3-mal wiederholt und anschließend wurde der Durchschnitt aus allen Ergebnissen berechnet. Um die Werte der CPU-Metriken, und damit den Zustand des Alerts, bestimmen zu können, wurde ein Tool verwendet, mit dem sich Metriken mit vorgegeben Werten an ein HTTP-Listener-Input eines Telegraf-Agenten senden lassen. Metriken werden im Abstand von 10 Sekunden erzeugt und von Telegraf alle 10 Sekunden an seine Outputs weitergeleitet. Für die Aggregation der Metriken wurde ein Intervall von 30 Sekunden in den Aggregator-Plugins der Telegraf-Agenten konfiguriert. Dadurch werden über den Zeitraum von 5 Minuten pro MA 30 Metriken und 10 Aggregate erzeugt. Es wurden zwei unterschiedliche Konfigurationen des Telegraf-Agenten getestet. In der ersten Konfiguration werden alle Metriken einzeln von den Outputs versandt, während in der zweiten Variante eine Batchverarbeitung der Metriken stattgefunden hat. Dadurch werden alle Metriken, Aggregate oder Alert-Metriken, die den Agenten innerhalb von 10 Sekunden erreichen und zum selben Output geroutet werden, in einer einzigen HTTP-Nachricht verschickt. Anhand der ersten Variante kann getestet werden, ob alle erwarteten Nachrichten versandt wurden. Dafür kann die gemessene Datenmenge durch die von den Outputs für das Senden einer Metrik benötigte geteilt werden. Aufgrund unterschiedlicher HTTP-Header benötigt der HTTP-Output im Durchschnitt 400 Bytes um eine Metrik zu versenden, während der InfluxDB-Output ca. 500 Bytes benötigt. Da das verwendete Format für Metriken nur wenige Bytes benötigt, ist davon auszugehen, dass ein Großteil der benötigten Bandbreite auf die eingesetzten Protokolle zurückzuführen ist.

[1] https://github.com/raboof/nethogs

Die Anzahl der durch die MAs gesendeten Kilobytes (KB) wenn kein Alert eintritt, ist in Abbildung 7.5 zu sehen. Hier unterscheiden sich die Werte zwischen Einzel- und Batchverarbeitung nicht. Bei der zentralen Auswertung des Alerts haben die MAs jeweils 14,8 KB gesendet, während sie bei den verteilten Auswertungen nur 5 KB gesendet haben. Dass die Anzahl der gesendeten KBs, bei der verteilten Auswertung, gegenüber der zentralen Auswertung nur ein Drittel beträgt, hängt damit zusammen, dass die ursprüngliche Metrik im Abstand von 10 Sekunden versendet wird und ein aggregierter Wert nur alle 30 Sekunden versendet wird. Da hier Metriken und Aggregate über den InfluxDB-Output der MAs versendet wurden, ergibt sich aus der genutzen Bandbreite, dass 30 Metriken bzw. 10 Aggregate versandt wurden. Es ist gut erkennbar, dass sich der Netzwerk-Traffic, durch eine verteilte Auswertung des Alerts, im Normalzustand reduzieren lässt. Durch eine Vergrößerung des Intervalls der Aggregation, kann die Menge der gesendeten KBs erheblich reduziert werden.

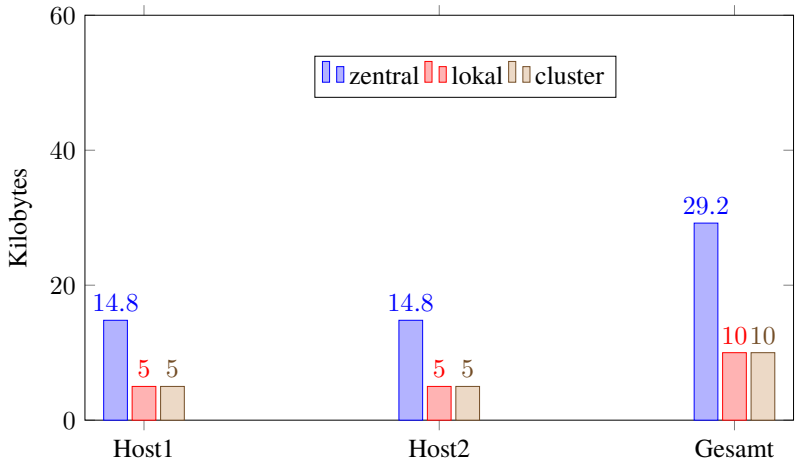

Abbildung 7.5.: Gesendete KBs im Normalzustand

Abbildung 7.6 zeigt die Anzahl der, durch die MAs, gesendeten KBs im Falle eines Alerts, bei Einzelverarbeitung der Metriken. Bei zentraler Auswertung ändert sich die Menge der KBs gegenüber dem Normalzustand nicht, da auf den Hosts keine Aggregate oder Alert-Metriken erzeugt werden. Beide MAs haben 14.8 KB versandt. Im Gegensatz zum Normalzustand liegt die Menge der gesendeten KBs bei beiden verteilten Auswertungen über der einer zentralen Auswertung. Dies liegt daran, dass die MAs neben den aggregierten Metriken nun auch die Alert-Metriken weiterleiten. Auf Grund der Konfiguration des LAS und der Alert-Bedingungen, wird für jede eingehende Metrik, eine Alert-Metrik erzeugt. Bei der lokalen Auswertung haben beide MAs 20.6 KB versendet. Davon entfallen ca. 5 KB auf die Aggregate, während die 30 Alert-Metriken ca. 15.5 KB benötigt haben. Für die Auswertung im Cluster ergibt sich ein anderes Bild. Während Host 1 nur 17.5 KB versandt hat, wurden von Host 2 36.6 KB versandt. Dass die von Host 1 benötigte Bandbreite beim Clustering geringer ausfällt, liegt daran, dass die Alert-Metriken über einen HTTP-Output an das LAS von Host 2 gesendet wurden. Dieser benötigt nur 400 KB pro Nachricht. Die hohe Datenmenge von Host 2 hängt damit zusammen, dass Host 2 für die Auswertung des gesamten Alerts zuständig ist und die Bedingungen des Alerts ein Zeitfenster besitzen. Dadurch führen einzelne Alert-Metriken von Host 1 dazu, dass eine Alert-Metrik auf Host 2 generiert wird, wenn Bedingung B2 in der letzten Minute erfüllt wurde. Über den Zeitraum von 5 Minuten werden so 60 Alert-Metriken von Host 2 generiert

und über den InfluxDB-Output versandt. Clustering verursachte den meisten Netzwerk-Traffic, da hier auch die meisten Nachrichten versandt wurden. Die Anzahl der versendeten Nachrichten ist in Tabelle 7.1 zu sehen. Sie stimmt mit den Erwartungen überein.

Auswertung	Metriken	Alert-Metriken	Aggregate
zentral	60	0	0
lokal	0	60	20
cluster	0	90	20

Tabelle 7.1.: Im Falle eines Alerts versandte Nachrichten

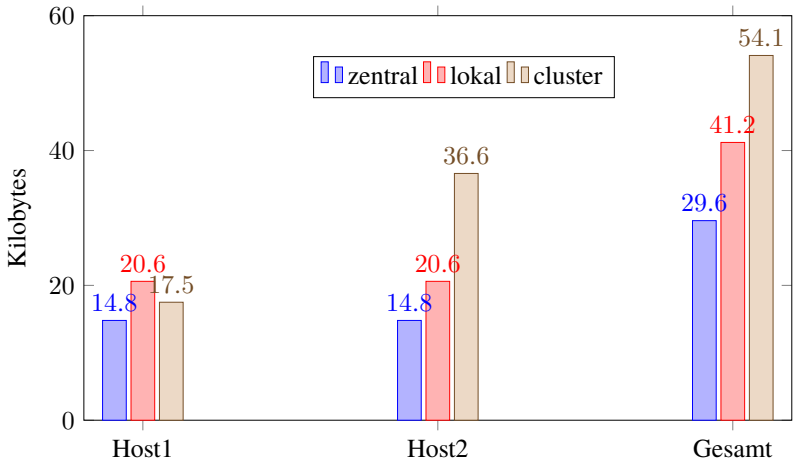

Abbildung 7.6.: Gesendete KBs im Falle eines Alerts mit Einzelverarbeitung

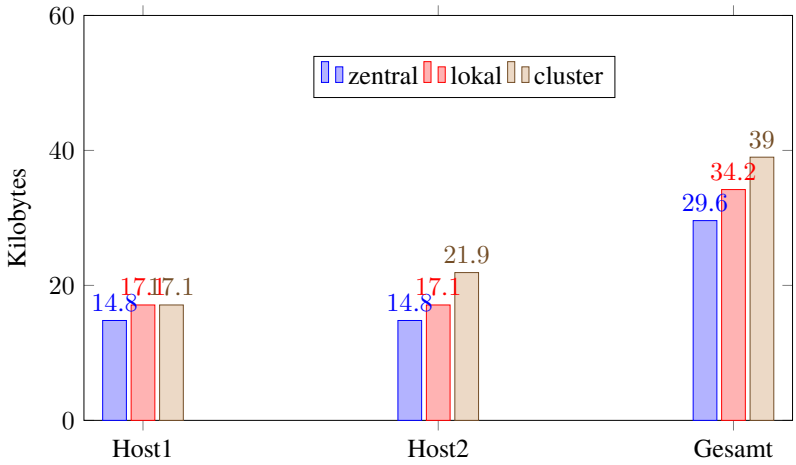

Abbildung 7.7.: Gesendete KBs im Falle eines Alerts mit Batchverarbeitung

Durch die Batchverarbeitung lässt sich die Menge der benötigten KBs, wie in Abbildung 7.7 zu sehen, deutlich reduzieren. Da bei der lokalen Auswertung Alert-Metriken und Aggregate über den selben InfluxDB-Output geroutet werden, sinkt die Anzahl der KBs von 20.6 auf 17.1. Beim Clustering konnte die von Host 2 versandte Datenmenge, von 36.6 KB auf 21.9 KB, erheblich reduziert werden. Dies ist vor allem darauf zurückzuführen, dass in einem 10 Sekunden Intervall

mehrere Alert-Metriken erzeugt werden, die nun in einer gemeinsamen HTTP-Nachricht versandt werden. Insgesamt haben sich die Werte der beiden verteilten Auswertung denen einer zentralen Auswertung stark angenähert.

Während dieser Tests haben die LAS auf den Hosts nie mehr als 0.1% CPU-Auslastung erzeugt und im Schnitt 100KB Speicher benötigt.

Die Tests haben gezeigt, dass sich der Netzwerk-Traffic, durch eine vom Mapper durchgeführte Verteilung eines Alerts, im Normalzustand erheblich reduzieren lässt. Durch die Wahl eines größeren Intervalls für die Aggregation kann die Menge der gesendeten Daten noch weiter reduziert werden. Im Falle eines Alerts war der Netzwerk-Traffic der verteilten Auswertungen höher als der der Zentralen. Mit Hilfe der Batchverarbeitung konnte der Netzwerk-Traffic bei einer verteilten Auswertung reduziert werden. Sie ist der Einzelverarbeitung immer vorzuziehen, da diese unnötigen Overhead erzeugt. Da das Eintreten eines Alerts nicht den Dauerzustand des Systems darstellt, können die großen Einsparungen im Normalzustand den Netzwerk-Traffic des Systems reduzieren. Selbst wenn sich ein System nur die Hälfte der Zeit im Normalzustand befindet, kann eine verteilte Auswertung den Netzwerk-Traffic reduzieren. Zwar wurden im Falle eines Alerts für die Auswertung im Cluster, die meisten KBs versandt, jedoch bedeutet dies nicht, dass diese Verteilung nutzlos ist. Betrachtet man nur die Menge der Daten, die an das CAS gesendet wurden, so schneidet die Auswertung im Cluster am besten ab. Host 1 sendet nur seine aggregierten Metriken in Höhe von ca. 5 KB an das CAS. Da Host 2 nur an das CAS sendet, ergeben sich 26,9 KB die an das CAS gesendet werden. Damit eignet sich eine Auswertung im Cluster vor allem dann, wenn die Bandbreite zum CAS beschränkt ist.

8. Zusammenfassung und Ausblick

In der Arbeit wurde ein Konzept für lokales Alerting in Monitoringsystemen präsentiert, welches es ermöglicht, zentral definierte Alerts auf lokalen AS zu platzieren. Mittelpunkt dieses Konzepts, stellt die Mapper-Komponente dar. Sie führt die automatisierte Verteilung der Alerts durch, wofür sie die CASs, LASs und MAs eines Systems konfigurieren muss. Für die Änderungen der Konfigurationen nutzt sie ihre Konfigurationsmodule. Durch die modulare Gestaltung der Schnittstellen zu den einzelnen Komponenten kann der Mapper an verschiedene Monitoringsysteme angepasst werden. Es wurde eine Implementierung des Konzepts erstellt, welche Grafana als CAS, die Esper-Komponente als LAS und Telegraf als MA nutzt. Für die Mapper-Komponente wurden die entsprechenden Konfigurationsmodule entwickelt. Aufgrund deren Modularität lässt sie sich aber auch mit anderen Komponenten nutzen. Erste Testergebnisse zeigen, dass sich durch Einsatz des Mappers eine automatische Verteilung der Alerts erzielen lässt, die, wie erwartet, eine Reduktion der Netzwerkauslastung hervorrufen kann. Dabei ist die Höhe der Reduktion stark von den Bedingungen des Alerts abhängig.

Ausblick

Bisher versucht der Mapper die Menge der versandten Metriken zu reduzieren, indem möglichst große Teilbäume eines Alerts lokal platziert werden. Dabei nutzt er weder die Topologie des Systems noch dessen Ressourcenauslastung. Mit Hilfe dieser Informationen könnte die Verteilung besser an das System angepasst werden. Die Bestimmung der Verteilung kann dabei extern stattfinden, da der Mapper mit den Alert-Policys die nötige Schnittstelle zur Verfügung stellt. Um dem Problem entgegenzuwirken, dass der Mapper, bei einem Absturz, alle Informationen über die Verteilungen der Alerts verliert, muss es ermöglicht werden, diese persistent zu speichern. Durch die Erweiterung des Mappers um eine Schnittstelle für Datenbanken könnte dies umgesetzt werden.

Literaturverzeichnis

[ABDP13] G. Aceto, A. Botta, W. de Donato, A. Pescapè. “Cloud monitoring: A survey”. In: *Computer Networks* 57.9 (2013), S. 2093–2115. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2013.04.001. URL: http://www.sciencedirect.com/science/article/pii/S1389128613001084 (zitiert auf S. 27).

[CM10] G. Cugola, A. Margara. “TESLA: A Formally Defined Event Specification Language”. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS ’10. Cambridge, United Kingdom: ACM, 2010, S. 50–61. ISBN: 978-1-60558-927-5. DOI: 10.1145/1827418.1827427. URL: http://doi.acm.org/10.1145/1827418.1827427 (zitiert auf S. 28).

[CM12] G. Cugola, A. Margara. “Complex Event Processing with T-REX”. In: *J. Syst. Softw.* 85.8 (Aug. 2012), S. 1709–1728. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.03.056. URL: http://dx.doi.org/10.1016/j.jss.2012.03.056 (zitiert auf S. 28).

[CM13] G. Cugola, A. Margara. “Deployment strategies for distributed complex event processing”. In: *Computing* 95.2 (Feb. 2013), S. 129–156. ISSN: 1436-5057. DOI: 10.1007/s00607-012-0217-9. URL: https://doi.org/10.1007/s00607-012-0217-9 (zitiert auf S. 28).

[col] collectd. *collectd - The system statistics collection daemon*. https://collectd.org/ (zitiert auf S. 27).

[Esp] EsperTech. *Esper CEP platform*. http://www.espertech.com/esper/ (zitiert auf S. 41).

[FEH+14] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, T. Lynn. “A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives”. In: *Journal of Parallel and Distributed Computing* 74.10 (2014), S. 2918–2933. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.06.007. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001099 (zitiert auf S. 27).

[Gra] Grafana. *Grafana - Analytics Platform*. https://grafana.com/ (zitiert auf S. 41).

[HW18] C. B. Hauser, S. Wesner. “Reviewing Cloud Monitoring: Towards Cloud Resource Profiling”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. Bd. 00. Juli 2018, S. 678–685. DOI: 10.1109/CLOUD.2018.00093. URL: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00093 (zitiert auf S. 27).

[Ici] Icinga. *Icinga*. https://icinga.com/ (zitiert auf S. 28).

[Infa] InfluxData. *InfluxDB - Time Teries Database*. https://www.influxdata.com/time-series-platform/influxdb/ (zitiert auf S. 41).

[Infb] InfluxData. *Open Source Time Series Platform | InfluxData*. https://www.influxdata.com/time-series-platform/ (zitiert auf S. 27, 41).

[Infc] InfluxData. *Telegraf - Agent*. https://www.influxdata.com/time-series-platform/telegraf/ (zitiert auf S. 41).

[MCC04] M. L. Massie, B. N. Chun, D. E. Culler. "The ganglia distributed monitoring system: design, implementation, and experience". In: *Parallel Computing* 30.7 (2004), S. 817–840. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2004.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0167819104000535 (zitiert auf S. 27).

[MG11] P. M. Mell, T. Grance. *The NIST Definition of Cloud Computing*. Techn. Ber. Gaithersburg, MD, United States: NIST, 2011 (zitiert auf S. 17).

[Mod] Mod-Gearman. *Mod-Gearman - distributed monitoring for Naemon*. https://mod-gearman.org/ (zitiert auf S. 28).

[Mooa] Moogsoft. *Moogsoft AIOps | Purpose-Built AIOps Platform for IT*. https://www.moogsoft.com/ (zitiert auf S. 27).

[Moob] Moogsoft. *Moogsoft Observe*. https://guides.moogsoft.com/display/AIOR/Observe+Overview (zitiert auf S. 27).

[MSM+13] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, G. Antoniu. "GMonE: A complete approach to cloud monitoring". In: *Future Generation Computer Systems* 29.8 (2013). Including Special sections: Advanced Cloud Monitoring Systems The fourth IEEE International Conference on e-Science 2011 — e-Science Applications and Tools Cluster, Grid, and Cloud Computing, S. 2026–2040. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.02.011. URL: http://www.sciencedirect.com/science/article/pii/S0167739X13000496 (zitiert auf S. 27).

[Nae] Naemon. *Naemon - Monitoring Suite*. http://www.naemon.org/ (zitiert auf S. 28).

[Nag] Nagios. *Nagios - The Industry Standard In IT Infrastructure Monitoring*. https://www.nagios.org/ (zitiert auf S. 27).

[PF4] PF4J. *Plugin Framework for Java*. https://pf4j.org/ (zitiert auf S. 48).

[PLL+13] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, L. Foschini. "DARGOS: A Highly Adaptable and Scalable Monitoring Architecture for Multi-tenant Clouds". In: *Future Gener. Comput. Syst.* 29.8 (Okt. 2013), S. 2041–2056. ISSN: 0167-739X. DOI: 10.1016/j.future.2013.04.022. URL: http://dx.doi.org/10.1016/j.future.2013.04.022 (zitiert auf S. 21).

[Pre] T. Preston-Werner. *Tom's Obvious, Minimal Language*. https://github.com/toml-lang/toml (zitiert auf S. 42).

[SGA+17] H. J. Syed, A. Gani, R. W. Ahmad, M. K. Khan, A. I. A. Ahmed. "Cloud monitoring: A review, taxonomy, and open research issues". In: *Journal of Network and Computer Applications* 98 (2017), S. 11–26. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.08.021. URL: http://www.sciencedirect.com/science/article/pii/S1084804517302783 (zitiert auf S. 19, 27).

[SMP09] N. P. Schultz-Møller, M. Migliavacca, P. Pietzuch. "Distributed Complex Event Processing with Query Rewriting". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. Nashville, Tennessee: ACM, 2009, 4:1–4:12. ISBN: 978-1-60558-665-6. DOI: 10.1145/1619258.1619264. URL: http://doi.acm.org/10.1145/1619258.1619264 (zitiert auf S. 28).

[sta] statsd. *statsd: Daemon for easy but powerful stats aggregation*. https://github.com/statsd/statsd (zitiert auf S. 27).

[WB14] J. S. Ward, A. Barker. "Observing the clouds: a survey and taxonomy of cloud monitoring". In: *Journal of Cloud Computing* 3.1 (Dez. 2014), S. 24. ISSN: 2192-113X. DOI: 10.1186/s13677-014-0024-2. URL: https://doi.org/10.1186/s13677-014-0024-2 (zitiert auf S. 27).

[WLY+10] L. Wang, G. von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, C. Fu. "Cloud Computing: a Perspective Study". In: *New Generation Computing* 28.2 (Apr. 2010), S. 137–146 (zitiert auf S. 19).

Alle URLs wurden zuletzt am 20. 05. 2019 geprüft.

A. Apendix

```
[ {
  "id" : "Alert1",
  "version" : 1,
  "policies" : { },
  "alertTree" : {
    "id" : "Alert1Bedingung1OR2",
    "policies" : {
      "placeOn" : ""
    },
    "children" : [ {
      "id" : "Alert1Bedingung1",
      "policies" : {
        "placeOn" : "192.168.192.5"
        "aggregationInterval": "30s",
        "aggregationMethod": "mean"
      },
      "children" : null
    }, {
      "id" : "Alert1Bedingung2",
      "policies" : {
        "placeOn" : "192.168.192.55"
        "aggregationInterval": "30s",
        "aggregationMethod": "mean"
      },
      "children" : null
    } ]
  }
} ]
```

Listing A.1: Beispiel einer Alert-Policys Datei

```
{
  "meta": {...},
  "dashboard": {
    "panels": [
      {
        "alert": {
          "conditions": [
            {
              "evaluator": {
                "params": [
                  100000000
                ],
                "type": "lt"
              },
              "operator": {
                "type": "and"
              },
              "query": {
                "params": [
                  "UsedMemory",
                  "1m",
                  "now"
                ]
              },
              "reducer": {
                "type": "max"
              },
              "type": "query"
            }
          ],
        },
        "datasource": "InfluxDB",
        "id": 4,
        "targets": [
          {
            "groupBy": [],
            "measurement": "mem",
            "refId": "UsedMemory",
            "select": [
              [
                {
                  "params": [
                    "free"
                  ],
                  "type": "field"
                }
              ]
            ],
            "tags": []
          }
        ],
      }
    ],
    "uid": "9_v7Z89iz",
  }
}
```

Listing A.2: Vereinfachte JSON-Repräsentation eines Dashboards

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift