HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

BLUE DIAMOND
asset management

Cyber Security Integration Engine

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2017

Author(s):	Janic Mikes, Marcel Maeder
Advisor:	Andreas Steffen
Project Partner:	Blue Diamond Asset Management AG, Pfäffikon SZ

Abstract

Problem Every company that is working with confidential data needs to fulfill certain regulations. Especially in the financial sector these requirements are crucial to a company's success. For small businesses this can become a severe problem. To help those small companies to fulfill these requirements we want to provide a prototype of an extensible software that will take care of such tasks.

Goal The goal is to get a zero-configuration security box that is easily extensible and able to scale. It should collect necessary data and be able to determine a baseline on what is normal behavior on a client's network. To get started we want to be able to keep a list of machines communicating on the network.

Method We will split the topic into three separate concerns. For the communication between the subsystems we will use a messaging system. Every information exchange goes through this system using messages. The probes are responsible for collecting potentially interesting information about its environment. This could be network metadata or other kinds of information. The probe then publishes the gathered information into the messaging system. Agents subscribe to messages they are capable of analyzing. Based on these messages they investigate and publish their acquired information back to the messaging system, so any other agent can reuse this information.

Results In our lab environment we were able to collect and pass network activity data between multiple participants within our framework. We implemented a network probe and an agent that could detect an unknown device based on its mac address. We implemented this using two different messaging systems and compared these two in terms of implementation complexity and message throughput performance.
We found that one messaging system is over 10 times faster than the other and had much better tooling.

Inhaltsverzeichnis

1 Einleitung

1.1 Problem

Es gibt zahlreiche Produkte auf dem Markt, welche einem Systemoperator oder Sicherheitsverantwortlichen in einer Firma unterstützen können, indem sie sicherheitsrelevante Vorkommnisse aufzuzeigen. Möchte man sämtliche Bereiche abdecken, so werden oft mehrere Tools benötigt, welche sich nur schwer zusammenfassen lassen. Dazu kommt, dass eine Person alleine nur schwer sämtliche Tools bedienen und den Überblick über die unterschiedlichen Dashboards behalten kann.
In dieser Arbeit soll als Machbarkeitsstudie ein Framework entwickelt werden, dass die Integration von Daten aus verschiedenen Quellen ermöglicht.

1.2 Aufgabenstellung des Kunden

Even for small companies, the requirements on cybersecurity are high and there is a huge challenge in good management. Big companies have the resources to have whole departments only dealing with cybersecurity and deploying big systems. For smaller companies it's harder, the requirements and regulations are the same but the resources are not there. There are a lot of different tools and systems to help out, and normally a set of tools are deployed to fulfill requirements. A typical small company needs,

- Antivirus
- Firewalls on client computers, servers, and network firewalls
- Intrusion detection systems
- Asset inventory system
- Backup systems
- Log servers

All those systems need to be correctly configured and monitored continuously. Given that the systems are isolated islands it's not always easy since each system has its own monitoring systems and they are normally also quite basic.
The idea is to build a central system which collects data from all different systems. On top of the collected data, there are agents analyzing the data. Those agents can apply a different kind of logic like machine learning in analyzing the data. Also, the agents can combine

different kinds of datasets. In this way, the monitoring can be better automated, be smarter and generate less noise.
The system should have a standardized REST-based interface so it's easy to integrate different front-end applications to it. Example of functionality, it should generate an event if:

- Someone plugs in a new computer on the network
- There is a lot of network scanning activity
- Someone logs in with VPN from an unknown AS
- Server X logs twice as much as usual
- Client A runs an unknown application
- Client B has no antivirus running.
- A new port is open on server Y
- Network traffic on port Z in Switch Y is a lot higher than normal.
- Response time on web server is higher than usual
- There are unusual commands running on server Y

There are a lot of different monitoring systems out there, for example, http://riemann.io/ has a power stream processing language. Nagios, Zabbix and so on our other popular tools. However, the analyses are often extremely basic and there is a focus on presentation. We want to build a generic open system for collecting and analyzing data related to cybersecurity. The system should be distributed and scalable where it's easy to plugin new components. The system should be built as a reactive system. (http://www.reactivemanifesto.org/)

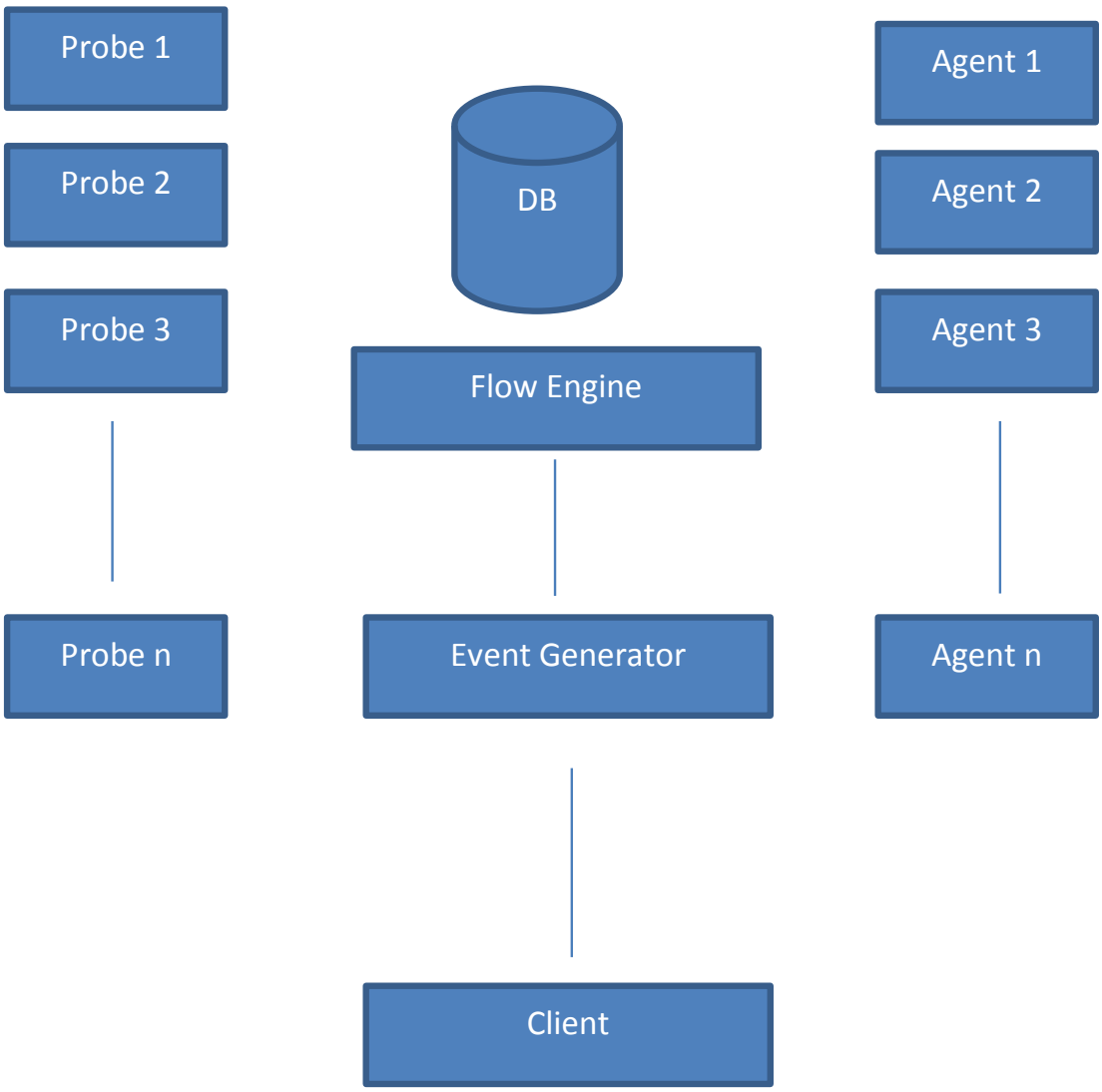

Abbildung 1.1: Example Architecture [1]

1.3 Abgrenzung

Diese Arbeit ist eine Machbarkeitsstudie. Es geht nicht darum, Software nach Best-Practices zu programmieren.

2 Use Cases

Im Auftrag des Kunden erarbeiteten wir Funktionalitätsanforderungen. Diese sollen den Umfang der Arbeit grob abstecken. Wir verwenden dafür User Storys.
Das Ziel ist der Aufbau eines Proof of Concepts. Wir fokussieren uns auf die Erkennung von unbekannten Geräten im Netzwerk.
Als Benutzer bezeichnen wir den Systemadministrator, der diese Lösung zum Monitoring seiner Umgebung verwendet.

2.1 User Storys

Geräteaktivität Als Benutzer möchte ich wissen, welche Geräte in den letzten X Minuten aktiv waren.

Alarmhistorie Als Benutzern möchte ich sehen, welche Alarme das System generiert hat.

Historische Daten Als Benutzer möchte ich die historischen Daten selbst analysieren.

Informationsbasis Als Benutzer möchte ich wissen, welche Geräte das System kennt.

Alarme Empfangen Als Benutzer möchte ich Alarme empfangen.

2.2 Demonstrator Storys

Demo / Testing Als Vorführer möchte ich Netzwerkverkehr simulieren, so dass ich das Verhalten des Systems vorführen kann.

3 Analyse

Die Use Cases verlangen vom System, dass es unbekannte Geräte am Netzwerk erkennt, meldet, und die historischen Daten darüber speichert. Wie in der Aufgabenstellung beschrieben wollen wir diese Funktionen auf verschiedene Teilsysteme aufteilen. Das Zielsystem besteht daher aus Agents, Probes und einem Message- und Persistenzlayer. Der User interagiert über den Message- und Persistenzlayer mit dem System.

3.1 Framework

Unser Framework sieht wie folgt aus.

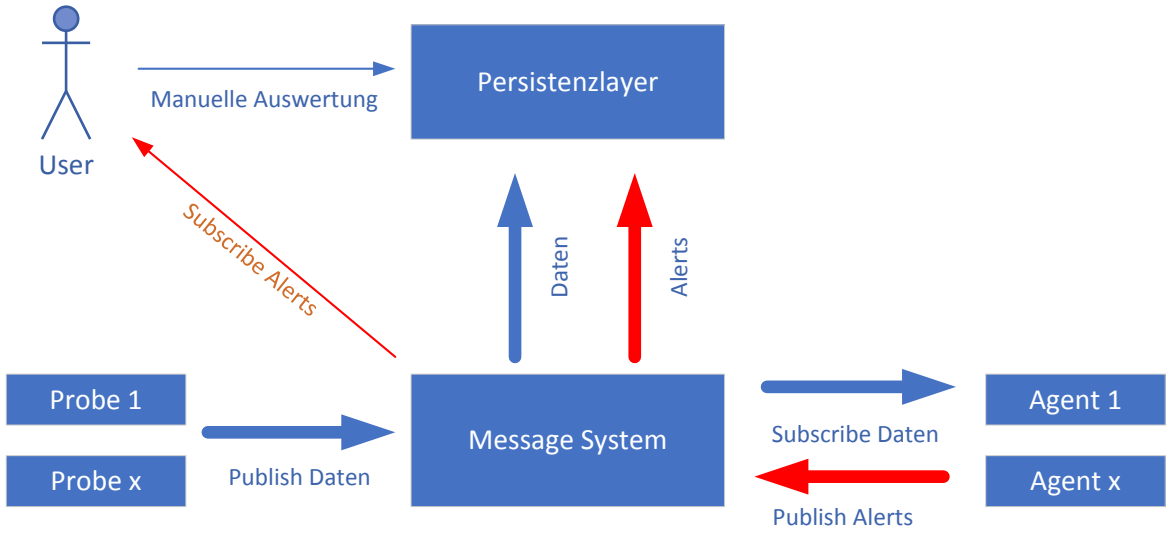

Abbildung 3.1: Architekturdiagramm

3.1.1 Rollen

User Der User in unserem System wäre der System- oder Sicherheitsverantwortliche innerhalb einer Firma. Er möchte möglichst ohne Verzögerung Alerts angezeigt bekommen. Darauf kann er geeignete Massnahmen ergreifen.

Messaging System Das Messaging System ist die Aorta des Frameworks. Es ist für die Vermittlung der Daten zwischen den Agents, Probes und des Persistenzlayers verantwortlich.

Probe Eine Probe sammelt Daten und stellt diese dem Messaging System zur Verfügung.

Agent Ein Agent analysiert Daten vom Messaging System. Stellt er etwas fest, sendet er Alerts an das Messaging System. Denkbar ist auch, dass ein Agent gewisse Aktionen auslösen kann, zum Beispiel beim Router einen gewissen IP Range blockieren.

Persistenzlayer Der Persistenzlayer sammelt alle Daten, die über das Messaging System gesendet wurden. Er ermöglicht es dem User, die historischen Daten zu analysieren.

3.2 Implementationen

3.2.1 Messaging System

Es gibt viele bestehende Messaging Systeme. Wir werden in den folgenden Kapiteln ein geeignetes evaluieren.

Queues

Zwischen folgenden Services braucht es Queues:

Queue-Name	Publisher	Subscriber
Netflow	Netflow-Probe, Mock-Probe	Netflow-Agent, Persistenzlayer
Alerts	Netflow-Agent	Persistenzlayer

Tabelle 3.1: Notwendige Message Queues

Die Namen werden entsprechend den Namenskonventionen des verwendeten Produktes vergeben.

3.2.2 Probes

Unbekannte Geräte sollen erkannt werden. Es gibt herstellerspezifische NAC-Lösungen (Network Admission Control) und offene Standards wie 802.1x. Unsere Idee ist es, Netflow dazu zu verwenden, da es auch MAC-Adressinformationen beinhalten kann. Dies ist jedoch abhängig von der Switch Plattform. Gewisse Typen können nicht für jeden einzelnen Flow einen Eintrag machen, sondern sie nehmen ein Sampling vor.

Wir brauchen daher einen Netflow-Adapter, der dieses Protokoll versteht und die Daten daraus dem Messaging System schicken kann.
Für den Demonstrator-Case werden wir einen Netflow-Adapter mocken welcher vordefinierte Netflow Daten in das System einspielt.

3.2.3 Agents

Unser Agent muss aus den Netflow-Daten, die er über das Messaging System erhält, Alerts generieren können. Er führt eine Liste ihm bekannter MAC-Adressen.

3.2.4 Persistenzlayer

Der Persistenzlayer muss strukturierte Daten mit unbekanntem Schema abspeichern und durchsuchbar machen. Auch hier werden wir ein Produkt suchen.

4 Evaluation

4.1 Messaging Lösung

Um die Daten möglichst performant zwischen verschiedenen Systemen auszutauschen wird eine Streaming Plattform benötigt. Diese ermöglicht es grundsätzlich, dass man Nachrichten veröffentlichen (publish) oder gewisse Queues abonnieren (subscribe) kann. Das Verteilen der Nachricht vom Publisher zum Subscriber übernimmt die Messaging Lösung.

4.1.1 Apache Kafka

Apache Kafka [2] ist darauf ausgelegt, grosse Datenmengen zuverlässig und in real-time über sogenannte Pipelines zu streamen. Es ermöglicht die Realisierung von real-time Applikationen, welche Datenstreams generieren oder auf solche reagieren.
Ein Vorteil von Apache Kafka ist, dass die Übermittlung der Nachrichten fehlertolerant geschieht. Dies ist vor allem bei einer sehr grossen Auslastung von Vorteil, da so eine sichere Übertagung sichergestellt werden kann.
Queues werden in Kafka als Topics bezeichnet. Sie haben keine Hierarchie. Der Subscriber kann mittels Regular Expressions mehrere Topics abonnieren, ohne die einzelnen Topics zu kennen.
Topics werden bei Kafka automatisch bei der ersten Schreib- oder Leseoperation erstellt.

4.1.2 XMPP mit Openfire

XMPP [3], auch bekannt unter dem Namen Jabber, ist eine weitverbreitete Messaging Lösung, um Nachrichten an einen oder mehrere Empfänger zu pushen. Es ist vor allem bei Instant Messaging Lösungen beliebt. Die Push basierte Nachrichtenübertragung erlaubt eine sehr schnelle und effiziente Übermittlung von Daten der Probes an die Agents.
Als Server verwenden wir Openfire [4].
Am besten wird XMPP im Publish and Subscribe (PubSub) Modus verwendet. Dies ermöglicht die Gruppierung von Nachrichten in einer Nodehierarchie. Eine Subscription auf einem Node beinhaltet auch alle Messages seiner Child-Nodes. Auf die Leaf-Nodes können die Probes dann ihre Messages publishen. Die Agents werden auf Nodes registriert und erhalten die Nachrichten automatisch vom Server sobald eine neue eingetroffen ist.
Das Protokoll ist XML basierend und ist standardisiert. Publish Subscribe ist in XEP-60 festgehalten.

4.1.3 Vergleichskriterien

Da wir mit beiden Plattformen noch nicht gearbeitet haben, versuchen wir, den Proof of Concept mit beiden Systemen zu implementieren. Aus den daraus gewonnenen Entwicklungserfahrungen kann in einem allfälligen Folgeprojekt direkt das geeignete System gewählt werden.

API Die Client Library ist gut dokumentiert. Standardaufrufe wie Publish und Subscribe sind einfach aufzurufen.

Security Die Plattform ermöglicht den Einsatz von Verschlüsselung. Eine Authentisierung ist möglich. Dies ist nicht relevant für den Proof of Concept, aber für einen produktiven Einsatz.

Overhead Kostet die Verwendung des Systems mehr Datenvolumen, als eine reine TCP-Socket Übertragung? Wie komplex ist das System?

Benchmark Wir entwickeln einen eigenen Benchmark, um die beiden Systeme unter verschiedenen Lastszenarien vergleichen zu können.

4.2 Basisimplementation Kafka

Voraussetzungen Die minimale Kafka-Lösung benötigt folgende Komponenten:

- Kafka Node
- Zookeeper (Kafka Node manager)
- Kafka Client Libraries (kafka-python)

API KafkaConsumer und KafkaProducer können bei kafka-python direkt instanziiert werden. Über die Messages kann über einen blockierenden Iterator iteriert werden. Mittels der *.send()* Funktion werden die Messages an die Queue geschickt.

Security Kafka ermöglicht die Verschlüsselung und Authentisierung zwischen allen Komponenten [5].

Overhead Kafka transportiert Byte-Arrays. Das Format sowie das Encoding der Messages liegt in der Verantwortung der Clients.

4.3 Basisimplementation XMPP

Voraussetzungen

- XMPP Server (Openfire)
- XMPP Client Libraries (Sleekxmpp für Python)

API

Die Autoren verwenden Sleekxmpp für die Implementierung der Probes und Agents. Sleekxmpp abstrahiert zwar den XMPP Teil, aber es ist immer noch viel von den XEP-60 XML Strukturen sichtbar. So muss beispielsweise für die Erstellung eines Nodes ein XML Form zusammengestellt werden.
Die Library verwendet verschiedene Background-Threads. Die *.send()* Funktion ist gemäss Dokumentation nicht blockierend. Nachrichten werden über eine Callback Funktion abgearbeitet.

Security

XEP-60 unterscheidet zwischen den Berechtigungen für Publish und Subscribe. Diese können beim Erstellen gesetzt oder zu einem späteren Zeitpunkt geändert werden. Openfire kann die Client-Server Kommunikation über TLS verschlüsseln.

Overhead

Jegliche Kommunikation mit dem Server wird in XML gepackt.

4.4 Persistenzlayer

Es gibt verschiedene Lösungen für die Persistenz, von der Eigenentwicklung bis zu einer kommerziellen Lösung wie Splunk. Wir sehen aus Zeitgründen davon ab, eine eigene Lösung zu entwickeln. Da wir ein möglichst offenes System haben möchten, werden wir diesen Layer mit dem Elastic Stack implementieren.

4.5 Netflow Collector

Als Netflow Collector verwenden wir das Opensource Projekt Pmacct [6]. Dieses Projekt kann Netflow Daten empfangen, und als JSON Objekt direkt an Kafka weitersenden. Für XMPP haben wir keine fertige Lösung gefunden.

5 Development

Für dieses Projekt haben wir verschiedene Applikationen selbst entwickelt so wie bestehende integriert. Dieses Kapitel beschreibt die von uns entwickelten Applikationen.
Wir verwenden für alle unsere Applikationen Docker Container. Die Applikationen, dazugehörige Hilfsprogramme sowie Szenarios befinden sich im Git Repository SAIDocker [7].

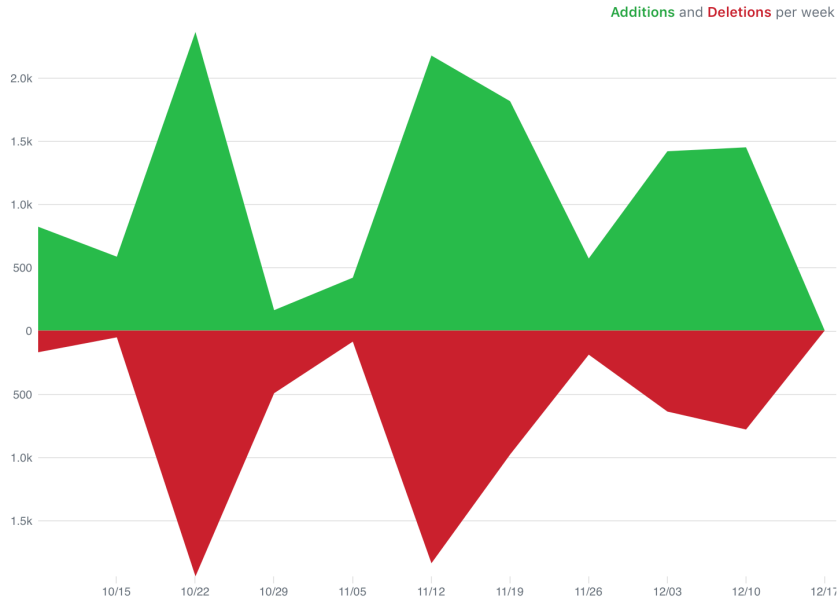

Abbildung 5.1: Unsere Aktivität auf GitHub (SAIDocker Repository)

5.1 Applications

5.1.1 Probes

LogReplay und LogRerunner

Die Applikationen LogReplay und LogRerunner ermöglichen es, ein File mit mehreren Zeilen JSON in einem definierten Zeitintervall an das Messaging System zu übertragen. Sie können verwendet werden, um Events von Probes zu simulieren.
In unseren Szenarien werden diese Applikationen verwendet, um die Netflow Daten zu simulieren. Dies ermöglicht es uns, gezielt Nachrichten zu generieren, welche die Agents erkennen und melden sollten.

5.1.2 Agents

MacMan und DeviceMan

Die Mac- und DeviceMan Applikationen sind Proof of Concept Implementationen eines Agents. Sie nehmen geparste Netflow Daten an und halten sich eine Liste über alle Geräte, die in den Logs aufgetaucht sind. Identifiziert werden diese aufgrund ihrer MAC-Adresse. Bei jeder MAC-Adresse, welche dieser Agent überprüft, wird ein Alert generiert. Taucht eine neue MAC-Adresse und somit ein unbekanntes Gerät auf, so wird die Severity des Alerts auf Danger gesetzt.
MacMan schreibt zudem beim Start des Programmes seine aktuelle Liste von bekannten MAC-Adressen in eine Message Queue, damit im Nachhinein die Entscheidungen von MacMan im Persistenzlayer nachvollzogen werden können.
Für die XMPP Python Skripts haben wir den Beispiel Code für PubSub des Sleekxmpp Projektes[8] als Basis verwendet.

BasicNotifier und SimpleNotifier

Diese Applikationen sind prototypen, um Alerts zu visualisieren. Sie melden sich jeweils bei einem Kafka Topic oder einem XMPP Node an und geben alle erhaltenen Nachrichten auf der Konsole aus.

Webnotifier

Der Webnotifier ist eine Weiterentwicklung des BasicNotifier. Er bietet eine Webseite wie auch den dazugehörigen Websocket über einen eigenen Webserver an. Dies ermöglicht es, die Alerts direkt über die Sockets zu pushen. Im Gegensatz zu den Applikationen BasicNotifier und SimpleNotifier stellt der Webnotifier die Severity eines Alerts farbig auf der Seite dar. Dies ermöglicht es, die wichtigen von den unwichtigen Meldungen zu unterscheiden.

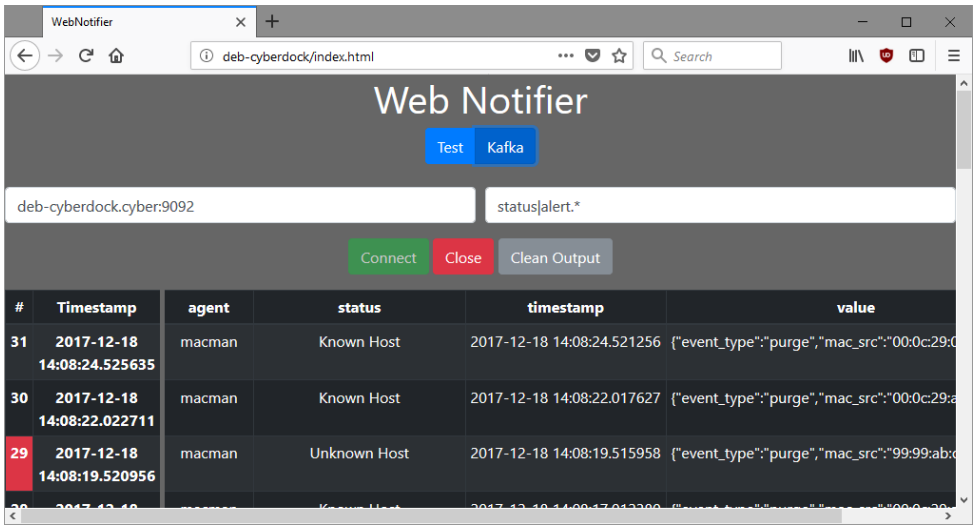

Abbildung 5.2: Webnotifier Screenshot

5.1.3 Hilfsprogramme

XMPP-Initializer

Der XMPP-Initializer ermöglicht es, ein Konstrukt von XMPP Nodes als JSON zu deklarieren und dann automatisch auf einem XMPP Server zu konfigurieren.
Dieses Hilfsprogramm ist entstanden, da bei XMPP das Konfigurieren von Nodes über das XEP-60 Protokoll geschieht, welches von Hand fast nicht zu bedienen ist.
Der XMPP-Initializer kann ein baumartiges Konstrukt von Nodes und Child-Nodes rekursiv einlesen und aufsetzen. Dies beinhaltet die notwendigen Publish-Berechtigungen sowie die gewünschten Subscriptions.
Der grosse Vorteil ist das schnelle Aufsetzen sowie die Möglichkeit, dass sich nicht jeder Agent nach dem Starten zuerst auf eine Node subscriben muss, um seine Messages zu bekommen. Ausserdem verhindert dies ein doppeltes Subscriben auf eine Node, was Openfire nicht verhindert.

Das folgende JSON deklariert eine Hierarchie mit drei Stufen und beschreibt die entsprechenden Berechtigungen.

```
{   "nodes": [
        {   "id": "/collection",
            "type": "collection",
            "title": "Collection Node",
            "description": "A collection node.",
            "subscriber": [ "subscriber@xmpp.host" ],
            "nodes": [
                {   "id": "/collection/subcollection",
                    "type": "collection",
                    "title": "Sub Collection Node",
                    "description": "An intermediate collection node.",
                    "subscriber": [ ],
                    "nodes": [
                        {   "id": "/collection/subcollection/leafnode",
                            "type": "leaf",
                            "title": "Leaf Node",
                            "description": "A leaf node.",
                            "publisher": [ "publisher@xmpp.host" ],
                            "subscriber": [ "leaf-subscriber@xmpp.host" ]
                        }
                    ]
                }
            ]
        }
    ]
}
```

Listing 5.1: Node setup JSON

Nachrichten welche von publisher@xmpp.host im Leaf-Node publiziert werden, werden automatisch nach oben propagiert. Am Ende bekommen sämtliche Subscriber die publizierte Nachricht.

6 Einsatzszenarien

Die entwickelten wie auch die evaluierten Applikationen können in verschiedenen Konstellationen betrieben werden. In diesem Kapitel beschreiben wir diese Szenarien, welche wir mittels Docker-Compose [9] automatisiert haben.
Den Persistenzlayer haben wir als eigenes Szenario abgebildet, da dieser Layer unabhängig von dem Messaging System laufen soll.

6.1 Kafka

Im Szenario Kafka wird ein Kafka Server als Messaging System eingesetzt. Eine Pmacct Instanz wird als Netflow Probe verwendet. Diese wandelt die Netflow Nachrichten in JSON um und spielt diese in das Messaging System ein. Danach erkennt MacMan, ob es sich bei den Teilnehmern im Netzwerk um bereits bekannte oder um neue Geräte handelt, und versendet dementsprechend Alerts.
Diese Alerts können dann mit Hilfe des Webnotifier direkt im Web über Sockets oder über die Konsole mit dem BasicNotifier angezeigt werden.

- Kafka Node
- Zookeeper (Kafka Node Manager)
- Pmacct
- MacMan
- BasicNotifier
- Webnotifier

Queue-Name	**Publisher**	**Subscriber**
mac.queue	Pmacct	MacMan, Webnotifier, Logstash
alert.mac	MacMan	BasicNotifier, Webnotifier, Logstash

Tabelle 6.1: Kafka Topics

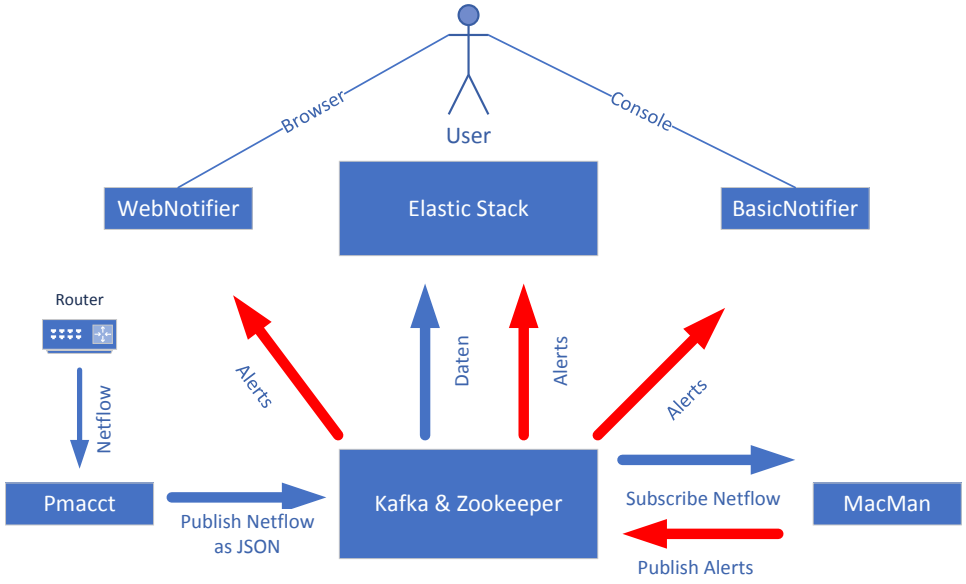

Abbildung 6.1: Kafka Einsatzszenario

6.2 XMPP

Im XMPP Szenario wird das Messaging von Openfire übernommen. Da es mit unserem Netflow Umwandler Pmacct leider nicht möglich ist, auf XMPP Nodes zu publishen, wird dieser durch die LogReplay Applikation substituiert.

- Openfire XMPP Server
- LogReplay
- DeviceMan
- SimpleNotifier

Queue-Name	**Publisher**	**Subscriber**
/input/netflow	LogReplay	DeviceMan, SimpleNotifier
/alert/devices	DeviceMan	SimpleNotifier

Tabelle 6.2: XMPP Nodes

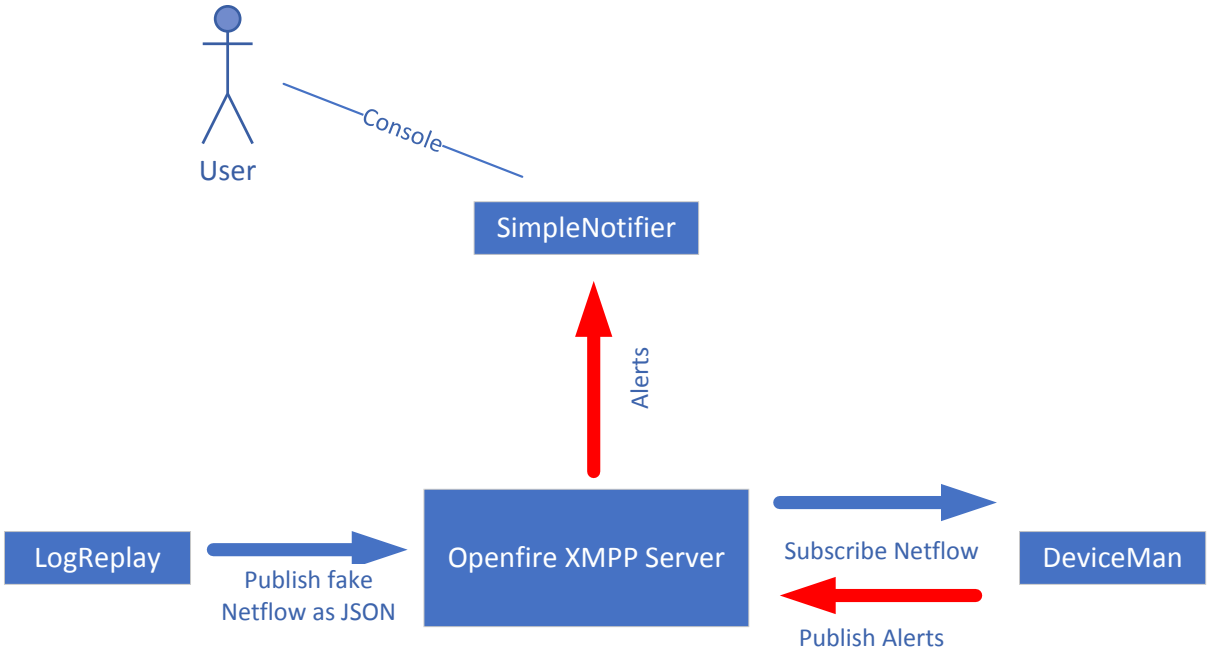

Abbildung 6.2: XMPP Einsatzszenario

6.3 Demo Szenario

Um unsere Arbeit etwas ansprechend zu präsentieren, haben wir auf das Kafka Szenario gesetzt. Anstelle von Pmacct, der echte Netflow Daten einspielt, verwenden wir den LogRerunner. Er erlaubt uns, die Daten kontrolliert in das System einzuspielen.

- Kafka Node
- Zookeeper (Kafka Node Manager)
- LogRerunner
- MacMan
- Webnotifier

Queue-Name	Publisher	Subscriber
mac.queue	LogRerunner	MacMan, Webnotifier, Logstash
alert.mac	MacMan	Webnotifier, Logstash
alert.random	LogRerunner	Webnotifier, Logstash

Tabelle 6.3: Kafka Topics Demo

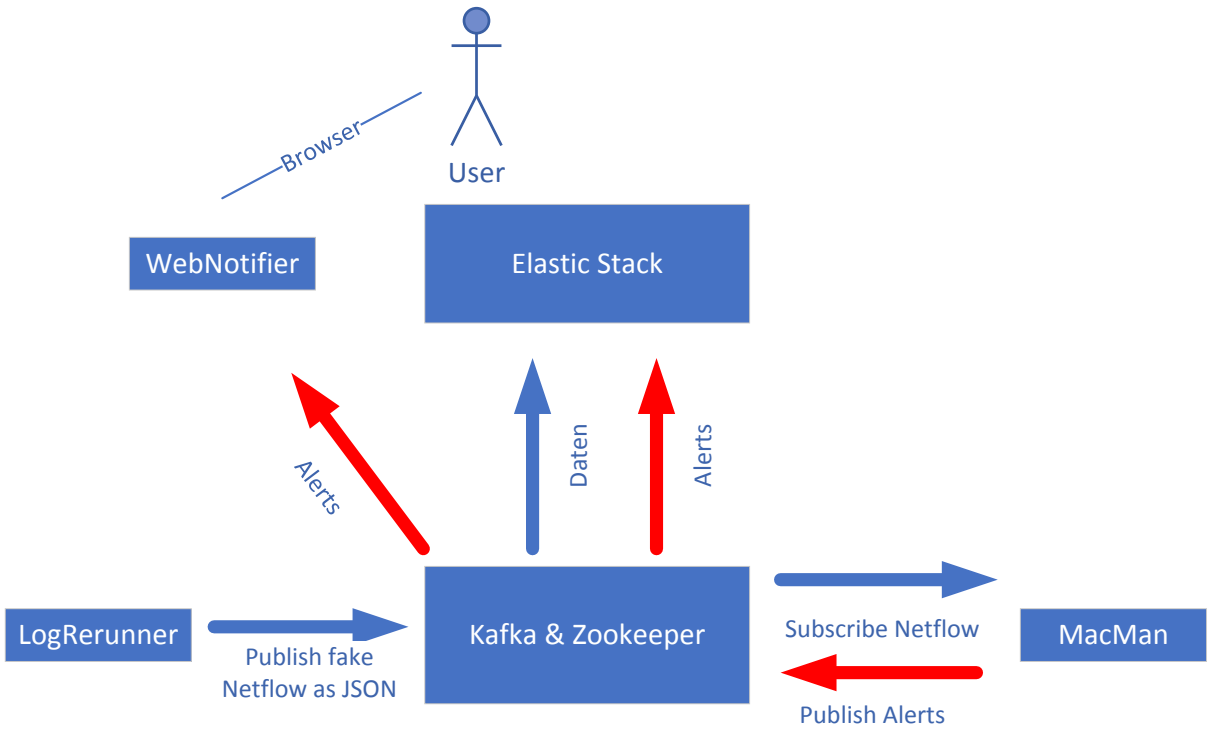

Abbildung 6.3: Demo Einsatzszenario

6.4 Persistenz

Die Persistenz der Daten übernimmt der Elastic Stack. Elasticsearch speichert die Daten. Logstash spielt mittels Plugins Daten von den Messaging Systemen auf Elasticsearch. Kibana ist ein WebUI, dass die Daten ansprechend darstellt.

- Elasticsearch
- Logstash für XMPP
- Logstash für Kafka
- Kibana

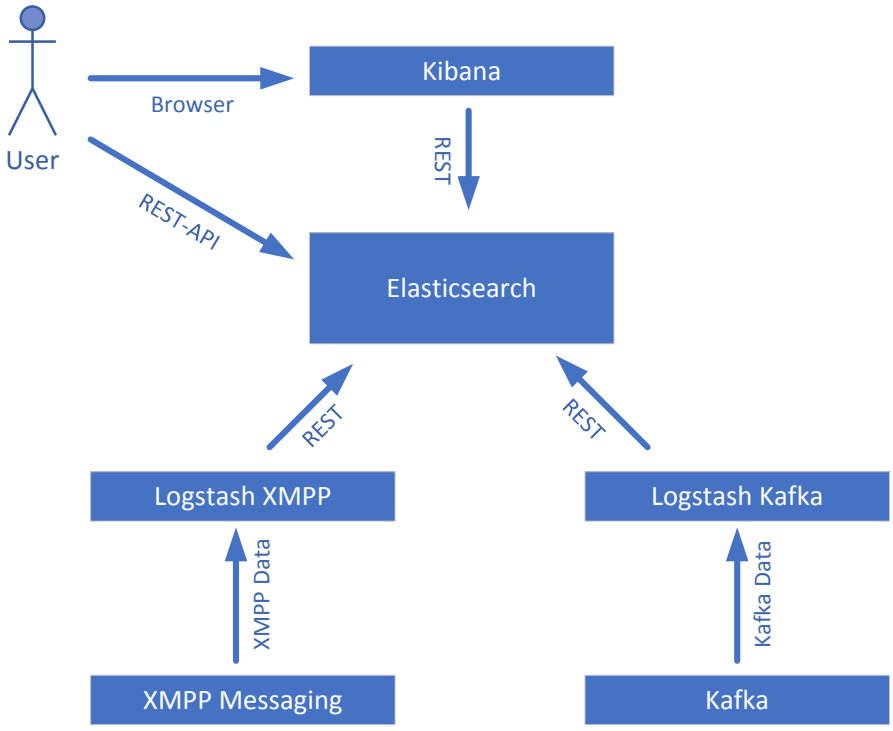

Abbildung 6.4: Persistenz Einsatzszenario

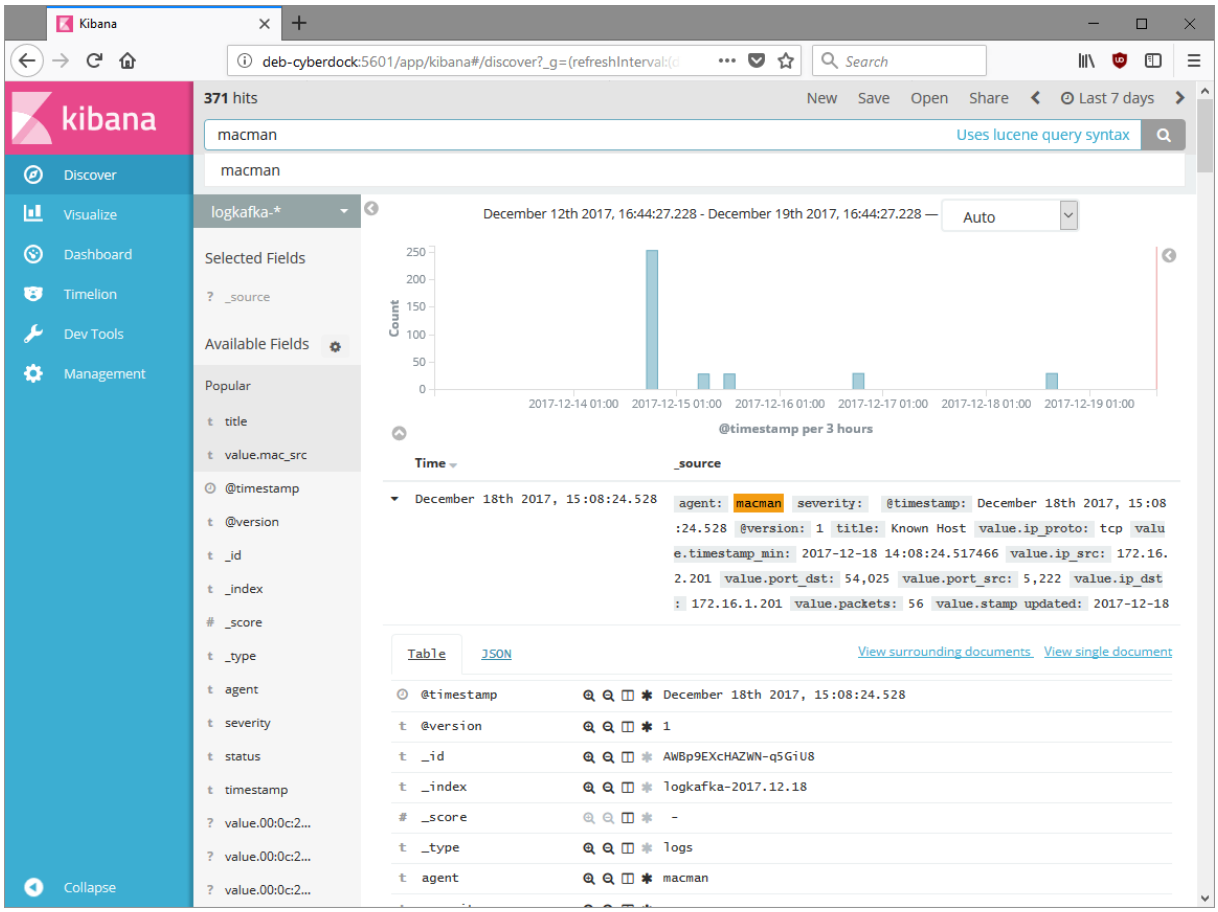

Abbildung 6.5: Screenshot Kibana

7 Benchmark

Um die zwei verschiedenen Messaging Kandidaten zu vergleichen, wird ein Benchmark durchgeführt. Es sollen kleine wie auch grosse Payloads getestet werden.

7.1 Vorgehen

Es wird ein Programm geschrieben, welches als Source und Sink fungiert. Die Source schickt Daten einer bestimmten Länge an eine bestimmte Queue des Messaging Systems. In einem weiteren Feld wird der Zeitpunkt der Übergabe an die Publish-Funktion des Messaging Systems vermerkt. Die Source misst die Zeit jeder Repetition.
Die Sink nimmt die Daten der Queue entgegen und misst die Zeit vom Eintreffen der ersten Nachricht bis zum Eintreffen der Letzten. Ausserdem bestimmt sie die absolute Zeit zwischen Übergabe der Nachricht an die Publish-Funktion und des Empfangs. Die Programme sind in Python geschrieben und verwenden die Libraries Sleekxmpp sowie kafka-python.
Es werden folgende Testsuiten verwendet:

	Small	**Large**
Size [Byte]	512	512'000
Count	200	200
Repetitions	5	5

Tabelle 7.1: Benchmark Testsuiten

Die Messungen werden auf einer Docker-VM durchgeführt. Damit soll erreicht werden, dass die physische Netzwerkinfrastruktur nicht mitgemessen wird. Es laufen nur die notwendigen Container. Wird eine Testsuite erneut getestet, werden die entsprechenden Container neu gestartet.
Um den Nebenaspekt des Overheads zu messen, lassen wir separat den Large Benchmark laufen, und schauen vor und nach der Durchführung die Interface-counters an. Docker erstellt jeweils eine virtuelle Bridge für jeden Container, daher müssen die Counters nicht zurückgesetzt werden.

7.2 Durchführungen

7.2.1 XMPP

XMPP wird mit Openfire 4.1.6 mit dem Compose-Szenario scenario_xmpp getestet. Die Benchmark Definitionen befinden sich im Szenario benchmark_xmpp.

Vorbereitung

Folgende Commands wird vor jeder Durchführung ausgeführt.

```
xmpp_initializer --mode cleanup
# alle laufenden Container beenden
docker-compose up -d openfire
xmpp_initializer.py --mode create
xmpp_initializer.py --mode status
```

Listing 7.1: Initialize XMPP Benchmark

Large

```
docker-compose -f large-test.yml build
docker-compose -f large-test.yml up -d
docker-compose -f large-test.yml logs sink > 20171203_xmpp_large_sink.txt
docker-compose -f large-test.yml logs source > 20171203_xmpp_large_source.txt
```

Listing 7.2: XMPP Large Benchmark

Small

```
docker-compose -f small-test.yml build
docker-compose -f small-test.yml up -d
docker-compose -f small-test.yml logs source > 20171203_xmpp_small_source.txt
docker-compose -f small-test.yml logs sink > 20171203_xmpp_small_sink.txt
```

Listing 7.3: XMPP Small Benchmark

7.2.2 Kafka

Vorbereitung

Folgende Commands werden vor jeder Durchführung ausgeführt.

```
# alle laufenden Container beenden
docker-compose up -d --build zookeeper kafka
```

Listing 7.4: Initialize Kafka Benchmark

Large

```
docker-compose -f large-test.yml build
docker-compose -f large-test.yml up -d
docker-compose -f large-test.yml logs source > 20171203_kafka_large_source.txt
docker-compose -f large-test.yml logs sink > 20171203_kafka_large_sink.txt
```

Listing 7.5: Kafka Large Benchmark

Small

```
docker-compose -f small-test.yml build
docker-compose -f small-test.yml up -d
docker-compose -f small-test.yml logs source > 20171203_kafka_small_source.txt
docker-compose -f small-test.yml logs sink > 20171203_kafka_small_sink.txt
```

Listing 7.6: Kafka Small Benchmark

7.3 Resultate

Folgendes haben wir gemessen:

	XMPP Large	**Kafka Large**	**XMPP Small**	**Kafka Small**
Zeit Senden avg [s]	23.842	1.363	0.129	0.017
Zeit Empfangen avg [s]	23.808	1.929	0.267	0.048
Zeit Übertragen min [s]	0.151	0.094	0.018	0.049
Zeit Übertragen max [s]	24.737	4.307	0.836	0.223
Zeit Übertragen avg [s]	12.125	2.626	0.616	0.166

Tabelle 7.2: Benchmark Resultate

	Source TX	**Sink RX**	**Source RX**	**Sink TX**
Kafka	512785043	513259703	447871	504087
XMPP	515710723	515703869	691103	2370602
XMPP Mehrfaktor	1.0057	1.0048	1.5431	4.7028

Tabelle 7.3: Interface-Counters [Byte] Resultate

7.3.1 Interpretation

Es ist offensichtlich, dass die Kafka Lösung die Datenmenge viel schneller verarbeitet. Einen so grossen Unterschied haben wir nicht erwartet.
Wir haben daher unser Programm einem profiling unterzogen, um einen eventuellen Flaschenhals zu finden. Im Output der Sink fällt auf, dass die einzelnen Messages bei XMPP

immer etwa dieselbe Zeit brauchen, während mit mehr Messages bei Kafka dies länger dauert. Ausserdem läuft bei XMPP die Source etwa gleich lange wie die Sink, obwohl wir bei Übergabe der Message den Block Parameter auf *False* setzen.

```
C:\apps.pstat% stats 5
Thu Nov 30 17:34:32 2017    C:\apps.pstat

         508100116 function calls (508087476 primitive calls) in 125.508 seconds

   Ordered by: internal time
   List reduced from 1599 to 5 due to restriction <5>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2500   42.175    0.017   61.434    0.025 tostring.py:142(escape)
        1   36.361   36.361   53.705   53.705 source.py:13(__init__)
256070604   17.606    0.000   17.606    0.000 {method 'get' of 'dict' objects}
251753914   17.344    0.000   17.344    0.000 xmpp.py:33(isready)
      527    4.787    0.009    4.787    0.009 {method 'acquire' of
                                                '_thread.lock' objects}
```

Listing 7.7: XMPP Profiling

Die Escape-Funktion von Sleekxmpp geht jeden zu sendenden String fünfmal durch und baut danach einen neuen String aus allen Zeichen zusammen. Dies benötigt fast 50 Prozent der gesamten Laufzeit.
Die Belastung des Netzwerkes ist in etwa gleich, interessanterweise brauchte aber der Traffic zwischen Openfire und der Sink etwa 4.7-mal mehr Daten als Kafka.

8 Outlook

Wie bereits im Abstract erwähnt, ist ein solches Framework nur so gut wie die eingesetzten Agents und Probes. Mit unserer Arbeit haben wir gerade einmal an der Spitze des Eisbergs gekratzt. Das Sicherstellen der IT-Security ist und bleibt ein sehr komplexes Gebiet, welches nicht so einfach durch ein Programm ersetzt werden kann. Dies zeigt die Komplexität von bestehenden Projekten in diesem Bereich. Der Ansatz von unserem Framework erlaubt aber eine sehr einfache Integration. Jeder, der etwas Programmierkenntnisse hat, ist in der Lage, eine Probe oder einen Agent zu schreiben. Bevor dieses Projekt veröffentlicht wird, muss man sich aber noch Gedanken über die Standardisierung machen, so dass die Probes und Agents sich auch untereinander verstehen können.

Nachfolgend stellen wir einige mögliche Erweiterungen vor:

8.1 Mögliche Probes

Syslog Dies erlaubt es, die Logs von nahezu allen Prozessen auf einem Server in die Überwachung einzuspielen. Die Menge an entsprechenden Agents beziehungsweise die Komplexität der Agents wird aber drastisch gesteigert, je mehr Arten von Logs man analysieren möchte.

Network Packets Anstatt nur Netflow zu verwenden, könnte man auch mit einem Mirror Port den gesamten Netzwerkverkehr in das System einspielen. Pmacct könnte man für diesen Use Case einsetzen.

External Ressources Externe Ressourcen wie Twitter Feeds von bekannten Sicherheitsforschern, Blogs oder Zeitschriften könnten nach spezifischen Themen gecrawlt und in das System eingespielt werden. Die könnte für eine Justierung der Agents, oder auch zur Information des Betreibers verwendet werden.

CVE Datenbank Es ist wichtig, dass das System laufend von neuen Schwachstellen und Exploits in Kenntnis gesetzt wird. Eine Anbindung an eine Vulnerability DB stellt dies sicher.

Client Tool Ein Programm, welches Informationen über die Kundensysteme bereitstellt. Aktuelle Paketversionen, stand der Antivirendatenbank, CPU Auslastung, physische Netzwerkadressen, usw.

8.2 Mögliche Agents

Netzwerk Patterns Wer kommuniziert mit wem, wieviel und wie oft. Erkennen von Clients, die so viel Traffic generieren wie die Server. Ein Client, der ausserordentlich viele Verbindungen mit anderen Clients aufbaut.

Portscan Detection Man bräuchte die Netflowdaten und analysiert diese auf einen Portscan.

Ransomware Detection Ist die CPU Auslastung eines Systems deutlich grösser als normalerweise, könnte dies auf einen Verschlüsselungsprozess durch Ransomware hindeuten.

Schwachstellen Eine laufende Überprüfung der aktuellen Sicherheitslücken und der verwendeten Software könnte Aufschluss über potenziell gefährdete Systeme im Unternehmen geben.

DDOS Attacke Durch das Analysieren der Syslogs kann bei einem Webserver festgestellt werden, ob er besonders viele Anfragen verwerfen muss. So kann ein gewisser IP Range blockiert werden, um die Auswirkungen einer solchen Attacke zu minimieren.

9 Projekt Management

Für das Projektmanagement wird die Software JIRA von Atlassian verwendet.

9.1 Projektplan

Für die Projektplanung haben wir uns vor dem Kick-Off Meeting je einmal mit unserem Business Partner und mit unserem Betreuer verabredet und unsere Gedanken ausgetauscht. Die Planung war insofern schwierig, da sich das Thema einer Cyber Security Integration Engine als sehr vage erwies.

Elaboration (Wochen 1 - 5) Die Elaboration soll bis Woche fünf dauern. Sie wird abgeschlossen mit dem Meilenstein «End of Elaboration». Aus der Elaboration soll hervorgehen, mit welchen Technologien der Prototyp umgesetzt wird. Aufgrund der offenen Projektbeschreibung wird diese Phase absichtlich etwas länger angelegt.

Construction (Wochen 6 - 12) Nach der Elaboration soll ein Softwareprojekt umgesetzt werden, welches die gewonnenen Erkenntnisse umsetzt. Am Ende von Woche neun soll ein Prototyp vorgestellt werden können, welcher die Erfassung, Übermittlung und Verarbeitung zeigen soll. Am Ende von Woche zwölf soll aus diesem Prototypen eine Art Minimum Viable Product (MVP) entstehen, welches dem Kunden präsentiert werden kann. Das MVP soll Nachrichten von einer Probe mit einem Agent weiterverarbeiten und Alerts werfen können.

Transition (Wochen 13 - 14) In der Transition Phase tragen wir die gefundenen Lösungsansätze zusammen zu einem Paket, welches wir am Ende der Studienarbeit dem Kunden präsentieren können. In dieser Phase ist auch Zeit, um die Dokumentation zu vervollständigen und zu bereinigen.

9.2 Arbeitspakete

Wir haben den Workflow im JIRA manuell angepasst. Unser Workflow ermöglicht es, Arbeitspakete einem Teammitglied zu einem Review zu übergeben.

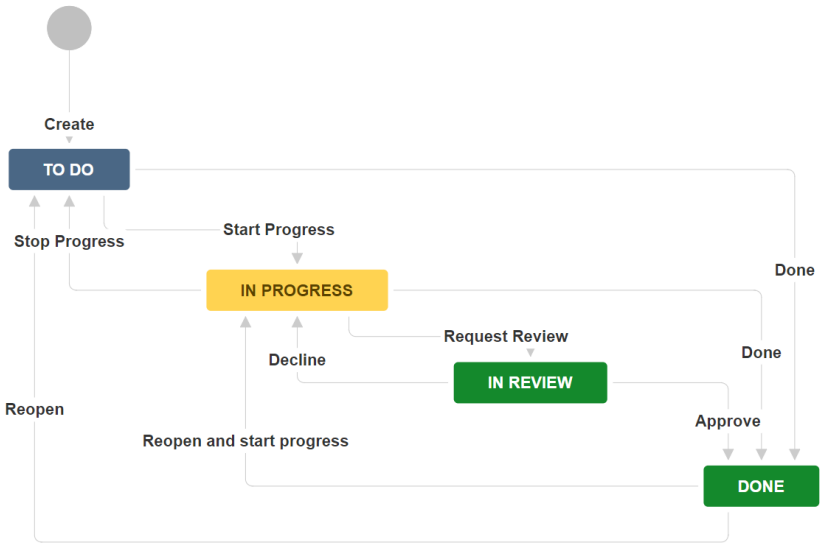

Abbildung 9.1: Workflow in JIRA.

9.2.1 Sprints

Elaborate 1

Das Lab funktioniert, die Infrastruktur ist bekannt. Es werden verschiedene Ansätze diskutiert.

Geplante Sprintlänge: 2 Wochen

Elaborate 2

Die Meilensteine können gesetzt werden. Erster PoC für Probes und Agents erstellen mit verschiedenen Technologien. Entwicklungsumgebungen sind einsatzbereit.

Geplante Sprintlänge: 2 Wochen

Compare / Decide

Die Möglichkeiten, welche man gefunden hat, werden dokumentiert. Man legt sich auf einen Lösungsansatz fest.

Geplante Sprintlänge: 1 Wochen

Prototype

Mit den Technologien welche man gelernt hat wird ein Prototyp erstellt.

Geplante Sprintlänge: 3 Wochen

Construction Aus dem Prototyp sollen die wichtigsten Funktionen in ein MVP umgewandelt werden welcher vom Kunden eingesetzt werden kann.

Geplante Sprintlänge: 3 Wochen

Transition Die Dokumentation wird aktualisiert mit den neusten Erkenntnissen. Abschliessen des MVP.

Geplante Sprintlänge: 1 Woche

Finalization Eine Woche als Reserve für administrative Aufgaben. SIe kann auch für das Schreiben der Dokumentation verwendet werden.

Geplante Sprintlänge: 1 Woche

9.2.2 Reviews und Retroperspektiven

Nach dem Beenden von jedem Sprint haben wir das Burndownchart in einer kurzen Retroperspektive besprochen. Sehr oft mussten Arbeitspakete in den nachfolgenden Sprint übertragen werden.

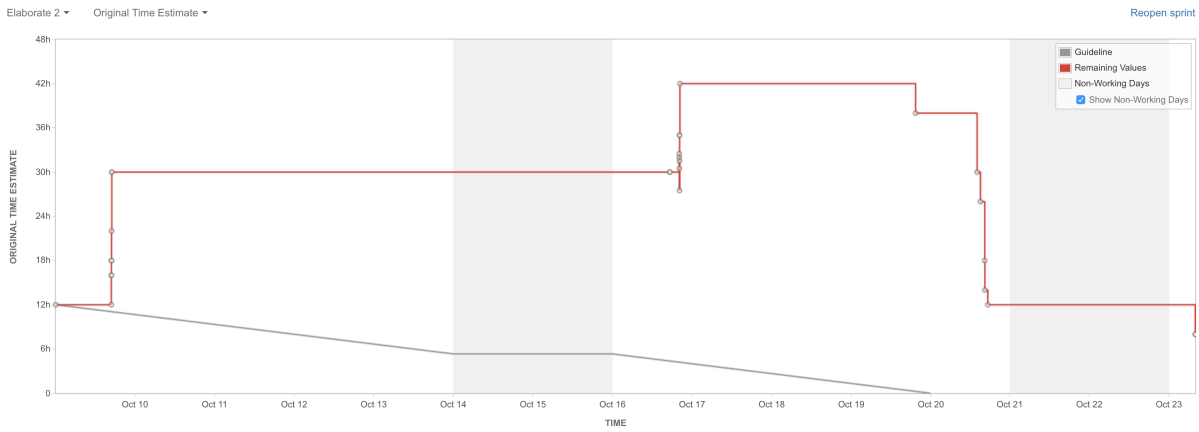

Abbildung 9.2: Burndown des Elaboration 2 Sprints

9.3 Meilensteine

Da wir uns in diesem Projekt mehrheitlich mit dem Erproben einer möglichen Lösung beschäftigt haben, sind die Phasen der Elaboration verhältnismässig lange.

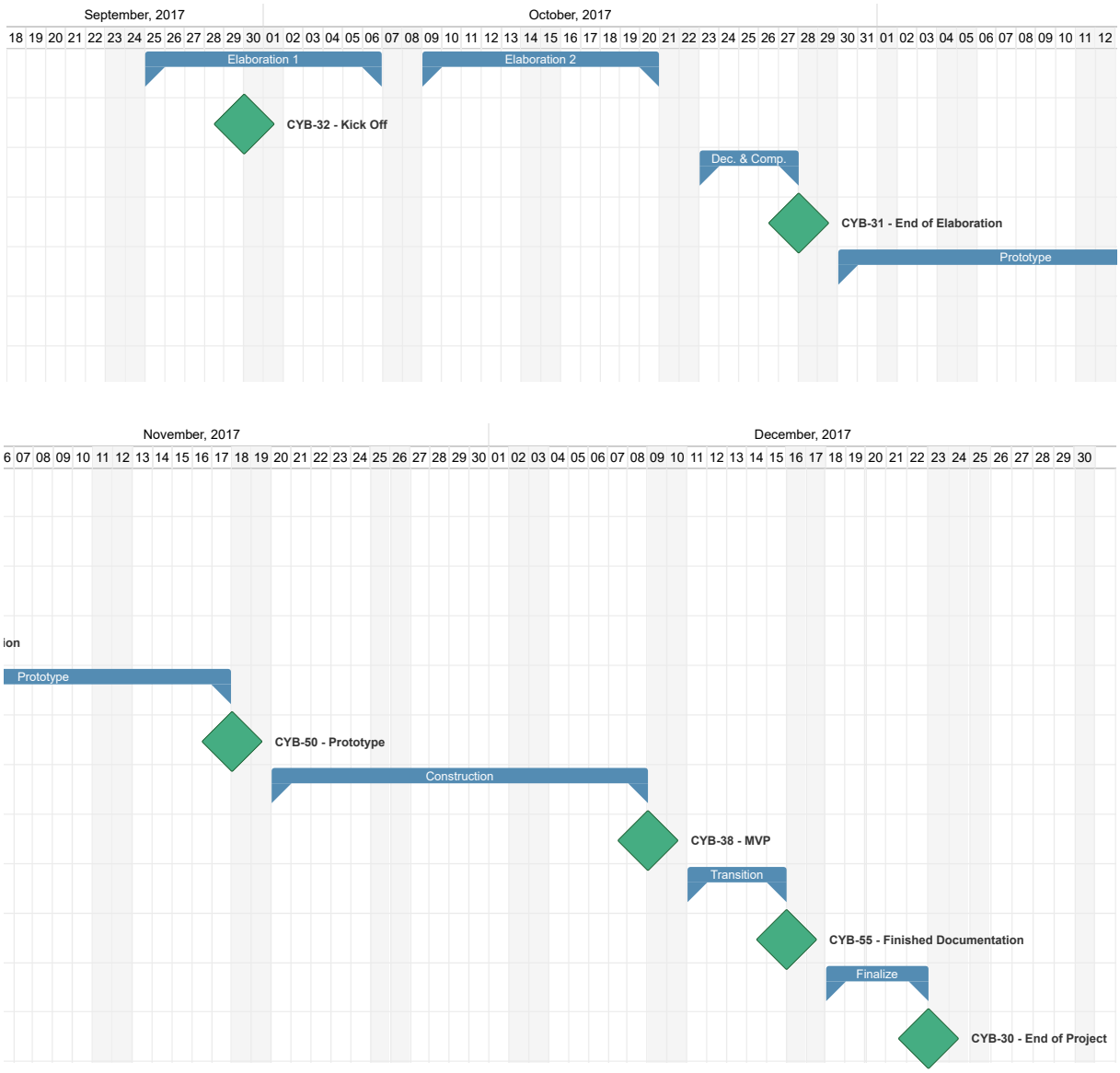

Abbildung 9.3: Meilensteine aufgeteilt in zwei Projekthälften

9.4 Termine

Obwohl wir mit einem agilen Projektmanagement unterwegs waren, haben wir die Meilensteine zeitlich festgelegt.

Kick Off	Geplant: 29.09.2017 Erreicht: 29.09.2017
End of Elaboration	Geplant: 27.10.2017 Erreicht: 03.11.2017 Das Einlesen in die neuen Themen brauchte mehr Zeit als geplant.
Prototype	Geplant: 17.11.2017 Erreicht: 08.12.2017
MVP	Geplant: 08.12.2017 Erreicht: – Wurde weggelassen. Dafür wurde der Prototyp weiterentwickelt.
End of Project	Geplant: 22.12.2017 Erreicht: 22.12.2017

9.5 Zeiterfassung

Die Zeiterfassung wurde ebenfalls innerhalb von JIRA mithilfe des TEMPO Plug-Ins gemacht. Dies ermöglicht eine einfache Auswertung der geplanten sowie der geleisteten Arbeitszeit.

10 Verzeichnisse

10.1 Glossar und Abkürzungen

Agent

Ein Subsystem, welches Informationen aus dem Messaging System liest und Ereignisse zurückspielt.

Apache Kafka

Verteiltes Messaging System basierend auf Java. [2]

certbot

Certbot ist eine Software, die automatisiert bei Let's encrypt Zertifikate bezieht. [10]

Docker

Docker ist ein Wrapper um Linux-Kernel Prozessvirtualsierungsfeatures und vereinfacht die Automatisierung von Applikationsdeployments.[9]

Elastic Stack

Elastic Stack, ehemals ELK-Stack, ist die Lösung von Elastic co., um grosse Datenmengen zu indexieren und durchsuchbar zu machen. [11]

ESXi

VMware ESXi ist ein Hypervisor. [12]

Jabber

Jabber ist die ursprüngliche Bezeichnung von XMPP.

Jira

Jira ist ein webbasierendes Projektmanagement Tool von Atlassian. [13]

kafka-python

Python Client Library für die Kommunikation mit Kafka Servern. [14]

Let's encrypt

Let's encrypt ist eine CA, die über eine API kostenlose Serverzertifikate herausgibt. [15]

Netflow

Ein von Cisco entwickeltes Protokoll, dass Metadaten über den Netzwerkverkehr an einen Server schickt.

Nginx

Nginx (Engine-X) ist ein Webserver und Reverse-Proxy. [16]

Openfire

XMPP Server basierend auf Java. [4]

Pmacct

Softwareprojekt, das Netflow parsen und weiterverabeiten kann. [6]

Postgres

Postgres ist ein freies Datenbank Management System (DBMS). [17]

Probe

Ein Subsystem, welches Informationen in das Messaging System einspielt.

PubSub

Publish and Subscribe

Sleekxmpp

Python Client Library für die Kommunikation mit XMPP Servern. [18]

XEP-60

Ein XEP Standard, welcher von XMPP Servern verwendet wird, um PubSub Nodes zu konfigurieren und Nachrichten auszutauschen. [19]

XMPP

Extensible Messaging and Presence Protocol. Ein Protokoll um Nachrichten auszutauschen. [3]

10.2 Abbildungsverzeichnis

10.3 Tabellenverzeichnis

10.4 Literatur

[1] F. Skoog, Image of Example Architecture, 2017.

[2] Apache Kafka - A distributed streaming platform, [Online; aufgerufen am 1. Dez. 2017]. Adresse: `https://kafka.apache.org`.

[3] XMPP - Extensible Messaging Presence Protocol, [Online; aufgerufen am 1. Dez. 2017]. Adresse: `https://xmpp.org/about`.

[4] Openfire Project, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://igniterealtime.org/projects/openfire.

[5] Kafka Documentation, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://kafka.apache.org/documentation.

[6] P. Lucente, Pmacct Project, [Online; aufgerufen am 1. Dez. 2017]. Adresse: http://www.pmacct.net.

[7] Private Repo SAIDocker. Adresse: https://github.com/maederm/SAIDocker.

[8] SleekXMPP Examples, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://github.com/fritzy/SleekXMPP/tree/develop/examples.

[9] Docker Project, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://www.docker.com.

[10] Certbot, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://certbot.eff.org.

[11] Elastic Stack, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://www.elastic.co/solutions/logging.

[12] vSphere Hypervisor, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://www.vmware.com/products/vsphere-hypervisor.html.

[13] Jira, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://www.atlassian.com/software/jira.

[14] Kafka Python client, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://github.com/dpkp/kafka-python.

[15] Let's Encrypt, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://letsencrypt.org.

[16] Nginx, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://nginx.org.

[17] PostgreSQL, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://www.postgresql.org.

[18] SleekXMPP, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://github.com/fritzy/SleekXMPP.

[19] XEP-0060: Publish-Subscribe, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://xmpp.org/extensions/xep-0060.html.

[20] MiKTeX, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://miktex.org.

[21] Travis Continous Integration, [Online; aufgerufen am 1. Dez. 2017]. Adresse: https://travis-ci.com.

11 Appendix

11.1 Work Environment

Unsere Szenarien liessen auf verschiedenen Infrastrukturen laufen, siehe Abbildung 11.1. Im SA-Zimmer haben wir auf einem unbenutzten Host VMware ESXi installiert und als lokales LAB verwendet. Auf diesem Hypervisor läuft primär ein Docker-Host, wie auch ein virtueller Router, um Testdaten zu erhalten, eine virtuelle Firewall und kleinere Traffic-Testsysteme.
Um auf dieser Infrastruktur zu arbeiten, installierten wir statische Routen auf unseren Jumphosts und setzten als DNS Resolver die virtuelle Firewall.
Im INS haben wir eine leistungsstarke Ubuntu VM erhalten, wie auch Netflow-Daten des Schulungsraumes. Da der verwendete Switch keine MAC-Adressinformationen über Netflow verschicken konnte, brauchten wir das System jedoch nicht lange.
In der Student-DMZ lassen wir Jira für das Projektmanagement laufen.

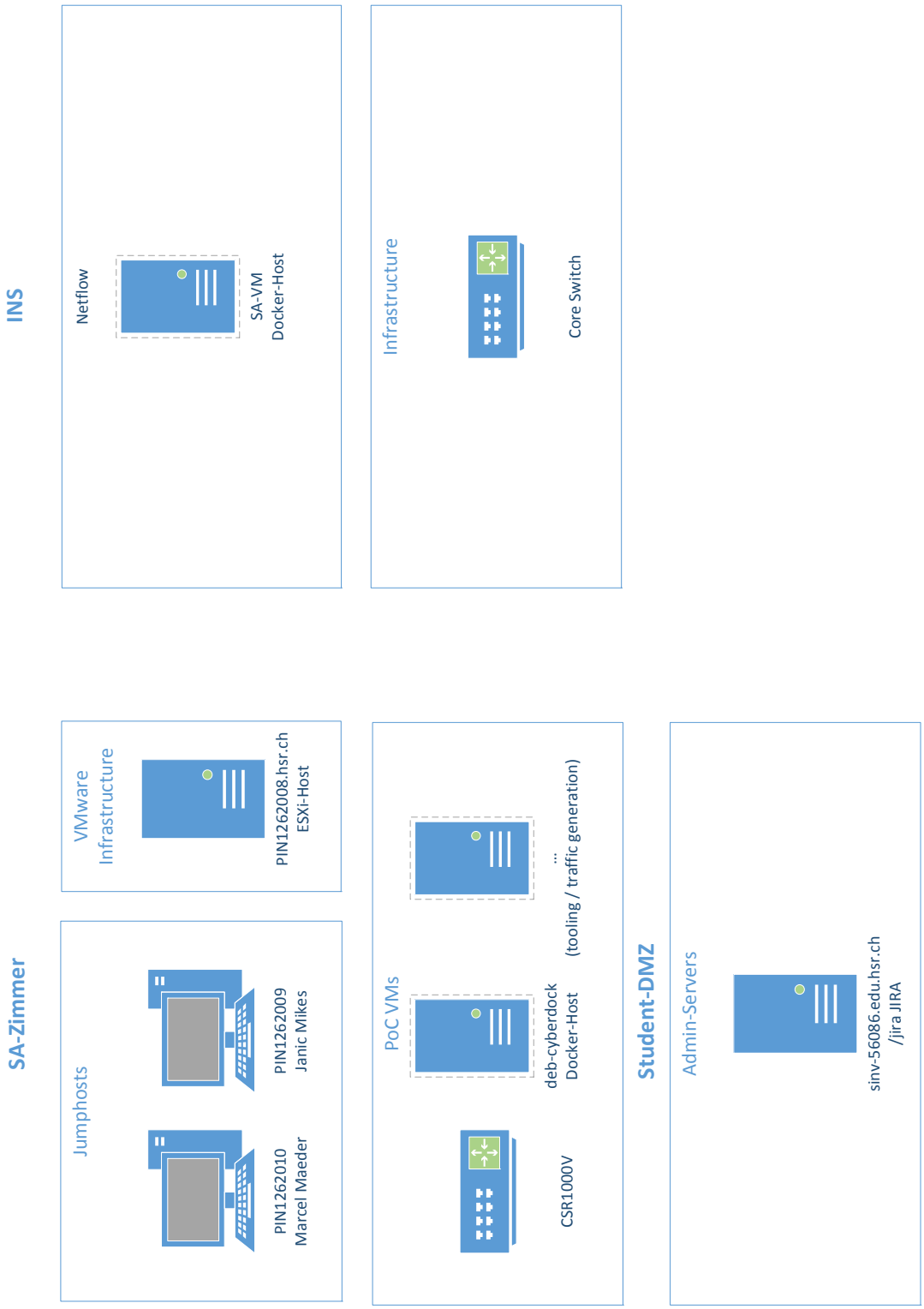

Abbildung 11.1: Infrastruktur

Jira

Jira läuft hinter dem Nginx Reverse-Proxy, der ein Let's encrypt Zertifikat mittels certbot eingespielt bekommt. Damit Jira damit umgehen kann, müssen einige Änderungen am Configfile server.xml vorgenommen werden. Als Datenbank verwenden wir Postgres.

```
[...]
<Service name="Catalina">
    <!-- Reverse Proxy Connector -->
    <Connector  port="8080"
                maxThreads="150"
                minSpareThreads="25"
                connectionTimeout="20000"
                enableLookups="false"
                maxHttpHeaderSize="8192"
                protocol="HTTP/1.1"
                useBodyEncodingForURI="true"
                redirectPort="8443"
                acceptCount="100"
                disableUploadTimeout="true"
                bindOnInit="false"
                address="127.0.0.1"
                scheme="https"
                proxyPort="443"
                proxyName="sinv-56086.edu.hsr.ch" />

    <!-- Debug Connector -->
    <Connector  port="8081"
                address="127.0.0.1"
                maxThreads="150"
                minSpareThreads="25"
                connectionTimeout="20000"
                enableLookups="false"
                maxHttpHeaderSize="8192"
                protocol="HTTP/1.1"
                useBodyEncodingForURI="true"
                redirectPort="8443"
                acceptCount="100"
                disableUploadTimeout="true"
                bindOnInit="false" />
[...]
```

Listing 11.1: Jira server.xml

Nginx lässt den Pfad /jira an den Appserver weitergeben.

```
server {
    listen sinv-56086.edu.hsr.ch:80;
    server_name sinv-56086.edu.hsr.ch;
    location /jira {
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Server $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://localhost:8080/jira;
        client_max_body_size 10M;
    }
}
```

Listing 11.2: Ngnix config

Certbot läuft nach dem Durchlaufen des Wizards automatisch und nimmt die notwendigen Änderungen an der Nginx Konfiguration selbst vor.
Weil es kein garantiertes Backup dieses Servers von der Schule gibt, haben wir dies automatisiert. Das Backup beinhaltet die Datenbank und die Jira-Daten. Das Archiv dieser Daten laden wir automatisiert auf OneDrive hoch. Das Backup müssen wir danach automatisch löschen, da der Platz auf dieser VM zu knapp ist.

```
#!/bin/sh
dirname=`date +%Y%m%d_%H%M`
mkdir $dirname
cd $dirname
su - postgres -c "pg_dump jiradb" > db_backup.sql
tar -cf jira-data.tar /var/atlassian/application-data/jira
cd ..
7z a $dirname.7z $dirname
rclone copy $dirname.7z jirabackup:/jirabackup/
rm -r $dirname
rm -r $dirname.7z
```

Listing 11.3: Jira automatisiertes Backup

11.2 Framework

Hier folgen Auszüge aus dem Quellcode unserer Programme und wichtige Konfigurationen.

Probes

Logreplay

```
def replay(self, rate):
    print('replay %s lines with offset at: %s' % (self.nooflines(), rate))
    with open(self.file) as fp:
        if self.file.lower().endswith(('.json', '.json.log')):
            prefix = '{ "data_type": "%s", "generator": "%s", "value":' % ('json',
 ↪  self.generator)
            postfix = ' }'
        else:
            prefix = '{ "data_type": "%s", "generator": "%s", "value":"' % ('text'
 ↪ , self.generator)
            postfix = '" }'

        for cnt, line in enumerate(fp):
            logline = line.replace("\n").replace("\r")
            if self.placeholder_ts != "":
                logline.replace(self.placeholder_ts, str(datetime.datetime.now()))
            self.xmpp.publish(self.node, '%s%s%s' % (prefix, logline, postfix))
            sleep(rate)

    logging.info("Replayed {} Lines of log in {}s".format(cnt+1, (cnt+1)*rate))
```

Listing 11.4: Logreplay most important source

Pmacct

```
nfacctd_port: 2055
nfacctd_templates_file: /var/cache/pmacct/netflow_templates.json
plugins: kafka[kafka], kafka[mac]

aggregate[kafka]: src_host,dst_host,src_port,dst_port,proto
kafka_output[kafka]: json
kafka_topic[kafka]: nf.queue
kafka_broker_host[kafka]: kafka
kafka_refresh_time[kafka]: 300
kafka_history[kafka]: 5m
kafka_history_roundoff[kafka]: m
nfacctd_stitching[kafka]: true

aggregate[mac]: src_mac
kafka_output[mac]: json
```

```
kafka_topic[mac]: mac.queue
kafka_broker_host[mac]: kafka
kafka_refresh_time[mac]: 300
kafka_history[mac]: 5m
kafka_history_roundoff[mac]: m
nfacctd_stitching[mac]: true
```

Listing 11.5: Pmacct config

Agents

MacMan

```
def handleMac(self, msg : json):
    mac = msg['mac_src']
    if mac not in self.db:
        logging.info('Unknown mac: %s', mac)
        self.db[mac] = {'first_seen': msg['stamp_updated'], 'last_seen': msg['
 ↪ stamp_updated']}
        self.sendAlert(msg, "danger", "Unknown Host")
    else:
        logging.info('Known mac: %s', mac)
        self.sendAlert(msg, "", "Known Host")
        self.db[mac]['last_seen'] = msg['stamp_updated']

    self.saveDb()
```

Listing 11.6: MacMan most important source

Simplenotifier

```
[...]
    self.add_event_handler('pubsub_publish', self.alertAdapter, threaded=True)

def alertAdapter(self, msg):
    data = msg['pubsub_event']['items']['item']['payload']
    self.callback(data.text)

def handleAlert(self, msg):
    logging.info("Received Msg: %s", msg)
```

Listing 11.7: SimpleNotifier most important source

Webnotifier

```python
from aiohttp import web

class WebNotifier:
    def __init__(self, host, port):
        self.host = host
        self.port = port

    async def push(self, message: dict, ws):
        await ws.send_str(json.dumps({"timestamp": str(datetime.datetime.now()), "
 ↪ payload": message}))

    async def notifier(self, ws, config):
        source = config['source']
        if source == 'kafka':
            bootstrap_servers = config['host']
            topic = config['topic']
            consumer = KafkaConsumer(bootstrap_servers=bootstrap_servers,
 ↪ auto_offset_reset='latest')

            consumer.subscribe(pattern=topic)

            async for message in AsyncIteratorExecutor(consumer):
                await self.push(json.loads(message.value), ws)

    async def wsHandler(self, request):
        print("HandleWS")
        ws = web.WebSocketResponse()
        await ws.prepare(request)
        async for msg in ws:
            if msg.type == aiohttp.WSMsgType.TEXT:
                config = json.loads(msg.data)
                print(config)
                try:
                    task = request.app.loop.create_task(self.notifier(ws, config))
                    await task
                finally:
                    task.cancel()
                    print("Done with client")

    def run(self):
        app = web.Application()
        app.router.add_get('/ws', self.wsHandler)
        app.router.add_get('/', lambda x: aiohttp.web.HTTPFound('/index.html'))
        PROJECT_ROOT = pathlib.Path(__file__).parent
        app.router.add_static('/', path=PROJECT_ROOT / 'static', name='static')
        web.run_app(app, host=self.host, port=self.port)
```

Listing 11.8: Webnotifier most important source

11.3 Messaging

Kafka

Installation

```
FROM java:8-jre

LABEL maintainer="janic.mikes@gmail.com"

ARG SCALA_VERSION
ARG KAFKA_VERSION

RUN  wget -q -O - http://mirror.switch.ch/mirror/apache/dist/kafka/$KAFKA_VERSION/kafka_$SCALA_VERSION-$KAFKA_VERSION.tgz | tar -xzf - -C /opt \
      && mv /opt/kafka_$SCALA_VERSION-$KAFKA_VERSION /opt/kafka

ENV PATH /opt/kafka/bin:$PATH

COPY docker-entrypoint.sh /docker-entrypoint.sh
ENTRYPOINT ["/docker-entrypoint.sh"]

VOLUME ["/opt/kafka/config"]

EXPOSE 9092

CMD  ["kafka-server-start.sh", "/opt/kafka/config/server.properties"]
```

Listing 11.9: Kafka Dockerfile

Konfiguration

Der Hostname, der im Listener verwendet wird, muss vom Client aufgelöst werden können.

```
listeners=PLAINTEXT://kafka:9092
```

Listing 11.10: Kafka Configuration server.properties

XMPP - Openfire

Installation

```
FROM java:8-jre

LABEL maintainer="janic.mikes@gmail.com"

ARG OPENFIRE_VERSION
ARG OPENFIRE_USER
ARG OPENFIRE_DATA_DIR
ARG OPENFIRE_CONF_DIR
ARG OPENFIRE_LOG_DIR

RUN wget "http://download.igniterealtime.org/openfire/openfire_${OPENFIRE_VERSION}
 ↪ _all.deb" -O /tmp/openfire_${OPENFIRE_VERSION}_all.deb \
 && dpkg -i /tmp/openfire_${OPENFIRE_VERSION}_all.deb \
 && mv /var/lib/openfire/plugins/admin /usr/share/openfire/plugin-admin \
 && rm -rf openfire_${OPENFIRE_VERSION}_all.deb

COPY docker-entrypoint.sh /docker-entrypoint.sh

# Volumes and Permissions
RUN chmod +x /docker-entrypoint.sh \
 && mkdir -p /etc/openfire \
 && chmod -R 0755 /etc/openfire

EXPOSE 3478/tcp 3479/tcp 5222/tcp 5223/tcp 5229/tcp 7070/tcp 7443/tcp 7777/tcp
 ↪ 9090/tcp 9091/tcp

VOLUME ["${OPENFIRE_DATA_DIR}"]

#USER ${OPENFIRE_USER}:${OPENFIRE_USER}

ENTRYPOINT ["/docker-entrypoint.sh"]
```

Listing 11.11: Openfire Dockerfile

Konfiguration

Die Konfiguration von Openfire konnten wir nicht automatisieren. Wir mussten bei der ersten Instanziierung manuell den Wizard durchgehen. Ausserdem benötigt XMPP eine lauffähige Namensauflösung.

11.4 Elastic Stack

Elastic Stack (zuvor auch ELK Stack genannt) steht für Elasticsearch, Logstash und Kibana. Dies ist ein Software Trio, welches von Elastic co. entwickelt wird. Es hilft dabei, Daten zu sammeln und wieder zu finden. Kibana ist ein Web GUI, welches die Interaktion mit Elasticsearch vereinfacht.

Queries

Statt über Kibana kann die Elasticsearch Datenbank auch direkt abgefragt werden. Hier ein Beispiel einer Query über alle Einträge von einem Gerät über die letzten 4 Stunden.

```
curl -XGET "http://elasticsearch:9200/logkafka-*/_search" -H 'Content-Type:
 ↪ application/json' -d'
{
  "query": {
    "bool": {
      "filter": [
        {
          "match_phrase": {
            "value.mac_src": "00:0c:29:ae:6f:06"
          }
        },
        {
          "range": {
            "@timestamp": {
              "gte": "now-4h",
              "lte": "now",
              "time_zone": "+01:00"
            }
          }
        }
      ]
    }
  }
}'
```

Listing 11.12: Beispielabfrage Elasticsearch

11.5 Docker Container

Sämtliche unsere Applikationen und Server laufen als eigene Container. Dies bietet die Möglichkeit, schnell und sauber mehrere Systeme zu starten und bei nicht gebrauch zu beenden. Die Konfiguration lässt sich dadurch unter Versionskontrolle bringen, was wiederrum einen Vorteil beim gemeinsamen Arbeiten bietet. Siehe auch Abbildung 11.2

Beispiel Dockerfile Pmacct

Für das Projekt Pmacct haben wir ein eigenes Dockerfile erstellt, welches zuerst die Build-Umgebung aufbaut und das Projekt kompiliert. Danach kopiert es die Build Artifacts in einen neuen Container, damit nur noch die notwendigen Binaries installiert sind. Für unsere eigenen Python Projekte haben wir auch solche Dockerfiles erstellt.

```
FROM alpine:latest

RUN apk add --update \
  build-base \
  librdkafka-dev \
  libpcap-dev \
  libtool \
  autoconf \
  automake \
  pkgconf

WORKDIR /root/build
COPY jansson/ /root/build/jansson/
RUN cd jansson && \
  autoreconf -i && \
  ./configure && \
  make && \
  make install && \
  cd ..

COPY pmacct/ /root/build/pmacct/

RUN cd pmacct && \
  ./autogen.sh && \
  ./configure \
    --prefix=/root/build/buildoutput \
    --enable-jansson \
    --enable-kafka && \
  make && \
  make install && \
  cd ..

# Smaller running enviorment
FROM alpine:latest
RUN apk add --update \
  libpcap \
```

```
  librdkafka \
  jansson

COPY --from=0 /root/build/buildoutput/ /usr/local/

RUN mkdir /var/cache/pmacct && \
  touch /var/cache/pmacct/netflow_templates.json && \
  chown -R nobody:nogroup /var/cache/pmacct

RUN mkdir /data && \
  chown -R nobody:nogroup /data
VOLUME ["/etc/pmacct", "/var/cache/pmacct", "/data"]

USER nobody:nogroup
EXPOSE 2055/udp

COPY docker-entrypoint.sh /docker-entrypoint.sh
ENTRYPOINT ["/docker-entrypoint.sh"]

CMD ["/usr/local/sbin/nfacctd", "-f", "/etc/pmacct/nfacctd.conf"]
```

Listing 11.13: Beispielabfrage Elasticsearch

11.6 LaTeXDokumentation

Als LaTeXUmgebung verwenden wir MIKTex [20]. Zum Builden erstellten wir Makefiles. Den Build der Dokumentation haben wir mittels Travis-CI [21] automatisiert. Die benötigten Packages, um das Dokument zu generieren, haben wir in ein Docker Image gepackt. Dies hat die Buildzeit annähernd halbiert.

Abbildung 11.2: Docker Container