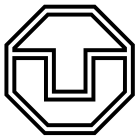


TECHNISCHE UNIVERSITÄT DRESDEN

Komplexpraktikum Rechnernetze: Flexibles Messsetup für Rechnernetze

Praktikumsbericht zum Modul
"Komplexpraktikum Rechnernetze"

vorgelegt von

Bengt Lennicke
Kevin Dominic Schubert
und
Janik Schönfelder

Professur für Rechnernetze
Fakultät Informatik
Bereich Ingenieurwissenschaften
Technische Universität Dresden
2022

Zusammenfassung

Ziel des Praktikums ist die Entwicklung eines Messsetups für typische Performanceparameter in Rechnernetzen. Das entwickelte Setup ist generisch und erweiterbar und ermöglicht ein einfaches Erfassen und Auswerten von Performanceparametern in Rechnernetzen. Dabei wird über eine Konfigurationsdatei die Messmethoden sowie zugehörige Argumente für jeden Knoten im Netzwerk verwaltet. Die ermittelten Messwerte werden in einer Prometheus-Datenbank gespeichert und mit Grafana visualisiert. Es sind drei verschiedene Messmethoden implementiert für die RAM-Auslastung, die *round trip time* zu einer beliebigen Addresse und Bandbreiten bei simuliertem Packetverlust. Das Praktikum umfasst des Weiteren diesen Bericht, welcher das Programm aus Entwicklersicht beschreibt und eine Nutzerdokumentation beinhaltet.

Contents

Chapter 1

Nutzerdokumentation

1.1 Voraussetzungen

Für eine problemlose Installation und Ausführung werden gewisse Vorkehrungen seitens des Nutzers benötigt. Um diese Voraussetzungen erfüllen zu können, werden Samples und Kurzanleitungen bereitgestellt.

1.1.1 Software

Das Messsystem benötigt grundlegende software seitige Vorraussetzungen. Einige davon werden durch das Programm selbst erfüllt (z.B. Aufsetzen von Grafana/Prometheus Containern), es gibt aber auch Bedingungen, die schon zum Starten des Programmes erfüllt werden müssen. Für jeden Rechner im Netzwerk wird als Betriebssystem Ubuntu Server LTS 20.04 und höher vorausgesetzt. Somit sind verwendete Programme wie ssh, python3 etc. eingeschlossen. Insbesondere wird also Linux vorrausgesetzt und die Shell "Bash" ist notwendig, um das Setup-Skript auszuführen.

1.1.2 Zugangsdaten

Das Messsetup wird von einem Node im Netzwerk ausgeführt/koordiniert. Für die Ausführung wird unter anderem Ansible Unterabschnitt 1.2.3 verwendet. Diese Software nutzt "ssh" bzw. das *secure shell protocol* und loggt sich mit gegebenen Nutzer ein, um verschiedene Aktionen auf den anderen Nodes im Netzwerk auszuführen. Für einen reibungsfreien Ablauf wird das public-key/private-key System von "ssh" verwendet. Entweder wird dafür ein neues Schlüsselpaar angelegt oder ein existierendes verwendet. Dafür bietet sich folgendes Command an:

```
ssh-keygen
```

Das Command erfragt interaktiv weitere Informationen. Hierbei ist lediglich der Pfad relevant, wenn das neue Schlüsselpaar nicht das Erste ist. Der öffentliche Schlüssel dieses Paars muss auf jedem Rechner im Netzwerk vorhanden seien. Mit folgendem Command lässt er sich an der richtigen Stelle hinterlegen:

```
ssh-copy-id <user>@<remote host>
```

Hierbei ist "user" der Nutzername, mit welchem Ansible im Programmlauf sich bei den Nodes anmeldet 1.2.3. An der Stelle von "remote host" gehört die Adresse des Nodes, bei welchem der Schlüssel hinterlassen werden soll. Sollten bei diesen Schritten noch Fragen offen bleiben sind die man pages der "ssh" Commands zu lesen.

1.2 Setup

1.2.1 Konfigurationsdatei

Das Messsetup und die Messmethoden benötigen Einstellungen bzw. Argumente. Diese Einstellungen sind zentral in einer Konfigurationsdatei mit dem Namen *setup.yml* festgelegt. Das verwendete Dateiformat ist *YAML*. Bei Änderungen ist die vorgegebene Syntax dementsprechend einzuhalten. Die Datei ist im Repository im Order "configs" zu finden.

Einstellungen des Messsetups

Die Ergebnisse der Messungen werden von einer Prometheus-Datenbank abgeholt und festgehalten und mit Grafana-Instanz visualisiert. Zusätzlich wird dafür ein Pushgateway- Server genutzt. Diese Services laufen in Docker Containern (näheres zum Setup unter Unterabschnitt 1.2.2). Die URLs für diese Services werden interaktiv bei der Verwendung des Setupskripts gesetzt und in der Konfigurationsdatei gespeichert.

- *gateway*: ist die URL des Pushgateway Nodes.
- *prometheus*: ist die URL des Prometheus Nodes.
- *grafana*: ist die URL des Grafana Nodes.

Argumente für die Messungen

Unter den Einstellungen des Messsetups sind die Argumente der einzelnen Messungen festgehalten. Die per Standardeinstellungen angegebenen Messmethoden umfassen *Ping*, *RAM* und *Iperf* (Näheres dazu unter Unterabschnitt 2.2.2).

Die Ping-Messung misst die Paketumlaufzeit (hier *round trip time*) und ermittelt den Durchschnitt davon. Dazu wird eine Zieladdresse unter dem Schlüsselwort "target" angegeben. Der "ping"-Command läuft ohne weitere Argumente endlos, daher ist auch noch die Anzahl der zu sendenden Pakete anzugeben mit dem Schlüssel "count".

Die RAM-Messung misst die relative Belastung des Arbeitsspeichers mittels des "free"-Commands und gibt diese in Prozent an die Datenbank weiter. Hierbei sind keine weiteren Argumente notwendig, daher ist der Bereich leer.

Die iperf-Messung besitzt einen komplexeren Aufbau. Hier wird ein künstlicher Paketverlust auf dem Router eingestellt und anschließend wird die Bandbreite zwischen zwei Netzwerkknoten gemessen. Hierbei wird die Routingtabelle der Netzwerkknoten unberührt gelassen, daher liegt es in der Verantwortung des Nutzers die korrekte Addresse des Routers anzugeben.

Sollten die beiden Netzwerkknoten nicht über den gegebenen Router kommunizieren, wird trotzdem versucht auf der gegeben Addresse den Paketverlust einzustellen. Bei Erfolg hat dieser Paketverlust dann aber keinerlei Einfluss auf die Messung. Ist die dort angebene Addresse für Ansible nicht erreichbar (z.B. fehlt der Router im InventoryAbschnitt 1.2.3) schlägt die Messung fehl. Die Routeraddresse Teil der Argumente: unter dem Namen "via" in der yml-Datei. Der Paketverlust am Router wird mit dem Argument "loss" in Prozent eingestellt. Die anderen Argumente sind für das Command "iperf". Das "target" ist das Ziel der Messung. Hier wird zuvor der iperf-Server gestartet. Dieser erwartet die Pakete am Port angegeben unter "port". Für ein genaueres Verständnis von "iperf" können Sie die man page lesen.

1.2.2 Docker

Docker ist eine Containervirtualisierungssoftware zur Isolierung von Anwendungen. Hierbei vereinfacht es die Bereitstellung von Anwendungen durch Containersysteme. Container gewährleisten eine einfache Trennung und Verwaltung von den restlichen Systemressourcen. Das Messsystem verwendet Docker zur Bereitstellung der nötigen Services Pushgateway, Prometheus und Grafana.

Pushgateway Das *Pushgateway* ist ein cachebasierter Webserver für das Sammeln von Metriken als batch jobs, die dann von Prometheus zyklisch abgefragt werden. Im Messsetup wird es als zentralisierte Sammelstelle für alle gemessenen Metriken genutzt.

Prometheus *Prometheus* ist eine freie Software für das Monitoring von verschiedenen Services. Es zeichnet Echtzeitmetriken von Services auf, die dann über HTTP abgefragt werden können.

Grafana *Grafana* ist eine Open-Source Webanwendung zur grafischen Darstellung von Daten. Hierbei wird Grafana in Verbindung mit Prometheus genutzt, um die gesammelten Metriken des Messprogramms in einem Dashboard grafisch zusammen zu fassen.

1.2.3 Ansible

Ansible ist ein Open-Source Tool, um Konfiguration, Administration und Orchestrierung von Computern zu automatisieren. Da das automatische Messsetup für eine beliebige Anzahl unabhängiger Systeme skalierbar sein soll, bietet sich die Verwendung einer solchen Automatisierungs-Software an. Ansible muss dabei nicht auf den Zielsystemen installiert werden. Die Automatisierung kann von einem einzigen Host-System angeleitet werden. Die Verwaltung der Zielsysteme erfolgt primär über SSH. Die zweite Voraussetzung für das Benutzen von Ansible ist eine vorinstallierte Python-Version.

Inventory Die Liste aller Zielsysteme wird in einem *Inventory* konfiguiert. Neben IP-Adressen / Hostnamen können auch für die Automatisierung interne Namen und Variablen für die Hosts vergeben werden. Die Hosts können gruppiert werden, um die nachfolgende Automatisierung zu vereinfachen. Das vorgefertigte Inventory ist in fünf Gruppen unterteilt. Es ist notwendig, dass der User die IP-Adressen auf seine eigenen Nodes anpasst. In Unterabschnitt 2.1.1 ist erklärt, welche Adresse an welcher Stelle einzutragen ist. Zudem muss der Nutzer den Nutzernamen, mit dem sich Ansible per SSH auf dem Zielsystem anmelden soll, mittels *ansible_user* angeben. Dieser muss mit dem User aus Unterabschnitt 1.1.2 übereinstimmen. Das Attribut *service_name* ist optional und dient nur der besseren Identifizierung.

Playbook In *Playbooks* werden die einzelnen Schritte der Automatisierung definiert. Es ist eine Sammlung aller Arbeitsschritte, die für die jeweilige Konfiguration eines bestimmten Szenarios notwendig sind. Ein einzelner Arbeitsschritt wird *Play* genannt. Ansible bietet hierbei für nahezu alle gängigen Schritte eigene Plugins an (z.B. Nutzung von python, pip, apt; setzen von Umgebungs- / Laufzeitvariablen). Falls dies nicht genügt, können beliebige Shell-Commands verwendet werden. Jedes Play kann einer Gruppe von Hosts zugeordnet werden. Für die Erweiterbarkeit muss der Nutzer eigene Plays bzw. Rollen schreiben und diese im Playbook einfügen.

Rollen Für eine bessere Übersichtlichkeit können verschiedene Aufgaben / Rollen auch direkt mittels strukturierten Playbooks abgebildet werden. Für jede Rolle wird ein eigenes Playbook geschrieben. Die Orchestrierung dieser Rollen wird anschließend durch ein übergeordnetes Playbook angeleitet. Für die Erweiterbarkeit muss der Nutzer eigene Rollen schreiben und diese im Root Playbook einfügen.

1.2.4 Grafana Dashboard

Um das Dashboard für Grafana einzurichten muss man im Webinterface von Grafana die im git befindliche json importieren. Dazu ruft man im Browser unter
http://GRAFANA_NODE:3000 das Grafana Webinterface auf. Standardlogin ist hierbei Username: *admin* und Passwort: *admin*. Um die json zu importieren nutzt man die Dashboard Importierfunktion, wählt dabei den json-Pfad und klickt auf "load". Danach wird man nach der Datasource gefragt. Hierbei wählt man die vorher eingerichte Prometheus Instanz als Source aus. Für weitere Hilfe beim Importieren kann man auch den Guide von Grafana nutzen.

1.3 Aufsetzen

Für das initiale Setup muss die *setup.sh* Datei ausgeführt werden. Dafür muss die Datei ggf. erst entsprechende Rechte erhalten:

```
chmode 777 setup.sh
```

Danach kann die Datei ausgeführt werden:

```
bash setup.sh
```

Hierdurch wird zuerst *pip* und *ansible* auf dem ausführenden System installiert. Im Anschluss sollen die Services *Grafana*, *Prometheus* und *Pushgateway* auf den vorkonfigurierten Adressen installiert werden. Dazu wird zunächst Docker installiert, falls auf dem Zielsystem noch nicht vorhanden. Im Anschluss wird das jeweilige Docker Image des zu installierenden Services heruntergeladen und unter Einbezug der Konfigurationsdateien eingerichtet. Nach diesen Schritten ist die Einrichtung der drei Services abgeschlossen. War die Einrichtung erfolgreich, wird das Ansible Playbook ausgeführt. Näheres dazu in Abschnitt 1.4.

1.4 Ausführung

Manuelle Ausführung

Die Ausführung der Messungen erfolgt mit dem Start des Ansible Playbooks. Um dies manuell zu starten, muss folgender Befehl ausgeführt werden:

```
ansible-playbook playbook.yml -i inventory.yml
   --private-key=ssh_key_path
```

Um den SSH-Schlüssel zu übergeben, welcher die SSH-Authentication mit den Zielsystemen vereinfachen soll, muss im Command *ssh_key_path* mit dem echten Pfad zum Schlüssel ersetzt werden. Nach Ausführung wird der aktuelle Stand des *script*-Ordners auf die *measure_nodes* kopiert und dort in einer virtuellen Python Umgebung installiert. Zuletzt wird jede im Playbook aufgeführte Messung ein Mal durchgeführt.

Ausführung zu einem bestimmtem Zeitpunkt

Um die Ausführung zu einem bestimmten Zeitpunkt zu starten, kann in Ubuntu vorinstallierte Paket "at" benutzt werden. Wenn die Messung in zwei Stunden ausgeführt werden soll, kann folgender Befehl genutzt werden:

```
at now + 2 hours -f $RUN_PLAYBOOK
```

Wenn die Messung um eine bestimmte Uhrzeit ausgeführt werden soll, z.B. um 0 Uhr, kann folgender Befehl genutzt werden:

```
$RUN_PLAYBOOK | at 00:00
```

$RUN_PLAYBOOK muss dabei mit dem ersten Befehl aus dem vorherigen Abschnitt ersetzt werden 1.4.

Regelmäßige Ausführung mittels "Cron"

Sollen die Messungen regelmäßig zu einem bestimmten Zeitpunkt ausgeführt werden, bietet sich die Einrichtung eines *crontabs* an, da Cron standardmäßig unter Ubuntu installiert ist. Um die Messung jeden Tag um 12 Uhr ausgeführt werden soll, kann folgender Befehl genutzt werden:

```
00 12 * * * $RUN_PLAYBOOK
```

$RUN_PLAYBOOK muss dabei mit dem ersten Befehl aus dem ersten Abschnitt ersetzt werden 1.4. Der Nutzer kann die Routine beliebig unter Einhaltung der Syntax für crontabs anpassen (`https://wiki.ubuntuusers.de/Cron/`).

1.5 Erweiterbarkeit

Das Messsetup bietet die Möglichkeit eigene Testmethoden hinzuzufügen. Dafür sind eine Ansible-Rolle und ein Python-Modul notwendig. Wie beides in Grundzügen aufgebaut werden muss ist im folgenden an einem simplen Beispiel gezeigt. Unter Kapitel 2 gibt es einen genaueren Einblick in die Struktur und Funktionsweise. Außerdem muss die Messung und zugehörige Argumente in der Konfigurationsdatei angegeben werden. Im Beispiel wollen wir eine Zufallszahl "messen". Diese Messung nennen wir "Random_Num".

Beispiel Konfiguration

Die Konfigurationsdatei *setup.yml* im Ordner "configs" beinhaltet bereits drei Messmethoden und deren Argumente (wenn nötig). An dieser Stelle sind neue Messmethoden einzufügen.

```
5           ...
6   modules:
7           Ping:
8                   target: 8.8.8.8
9                   count : 5
10          RAM:
11          Iperf:
12                  target: 172.18.0.6
13                  time : 5
14                  interval: 1
15                  port : 5001
16                  via: 172.18.0.8
17          Random_Num:
18                  lower_limit:  0
19                  upper_limit:  10
```

Unter dem Punkt 'modules' wurde die neue Messung Random_Num hinzugefügt (Zeile 17-19). Damit sind später die Argument für die Messung zu finden. Hier geben die Argumente lediglich vor in welchem Bereich die zufällige Zahl liegen soll.

Beispiel Modul

Das neue Python-Modul für die Messung wird bei den anderen Modulen erstellt also unter:

```
<user>@host:~/KP_Rechnernetze/script/src/modules$ touch RandomNum.py
```

Das Modul in unserem Beispiel heißt "RandomNum.py" und könnte wie folgt aussehen:

```
from random import randrange

class RandomMeasurement:
  def __init__(self, lower_limit, upper_limit):
    self.lower_limit=lower_limit
    self.upper_limit=upper_limit

    self._check()

  def _check(self):
    if self.lower_limit > self.upper_limit:
      raise Exception("Untere Intervalgrenze muss kleiner als die Obere sein.")

  def measure(self):
    return randrange(self.lower_limit, self.upper_limit)
```

Das Modul enthält eine Klasse mit drei Methoden. Der Name der Klasse ist nicht relevant für die Funktionsweise des Programms. Bei den Argumenten der "__init__"-Methode sind die Argumente aus der Konfigurationsdatei wieder zu finden. Hier ist es relevant, dass die gleichen Namen verwendet werden.

Die "_check"-Methode ist optional für die Messklassen, aber es empfiehlt sich vorher zu Überprüfen, ob die Messung fehlerfrei laufen kann. Im Fall des Beispiels ist es sinnvoll die Argumente auf Korrektheit zu überprüfen. Je komplexer die Messung ist, desto mehr Probleme könnten auftreten und sollten in dieser Methode vorgebeugt werden.

Die "measure"-Methode ist nicht optional (siehe Unterabschnitt 2.2.2)). Diese Funktion wird vom Programm aufgerufen für die Messung und es wird eine Zahl als Rückgabewert erwartet. Diese Zahl sollte das Ergebnis der Messung sein, da sie schlussendlich in der Prometheus Datenbank landet.

Das Beispiel ist der Übersichthalber simpel gehalten, aber solange die Klasse im Messmodul die korrekten Argumentnamen verwendet und eine "measure"-Methode hat, die das Ergebnis zurück gibt, kann in der Klasse alles python-mögliche gemacht werden. D.h. zum Beispiel können auch Kommandozeilen Werkzeuge ausgeführt werden mit der "subprocess" Bibliothek. Für das Parsen von Ergebnissen kann man z.B. regex verwenden ("re" Bibliothek).

Beispiel Rolle

Rollen werden für Ansible nach einem festen Schema erstellt und gespeichert. Das bedeutet für die neue Rolle muss im Repository unter "roles" ein neuer Ordner erstellt werden. Der Name dieses Ordners ist der Name der Rolle und dieser ist dann im Playbook zu verwenden. Dieser Ordner braucht einen "tasks" Unterordner, in welchem wir ein File *main.yml* erstellen.

```
<user>@host:~/KP_Rechnernetze/roles/<Neuen_Rolle>/tasks$ touch main.yml
```

Der Inhalt von *main.yml* könnte im einfachen Fall wie folgt aussehen.

```
- name: get variables from config
  set_fact:
    _modules: "{{ (lookup('template', './configs/setup.yml')|from_yaml).modules
        }}"
    _gateway_url: "{{ (lookup('template',
        './configs/setup.yml')|from_yaml).gateway }}"

- name: get the args for this module
  set_fact:
    _args: "{{ item.value }}"
  loop: "{{ lookup('dict', _modules) }}"
  when: "'Random_Num' in item.key"

- name: run measurement
  shell:
```

```
14      chdir: ~/measurement_script
15      cmd: . .venv/bin/activate && mess_programm {{ _gateway_url }} {{
            hostvars[inventory_hostname]['ansible_env'].SSH_CONNECTION.split(' ')[2]
            }} RandomNum "{{ _args }}"
16    register: mess_programm
17
18  - name: print measurement output
19    debug: var=mess_programm.stdout_lines
```

Diese YAML-Datei ist auf den ersten Blick sehr konfus, kann aber fast genauso immer wieder übernommen werden, wenn die Konfigurationsdatei und das Python-Modul korrekt aufgesetzt sind. Lediglich zwei Stellen sind für eine neue Messung stehts zu ändern: In Zeile 10 der Name der Messung wie er in der Konfigurationsdatei steht eingesetzt und in Zeile 15 für der Name des Python Moduls (*RandomNum.py*), welches zu dieser Messung gehört.

Die Rolle arbeitet 4 Schritte ab. Zu Beginn werden aus der Konfigurationsdatei *setup.yml* die Gateway IP Unterabschnitt 1.2.2 und die Argumente für alle Messmethoden geladen. Anschließend werden daraus die Argumente für die auszuführende Messung selektiert. Das Python Programm wird in einer Virtual Environment installiert, also wird es darin ausgeführt mit den ermittelten Argumenten. Der letzte Schritt zeigt lediglich den Standard Output des Python Programms in Ansible. Dieser Schritt ist optional.

Für node-übergreifende Messungen kann man an dieser Stelle mit den Werkzeugen von Ansible noch notwendige Einstellungen vor und nach der Messung tätigen. Ein Beispiel dafür ist in der Rolle für die iperf-Messung zu finden.

Chapter 2

Programmdetails

2.1 Ansible auf dem Master-Node

2.1.1 Inventar

Das Ansible Inventory des Messsetups wird in die Funktionen der Zielsysteme unterteilt:

- *measure_nodes*: Alle Nodes, die Messungen durchführen sollen
  - *ping_group*: Nodes, welche die Ping-Messung durchführen sollen
  - *ram_group*: Nodes, welche die RAM-Messung durchführen sollen
  - *iperf_group*: Nodes, welche die Iperf-Messung durchführen sollen
- *router*: IP-Adresse des Routers

2.1.2 Playbooks

Das Playbook besteht aus einer Reihe an Aufgaben, die auf unterschiedlichen Zielsystemen der Reihe nach ausgeführt werden:

1. Script Ordner auf *measure_nodes* kopieren und installieren
2. Ping-Einstellungen bei Nodes der *ping_group* setzen und Messung durchführen
3. RAM-Einstellungen bei Nodes der *ram_group* setzen und Messung durchführen
4. Iperf-Einstellungen bei Nodes der *iperf_group* setzen und Messung durchführen

2.1.3 Rollen

Es werden Rollen verwendet, um die einzelnen Tests zu strukturieren. Für einen besseren Einblick in Rollen im Zusammenhang mit Ansible lohnt sich ein Blick in deren Dokumentation.

Für jede Messung ist eine Rolle angelegt, sodass im Playbook Gruppen von oder einzelnen Netzwerkknoten diese zugeordnet werden können. Somit ist es einfach möglich zu strukturieren, welche Tests wo ausgeführt werden sollen.

In den einzelnen Rollen werden zu Beginn die notwendigen Informationen aus der Konfigurationsdatei geladen. Wenn notwendig werden noch weitere Umgebungsfaktoren aufgesetzt (z.B. bei der Iperf Messung der Server gestartet). Diese Änderungen werden im Anschluss an die Messung wieder rückgängig gemacht.

2.2 Das Python3 Programm

Ein Teil des Messsetups ist ein Python Package mit dem Namen "messprogramm". Dieses Package ist mit pip installierbar und beinhaltet daher eine *pyproject.toml*, eine *setup.py* und eine *setup.cfg* Datei. Die Wahl beim Package-Manager ist auf pip gefallen, weil es der Standard und häufig bereits installiert ist.

"messprogramm" erfüllt zwei Aufgaben. Die Erste ist das Ausführen der Messklassen. In diesen Klassen wird die Messung definiert und das Ergebnis geparst, sodass dieses von Prometheus angenommen werden kann. Die Zweite ist das Senden der ermittelten Daten an den pushgateway Dienst. Das Programm selbst besteht im Wesentlichen aus 3 Teilen:

- *Kern*: legt den primären Ablauf fest
- *Exporter*: bildet die Schnittstelle zum pushgateway
- *Messklassen*: definiert einzelne Messungen

Diese Aufteilung wurde gewählt, um eine Erweiterung für neue Messmethoden zu ermöglichen (mehr dazu unter Abschnitt 1.5). Der Kern des Programms ist in der Lage jede der Messmodule auszuführen, wenn der Name des Moduls mit einem Dictionary an Argumenten dem Python-Programm übergeben wird. Das bedeutet, wenn für eine neue Messung eine neue Messklasse geschrieben wird (entsprechend einiger Vorgaben 2.2.2), dann kann das Messprogramm diese neue Messung ausführen. Die Messklasse selbst ist notwendig, da eine Messmethode alles Mögliche sein kann. Es kann ein einfacher Aufruf eines Commands sein oder eine Kette von Befehlen. Außerdem ist der Output für jede Methode unterschiedlich und muss für die Datenbank auf das relevante Maß heruntergebrochen werden, d.h. geparst werden. Wie die Methode auszuführen ist und der Output zu interpretieren wird in den Messklassen festgelegt.

2.2.1 Core der Anwendung

Der Kern der Anwendung beinhaltet die "main"-Funktion, welche den Entry-Point für das Pythonprogramm darstellt. Daher werden hier die Argumente überprüft und das Dictionary mit den Argumenten für die Messmethode geladen. Anschließend wird ein Exporterobjekt initialisiert und die Messung gestarted. Hierbei wird die Messklasse über den gegebenen Namen mit den Argumenten aus dem Dictionary geladen und die "measure"-Methode der Messklasse wird ausgeführt. Der Rückgabewert wird anschließend an den Exporter übergeben.

2.2.2 Module der Messklassen

Die Messklassen dienen, wie zuvor beschrieben, als Wrapper für verschiedene Messmethoden. Damit diese reibungsfrei vom Kern des Pythonprogramms verwendet werden können, müssen die Messmodule nach bestimmten Bedingungen erstellt werden.

Zunächst wird das Modul beim Programmlauf über seinen Namen importiert. Bedeutet der Name des Moduls muss genau dem Namen für die Messung in der Konfigurationsdatei entsprechen. Wird beispielsweise in der Config unter "modules" das Modul "Ping" gelistet, dann muss der Name des zugehörigen Moduls *Ping.py* sein.

Desweiteren wird für die Messung die Messklasse aus dem Modul initialisiert. Genauer gesagt wird die erste Klasse, welche beim Importieren des Moduls gefunden wird, initialisiert. Eine Methode dieser Klassen muss "measure" sein. "measure" gibt das Ergebnis der Messung in Form einer Zahl zurück. Weitere Methoden sind optional für die korrekte Funktionsweise, allerdings empfehlt sich eine Methode, welche sicher stellt, dass die Bedingungen für eine Messung richtig sind. Diese Methode wird in den drei gegebenen Messmodulen "_check" genannt und wird beim Initialisieren ausgeführt.

Ping Modul

Die einzelnen Messmodule, die standardmäßig Teil des Programms sind, sind sich alle sehr ähnlich daher wird hier lediglich das Modul für eine Ping Messung vorgestellt.

Das Ping Modul bildet ein Wrapper für das Kommandozeilen Programm "ping". Das Diagnose-Tool kann verwendet werden, um die Erreichbarkeit eines bestimmten Hosts zu testen bzw. die *round trip time* zu diesem Host zu messen. Dieses Modul zielt auf Zweiteres ab.

Es exitiert genau eine Klasse im Modul mit dem Namen "PingMeasurement". Diese Klasse hat die 3 Methoden "__init__", "_check" und "measure". Für die Messung wird ein Ziel benötigt (im Code 'target') und die Anzahl an Paketen benötigt. Zweiteres ist notwendig, da "ping" ohne Nutzerinteraktion nicht terminiert. Beide Argumente werden auf standardmäßig beim Initialisieren der Klasse in der __init__ der Methode festgehalten, welche anschließend noch _check aufruft. Sollte hierbei keine Exception auftreten ist das Initialisieren der Klasse erfolgreich.

Um Fehler bei der Messung zu verhindern, wird in _check zunächst sichergestellt, dass "bash" und "ping" exitstieren, indem unter der Umgebungsvariable "$PATH" danach gesucht wird. Beides ist relevant dafür, dass der Befehl für die Messung überhaupt ausführbar ist. Anschließend wird auch die Messung einmal probeweise durchgeführt. Das kostet zwar Zeit, aber somit werden Probleme vor der echten Messung bemerkt.

Die Methode mit dem Namen "measure" wird im Programmablauf ausgeführt, um den gewünschten Wert (hier die *round trip time*) zu ermitteln. Dazu wird zunächst "ping" mit den gegebenen Argumenten ausgeführt und der Output zwischengespeichert. Prometheus arbeitet mit einzelnen Zahlenwerten, deshalb muss der Text des Outputs noch interpretiert werden. Der Text listet unter Anderem auf, wie lange jedes der Pakete, die gesendet worden, gebraucht hat. Daraus wird der Mittelwert gebildet, zum Einen um diese Vielzahl an Werten auf einen runter zu brechen, zum Anderen bekommen ergibt sich so ein aussagekräftigere Wert. Dieser Rückgabewert wird auf zwei Dezimalstellen gerundet, da weitere Stellen an Relevanz verlieren.

2.2.3 Exporter

Der Exporter sendet die gesammelten Daten der einzelne Module nach Messabschluss an das Pushgateway.

Es existiert eine Klasse mit dem Namen "Exporter". Diese Klasser besitzt die Methoden "__new__", "__init__", "pushToGate", "initializeMeasurement" und "setMeasurement".

Die Klasse ist im sogenannten Singleton-Pattern aufgebaut. Somit ist im Programmablauf gewährleistet, dass es zu jeder Zeit maximal ein Objekt dieser Klasse existiert. Hierbei wird in "__new__" bei Instanziierung des Objekts geprüft, ob eine Instanz dieser Klasse schon existiert. Wenn dies nicht der Fall ist, dann wird die sogenannte "cls.instance" mit dieser erstellen Instanz initialisiert. Der hier verwendete Python-Parameter "cls" besitzt als Attribut die "Klasse zu der diese Methode gehört" und ist gleichzusetzen mit dem verbreiteten "this"-Call. Ist das Exporter-Objekt instanziiert, so besitzt es 2 globale Variablen. Das "registry", was als "CollectorRegistry()" intialisiert wird, ist ein Collector der verwendeten "prometheus_client" Bibliothek. Dieser Collector enthält alle registrierten Metriken, die aus den Messklassen resultieren. Einfachheitshalber wird ebenso eine zweite globale Variable initialisiert. "measurements" ist ein Array, was im weiteren Programmablauf ebenso alle registrierten Metriken hält. Es existiert für das einfache Loopen der einzelnen Metriken an anderer Stelle im Exporter.

Die "__init__"-Methode, die während der Instanziierung des Exporter-Objekts aufgerufen wird, besitzt die Argumente "gateway_url" (IP des Pushgateways als String), "instance" (IP des Nodes als String), "module" (Name des Messmoduls als String) und "args" (weitere freie Argumente für einzelne Ausführung in einer Shell). Alle Argumente werden beim Initialisierung vom Kern übergeben.

Bevor eine Messklasse ausgeführt wird, wird das Exporter-Objekt dafür vorbereitet. Das bedeutet die Art der Messung sowie die Argumente werden übergeben. Dazu wird in der "run_measurement_module" Funktion des Kerns die Exporter Methode "initializeMeasurement" aufgerufen. Hierbei werden "title" (Name des Messmoduls), "desc" (Beschreibung der Messung) und "labels" (Modulname und Instanz als Labelkeys) als Argumente übergeben. In dieser Methode wird darauf eine Gauge-Metrik mit den übergebenen Argumenten initialisiert, im "registry" registriert und in "measurements" angehängt.

Ist eine Messung durch eine Messklasse fertig, so wird "setMeasurement" aufgerufen. Hier werden als Argumente "title" (Name des Messmoduls), "value" (Ergebnis der Messung) und "labelvalues" (Erwartete Werte für die Labelkeys) als Argumente übergeben. Daraufhin looped das Programm durch die globale Variable "measurements" und vergleicht die darin enthaltenen Metriken mit dem übergebenen "title". Kommt es zu einer Übereinstimmung, so ruft diese Metrik ihre Methode "labels" auf und initialisiert "module" und "instance", welche vorher als Keys gesetzt wurden, mit ihren Werten. "instance" ist in diesem Zusammen-

hang die IP des Nodes auf dem die Messung läuft. Danach ruft die Metrik ihre "set"-Methode auf und setzt schließlich den übergebenen Wert "value" aus der eigentlichen Messung. Zum Schluss ruft dann das Exporter-Objekt durch "self" die Methode "pushToGate" auf.

Die Methode "pushToGate" ist die Methode, die sich mit dem eigentlichen Export der Daten zum Pushgateway beschäftigt. Am Anfang der Methode wird der sogenannte "grouping key" gesetzt. Dies ist ein Dictionary, was das derzeitige Modul und die derzeitige Instanz beinhaltet. Grouping keys sind entscheident dafür, dass das Pushgateway Messungen von verschiedenen Nodes nicht miteinander überschreibt, sowie auch nicht verschiedene Messungen des selben Nodes . Die Methode "push_to_gate", eine Methode des prometheus_client, übernimmt dann die Verbindung zum Pushgateway, indem es die Gateway-URL, den einzuordnenden Job für Prometheus, die Grouping Keys und das Registry mit den enthaltenden Metriken übergeben bekommt. Diese Methode baut dann schließlich eine http-Verbindung zum Pushgateway auf und übergibt schlussendlich die Metriken.

Für ein besseres Verständis vom Exportieren zum Pushgateway empfiehlt sich die Dokumentation des Prometheus Clients für Python.

Chapter 3

Testsetup

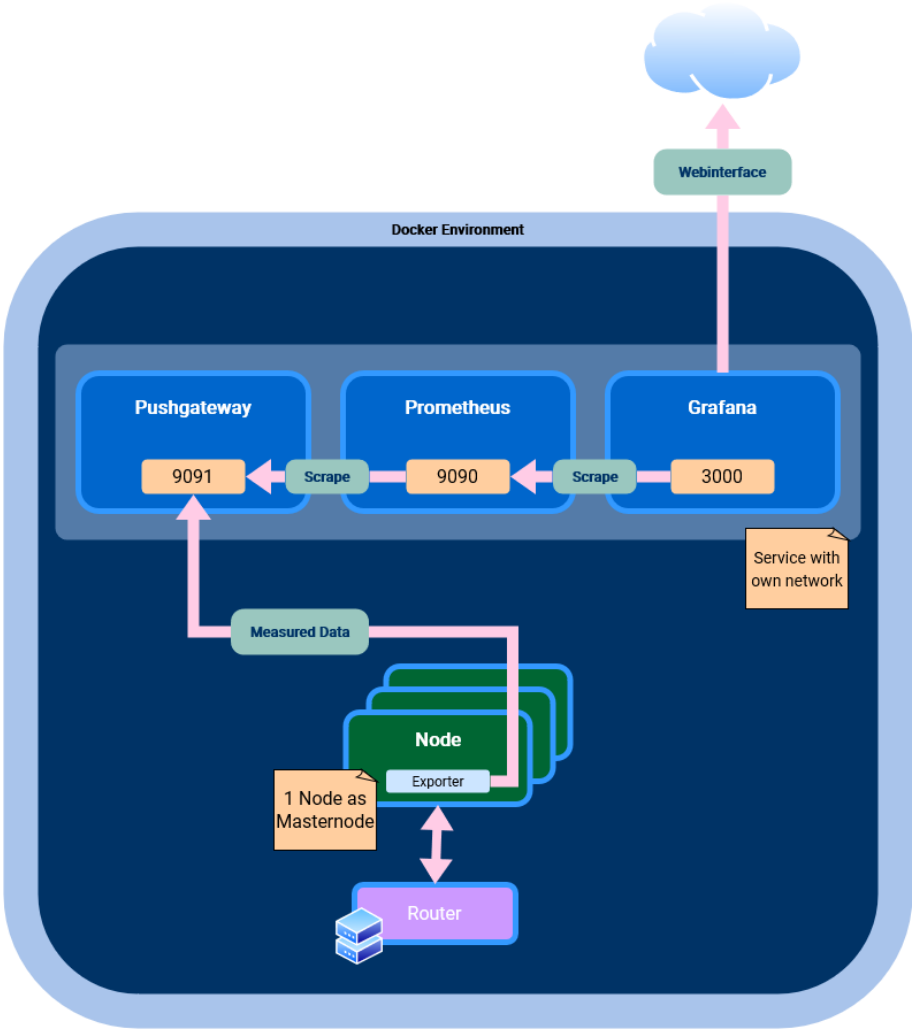
3.1 Tatsächliche Implementierung

Das gesamte Testsetup läuft auf einem *Raspberry Pi 4* mit dem Raspberry Pi OS, einem Port von Debian Bullseye. Darauf kam die Virtualisierungssoftware *Docker* (siehe Abschnitt 1.2.2) zum Einsatz. Innerhalb Dockers wurden die Container eingeteilt zwischen Services (*Prometheus*, *Grafana*, *etc.*), Hilfssoftware (z.B. ein Routingcontainer für den IPerf Test) und die eigentlichen Nodes, auf denen die Messanwendung läuft. Eines der Messprogrammnodes wird manuell als Masternode ausgewählt und deployed mit Hilfe von der Software *Ansible* alle nötigen Nodes und Systeme.

Die Servicecontainer nutzen untereinander ein geteiltes virtuelles Netzwerk im bridge-Modus um miteinander Daten austauschen zu können. Hierbei wurde jedem Service ein Port zugewiesen, wie z.B. *Prometheus*, der Port 9090 besitzt.

Die einzelnen Nodes kommunizieren untereinander über ein Router-Container. Um zu vermeiden, dass eine echte Routing-Software verwendet werden muss, sind die Routingtabellen der Nodes so angepasst, dass Traffic zu anderen Nodes im Netzwerk über den Router-Container fließt. Der Router-Container dient zur Simulation eines echten Routers um z.B. "iperf"-Tests durchzuführen. Im bridge-Mode würden die Nodes direkt und ohne Router kommunizieren, sodass man einen Zwischennode nutzt, um Parameter wie Package-Loss simulieren zu können.

Als Base-Image für die Nodes haben wir das verfügbare Ubuntu-Image aus dem Docker Hub genutzt. Schwierigkeit hierbei war, dass es sich um eine Baseversion des OS handelt und man nahezu alle wichtigen Services, wie *SSH*, *git*, *IPerf* oder auch *Ping* händisch installieren musste. Waren alle, für uns relevanten, Programm abgedeckt, haben wir aus dieser Runtime ein eingerichtetes Image generiert, was als Basis für weitere Container-Deployments diente.



Setup Architecture

3.2 Probleme beim Aufsetzen

In der Anfangsphase ist recht schnell klar geworden, dass die Realisierung des Setups nicht alleinig mit Prometheus und Grafana möglich ist, da die Aufgabenstellung für das Messprogramm keinen cachebasierten HTTP-Server vorgesehen hat, der die gemessten Werte zwischenspeichert und für den Scrape-Prozess von Prometheus bereitstellt. Als Lösung hierbei konnten wir schnell einen weiteren Service von Prometheus namens "Pushgateway" ausmachen und setzen somit voraus, dass dieser, extra für den Einsatz von Prometheus geschriebener, HTTP-Server bereits vorhanden ist oder mitinstalliert werden muss.

Durch das Bereitstellen von verschiedenen Configs für die einzelnen Services war die Distribution der einzelnen Konfigurationsdateien ebenso ein Problem. Anfangs wussten wir nicht genau wie wir diese (halb)-dynamisch (viele Werte werden bei Ausführung von der setup.sh erst eingetragen) generierten Dateien auf die verschiedenen Nodes für die Docker-Services kopieren und einbinden können. Schließlich wählten wir den Ansatz, dass jeder Node über ein ssh-Zugang verfügen muss, über den dann per "scp"-Command die Configs kopiert werden. Da Docker Container virtualisiert sind und sich somit vom restlichen System abgrenzen, muss man bei Installation des Containers per Argument manuell die Konfigurationsdatei des Hosts per Hardlink mit der Datei innerhalb des Containers mounten. Dazu nutzt man das "mount"-Argument wie folgt

```
docker run ... --mount type=bind,source=/PFAD/ZUR/HOST/CONFIG.yml,target=/PFAD/ZUR/CONTAINER/CONFIG.yml ...
```

Im weiteren Verlauf der Entwicklung widmeten wir uns der iperf-Klasse. Die Python-Nodes sind im

Docker Environment über den sogenannten Bride-Network miteinander verbunden. Bridge-Networks sind aber so konzipiert, dass sie ohne ein Routing auskommen und Nodes in solchen Netzwerken direkt miteinander kommunzieren können. Da wir aber mittels iperf auch Packageloss simulieren wollen, war dieser Netzwerkansatz problematisch. Somit haben wir uns für einen weiteren Node entschieden, der die Funktion eines Routers übernehmen soll. Dieser Router dient somit als "Vermittler" zwischen 2 Nodes, wenn ein iperf-Test ausgeführt werden. Realisiert wird dies durch das Anlegen und Verändern von Routing-Tabellen auf den Mess-Nodes. Andere Ansätze, wie das Nutzen des Routers an dem der Raspberry Pi angeschlossen ist, hätten zwar auch funktioniert, aber wir haben uns für die simple und elegantere Variante entschieden.

Chapter 4

Fazit

Das Praktikums Ziel ist erreicht worden. Durch das Zusammenarbeiten von den Ansible und einem Python Programm ist ein Messsetup erstellt worden. Das Setup ist in der Lage einen lokalen Parameter (RAM Auslastung) und zwei Netzwerkparameter (*round trip time* und Bandbreite mit Packetverlust) zu messen und in einer Prometheus Datenbank festzuhalten. Die gesammelten Information werden mit einem Grafana Dashboard visualisiert.
Ein erwartete Komponente ist die Erweiterbarkeit des Setups um weitere Messmethoden. Auch diese Anforderung ist erfüllt worden, allerdings ist dafür genaues Lesen der Nutzerdokumentation notwendig und für komplexere Netzwerktests auch ein gutes Verständnis von Ansible. Ansible selbst bietet weitreichende Möglichkeiten an und mit einem sehr guten Verständnis kann man daraus alleine ein Messsetup designen. Dadurch würde das Pythonprogramm obsolet und für eine Erweiterung wäre nur eine neue Rolle und die Anpassung in der Konfigurationsdatei notwendig.
Das Nutzen von Prometheus hat sich zu Anfang als ungünstig rausgestellt, da der Service zum Überwachung von Metriken gedacht ist. Das bedeudet Prometheus ruft die Daten selbst ab, statt sie z.B. durch ein PUT oder ähnliches gesendet zu bekommen. Dieses Problem wurde gelöst durch einen weiteren Service, das Pushgateway. Diesem Docker Service werden die Daten gesendet und Prometheus holt diese von dort regelmäßig ab. Dennoch bleibt das Problem, dass die Werte im Pushgateway nur nach einer neuen Messung verändert werden, aber Prometheus trotzdem regelmäßig die Werte abgreift. Somit enthält die Datenbank mehrfach die gleichen Werte von der gleichen Messung mit unterschiedlichen Zeitstempeln. Diese sind nicht der Zeitpunkt der Messung sondern der Zeitpunkt zu dem Prometheus die Daten abgerufen hat. Dieser Sachverhalt spiegelt sich auch in Grafana wieder. Hier wird in Diagrammen stehts dieser Zeitstempel von Prometheus verwendet. Daher entstehen, wenn nicht häufig genug gemessen wird, horizontale Linien. Diese suggerieren irrtümlich konstante Werte über die Zeit. Würden wir nochmal von vorne Anfangen würde es sich lohnen auch alternative Datenbank-Systeme in Betracht zu ziehen.
Für das Messsetup sollten drei Messmethoden bereitgestellt werden. Diese sollten unterschiedlich viele Nodes im Netzwerk betreffen. Die RAM-Messung betrifft nur eine Node, "Ping" ist in der Lage die RTT zwischen zwei Nodes zu bestimmen und die "iperf"-Messung arbeitet mit drei Nodes. Die RAM- und "Ping"-Messung sind mit der von uns gewählten Struktur sehr einfach umzusetzen gewesen. Für die "iperf"-Messung sind zusätzlich zum Python-Messprogramm noch Einstellungen über Ansible notwendig. Diese Einstellungen müssen korrekt funktionieren, da die Ansible-Rolle sonst mit einem Fehler beendet. Error-Handling erwies sich als schwierig, da z.B. die Ansible-Tools "ignore_error" oder "ignore_unreachable" für das Einstellen des Paketverlustes nützlich gewesen wären, aber nicht funktionieren. Somit wäre eine Bandbreiten Messung auch möglich gewesen ohne einen Router anzugeben, das ist jetzt nicht optional.