



universität
wien

MESSUNGEN UND VERGLEICH VON
VIRTUALISIERUNGSUMGEBUNGEN
ENTWICKLUNG EINES TESTFRAMEWORKS

eingereicht von
Christoph Steindl (0706052)

BACHELORARBEIT

Studienrichtung: Medieninformatik
Fakultät für Informatik der Universität Wien

Betreuer:
Prof. Dr. K. Tutschku
Bakk. (FH) A. Rafetseder, MSc

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht.

Wien, 27. Oktober 2011

Christoph Steindl

Zusammenfassung

Im Zuge dieser Arbeit wurde ein Framework implementiert, mit dem man Aussagen über die Fairness eines VMM (Virtual Machine Monitor) machen kann. Um dies zu erreichen werden in mehreren virtuellen Maschinen Belastungstests synchron durchgeführt und die Laufzeiten für ebendiese geloggt. Anhand dieser Zeiten wird versucht herauszufinden, ob gewisse virtuelle Maschinen bevorzugt werden, oder nicht und daher der VMM, beziehungsweise dessen Scheduling-Mechanismus, unfair arbeitet.

Abstract

Alleine das immer häufigere Auftreten des Begriffes „Cloud Computing“ in Zeitungen, Magazinen und vergleichbaren Medien zeigt, dass der Begriff „Virtualisierung“ immer präsenter wird. Diese so genannte „Cloud“ stellt jedermann virtuelle Ressourcen zur Verfügung, die er selbst gar nicht besitzt, aber trotzdem verwenden kann. Doch nicht nur in der „Cloud“ wird virtuell gearbeitet, denn der Schritt der Virtualisierung beginnt schon viel früher, nämlich zum Beispiel bei seiner eigenen Hardware. Man ist eigentlich fast tagtäglich damit konfrontiert ohne es zu merken. Doch nur weil etwas virtuell ist, heißt dies noch lange nicht, dass es keine Güte besitzt und die Qualitäten der einzelnen virtuellen Services nicht variieren können. Dies ist der Punkt, wo diese Bachelorarbeit ansetzt. Im Zuge dieser Arbeit wurde ein Framework erstellt, mit dem man Aussagen über Virtuelle Maschinen, beziehungsweise über die Monitore, welche die Virtuellen Maschinen steuern, machen kann. Dieses Framework soll zeigen, ob diese Monitore immer fair arbeiten und somit die einzelnen Virtuellen Maschinen gleichgestellt sind.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Aufgabenstellung	1
2. Virtuelle Maschinen	2
2.1. Virtualisierung	2
2.1.1. Vorteile Virtualisierung/Wieso VM	2
2.1.2. Scheduling	3
2.2. Aufbau einer Virtuellen Maschine	4
3. Architektur	5
3.1. Design	5
3.2. Umsetzung	6
3.2.1. Sequenzdiagramm	6
3.2.2. Aufbau der Architektur	7
3.2.3. Klassen-, Datei-, Skriptbeschreibung	8
3.2.4. Tests	12
4. Testfälle	14
4.1. Allgemeine Beschreibung	14
4.2. Ergebnisse	14
4.2.1. TCP	14
4.2.2. Mattest	14
4.2.3. IO	15
4.2.4. SeekAndWrite	15
4.2.5. Plots	16
5. Zukunftspläne	17
5.1. Dokumentation	17
5.2. Messkampagnen und Messstrategien	18
5.3. Ansteuerung des Hypervisors	19
5.4. Weitere Tests	19
5.5. Vergleich nativer und virtualisierter Tests	20
6. Schluss	21
A. Lessons learned	22
A.1. Portierung	22
A.2. Expect the unexpected	23
A.3. Synchronizität	24
B. Eigene Testklasse erstellen	24

Tabellenverzeichnis

1.	Liste unterschiedlicher Hypervisor	2
2.	Konfiguration des TCP-Tests	16
3.	Konfiguration des Mattests	17
4.	Konfiguration des IO-Tests	18
5.	Konfiguration des SeekAndWrite-Tests	19

Abbildungsverzeichnis

1.	[1] bringt diese schematische Darstellung des Xen Hypervisors	5
2.	Die Abläufe zwischen der Klasse <i>Host</i> und einer Klasse <i>Client</i> dargestellt in einem Sequenzdiagramm	7
3.	Komponentendiagramm des Frameworks mit zwei virtuellen Maschinen	8
4.	Scatterplot eines TCP-Tests	16
5.	Histogramm eines TCP-Tests	17
6.	Scatterplot eines Mattests	18
7.	Histogramm eines Mattests	19
8.	Scatterplot eines IO-Tests (256MB Hauptspeicher)	20
9.	Scatterplot eines IO-Tests (128MB Hauptspeicher)	21
10.	Scatterplot eines SeekAndWrite-Tests	22
11.	Summation der Laufzeiten eines SeekAndWrite-Tests	23

1. Einleitung

1.1. Aufgabenstellung

Das Thema dieser Arbeit „Messungen und Vergleich von Virtualisierungsumgebungen“ beinhaltet schon in seinem Titel eine der zentralen Komponenten dieser Arbeit, nämlich den Begriff der „Virtualisierung“. In größeren Firmen ist es schon gängige Praxis nicht mehr für jeden Mitarbeiter einen eigenen vollwertigen PC zu kaufen, sondern man steigt um auf so genannte „Slim Clients“. Diese Clients dienen als Schnittstelle zwischen dem Mitarbeiter der Firma und der virtuellen Instanz eines Betriebssystems, welches auf einem Server läuft. Da man einen geringeren finanziellen Aufwand mit virtuellen Umgebungen, Instanzen und Systemen hat, ist diese Praxis auch im Privatbereich verbreitet. Doch muss natürlich auch hier garantiert werden, dass die Virtualisierung von beliebigen Komponenten fair abläuft, also dass es keine Bevorzugung von einzelnen Komponenten gegenüber anderen gibt. Auch wenn man diese Fairness natürlich auch auf einer sehr Hardware-Nahen Ebene testen könnte, wird in der folgenden Arbeit vor allem auf virtuelle Maschinen, also Betriebssysteme, welche auf virtuellen Hardware-Komponenten basieren, eingegangen. Um diese Fairness gewährleisten zu können, gilt es ein Framework zu erstellen, mit dem Aussagen über die Gleichberechtigung einzelner virtueller Maschinen gemacht werden können. Neben der Implementierung des Frameworks ist auch das Ausführen von Tests und deren Analyse Teil dieser Arbeit. Zusätzliche Anforderungen sind eine möglichst hohe Portabilität zu erreichen, sodass das Framework möglichst auf allen Systemen eingesetzt werden kann und den Aufwand des Testers/Users, der die Tests durchführen will zu minimieren. Es soll also darauf hinauslaufen, dass möglichst große Teile des Frameworks automatisch ablaufen können.

Hypervisor	zugehörige Firmen
Oracle VM VirtualBox [3]	Oracle
Windows Virtual PC [4]	Microsoft
Xen Hypervisor [5]	Citrix Systems, Inc.
VMware [6]	VMware, Inc.

Tabelle 1: Liste unterschiedlicher Hypervisor

2. Virtuelle Maschinen

Dass das Erstellen von virtuellen Maschinen nicht nur ein kurzfristiges Phänomen ist, zeigt wahrscheinlich am allerdeutlichsten, dass große IT-Firmen jeweils ihre eigenen Programme zur Erstellung ebendieser implementiert haben, beziehungsweise jeweils eigene Hypervisor zur Verfügung stellen. Aber auch die Open-Source-Community hat mit dem Projekt Xen Hypervisor das Thema der Virtualisierung aufgegriffen, welches unter der GPL-Lizenz frei verfügbar ist. Eine Auflistung einiger Virtualisierungsprojekte von unterschiedlichen Firmen ist in Tabelle 1 zu finden. Aber auch ein Großunternehmen wie IBM setzt, auch wenn nicht eigens eine Software implementiert auf Virtualisierung, wie die enge Integration anderer Virtualisierungsumgebungen in [2].

2.1. Virtualisierung

2.1.1. Vorteile Virtualisierung/Wieso VM

Nun stellt sich die Frage, wieso man diesen Schritt zur Virtualisierung von Hardware überhaupt machen soll? Diese Frage wird vor allem in [7] sehr gut beantwortet. Er zählt gleich mehrere Gründe auf:

- Security: Durch die Abkapslung der einzelnen virtuellen Komponenten von der echten Hardware und die Abkapslung zwischen den einzelnen Betriebssystemen, fällt ein Angriff auf die Sicherheit eines Systems nicht sehr gravierend aus. Der Angriff kann immer nur innerhalb eines Systems gelingen und beeinflusst im Normalfall die anderen Systeme nicht.
- Reliability and availability: Kommt es zu einem Software-Fehler innerhalb einer VM so ist das System rundherum nicht betroffen.
- Cost: Wie bereits erwähnt kommt mit Virtualisierung meist auch eine Ersparnis an Kosten, durch geringeren Anschaffungsaufwand vor allem in größeren Unternehmen.
- Adaptability to Workload Variations: Braucht ein Benutzer aus einem gewissen Grund kurzfristig mehr Ressourcen, so können ihm diese ohne Hardware

zu verändern zur Verfügung gestellt werden. Voraussetzung für dieses Szenario ist natürlich, dass die Anforderungen des Benutzers nicht über denen der Hardware liegen.

- **Legacy Applications:** Durch Generieren einer VM ist es immer möglich das richtige Betriebssystem für diverse Applikationen zu wählen. Dies bedeutet, dass trotz einer möglichen Portierung auf ein neues System die Applikation nach wie vor in der virtuellen Maschine ausgeführt werden kann.

2.1.2. Scheduling

Jeder Hypervisor einer virtuellen Maschine muss ein Scheduling betreiben, mit dem er Ressourcen den einzelnen virtuellen Maschinen zuweisen kann. Dieses Scheduling-Verfahren kann in der Norm zweierlei Qualitätskriterien erfüllen:

- **Maximale Auslastung:** Oberstes Ziel ist es, dass soviel Durchsatz auf den einzelnen Hardwarekomponenten wie möglich erzeugt wird. Dies bedeutet, dass beispielsweise die CPU immer ausgelastet sein sollte, auch wenn nicht immer alle Instanzen einer virtuellen Maschine gleich behandelt werden.
- **Maximale Fairness:** Hier sollen alle Instanzen gleich behandelt werden. Es kann in diesem Fall auch dazu kommen, dass gewisse Komponenten Leerlaufzeiten haben, obwohl eigentlich Aufträge für diese bereit wären.

Auch wenn hier die Rede von virtuellen Maschinen ist, so kann man obige Überlegung auf jederlei Schedulingmechanismus übertragen werden.

Xen-Scheduling Wie bereits erwähnt gibt es vor allem proprietäre Virtualisierungsprodukte auf dem Markt. Aus diesem Grund ist es nicht so einfach, dass man Informationen über das Scheduling erhält. Allerdings hat man beim Open-Source-Projekt des Xen Hypervisors mehr Chancen. Beim Scheduling-Verfahren von Xen handelt sich um ein so genanntes „Credit-Scheduling“. Dies bedeutet, dass jede virtuelle CPU eine gewisse Zeit hat, um ihren Task ausführen zu können. Diese Zeitslots werden in regelmäßigen Abständen neu vergeben und dann ist es die Aufgabe des Schedulers zu schauen, in welchem Status sich eine virtuelle CPU befindet:

- $credit > 0$: Es sind noch Zeitressourcen übrig und der Zustand ist UNDER.
- $credit \leq 0$: Alle Zeitressourcen wurden aufgebraucht und der Zustand ist OVER.

Nun werden vom Scheduler diese Prozesse von folgenden virtuellen CPUs zuerst gewählt, deren Zustände auf UNDER sind, welche von den Prozessen gefolgt werden, wo der Zustand OVER ist. In Sonderfällen können die virtuellen CPUs auch auf den Zustand BOOST gesetzt werden, womit sie anderen vorgezogen werden.

2.2. Aufbau einer Virtuellen Maschine

Um eine virtuelle Maschine einrichten zu können, bedarf es mehrerer Komponenten:

- Hardware: Auf den Hardwarekomponenten werden alle Vorgänge, welche in einer Virtuellen Maschine vorgehen, tatsächlich ausgeführt.
- VMM (Virtual Machine Monitor)/Hypervisor: Der VMM regelt die Kommunikation zwischen der Hardware und den virtuellen Maschinen. Der VMM legt für die einzelnen virtuellen Maschinen virtuelle Ressourcen an, welche diese dann verwenden können. Es wird zwischen zwei Typen an VMM unterschieden[2]:
 - Type 1: Der VMM wird direkt auf der Hardware betrieben.
 - Type 2: Der VMM wird innerhalb des existierenden Betriebssystems installiert. Dieses funktioniert weiterhin ohne Einschränkungen und die Kommunikation mit der Hardware wird über dieses so genannte Host-Betriebssystem geregelt.
 - Hybrid: [8] und [9] erweitern diese Typen und sprechen von einem Hybriden-Ansatz, nur aus einem kleinen Hypervisor besteht, welcher nur CPU und Speicher verwaltet, während I/O von den nativen Treibern eines Service-Betriebssystems geregelt werden.
- VM (Virtual Machine): Die VM besteht aus den Komponenten, welche von dem VMM für sie angelegt wurden. Diese Komponenten können ganz unterschiedliche Spezifikationen, als die der „echten“ Hardware haben, indem sie vom VMM emuliert werden.
- Guest-Betriebssystem: Innerhalb der VM muss wiederum zur Kommunikation mit der virtuellen Hardware ein weiteres Betriebssystem installiert werden. Dieses kann beliebig gewählt werden, da es vom Host-Betriebssystem vollkommen abgekapselt ist.

Zur Veranschaulichung wird in Abbildung 1 der Hypervisor der Xen Virtualisierungsumgebung dargestellt. An der links oben abgekoppelten „Xen Control Software“ die über die Driver-Domain mit dem Hypervisor verbunden ist, kann man erkennen, dass es sich hier um einen Type 1-VMM handelt. Ebenfalls gut erkenntlich ist, dass jede Hardwareressource abgekoppelt von der virtualisierten Hardware arbeitet. Ebenfalls sind drei virtuelle Maschinen abgebildet, welche nur über den Hypervisor mit der Hardware kommunizieren können.

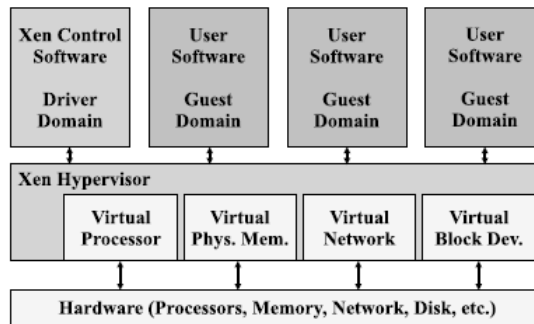


Abbildung 1: [1] bringt diese schematische Darstellung des Xen Hypervisors

3. Architektur

3.1. Design

Ein solches Projekt beginnt man klein und versucht es dann nach außen hin zu vergrößern. So war es auch hier und so mussten zuerst Überlegungen über den Aufbau angestellt werden. Es gibt hier zweierlei Herangehensweisen, um zu den gewünschten Ergebnissen zu kommen:

- **Sequentiell:** Es ist möglich einen Test für eine virtuelle Maschine zu erstellen. Dieser wird auf ebendieser ausgeführt und durch eine Vielzahl an Tests mit unterschiedlichen Konfigurationen kann man einen Schluss über das Gesamtsystem ziehen. Hier ist wichtig, dass die Tests nacheinander ausgeführt werden, damit sie sich untereinander nicht beeinträchtigen.
- **Synchron:** Es laufen mehrere virtuelle Maschinen, auf welchen jeweils der selbe Test ausgeführt wird. Hier stehen die VMs kompetativ zueinander und man kann direkt ablesen, welche VM in einem gewissen Zeitabschnitt eine bessere Performance gezeigt hat. Die Synchronität des Systems gelingt natürlich immer nur innerhalb gewisser (Rand-)Bedingungen, welche physikalischer Natur sind, weshalb es schwieriger ist, ein synchrones System zu implementieren.

Die Entscheidung wurde für System, das zumeist im synchronen Betrieb laufen soll, gefällt, trotzdem wird der sequentielle Betrieb aber unterstützt. Diese Entscheidung ist allerdings gleichzeitig ein Indikator für die Architektur des Frameworks. Muss man Prozesse synchron halten, so empfiehlt es sich eine Instanz zu implementieren, welche als Monitor fungiert. Diese Instanz hat im Idealfall mit den einzelnen virtuellen Maschinen nichts zu tun und kümmert sich nur darum, wann welche Kommandos für welche Prozesse an die virtuellen Maschinen erteilt werden. Da der Monitor unabhängig sein soll, hat sich auch der Platz, wo dieser ausgeführt wird empfohlen. Man kann diesen einfach im Host-Betriebssystem ausführen und

jede einzelne virtuelle Maschine verbindet sich mit diesem. Man kann also sagen, dass das Framework eine Client-Server-Architektur beinhaltet.

Nachdem diese Entscheidung gefallen war, standen als nächstes einige Überlegungen zur Kommunikation zwischen dem Client und dem Server an. Es stellte sich die Frage, wie groß der Kommunikationsaufwand ist und wie dieser bewältigt werden sollte. Da sich der Aufwand wahrscheinlich in Grenzen hält, fiel die Entscheidung auf eine Kommunikation mit TCP. Weil nur kleine Datenmengen, wie Steuerdaten und kleine Dateien, über die Netzwerkschnittstellen von beiden Seiten gesendet werden, erhielt dieses Protokoll, zu Gunsten der Verbindungssicherheit von TCP, da bei UDP immer mit Datenverlust gerechnet werden muss, den Vorzug.

In Folge dieser Entscheidung kristallisierte sich auch nach und nach ein Favorit bezüglich Programmiersprache heraus. Der gute und einfach zu handhabende Socket-Support in Python erwies sich als gelegen und in Verbindung mit der einfachen Installation (bzw. der standardmäßigen Vorinstallation in einigen Betriebssystemen) wurde Python als Implementierungssprache festgelegt. Das Framework soll möglichst vielseitig, also für möglichst jede Art von VMM, VM und Betriebssystem, einsetzbar sein, weshalb Portabilität eine wesentliche Rolle für das Framework spielt. Daher wurde versucht im Folgenden möglichst mit der Standardbibliothek aus Python in der Implementierung auszukommen, um den Installationsaufwand möglichst gering zu halten.

Einzig, wie und was getestet werden soll musste noch festgelegt werden. Aber auch hier fanden sich schnell zwei Gebiete, auf die näher eingegangen werden sollten. Einerseits sollte die Zuweisung der Rechenleistung und der Speicherzuteilung genauer unter die Lupe genommen werden, andererseits ist die erreichbare Performance bei hoher Netzlast wichtig. Es wurden Testszenarien in diesen Gebieten entwickelt, welche dann auf den virtuellen Maschinen verwendet werden können.

3.2. Umsetzung

Folgend wird auf die endgültige Implementation näher eingegangen. Das Framework reicht von der Abhandlung einzelner Tests bis hin zum Erstellen erster Grafiken und Messwerte, welche die zuvor ausgeführten Tests als Basis nehmen, welche später in dieser Arbeit in Abschnitt 3.2.4 genauer beschrieben werden.

3.2.1. Sequenzdiagramm

Anhand des Sequenzdiagramms in Abbildung 2 soll der Ablauf der des Frameworks noch einmal veranschaulicht werden. Nachdem der Tester die Einstellungen für die Tests richtig getroffen hat startet er zuerst das Skript *host.py* mit der Klasse *Host* und danach auf jeder VM die Klasse das Skript *client.py* mit der Klasse *Client*. Der

Client verbindet sich mit dem Host welcher dann das Ausführen des Tests initiiert. Der Test wird im Weiteren durchgeführt und eine Evaluation (sofern in der Konfiguration nicht anders vorgesehen) der Messwerte durchgeführt.

Anmerkung Das Sequenzdiagramm wurde zur Veranschaulichung im Bereich des Ausführen eines Tests („4: run test“) vereinfacht. In Wirklichkeit bedarf es hier noch einer Kommunikation zwischen den beiden Instanzen, um zum Beispiel die einzelnen Iterationen synchron zu halten und die Beendigung eines Tests anzuzeigen.

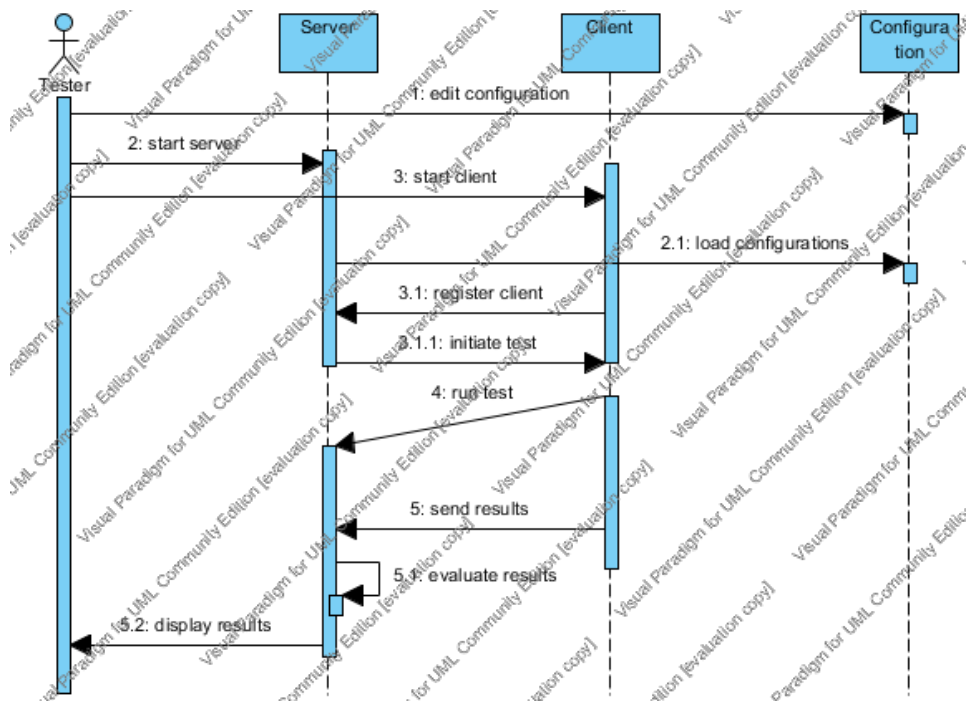


Abbildung 2: Die Abläufe zwischen der Klasse *Host* und einer Klasse *Client* dargestellt in einem Sequenzdiagramm

3.2.2. Aufbau der Architektur

Wie bereits erwähnt, handelt es sich um eine Client-Server-Architektur. Es wird eine Instanz der Klasse *Host* aufgerufen, welche auf einem beliebigen Rechner liegen kann. Innerhalb jeder virtuellen Maschine wird eine Instanz der Klasse *Client*

aufgerufen, welche sich mit dem Host verbindet (vgl. Abschnitt 3.2.3). Die Steu-
erkommunikation findet ausschließlich zwischen diesen beiden Instanzen statt. In
Abbildung 3 ist das Framework nochmals grafisch dargestellt.

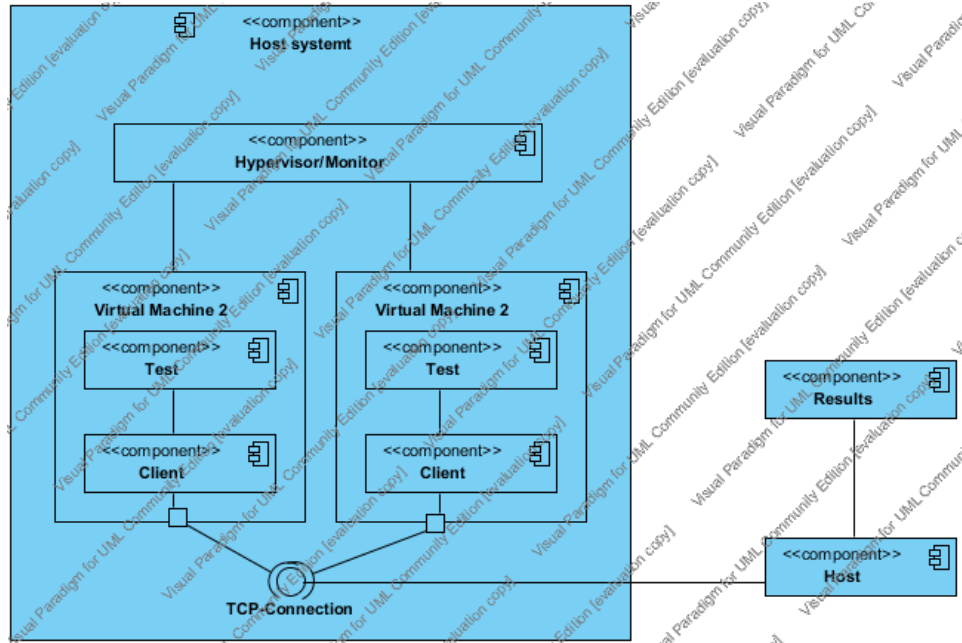


Abbildung 3: Komponentendiagramm des Frameworks mit zwei virtuellen Maschinen

3.2.3. Klassen-, Datei-, Skriptbeschreibung

Host Die Klasse Host ist jene Klasse, welche auf dem Server gestartet werden soll. Sie muss mit folgenden drei Parametern initiiert werden:

- Port to listen <int>: Es muss ein gewisser Port (TCP/IP-Model - Transport Layer) ausgewählt und deklariert werden, über den eine Verbindung zu dem Server hergestellt werden kann. Die Auswahl dazu sollte im Bereich der frei wählbaren Ports liegen und demnach größer als 1024 sein.
- Number of clients <int>: Hier wird vom Benutzer ausgewählt, wieviele Clients (also virtuelle Maschinen) sich mit dem Server verbinden sollten. Bevor besagte Anzahl nicht erreicht ist, werden die Tests nicht gestartet.
- Number of iterations <float>: Es wird angegeben, wie oft ein einzelner Testvorgang jeweils wiederholt werden soll.

Nachdem eine Instanz der Klasse aufgerufen wurde, wartet sie, dass sich die virtuellen Maschinen verbinden. Ist dies geschehen, beginnt sie eine Liste an Befehlen abzuarbeiten und diese den Clients mitzuteilen. Diese Liste wird von einer Konfigurationsdatei eingelesen und kann folgendes beinhalten:

- IO: Es wird ein Ein-Ausgabetest gestartet.
- Net: Es wird ein Netzwerktest gestartet.
- data: Die gesammelten Daten der Clients sollen an den Host geschickt werden.
- config: Eine Zusammenstellung der Konfiguration der virtuellen Maschinen (Größe Arbeitsspeicher, Prozessor, Betriebssystem) soll an den Host übermittelt werden.
- stopClient: Es wird der Befehl zum Stoppen der Clients an diese übertragen.
- plot: Aus den gesammelten Daten der Clients sollen Graphen erstellt werden.

Wird ein Test gestartet, so ist es weiters Aufgabe des Hosts die einzelnen Iterationen synchron zu halten. Es wird für jede zu absolvierende Iteration des Clients ein „iteration“-Kommando gesendet, nach dessen Erhalt die Clients damit starten ihren Test durchzuführen. Sind alle Iterationen abgehandelt, wird das nächste Kommando aus der Liste abgearbeitet. Im Falle von Netzwerktests und der Übermittlung von Daten (config, data) stellt der Server zwischen den Kommandos der Liste den Clients zusätzlich ein Socket zur Verfügung, um sich verbinden zu können.

Client Die Client-Klasse wird ebenfalls vom User aufgerufen und dieser Aufruf beinhaltet zwei Parameter:

- Port to connect <int>: Hier muss der gleiche Port angegeben werden, wie beim Host, damit sich beide Einheiten über diesen Port verbinden können.
- Server IP <String>: Um den Port einer Maschine zuteilen zu können, muss ebenfalls die IP-Adresse des Hosts angegeben werden. Es ist wichtig dabei die des richtigen Netzwerk-Interfaces zu wählen, welches mit dem Client verbunden ist. Da der Host potentiell auf derselben Maschine ausgeführt wird, wo auch der VMM installiert ist, besitzt sie in diesem Fall zumindest zwei Interfaces, nämlich ein reelles und ein virtuelles.

Die Client-Klasse verbindet sich also mit dem Host und erhält von diesem ihre Kommandos. Diese sind jeweils zu Paaren zusammengefasst und haben folgende Struktur: [Kommando, Zusatzinformation]. Diese Kommandos werden ausgelesen und dann jeweils abgearbeitet. Die Abarbeitung ist entweder das Durchführen eines Tests, oder das Senden von Information an den Host. Die Testergebnisse werden in einer CSV-Datei, welche mit einem Zeitstempel versehen ist, abgespeichert, damit man die Rohdaten weiterhin besitzt, falls bei der späteren Übertragung Fehler geschehen.

Enum Die zuvor genannten Klassen können ebenfalls eine Enum-Klasse generieren. Diese Klasse enthält keine Methoden, sondern nur Membervariablen. Die Klasse ist ein Wörterbuch für die Kommunikation zwischen Host und Client, weshalb sie vor allem String-Variablen enthält, die gewisse Befehle bedeuten. Es können aber auch Konstanten, welche sowohl für die Server- als auch die Clientseite gelten definiert werden. Verwendet man dieses Enum hat dies zum Vorteil, dass man die Konstanten jeweils (seien es numerische Werte oder auch String-Variablen) nur einmal ändern muss und nicht bei jeder Klasse einzeln.

Results In der Klasse Results werden die gesammelten Werte-Tupel verwertet. Der Host erstellt, wenn in seiner Konfigurationsdatei „data“ gesetzt ist, eine Datei mit dem Namen „results.csv“ innerhalb eines Ordners mit dem aktuellen Zeitstempel. Dort sind die Ergebnisse aller Clients nacheinander gespeichert. Es wird dabei immer abgespeichert, um welchen Test es sich handelt und von welchem Client die Daten stammen. Die Datei wird also ausgelesen und mit den Werten danach Graphen erstellt. Um dieses Feature nutzen zu können müssen die Bibliotheken *matplotlib* [10] und *NumPy* [11] installiert sein. Es können hierbei folgende Formate ausgewählt werden.

- *scatter*: Bei diesem Format wird die Laufzeit der einzelnen Iterationen für alle Clients als Scatter-Plot aufgetragen.
- *sum*: Hier wird die Laufzeit der einzelnen Iterationen jeweils summiert und dann wiederum gegen die Iterationen aufgetragen. Dies geschieht wieder für alle Clients in einem Plot.
- *stats*: Es wird eine Datei mit dem Namen „stats.csv“ in dem Ordner, welcher von der Host-Klasse mit dem Zeitstempel erstellt wurde, erzeugt. In dieser wird für jeden Client folgende statistische Information abgespeichert:
 - Mean-Value: Der Mittelwert aller Iterationen
 - Variance: Die Varianz aller Iterationen
 - Std_Variation: Die Standardabweichung berechnet aus der Varianz
 - Maximum: Der Maximalwert aller Iterationen
 - Minimum: Der Minimalwert aller Iterationen
 - Sum: Die Summe aller Werten der Iterationen
- *hist*: Es wird ein Histogramm erzeugt, dass die Verteilung aller Clients zeigt.
- *scatter90*: Es wird ein Scatterplot erzeugt, bei dem alle Werte, welche zwischen $0.95 * max \leq x \leq max$ und $0.95 * max \geq x \geq max$ ignoriert werden. Dies dient dazu um Ausreißer zu entfernen, die unter Umständen für das Messergebnis nicht relevant sind.

VBox.py Dieses Skript dient dazu, um bei der Verwendung von „Oracle VM VirtualBox“ als VMM, die Abläufe automatisieren zu können. Das Skript ruft nicht nur die Client-Klasse innerhalb der virtuellen Maschine auf, sondern startet auch diese zuvor beziehungsweise beendet diese danach. In einer Schleife geschaltet, können so ohne Zwischeneingabe des Benutzers Tests mit dem Ausführen von nur einem Skriptbefehl abgehandelt werden. Um dieses Skript verwenden zu können ist es notwendig die

- Name <String>: Als Unterscheidungsmerkmal zwischen den einzelnen virtuellen Maschinen dient in VirtualBox ein Name. Dieser muss beim Aufruf angegeben werden, um die richtige Maschine starten zu können.
- Path <String>: In der Path-Variable muss angegeben werden, wo sich in Bezug auf das virtuelle Dateisystem der virtuellen Maschine das Arbeitsverzeichnis des Frameworks liegt, damit einerseits von dort die Dateien des Frameworks aufgerufen werden können und andererseits Daten dorthin gespeichert werden können.
- Port to connect <int>: Analog wie beim Einzelaufruf muss auch hier ein Port angegeben werden. Geschieht dies nicht, so wird ein Standardport (50007) verwendet.
- Server IP <String>: Analog wie beim Einzelaufruf muss auch hier eine IP-Adresse angegeben werden. Geschieht dies nicht, so wird die IP-Adresse des Systems, auf dem das Skript gestartet wird herausgefunden und dafür verwendet.

Um dieses Skript einsetzen zu können ist die Installation der VirtualBox-SDK Voraussetzung. Diese SDK kann jederzeit unter [12] nachinstalliert werden.

config.csv In der „config.csv“-Datei, werden die Prozesse/Tests abgespeichert, welche bei einem Testdurchlauf abgehandelt werden. Aufbauend auf den Kommandos der Enum-Klasse, werden dort die Kommandos, welche im Abschnitt 3.2.3 genauer beschrieben werden abgespeichert. Der Vorteil des gewählten CSV-Formats ist, dass es einerseits mit Tabellenkalkulationsprogrammen, die dieses Format zumeist unterstützen, andererseits mit einem einfachen Editor verändert werden kann.

Test Die Klasse Test ist eine abstrakte Klasse und beschreibt nur die Funktionen und Variablen, welche allen Subtests zur Verfügung stehen müssen. Es wird zum Beispiel allgemein für jede Art von Test Membervariablen wie „start“ (die Startzeit des Tests), „path“ (den Pfad zum CSV-Dokument mit den Daten), oder ein Socket-Objekt für die Kommunikation mit dem Host bei gewissen Tests erzeugt. Um später ein Kompositum aus den einzelnen Tests zu entwickeln, wurden auch die Funktionen zum Erzeugen von CPU-, HD- und Netzwerklast in dieser Klasse definiert. Der Aufruf für diese erfolgt allerdings erst in der Sub-Klasse. Die Ergebnisse

der einzelnen Subtests werden in einer CSV-Datei gespeichert und haben folgende Struktur, wobei gilt $i = \textit{Iteration}$:

```

1 NewTest, <Test type>, <Client name>
2 Number of i <int>, Duration of i <float>, Starttime of i <float>
3 .
4 .
5 EndTest

```

3.2.4. Tests

Mattest Beim Mattest geht es darum den/die Prozessorkern(e) mit einer Rekursion möglichst auszulasten. Dies wird beim Mattest erreicht, indem die in der Informatik häufig gebrauchte Fibonacci-Folge zu berechnen, welche folgendermaßen definiert ist:

$$f_n = f_{n-1} + f_{n-2}$$

für $n \geq 2$. Des weiteren gilt $f_0 = 0$ und $f_1 = 1$. Diese Funktion kann man relativ übersichtlich in Python rekursiv beschreiben und kommt zu folgendem Ergebnis:

```

1     def fib(self, n):
2         if n <= 0:
3             return 0
4         elif n == 1:
5             return 1
6         else:
7             return self.fib(n-1) + self.fib(n-2)

```

Der Rechenaufwand steigt mit größer werdendem n exponentiell und somit auch die Laufzeit, weshalb dies Funktion bestens für diesen Test geeignet ist.

I-O Beim I/O-Test wird eine Datei einer gewissen Größe (bei meinen Tests 20MB) geöffnet und danach eingelesen. Der eingelesene Inhalt wird sofort danach wieder in eine andere Datei geschrieben. Es entstehen hier größere Lasten am Prozessor und bei der Festplatte. Die Realisierung sieht folgendermaßen aus:

```

1     def IO(self):
2         o = open('lorem.txt', 'r')
3         data = o.read()
4         i = open('temp.txt', 'w')
5         i.write(str(data))
6         o.close()
7         i.close()

```


SeekAndWrite Dieser Test ähnelt dem zuvor genannten I/O-Test. Es wird hier wiederum eine Datei, welche diesmal wesentlich größer dimensioniert (in meinem Fall 2,5GB) ist, geöffnet und nach einem gewissen String-Muster gesucht, welches vom Host bei der jeweiligen Iteration mitgeliefert wurde. Nun ist es Aufgabe des Tests die Datei zu öffnen und nach diesem Teilstring zu suchen. Wurde er gefunden, so muss zu dem Index gesprungen werden und der Teilstring von der Datei plus einer gewissen Anzahl an Zeichen, welche darauf folgen in eine andere temporäre Datei geschrieben werden. Der Suchalgorithmus wurde folgendermaßen implementiert.

```

1      def seek(self, s):
2          o = open('lorem1.txt', 'r')
3
4          lastString = ''
5          while True:
6              currentString = o.read(self.enum.chunkSize)
7              lastString = lastString + currentString
8              index1 = lastString.find(s)
9              if index1 > 0:
10                 index1 = o.tell()+index1-(2*self.enum.chunkSize)
11                 break
12             else:
13                 lastString = currentString

```

TCP Der TCP-Test ist ein Netzwerkttest und ist folgendermaßen aufgebaut. Nachdem der Client den Befehl zum Start des Tests bekommen hat, wird vom Host ein Socket geöffnet. Der Client versucht sich dorthin via TCP zu verbinden und schickt einen Teststring an den Host. Danach wird die Verbindung wieder abgebaut. Pro Iteration wird dieser Vorgang mehrere Male wiederholt, um auf eine annähernd messbare Größe zu kommen. Der zugehörige Auszug aus dem Source-Code für den Verbindungsaufbau sieht folgendermaßen aus:

```

1      def TCP(self, message, serverIP, port):
2          s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3          s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
4          s.connect((serverIP, port))
5          s.send(message)
6          if message == self.enum.stop:
7              while True:
8                  data = s.recv(512)
9                  if data:
10                     break
11          s.close()

```

4. Testfälle

4.1. Allgemeine Beschreibung

Das Entwickeln der Tests lief prinzipiell in zwei Phasen ab. Zuerst wurde ein entwickelter Test nativ, also nicht in einer VM, getestet. Hier lief in derselben Windows 7-Umgebung sowohl der Host, als auch der Client. Auf den virtualisierten Teil wurde anfangs immer verzichtet. War der Test für einen und später für mehrere Clients auf diesem nativen System stabil genug, so wurde das Testframework in der zweiten Phase auf eine Testumgebung portiert, um dort ausführlichere Testungen durchführen zu können.

Die Testumgebung für die unten angeführten Tests besteht (sofern nicht anders angeführt) aus einem „AMD Athlon(tm) 64 X2 Dual Core Processor 5000+“, mit 3GB Hauptspeicher. Als Betriebssystem wurde sowohl für das Host-System, als auch für die VMs die Linux-Distribution „Ubuntu“ in der Version 11.04 verwendet. Anzumerken ist ebenso, dass im Host-Betriebssystem eine 64Bit-Variante und in den Guest-Betriebssystemen eine 32Bit-Variante gewählt wurde. Als VMM kam VirtualBox von der Firma Oracle in der Version 4.1.2 zum Einsatz. Als Testmaschinen wurden bis zu vier, durch klonen, baugleiche Virtuelle Maschinen verwendet. Diese verfügten entweder über 256MB oder 128MB Arbeitsspeicher und es wurde ihnen eine 10GB-Festplatte von fixer Größe zugewiesen. Ebenfalls wurde der Host des Frameworks immer im Host-Betriebssystem ausgeführt und nie irgendwo anders, also remote oder innerhalb einer virtuellen Maschine.

4.2. Ergebnisse

4.2.1. TCP

Besonders interessant am TCP-Test ist dieses sprunghafte Verhalten der einzelnen Iterationen. Besonders gut im Histogramm in Abbildung 5 sieht man, dass die einzelnen Messwerte immer auf horizontalen Linien zu liegen scheinen. Diese erklären sich allerdings ziemlich rasch, wenn man sich etwas genauer mit dem TCP-Protokoll näher auseinandersetzt. Es geht hier darum, dass das Protokoll ein Standardtimeout hat, welches abgewartet werden muss, falls Pakete nicht übermittelt werden können. Dieses standardmäßig auf drei Sekunden eingestellt, weshalb und wenn man das Histogramm genauer betrachtet, so sind die Peaks immer wieder um etwa drei Sekunden zeitversetzt.

4.2.2. Mattest

Betrachtet man diesen Test, so werden findet man keine außergewöhnlichen Ergebnisse vor. Die Werte liegen im Mittel für die Maschinen zwischen 33,3 und 33,8

Sekunden dementsprechend ist die Varianz und die Standardabweichung eher klein. Das Histogramm (Abbildung 6) bestätigt dieses Ergebnis, wo deutlich eine Normalverteilung für alle vier Maschinen erkenntlich ist.

4.2.3. IO

In diesem Test gibt es zweierlei erstaunliche Sachen. Einerseits gibt es offenbar zwei „Modi“ in denen der IO-Test durchgeführt wird. Denn sowohl mit 256MB als auch mit 128MB Hauptspeichergröße gibt es im Histogramm jeweils zwei Peaks, wobei einer von diesen jeweils etwa unter 10 Sekunden ist und der andere jeweils darüber. Des weiteren interessant ist allerdings auch, dass die Laufzeiten bei 128MB entgegen der Erwartungen etwas besser sind. Eine Mutmaßung für dieses Verhalten ist ein nicht so performantes Swap-Verhalten im Falle von 256MB.

4.2.4. SeekAndWrite

Bei diesem Test sind auf den ersten Blick keine großen Überraschungen zu erwarten. Die sehr verstreuten Messpunkte im Scatterplot sind dadurch zu erklären, dass die Laufzeit abhängig von der Stelle in der Daten-Datei bis zu der gesucht werden muss. Also desto später in der Datei der gesuchte String vorkommt, desto länger dauert dies. Da dieser String aber mit einer Zufallszahl bestimmt wird, variieren die Messwerte sehr. Betrachtet man die Ergebnisse allerdings genauer, so fällt schnell auf, dass die Gesamtlaufzeit summiert über alle Iterationen große Unterschiede aufweist. So ist der Unterschied zwischen langsamster und schnellster Laufzeit circa 435 Sekunden beträgt, was bei einer Gesamtlaufzeit von 30000 Sekunden immerhin 1,4% sind.

Anzahl Clients	Hauptspeicher [MB]	Iterationen	Connects pro Iteration
4	256	200	5000

Tabelle 2: Konfiguration des TCP-Tests

4.2.5. Plots

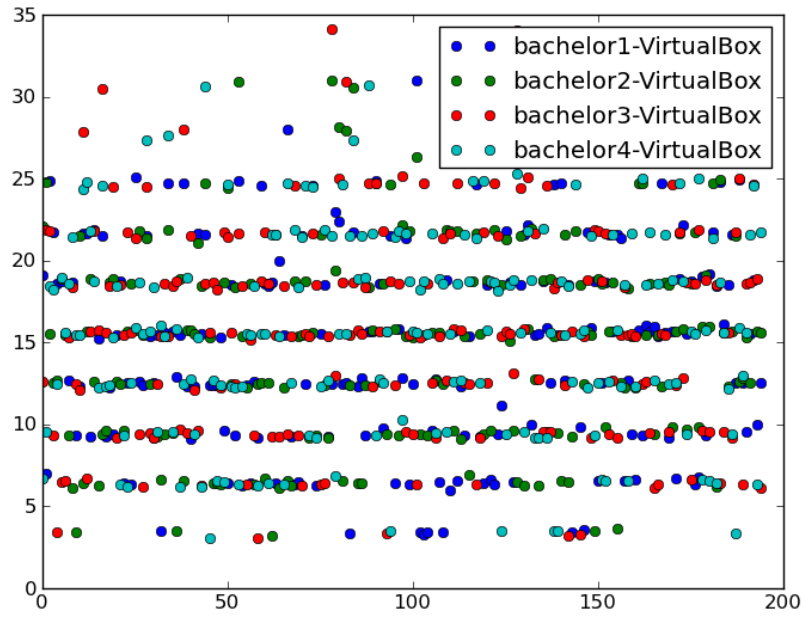


Abbildung 4: Scatterplot eines TCP-Tests

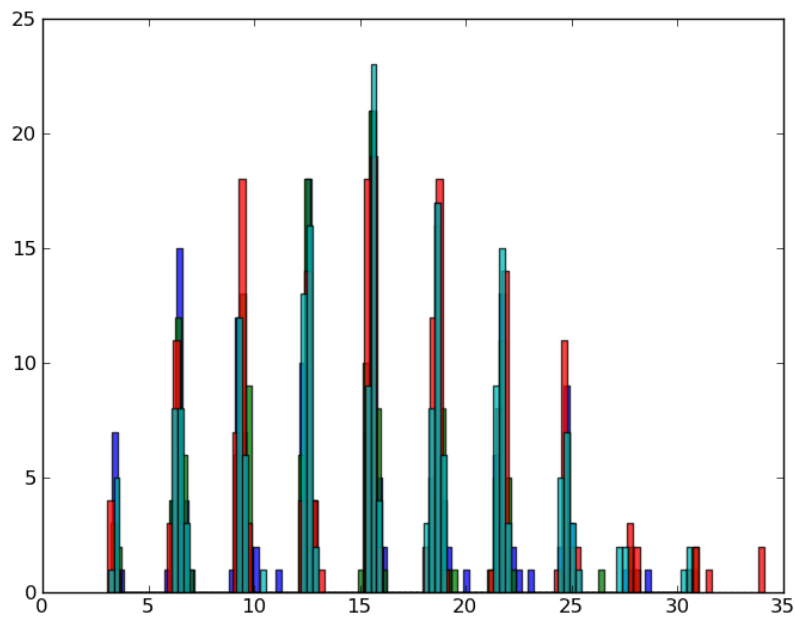


Abbildung 5: Histogramm eines TCP-Tests

Anzahl Clients	Hauptspeicher [MB]	Iterationen	Rekursionstiefe
4	256	1000	35

Tabelle 3: Konfiguration des Mattests

5. Zukunftspläne

5.1. Dokumentation

Um das Framework auch außenstehenden zur Verfügung zu stellen, wäre es noch gut eine Dokumentation außerhalb des Codes und dieser Arbeit anzufertigen. Es würde sich hierfür wahrscheinlich „Sphinx - Python Document Generator“ [13] anbieten, mit dem auch die reguläre Dokumentation von Python gemacht wird.

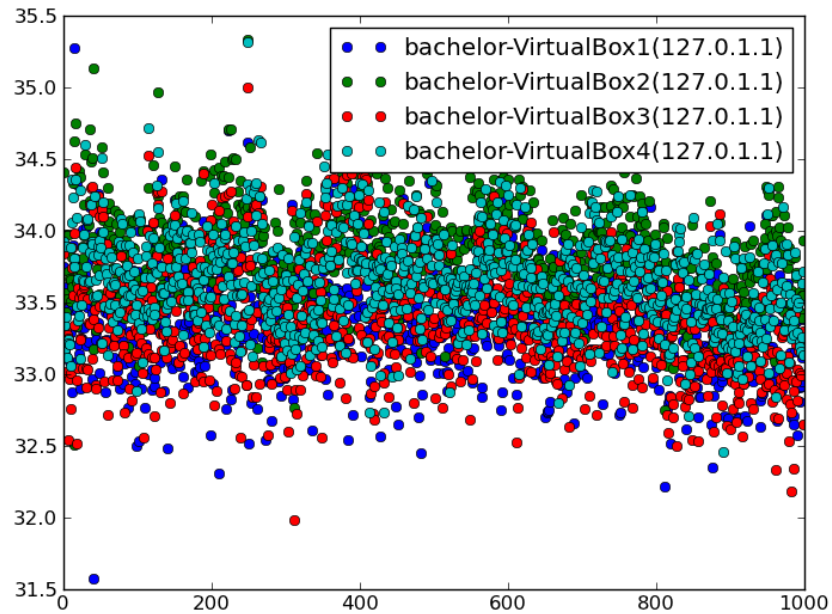


Abbildung 6: Scatterplot eines Mattests

Anzahl Clients	Hauptspeicher [MB]	Iterationen
4	128-256	500

Tabelle 4: Konfiguration des IO-Tests

5.2. Messkampagnen und Messstrategien

In den bis jetzt ausgeführten Tests wurde ausschließlich die Größe des Hauptspeichers variiert. Natürlich gibt es hier mehrere Variablen denen man seine Aufmerksamkeit schenken könnte. Man könnte beispielsweise genauso die virtuelle Prozessorleistung variieren, beziehungsweise Restriktionen für den Netzwerkverkehr setzen, um herauszufinden, ob der VMM in diesen Fällen immer noch fair agiert.

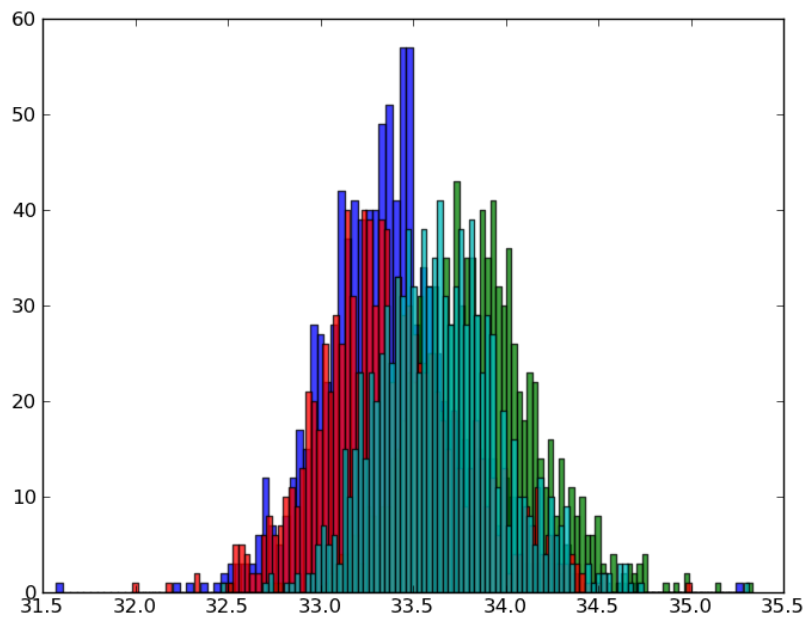


Abbildung 7: Histogramm eines Mattests

Anzahl Clients	Hauptspeicher [MB]	Iterationen
4	256	200

Tabelle 5: Konfiguration des SeekAndWrite-Tests

5.3. Ansteuerung des Hypervisors

Um das Durchführen von den zuvor genannten Messkampagnen und -strategien möglichst effizient durchsetzen zu können, wäre das Ansteuern des VMM zwischen einzelnen Tests noch eine wichtige Komponente. Somit wäre es zum Beispiel möglich einen Test zu starten und nach dessen Beendigung die Konfiguration des VMM zu verändern und den Test erneut zu starten.

5.4. Weitere Tests

Auch die Eingliederung von neuen Tests in das Framework wäre eine Bereicherung für ebendieses. Die Art und Weise, wie dieser zu erstellen ist wird im Anhang

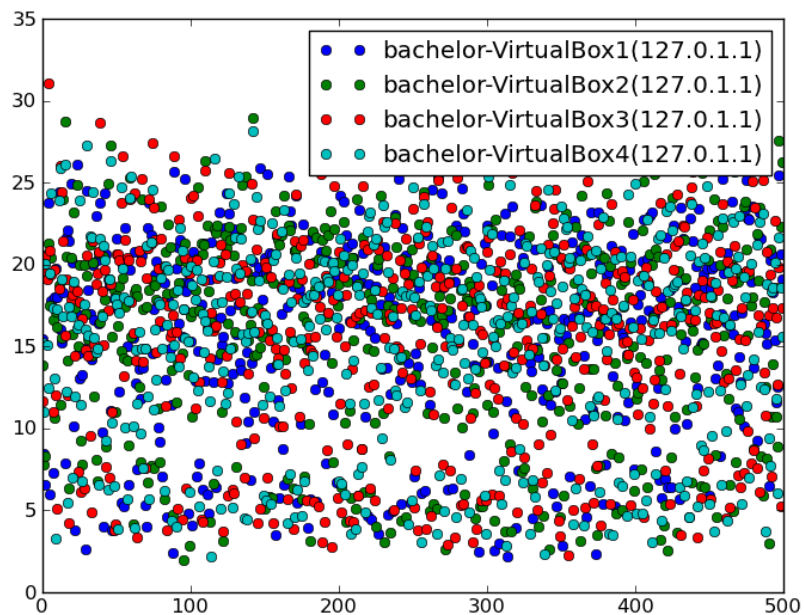


Abbildung 8: Scatterplot eines IO-Tests (256MB Hauptspeicher)

B genauer beschrieben. Die Bandbreite von noch zu erstellenden Tests ist denkbar groß, so könnten beispielsweise diverse spezifische Komponententests implementiert werden, oder auch die bestehenden noch erweitert werden.

5.5. Vergleich nativer und virtualisierter Tests

Prinzipiell besteht ebenfalls die Möglichkeit, neben den virtualisierten Messungen ebenfalls welche auf nativen Systemen durchzuführen. Hier müssen allerdings die Voraussetzungen passen und es sollten gleiche Maschinen verwendet werden. Eine andere Überlegung wäre, während der virtualisierten Tests einen Test nativ laufen zu lassen. Heikel wird es hier allerdings bei der Interpretation der Ergebnisse, denn es ist unmöglich die gewonnen Messdaten im nativen System direkt als Referenzwert zu verwenden.

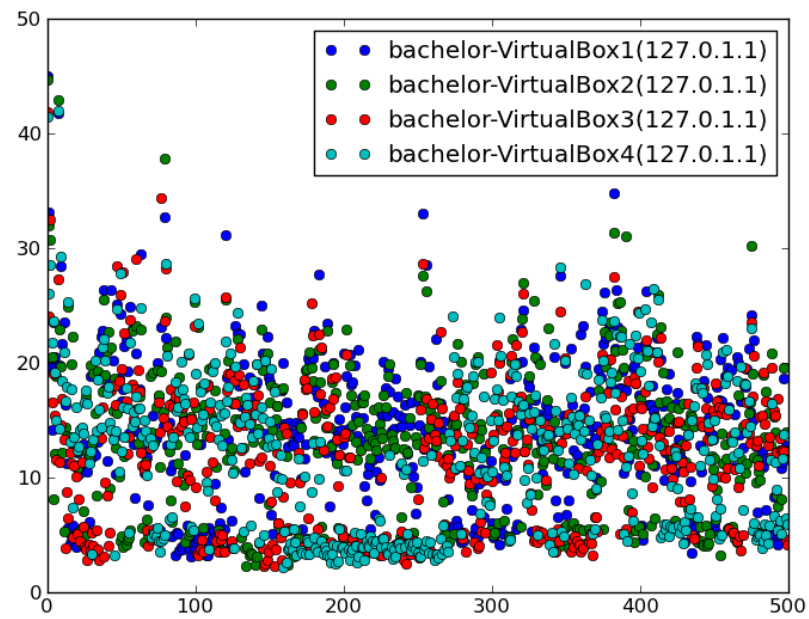


Abbildung 9: Scatterplot eines IO-Tests (128MB Hauptspeicher)

6. Schluss

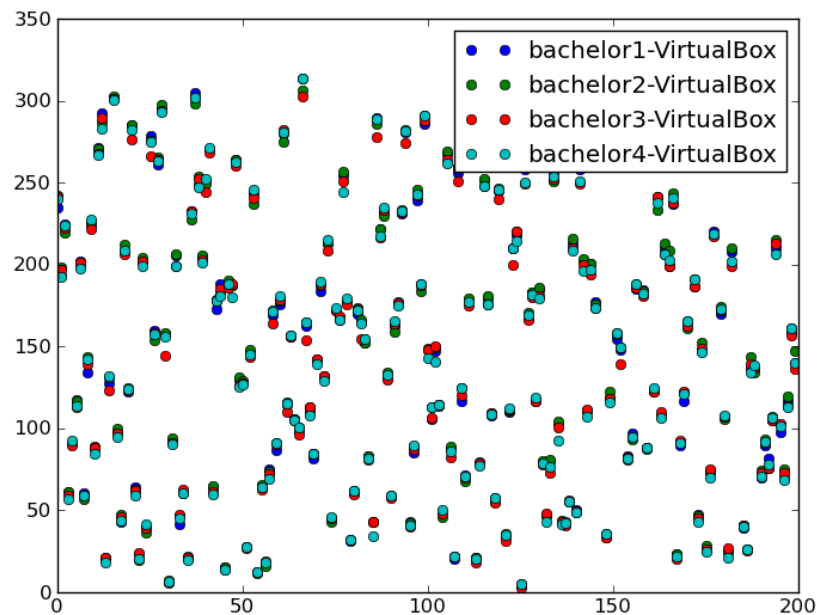


Abbildung 10: Scatterplot eines SeekAndWrite-Tests

A. Lessons learned

A.1. Portierung

Ein gewisses Problem hat die Portierung von einem Betriebssystem auf ein anderes dargestellt. Man musste während der Entwicklung immer genau darauf achten, dass der Source-Code auf gängigen Betriebssystemen reibungsfrei läuft. In meinem Fall hat es neben ein paar Kleinigkeiten auch einmal ein etwas größeres Problem gegeben. Beim TCP-Test war der ursprüngliche Code so beschaffen, dass ein Socket-Objekt initialisiert und danach verwendet wird, was folgendermaßen ausgesehen hat:

```

1  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2  s.connect((serverIP, port))
3  s.send(message)

```

Dieser Aufruf geschieht mehrere Male in Folge und unter dem Betriebssystem „Windows 7“ hat dies ohne Tadel geklappt. Als es dann zur Portierung auf einen Ubuntu-Rechner gekommen ist, wurde permanent eine Exception geworfen. Der

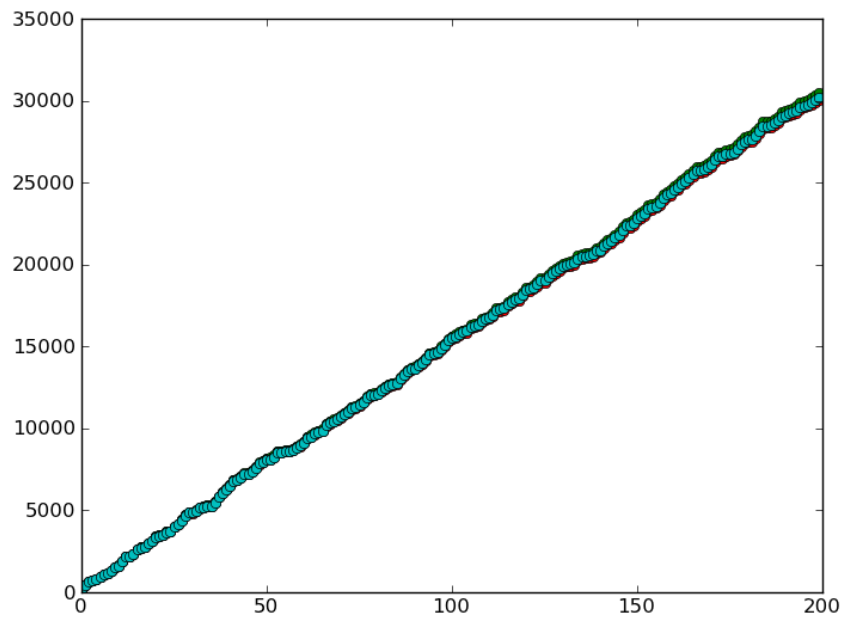


Abbildung 11: Summation der Laufzeiten eines SeekAndWrite-Tests

Grund stellt sich als folgender heraus. Unter Ubuntu kann man ein und das selbe Socket standardmäßig nicht mehrmalig hintereinander verwenden. Es bedarf einer Änderung in den Optionen des Sockets, welche zwar durch eine einzige Zeile erledigt werden kann, aber der Grund dafür mühsam herauszufinden war. Mit der Option

```
1 s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

hat der TCP-Test auch unter Ubuntu richtig funktioniert.

A.2. Expect the unexpected

In einem sehr frühen Stadium des Frameworks wurden Tests durchgeführt. Das Ergebnis war ziemlich interessant, denn es scheint sich eine gewisse Präferenz einer VM abzuzeichnen. Doch schnell war ein möglicher Grund für dieses Verhalten gefunden, nämlich die Reihenfolge in welcher die virtuellen Maschinen gestartet wurden. Allerdings war es mir im Nachhinein nicht mehr möglich diese Reihenfolge zu rekonstruieren und so sank die Wertigkeit der Testmessung im selben Moment. Ein uner-

wartetes Ereignis führte also dazu, dass man nochmals von Vorne beginnen konnte. Herr über das Unerwartete zu werden ist denkbar schwierig, einzig und allein eine stetige Dokumentation von Arbeitsschritten, die im ersten Moment noch unwichtig erscheinen mögen, könnte so ein Vorkommen abwenden.

A.3. Synchronizität

Es ist ein denkbar schmaler Grad im Bereich der Synchronität. Einerseits versucht man alles so synchron als möglich zu halten, andererseits gibt es keine absolute Synchronität. Es ist natürlich notwendig die Abläufe in einem Test möglichst synchron zu halten, allerdings kann es auch sein, dass man durch „übertriebene Genauigkeit“ sein eigenes Testergebnis verfälscht. Beispielsweise versucht man nur einen TCP-connect von mehreren Maschinen synchron ablaufen zu lassen, so wird der Fehler in der Zeitnehmung vielleicht sogar größer sein, als die genommene Zeit. Während eine der virtuellen Maschinen noch nicht mal den Befehl zum Connect erhalten hat, hat die andere ihr Pensum bereits durchlaufen.

Aber auch in die gegenteilige Richtung ist die Synchronität nicht immer schwer zu handhaben. Nimmt man hier zum Beispiel eine sehr große Anzahl an Connects, so kann es passieren, dass eine virtuelle Maschine schon lange fertig ist, während die andere noch eine sehr große Zahl an Connects vor sich hat. Die Belastung ist demnach nicht mehr dieselbe, weshalb die Ergebnisse, wo nur mehr eine VM arbeitet eigentlich zu ignorieren wären.

Aus diesem Grund wurde in dem Framework die Größe der Iteration eingeführt. Man versucht die Laufzeiten einer Iteration so zu wählen, dass die beiden oben genannten Erscheinungen nur vernachlässigbare Nebeneffekte sind und diese Messdaten dann auszuwerten.

B. Eigene Testklasse erstellen

Um eine eigene Testklasse für das Framework zu erstellen, müssen einige Dinge befolgt werden, um vernünftige Ergebnisse zu erzielen. Die Klasse *UserTest* sollte immer eine Sub-Klasse der Klasse *Test* sein, damit ihr die richtigen Membervariablen und Funktionen zur Verfügung stehen. Braucht man jetzt noch zusätzliche Membervariablen, können diese an erster Stelle deklariert werden. Die Klasse *UserTest* braucht zumindest eine Funktion mit dem Namen *startUser(self)*, welche von der *Client*-Instanz aufgerufen werden kann. Dort findet der eigentliche Testablauf statt. Die Funktion muss folgendermaßen aufgebaut sein:

```
1 class UserTest ( Test ):
2
3     def startUser ( self ):
4         self.initTest ()
```

```

5
6         while True:
7             i = self.receive()
8             if i[0] == self.enum.iteration:
9                 start = time.time()
10                \textit{UserTestFunction()}
11                dur = time.time() - start
12                values = [str(i[1]), str(dur), str(start)]
13                self.results.writerow(values)
14            if i[0] == self.enum.stop:
15                break
16
17        self.endTest()

```

Was hier passiert kann folgendermaßen veranschaulicht werden:

- Zeile 4: Der Test wird initialisiert. Es stehen für den eigens kreierten Test folgende Variablen zur Verfügung, die verwendet werden können, allerdings natürlich nicht müssen:
 - *self.path*: Der FileObject zu der CSV-Datei, in welcher alle Daten dieser Testserie gespeichert werden.
 - *self.serverIP*: Die IP-Adresse des Servers, zu dem man verbunden ist, als String gespeichert.
 - *self.port*: Der Port des Servers, zu dem man verbunden ist, als String gespeichert.
 - *self.enum*: Eine Objekt der Klasse *Enum*, um auf die Kommunikationsbefehle Zugriff zu haben.
 - *self.init*: Die Zeit gemessen via `time.time()` in Zeile 4 des Aufrufes *startUser*
 - *self.start*: Eine Variable zum Speichern der Startzeit einer Iteration.
 - *self.dur*: Eine Variable zum Speichern der Laufzeit einer Iteration.
- Zeile 6ff.: Um adequat die einzelnen Iterationen gleichzeitig mit allen anderen virtuellen Maschinen durchführen zu können, ist dieses Schleifenkonstrukt notwendig. Prinzipiell kann man dieses folgend beschreiben. Zuerst wird eine Verbindung mit dem *Host* aufgebaut und eine Nachricht wird entgegengenommen. Diese ist zweiteilig und enthält entweder *i[0]=Befehl zur Iteration*, *i[1]=Nummer der Iteration* oder *i[0]=Befehl zur Beendigung*, *i[1]=0*.
- Zeile 8, 14: In den If-Abfragen wird zwischen diesen beiden Fällen unterschieden und entweder der Code zum Testen ausgeführt, oder die Schleife terminiert.

- Zeile 9-13: Dies ist ein Vorschlag für die Zeitnehmung und das Abspeichern der Daten. Der Ablauf muss natürlich nicht genauso sein, es empfiehlt sich aber einen ähnlichen Ablauf einzuhalten. Tut man dies, so können anschließend automatisch Graphen und gewisse statistische Werte mit der Klasse Results berechnet werden. Zusätzlich ist noch anzumerken, dass in Zeile 12 und 13 nicht zwingendermaßen die Nummer der Iteration, die Laufzeit der Iteration und die Startzeit der Iteration abgespeichert werden müssen. Allgemeiner könnte man diese Zeilen wahrscheinlich als

```
1 values = [str(Graph_X_Value), str(Graph_Y_Value), str(AnyValue)]
2 self.results.writerow(values)
```

beschreiben. Es ist also wichtig dass an der ersten Stelle der x-Wert dessen, was später im Graph angezeigt werden soll steht und an zweiter Stelle der y-Wert. Der dritte Wert ist nicht entscheidend und dient nur dazu, um zum Beispiel eine weitere Messgröße ebenfalls zu loggen. Sie wird allerdings beim Erstellen der Graphen nicht berücksichtigt.

- Zeile 17: Die Methode *startUser* sollte zuletzt noch mit der Methode *self.endTest()* finalisiert werden. Hier werden noch Kleinigkeiten in die CSV-Datei geschrieben und weiters die Variablen freigegeben.

Um selbst weitere Tests im Bereich der Netzwerkauslastung zu erstellen, muss man dafür seinen eigenen Server definieren, da nicht allgemein gewährleistet werden kann, dass jeder Testaufbau in diesem Fall analog ist. Zum Beispiel kann es zu einer gewollten Vertauschung von `socket.recv()` und `socket.send()` auf Server- und Clientseite kommen, was im Voraus in diesem Framework natürlich nicht berücksichtigt werden kann.

Literatur

- [1] D. Ongaro, A. L. Cox, and S. Rixner, eds., *Scheduling I/O in virtual machine monitors*, VEE {grq}08, (New York and NY and USA), ACM, 2008.
- [2] IBM Systems, “Virtualization,” 2005.
- [3] “Homepage Oracle VM VirtualBox (<https://www.virtualbox.org/>).”
- [4] “Homepage Windows Virtual PC (<http://www.microsoft.com/windows/virtual-pc/>).”
- [5] “Homepage Xen Hypervisor (<http://xen.org/>).”
- [6] “Homepage VMware (<http://www.vmware.com/>).”
- [7] D. A. Menascé, “Virtualization: Concepts, Applications, and Performance Modeling,” 2005.

- [8] D. Abramson, “Intel(R) Virtualization Technology for Directed I/O,” *Intel Technology Journal*, vol. 10, no. 03, 2006.
- [9] C. Weng, Z. Wang, M. Li, and X. Lu, eds., *The hybrid scheduling framework for virtual machine systems*, VEE {\\grq}09, (New York and NY and USA), ACM, 2009.
- [10] “Homepage Matplotlib (<http://matplotlib.sourceforge.net/>).”
- [11] “Homepage Numpy (<http://numpy.scipy.org/>).”
- [12] N. Igotti, “Python API to the VirtualBox VM (http://blogs.oracle.com/nike/entry/python_api_to_the_virtualbox),” 5.9.2008.
- [13] “Homepage Sphinx (<http://sphinx.pocoo.org/>).”