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, 15. November 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 deren Laufzeiten geloggt. Anhand dieser Zeiten wird versucht herauszufinden, ob gewisse virtuelle Maschinen bevorzugt werden und daher der VMM beziehungsweise dessen Scheduling-Mechanismus fair 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“ wirt 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

Tabellenverzeichnis

Abbildungsverzeichnis

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 hardwarenahen 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 Gleichbereichtigung einzelner virtueller Maschinen gemacht werden können. Neben der Implementierung des Frameworks ist auch das Ausführen von Tests und deren Analyse Teil dieser Arbeit. Eine zusätzliche Anforderung ist Portabilität zu erreichen, sodass das Framework möglichst auf allen unterschiedlichen Konfigurationen von Hardware und Software eingesetzt werden kann und der Aufwand des Testers/Users, der die Tests durchführen will, minimiert werden kann. Möglichst große Teile des Frameworks sollen daher 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 die Verwendung 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. Eine Auflistung einiger Virtualisierungsprojekte von unterschiedlichen Firmen ist in Tabelle 1 zu finden. Auch ein Großunternehmen wie *IBM* setzt, auch wenn nicht eigens eine Software implementiert wird, auf Virtualisierung und zwar mit einer engen Integration anderer Virtualisierungsumgebungen [2].

2.1. Virtualisierung

2.1.1. Vorteile von Virtualisierung

Nun stellt sich die Frage, wieso man diesen Schritt zur Virtualisierung von Hardware überhaupt machen soll. [7] gibt mögliche Antworten und zählt mehrere Gründe auf:

- Sicherheit: Durch die Abkapslung der einzelnen virtuellen Komponenten von der echten Hardware und zwischen den einzelnen Betriebssysteme fällt ein Angriff auf die Sicherheit eines Systems nicht sehr gravierend aus. Kommt es zu einem Angriff auf eines der virtuellen Systeme, bleiben andere Systeme, die auf der gleichen Hardware laufen, unbetroffen.
- Zuverlässigkeit und Verfügbarkeit: Kommt es zu einem Software-Fehler innerhalb einer VM so ist das System rundherum nicht betroffen und die übrigen virtuellen Instanzen stehen nach wie vor zur Verfügung. Darüber hinaus können virtuelle Maschinen auch in Echtzeit neu organisiert oder auf andere Maschinen migiriert werden („Live Migration“). Die virtuelle Maschine ist während einer Migration für andere Prozesse während der ganzen Dauer verfügbar [8].

- Kostenaufwand: Wie bereits erwähnt kommt mit Virtualisierung meist auch eine Ersparnis an Kosten, durch geringeren Anschaffungsaufwand, vor allem in größeren Unternehmen.
- Anpassung bei Belastungsänderungen: 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 zugrunde liegenden Hardware sind.
- Belastungsänderung: Wird die Belastung für ein System zu groß, können einzelne virtuelle Maschinen auf andere Systeme migriert werden, um die Last unter diesen Systemen aufzuteilen.
- Alte Applikationen: 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 eine alte Applikation nach wie vor in der virtuellen Maschine ausgeführt werden kann.

2.1.2. Nachteile von Virtualisierung

Doch Virtualisierung bringt auch Nachteile mit sich. [9] zeigt einige Schwachstellen von virtuellen Systemen auf:

- Overhead: Der Aufwand eine virtuelle Maschine zu betreiben ist größer, als würde man nur ein natives System verwenden. Man muss die Hardwarekomponenten des Systems mehrfach belasten, was einen Verfall der Performance mit sich bringt. Ziel der Entwicklung in der Virtualisierung muss es sein, dass der Performance-Unterschied zwischen virtuellem und physischem System minimiert wird.
- Schwachstellen/Stabilität: Die virtuelle Maschine ist natürlich abhängig von der physischen Hardware. Kommt es hier zu einem Fehler, so kann die virtuelle Maschine davon ebenfalls betroffen sein. Die Hardwarekomponenten, die auch unter größerer Last stehen, müssen abgesichert werden. Ein Fehler in der Hardware kann gleich mehrere Systeme zum Stillstand bringen.
- Konfigurationsschnittstelle: Die Schnittstelle zur Konfiguration (VMM) ist sehr eng mit den virtuellen Maschinen verbunden. Kommt es zu unerwarteten Zwischenfällen, sind auch hier gleich mehrere Systeme betroffen. Mit dem VMM gibt es eine Komponente mehr, die gegen Fehler geschützt werden muss, damit das System reibungslos läuft.

2.1.3. Scheduling

Jeder Hypervisor einer virtuellen Maschine muss eine Zugriffsregelung (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.

Xen-Scheduling Bei genannten proprietären Virtualisierungsprodukten ist es schwierig, 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 die Prozesse von den 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 meherer 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 zumindest zwischen drei 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: [10] und [11] erweitern diese Typen und sprechen von einem Hybriden-Ansatz, der 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[1] 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. Weiters sind drei virtuelle Maschinen abgebildet, welche nur über den Hypervisor mit der Hardware kommunizieren können.

[1] Eine Domäne die Hardwaretreiber enthält, um die Kommunikation von nativen und virtuellen Komponenten zu ermöglichen.

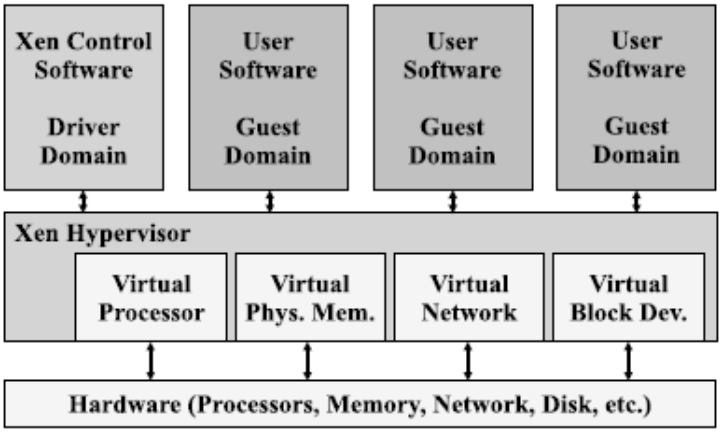


Abbildung 1: Schematische Darstellung des Xen Hypervisors [1].

3. Architektur des Frameworks

3.1. Design

Das Design des Frameworks ermöglicht zwei unterschiedliche Kategorien von Tests für virtuelle Maschinen:

- Sequenziell: Ein Test kann auf einer virtuellen Maschine ausgeführt werden. Durch eine Vielzahl an Tests mit unterschiedlichen Konfigurationen kann man einen Schluss über die Skalierung des VMM und der virtuellen Mascheinen ziehen. Hier ist wichtig, dass die Tests nacheinander ausgeführt werden, damit sie sich untereinander nicht beeinträchtigen.
- Synchron: Es laufen mehrere virtuelle Maschinen, auf welchen gleichzeitig derselbe 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 ein System, das zumeist im synchronen Betrieb laufen soll, gefällt, trotzdem wird der sequentielle Betrieb aber unterstützt. Um die Prozesse im synchronen Fall möglichst gleichzeitig ablaufen lassen zu können, wird eine Instanz implementiert, welche als Monitor fungiert. Die Aufgabe dieser Instanz ist, zu regeln 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 als Server-Prozess im Host-Betriebssystem ausführen und jede einzelne virtuelle Maschine (Client) verbindet sich mit diesem.

Zur Kommunikation zwischen den Instanzen Client und Server wird das verbindungssichere Transport-Control-Protocol (TCP) verwendet.

Weiters wurde Python als Implementierungssprache festgelegt. Grund dafür war der gute und einfach zu handhabende Socket-Support in Python und außerdem die einfache Installation im Betriebssystem (bzw. die standardmäßige Vorinstallation in einigen Betriebssystemen). 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.

Für die Leistungsuntersuchung wurde der Fokus auf zwei Gebiete gelegt, auf die näher eingegangen wird. Einerseits sollte die Zuweisung der Rechenleistung und der Speicherzuteilung, sowie die Performance bei Ein- und Ausgabeoperationen auf der Festplatte 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.

3.1.1. Aufbau der Architektur

Wie bereits angedeutet, 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 3.2.2). Die Steuerkommunikation findet ausschließlich zwischen diesen beiden Instanzen statt. In Abbildung 2 ist das Framework nochmals grafisch dargestellt.

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 genannten Tests als Basis nehmen. Die Tests werden in Abschnitt 3.2.3 genauer beschrieben.

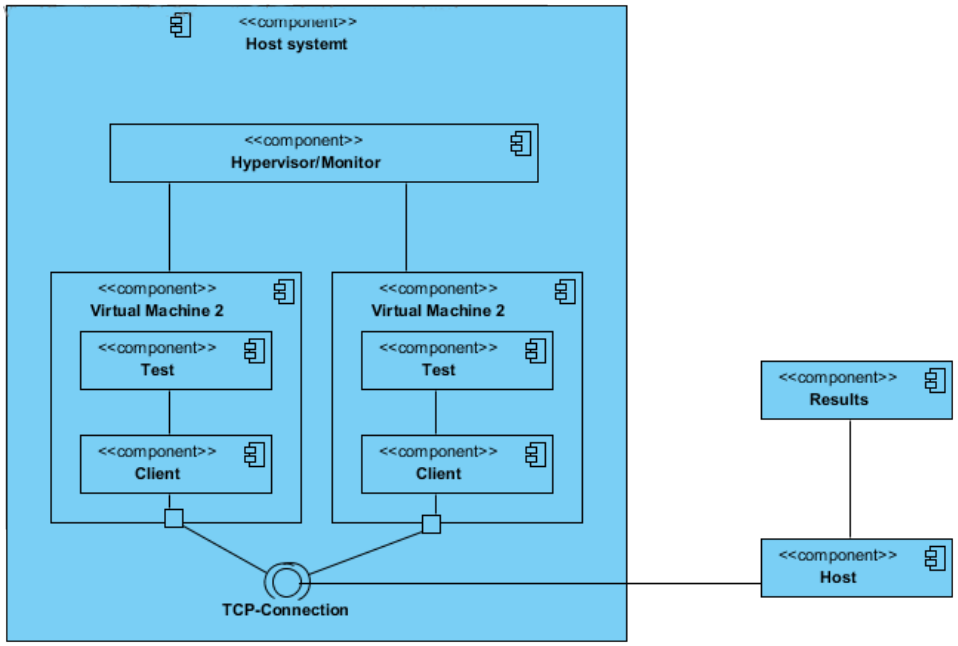


Abbildung 2: Komponentendiagramm des Frameworks mit zwei virtuellen Maschinen

3.2.1. Sequenzdiagramm

Das Sequenzdiagramms in Abbildung 3 soll den Ablauf des Frameworks genauer veranschaulichen. 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 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 der Messwerte durchgeführt.

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.

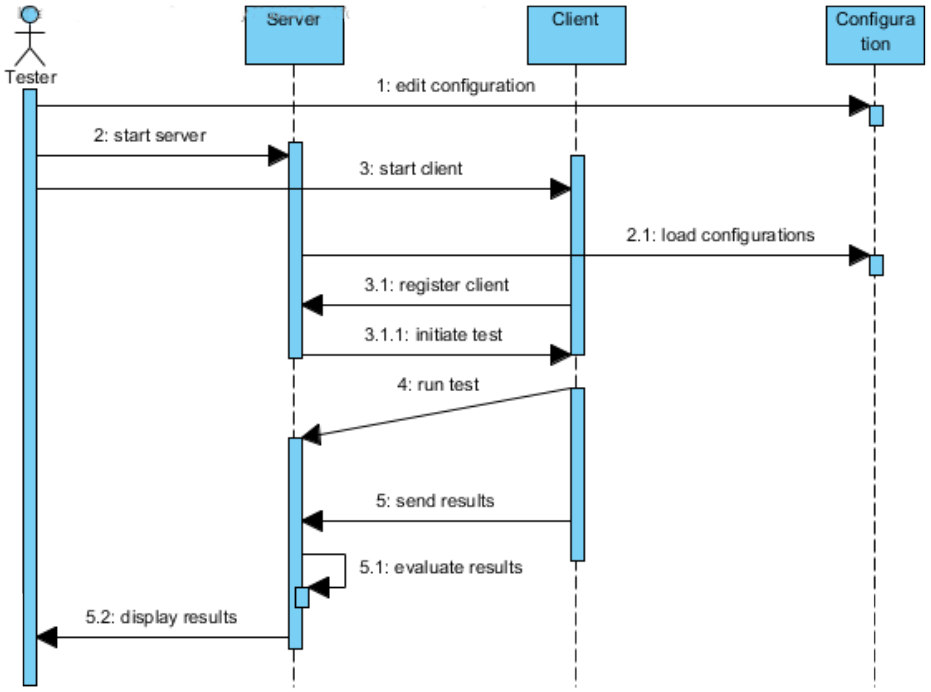


Abbildung 3: Der Ablauf zwischen der Klasse *Host* und einer Klasse *Client* in einem Sequenzdiagramm

3.2.2. Klassen-, Datei-, Skriptbeschreibung

Die Umsetzung des Frameworks erfolgt zu einem großen Teil in Python-Klassen. Es wurde nur bei einigen Implementierungen (z.B. Evaluierung der Resultate) auf das Klassen-Modell verzichtet, da es nicht nötig erschien und den Ablauf nicht wesentlich erleichterte. Für die Speicherung von Daten wurde das CSV-Format verwendet, da dessen Bedienung einfach und intuitiv ist. Abbildung 4 verschafft einen Überblick, wo die einzelnen Klassen/Skripte instanziiert werden und zeigt die wesentlichen Assoziationen untereinander auf.

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 TCP-Port 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.

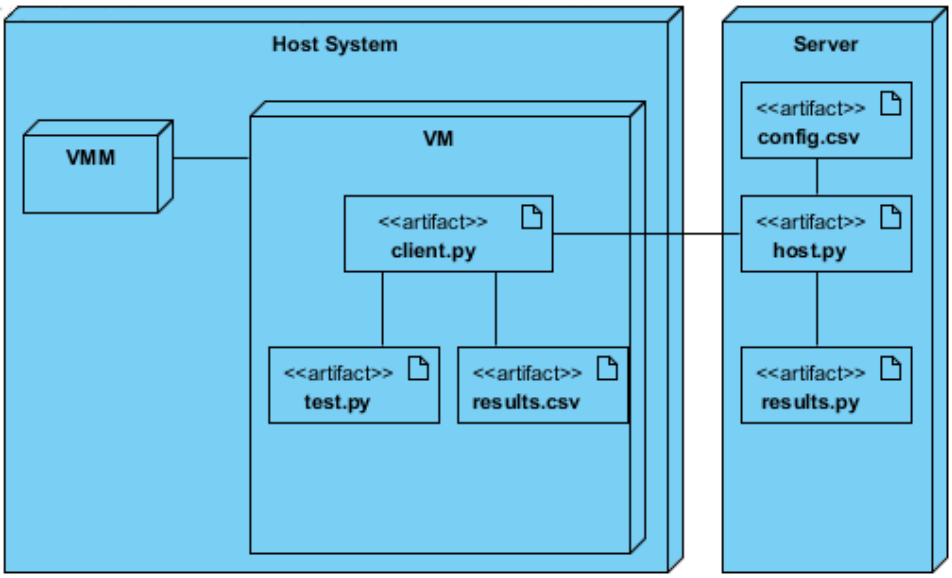


Abbildung 4: Struktur und Aufteilung der Klassen innerhalb des Frameworks

- 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 <int>: Es wird angegeben, wie oft ein einzelner Testvorgang jeweils wiederholt werden soll.

Sobald die Host-Instanz läuft, 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 übermittlet 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 einzehlnen 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 vom User innerhalb der jeweiligen virtuellen Maschine aufgerufen. 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 dabei wichtig die IP des richtigen Netzwerk-Interfaces zu wählen, welches mit dem Client verbunden ist. Da der Host potenziell 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 Tupeln zusammengefasst und haben folgende Struktur: [Kommando, Zusatzinformation]. Diese Kommandos werden ausgelesen und dann in dieser Reihenfolge abgearbeitet. Die Abarbeitung ist entweder das Durchführen eines Tests, oder das Senden von Information an den Host. Die Testergebnisse werden in einer CSV-Datei abgespeichert, welche mit einem Zeitsptempel versehen ist, damit es zu keinem Verlust der Rohdaten kommt, falls es bei der späteren Übertragung zu Fehlern kommt.

Const *Host*- und *Client*-Klasse können jeweils eine *Const*-Klasse generieren. Diese Klasse enthält keine Methoden, sondern 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 festgelegt werden, welche sowohl für die Server- als auch die Clientseite gelten. Verwendet man diese Klasse *Const* 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.py In dem Skript von „results.py“ 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 ausgelesen und mit den Werten Graphen erstellt. Um dieses Feature nutzen zu können müssen die Bibliotheken *matplotlib* [12] und *NumPy* [13] 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 Werte der Iterationen
- *hist:* Es wird ein Histogramm erzeugt, das 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 $1.05 * min \geq x \geq min$ liegen, ignoriert werden. Dies dient dazu um Außreißer zu entfernen, die unter Umständen für das Messergebnis nicht relevant sind.

Die Plot-Kommandos sollten in der „config.csv“ immer als letzte Kommandos gesetzt werden. Weiters ist darauf zu achten, dass die Maschinen unterschiedliche Namen haben (es muss also `socket.gethostname()` jeweils einen unterschiedlichen Wert zurückliefern), da es sonst zu einer falschen Zuordnung der Werte kommen kann.

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 bziehungsweise beendet diese danach. In einer Schleife geschalten, 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 folgenden Parameter anzugeben:

- 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 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 kann.
- 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* und der *VirtualBox Guest Additions* Voraussetzung. Diese SDK kann jederzeit unter [14] nachinstalliert werden. Die Gasterweiterung für VirtualBox ist Bestandteil der Installation.

config.csv In der „config.csv“-Datei, werden die Prozesse/Tests abgespeichert, welche bei einem Testdurchlauf abgehandelt werden. Aufbauend auf den Kommandos der *Const*-Klasse, werden dort die Kommandos, welche im Abschnitt 3.2.2 genauer beschrieben werden, abgespeichert. Der Vorteil des gewählten CSV-Formats ist, dass es einerseits mit Tabellenkalukulationsprogrammen, 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 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 Stuktur, wobei gilt $i = Iteration$:

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

```
4   .
5   EndTest
```

3.2.3. Implementierte Tests

Mattest Beim Mattest geht es darum den/die Prozessorkern(e) mit einer Rekursion möglichst auszulasten. Dies wird erreicht, indem die Fibonacci-Folge berechnet wird, welche wie folgt 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 folgendermaßen in Python rekursiv beschreiben:

```
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 auf den einzelnen virtuellen Maschinen.

I-O Beim I/O-Test wird eine Datei einer gewissen Größe (bei den in Abschnitt 4.2 gezeigten Tests 20MB) geöffnet und danach eingelesen. Der eingelesene Inhalt wird sofort danach wieder in ein 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 geöffnet, welche diesmal wesentlich größer dimensioniert (bei den in Abschnitt 4.2 gezeigten Tests 2,5GB) ist. Danach wird nach einem gewissen String-Muster gesucht, welches vom Host bei der jeweiligen Iteration mitgesendet wird. 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 folgenden Zeichen 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.const.chunkSize)
7               lastString = lastString + currentString
8               index1 = lastString.find(s)
9               if index1 > 0:
10                  index1 = o.tell()+index1-(2*self.const.chunkSize)
11                  break
12              else:
13                  lastString = currentString
```

TCP Der TCP-Test ist ein Netwerktest, der das Verhalten des VMM bei vielen Verbindungsversuchen via TCP beschreibt. Nachdem der Client den Befehl zum Start des Tests bekommen hat, wird vom Host ein Socket geöffnet. Der Client versucht sich dorthin mit 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 Laufzeit 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.const.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 im Host-Betriebssystem und nicht in einer VM, getestet. Hier lief in derselben Windows 7-Umgebung sowohl der Host, als auch der Client. In der zweiten Phase wurde das Testframework später auf eine virtualisierte Testumgebung portiert, um dort ausführlichere Testungen durchführen zu können.

Die Testumgebung für die unten angeführten Tests besteht 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. Weiters wurde der Host des Frameworks immer im Host-Betriebssystem ausgeführt.

4.2. Ergebnisse

Im folgenden Abschnitt werden Ergebnisse von Tests diskutiert, welche auf der Testumgebung durchgeführt wurden. Plots der Ergebnisse sollen einen Überblick über das Verhalten des VMM geben. Die Konfigurationen der einzelnen Tests sind in Tabelle 2 nochmals zusammengefasst. Um die Plots übersichtlicher zu gestalten wurde auf die Legenden innerhalb der einzelnen Bilder verzichtet. Eine Legende, die für alle Bilder der Tests gilt, kann man in Abbildung 5 finden. Die virtuellen Maschinen sind von 1 bis 4 durchnummeriert und sie wurden auch immer in dieser Reihenfolge hochgefahren und der *Client* des Frameworks gestartet.

4.2.1. TCP

Vgl. Abbildung 6 auf Seite 19

Besonders interessant am TCP-Test ist das sprunghafte Verhalten der einzelnen Iterationen. Besonders gut im Histogramm in Abbildung 6(b) sieht man, dass die einzelnen Messwerte immer auf vertikalen Linien zu liegen scheinen. Diese entstehen, da das Protokoll ein Standardtimeout hat, welches abgewartet werden

muss, falls Pakete nicht übermittelt werden können. Das Timeout ist standardmäßig auf drei Sekunden eingestellt, weshalb im Histogramm die Peaks immer wieder um etwa drei Sekunden zeitversetzt auftreten [15][16]. Zur Fairness muss man sagen, dass diese nicht besonders beeinträchtigt ist. Die Mittelwerte liegen nämlich zwischen $\bar{t}_{min} = 14,82s$ und $\bar{t}_{max} = 16,00s$, somit deren maximale Differenz bei $1,18s$, was bei einer Laufzeit von rund $3000s$ eher vernachlässigbar erscheint.

4.2.2. Mattest

Vgl. Abbildung 7 auf Seite 20

Betrachtet man diesen Test, so findet man keine außergewöhnlichen Ergebnisse vor. Die Werte liegen im Mittel für die Maschinen zwischen $\bar{t}_{min} = 33,36s$ und $\bar{t}_{max} = 33,84s$ dementsprechend ist die Varianz und die Standardabweihung klein. Die Standardabweichung schwankt für die vier virtuellen Maschinen im Intervall $\sigma_{min} = 0,33s$ und $\sigma_{max} = 0,40s$. Das Histogramm (Abbildung 7(a)) verdeutlicht die sehr nahe am Mittelwert liegenden Werte.

4.2.3. IO

Vgl. Abbildung 8 auf Seite 21

Vgl. Abbildung 9 auf Seite 22

In diesem Test fallen zwei Phänomene auf. Einerseits sind die Laufzeiten bei diesem Test bimodal verteilt. Sowohl mit 256MB (Abbildung 8(a)) als auch mit 128MB (Abbildung 9(a)) 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 auch, dass die Laufzeiten bei 128MB kürzer sind. Eine Mutmaßung für dieses Verhalten ist ein weniger performantes Swap-Verhalten[2] im Falle von 256MB Hauptspeicher.

4.2.4. SeekAndWrite

Vgl. Abbildung 10 auf Seite 23

Die deutlich schwankenden Laufzeiten in Abbildung 10(a) sind vielleicht dadurch zu erklären, wenn man annimmt, dass die Laufzeiten abhängig von der Position in der Datei sind, 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

[2] Da in Prozessornähe nur kleine Arbeitsspeicher zur Verfügung stehen, müssen Daten ständig neu abgerufen und verschoben werden. Diese werden dann in andere Speichereinheiten verschoben, was man als „swap“ versteht

Testname	# Clients	Hauptspeicher	Iterationen	Konfiguration
TCP-Test	4	256MB	200	5000 Connects/Iteration
MatTest	4	256MB	1000	35 (Rekursionstiefe)
IO-Test	4	128-256MB	500	
SeekAndWrite-Test	4	256MB	200	

Tabelle 2: Auflistung der Konfigurationen

Zufallszahl bestimmt wird, variieren die Messwerte sehr. Betrachtet man die Ergebnisse in Abbildung 10(b), so fällt auf, dass die Gesamtlaufzeit summiert über alle Iterationen große Unterschiede aufweist. So beträgt der Unterschied zwischen langsamster und schnellster Laufzeit circa 435 Sekunden oder $1,4\%$ der Gesamtlaufzeit von rund 30000 Sekunden.

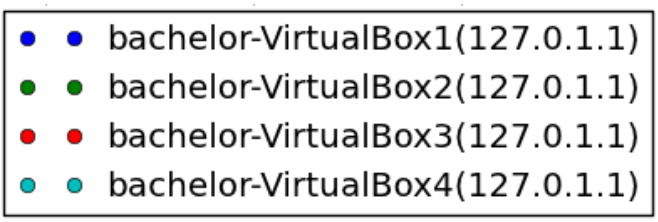
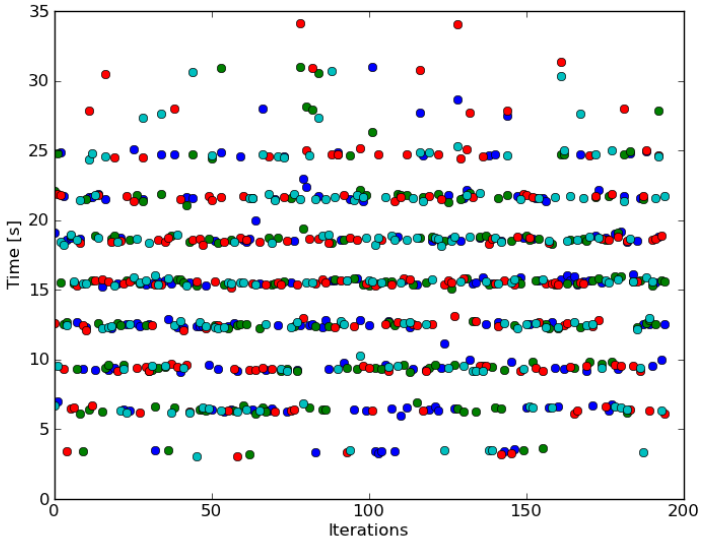
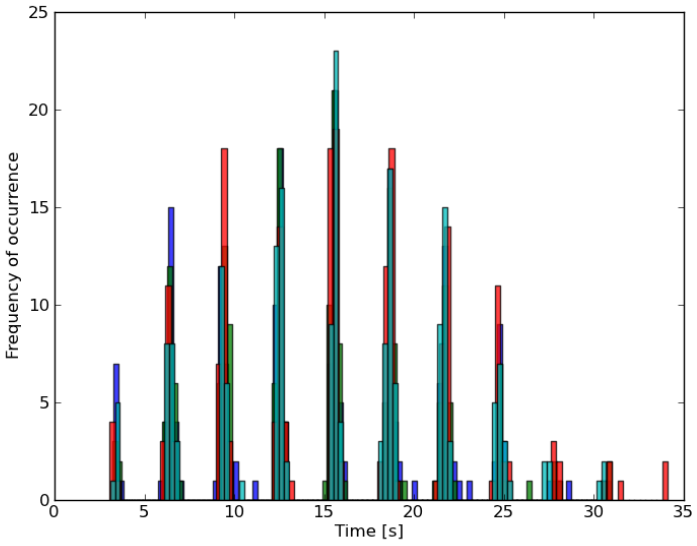


Abbildung 5: Legende für die Plots von Seite 19 bis 23

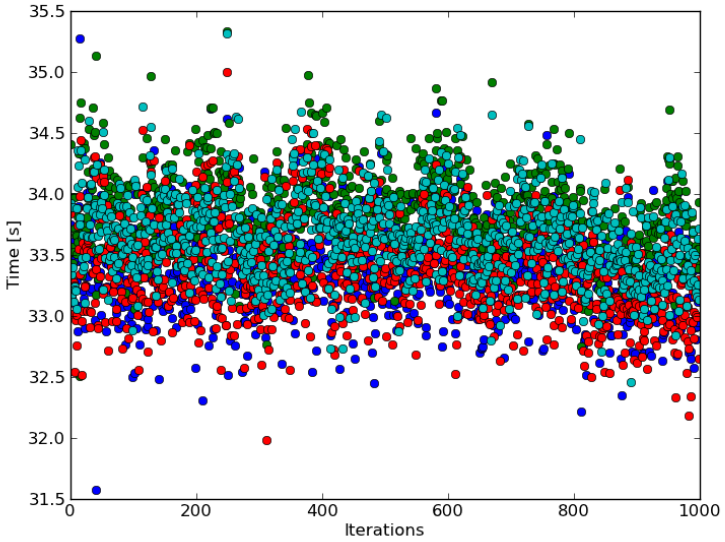


(a) Scatterplot eines TCP-Tests

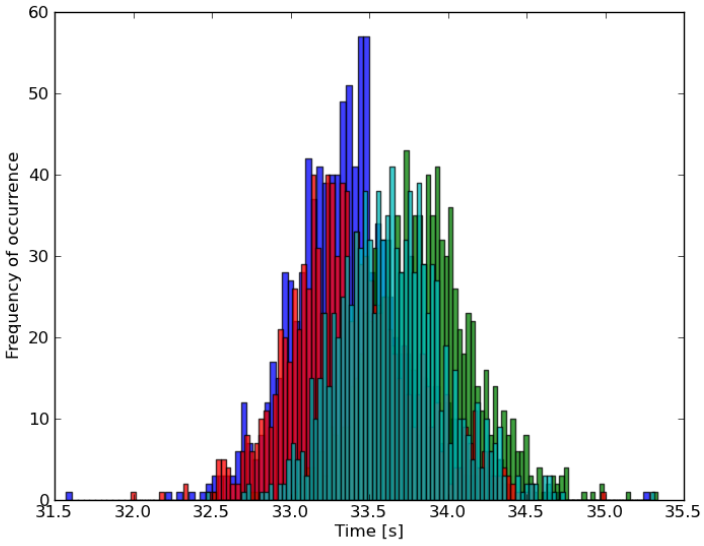


(b) Histogramm eines TCP-Tests

Abbildung 6: Ergebnisse des TCP-Tests

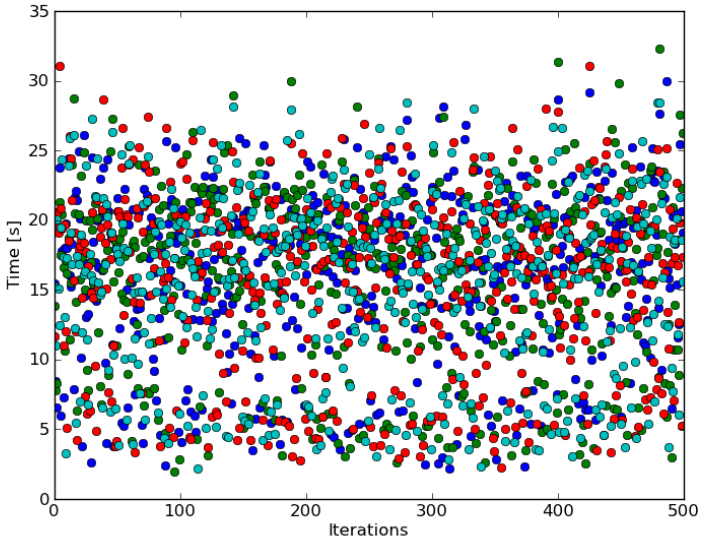


(a) Scatterplot eines Mattests

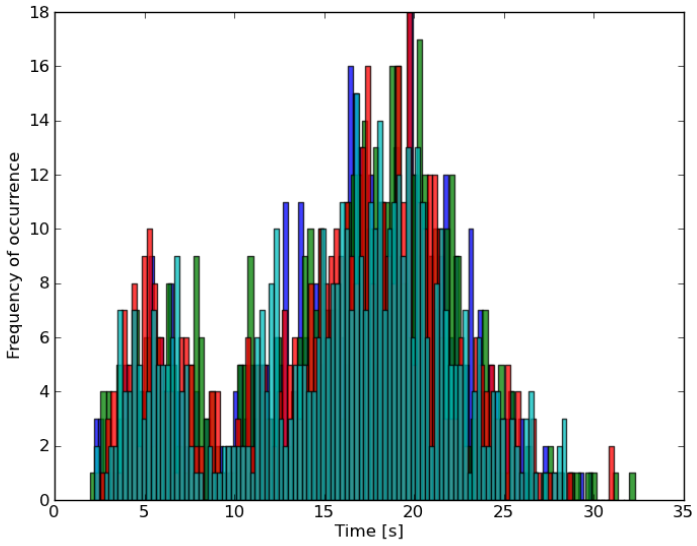


(b) Histogramm eines Mattests

Abbildung 7: Ergebnisse des Mattests

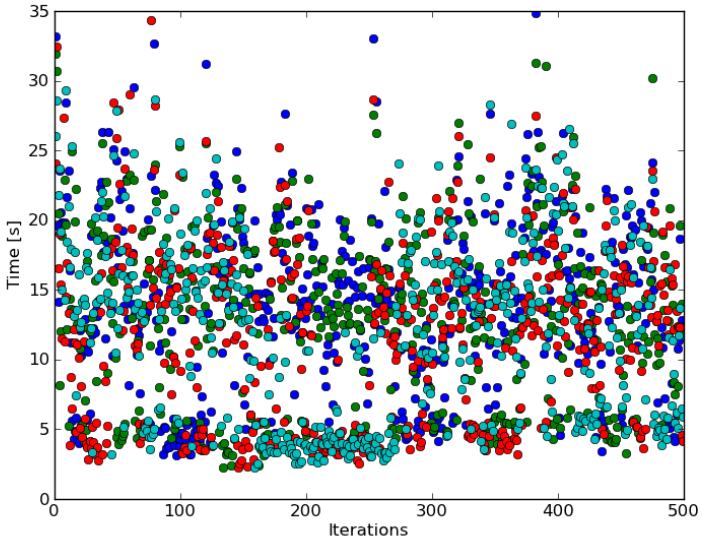


(a) Scatterplot eines IO-Tests (256MB Hauptspeicher)

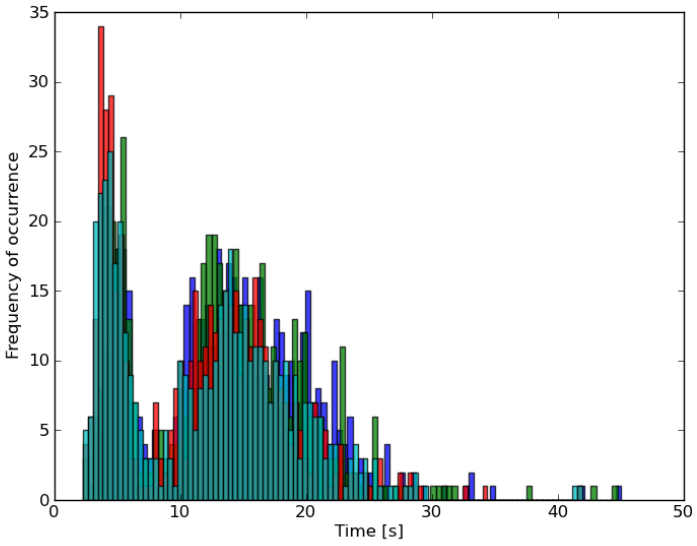


(b) Histogramm eines IO-Tests (256MB Hauptspeicher)

Abbildung 8: Ergebnisse des IO-Tests bei 256MB Hauptspeicher

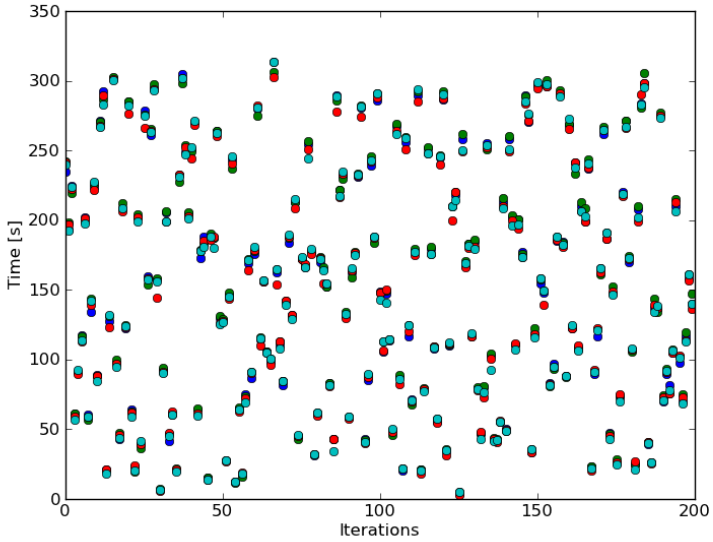


(a) Scatterplot eines IO-Tests (128MB Hauptspeicher)

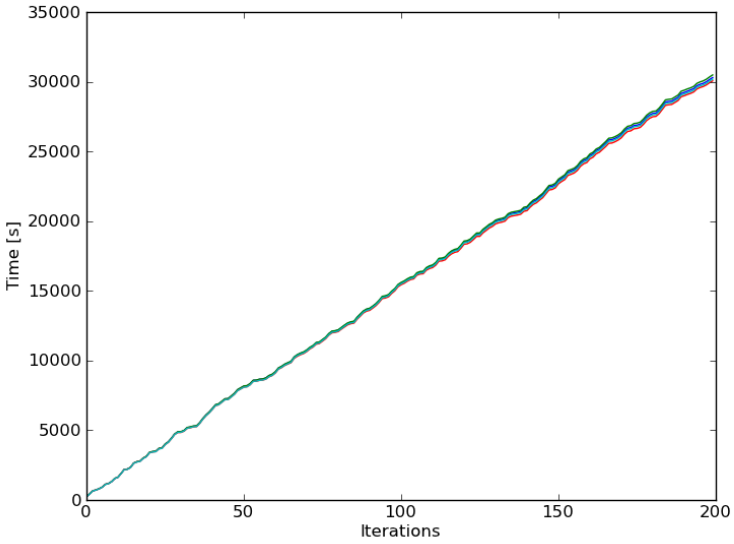


(b) Histogramm eines IO-Tests (256MB Hauptspeicher)

Abbildung 9: Ergebnisse des IO-Tests bei 128MB Hauptspeicher



(a) Scatterplot eines SeekAndWrite-Tests



(b) Kumulierte Laufzeit eines SeekAndWrite-Tests

Abbildung 10: Ergebnisse des SeekAndWrite-Tests

5. Zukunftspläne

5.1. Dokumentation

Um das Framework auch Außenstehenden zur Verfügung zu stellen, wäre eine Dokumentation außerhalb des Codes und dieser Arbeit wünschenswert. Es würde sich hierführ wahrscheinlich „Sphynx - Python Document Generator“ [17] anbieten, mit dem auch die reguläre Dokumentation von Python gemacht wird.

5.2. Messkampagnen und Messstrategien

In den bis jetzt ausgeführten Tests wurde die Größe des Hauptspeichers variiert. In anderen Messkampagnen könnte man genauso andere Hardwarekomponenten skalieren. Man könnte beispielsweise genauso die virtuelle Prozessorleistung variieren, beziehungsweise Restriktionen für den Netzwerkverkehr setzten, um herauszufinden, ob der VMM in diesen Fällen immer noch fair agiert. Es gibt auch die möglichkeit zu komplexen Messstrategien, bei denen man mehr als eine Komponente variabel hält.

5.3. Ansteuerung des Hypervisors

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

Neben den in der virtuellen Testumgebung durchgeführten Tests können auch Vergleichsmessungen im nativen System abgehalten werden. Eine Möglichkeit wäre für diesen Fall, analog wie bei den geklonten virtuellen Maschinen, baugleiche physische Maschinen zu verwenden. Auch das parallele Ablaufen der Tests auf dem nativen

und dem virtuellen System wäre denkbar. Man muss allerdings die Korrelation zwischen den Testergebnissen in den unterschiedlichen Systemen herausfinden, um die Messwerte vergleichbar zu machen.

6. Zusammenfassung

Abschließend muss man nochmals einige Dinge hervorheben. Die Vielfalt an Virtualisierung ist in den vergangenen Jahren sehr groß geworden und gipfelt in der Virtualisierung von ganzen Betriebssystemen. Zwar sind die Nachteile der Virtualisierung nicht zu vernachlässigen, aber die Überzeugung meinerseits ist groß, dass auch diese bald eine untergeordnete Rolle spielen. Die Anschaffungskosten werden mit zunehmender Verbreitung von virtuellen Maschinen geringer werden und darüber hinaus werden die VMM mit zunehmender Zeit stabiler und leistungsfähiger.

Mit Hilfe des Frameworks sollten Aussagen über die aktuelle Performance von Monitoren gemacht werden. Im Zug der Entwicklung des Frameworks wurde auch versucht den verwendeten Hypervisor zu bewerten, wobei das Ergbnis durchaus gut ist. Es kommt zwar zu Abweichungen bei den Laufzeiten zwischen den virtuellen Maschinen, aber sie sind eher selten in einem Außmaß bei dem man von eklatant sprechen kann. Auch muss man hier eine gewisse Messunschärfe berücksichtigen, die zum Beispiel durch ungenaue Zeitnehmung zustande kommen kann.

Auch wenn mit dem bis dato implementierten Framework vielleicht nur die berühmte „Spitze des Eisberges“ behandelt wird, so bleibt zu hoffen, dass durch Erweiterungen und detaillierte Ausarbeitungen einzelne VMM auch genauer analysiert werden können. Die Vielfältigkeit dieses Themas ist groß und dementsprechend gibt es auch einige Einsatzgebiete. Auch wird mit zunehmender Virtualisierung vielleicht der Bedarf an ebendiesen Messungen steigen, schließlich soll ja auch in virtuellen Instanzen ein faires Scheduling gewährleistet werden.

A. Eigene Testklasse erstellen

Um eine eigene Testklasse für das Framework zu erstllen, müssen einige Dinge befolgt werden, um vernünftige Ergebnisse zu erziehlen. 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('UserTest')
5
6          while True:
7              i = self.recieve()
8              if i[0] == self.const.iteration:
9                  self.start = time.time()
10                 UserTestFunction()
11                 self.dur = time.time() - self.start
12                 values = [str(i[1]), str(self.dur), str(self.start)]
13                 self.results.writerow(values)
14             if i[0] == self.const.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.const*: Eine Objekt der Klasse *Const*, um auf die Kommunikationsbefehele Zugriff zu haben.
  - *self.init*: Die Zeit gemessen via time.time() zur Zeit des Aufrufes von Zeile 4 *initTest()*

- *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 Vorschlg 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 dem Skript *reults.py* 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.

B. Lessons lerned

B.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 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.

B.2. Expect the unexpected

In einem sehr frühen Stadium des Frameworks wurden Tests durchgeführt. Das Ergbnis war ziemlich interessant, denn es scheinte 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 unerwartetes 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.

B.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 leicht 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.

Literatur

[1] D. Ongaro, A. L. Cox, and S. Rixner, eds., *Scheduling I/O in virtual machine monitors*, VEE '08, (New York and NY and USA), ACM, 2008.

[2] IBM Systems, "Virtualization (http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf)," 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] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI '05, pp. 273–286, USENIX Association, 2005.

[9] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues," *Computer and Network Technology, International Conference on*, vol. 0, pp. 222–226, 2010.

[10] D. Abramson, "Intel(R) Virtualization Technology for Directed I/O," *Intel Technology Journal*, vol. 10, no. 03, 2006.

[11] C. Weng, Z. Wang, M. Li, and X. Lu, eds., *The hybrid scheduling framework for virtual machine systems*, VEE '09, (New York and NY and USA), ACM, 2009.

[12] "Homepage Matplotlib (http://matplotlib.sourceforge.net/)."

[13] "Homepage Numpy (http://numpy.scipy.org/)."

[14] N. Igotti, "Python API to the VirtualBox VM (http://blogs.oracle.com/nike/entry/python_api_to_the_virtualbox)," 5.9.2008.

[15] N. Seddigh and M. Devetsikiotis, "Studies of TCP's retransmission timeout mechanism," in *Communications, 2001. ICC 2001. IEEE International Conference on*, vol. 6, pp. 1834 –1840 vol.6, 2001.

[16] J. Postel, "Transmission Control Protocol (http://www.ietf.org/rfc/rfc793.txt)," 1981.

[17] "Homepage Sphinx (http://sphinx.pocoo.org/)."