UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Institut für Informatik

Netzwerkvirtualisierung mit QEMU und VDE-Switches

Diplomarbeit

zur Erlangung des Grades eines
Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Christopher Israel

1. Betreuer: Prof. Dr. Christoph Steigner,
Institut für Informatik, AG Rechnernetze, Fachbereich 4: Informatik
2. Betreuer: Dipl. Inf. Frank Bohdanowicz,
Institut für Informatik, AG Rechnernetze, Fachbereich 4: Informatik

Koblenz, im September 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

Ort, Datum Unterschrift

Inhaltsverzeichnis

1. Einleitung

Diese Ausarbeitung wurde von Christopher Israel im Rahmen einer Diplomarbeit bei Herrn Prof. Dr. Christoph Steigner erstellt.

QEMU ist eine Open-Source Virtualisierungssoftware, durch die Computersysteme simuliert werden können. Durch *VDE* (*Virtual Distributed Ethernet*) ist es möglich, mehrere durch *QEMU* virtualisierte Computer miteinander zu verbinden und so ein virtuelles Netzwerk zu erzeugen. Mithilfe von Virtualisierung können Netze zum Testen von Netzwerkanwendungen und -protokollen erzeugt werden, deren Aufbau oder Nutzung ohne Virtualisierung impraktikabel oder schlicht unerschwinglich wäre.

Zum Zeitpunkt der Erstellung dieser Arbeit ist in der Arbeitsgruppe Rechnernetze zum Aufbau von virtuellen Netzen das Programm *VNUML*[1] (*Virtual Network User Mode Linux*) im Einsatz, welches zur Virtualisierung *UML* (*User Mode Linux*) verwendet.
VNUML wird jedoch zum Zeitpunkt der Erstellung der Diplomarbeit seit 2009 nicht mehr weiterentwickelt[37] (die letzte Version ist Version 1.8.9 vom 22.05.2009).
Längerfristig ist es daher notwendig, einen Ersatz für *VNUML* zu finden.

Ziel dieser Arbeit ist es, ein Programm zu entwickeln, welches eine in einer Szenario-Datei hinterlegte Netzwerktopologie mit *QEMU* und *VDE-Switches* aufbauen kann. Es soll ein Vergleich angestellt werden zwischen Netzen, die mit *QEMU* aufgebaut wurden und solchen, die über *VNUML* aufgebaut wurden.

2. Virtualisieren mit QEMU

2.1. QEMU und KVM

QEMU steht für „*Quick Emulator*“ und ist ursprünglich kein Virtualisierer, sondern ein Emulator. Er wurde 2005 von Fabrice Bellard entwickelt.[7] *QEMU* kann mehrere Prozessorarchitekturen emulieren, indem während der Laufzeit Instruktionen der VM-Architektur in Instruktionen des Hostsystems übersetzt werden („dynamic translation“).

Ein Virtualisierer hingegen führt die meisten Instruktionen einer virtuellen Maschine direkt auf dem Host aus. Die virtuelle Maschine muss dabei jedoch dieselbe Prozessorarchitektur besitzen wie der Host. Durch CPU-Erweiterungen, die Hardware-Virtualisierung ermöglichen, wie „Intel VT“[2] oder „AMD-V“[3] wird die Virtualisierung noch effektiver, da so ein Großteil der Instruktionen ohne Modifikation direkt an die CPU weitergeleitet werden kann.

[1] http://neweb.dit.upm.es/vnumlwiki/index.php/Main_Page
[2] http://www.intel.com/technology/virtualization
[3] http://sites.amd.com/de/business/it-solutions/virtualization/Pages/amd-v.aspx

QEMU kann durch Nutzung des *KVM*[4]-Kernelmoduls ebenfalls auf diese Techniken zugreifen. In Verbindung mit *KVM* wird *QEMU* zum Virtualisierer. Der Kernel des Hostsystems arbeitet mit dem *KVM*-Modul dann als Hypervisor und steuert die Ausführung der virtuellen Maschinen.

User Mode Linux[12][5] auf der anderen Seite ist ein Linux-Kernel, der als normale Linux-Anwendung kompiliert wurde. Unter Verwendung eines Host-Kernels, der mit dem SKAS-Patch[6] erweitert wurde, läuft eine VM auf Basis von User Mode Linux mit einer brauchbaren Geschwindigkeit.[1]

2.2. Erstellung eines Festplattenimages

Eine virtuelle Maschine benötigt zum Start eine virtuelle Festplatte in Form eines Festplattenimages, auf dem ein Betriebssystem installiert ist. Prinzipiell kann dazu jedes Betriebssystem verwendet werden. Um jedoch die virtuellen Maschinen in einer Simulation automatisiert konfigurieren zu können, ist es notwendig, dass die VMs über *SSH*[7] steuerbar sind. Zudem sollte das Betriebssystem im verwendeten Festplattenimage mit wenig Arbeitsspeicher auskommen, damit möglichst viele VMs gleichzeitig eingesetzt werden können.

Vorteilhaft ist es auch, wenn das verwendete Betriebssystem einen Paketmanager hat, sodass es leicht ist, die Software zu installieren, die innerhalb der Simulation benötigt wird.

Um schnell ein zum Testen verwendbares Festplattenimage zu erhalten wurde eine Festplatteninstallation einer Live CD[8] der *Grml*-Distribution[9] verwendet. *Grml* ist eine Debian-basierte Live CD, die sich an Systemadministratoren und Nutzer von Kommandozeilen-Programmen richtet.

Da das System standardmäßig im Textmodus startet, benötigt es verhältnismäßig wenig Arbeitsspeicher, was vorteilhaft ist, da so die Zahl der gleichzeitig betreibbaren VMs steigt. Nachteilig ist jedoch, dass selbst die kleinste Variante der Live CD noch viel Software mitbringt, wodurch die Größe des für QEMU benötigten Festplattenimages steigt. Zudem dauert der Bootvorgang im Vergleich zu dem von *VNUML* verwendeten Festplattenimage relativ lange.

Um einen aussagekräftigen Vergleich mit *VNUML* zu ermöglichen, müssen jedoch das Startscript und *VNUML* ähnliche Festplattenimages verwenden. Die *VNUML*-Distribution liefert unter der Bezeichnung „rootfs_tutorial“ ein Festplattenimage auf Basis der Debian-Distribution[10] mit[38], jedoch wurde zum Zeitpunkt der Erstellung dieser Arbeit seit der Version 0.6.0 im Mai 2009 kein neues Festplattenimage veröffentlicht.[37] Das Festplattenimage ist für die Benutzung mit *User Mode Linux* und *VNUML* gedacht und nicht direkt mit *QEMU* verwendbar. Es kann jedoch modifiziert werden, um auch innerhalb einer *QEMU*-VM lauffähig gemacht zu werden:

[4] *KVM* steht für „*Kernel-based Virtual Machine*“[4]
[5] http://user-mode-linux.sf.net
[6] Seperate Kernel Address Space: Prozesse innerhalb des User Mode Linux werden nicht innerhalb des Host-Betriebssystems geführt
[7] *Secure Shell* - Siehe Kapitel 2.6 auf Seite 12
[8] Eine Live CD ist ein Betriebssystem, das direkt von einer CD oder DVD gestartet und ohne Installation eingesetzt werden kann.
[9] http://grml.org/
[10] http://www.debian.org

Der Grund dafür, dass das Festplattenimage unter *QEMU* nicht brauchbar ist, liegt darin, dass einerseits der Eintrag für das Root-Verzeichnis `/` in der `/etc/fstab` des Festplattenimages auf Gerätedateien zeigt, die nur innerhalb eines User Mode Linux funktionieren, und andererseits daran, dass der mit dem Festplattenimage verwendete Linux-Kernel ein User Mode Linux-Kernel ist, der sich unter *QEMU* nicht verwenden lässt.

QEMU ermöglicht es jedoch ebenfalls, einen außerhalb des Festplattenimages liegenden Kernel anzugeben, um mit diesem ein im Festplattenimage installiertes Linux-System zu starten.[8] Auf diesem Image muss dann kein Kernel liegen, die zum externen Kernel passenden Module sollten allerdings in das Image kopiert werden.

Auf diese Weise lässt sich mit einem beliebigen Linux-Kernel ein für *VNUML* gedachtes Festplattenimage booten, jedoch sind weitere Veränderungen nötig, um das Festplattenimage tatsächlich sinnvoll unter *QEMU* benutzen zu können.

Zum einen verwendet *VNUML* ein zweites Festplattenimage mit einem Shellscript, um die VM nach dem Booten einzurichten. Dieses Image wird innerhalb der VM von einem `/dev/ubdb`-Device gemountet. Die `ubda`- und `ubdb`-Devices sind jedoch User Mode Linux spezifisch und existieren unter *QEMU* nicht.

```
1 proc /proc proc defaults 0 0
2 devpts /dev/pts devpts mode=0622 0 0
3 /dev/ubda / ext3 defaults 0 1
4 /dev/ubdb /mnt/vnuml iso9660 defaults 0 0
```

Quelltext 1: /etc/fstab im root_fs_tutorial 0.6.0

Zum anderen muss der Einhängepunkt des Root-Verzeichnisses in der `fstab`-Datei des Festplattenimages auf das Device geändert werden, unter dem in der *QEMU*-VM das Festplattenimage verfügbar gemacht wird. Wenn das Festplattenimage in der Befehlszeile des *QEMU*-Aufrufs mittels der Option `-hda DATEINAME` angegeben wurde, ist das passende Device innerhalb der VM aller Wahrscheinlichkeit nach `/dev/sda`.

Es ist bei Verwendung eines externen Kernels wichtig, dass dem Kernel über die Kernel-Befehlszeile mitgeteilt wird, auf welchem Device das Root-Verzeichnis liegt. Dies lässt sich über die `-append`-Option von *QEMU* bewerkstelligen[11].

Zudem kann es vorkommen, dass aufgrund von Einträgen in der `/dev/inittab`-Datei des Festplattenimages mehrere Shells[12] überlagert auf derselben virtuellen Konsole gestartet werden. Dies äußert sich insofern, dass zwar die Ausgabe einer Shell angezeigt wird, jedoch Benutzereingaben nicht an diese Shell, sondern an eine weitere Shell weitergeleitet werden, die im Hintergrund arbeitet. Dies erschwert es, die VM über die grafische Schnittstelle von *QEMU* zu bedienen.

[11] im Fall, dass das Root-Verzeichnis unter `/dev/sda` liegt, müsste also *QEMU* der Parameter `-append "root=/dev/sda"` übergeben werden

[12] Eine Shell, bzw. ein Kommandozeileninterpreter, ist ein Programm, das Eingaben vom Benutzer entgegennimmt und als Befehle ausführt. Ein unter Linux weit verbreiteter Kommandozeileninterpreter ist z. B. die bash (Bourne-again shell).

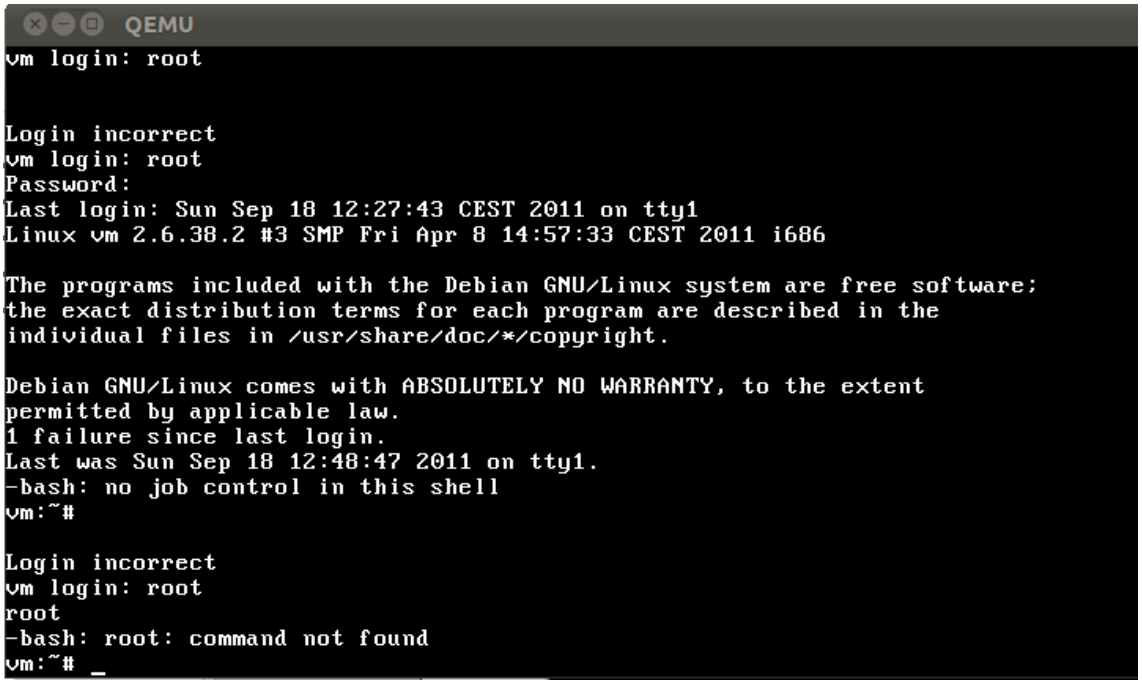

Abbildung 1: Screenshot, der die Überlagerung mehrerer Shells auf einer virtuellen Konsole zeigt

Es besteht neben der Bedienung der VM über die grafische Schnittstelle von *QEMU* noch die Möglichkeit, ein Management-Netz zwischen der VM und dem Hostrechner aufzubauen und über diese virtuelle Netzwerkverbindung die VM fernzusteuern (siehe hierzu auch Kapitel 2.4 auf Seite 9 und Kapitel 2.6 auf Seite 12). Da die Netzwerkschnittstellen der VM jedoch beim Start der VM nicht konfiguriert sind, ist vom Host aus kein Einloggen in die VM möglich.

Es muss daher entweder in Anlehnung an die Vorgehensweise von *VNUML* ein beim Start der VM ausgeführtes Shellscript[13] erstellt werden, durch welches die IP-Adresse des Interfaces zum Management-Netz gesetzt wird, oder es muss die `/etc/network/interfaces`-Datei abgeändert werden[14], sofern diese vorhanden ist, bzw. in der verwendeten Linux-Distribution benutzt wird.

2.3. Overlay-Images

Es kommt häufig vor, dass innerhalb einer Simulation mehrere virtuelle Maschinen gestartet werden müssen, die auf demselben Festplattenimage basieren.
Jedoch ist es nicht sinnvoll, mehrere VMs direkt von ein- und demselben Festplattenimage starten

[13] Ein Shellscript ist eine Datei, die Befehle für eine Shell enthält. Da viele Shells vollwertige Programmiersprachen enthalten, können mit einem Shellscript komplexe Vorgänge automatisiert werden.

[14] Die `/etc/network/interfaces`-Datei dient zur Konfiguration der Netzwerkinterfaces beim Aufruf der `ifdown`- oder `ifup`-Kommandos. Neben dem festen Einstellen der Adressen bzw. der automatischen Konfiguration der Interfaces über DHCP können in der Datei auch Kommandos angegeben werden, die vor oder nach der (De-)Aktivierung eines Interfaces ausgeführt werden sollen.[19]

zu lassen, da so jede VM Schreibzugriff auf das Image hat und es unweigerlich zu Dateisystemfehlern in diesem kommt.

Um mehrere virtuelle Maschinen vom selben Festplattenimage starten zu können, muss also für jede VM ein Overlay-Image erstellt werden.

Ein Overlay-Image ist ein Festplattenimage, das nur die Abweichungen von einem anderen, als Basis verwendeten Festplattenimage festhält. Dabei wird das Basis-Image nicht verändert, sondern Änderungen per Copy on Write im Overlay-Image vermerkt.

Durch die Verwendung von Overlay-Images sind die VMs getrennt und es wird nicht so viel Festplattenspeicher verbraucht, als wenn für jede VM eine vollständige Kopie des Basis-Images angefertigt würde.

Konkret wird ein Overlay-Image angelegt, indem das Kommando `qemu-img create -b` *`FESTPLATTENIMAGE`* `-f qcow2` *`OVERLAYIMAGE`* ausgeführt wird. Durch die Kommandozeilenoption `-f qcow2` wird dabei angegeben, dass das Overlay-Image im QCOW2-Format[24] anzulegen ist, die Option `-b` *`FESTPLATTENIMAGE`* legt das Basis-Image fest.

Beim Erstellen des Overlay-Images ist jedoch zu beachten, dass der Pfad zum Basis-Festplattenimage, wenn er bei der Erstellung als relativer Pfad[15]angegeben wurde, er auch beim späteren Zugriff als relativer Pfad verwendet wird. Wird das Overlay-Image nach der Erstellung in einen anderen Ordner verschoben, so muss auch das zum Overlay-Image gehörige Basis-Image verschoben - oder zumindest ein Link im Zielordner erzeugt werden. Wenn Overlay-Images erstellt werden, die nicht von einem „vollständigen“ Festplattenimage abhängen, sondern von einem anderen Overlay-Image, so muss ein solcher Link rekursiv für jedes referenzierte Festplattenimage erstellt werden. Alternativ ist es möglich, beim Erstellen des Overlay-Images den Pfad zum verwendeten Basis-Image als absoluten Pfad anzugeben.

Das QCOW2-Format der Festplattenimages erlaubt es auch, sogenannte „Snapshots“ anzulegen. Ein Snapshot hält den Zustand des Festplattenimages zu einem Zeitpunkt fest. Durch einen Snapshots ist es jederzeit möglich, den Inhalt des Festplattenimages auf den Stand zurückzusetzen, der bei der Erstellung des Snapshots vorzufinden war. Ebenso ist es möglich, aus einer laufenden *QEMU*-VM heraus den Ausführungszustand der VM in einen Snapshot des verwendeten Festplattenimages zu speichern.

Beim Erstellen eines Overlay-Images vom einem Festplattenimage, welches Snapshots enthält, werden jedoch diese Snapshots nicht in das Overlay-Image übertragen. Es ist also nicht möglich, einen gespeicherten Zustand im Basis-Image anzulegen (z. B. durch Starten einer VM direkt vom Basis-Image), Overlay-Images von diesem Basis-Image zu erzeugen, und dann über ein so erzeugtes Overlay-Image eine VM in den gespeicherten Zustand zu laden, weil dieser Zustand in den Overlay-Images nicht mehr vorhanden ist.

Soll beim Start der VM ein gesicherter VM-Zustand geladen werden, so ist die sinnvollste Methode dazu, den Zustand in einem Overlay-Image zu speichern und von diesem Overlay-Image

[15]Pfade können *absolut* oder *relativ* angegeben werden. Ein *absoluter* Pfad ist der vollständige Pfad zu einer Datei, z. B. */usr/local/lib/libvdeplug.so*. *Relative* Pfade hingegen geben den Pfad einer Datei relativ zu einem Verzeichnis an. Der *relative* Pfad zu der zuvor angebenen Datei wäre aus dem Verzeichnis */usr/local/* heraus z. B. *lib/libvdeplug.so*.

eine Kopie für jede VM anzufertigen. Abhängig vom verwendeten Dateisystem und der Größe des Overlay-Images dauert das Kopieren jedoch bereits länger als das Anlegen eines neuen Overlay-Images und das normale Starten der VM (ohne Laden des gespeicherten Zustands).

2.4. Netzwerk-Möglichkeiten von QEMU

QEMU bietet mehrere Möglichkeiten an, eine VM mit einem reellen oder virtuellen Netzwerk zu verbinden. Hierbei wird zwischen Netzwerk-Backends, „VLANs“ und Netzwerk-Interfaces (virtuellen Netzwerkkarten) unterschieden. „VLANs“ im *QEMU*-Kontext sind eigentlich keine VLANs[16]. Sie bezeichnen nur Nummern, die den Netzwerk-Backends und -Interfaces einer VM zugeordnet werden, um zu bestimmen, welche Netzwerke und Interfaces miteinander verbunden sind. In der *QEMU*-Manpage[8] werden dabei fünf Typen von Netzwerk-Backends beschrieben, die *QEMU* unterstützt:

Usermode-Netzwerk: Hierbei agiert der *QEMU*-Prozess als NAT-Router[17], welcher der VM den Zugang zum physikalischen Netzwerk des Hostrechners ermöglicht. Die VM erhält ihre Netzwerk-Konfiguration über DHCP[18], ist aber vom Host-Rechner nicht direkt, sondern über Portweiterleitungen erreichbar.

TAP-Netzwerk: Hierbei wird die VM an ein TUN/TAP-Device[19] auf dem Host-Rechner angebunden. Es muss jedoch dafür gesorgt werden, dass der *QEMU*-Prozess die notwendigen Berechtigungen hat, um auf das Device zugreifen zu dürfen.

Socket-Netzwerk: Zwei *QEMU*-VMs können über ein Socket miteinander verbunden werden. Hierzu muss eine der VMs als Server agieren und den Socket erzeugen und die andere VM muss zu dem Socket eine Verbindung aufbauen. Über Multicast-Sockets[20] oder die Verbindung mehrerer Netzwerke innerhalb einer VM können so auch mehrere *QEMU*-VMs miteinander vernetzt werden.

Dump-Netzwerk: Schneidet den gesamten Traffic an einem *QEMU*-VLAN in eine Datei mit.

VDE-Netzwerk: Verbindet eine *QEMU*-VM an einen Port eines *VDE-Switches*.

[16] Virtual Local Area Networks stellen eine Möglichkeit dar, ein physikalisches lokales Netzwerk (LAN - Local Area Network) in mehrere Teilnetze aufzutrennen.[27]

[17] NAT (Network Address Translation) ist eine Technologie, durch die Adressen von einem lokalen Adressbereich in einen Adressen aus einem globalen Adressbereich übersetzt werden können. NAT hat den Nachteil, dass die Hosts im lokalen Adressbereich keine global eindeutige Adresse mehr haben und somit von außerhalb nicht direkt erreichbar sind[28]

[18] DHCP (Dynamic Host Configuration Protocol) dient zur automatischen Konfiguration der Netzwerk-Einstellungen von Hosts in einem Netzwerk. Die notwendigen Einstellungen werden dabei zentral in einem DHCP-Server festgelegt und durch DHCP-Clients auf den Hosts abgerufen.[26]

[19] TUN/TAP-Devices sind virtuelle Netzwerkschnittstellen, die durch ein Kernelmodul des Hostsystems bereitgestellt werden und es ermöglichen, eine Netzwerkverbindung zu einer VM zu simulieren.[22]

[20] Durch Multicast ist es möglich, eine Kommunikation zwischen einer Gruppe von Teilnehmern zu realisieren. Pakete, die an eine Gruppen-Adresse geschickt werden, werden an alle Mitglieder der Gruppe verteilt. Um einer Multicast-Gruppe beizutreten, muss ein Host das IGMP (Internet Group Management Protocol) verwenden.[29]

2.5. Scriptgesteuerte Konfiguration der VMs

Um eine VM zu steuern ist es notwendig, eine Management-Schnittstelle zu haben, welche die VM mit dem Host verbindet. Es gibt hierzu mehrere Möglichkeiten:

- über ein beim Start der VM ausgeführtes Shellscript
- über einen Kommunikationskanal zwischen Host und VM:
  - über eine serielle Schnittstelle der VM
  - über die „mconsole“ (nur für Usermode Linux)
  - mit SSH, über eine Netzwerkverbindung:
    - „user mode network stack“ von QEMU
    - TUN/TAP-Devices

Die erste Möglichkeit besteht darin, ein Shellscript zu erstellen, welches während des Bootvorganges der VM ausgeführt wird. Unter Linux muss dazu üblicherweise das Shellscript oder ein Link dazu unter `/etc/rc2.d` abgelegt werden (wenn beim Start des Systems der Runlevel 2 gestartet wird). Diese Methode wird von *VNUML* in Kombination mit einem zweiten Image genutzt, das vor dem Start der VM erzeugt wird und das passende Shellscript enthält.

Zur Benutzung des beim Booten der VM gestarteten Shellscriptes müssen ein Einhängepunkt für das Image angelegt und der Symlink auf das Script erzeugt werden. Ein solches Shellscript benötigt keine Netzwerkverbindung zwischen virtueller Maschine und Hostrechner, jedoch müssen die vom Script ausgeführten Befehle vor dem Start der VM festgelegt werden.

Die zweite Möglichkeit besteht darin, über einen Kommunikationskanal Befehle an die VM zu schicken, nachdem diese gestartet wurde.
In Frage kommen dazu einerseits eine Netzwerkschnittstelle über die eine *SSH*-Verbindung aufgebaut wird oder andererseits eine der seriellen Schnittstellen der VM, auf denen mit `getty` VM-seitig eine serielle Konsole gestartet werden kann. Dazu müssen jedoch Einstellungen im Festplattenimage der VM getroffen werden, sodass beim Start der VM auf deren serieller Schnittstelle eine Shell gestartet wird. Dies ist üblicherweise durch Einträge in der `/etc/inittab`-Datei oder durch eine Konfigurationsdatei unter `/etc/init/` möglich. Auf der Hostseite können dann Befehle an die virtuelle serielle Schnittstelle der VM geschickt werden. Es muss beim Senden von Befehlen darauf geachtet werden, dass diese im Hintergrund ausgeführt werden, da ansonsten die Ausführung weiterer Befehle blockiert wird.

VNUML kann auch die „mconsole“ des User Mode Linux-Kernels benutzen, um auf der Gastseite Befehle auszuführen. Die mconsole ist allerdings nur bei User Mode Linux verwendbar und kann somit bei *QEMU* nicht eingesetzt werden.

Zum Aufbau der *SSH*-Verbindung müssen die Schlüssel zwischen Host- und Gastrechner ausgetauscht werden (siehe 2.6.1 auf Seite 12). Der Nachteil besteht darin, dass über SSH keine

Konfiguration der VM und ihrer Netzwerkschnittstellen möglich ist, solange keine Netzwerkverbindung zum Hostrechner besteht.

QEMU bietet von Haus aus einen „user mode network stack“ auf Basis von *SLIRP*[33], der jedoch keine direkte Verbindung zwischen Host und VM herstellt. Zunächst erhält nur die VM Zugriff auf den Host-Rechner und dessen Netzwerkverbindungen. Der Host kann auf die VM nur indirekt über Portweiterleitungen zugreifen, die beim *QEMU*-Aufruf in den Kommandozeilenargumenten eingerichtet werden können. Durch einen in *QEMU* integrierten DHCP-Server wird die Netzwerkschnittstelle der VM automatisch konfiguriert.

Alternativ bietet sich an, für jede VM ein TUN/TAP-Device zu erzeugen und über diese mit den VMs zu kommunizieren. Bei dieser Lösung sind die VMs direkt vom Host über ihre IP-Adressen erreichbar. Wenn für jede VM einen Eintrag in der hosts-Datei[21] des Systems erstellt wird, kann der Benutzer die VMs auch über ihre Hostnamen ansprechen.

Der Nachteil an dieser Lösung ist, dass sowohl zum Erstellen eines TUN/TAP-Devices sowie zum Benutzen desselben die Berechtigung des root-Benutzers[22] notwendig ist. Darüber hinaus muss dafür gesorgt werden, dass die Netzwerkschnittstelle der VM konfiguriert wird. Bei Verwendung des „user mode network stacks“ entfällt dieser Schritt und es werden keine erweiterten Benutzerrechte benötigt.

Die im Startscript verwendete Variante zur Steuerung der VMs ist daher eine *SSH*-Verbindung über den „user mode network stack“. Auf diese Weise ist jedes VM-System verwendbar, durch das die automatische Konfiguration von Netzwerkschnittstellen über DHCP unterstützt wird und auf dem ein SSH-Server läuft.

Um sich mit *SSH* Zugang zu den einzelnen VMs zu verschaffen, reicht es aus, die in *QEMU* eingebaute Netzwerk-Variante (den „user mode network stack“) zu verwenden und eine Portweiterleitung auf den *SSH*-Port der VM einzurichten. Um beispielsweise eine Weiterleitung vom Port 2222 des Host-Systems auf den *SSH*-Port eines Gastsystems einzurichten, muss bei der Einrichtung des Netzwerk-Interfaces die `hostfwd`-Option angegeben werden:

```
[...]  -net user,hostfwd=tcp::2222-:22   [...]
```

Dabei ist zu beachten, dass auch eine passende virtuelle Netzwerkkarte erzeugt und mit dem User-Netzwerk verbunden werden muss. Dazu muss die Kommandozeilenoption `-net nic` verwendet und dem Netzwerk und der virtuellen Netzwerkkarte dieselbe VLAN-Nummer zugewiesen werden, womit *QEMU* mitgeteilt wird, dass das Netz und das Interface miteinander verbunden sind (siehe hierzu auch Kapitel 2.7.2 auf Seite 14).

Bei Verwendung des „user mode network stacks“ muss nicht, wie bei den anderen Netzwerkschnittstellen, darauf geachtet werden, dass die von den Netzwerkkarten verwendeten MAC-Adressen[23] sich voneinander unterscheiden. Dies liegt daran, dass die für die Management-

[21] Durch die hosts-Datei eines Systems (unter Linux */etc/hosts*, unter Windows z. B. *C:\Windows\system32\drivers\etc\hosts*) können Hostnamen statisch in IP-Adressen aufgelöst werden.[36]

[22] Unter Linux besitzt der root-Benutzer die vollen Zugriffsrechte auf das System. Er entspricht dem Administrator-Konto auf einem Windows-System.

[23] MAC-Adressen (Media Access Control Adressen) sind die auf Schicht 2 des OSI-Modells angesiedelten Hardware-Adressen der Netzwerkschnittstellen.[43]

Schnittstellen verwendeten Netzwerkkarten nur jeweils für die VM und ihren zugehörigen *QEMU*-Prozess sichtbar sind und über die Management-Schnittstelle keine Verbindung zwischen den VMs untereinander besteht.

2.6. Secure Shell

SSH steht für „Secure Shell“ und steht für ein Programmpaket, das es ermöglicht, über eine verschlüsselte Verbindung auf einem entfernten Rechner eine Shell zu bedienen.

Neben der Fernsteuerung einer Shell im interaktiven Modus ist es auch möglich, auf der Gegenseite im nicht-interaktiven Modus beliebige Programme zu starten oder Netzwerkverbindungen über den verschlüsselten Kanal zu tunneln. Interessant ist hier vor allem die Möglichkeit, über das Netzwerk Programme auszuführen.

2.6.1. Einrichten des SSH-Zugangs

Damit ein Script per *SSH* Befehle auf einer VM ausführen kann, ohne dass bei jedem Befehl ein Passwort eingegeben werden muss, kann die Public-Key-Authentifizierung von *SSH* benutzt werden.

SSH unterstützt neben der Authentifizierung des Benutzers über die Eingabe eines Passworts unter anderem auch die Authentifizierung über ein Public-Key-Verfahren.[34][47] Üblicherweise wird bei dieser Variante der Authentifizierung auf dem Client-Rechner mit `ssh-keygen` ein Schlüsselpaar angelegt, aus dem der öffentliche Schlüssel auf den Server-Rechner (in diesem Fall der VM) kopiert wird und dort in die *`authorized_hosts`*-Datei eingetragen wird.

Sofern die Public-Key-Authentifizierung in der Konfiguration des *SSH*-Servers aktiviert ist, kann sich anschließend über *SSH* in den Server-Rechner eingeloggt werden, ohne dass ein Passwort eingegeben werden muss.

Ein Problem stellt noch die *`known_hosts`*-Datei dar:

Um Man-in-the-middle-Angriffe zu vermeiden, überprüft der *SSH*-Client bei jeder Verbindung zu einem *SSH*-Server, ob dessen öffentlicher Schlüssel in der lokalen *`known_hosts`*-Datei abgelegt ist. Ist der *SSH*-Server dem Client noch unbekannt, so wird nach einer Sicherheitsabfrage an den Benutzer die Kombination aus Benutzername, Adresse und Schlüssel in die *`known_hosts`*-Datei eingetragen.
Gibt es zu der Adresse des *SSH*-Servers passende Einträge in der *`known_hosts`*-Datei, passt aber der öffentliche Schlüssel des *SSH*-Servers zu keinem dieser Einträge, so verhindert der *SSH*-Client, dass eine Verbindung aufgebaut wird.
Ausschließlich, wenn der öffentliche Schlüssel des *SSH*-Servers mit einem Eintrag in der *`known_hosts`*-Datei übereinstimmt, kann eine Verbindung zustande kommen.

Folglich muss auf der VM ein Host-Key erzeugt und der öffentliche Teil dieses Schlüssels auf der Seite des *SSH*-Clients in die lokale *`known_hosts`*-Datei eingetragen werden. Da beim Betrieb von

mehreren VMs diese auch verschiedene Adressen haben, muss beim Eintrag in die *known_hosts*-Datei im Feld für die Adresse ein Wildcard verwendet werden. Dadurch wird dafür gesorgt, dass jeder *SSH*-Server akzeptiert wird, dessen Host-Key zum Eintrag passt, unabhängig von dessen Adresse.

Dennoch kann es bei der Verwendung des Startscriptes in Verbindung mit der *known_hosts*-Datei zu Problemen kommen: Ist in der *known_hosts*-Datei des Benutzers bereits ein Eintrag, der jedoch nicht mit dem Hostschlüssel der VM übereinstimmt, so kommt keine Verbindung zustande und die VM kann nicht automatisch konfiguriert werden.
Die Lösung des Problems ist es, eine alternative *known_hosts*-Datei an einer anderen Stelle anzulegen, die nur den Hostschlüssel der VM beinhaltet. Auf diese Weise kann es nicht zu Konflikten mit der *known_hosts*-Datei eines Benutzers kommen. Die Kommandozeilenoption[48], die dem *SSH*-Client dazu übergeben werden muss, lautet:

```
-oUserKnownHostsFile=PFAD_ZUR_KNOWNHOSTS_DATEI
```

Um nicht für jeden Rechner, auf dem die *QEMU*-VM ausgeführt werden soll, dessen öffentlichen Schlüssel in der *authorized_hosts*-Datei des Festplattenimages eintragen zu müssen, kann dem *SSH*-Client auch ein alternativer Schlüssel angegeben werden.

Dies ist mit der Kommandozeilenoption `-i DATEINAME` möglich. *DATEINAME* gibt hier den Dateinamen des privaten Teils des Schlüsselpaars an. Es muss darauf geachtet werden, dass der private Schlüssel nur vom Benutzer lesbar ist, der den *SSH*-Client ausführt (z. B. mit `chmod 600 DATEINAME`). Ist das nicht der Fall, bricht *SSH* mit einer Fehlermeldung ab.

Die endgültige Kommandozeile lautet dann:

```
ssh -p2222 -oUserKnownHostsFile=./knownhosts -i hostkey root@localhost
```

2.7. VDE-Switches

2.7.1. Besonderheiten von VDE-Switches

Die *VDE-Switches* gehören zum *VDE*-Paket. *VDE* steht für „*Virtual Distributed Ethernet*“. Es ist Teil des VirtualSquare-Projektes der University of Bologna, dessen Ziel es ist, eine einheitliche Umgebung für die Interaktion zwischen (virtuellen) Maschinen und Netzen zu schaffen.[31] *VDE-Switches* sollen es primär ermöglichen, virtuelle Maschinen auch über physikalische Netzwerke hinweg miteinander zu vernetzen.

Ein *VDE-Switch* ist eine vollständige Simulation eines Netzwerk-Switches, der auch das Spanning-Tree Protocol (STP) verwendet und so ermöglicht, mehrere Switches miteinander zu vernetzen, ohne befürchten zu müssen, dass der Netzwerkverkehr durch Schleifen zum Erliegen kommt.

Es existieren vielfältige Möglichkeiten der Verbindung von *VDE-Switches*: Sie können lokal innerhalb eines Rechners vernetzt werden, es können aber auch zwei auf verschiedenen Rechnern laufende *VDE-Switches* über eine Netzwerkverbindung, z. B. über *SSH*, verbunden werden. Es

kann auch ein TUN/TAP-Device an einen Switch angebunden werden, um Pakete direkt von einer Host-Maschine an das vom Switch getragene Netzwerk zu schicken.

Der Switch-Prozess stellt eine Management-Konsole zur Verfügung, über die (durch Verwendung eines Plugins) der über den Switch laufende Netzwerkverkehr mitgeschnitten, angeschlossene Netzwerkteilnehmer vom Switch getrennt oder neue Netzwerkteilnehmer an den Switch angeschlossen werden können. Zudem unterstützt der Switch VLANs, die über die Management-Konsole konfiguriert werden können.

Auf die Management-Konsole des Switches kann entweder durch direkten Aufruf des Switch-Programmes von der Shell oder durch Verwendung eines UNIX-Sockets zugegriffen werden. Siehe hierzu auch Kapitel 2.9 auf Seite 16.

Ebenfalls zum VirtualSquare-Projekt gehört das Werkzeug `wirefilter`, das als Verbindung zwischen zwei *VDE-Switches* fungiert und die unter anderem die Simulation von Paketverlusten, Verzögerungen und beschädigten Paketen ermöglicht.[32]

2.7.2. Anbindung an VDE-Switches

Bevor eine VM zu einem *VDE-Switch* verbunden werden kann, muss dieser zunächst gestartet werden. Hierzu wird folgender Aufruf verwendet:

`vde_switch --sock` *`SOCKETNAME`* `-M` *`MGMTSOCKET`* `--pidfile` *`PIDFILE`*

Hierbei wird mit *`SOCKETNAME`* ein Pfad angegeben, unter dem dann vom *VDE-Switch*-Prozess ein Verzeichnis erstellt wird, welches die Kommunikations-Sockets des Switches enthält. Dieser Pfad muss allen VMs, die an den Switch angebunden werden sollen, übergeben werden.

`MGMTSOCKET` bezeichnet den Dateinamen, unter dem der Switchprozess einen Socket anlegt, an dem die Managementkonsole des Switches verfügbar gemacht wird.

Mit der Option `-pidfile` *`PIDFILE`* wird angegeben, dass der *VDE-Switch* beim Start seine Prozessnummer in die Datei *`PIDFILE`* schreibt.

Mehrere *QEMU*-VMs können über *VDE-Switches* verbunden werden. Hierfür muss beim *QEMU*-Aufruf die Kommandozeilenoption `-net vde` mit dem Kommunikations-Socket des anzuschließenden *VDE-Switches* angegeben werden. Um jedoch das auf diese Weise angeschlossene Netzwerk innerhalb der VM nutzen zu können, muss zusätzlich eine virtuelle Netzwerkkarte erzeugt werden. Dies geschieht durch die Kommandozeilenoption `-net nic`.

Sollen mehrere Netze an eine VM angeschlossen werden, werden in dieser VM auch mehrere virtuelle Netzwerkkarten benötigt. Durch das `vlan`-Argument kann dann festgelegt werden, welche Netze mit welchen Netzwerkkarten zu verbinden sind.

Es ist wichtig, dass den virtuellen Netzwerkkarten der VMs verschiedene MAC-Adressen zugeteilt werden. Da die MAC-Adressen jeder *QEMU*-Instanz im Normalfall bei `52:54:00:12:34:56` anfangen und inkrementiert werden, kommt es beim Verbinden mehrerer VMs zu Adresskolli-

sionen. Die MAC-Adressen müssen daher über das `macaddr`-Argument der `-net nic`-Option explizit festgelegt werden.

Zwar unterstützt *QEMU* die Anbindung an *VDE*-Switches, jedoch ist diese Unterstützung in den offiziellen Paketen mancher Linux-Distributoren, wie z. B. Ubuntu, nicht mit einkompiliert. Daher muss entweder der Wrapper `vdeqemu` verwendet, oder *QEMU* mit der VDE-Unterstützung neu kompiliert werden.

`vdeqemu` ist eine symbolische Verknüpfung zu `vdeq`. Es ist ein Wrapper, der den Aufruf von `qemu` bzw. `kvm` so umschreibt, dass alle Vorkommen von `-net vde` ersetzt werden durch `-net tap`, wobei kein TAP-Device, sondern ein Filedeskriptor übergeben wird, an dem der `vdeq`-Prozess die Schnittstelle zum jeweiligen *VDE-Switch* bereitstellt.

Um *QEMU* neu zu kompilieren, muss `configure` mit den Optionen `-enable-kvm` und `-enable-vde` aufgerufen werden.

2.8. Überwachung des Startvorgangs einer VM

Um mit einer VM arbeiten zu können, muss diese ihren Startvorgang abgeschlossen haben. Während ein Benutzer zu diesem Zweck den (virtuellen) Bildschirm der VM betrachten und so den Startvorgang überwachen kann, muss bei einem Script eine andere Methode gefunden werden.

Ein Script muss, um über *SSH* Kommandos auf einer VM ausführen zu können, sicherstellen, dass diese bereits gestartet und erreichbar ist. Wird versucht, über SSH einen Befehl auf einer VM auszuführen, die noch nicht vollständig gestartet ist, ist es möglich, dass der SSH-Aufruf durch einen Timeout abbricht. Der auszuführende Befehl wird dadurch nicht mehr ausgeführt.

Zudem soll nach Beendigung des Scriptes dem Benutzer ein laufendes System übergeben werden.

Als Lösung dieses Problems wurde folgende Vorgehensweise gewählt:

Nach dem Start aller VM-Prozesse wird versucht, zu jeder VM eine *SSH*-Verbindung aufzubauen. Beendet sich ein *SSH*-Prozess dabei mit einem Fehler, wird er erneut gestartet.

Nach dem dreimaligen Auftreten eines Fehlers oder dem Ablauf eines vor dem Start festgelegten Timeouts wird dem Benutzer mitgeteilt, welche der startenden VMs nicht reagieren.
Es wird eine Abfrage an den Benutzer gestellt, ob weiter versucht werden soll, Verbindungen zu den nicht antwortenden VMs herzustellen, ob der Startvorgang abgebrochen oder ohne weitere Überprüfungen fortgesetzt werden soll.

Durch diese Vorgehensweise kann der Benutzer auch schnell feststellen, wenn beispielsweise durch die Konfiguration ein Fehler aufgetreten ist. Vor allem, wenn eine VM gar nicht gestartet wurde ist diese Funktion sehr praktisch, da in diesem Fall der entsprechende *SSH*-Port nicht geöffnet ist und ein *SSH*-Prozess sofort abbricht. Der Benutzer kann so schnell feststellen, ob eine VM nicht gestartet werden konnte.

2.9. Management-Konsole/Monitor

QEMU und *VDE-Switches* haben Management-Konsolen oder Monitore, die es ermöglichen, während der Laufzeit Einstellungen an der VM bzw. am virtuellen Switch zu verändern.

Gewöhnlicherweise werden diese Konsolen dem Nutzer über die Standard-Ein/Ausgabe zugänglich gemacht. Da die Programme jedoch über ein Script im Hintergrund gestartet werden, sind die Ein-/Ausgabestreams und damit auch die Management-Interfaces nicht mehr zugänglich. *VDE-Switches* und *QEMU* bieten jedoch auch die Möglichkeit, die Management-Konsole auf ein UNIX Domain Socket zu legen.

Ein UNIX Domain Socket ist eine Datei im Dateisystem, die zwei Prozessen ermöglicht, Daten auszutauschen. UNIX Sockets sind bidirektional und können über eine Socket-API angesprochen werden.

Wenn die Management-Konsole des Programms an *MANAGEMENTSOCKET* gebunden ist, kann der Benutzer mit dem Befehl `unixterm` *MANAGEMENTSOCKET* mit der Management-Konsole interagieren. Für *MANAGEMENTSOCKET* muss der vollständige Pfad der Datei angegeben werden, oder der Befehl muss in dem Verzeichnis ausgeführt werden, in dem sich die Datei befindet. Das Programm `unixterm` ist Teil des *VDE*-Projekts und verbindet die Ein/Ausgabe des Benutzerterminals mit dem Ein/Ausgabestream eines Sockets.

Mit der Befehlszeile `socat STDIO unix:`*MANAGEMENTSOCKET* kann eine vergleichbare Funktionalität erreicht werden. `socat` ist ein universelles Werkzeug, das Verbindungen zwischen Bytestreams herstellt. Es kann auch benutzt werden, um in Shell-Scripten Eingaben an Sockets zu schicken.

Durch das VDE-Projekt ist auch das Programm `vdeterm` entstanden, welches im Gegensatz zu `unixterm` und `socat` weitere Features enthält, wie eine Befehls-Historie und Befehlsvervollständigung.

Der Monitor von *QEMU* ermöglicht es unter anderem, die virtuelle Maschine während des Betriebs anzuhalten und wieder zu starten, (virtuelle) Geräte anzuschließen oder abzuklemmen, sowie Verbindungen zu virtuellen Netzen nachträglich herzustellen oder zu kappen.

Es ist auch möglich, den Zustand einer laufenden virtuellen Maschine zu speichern. Der gespeicherte Zustand kann nachher von anderen virtuellen Maschinen geladen werden. Dieses Feature ermöglicht es somit theoretisch, den Bootvorgang einer virtuellen Maschine zu verkürzen.

Eine weitere Möglichkeit, die der Monitor bietet, ist es, den einer virtuellen Maschine zugeteilten Speicher nachträglich per Eingabe zu reduzieren. Hierzu muss die VM jedoch mit der Option `-balloon virtio` gestartet werden. Durch diese Technik sollte es möglich sein, die Anzahl der gleichzeitig lauffähigen VMs auf einem Hostrechner zu erhöhen.

Dazu wird es allerdings notwendig, den freien Arbeitsspeicher auf den virtuellen Maschinen zu überwachen (beispielsweise durch das Linux-Kommando `free`), und aufgrund dieser Angaben zu bestimmen, wie viel Speicher die VM tatsächlich benötigt.

3. Ersatz für VNUML

3.1. XML

XML steht für „*Extensible Markup Language*“, also für eine erweiterbare Auszeichnungssprache. Die Sprache ist dazu gedacht, Dokumente zu erstellen, die menschenlesbar, von Programmen einfach zu verarbeiten und einfach zu erstellen sind.[41][18] *XML* ist eine Metasprache, d. h. die verfügbaren Tags und Elemente sind nicht vorgegeben, sondern können frei gewählt werden.

Die Struktur des *XML*-Dokuments wird dabei in einer Schemadatei (z. B. einer *DTD - Document Type Declaration*) festgehalten. Eine *DTD* beschreibt, welche Elemente in welcher Reihenfolge im Dokument vorkommen, welche Attribute sie besitzen und welche Art von Daten sie enthalten. So ist auch das von VNUML verwendete XML-Format in einer DTD beschrieben. Diese DTD liegt bei einer normalen VNUML-Installation unter `/usr/local/share/xml/vnuml/vnuml.dtd`. *DTD*s selbst sind keine *XML*-Dokumente, obwohl die Syntax auf den ersten Blick diesen Schluss nahe legt. Es gibt auch andere, teilweise mächtigere Schemaformate, die zur Beschreibung eines *XML*-Formates verwendet werden können. Dazu gehören zum Beispiel *XML Schema* (*XSD*)[15], *Schematron*[2] oder *RELAX NG*[25].

Ein *XML*-Parser überprüft beim Parsen eines Dokuments, ob das Dokument seinem Schema entspricht. Entspricht ein Dokument nicht dem Schema, bricht für gewöhnlich der Einlesevorgang ab. Beim Erstellen eines *XML*-Dokumentes muss daher auch sorgfältig darauf geachtet werden, dass die Reihenfolge der enthaltenen Elemente der im Schema vorgegebenen Reihenfolge entspricht.

3.2. Das VNUML-Format

VNUML benutzt zur Beschreibung von Szenarien ein *XML*-Format. Das Format definiert eine größere Menge von Tags, von denen im Folgenden die Wichtigsten aufgezählt werden:[17]

global - beschreibt Einstellungen, die für alle VMs der Simulation gelten. Dazu gehören unter anderem:

- simulation_name - enthält den Namen der Simulation
- vm_mgmt - beschreibt das Management-Netz, über das die VMs gesteuert werden
- vm_defaults - enthält Einstellungen für alle VMs, insbesondere:
  - filesystem - Root-Dateisystem, von dem die VM gestartet wird
  - kernel - User Mode Linux Kernel, der für die VM verwendet wird

net - beschreibt ein Netzwerk (identifiziert über das Attribut „name“)

vm - spezielle Einstellungen für eine VM (mit Attribut „name“ für den Hostnamen der VM)

  if - beschreibt ein Netzwerk-Interface der VM (mit Attribut „net“, das sich auf das „name“-Attribut eines Netzes bezieht)

  filetree - gibt über das „root“-Attribut einen Pfad auf dem Host an, der vor Aufruf der über das „seq“-Attribut angegebenen Befehlssequenz auf die VM kopiert wird. Der Inhalt des Elements gibt das Zielverzeichnis auf der VM an.

  exec - beschreibt ein Kommando einer Befehlssequenz, die nach Start der VM aufgerufen werden kann. Das „seq“-Attribut gibt den Namen der Befehlssequenz an. Es können auch mehrere exec-Elemente mit demselben „seq“-Attribut angegeben werden. Diese werden dann beim Aufruf der Befehlssequenz nacheinander abgearbeitet und mittels *SSH* auf den VMs ausgeführt.

3.3. Wahl des Dateiformats

Zu Beginn der Entwicklung des Startscriptes für *QEMU*-VMs musste entschieden werden, welches Dateiformat zur Beschreibung der Netzwerktopologie und der Konfiguration der virtuellen Maschinen verwendet werden sollte. Dazu standen mehrere Dateiformate zur Auswahl:

- das *VNUML*-Dateiformat oder ein darauf aufbauendes Format
- eine Neuentwicklung z.B. auf Basis von *XML* oder *JSON*[3]
- eine einfache Topologiebeschreibung in einer Textdatei

Da es oft notwendig ist, weitergehende Einstellungen zu tätigen, fallen die Topologiebeschreibungen als Dateiformat bereits aus dem Rahmen.
Das *VNUML*-Dateiformat ist bereits brauchbar, und da eine Neuentwicklung vermutlich ebenfalls auf *XML* aufsetzen würde, ist der zum Anpassen des Formates und der Implementation eines Parsers notwendige Aufwand vermutlich geringer als der Aufwand, der zum Entwerfen eines neuen Formates notwendig gewesen wäre.
Aufgrund der großen Anzahl bereits vorhandener *VNUML*-Szenarien und den in der Arbeitsgruppe verwendeten Tools, die ebenfalls auf *VNUML*-Szenarien arbeiten, fiel die Entscheidung auf das *VNUML*-Dateiformat.

Innerhalb des Scriptes wird allerdings zum Speichern der Daten eine eigene Datenstruktur verwendet. Siehe hierzu auch Kapitel 4.3 auf Seite 23.

3.4. Abänderungen für QEMU

Um auch *QEMU*-basierte Szenarios mit *VNUML*-Szenariobeschreibungen aufbauen zu können, waren einige Ergänzungen vonnöten. Die `vnuml.dtd` wurde um die Beschreibungen für folgende Tags ergänzt:

qemu_executable - gibt den Pfad der *QEMU*-Datei an

qemu_fs - gibt das Festplattenimage für die *QEMU*-VMs an
(Dies ist notwendig, wenn das für *VNUML* verwendete Festplattenimage keine mit *QEMU* nutzbaren Kernel und Kernelmodule enthält)

qemu_kernel - gibt einen außerhalb des durch `<qemu_fs>` angegebenen Festplattenimages liegenden Kernel an, mit dem die VM gestartet werden soll; es können zusätzlich die Attribute „initrd“ und „root“ angegeben werden.
(„initrd“ ist ein optionales Attribut und gibt eine *Initial Ramdisk*[24] an, die beim Start verwendet werden soll und „root“ gibt die Gerätedatei an, unter der der Kernel beim Start die Root-Partition finden kann. Wird das „root“-Attribut nicht angegeben, wird als Standardwert `/dev/sda` angenommen.)

qemu_cdrom - gibt ein ISO-Image an, das als CDROM-Laufwerk der VMs eingebunden wird

qemu_timeout - ermöglicht es, den Timeout beim Starten der VMs einzustellen

qemu_loadvm - ermöglicht es, beim Start einer VM direkt einen gespeicherten Zustand zu laden; es müssen zusätzlich die Attribute „overlayfile“ und „state“ angegeben werden.
(„overlayfile“ bezeichnet das Overlay-Image, das den gespeicherten Zustand enthält und „state“ bezeichnet den Namen des im Image gespeicherten Zustands)

extra_args - ermöglicht, beim *QEMU*-Aufruf zusätzliche Kommandozeilenoptionen zu übergeben

ssh_port_start - gibt innerhalb von vm_mgmt an, ab welchem Port des Hostrechners die *SSH*-Weiterleitungen an die Gastrechner beginnen

vnc_offset - gibt innerhalb von vm_mgmt an, ab welcher *VNC*-Displaynummer die *VNC*-Displays der Gastrechner beginnen

[24] Eine Initial Ramdisk ist ein Festplattenimage, welches Dateien enthält, die beim Start des Betriebssystems benötigt werden (z. B. Treiber für die Festplatte, die das Root-Dateisystem enthält). Beim Start des Systems wird das Image in den Arbeitsspeicher geladen und im Dateisystem eingebunden. Anschließend wird ein Script gestartet, das die Aufgabe hat, das eigentliche Root-Dateisystem einzubinden und den Startvorgang fortzuführen. [42]

4. Das Startscript

4.1. Ablauf des startnet.py-Scripts

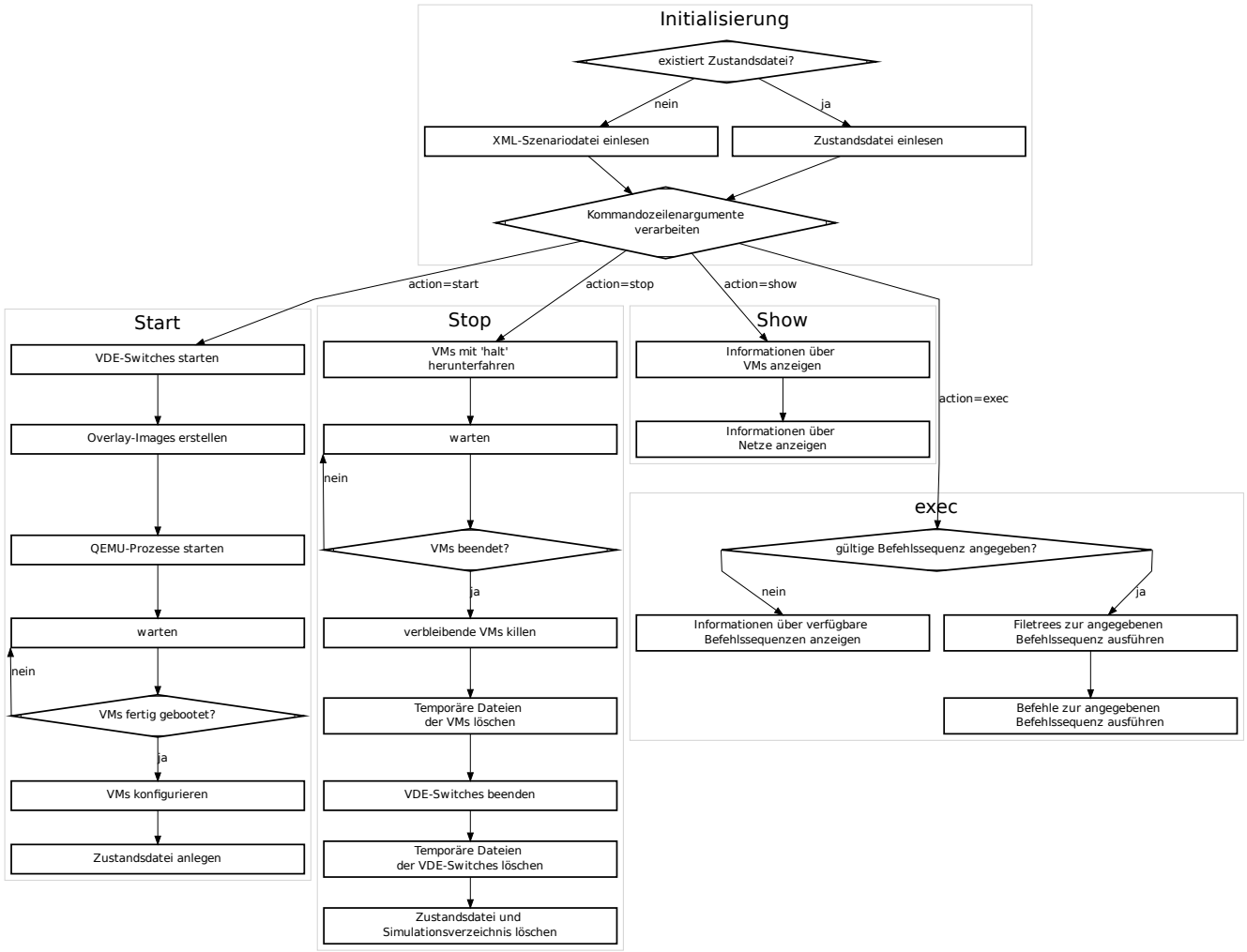

Abbildung 2: Programmablaufplan für startnet.py

Das startnet.py-Script übernimmt alle Tätigkeiten, die zum Starten und Einrichten, sowie zum Beenden der virtuellen Maschinen nötig sind.

Es erfordert als Eingabe eine XML-Datei, die das Szenario beschreibt, sowie eine Aktion. Mögliche Aktionen sind

- „`start`“, womit die in der Szenariodatei beschriebene Netzwerksimulation aufgebaut wird,
- „`stop`“, womit die Simulationsumgebung abgebaut und alle dazu gehörigen virtuellen Maschinen beendet werden,
- „`exec`“, wodurch in Kombination mit dem Kommandozeilenargument `-x` in der Szenariobeschreibung angelegte Befehlssequenzen ausgeführt werden können. Ohne die Angabe zusätzlicher Argumente wird eine Liste der vorhandenen Befehlssequenzen angezeigt.
- und „`show`“, was Informationen über das Szenario ausgibt.

Nach dem Start des Scripts wird überprüft, ob bereits eine Zustandsdatei existiert (siehe auch Kapitel 4.2 auf Seite 22). Ist dies der Fall, wird der in der Datei gespeicherte Zustand geladen. Sonst wird die angegebene XML-Datei eingelesen. Anschließend wird die angegebene Aktion durchgeführt.

Beim Starten der VMs werden zuerst die *VDE-Switch*-Instanzen gestartet, die im Folgenden die Netzwerkverbindungen zwischen den VMs darstellen.
Nach dem Starten der *VDE-Switches* werden die Overlay-Images für die VMs erzeugt. Soll jedoch ein gespeicherter VM-Zustand wiederhergestellt werden, wird für die betroffene VM kein neues Overlay-Image erstellt, sondern die Overlay-Datei, welche den gespeicherten Zustand enthält, kopiert.

Im Anschluss an das Anlegen der Overlay-Images werden die *QEMU*-Prozesse ausgeführt. Die Ausführung findet dabei parallel mithilfe von nebenläufigen Subprozessen statt. Die PIDs aller gestarteten Programme werden für den späteren Gebrauch gespeichert.
Vor der Konfiguration der VMs muss sichergestellt sein, dass diese gestartet sind und Befehle über *SSH* empfangen können. Daher wird zunächst gewartet, bis alle VMs *SSH*-Verbindungen akzeptieren. Anschließend werden die Netzwerkschnittstellen der VMs konfiguriert. Zuletzt werden alle Informationen in einer Zustandsdatei gespeichert.

Vor dem Ausführen von Befehlssequenzen wird überprüft, ob die vom Benutzer angegebenen Befehlssequenzen in der Szenariodatei tatsächlich vorhanden sind. Es ist dabei sowohl möglich, eine Befehlssequenz nur auf einer einzelnen VM auszuführen, als auch, eine Befehlssequenz auf allen VMs auszuführen, für die sie definiert ist.
Vor dem Ausführen wird überprüft, ob es ein Filetree-Tag mit derselben Sequenz gibt. In diesem Fall wird vor dem Ausführen der Sequenz durch Kopieren der im Filetree-Tag angegebenen Dateien oder Verzeichnisse das Filetree-Tag ausgeführt.

Beim Stoppen der VMs wird zunächst per *SSH* ein `halt` an alle VMs geschickt. Anschließend wird abgewartet, bis entweder keine VM mehr läuft oder bis eine vorgegebene Zeitspanne abge-

laufen ist. Die danach noch laufenden VMs werden dann über ein SIGTERM[25] beendet.
Darauf folgend werden die beim Starten erzeugten temporären Dateien (Overlay-Images, PID-Files, Logdateien und Sockets) entfernt. Dieser Schritt kann für Debug-Zwecke auch übersprungen werden.
Nachdem die VMs beendet wurden, werden auch die Prozesse der *VDE-Switches* mit einem SIGTERM beendet und ihre temporären Dateien entfernt. Als letzter Schritt wird die beim Starten der Simulation erstellte Zustandsdatei gelöscht.

4.2. Zustandsdatei

Beim Starten der VMs wird eine Zustandsdatei angelegt. In dieser werden alle notwendigen Informationen über die Simulation und den Zustand der VMs gespeichert.
Dazu gehören sowohl alle Informationen aus der Szenariobeschreibung, wie die Netzwerktopologie, die Einstellungen der VMs und die Befehlssequenzen, als auch die Informationen über den Zustand der Simulation, wie die PID-Nummern der gestarteten Prozesse, die Pfade der Steuersockets der VMs und *VDE-Switches*, und die Portnummern, unter denen *SSH-* oder *VNC*-Schnittstellen[26] der VMs erreichbar sind.

Es gibt mehrere Gründe, warum es sinnvoll ist, den Zustand in einer separaten Datei zu speichern:

Einerseits ist es notwendig, weil manche Informationen, wie die PIDs der *QEMU*-Prozesse, nach dem Start nicht mehr leicht herauszufinden sind. Die `-pidfile`-Option erzeugt zwar eine Datei mit der PID des dazugehörigen *QEMU*-Prozesses, jedoch werden in Kombination mit der `-daemonize`-Option zwei *QEMU*-Prozesse erzeugt, die beim Beenden des durch die PID-Datei angegebenen Prozesses mit einem KILL-Signal nicht beide beendet werden.
Andererseits kann es vorkommen, dass die zum Aufbau der Simulationsumgebung verwendete Szenariodatei zwischen dem Aufbau und dem Abbau vom Benutzer verändert wird. Um in einem solchen Fall einen konsistenten Zustand zu bewahren, werden beim Start der Simulation alle verfügbaren Informationen in die Zustandsdatei geschrieben.

Beim Start des Scriptes wird immer überprüft, ob zu der angegebenen Szenariobeschreibung bereits eine Zustandsdatei existiert. Ist dies der Fall, wird nicht die Szenariobeschreibung ausgelesen, sondern die in der Zustandsdatei gespeicherten Informationen geladen.

Die Zustandsdatei ist tatsächlich nur die serialisierte Form der Datenstrukturen des Scriptes. Zur Serialisierung wird das *YAML*-Modul von Python verwendet. *YAML* steht für „*YAML Ain't*

[25] Unter UNIX-artigen Betriebssystemen ist es möglich, laufenden Prozessen Signale zu schicken, um den Programmablauf zu beeinflussen. Zum Beenden eines Prozesses können die Signale SIGTERM und SIGKILL verwendet werden. SIGTERM ist dabei die sanftere Variante, die dem Prozess die Möglichkeit gibt, ungespeicherte Daten auf die Festplatte zu schreiben und sich sorgfältig zu beenden, die aber auch ganz vom Prozess ignoriert werden kann. SIGKILL beendet einen Prozess sofort, ohne dass der Prozess das Signal abfangen oder ignorieren kann.

[26] *VNC* steht für *Virtual Network Computing* und ist ein System, das es erlaubt, einen Computer, auf dem ein VNC-Server läuft, über ein Netzwerk fernzusteuern. Dabei wird der Bildschirminhalt des Servers an den VNC-Client übertragen und die Tastatur- und Maus-Eingaben am Client an den Server. QEMU ermöglicht es, eine VM über das VNC-Protokoll zu bedienen. Der QEMU-Prozess betreibt einen eigenen VNC-Server, der die über das Netzwerk übertragene Bildschirmausgabe direkt von der virtuellen Grafikkarte der VM abgreift, sodass innerhalb des VM-Betriebssystems kein VNC-Server laufen muss.

Markup Language“ („*YAML ist keine Auszeichnungssprache*“) und ist ein Format, das durch eine einfache Syntax und durch die Verwendung von Einrückungen besonders menschenlesbar ist.

Zwar sind manche Objekte nicht vollständig serialisierbar, wie die Ein/Ausgabepuffer der Subprozess-Objekte, jedoch macht dies im Normalfall nichts aus, da die Ein- und Ausgabe der gestarteten Prozesse entweder nicht verwendet oder direkt auf eine Logdatei umgeleitet werden.

4.3. Aufbau des Scriptes

Das Startscript besteht aus den im UML-Diagramm in Abbildung 3 abgebildeten Klassen. Beim Start des Programms wird zunächst - noch vor dem Einlesen der Szenario- oder Zustands-Datei - die `Simulation`-Klasse instanziiert. Das `Simulation`-Objekt verwaltet alle wichtigen Informationen über das simulierte virtuelle Netzwerk, inklusive einer Liste der VMs, der Netze und der Einstellungen.

Nach der Instanziierung des `Simulation`-Objektes muss mit der Methode `setFilename` der Dateiname der Szenariobeschreibung angegeben werden. Diese Methode sucht im Arbeitsverzeichnis nach einer möglicherweise vorhandenen Zustandsdatei und lädt die in dieser gespeicherten Datenstrukturen oder liest, falls keine Zustandsdatei vorhanden ist, die XML-Datei durch den Aufruf der Methode `parse_vnuml_file` ein.

Als Resultat erhält das `Simulation`-Objekt ein `Settings`-Objekt, welches als Dictionary[27] angelegt ist und allgemeine Einstellungen enthält, die für die gesamte Simulation gültig sind.

Es existiert in dem Simulation-Objekt mit `networks-dict` ein Dictionary, das jedem Netznamen das passende `Network`-Objekt zuordnet, und mit `vm_list` eine Liste mit `VM`-Objekten.

Ein `VM`-Objekt beschreibt die Konfiguration und den Zustand genau einer virtuellen Maschine innerhalb der Simulation. Jedes `VM`-Objekt hat ein `UserManagementInterface`-Objekt, das als `ManagementInterface` Methoden bereitstellt, über die Befehle auf der VM ausgeführt sowie Dateien oder Verzeichnisse auf die VM kopiert werden können.
Das Objekt enthält außerdem eine Liste von `Interface`-Objekten, die jeweils einen Verweis auf das `Network`-Objekt des Netzes besitzen, zu dem sie verbunden sind.

Außerdem hat jede VM ein eigenes `VMSettings`-Objekt, das ähnlich wie das `Settings`-Objekt des `Simulation`-Objektes aufgebaut ist und VM-spezifische Einstellungen verwaltet. Wird über das `VMSettings`-Objekt versucht, auf eine Einstellung zuzugreifen, die nicht in diesem existiert, greift das `VMSettings`-Objekt auf die Einstellungen des übergeordneten `Settings`-Objektes zurück.

Ein `Simulation`-Objekt stellt vier wichtige Methoden zur Verfügung:

[27] In Python werden assoziative Arrays als Dictionaries bezeichnet. Ein Dictionary wird angelegt, indem einer Variable ein Dictionary-Literal zugewiesen wird. Ein solcher besteht aus geschweiften Klammern, die beliebig viele Schlüssel-Wert-Paaren enthalten. Diese sind durch Kommata unterteilt. Schlüssel und Wert sind dabei durch einen Doppelpunkt voneinander getrennt. Beispiel: `mein_dictionary={"schluessel1":"wert1", "schluessel2":"wert2"}`

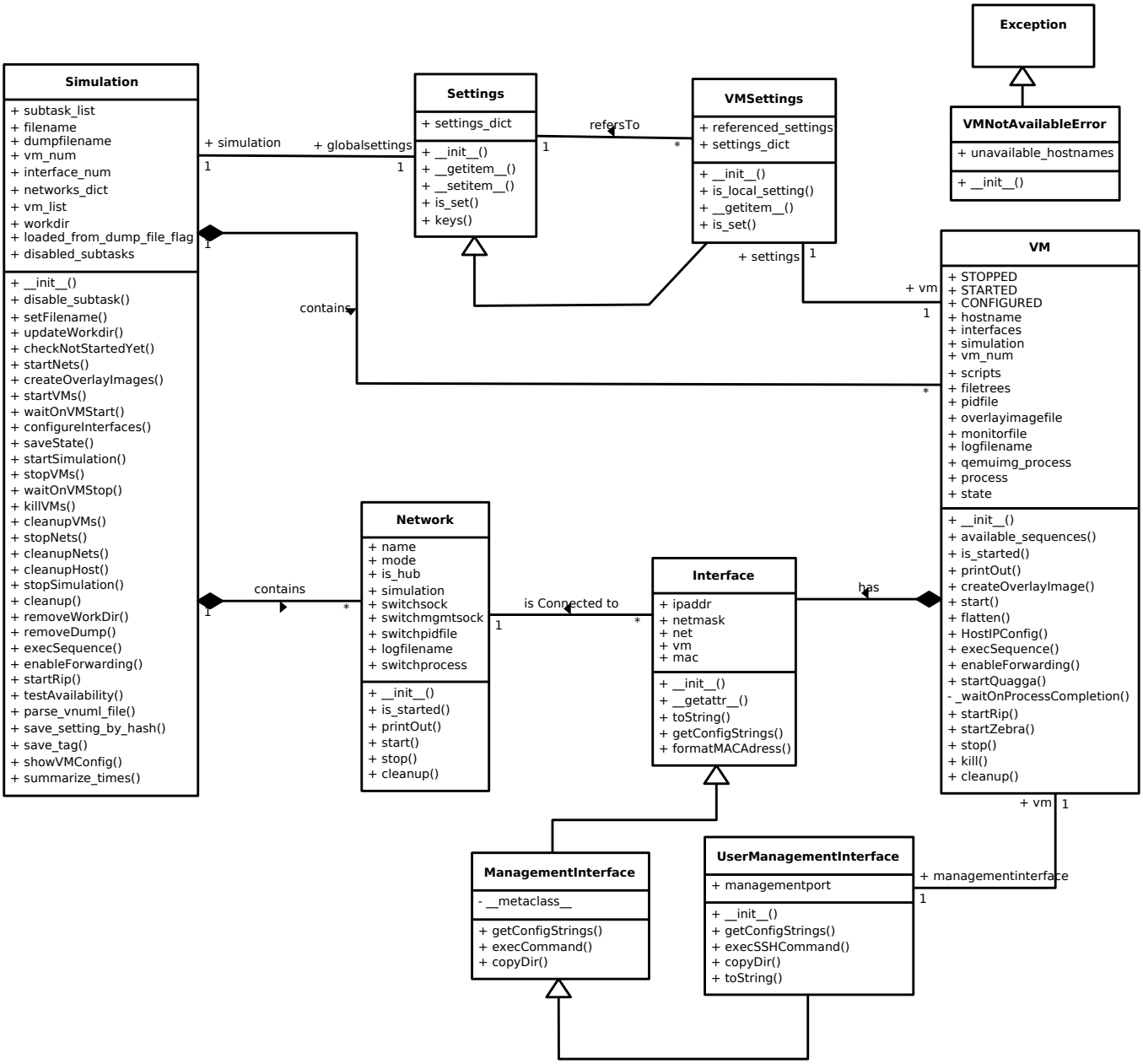

Abbildung 3: UML-Klassendiagramm von startnet.py

showVMConfig() gibt die Konfiguration aller VMs auf der Standardausgabe aus

execSequence() führt eine Befehlssequenz aus

startSimulation() baut das virtuelle Netzwerk auf

stopSimulation() baut das virtuelle Netzwerk wieder ab

Beim Aufruf des Startscriptes wird eine dieser vier Methoden aufgerufen, abhängig davon, welche Aktion durch den Benutzer auf der Kommandozeile angegeben wird.

Die beiden Methoden `startSimulation` und `stopSimulation`, die die Aktion `start`, bzw. `stop` bereitstellen, sind wiederum in Teilschritte, Subtasks, aufgespalten. In der Bedienungsanleitung (Kapitel 6.3 auf Seite 45) sind die Subtasks angegeben, die abhängig von der aufgerufenen Aktion gestartet werden.

Beim Aufruf jeder Subtask-Methode wird eine Ausgabe auf dem Terminal erzeugt, die den Namen und die Beschreibung des jeweiligen Subtasks beinhaltet.

Dies wird durch den Decorator[28] registerSubtask erreicht, der auf eine Funktion angewandt werden kann und der außerdem die Ausführungszeit des Subtasks misst.

Ein weiterer Decorator namens `class_find_subtasks`, der die Klasse `Simulation` dekoriert, sammelt beim Programmstart alle Subtasks in einer Liste, sodass die Hilfefunktion des Startscriptes die Namen der Subtasks angeben kann.[21]

VM und Network haben `start`- und `stop`-Methoden, die verwendet werden, um tatsächlich den *QEMU*- bzw. *VDE-Switch*-Prozess zu starten oder zu stoppen. In der Klasse `VM` gibt es zudem noch die `kill`-Methode, welche den *QEMU*-Prozess einer VM beenden kann, wenn dieser von der `stop`-Methode nicht beendet werden konnte. Die Klassen `VM`, `Network` und `Simulation` enthalten auch `cleanup`-Methoden, die nach dem Beenden der jeweiligen Prozesse aufgerufen werden können, um temporär angelegte Dateien und Verzeichnisse wieder zu entfernen.

Zum Starten der Prozesse wird das *subprocess*-Modul der Python Standard Library verwendet. Durch dieses wird der Prozess im Hintergrund gestartet und es wird ermöglicht, die Ein- und Ausgabe in einen Puffer oder eine Datei umzuleiten.
Die API des subprozess-Moduls erlaubt es, die PID und einen eventuellen Rückgabestatus des Prozesses abzufragen und den Prozess zu beenden.
Das Starten eines neuen Prozesses über das Modul wird in der Funktion `makeSubProcess` gekapselt. Dieser Funktion wird der Befehl zum Starten des Prozesses als Liste von Strings[29], und optional Filehandles zum Umleiten der Ein- und Ausgabestreams des Prozesses übergeben.

4.4. Parsen der XML-Datei

Zum Parsen des XML-Formates der Szenariodatei wird der XML-Parser *lxml*[5] verwendet. Dieser ermöglicht es, die Datei einzulesen und als eine Baumstruktur aufzubauen. Im aufgebauten Baum kann mithilfe von XPath-Ausdrücken nach Tags gesucht werden. Es ist so möglich, komfortabel auf die Informationen der Tags zuzugreifen.[30]

[28] In Python ist ein Decorator eine Funktion, die ein Funktions-Objekt entgegennimmt, um eine (veränderte) Funktion zurückzugeben. Eine Funktion wird dekoriert, indem vor der Funktionsdeklaration eine Zeile mit `@DEKORATORNAME` gesetzt wird.[10] Es gibt außerdem die Möglichkeit, mit derselben Syntax Klassen zu dekorieren.

[29] Der Befehl wird nicht als einzelner String, sondern als Liste von Strings übergeben.

Die Szenariodatei kann grob in mehrere Teile aufgegliedert werden:

- ein globaler Teil für die Konfiguration der Simulation
- in diesem auch ein Teil mit Standard-Einstellungen, die für alle VMs gelten
- die Netze innerhalb der Simulation
- die VMs der Simulation und ihre Konfiguration

Hierbei werden in den Standard-Einstellungen für alle VMs bis auf wenige Ausnahmen dieselben XML-Tags verwendet wie bei den Einstellungen für eine einzelne VM. Es ist daher möglich, beide Teile gleich zu behandeln.

Im Startscript wird zum Parsen der Szenariobeschreibung die Methode `parse_vnuml_file` der Klasse `Simulation` verwendet. Diese Methode liest die Szenariodatei ein und baut daraus die interne Datenstruktur, bestehend aus `VM`-, `Network`-, `Interface`- und `VMSettings`-Objekten auf. Hierbei wird für die globalen Einstellungen der Simulation und für die Standard-Einstellungen der VMs ein `Settings`-Objekt erzeugt, auf das sich die `VMSettings`-Objekte der VMs beziehen.

Zu Beginn muss mit der Funktion `etree.parse` die Szenariodatei als Baum eingelesen werden.

```
11 from lxml import etree
```

Quelltext 2: Import des lxml-Moduls

etree gehört hier zum *lxml*-Modul, nicht zur *ElementTree XML API*[30] der Python Standard Library, obwohl das *lxml*-Modul teilweise kompatibel zu diesem ist.

```
880         tree=etree.parse(filename, etree.XMLParser(dtd_validation=True))
```

Quelltext 3: Einlesen der Szenariodatei mit etree.parse

Hier wird neben dem Dateinamen der XML-Datei noch ein XML-Parser-Objekt übergeben, das über die Option `dtd_validation=True` so eingestellt wurde, dass beim Einlesen überprüft wird, ob die Szenariodatei konform ist zur Document Type Definition. Wenn dies nicht der Fall ist, bricht der Vorgang mit einer Exception und einer Fehlermeldung ab.

Die Ausgabe der `parse`-Funktion ist ein `ElementTree`-Objekt. Es bietet unter anderem folgende Methoden an:

[30] http://docs.python.org/py3k/library/xml.etree.elementtree.html

find – sucht im Baum abwärts und liefert das erste Element zurück, welches einem angegebenen Pfad-Ausdruck entspricht; liefert `None`[31] zurück, wenn kein Element gefunden werden kann

findall – sucht im Baum abwärts und liefert alle Elemente zurück, welche einem angegebenen Pfad-Ausdruck entsprechen; liefert eine leere Liste zurück, wenn kein Element gefunden werden kann

iter – ermöglicht es, über alle Elemente des Baumes zu iterieren

Um an die Informationen eines Elementes zu gelangen, muss auf die Attribute *text* und *tag*, sowie auf die Methode *get* zurückgegriffen werden. Erstere enthalten den Inhalt bzw. den Namen des Elements. Die Methode *get* gibt bei Angabe eines Attributnamens den Inhalt des entsprechenden Attributs im Element zurück.

Durch Verwendung dieser API können die Informationen in der Szenariodatei ausgelesen werden:

Zunächst wird der globale Teil der Szenariobeschreibung geparst. Hierzu wird zunächst das `<global>`-Tag gefunden und die in diesem enthaltenen Elemente verarbeitet:

```
884         xml_global=tree.find("global")
          [...]
910         for elem in xml_global.iter(tag=etree.Element):
911             save_tag(elem)
```

Quelltext 4: Verarbeiten des <global>-Tags

Der Parameter `tag=etree.Element` des iter-Aufrufs stellt sicher, dass nur XML-Elemente durchlaufen werden, keine Kommentare oder Verarbeitungsinstruktionen.[6] Die hier verwendete Hilfsfunktion *save_tag* stellt dabei fest, ob ein Element verwertet wird, und speichert dessen Informationen in dem globalen *Settings*-Objekt der Simulation.

[31] `None` ist eine Python-Konstante, die die Abwesenheit eines Wertes angibt.[16] `None` kann als das Python-Äquivalent zu `null` in anderen Programmiersprachen betrachtet werden.

```
885         save_into={"qemu_fs":"base_image", "qemu_cdrom":"qemu_cdrom", ↩
                "qemu_executable":"qemu_executable", "mem":"mem", \
886                 "simulation_name":"simulation_name", "vnc_offset":"vnc_offset", \
887                 "ssh_port_start":"ssh_port_start", "extra_args":"extra_args", \
888                 "qemu_timeout":"timeout", "forwarding":"forwarding", ↩
                        "qemu_loadvm":"loadvm", "qemu_kernel":"kernel"}
889         # dictionary used to decide which tags to save and where
890
891         tag_filter={"qemu_timeout": lambda tag: int(tag.text), \
892                 "forwarding":  lambda tag: tag.get("type"), \
893                 "qemu_loadvm": lambda tag: (tag.get("overlayfile"), tag.get("state")),\
894                 "qemu_kernel": lambda tag: {"kernel":tag.text, "root":tag.get("root")}}
895         # dictionary used to process a tag before saving it
896
897         def save_setting_by_hash(settings_container, tag):
898             try:
899                 content=tag_filter[tag.tag](tag)
900             except: content=tag.text
901             settings_container[save_into[tag.tag]]=content
902         def save_tag(tag):
903             if tag.tag in save_into.keys(): ↩
                    save_setting_by_hash(self.globalsettings,tag)
904             if tag.tag == "simulation_name": self.updateWorkdir(tag.text)
905             if tag.tag == "vm_mgmt":
906                 self.globalsettings["mgmt_net_addr"]=tag.get("network")
907                 self.globalsettings["mgmt_net_mask"]=tag.get("mask")
```

Quelltext 5: Verarbeiten und Speichern der Tags

Hierzu wird zunächst überprüft, ob das Tag in dem Dictionary *save_into* vorhanden ist. In diesem Dictionary ist zu jedem zu verarbeitenden Tag ein Name eingetragen, unter dem die entsprechende Information im *Settings*-Objekt hinterlegt werden soll.

Mögliche Einträge im *tag_filter*-Dictionary geben Verarbeitungsschritte an, die vor dem Speichern der Information auf dem Tag angewendet werden müssen (zum Beispiel das Extrahieren von bestimmten Attributen des Tags oder die Umwandlung eines Datentyps).

Die Vorverarbeitung über die im *tag_filter* eingetragene anonyme Funktion und die anschließende Speicherung in dem passenden *Settings*-Objekt wird über eine weitere Hilfsfunktion, *save_settings_by_hash* erledigt. Tags, die nicht vollständig über *save_settings_by_hash* verarbeitet werden können, werden von der *save_tag*-Funktion separat behandelt.

Im nächsten Schritt wird die Liste der Netze erstellt:

```
912 self.networks_dict=dict( ↩
        [net.get("name"),Network(net.get("name"),net.get("mode"),net.get("hub"),self)] ↩
        for net in tree.findall("net") )
```

Quelltext 6: Verarbeiten der <net>-Tags

In einer List Comprehension[32] wird ein Dictionary angelegt, in dem der Name eines Netzes auf das entsprechende *Network*-Objekt verweist. Diese *Network*-Objekte werden hier erzeugt.

Im letzten Schritt werden die Einträge der einzelnen VMs betrachtet:

```
915 for vm in tree.findall("vm"):
916     entry=VM(vm.get("name"), self)
917     for tag in vm.iter(tag=etree.Element):
918         if tag.tag in save_into.keys(): save_setting_by_hash(entry.settings, tag)
919 
920     entry.interfaces=[ Interface( interface.find("ipv4").text,              \
921                                   interface.find("ipv4").get("mask"),       \
922                                   self.networks_dict[interface.get("net")], \
923                                   entry )      for interface in             \
924                         sorted(vm.findall("if"), key=lambda tag: tag.get("id"))]
925     exec_sequence_names = set([exc.get("seq") for exc in vm.findall("exec")])
926     entry.scripts       = dict( [sequence_name,[cmd.text \
927                             for cmd in vm.findall("exec[@seq=\'"+sequence_name+"\']")]]\
928                             for sequence_name in exec_sequence_names )
929     ft_sequence_names   = set([ft.get("seq") for ft in vm.findall("filetree")])
930     entry.filetrees     = dict( [sequence_name,[(ft.text,ft.get("root")) \
931                             for ft in ↩
                                    vm.findall("filetree[@seq=\'"+sequence_name+"\']")] ] \
932                             for sequence_name in ft_sequence_names)
933     self.vm_list.append(entry)
```

Quelltext 7: Verarbeiten der <vm>-Tags

Es werden alle <vm>-Tags durchlaufen und Einträge erzeugt, die in die Liste *vm_list* der *VM*-Objekte im *Simulation*-Objekt eingetragen werden. Dabei werden zunächst alle Elemente innerhalb des <vm>-Elements mit der schon im globalen Teil verwendeten Hilfsfunktion *save_settings_by_hash* verarbeitet.

Im Gegensatz zur Verwendung im globalen Teil wird hier jedoch als Container für die Einstellungen das zur jeweiligen VM zugehörige *Settings*-Objekt verwendet, nicht das globale *Settings*-Objekt. Dadurch werden die Einstellungen für die VM auch nur im *Settings*-Objekt der VM gespeichert, wobei es möglich ist, über dasselbe *Settings*-Objekt auch auf die globalen Einstellungen im *Settings*-Objekt der Simulation zuzugreifen.

Eine Liste der Interfaces der VM wird über eine List Comprehension erstellt. Hier wird durch Nachschlagen des im Tag vermerkten Netzwerknamens im zuvor angelegen Dictionary *networks_dict* der Bezug zum passenden *Network*-Objekt hergestellt.

Die Interfaces werden in der Reihenfolge hinzugefügt, die durch die „id“-Attribute der <if>-Tags bestimmt wurde. Es ist jedoch nicht möglich, die Bezeichnung der Interfaces innerhalb der VM direkt festzulegen, daher stimmen die in der XML-Datei vorgegebenen IDs nicht mehr mit den

[32] List Comprehensions sind eine Spracherweiterung von Python (definiert in PEP 202[40]) die, angelehnt an funktionale Sprachen wie Haskell, ermöglichen, mit einer kompakten Notation Listen zu erstellen, indem Operationen auf Elemente aus anderen Listen angewendet werden und Elemente ausgefiltert werden. Durch Anwendung von eingebauten Funktionen wie dict oder set können mit List Comprehensions auch Dictionaries oder Mengen erzeugt werden.

Interface-Namen innerhalb der VM überein, wenn Lücken in der Aufzählung der Interface-IDs bestehen.

Zur Erfassung der Filetrees und Befehlssequenzen werden verschachtelte List Comprehensions verwendet. Zuvor müssen jedoch über eine weitere List Comprehension die verwendeten Sequenznamen festgestellt werden. Diese werden in *exec_sequence_names* bzw. *ft_sequence_names* gespeichert. Mit den Sequenznamen können die jeweils passenden <exec> bzw. <filetree>-Tags gefunden werden. So wird für die Filetrees und Befehlssequenzen jeweils ein Dictionary erzeugt, dass einem Sequenznamen eine Liste mit Befehlen der zugehörigen Befehlssequenz bzw. Quellen und Zielen für die Kopiervorgänge des zugehörigen Filetrees zuweist.

In der gesamten Funktion muss nicht überprüft werden, ob ein bestimmtes Attribut tatsächlich in der XML-Datei existiert. Einerseits ist für die meisten der Attribute durch die *DTD* vorgeschrieben, dass sie im XML-Tag angegeben werden müssen. Andererseits werfen die Funktionen des *lxml*-Moduls keine Exceptions, sondern geben nur den Wert `None` zurück, wenn ein Attribut nicht existiert. Beim lesenden Zugriff auf Einstellungen in einem *Settings*-Objekt muss jedoch unter Umständen überprüft werden, ob eine Einstellung gesetzt wurde. Hierzu besitzt die Klasse *Settings* die Methode *is_set(setting_name)*. Folgender Code aus der *start*-Methode der Klasse *VM* prüft also, ob eine Einstellung für die zugewiesene Speichermenge einer VM existiert und fügt das entsprechende Kommandozeilenargument für den *QEMU*-Aufruf hinzu:

```
127 if self.settings.is_set("mem"): args.extend(["-m", self.settings["mem"]])
```

Quelltext 8: Beispiel für den Zugriff auf Einstellungen in einem Settings-Objekt

Soll dem Parser ermöglicht werden, weitere XML-Tags zu verarbeiten, gibt es mehrere Möglichkeiten:

1. Wenn das Tag pro VM und im globalen Bereich nur einmal vorkommt und keine Attribute hat, die verarbeitet werden müssen, kann einfach ein Eintrag in dem *save_into*-Dictionary erzeugt werden, durch den der Inhalt des Tags in den *Settings*-Objekten abgelegt wird.

2. Hat das Tag zusätzlich Attribute, die gespeichert werden müssen, oder enthält das Tag einen Wert, der in einen Integer- oder einen Float-Wert umgewandelt werden muss, so kann im *tag_filter*-Dictionary für das Tag eine anonyme Funktion abgelegt werden, mit der das Tag verarbeitet wird.

3. Steht das Tag im Zusammenhang mit anderen Tags, oder kommt es mehrfach vor, muss an einer passenden Stelle der *parse_vnuml_file*-Funktion angelehnt an die Behandlung der <net>-Tags eine Verarbeitungsinstruktion eingefügt werden.

Ist das neue XML-Tag nicht Teil der *VNUML*-Sprache, so muss zusätzlich die *DTD*-Datei verändert werden, um das neue Tag aufzunehmen.

Zuletzt muss dort, wo die Informationen des Tags benötigt werden, Code eingefügt werden, durch den diese aus dem *Settings*-Objekt wieder ausgelesen werden.

4.5. Erweitern um neue Befehle

Es ist verhältnismäßig einfach, neue Befehle in das Startscript zu integrieren. Dazu muss zunächst die Konfiguration des *argparse*-Moduls verändert werden. *Argparse* dient dem einfachen Verarbeiten von Kommandozeilenargumenten. Zuerst muss der Name der neuen Aktion in die *choices*-Liste des `action`-Arguments in der Zeile

```
992   parser.add_argument("action", choices=["show", "start", "stop", "exec"])
```

Quelltext 9: Einstellen des action-Arguments in der argparse-Konfiguration

eingetragen werden. Diese Zeile bewirkt, dass *Argparse* das „`action`“-Argument versteht. Durch die Angabe des Parameters „`choices`“ wird eingestellt, dass beim Einlesen des Arguments zu überprüfen ist, ob die Eingabe korrekt war.

Wenn eine ungültige Eingabe getätigt wurde, bricht das Programm ab und dem Benutzer wird mitgeteilt, welche Eingaben zulässig sind.

Die Zeile befindet sich innerhalb der If-Abfrage `if __name__ == "__main__":` .
Diese Abfrage wird wahr, wenn das Script direkt ausgeführt wird, also nicht als Modul in einem anderen Programm benutzt wird. Der Inhalt der if-Abfrage entspricht also der *main*-Funktion in einem C-Programm.

Zusätzlich muss zu der Aktion eine Funktion in das *action_name_to_function*-Dictionary eingetragen werden:

```
1034    action_name_to_function={\
1035            "show":  (simulation.showVMConfig,    []),\
1036            "exec":  (simulation.execSequence,    [arguments.sequence]),\
1037            "start": (simulation.startSimulation, []),\
1038            "stop":  (simulation.stopSimulation,  [])\
1039    }
```

Quelltext 10: Definition des action_name_to_function-Dictionaries

Das Dictionary weist einem Aktionsnamen ein Tupel, bestehend aus einem Funktionsobjekt und einer Liste von Argumenten, zu. Wenn auf der Befehlszeile eine Aktion angegeben wird, prüft zunächst *Argparse*, ob die angegebene Aktion in der *choices*-Liste angegeben ist und somit eine gültige Aktion ist.

Nach der Verarbeitung der Kommandozeilenargumente wird der Aktionsname im Dictionary nachgeschlagen und die passende Funktion aufgerufen, wobei die Argumente aus der Liste übergeben werden. Benötigt die neue Aktion zusätzliche Argumente, so müssen diese in der *argparse*-Konfiguration angegeben werden. Genauere Informationen dazu finden sich in der API-Dokumentation von *argparse*[33].

[33] http://docs.python.org/dev/library/argparse.html

Die Funktion, die in der neuen Aktion verwendet wird, sollte am besten eine Methode der Klasse *Simulation* sein, oder ein Objekt der Klasse *Simulation* als Argument übergeben bekommen, um auf deren Attribute zugreifen zu können.

5. Messungen

5.1. Vorgehensweise

Um das Startscript und VNUML vergleichen zu können, werden zwei Kriterien herangezogen:

- Der vom virtuellen Netzwerk verbrauchte Arbeitsspeicher
- Die Zeit, die benötigt wird, bis das virtuelle Netzwerk aufgebaut ist

Der verbrauchte Arbeitsspeicher ist daher interessant, weil er die Anzahl der VMs begrenzt, die gleichzeitig betrieben werden können. Die Startzeit des virtuellen Netzwerks ist ein Maß für die Geschwindigkeit der VMs.

5.1.1. Speicherverbrauch

Den Speicherverbrauch eines Prozesses oder einer Gruppe von Prozessen zu messen ist unter Linux verhältnismäßig schwierig. Zwar existieren verschiedene Werkzeuge, die Informationen über den Speicherverbrauch eines Prozesses liefern, jedoch sind auf diese Weise keine Informationen zu erhalten, wieviel Speicher sich ein Prozess (z. B. über shared libraries[34]) mit anderen Prozessen teilt. Außerdem ist es bei der Vielzahl der verfügbaren Werkzeuge schwierig, den Überblick zu behalten, welche Messwerte wie miteinander zusammenhängen.

Ein Ansatz ist es, mithilfe des Werkzeugs `free`[13] den Arbeitsspeicherverbrauch des gesamten Systems zu messen. [35] Dieses Werkzeug fasst Informationen aus der Datei */proc/meminfo* zusammen und gibt an, wie viel physikalischer Arbeitsspeicher frei ist bzw. verwendet wird (in den Spalten `free` und `used`), wie viel Speicher für Buffer und Caches verwendet wird (`buffers` und `cached`) sowie wie viel Speicher auf die Festplatte ausgelagert wurde.

```
> # free
             total       used       free     shared    buffers     cached
Mem:       2028904    1962616      66288          0     149704     748124
-/+ buffers/cache:    1064788     964116
Swap:      2065096       5092    2060004
```

Quelltext 11: Beispiel für die Ausgabe von free

[34] Programmbibliotheken, die von mehreren Programmen gleichzeitig genutzt werden können, wobei sich zumeist nur ein Exemplar der Programmbibliothek im Speicher befindet[44]

Um den Speicherverbrauch eines oder mehrerer Prozesse zu bestimmen, reicht es allerdings nicht, nur den Verlauf der `free`- und `used`-Werte zu betrachten.
Der Linux-Kernel versucht, den vorhandenen physikalischen Arbeitsspeicher zu jedem Zeitpunkt möglichst optimal auszunutzen und verwendet daher einen großen Teil des eigentlich „freien“ Speichers für Buffer und Caches. Verbraucht ein Prozess den verbleibenden freien Speicher, wird wieder Speicher von den Buffern und Caches abgezogen. Theoretisch können also die Werte für Buffer und Caches zum freien Speicher gezählt werden.
Aus diesem Grund zeigt `free` eine Zeile mit „buffer-adjusted“-Werten an, mit $used' = used - (buffers + cached)$ und $free' = free + buffers + cached$.
Praktisch gesehen ist jedoch ein gewisses Minimum an Buffern und Caches notwendig, damit das System ausreichend performant bleibt.
Leider sind die aus `free` bzw. `/proc/meminfo` erhaltenen Informationen im Falle von VNUML schwer zu interpretieren, da dort gleichzeitig mit den Werten für den verwendeten Speicher auch die Werte für die Caches steigen. So erscheint es bei Betrachtung der „buffer-adjusted“-Werte so, als verbrauche das VNUML-Szenario überhaupt keinen Speicher.

Dennoch ist es schwierig, aus diesen Werten auf den Speicherverbrauch einer Gruppe von Prozessen zu schließen. Es gibt aber eine Reihe von Werkzeugen, die Informationen über einzelne Prozesse liefern:

Die gebräuchlichen Werkzeuge, wie `ps`[23] oder `top`[39], liefern für einen Prozess die Werte *RSS*, *VSZ* und *SHR*.

RSS steht dabei für *Resident Set Size* und umfasst den Teil des Speichers eines Programmes, der im Arbeitsspeicher gehalten wird.

VSZ oder VIRT steht für *Virtual Set Size* und beschreibt die Größe des gesamten dem Programm zugeordneten Adressraums (unabhängig davon, wie viel Speicher in diesem Adressraum tatsächlich verwendet wird).

SHR steht für *shared memory* und bezeichnet laut der `top`-Manpage[39] den Speicher des Programms, der eventuell mit anderen Prozessen geteilt werden könnte.

Somit ist nur *RSS* potenziell ein brauchbares Maß für den Speicherverbrauch eines Programms. Soll jedoch den Speicherverbrauch einer Gruppe von Programmen gemessen werden, so ist *RSS* nicht optimal, da der Wert auch Speicher beinhaltet, der von mehreren Programmen verwendet wird. Das bedeutet, dass beim Aufsummieren der RSS-Werte mehrerer Prozesse ein zu hoher Speicherverbrauch herauskommt.

Eine Abhilfe zu diesem Problem bietet das Programm `pmap`[9][11]. Es gibt für einen Prozess eine Tabelle aus, in welcher der virtuelle Speicherverbrauch des Prozesses unterteilt wird. So lässt sich zuordnen, welcher Speicher vom Prozess alleine gebraucht wird und welche Speicheranteile auf shared libraries zugeordnet sind. Die so erhaltenen Angaben sind allerdings in Hinblick auf den physikalischen Speicherverbrauch noch ungenau.

Ein noch besseres Werkzeug ist `smem`[35]. Es verwendet die */proc/PID/pagemap*-Datei jedes Prozesses. Diese Datei gibt die Zuordnung von virtuellen Speicherseiten zu physikalischen Speicherseiten wieder. Die so erhaltenen Informationen werden korreliert und so für jeden Prozess zwei Metriken berechnet:

USS steht für *Unique Set Size* und umfasst den Speicher, der alleinig vom Prozess genutzt wird.

PSS steht für *Proportional Set Size* und stellt die Summe der vom Prozess genutzten Speicherseiten dar, jeweils geteilt durch die Anzahl der Prozesse, die sich die jeweilige Speicherseite teilen.

Der *PSS*-Wert scheint der brauchbarste Wert zu sein, da er durch Addition auch für mehrere Prozesse gültige Werte ergibt. Der Nachteil an `smem` ist, dass die Ausführung je nach Anzahl der Prozesse im System ungefähr ein bis zwei Sekunden benötigt. Dies ist bedingt dadurch, dass die *pagemap*-Dateien aller Prozesse eingelesen und verglichen werden müssen. Es gibt jedoch auch die Möglichkeit, die Auswertung von vornherein nur auf eine Auswahl von Prozessen zu beschränken, wodurch sich die Ausführungszeit drastisch verkürzt.

5.1.2. KSM

KSM (Kernel Samepage Merging) ist eine in neueren Linux-Kerneln enthaltene Funktionalität, die es ermöglicht, identische Speicherseiten im Arbeitsspeicher zu finden und zusammenzuführen, um so Arbeitsspeicher zu sparen (Memory Deduplication).

Dies wird dadurch bewerkstelligt, dass der KSM-Prozess periodisch einen bestimmten Speicherbereich nach doppelten Speicherseiten durchsucht. Es werden hierbei jedoch nur Speicherbereiche durchsucht, die von den besitzenden Anwendungen als vereinigbar markiert wurden.[20] KSM kann daher seine Vorteile nur bei Programmen ausspielen, die mit Unterstützung für KSM programmiert wurden. Dies ist beispielsweise bei QEMU und KVM der Fall, bei Usermode Linux jedoch zum Zeitpunkt der Erstellung dieser Arbeit nicht.

KSM lässt sich durch Einstellungsdateien unter */sys/kernel/mm/ksm/* steuern und überwachen:[14]

full_scans – gibt an, wie viele Scans über den ganzen von KSM überwachten Speicherbereich durchgeführt wurden

pages_shared – gibt an, wie viele mehrfach genutzte Speicherseiten es gibt

pages_sharing – gibt an, wie viele Speicherseiten auf diese mehrfach genutzten Speicherseiten verweisen

pages_unshared – gibt an, wie viele Speicherseiten nicht mehrfach genutzt werden

pages_volatile – gibt an, wie viele Speicherseiten sich zu schnell ändern, um gemeinsam genutzt zu werden

[35] http://www.selenic.com/smem/

run – steuert, ob KSM aktiv ist (mit `1` für aktiv und `0` für inaktiv)

pages_to_scan – stellt ein, wie viele Speicherseiten in einem einzelnen Scanvorgang durchsucht werden

sleep_millisecs – stellt ein, wie lange zwischen Speicherscans in Millisekunden gewartet wird, ein sinnvoller Wert ist z. B. `20` Millisekunden

5.1.3. Messablauf

Ursprünglich war geplant, eine *QEMU*-VM als „Mess-VM“ zu verwenden, innerhalb derer die virtuellen Netze aufgebaut werden sollten. Um sicherzustellen, dass die einzelnen Messungen sich nicht gegenseitig beeinflussen und um einen einheitlichen Startzustand zu gewährleisten, sollte die Mess-VM vor jeder Messung auf einen gespeicherten Zustand zurückgesetzt werden.

Leider hat sich herausgestellt, dass die Ausführung einer *QEMU*-VM innerhalb einer anderen *QEMU*-VM selbst bei Verwendung des „kvm nesting“[36] erheblich verlangsamt ist, während User Mode Linux Prozesse innerhalb der Mess-VM praktisch nicht beeinflusst werden.

Daher wurden die Messungen sowohl für das Startscript als auch für VNUML nicht innerhalb einer VM, sondern direkt auf einem Hostsystem ausgeführt.

Zu Beginn einer Messung wird ein Shellscript gestartet, das periodisch (im Abstand von jeweils einer Sekunde) mehrere Messwerte sichert und mit Zeitstempeln versieht. Dazu gehören:

- die Ausgabe von `smem`
- der Inhalt von */proc/meminfo*
- der Inhalt des */sys/kernel/mm/ksm/*-Verzeichnisses

Zehn Sekunden nach Start der Messung wird die Simulation gestartet. Die Zeitpunkte des Programmstarts und der Programmbeendung werden in einer Logdatei festgehalten. 60 Sekunden nach dem erfolgten Aufbau des virtuellen Netzes wird der Befehl ausgeführt, um das Netzwerk wieder abzubauen. Auch hierbei werden die Start- und Endzeiten festgehalten. Nach weiteren 10 Sekunden wird die Messung gestoppt und die Messdaten aus der Mess-VM extrahiert.

Nach Beendigung der Messung werden die gesammelten Daten mittels eines Python-Scripts ausgelesen und ausgewertet:

Dieses liest zunächst die Logdatei ein, um die Dauer des Startvorgangs zu bestimmen.

Die Zeitstempel der einzelnen Messpunkte werden nun umgerechnet, sodass die Zeitbasis nun relativ zum Startzeitpunkt der Messung ist. So kann später einfach zugeordnet werden, zu welchem Schritt welcher Speicher verbraucht wurde.

[36] Verfügbar machen der Virtualisierungshardware des Hostes für VM-Gäste innerhalb einer VM

Bei den */proc/meminfo*- und *KSM*-Messungen können die gespeicherten Messwerte direkt übernommen werden, ohne dass eine weitere Vorverarbeitung notwendig ist. Die Ausgabe von `smem` jedoch muss, um als brauchbarer Messwert dienen zu können, erst zusammengefasst werden:

`smem` gibt eine Liste von Prozessen mit jeweils mehreren Kennzahlen zu deren Speicherverbrauch aus. Um den gesamten Speicherverbrauch der Simulation zu erhalten, wird die Summe der PSS-Werte der zur Simulationsumgebung gehörigen Prozesse gebildet.

Um diese Menge von Prozessen zu identifizieren, wird festgehalten, welche Prozesse vor dem Start der Simulation auf dem Computer bereits gestartet waren. Diejenigen Prozesse, die ab dem Startzeitpunkt neu hinzukommen, werden als zur Simulationsumgebung zugehörig betrachtet und in die Summe mit aufgenommen. Zuvor werden jedoch Prozesse herausgefiltert, die bestimmten Mustern entsprechen, bzw. nicht entsprechen. So werden Prozesse mit dem Prozessnamen des Startscriptes oder von VNUML, Switchprozesse, Usermode-Linux- und QEMU-Prozesse sowie ausgehende SSH-Verbindungen (mit dem Prozessnamen `ssh`) zu den Prozessen des virtuellen Netzwerks gezählt, eingehende SSH-Verbindungen (mit dem Prozessnamen `sshd`) und `smem`-Aufrufe jedoch nicht.

Zusätzlich wird eine Summe der PSS-Werte aller Prozesse im System erstellt.

5.2. Messergebnisse

Es wurden folgende Messungen durchgeführt:

- *QEMU* mit aktiviertem *KSM* (startnet-with_ksm)
- *QEMU* mit deaktiviertem *KSM* (startnet-no_ksm)
- *VNUML* im *SKAS0*-Betrieb (vnuml-skas-0)

Es wurde keine Messung mit *VNUML* im *SKAS3*-Betrieb[12] durchgeführt, da es nicht möglich war, den dazu notwendigen Kernel-Patch auf einen aktuellen Linux-Kernel anzuwenden. Zwar ist es denkbar, dass ein *VNUML*-Szenario unter einem *SKAS3*-gepatchten Kernel schneller startet als unter einem Kernel ohne den Patch, jedoch ist es nicht sinnvoll, eine Variante zu testen, die nur noch mit erheblichen Schwierigkeiten bzw. gar nicht mehr einsetzbar ist. Zum Beispiel funktioniert zum Zeitpunkt der Erstellung dieser Arbeit der SKAS3-Patch in aktuellen Linux-Kerneln nur noch auf 32-Bit-Systemen.

Die Szenarien, auf denen die Messungen durchgeführt wurden, waren Circle-Topologien[37] mit 3,4,5,6,7,8,16,32 und 50 Knoten. Da die Software-Switches zwischen den Knoten aufgrund ihres geringen Speicherverbrauchs und ihrer sehr kurzen Startzeit einen sehr geringen Einfluss auf das Messergebnis haben, konnte auf Messungen mit komplexeren Topologien verzichtet werden.

[37] Eine Circle- bzw. Ring-Topologie stellt einen geschlossenen Kreis aus Rechnern dar, die jeweils nur mit ihrem Vorgänger und Nachfolger im Kreis verbunden sind.[45]

5.2.1. Startzeiten

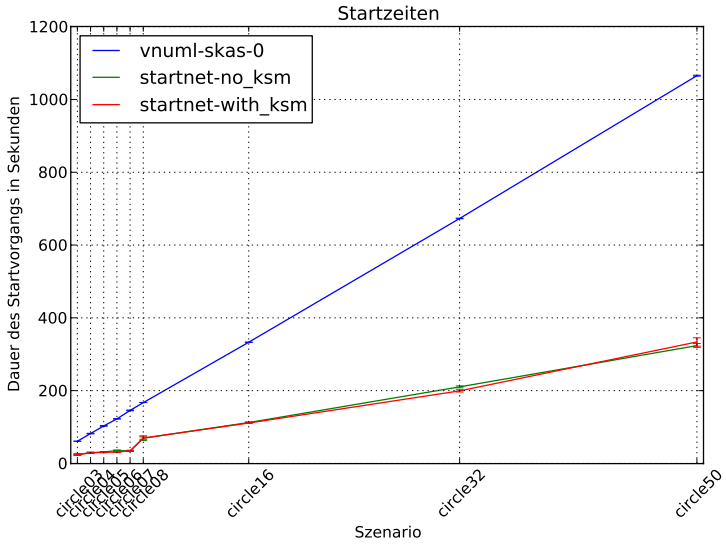

Abbildung 4: Startzeiten der VMs

Das hier abgedruckte Diagramm stellt die Startzeiten der VMs in Sekunden abhängig vom Szenario, bzw. der Anzahl der VMs dar. Für jeden Datenpunkt wurde der Mittelwert aus drei Messungen genommen. Fehlerbalken geben die Standardabweichung der Messwerte an.

Wie sich am Diagramm erkennen lässt, liegen die Startzeiten der *VNUML*-Szenarien durchweg über denen der mit *QEMU* simulierten Szenarien. Die *QEMU*-Szenarien mit aktiviertem und deaktiviertem *KSM* weisen dabei hinsichtlich der Startzeit keinen nennenswerten Unterschied zueinander auf, was darauf hinweist, dass die zusätzliche Prozessorbelastung durch das *KSM* nur geringen Einfluss auf die Leistung der VMs hat.

Da beim Start der VMs auf große Festplattenimages zugegriffen werden muss, lässt sich vermuten, dass die Geschwindigkeit der Festplatte des Hostsystems einen großen Einfluss auf die Startzeit hat. Bei Versuchen mit einem großen Szenario, in denen das verwendete Festplattenimage auf einer Ramdisk[38] abgelegt war, konnten jedoch nur geringe Unterschiede zu den vorher durchgeführten Versuchen ohne Verwendung einer Ramdisk festgestellt werden. Folglich liegt der begrenzende Faktor für die Startzeit der VMs in der Kombination aus Prozessorgeschwindigkeit und verfügbarem Arbeitsspeicher.

[38] Eine Ramdisk ist ein Bereich im Arbeitsspeicher, der als virtuelle Festplatte eingebunden wird. Der Vorteil einer Ramdisk gegenüber einer physikalischen Festplatte ist die schnelle Zugriffszeit und die hohe Datenübertragungsrate. Allerdings sind die auf einer Ramdisk abgelegten Daten flüchtig und gehen beim Herunterfahren des Rechners verloren.

5.2.2. Speichernutzung

In Abbildung 5 wird der Speicherverbrauch der VMs abhängig vom Szenario dargestellt. Die Einheit für den Speicherverbrauch ist hierbei der vom `smem`-Programm erfasste PSS-Wert als Summe von allen zum virtuellen Netzwerk zugehörigen Prozessen. Dieser Wert kann auch als Anzahl der belegten Speicherseiten betrachtet werden.

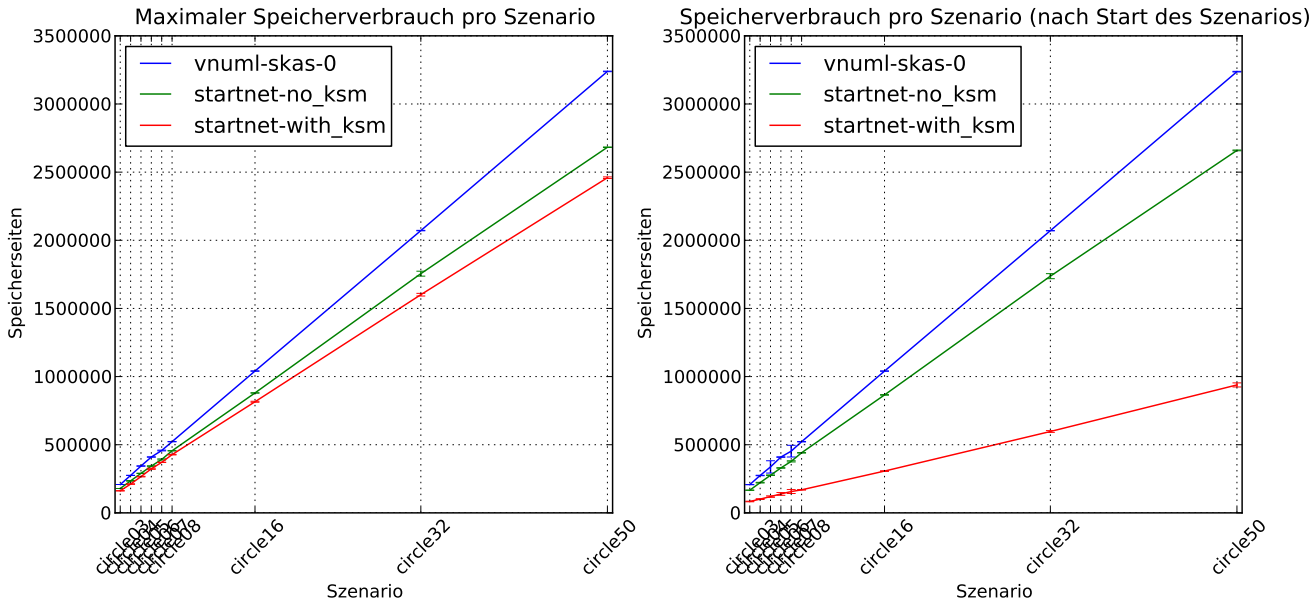

Abbildung 5: Speicherverbrauch der VMs

Bei dem linken Diagramm wurde der jeweils maximale Speicherverbrauch während des ganzen Messvorganges aufgetragen, bei dem rechten Diagramm der durchschnittliche Speicherverbrauch zwischen erfolgtem Aufbau und begonnenem Abbau der Simulationsumgebung.

Auch hier wurde jeweils das Mittel der Werte genommen und die Standardabweichung mit Fehlerbalken dargestellt. Die Schwankung ist jedoch so gering, dass die Fehlerbalken größtenteils nicht zu erkennen sind.

An den Diagrammen lässt sich erkennen, dass bei beiden Systemen, *VNUML* und *QEMU*, der Speicherverbrauch linear mit der Anzahl der VMs steigt. Beim maximalen wie beim durchschnittlichen Speicherverbrauch schneidet *VNUML* stets schlechter ab als *QEMU*. Weiter lässt sich feststellen, dass sich der maximale Speicherverbrauch der VMs im Allgemeinen nicht vom durchschnittlichen Speicherverbrauch unterscheidet. Eine Ausnahme dafür bilden die *QEMU*-Szenarios, die mit *KSM* gestartet wurden. Hier ist der durchschnittliche Speicherverbrauch jeweils nur ungefähr halb so groß wie der maximale Speicherverbrauch.

Der Grund dafür wird Abbildung 6 ersichtlich. Aufgetragen ist hier der zeitliche Verlauf der Speichernutzung für *VNUML*, *QEMU* und *QEMU* mit *KSM*. Das gestartete Szenario ist ein Circle mit 50 VMs. Pfeile zeigen an, wann die jeweilige Simulation fertig aufgebaut wurde und wann der Befehl zum Abbau gegeben wurde. Der Speicherverbrauch von *VNUML* ist in blau aufgetragen, der von *QEMU* mit *KSM* in rot und der Speicherverbrauch von *QEMU* ohne *KSM* in grün.

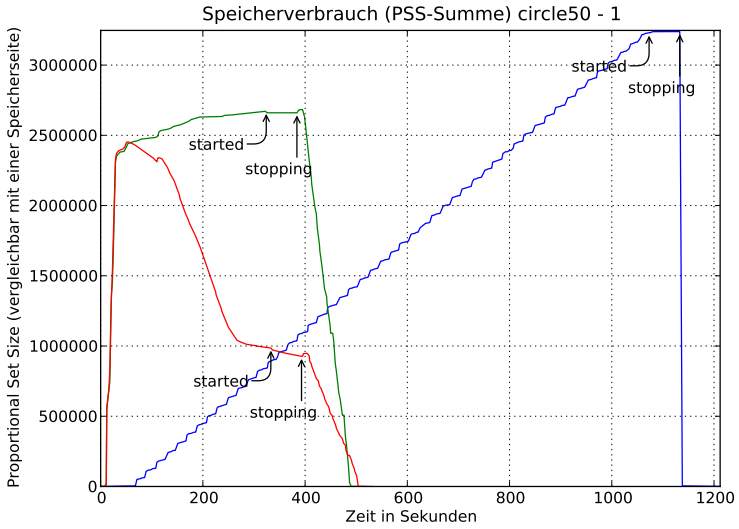

Abbildung 6: Speicherverbrauch nach Zeit

Der zu beobachtende Unterschied zwischen *VNUML* und *QEMU* ist, dass *VNUML* den Speicher nach und nach aufbraucht und aufhört, weiteren Speicher zu verbrauchen, wenn der Startvorgang der Simulation abgeschlossen ist, während bei *QEMU* direkt der maximale Speicher angefordert wird. Bei der Verwendung von *QEMU* mit *KSM* wird ebenfalls zu Beginn der gesamte Speicher angefordert, jedoch sorgt *KSM* dafür, dass nach und nach wieder Speicher freigegeben wird, bis sich der Speicherverbrauch auf einer bestimmten Höhe stabilisiert. Folglich ist der maximale Speicherverbrauch bei der *KSM*-Variante ähnlich hoch wie bei der Variante ohne *KSM*, jedoch ist der Speicherverbrauch nach Start der Simulation geringer.

Um auch den hohen maximalen Speicherverbrauch zu vermeiden, könnte versucht werden, die virtuellen Maschinen nicht gleichzeitig, sondern zeitlich versetzt zu starten. So könnte es auch möglich sein, Szenarien mit einer großen Zahl von VMs startbar zu machen. Ein Nachteil einer derartigen Vorgehensweisen könnte jedoch sein, dass dadurch die Startzeit des Szenarios leidet.

Die hohe Speichereinsparung durch den Einsatz von *KSM* liegt daran, dass in den getesteten Szenarien alle VMs dasselbe Festplattenimage als Basis verwendeten. So konnte *KSM* viele Speicherseiten mit gleichem Inhalt finden und vereinigen. Die Effizienz hängt davon ab, wie viele verschiedene Systeme in einem Szenario verwendet werden.

Abbildung 7 zeigt die gemessenen *KSM*-Statistiken abhängig vom Szenario. Ein Messpunkt gehört jeweils zu dem Wertetupel, dessen „pages_sharing“-Wert innerhalb des Messvorganges am höchsten war. Es wird also nur jeweils der Zeitpunkt innerhalb eines Durchlaufs betrachtet, zu dem die Anzahl der Speicherseiten, die auf eine gemeinsame Speicherseite verwiesen, am höchsten war. Dies ist der Zeitpunkt, zu dem die Speichereinsparung durch KSM am höchsten war.

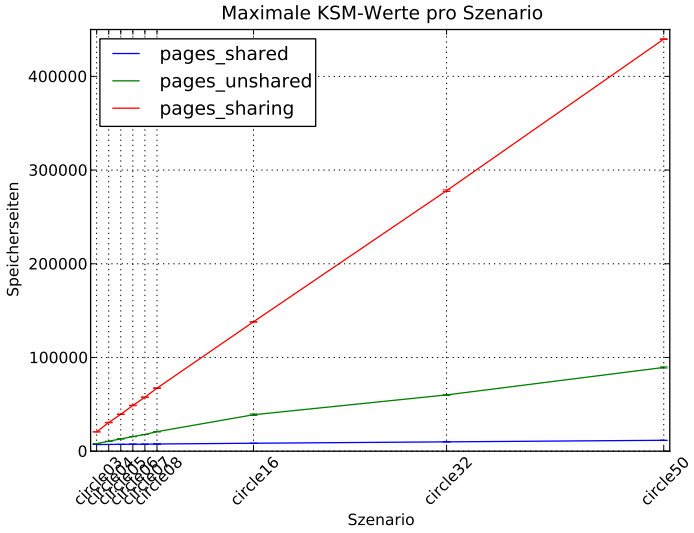

Abbildung 7: KSM-Statistiken nach Szenario

Der *KSM*-Dokumentation zufolge weist ein hohes Verhältnis von „pages_sharing“ zu „pages_shared“ auf eine hohe Effizienz hin: Ein hohes Verhältnis von „pages_unshared“ zu „pages_sharing“ soll dahingegen anzeigen, dass die Effizienz niedrig ist:
„A high ratio of pages_sharing to pages_shared indicates good sharing, but a high ratio of pages_unshared to pages_sharing indicates wasted effort.“[14]

Das Diagramm zeigt, dass der „pages_shared“-Wert weitestgehend unabhängig vom Szenario und somit der Anzahl der VMs ist. Er stellt die Speicherseiten einer VM dar, die alle VMs gemeinsam haben. Der „pages_sharing“-Wert steigt linear mit der Anzahl der VMs, was bedeutet, dass, da der „pages_shared“-Wert gleich bleibt, die Effizienz ebenfalls mit der Anzahl der VMs steigt. Der „pages_unshared“-Wert, der die Anzahl der Seiten angibt, die nicht von mehreren VMs geteilt werden, steigt ebenfalls mit der Anzahl der VMs in einem Szenario, jedoch ist die Steigung weitaus kleiner als die des „pages_sharing“-Wertes.

In Abbildung 8 wird als Beispiel der zeitliche Verlauf der *KSM*-Werte in Zusammenhang mit dem Speicherverbrauch eines *QEMU*-Szenarios gezeigt. Das hierzu verwendete Szenario ist wieder ein Circle mit 50 VMs.

Hier ist zu sehen, dass nach dem Start der Simulation der Speicherverbrauch zu fallen und gleichzeitig der „pages_sharing“-Wert und der „pages_unshared“ zu steigen beginnt, während sich der „pages_shared“-Wert nur geringfügig verändert. Hierbei steigt der „pages_sharing“-Wert stärker als der „pages_unshared“-Wert, sodass sich am Ende der Speicherrückgewinnung

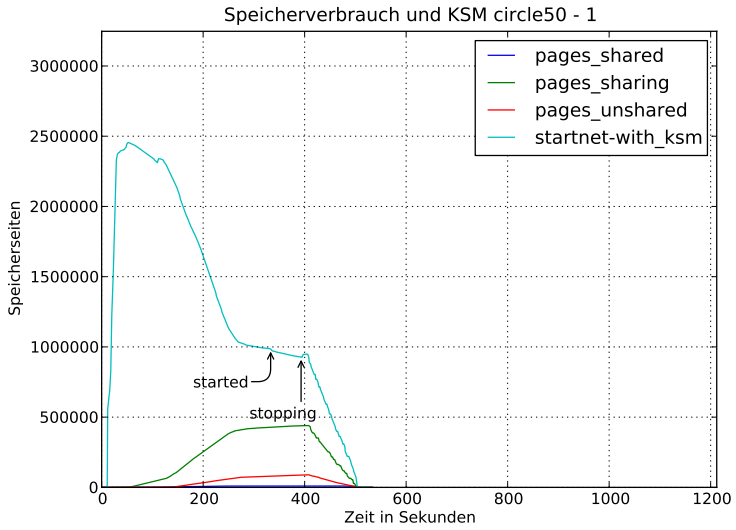

Abbildung 8: KSM-Statistiken nach Zeit

ein hohes Verhältnis von „pages_sharing“ zu „pages_shared“ und ein geringeres Verhältnis von „pages_unshared“ zu „pages_shared“ bildet.

6. Bedienungsanleitung

6.1. Installation

Das Startscript besteht aus zwei Dateien: `startnet.py` und `decorators.py`. Zusätzlich werden die *DTD*-Datei des veränderten *VNUML*-Dateiformats `vnuml.dtd`, eine Szenariodatei, das zum Starten des Szenarios benötigte Festplattenimage und, falls in der Szenariodatei angegeben, ein Linux-Kernel, der zum Booten des Festplattenimages verwendet wird, benötigt.

Alle diese Dateien lönnen durch das Auschecken des Startscriptes aus dem Git-Repository auf http://git.uni-koblenz.de/startnet-py erhalten werden.

Git ist ein verteiltes Versionverwaltungssystem, das sich unter anderem von älteren Versionverwaltungssystemen wie CVS oder SVN dadurch unterscheidet, dass jeder Benutzer beim Auschecken die vollständige Historie eines Projektes auf seinen Rechner lädt, wodurch die Notwendigkeit für ein zentrales Repository entfällt.

Durch den Befehl

```
git clone git://git.uni-koblenz.de/startnet-py/startnet.git
```

kann das Repository ausgecheckt werden. Es befindet sich anschließend im Unterverzeichnis `startnet` des aktuellen Arbeitsverzeichnisses.

Das Repository enthält auch sogenannte „submodules“, Unterverzeichnisse des Repositories, die Verweise auf andere Git-Repositories darstellen. Eines dieser Submodule im Repository des Startscriptes wird mit „vm-ressources“ bezeichnet und enthält Kernel und Festplattenimages, die mit dem Startscript verwendet werden können.

Um sich die vorhandenen Submodule anzeigen zu lassen, kann im Repository-Verzeichnis folgender Befehl ausgeführt werden:

```
git submodule status
```

Um anschließend das Submodul herunterzuladen, kann folgender Befehl ausgeführt werden:

```
git submodule update --init vm-ressources
```

Das Startscript selbst ist in Python geschrieben und benötigt einen Interpreter für Python 3. Unter Debian-basierten Systemen lässt Python sich mit

```
apt-get install python3
```

installieren. Zusätzlich zum Interpreter werden noch einige Module benötigt, die nicht in der Standard-Library enthalten sind. Dazu gehören *PyYAML*, *argparse* und *lxml*. Sind die Setup-Tools von Python installiert, können die benötigten Module einfach mit

```
easy_install3 pyYAML lxml argparse
```

installiert werden. Um das *lxml*-Modul zu kompilieren, müssen allerdings zunächst die Header-Dateien von *libxml2* und *libxslt* vorhanden sein. Die passenden Debian-Pakete hierzu sind `libxml2-dev` und `libxslt1-dev`. Unter Debian werden die Setup-Tools und die benötigten Pakete am einfachsten folgenden Befehl installiert:

```
apt-get install python3-dev python3-setuptools libxml2-dev libxslt1-dev
```

Alternativ zum Verwenden der Setup-Tools kann der Quellcode der Module von Hand vom Python Package Index (PyPI) heruntergeladen und nach dem Entpacken über

```
python3 setup.py build
```

und

```
python3 setup.py install
```

installiert werden.

Zusätzlich zum Startscript wird noch *QEMU* und *VDE-Switches* benötigt. Unter einem Debian-basierten System lautet der Befehl zum Installieren der Pakete:

```
apt-get install qemu-kvm vde2 libvdeplug2
```

Die Pakete enthalten jedoch eine Version von *QEMU*, in der die Unterstützung für *VDE* nicht mit einkompiliert ist. Um auf den Wrapper `vdeqemu` verzichten zu können, muss daher der Quellcode für *QEMU* von `www.qemu.org` besorgt und selbst kompiliert werden.

Die Installation verläuft beim Selbst-Kompilieren des *QEMU*-Sourcecodes Linux-üblich per `./configure`, `make` und `make install` (als root-Benutzer).
Bei Debian-basierten Systemen ist es außerdem von Vorteil, `checkinstall` anstelle von `make install` zu verwenden, und so die Software als `.deb`-Paket zu installieren, welches sich nachträglich über den Paketmanager wieder restlos entfernen lässt.

Dem `configure`-Script des *QEMU*-Sourcecodes müssen vor dem `make`-Schritt noch Argumente übergeben werden, durch die die gewünschten Features aktiviert werden:

```
./configure --enable-kvm --enable-vde --enable-vnc-thread --enable-io-thread
```

Schlägt der Aufruf dieses Befehls fehl, kann es sein, dass die Header-Dateien der *VDE*-Libraries nicht vorhanden sind oder nicht gefunden werden können.

Die Debian-Pakete, welche diese enthalten, sind `libvde-dev` und `libvdeplug2-dev`. Soll mit `-enable-vnc-png` die PNG-Kompression der VNC-Konsole aktiviert werden, wird zusätzlich die Header-Dateien der libpng benötigt, installierbar mittels des Pakets `libpng12-dev`.

6.2. Erzeugen der HTML-Dokumentation

Es ist möglich, aus den Kommentaren und Docstrings[39] des Sourcecodes eine Dokumentation des Programms als HTML-Dokument zu erzeugen. In diesem sind die vorhandenen Klassen des Programms und die in ihnen enthaltenen Variablen und Methoden mit ihrer Funktion aufgelistet.

Zum Erstellen der HTML-Dokumentation wird das Programm `epydoc` verwendet[40]. Da es zur Dokumentation von Programmen in der zweiten Version von Python geschrieben wurde, das Startscript jedoch in der dritten Version der Python-Programmiersprache verfasst wurde, können beim Aufruf Fehlermeldungen auftreten, die jedoch ignoriert werden können.

Der zum Erstellen der Dokumentation notwendige Aufruf lautet:

```
epydoc -v --debug --graph all startnet.py decorators.py
```

Die Dokumentation liegt anschließend im Unterverzeichnis *`html`* und kann durch Laden der Datei *`index.html`* in einem Webbrowser angezeigt werden.

[39] Docstrings sind Kommentare, die nicht vom Interpreter bzw. Compiler ignoriert werden, sondern auf die zur Laufzeit des Programms zugegriffen werden kann. In Python ist dies über das Attribut `__doc__` eines Objektes möglich. Einer der Vorteile der Docstrings ist, dass z. B. während der interaktiven Benutzung des Interpreters auf die Dokumentation zugegriffen werden kann.
Im Startscript werden Docstrings genutzt, um die Namen und die Funktionen der Teilschritte beim Starten oder Beenden der VMs auf der Konsole auszugeben.

[40] `http://epydoc.sourceforge.net/`

6.3. Bedienung des Scripts

Das Startscript ist in der Datei *startnet.py* enthalten. Das Programm hat folgende Syntax:

```
./startnet.py OPTIONEN AKTION szenariodatei
```

Zum Start des Scripts muss eine auszuführende Aktion und der Dateiname einer Szenariodatei angegeben werden.

Es gibt vier mögliche Aktionen: `show`, `start`, `stop` und `exec`

`show` zeigt Informationen über die Konfiguration des in der Szenariodatei beschriebenen virtuellen Netzwerks an. Ist die Netzwerksimulation bereits gestartet, werden zusätzliche Informationen, wie die PIDs der gestarteten Prozesse, die *SSH*-Ports und *VNC*-Displays der virtuellen Maschinen angezeigt.

`start` startet das in der Szenariodatei beschriebene virtuelle Netzwerk.

`stop` hält ein bereits laufendes virtuelles Netzwerk wieder an.

`exec` führt in der Szenariodatei beschriebene Filetrees und Befehlssequenzen aus.

Über die Option `-x` bzw. `-exec` oder `-seq` kann der Name jeweils einer Befehlssequenz angegeben werden. Es wird dann auf jeder VM versucht, zuerst die Filetree-Tags und anschließend die Befehlssequenzen mit der angegebenen Sequenz auszuführen.

Es ist auch möglich, eine Sequenz nur auf einer ausgewählten VM auszuführen. Dazu muss nach dem Sequenznamen ein @-Zeichen und der Hostnamen der VM angegeben werden (z.B. `-x start@bravo`, um die Sequenz `start` auf der VM `bravo` zu starten).

Mit einer `-x`-Option kann auch nur ein Sequenzname angegeben werden.
Sollen mehrere Sequenzen gestartet werden, so können auch mehrere `-x`-Optionen angegeben werden. Die Sequenzen werden dann nacheinander gestartet, in der Reihenfolge, in der sie auf der Befehlszeile angegeben wurden.

Zusätzlich können Optionen angegeben werden, die den Programmablauf beeinflussen:

`-Q` oder `-qemu-executable` – stellt ein, mit welchem Befehl *QEMU* aufgerufen werden soll (standardmäßig `qemu`).

`-d` oder `-debug` – stellt ein, dass keine Aktion tatsächlich durchgeführt wird - es wird nur angezeigt, welche Befehle ausgeführt werden würden.

`-k` oder `-keepimage` – sorgt dafür, dass die während der Simulation erzeugten Dateien (Images, Sockets, u. ä.) nach dem Beenden nicht gelöscht werden.

`-T` oder `-timeout` – legt die Zeit fest, in der eine VM gestartet sein soll. Der Wert wird mit der Anzahl der VMs multipliziert, um den tatsächlichen Timeout-Wert zu erhalten.

`-p` oder `-ssh-port-start` – stellt ein, ab welchem Port des Hostsystems die *SSH*-Ports der Gastrechner anfangen sollen. Standardmäßig fangen die Weiterleitungen bei Port 2222 an.

`-m` oder `-macaddr-start` – stellt ein, bei welcher MAC-Adresse die Netzwerkkarten der Gastrechner anfangen. Standardmäßig fangen die Adressen bei 52:54:00:12:34:56 an.

`-X` oder `-extra-args` – stellt Argumente ein, die jeder *QEMU*-VM beim Start zusätzlich übergeben werden.

`-t` oder `-timing-csv` – stellt den Dateinamen einer CSV-Datei ein, in die nach der Ausführung des Scriptes Informationen über die Ausführungszeiten der einzelnen Teilschritte gespeichert werden.

`-s` oder `-select` legt fest, welche Subtasks (Teilschritte) der ausgewählten Aktion ausgeführt werden sollen.

`-S` oder `-unselect` legt fest, welche Subtasks übersprungen werden sollen. Standardmäßig werden alle Subtasks der Aktionen ausgeführt.

`-s` und `-S` schließen sich gegenseitig aus. Die Option kann mehrfach angegeben werden, um mehrere Subtasks zum Ausführen oder Überspringen auszuwählen.

Standardmäßig werden bei den Aktionen folgende Subtasks ausgeführt:
Die Aktion „start“ führt die Subtasks startNets, createOverlayImages, startVMs, waitOnVMStart, configureInterfaces und saveState aus.
Die Aktion „stop“ führt die Subtasks stopVMs, waitOnVMStop, killVMs, cleanupVMs, stopNets, cleanupNets und cleanupHost aus.
Die Aktionen „show“ und „exec“ führen keine Subtasks aus.

6.4. Beispiel

Im folgenden Beispiel wird eine Y-Topologie als ein *QEMU*-Szenario aufgebaut. Die Netzwerksimulation wird mit dem Startscript gestartet. Anschließend werden auf den virtuellen Maschinen RIP-Router gestartet. Zur Demonstration der bestehenden Verbindungen zwischen den VMs wird ein Ping durchgeführt. Anschließend wird die Simulation beendet.

Zunächst muss eine Szenariodatei erstellt werden. Diese beginnt mit einem Header:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE vnuml SYSTEM "./vnuml.dtd">
```

Quelltext 12: Header der Szenariodatei

Hier wird beschrieben, dass die Datei eine XML-Datei ist, die mit UTF-8[46] codiert ist. Die zweite Zeile gibt an, dass das Wurzelelement „vnuml“ ist und in der *Document Type Declaration* `./vnuml.dtd` beschrieben wird. Die hier angegebene *DTD* unterscheidet sich von derjenigen, die von *VNUML* benutzt wird; diese liegt normalerweise unter `/usr/share/xml/vnuml/vnuml.dtd`. Die Unterschiede bestehen in den Tags, die für die Benutzung von *QEMU* eingeführt wurden (siehe Kapitel 3.2 auf Seite 17).

```
4  <vnuml>
5    <global>
6      <version>1.8</version>
7      <simulation_name>y-topologie</simulation_name>
8      <ssh_version>2</ssh_version>
9      <ssh_key>~/.ssh/id_rsa.pub</ssh_key>
10     <automac/>
11     <vm_mgmt type="private" network="192.168.0.0" mask="24" offset="100" >
12        <host_mapping/>
13     </vm_mgmt>
14     <vm_defaults>
15      <qemu_executable>qemu</qemu_executable>
16        <filesystem type="cow">/usr/share/vnuml/filesystems/root_fs_tutorial</filesystem>
17      <qemu_fs>vm-ressources/images/rootfs_tutorial_forqemu.qcow2</qemu_fs>
18      <extra_args>--enable-kvm</extra_args>
19      <qemu_kernel>vm-ressources/kernels/2.6.38.2/bzImage-2.6.38.2</qemu_kernel>
20        <kernel>/usr/share/vnuml/kernels/linux</kernel>
21     </vm_defaults>
22   </global>
```

Quelltext 13: <global>-Teil der Szenariodatei

Nach dem Header beginnt mit dem Tag `<vnuml>` das eigentliche Dokument.
Innerhalb des `<vnuml>`-Tags kommt zunächst das `<global>`-Tag. Es beinhaltet Einstellungen, die für die gesamte Simulation gelten. Anhand der *DTD* muss das `<global>`-Tag existieren und mindestens die Tags `<version>` und `<simulation_name>` enthalten.

Das `<version>`-Tag soll identifizieren, für welche *VNUML*-Version das Szenario geschrieben ist und wird daher vom *QEMU*-Startscript ignoriert.
Das `<simulation_name>`-Tag hingegen beschreibt den Simulationsnamen und wird vom Startscript für die Benennung des Arbeitsordners verwendet.

Es folgen die optionalen Tags `<ssh_version>`, `<ssh_key>`, `<automac>`, `<netconfig>`, `<vm_mgmt>`, `<tun_device>` und `<vm_defaults>`.
Das Startscript verarbeitet jedoch von diesen Tags nur das `<vm_mgmt>`-Tag und das `<vm_defaults>`-Tag.

Aus dem `<vm_mgmt>`-Tag wird die für das Management-Netz zu verwendende Netzwerkadresse entnommen.
Die im `<vm_mgmt>`-Tag enthaltenen Tags `<ssh_port_start>` und `<vnc_offset>` wurden neu eingeführt und stellen ein, bei welchem Port auf dem Host-Rechner der *SSH*-Port der ersten VM liegt, und welches *VNC*-Display die erste VM hat.
Bei den nächsten VMs wird jeweils der darauf folgende Port bzw. das nächste *VNC*-Display genommen.

Das <host_mapping>-Tag wird nur von *VNUML* benutzt und sorgt dort dafür, dass die Hostnamen der VMs in die */etc/hosts*-Datei des Hosts eingetragen werden.

Das optionale <vm_defaults>-Tag enthält Standard-Einstellungen, die für alle VMs gelten. *QEMU* betreffen hier nur die Tags <qemu_executable>, <qemu_fs>, <extra_args> und <qemu_kernel>.

<qemu_executable> stellt hier ein, welcher Befehl zum Start von *QEMU* verwendet werden soll. Ohne Angabe des Tags wird vom Startscript der Wert qemu angenommen. Je nach installierter *QEMU*-Version und Betriebssystem kann es jedoch sein, dass unter Verwendung dieses Befehls keine Unterstützung für *VDE-Switches* verfügbar ist. Sollte dies der Fall sein, kann hier ein anderer Befehl angegeben werden, wie hier der Befehl des Wrapperscriptes (siehe hierzu auch Kapitel 2.7.2 auf Seite 15).

<qemu_fs> stellt das Festplattenimage ein, von dem die *QEMU*-VMs gestartet werden sollen. Dieses Tag ist das *QEMU*-Äquivalent zum <filesystem>-Tag, durch welches das Festplattenimage angegeben wird, mit dem bei *VNUML* die User Mode Linux VMs gestartet werden. Dadurch, dass für *QEMU* ein separates Festplattenimage angegeben wird, kann ein Szenario so geschrieben werden, dass es sowohl mit QEMU als auch mit Usermode Linux startfähig ist.

<extra_args> dient dazu, zusätzliche Kommandozeilenargumente anzugeben, die den *QEMU*-Aufrufen angehängt werden.
In diesem Beispiel ist das zusätzliche Argument -enable-kvm. Es soll bewirken, dass *QEMU* versucht das *KVM*-Kernelmodul zu benutzen, um die Ausführung der VMs zu beschleunigen.

<qemu_kernel> ermöglicht es, einen externen Kernel anzugeben, der anstelle eines möglicherweise im Festplattenimage vorhandenen Kernels geladen wird.
Im Tag kann auch ein Attribut root angegeben werden, da dem Kernel mitgeteilt werden muss, auf welcher (virtuellen) Festplatte sich das Wurzeldateisystem befindet. Wird dieses Attribut nicht angegeben, wird angenommen, dass das Wurzeldateisystem sich innerhalb der VM unter */dev/sda* befindet.

```
24   <net name="net1" mode="uml_switch"/>
25   <net name="net2" mode="uml_switch"/>
26   <net name="net3" mode="uml_switch"/>
27   <net name="net4" mode="uml_switch"/>
28   <net name="net5" mode="uml_switch"/>
```

Quelltext 14: Definition der Netze in der Szenariodatei

Im nächsten Abschnitt werden die Netze definiert. Jedes <net>-Tag wird vom Startscript als ein eigener vde_switch-Prozess gestartet. Das name-Attribut des <net>-Tags wird dazu verwendet, um die Interfaces der VMs den passenden Netzen zuzuordnen. Das mode-Attribut wird in dieser Version des Startscriptes noch ignoriert.

```
30   <vm name="alpha">
31     <if id="1" net="net1">
32       <ipv4 mask="255.255.255.0">10.1.1.1</ipv4>
33     </if>
34    <if id="2" net="net2">
35       <ipv4 mask="255.255.255.0">10.1.2.1</ipv4>
36     </if>
37     <forwarding type="ipv4" />
38
39     <filetree root="/etc/quagga" seq="start">conf</filetree>
40       <exec seq="start" type="verbatim">hostname</exec>
41       <exec seq="start" type="verbatim">/usr/lib/quagga/zebra -f ↩
             /etc/quagga/zebra.conf -d</exec>
42       <exec seq="start" type="verbatim">/usr/lib/quagga/ripd -f /etc/quagga/ripd.conf ↩
             -d</exec>
43       <exec seq="stop" type="verbatim">hostname</exec>
44       <exec seq="stop" type="verbatim">killall zebra</exec>
45       <exec seq="stop" type="verbatim">killall ripd</exec>
46     </vm>
```

Quelltext 15: Definition der VM alpha in der Szenariodatei

Nach der Deklaration der Netze folgen die Beschreibungen der VMs:

Jedes `<vm>`-Tag hat ein `name`-Attribut, dessen Inhalt als Hostname der VM übernommen wird. Neben den Einstellungen, die auch im `<vm_defaults>`-Tag möglich sind, können hier die Netzwerkschnittstellen und Befehlssequenzen der VM angelegt werden.

Das `<if>`-Tag beschreibt eine Netzwerkschnittstelle einer VM. Es besitzt die Attribute `id` und `net`. `net` entspricht dabei dem Namen eines der zuvor deklarierten Netze, während `id` eine Nummerierung der Interfaces wiedergibt. Innerhalb des `<if>`-Tags können die Tags `<mac>` zur Angabe einer MAC-Adresse, sowie die Tags `<ipv4>` und `<ipv6>` zur Angabe einer IPv4-, bzw. IPv6-Adresse eingetragen werden.
Das Startscript beachtet von diesen jedoch nur das `<ipv4>`-Tag.

Nach den Interfaces folgt das `<forwarding>`-Tag, das bewirkt, dass auf der jeweiligen VM IP Forwarding angeschaltet wird. Dies geschieht durch einen `sysctl`-Aufruf, der über *SSH* innerhalb der VM ausgeführt wird.

Schließlich folgen die `<filetree>` und `<exec>`-Tags.

Die `<filetree>`-Tags bewirken, dass der im Inhalt des Tags angegebene Ordner, bzw. die angegebene Datei auf dem Hostrechner, in das im `root`-Attribut des Tags angegebenen Verzeichnis auf der VM kopiert wird.

Das `seq`-Attribut gibt den Namen einer Befehlssequenz an, vor der diese Aktion ausgeführt wird. Das `root`-Attribut gibt das Zielverzeichnis innerhalb der VM an. Der Inhalt des Tags bestimmt die Datei oder das Verzeichnis, das auf die VM kopiert werden soll.

Das in Quelltext 15 angegebene `<filetree>`-Tag gibt also an, dass vor der Ausführung der Startsequenz `start` das Verzeichnis *`conf`* auf dem Host in das Verzeichnis */etc/quagga* auf der VM kopiert werden soll.

In den `<exec>`-Tags wird jeweils ein Befehl einer Befehlssequenz beschrieben.
Das Attribut `seq` des Tags gibt den Namen der Befehlssequenz an.
Das Attribut `type` gibt an, ob der Inhalt des Tags ein direkt auszuführender Befehl oder der Pfad zu einer Datei ist, deren Inhalt auf der VM ausgeführt werden soll. Es wird vom Startscript jedoch noch nicht unterstützt. Es wird stattdessen angenommen, dass der Inhalt des Tags ein Befehl ist, der direkt auf der VM ausgeführt werden kann.

```
48    <vm name="bravo">
49      <if id="1" net="net1">
50        <ipv4 mask="255.255.255.0">10.1.1.2</ipv4>
51      </if>
52     <if id="2" net="net3">
53        <ipv4 mask="255.255.255.0">10.1.3.1</ipv4>
54      </if>
55      <forwarding type="ipv4" />
          [...]
65     </vm>
66
67    <vm name="charly">
68      <if id="1" net="net2">
69        <ipv4 mask="255.255.255.0">10.1.2.2</ipv4>
70      </if>
71     <if id="2" net="net3">
72        <ipv4 mask="255.255.255.0">10.1.3.2</ipv4>
73      </if>
74     <if id="3" net="net4">
75        <ipv4 mask="255.255.255.0">10.1.4.1</ipv4>
76      </if>
77      <forwarding type="ipv4" />
          [...]
86     </vm>
87
88    <vm name="delta">
89      <if id="1" net="net4">
90        <ipv4 mask="255.255.255.0">10.1.4.2</ipv4>
91      </if>
92     <if id="2" net="net5">
93        <ipv4 mask="255.255.255.0">10.1.5.1</ipv4>
94      </if>
95      <forwarding type="ipv4" />
          [...]
104    </vm>
106
107 </vnuml>
```

Quelltext 16: Definition der verbleibenden VMs in der Szenariodatei

Da die `<exec>`- und `<filetree>`-Tags bei den restlichen VMs denselben Inhalt wie bei der Definition der VM alpha haben, wurden sie hier ausgelassen.

Die hier beschriebene Szenariodatei bildet die in Abbildung 9 visualisierte Topologie ab:

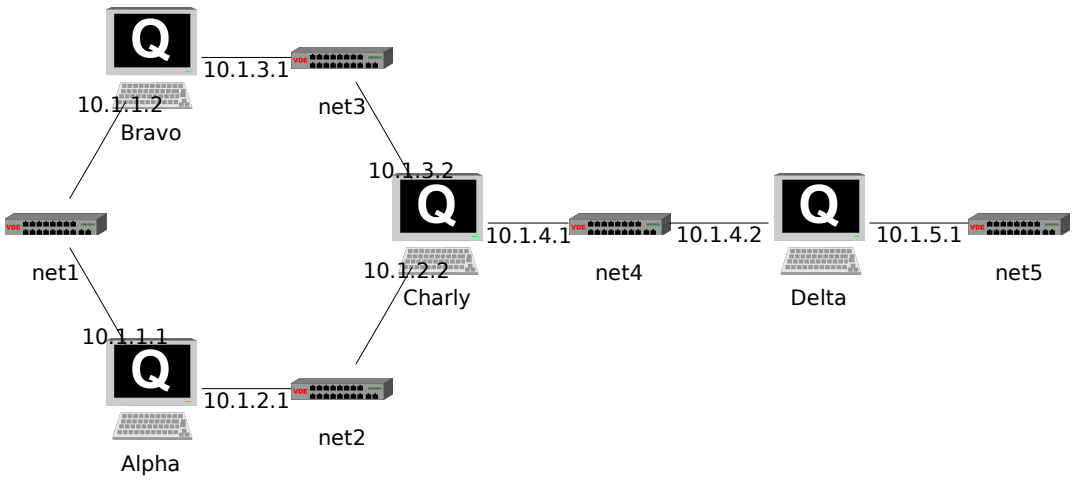

Abbildung 9: Y-Topologie

Ist die Szenariobeschreibung in der Datei *y-topologie.xml* abgespeichert, so kann mit dem Aufruf `./startnet.py start y-topologie.xml` das Startscript dazu angewiesen werden, die Simulation zu starten.

Das Programm wird daraufhin folgende Ausgabe liefern:

```
> # ./startnet.py start y-topologie.xml

*-------------------------------------------*
|starting startNets:  start all vde_switches |
*-------------------------------------------*

starte vde_switches
running: vde_switch -F --sock ./y-topologie/switchnet5.sock --pidfile ↩
    ./y-topologie/switchnet5.pid -M ./y-topologie/switchnet5.msock
running: vde_switch -F --sock ./y-topologie/switchnet4.sock --pidfile ↩
    ./y-topologie/switchnet4.pid -M ./y-topologie/switchnet4.msock
running: vde_switch -F --sock ./y-topologie/switchnet3.sock --pidfile ↩
    ./y-topologie/switchnet3.pid -M ./y-topologie/switchnet3.msock
running: vde_switch -F --sock ./y-topologie/switchnet2.sock --pidfile ↩
    ./y-topologie/switchnet2.pid -M ./y-topologie/switchnet2.msock
running: vde_switch -F --sock ./y-topologie/switchnet1.sock --pidfile ↩
    ./y-topologie/switchnet1.pid -M ./y-topologie/switchnet1.msock
..gestartet

Task startNets executed in 1.01 seconds

*-------------------------------------------------------------------*
|starting createOverlayImages:  create overlay images for all VMs |
```

```
*------------------------------------------------------------------*

erzeuge overlay-images
running: qemu-img create -b ../vm-ressources/images/grml-img.qcow2 -f qcow2 ↩
    ./y-topologie/vm0.ovl
running: qemu-img create -b ../vm-ressources/images/grml-img.qcow2 -f qcow2 ↩
    ./y-topologie/vm1.ovl
running: qemu-img create -b ../vm-ressources/images/grml-img.qcow2 -f qcow2 ↩
    ./y-topologie/vm2.ovl
running: qemu-img create -b ../vm-ressources/images/grml-img.qcow2 -f qcow2 ↩
    ./y-topologie/vm3.ovl

Task createOverlayImages executed in 0.02 seconds

*--------------------------------------*
|starting startVMs:  start all QEMU VMs |
*--------------------------------------*

qemu-prozesse starten
running: qemu -net nic,vlan=0,model=virtio,macaddr=52:54:00:12:34:56 -net ↩
    user,net=192.168.0.0/24,vlan=0,hostfwd=tcp\dots2222-:22 -net ↩
    nic,vlan=1,model=virtio,macaddr=52:54:00:12:34:58 -net ↩
    vde,vlan=1,sock=./y-topologie/switchnet1.sock -net ↩
    nic,vlan=2,model=virtio,macaddr=52:54:00:12:34:59 -net ↩
    vde,vlan=2,sock=./y-topologie/switchnet2.sock --enable-kvm -chardev ↩
    socket,id=monitor,path=./y-topologie/vm0.sock,server,nowait -mon ↩
    chardev=monitor,mode=readline -k de -usbdevice tablet -vnc :0 -pidfile ↩
    ./y-topologie/vm0.pid ./y-topologie/vm0.ovl
running: qemu -net nic,vlan=0,model=virtio,macaddr=52:54:00:12:34:56 -net ↩
    user,net=192.168.0.0/24,vlan=0,hostfwd=tcp\dots2223-:22 -net ↩
    nic,vlan=1,model=virtio,macaddr=52:54:00:12:34:5b -net ↩
    vde,vlan=1,sock=./y-topologie/switchnet1.sock -net ↩
    nic,vlan=2,model=virtio,macaddr=52:54:00:12:34:5c -net ↩
    vde,vlan=2,sock=./y-topologie/switchnet3.sock --enable-kvm -chardev ↩
    socket,id=monitor,path=./y-topologie/vm1.sock,server,nowait -mon ↩
    chardev=monitor,mode=readline -k de -usbdevice tablet -vnc :1 -pidfile ↩
    ./y-topologie/vm1.pid ./y-topologie/vm1.ovl
running: qemu -net nic,vlan=0,model=virtio,macaddr=52:54:00:12:34:56 -net ↩
    user,net=192.168.0.0/24,vlan=0,hostfwd=tcp\dots2224-:22 -net ↩
    nic,vlan=1,model=virtio,macaddr=52:54:00:12:34:5e -net ↩
    vde,vlan=1,sock=./y-topologie/switchnet2.sock -net ↩
    nic,vlan=2,model=virtio,macaddr=52:54:00:12:34:5f -net ↩
    vde,vlan=2,sock=./y-topologie/switchnet3.sock -net ↩
    nic,vlan=3,model=virtio,macaddr=52:54:00:12:34:60 -net ↩
    vde,vlan=3,sock=./y-topologie/switchnet4.sock --enable-kvm -chardev ↩
    socket,id=monitor,path=./y-topologie/vm2.sock,server,nowait -mon ↩
    chardev=monitor,mode=readline -k de -usbdevice tablet -vnc :2 -pidfile ↩
    ./y-topologie/vm2.pid ./y-topologie/vm2.ovl
running: qemu -net nic,vlan=0,model=virtio,macaddr=52:54:00:12:34:56 -net ↩
    user,net=192.168.0.0/24,vlan=0,hostfwd=tcp\dots2225-:22 -net ↩
    nic,vlan=1,model=virtio,macaddr=52:54:00:12:34:62 -net ↩
    vde,vlan=1,sock=./y-topologie/switchnet4.sock -net ↩
    nic,vlan=2,model=virtio,macaddr=52:54:00:12:34:63 -net ↩
    vde,vlan=2,sock=./y-topologie/switchnet5.sock --enable-kvm -chardev ↩
    socket,id=monitor,path=./y-topologie/vm3.sock,server,nowait -mon ↩
    chardev=monitor,mode=readline -k de -usbdevice tablet -vnc :3 -pidfile ↩
    ./y-topologie/vm3.pid ./y-topologie/vm3.ovl

Task startVMs executed in 0.10 seconds

*------------------------------------------------------------------*
```

```
|starting waitOnVMStart:  wait until all VMs can be reached by SSH |
*-----------------------------------------------------------------*

warte, bis VMs gestartet sind
testing virtual machines ['alpha', 'bravo', 'charly', 'delta'] for availability
...gestartet

Task waitOnVMStart executed in 34.30 seconds

*-------------------------------------------------------------------------*
|starting configureInterfaces:  configure hostname and interfaces on all VMs |
*-------------------------------------------------------------------------*

setze Hostnamen und IP-Adressen
running "`hostname alpha"' on host alpha
running "`ifconfig eth1 10.1.1.1 netmask 255.255.255.0"' on host alpha
running "`ifconfig eth2 10.1.2.1 netmask 255.255.255.0"' on host alpha
running "`sysctl -w net.ipv4.ip_forward=1"' on host alpha
net.ipv4.ip_forward = 1
running "`hostname bravo"' on host bravo
running "`ifconfig eth1 10.1.1.2 netmask 255.255.255.0"' on host bravo
running "`ifconfig eth2 10.1.3.1 netmask 255.255.255.0"' on host bravo
running "`sysctl -w net.ipv4.ip_forward=1"' on host bravo
net.ipv4.ip_forward = 1
running "`hostname charly"' on host charly
running "`ifconfig eth1 10.1.2.2 netmask 255.255.255.0"' on host charly
running "`ifconfig eth2 10.1.3.2 netmask 255.255.255.0"' on host charly
running "`ifconfig eth3 10.1.4.1 netmask 255.255.255.0"' on host charly
running "`sysctl -w net.ipv4.ip_forward=1"' on host charly
net.ipv4.ip_forward = 1
running "`hostname delta"' on host delta
running "`ifconfig eth1 10.1.4.2 netmask 255.255.255.0"' on host delta
running "`ifconfig eth2 10.1.5.1 netmask 255.255.255.0"' on host delta
running "`sysctl -w net.ipv4.ip_forward=1"' on host delta
net.ipv4.ip_forward = 1

Task configureInterfaces executed in 5.42 seconds

*-------------------------------------------------------------*
|starting saveState:  save the current state in a YAML dump |
*-------------------------------------------------------------*

Task saveState executed in 0.09 seconds
startVMs called 1 time(s) with a total execution time of 0.10 seconds
saveState called 1 time(s) with a total execution time of 0.09 seconds
configureInterfaces called 1 time(s) with a total execution time of 5.42 seconds
createOverlayImages called 1 time(s) with a total execution time of 0.02 seconds
waitOnVMStart called 1 time(s) with a total execution time of 34.30 seconds
startNets called 1 time(s) with a total execution time of 1.01 seconds
```

Quelltext 17: Ausgabe des Startscriptes beim Start einer Simulation

Nachdem die Netzwerksimulation nun gestartet ist, kann mit *SSH* auf die virtuellen Maschinen zugegriffen werden. Es ist sinnvoll, sich vorher mit dem Befehl `./startnet show y-topologie.xml` eine Übersicht über die Konfiguration des virtuellen Netzwerkes geben zu lassen:

```
> # ./startnet.py show y-topologie.xml
alpha:
   SSH @ Host Port: 2222
   Interfaces:
   Address:10.1.1.1 Netmask:255.255.255.0 MAC:52:54:00:12:34:58 - Connected to Net net1
   Address:10.1.2.1 Netmask:255.255.255.0 MAC:52:54:00:12:34:59 - Connected to Net net2
   PID: 6744 - Monitor: ./y-topologie/vm0.sock
bravo:
   SSH @ Host Port: 2223
   Interfaces:
   Address:10.1.1.2 Netmask:255.255.255.0 MAC:52:54:00:12:34:5b - Connected to Net net1
   Address:10.1.3.1 Netmask:255.255.255.0 MAC:52:54:00:12:34:5c - Connected to Net net3
   PID: 6745 - Monitor: ./y-topologie/vm1.sock
charly:
   SSH @ Host Port: 2224
   Interfaces:
   Address:10.1.2.2 Netmask:255.255.255.0 MAC:52:54:00:12:34:5e - Connected to Net net2
   Address:10.1.3.2 Netmask:255.255.255.0 MAC:52:54:00:12:34:5f - Connected to Net net3
   Address:10.1.4.1 Netmask:255.255.255.0 MAC:52:54:00:12:34:60 - Connected to Net net4
   PID: 6746 - Monitor: ./y-topologie/vm2.sock
delta:
   SSH @ Host Port: 2225
   Interfaces:
   Address:10.1.4.2 Netmask:255.255.255.0 MAC:52:54:00:12:34:62 - Connected to Net net4
   Address:10.1.5.1 Netmask:255.255.255.0 MAC:52:54:00:12:34:63 - Connected to Net net5
   PID: 6748 - Monitor: ./y-topologie/vm3.sock

Network net5: mode: uml_switch
   PID: 6735 - Comm Socket: ./y-topologie/switchnet5.sock - MGMT Socket: ↩
       ./y-topologie/switchnet5.msock
Network net4: mode: uml_switch
   PID: 6736 - Comm Socket: ./y-topologie/switchnet4.sock - MGMT Socket: ↩
       ./y-topologie/switchnet4.msock
Network net3: mode: uml_switch
   PID: 6737 - Comm Socket: ./y-topologie/switchnet3.sock - MGMT Socket: ↩
       ./y-topologie/switchnet3.msock
Network net2: mode: uml_switch
   PID: 6738 - Comm Socket: ./y-topologie/switchnet2.sock - MGMT Socket: ↩
       ./y-topologie/switchnet2.msock
Network net1: mode: uml_switch
   PID: 6739 - Comm Socket: ./y-topologie/switchnet1.sock - MGMT Socket: ↩
       ./y-topologie/switchnet1.msock
```

Quelltext 18: Ausgabe der show-Aktion des Startscriptes

Aus der Ausgabe ist zu erfahren, dass der *SSH*-Port der VM alpha auf dem Host-Port 2222 liegt, der *SSH*-Port von bravo auf 2223, charly auf Port 2224 und delta auf 2225. Um nun eine *SSH*-Verbindung zu der VM alpha aufzubauen, muss sich nun zu Port 2222 auf dem Hostrechner verbunden werden:

```
ssh -p2222 -oUserKnownHostsFile=./knownhosts -i hostkey root@localhost
```

Die Option `-p2222` gibt dem *SSH*-Client an, dass der Port des *SSH*-Servers auf dem Port 2222 liegt. Durch die Option `-oUserKnownHostsFile=./knownhosts` wird nicht die knownhosts-Datei im Homeverzeichnis des Benutzers verwendet, sondern eine separate im lokalen Verzeichnis.

Durch `-i hostkey` wird ein privater Schlüssel angegeben, dessen zugehöriger öffentlicher Schlüssel bereits vorher im Festplattenimage eingetragen wurde. Dadurch entfällt die Notwendigkeit, ein Passwort eingeben zu müssen. Näheres hierzu in Kapitel 2.6.1.

Um nun eine der in der XML-Datei definierten Startsequenzen auszuführen, kann die Aktion `exec` des Startscriptes eingesetzt werden. Durch die Eingabe der Aktion `exec` ohne weitere Optionen wird jedoch nur ausgegeben, welche Befehlssequenzen durch `<filetree>`- und `<exec>`-Tags in der XML-Datei definiert wurden.

```
> # ./startnet.py exec ctinetz.xml
available command sequences
@alpha: stop, start
@bravo: stop, start
@charly: stop, start
@delta: stop, start

available filetrees
@alpha: start
@bravo: start
@charly: start
@delta: start
```

Quelltext 19: Ausgabe der exec-Aktion des Startscriptes

Nur durch zusätzliche Angabe einer `-x`-Option mit dem Namen einer Befehlssequenz wird diese ausgeführt. So führt der Befehl `./startnet exec y-topologie.xml -x start` zunächst den Filetree und anschließend die Befehlssequenz „start“ auf allen VMs der Simulation aus. Im vorliegenden Szenario bedeutet das, dass auf jeder VM ein RIP-Router gestartet wird. Folglich kann nun jede VM jede andere VM erreichen, selbst wenn sie nicht direkt miteinander verbunden sind:

```
root@alpha ~ # route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.0.0     *               255.255.255.0   U     0      0        0 eth0
10.1.4.0        10.1.2.2        255.255.255.0   UG    2      0        0 eth2
10.1.5.0        10.1.2.2        255.255.255.0   UG    3      0        0 eth2
10.1.1.0        *               255.255.255.0   U     0      0        0 eth1
10.1.2.0        *               255.255.255.0   U     0      0        0 eth2
10.1.3.0        10.1.1.2        255.255.255.0   UG    2      0        0 eth1
default         192.168.0.2     0.0.0.0         UG    0      0        0 eth0

root@alpha ~ # traceroute 10.1.5.1
traceroute to 10.1.5.1 (10.1.5.1), 30 hops max, 60 byte packets
 1  10.1.2.2 (10.1.2.2)  2.292 ms  2.895 ms  3.143 ms
 2  10.1.5.1 (10.1.5.1)  6.801 ms  6.827 ms  7.129 ms
```

Quelltext 20: Ausgaben von route und traceroute

Schließlich kann durch die Aktion „stop“ die Simulation beendet werden:

```
> # ./startnet.py stop y-topologie.xml

*-------------------------------------------------------*
|starting stopVMs:  send "'halt"' to all QEMU Processes |
*-------------------------------------------------------*

running "'halt"' on host alpha
running "'halt"' on host bravo
running "'halt"' on host charly
running "'halt"' on host delta

Task stopVMs executed in 1.60 seconds

*-------------------------------------------------------------------------*
|starting waitOnVMStop:  wait until all VMs can't be reached by SSH anymore |
*-------------------------------------------------------------------------*

*-------------------------------------------------------------------------*
|starting waitOnVMStop:  wait until all VMs can't be reached by SSH anymore |
*-------------------------------------------------------------------------*

*-------------------------------------------------------------------------*
|starting waitOnVMStop:  wait until all VMs can't be reached by SSH anymore |
*-------------------------------------------------------------------------*

*-------------------------------------------------------------------------*
|starting waitOnVMStop:  wait until all VMs can't be reached by SSH anymore |
*-------------------------------------------------------------------------*

Task waitOnVMStop executed in 0.00 seconds

Task waitOnVMStop executed in 0.43 seconds

Task waitOnVMStop executed in 2.07 seconds

Task waitOnVMStop executed in 8.99 seconds

*-----------------------------------------------------------*
|starting killVMs:  terminate all remaining QEMU processes |
*-----------------------------------------------------------*

Task killVMs executed in 1.00 seconds

*-------------------------------------------------------------------------------------------------↩
     ---------------*
|starting cleanupVMs:  kill all remaining QEMU processes and remove the overlay images↩
     and PID-files |
*-------------------------------------------------------------------------------------------------↩
     ---------------*

Task cleanupVMs executed in 0.03 seconds

*-------------------------------------------------*
|starting stopNets:  kill all vde_switch processes |
*-------------------------------------------------*

Task stopNets executed in 1.00 seconds

*-------------------------------------------------------------------------------- ↩
    ----------------------*
```

```
|starting cleanupNets:  kill all remaining QEMU processes and remove the overlay ↩
    images and PID-files |
*------------------------------------------------------------------------------- ↩
    ----------------------*

Task cleanupNets executed in 0.00 seconds

*------------------------------------------------------------------------------- ↩
    -----------*
|starting cleanupHost:  if possible remove created files, the dump file and the work ↩
    directory |
*------------------------------------------------------------------------------- ↩
    -----------*

Task cleanupHost executed in 0.00 seconds
cleanupVMs called 1 time(s) with a total execution time of 0.03 seconds
stopVMs called 1 time(s) with a total execution time of 1.60 seconds
cleanupHost called 1 time(s) with a total execution time of 0.00 seconds
killVMs called 1 time(s) with a total execution time of 1.00 seconds
waitOnVMStop called 4 time(s) with a total execution time of 11.49 seconds
stopNets called 1 time(s) with a total execution time of 1.00 seconds
cleanupNets called 1 time(s) with a total execution time of 0.00 seconds
total execution time: 15.13 seconds
```

Quelltext 21: Ausgabe des Startscriptes beim Beenden der Simulation

7. Fazit

In der vorliegenden Diplomarbeit wurde ein Programm entwickelt, das virtuelle Netzwerke, deren Topologie in einer XML-Datei beschrieben wird, mithilfe des Open-Source-Virtualisierers *QEMU* und virtuellen Netzwerkswitches aufbauen kann.

Messungen haben gezeigt, dass das neu entwickelte System schnellere Startzeiten von virtuellen Netzwerken ermöglicht und einen geringeren Ressourcenbedarf hat als das *VNUML*-System. *VNUML* basiert auf User Mode Linux und wird im Gegensatz zu QEMU und KVM zum Zeitpunkt der Erstellung dieser Arbeit nicht mehr aktiv weiterentwickelt. Während *QEMU* und *KVM*, die Virtualisierungserweiterung zu *QEMU*, unter den verbreiteten Linux-Distributionen gut unterstützt sind, ist es schwierig, User Mode Linux optimal zum Laufen zu bringen: Damit die Performance eines User Mode Linux-Systems optimal ist, muss der Kernel des Hostsystems mit dem SKAS-Patch gepatcht sein. Die Variante dieses Patches (SKAS3), die den größten Performancegewinn bringt, funktioniert jedoch in aktuellen Linux-Kerneln nur auf 32-Bit-Systemen.

Die Codebasis der in Python programmierten Neuentwicklung ist überschaulich und gut erweiterbar. Es wurde Wert darauf gelegt, den Code gut zu dokumentieren, so dass weitere Arbeiten an dem System möglich sind.

Bei einer Erweiterung des Systems sollte versucht werden, die Robustheit des Startscriptes gegenüber Bedienungfehlern zu verbessern und bei Problemen aussagekräftige Fehlermeldungen auszugeben. Zudem gibt es noch einige XML-Tags, deren Implementation im Startscript lohnenswert wäre: Zum einen sind einige *VNUML*-Tags noch nicht unterstützt, wie zum Beispiel die Tags, die mit IPv6 zusammenhängen. Auch ist es noch nicht möglich, das Startscript zum

automatisierten Kopieren der vorhandenen *SSH*-Keys des Anwenders auf die VMs zu benutzen, wodurch der zum Einloggen in die VMs notwendige `ssh`-Aufruf weniger kompliziert würde (Siehe hierzu auch Kapitel 2.6 auf Seite 12). Zum anderen sollte es mithilfe des `<vde_plug>`-Tags möglich sein, mehrere *VDE-Switches* miteinander zu verbinden, um so geswitchte Netze zu bilden.

Als Ersatz für das momentan genutzte Management-Interface über das User-Netzwerk-Interface von *QEMU* kann ein Management-Interface über TUN/TAP-Devices implementiert werden. Der sich daraus ergebende Vorteil wäre, dass die VMs vom Host über IP-Adressen, bzw. bei zusätzlicher Manipulation der Hosts-Datei über ihre Hostnamen erreichbar wären. Hierzu muss jedoch auch sichergestellt sein, dass die Hosts-Datei nach Abbau des virtuellen Netzwerks wieder auf ihren ursprünglichen Zustand zurückgesetzt werden kann.

Weiterhin wäre es sinnvoll, ein optimiertes Festplattenimage für *QEMU* zu bauen, das den Speicherverbrauch und die Startzeit pro VM minimiert.

A. Glossar

DHCP Dynamic Host Configuration Protocol - Protokoll, durch das Netzwerkschnittstellen von mehreren Netzwerkteilnehmern zentral konfiguriert werden können

Docstrings Kommentare, die vom Python-Interpreter verarbeitet werden

DTD Document Type Declaration - Schemadatei, die die Struktur eines XML-Dokuments beschreibt

Festplattenimage Abbild des Inhaltes einer Festplatte in einer Datei, beinhaltet in diesem Fall ein Betriebssystem

Hypervisor Software, die auf niedriger Ebene arbeitet und virtuelle Maschinen verwaltet

KSM Kernel Shared Memory - Technik in neueren Linux-Kerneln, die durch Durchsuchen des Arbeisspeichers nach duplizierten Speicherseiten und Zusammenführen derselben, den Speicherverbrauch des Systems reduzieren kann, insbesondere beim Ausführen mehrerer gleichartiger VMs

KVM Kernel Virtual Machine - Modul für den Linux-Kernel, der durch dieses Modul als Hypervisor für VMs agieren und so VMs verwalten kann

Man-in-the-middle-Angriff Angriff auf die Kommunikation zwischen zwei Netzwerkteilnehmern, bei dem der Angreifer die Kommunikation zwischen den Teilnehmern abfängt und sich den Teilnehmern gegenüber als der jeweilige Kommunikationspartner ausgibt.

PID process identifier - Nummer, die einen Prozess eines Betriebssystems identifiziert

QEMU „Quick Emulator“ - Emulator und Virtualisierer

SKAS Separate Kernel Address Space - Patch für den Linux-Kernel, der zur effizienten Nutzung von User Mode Linux notwendig ist

smem Programm, das den anteiligen Speicherverbrauch eines Programms bestimmen kann.

SSH Secure Shell - Programm, das die Bedienung einer Shell eines Rechners über das Netzwerk erlaubt

STP Spanning Tree Protocol - Protokoll, das von Netzwerkswitches zur Schleifenvermeidung genutzt wird

TUN/TAP-Device virtuelle Netzwerkgeräte, ursprünglich gedacht um Tunnel zu erzeugen, können jedoch auch als Verbindung zu den Netzwerkschnittstellen von virtuellen Maschinen genutzt werden

UML User Mode Linux - Linux-Kernel, der als Anwendungsprogramm kompiliert wurde und somit ermöglicht, Linux-Maschinen innerhalb eines anderen Systems zu starten

VDE-Switch Teil des VDE-Pakets - Software, die einen managebaren Netzwerkswitch simuliert

VDE „Virtual Distributed Ethernet“ - Teil des VirtualSquare-Projekts der University of Bologna

VM virtuelle Maschine - ein virtuelles Computersystem, das parallel zu anderen VMs auf einem physikalischen Computersystem laufen kann.

VNC Virtual Network Computing - System, durch das ein Rechner über das Netzwerk ferngesteuert werden kann. Im Gegensatz zu SSH werden hier der gesamte Bildschirminhalt sowie Tastatur- und Mauseingaben übertragen

VNUML Virtual Network User Mode Linux - Netzwerksimulator auf Basis von User Mode Linux

XML Extensible Markup Language - Einfache und erweiterbare Sprache, die menschen- und maschinenlesbar ist und dem einfachen Austausch von Informationen dient

Literatur

[1] The User-mode Linux Kernel Home Page: *skas mode*. http://user-mode-linux.sourceforge.net/old/skas.html, 2003. – [Online; Stand 13. September 2011]

[2] *schematron*. http://www.schematron.com/, 2010. – [Online; Stand 19. September 2011]

[3] *Introducing JSON*. http://www.json.org, 2011. – [Online; Stand 19. September 2011]

[4] *Kernel Based Virtual Machine*. http://www.linux-kvm.org, 2011. – [Online; Stand 13. September 2011]

[5] Behnel, Stefan: *lxml - XML and HTML with Python*. http://lxml.de, 2011. – [Online; Stand 13. September 2011]

[6] Behnel, Stefan: *The lxml.etree Tutorial*. http://lxml.de/tutorial.html#tree-iteration: lxml.de, 3 2011. – [Online; Stand 13. September 2011]

[7] Bellard, F.: QEMU, a fast and portable dynamic translator. In: *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, S. 41–46

[8] Bellard, Fabrice: *qemu-doc - QEMU Emulator User Documentation*, 7 2011

[9] Cahalan, Albert: *pmap(1) - Linux man page*. 10 2002

[10] David: *Python Tips, Tricks, and Hacks*. http://www.siafoo.net/article/52, 4 2011. – [Online; Stand 13. September 2011]

[11] Devin: *Understanding memory usage on Linux*. http://virtualthreads.blogspot.com/2006/02/understanding-memory-usage-on-linux.html: Virtual Threads Blog, 2 2006. – [Online; Stand 13. September 2011]

[12] Dike, Jeff: *User Mode Linux*. Prentice Hall, 2006

[13] Edmonds, Brian: *free(1) - Linux man page*. 10 2009

[14] Eidus, I. ; Dickins, H.: *How to use the Kernel Samepage Merging feature*, 11 2009. – innerhalb eines Kernel-Sourcecodes unter `Documentation/vm/ksm.txt`

[15] Fallside, David C. ; Walmsley, Priscilla: *XML Schema Part 0: Primer Second Edition*. http://www.w3.org/TR/xmlschema-0/, 4 2010. – [Online; Stand 19. September 2011]

[16] Foundation, Python S.: *3. Built-in Constants – Python v3.2.2 documentation*. http://docs.python.org/py3k/library/constants.html#None, 2011. – [Online; Stand 13. September 2011]

[17] Galán, Fermín ; Deccio, Casey T.: *VNUML Language Reference*. http://neweb.dit.upm.es/vnumlwiki/index.php/Reference_1.8: Departamento de Ingeniería de Sistemas Telemáticos (DIT) of the Universidad Politécnica de Madrid (UPM), 9 2008. – [Online; Stand 13. September 2011]

[18] Harold, E.R. ; Means, W.S.: *XML in a nutshell*. http://books.google.de/books?id=NBwnSfoCStAC : O'Reilly, 2004 (In a nutshell). – ISBN 9780596007645

[19] Hess, Joey: */etc/network/interfaces - network interface configuration for ifup and ifdown*, 4 2004

[20] Jones, M. T.: Anatomy of Linux Kernel Shared Memory. (2010), 4. – [Online; Stand 13. September 2011]

[21] Kirby, Dave: *Can a Python Decorator of an Instance Method Access the Class?* http://stackoverflow.com/questions/2366713/can-a-python-decorator-of-an-instance-method-access-the-class/2367605#2367605, 3 2010. – [Online; Stand 23. August 2011]

[22] Krasnyansky, Maxim: *Univesal TUN/TAP driver - FAQ.* http://vtun.sourceforge.net/tun/faq.html, 2001. – [Online; Stand 15. September 2011]

[23] Lankester, Branko ; Johnson, Michael K. ; Shields, Michael ; Blake, Charles ; Mossberger-Tang, David ; Cahalan, Albert: *ps(1) - Linux man page.* 2 2010

[24] McLoughlin, Mark: *The QCOW2 Image Format.* http://people.gnome.org/~markmc/qcow-image-format.html, 9 2008. – [Online; Stand 12. September 2011]

[25] Murata, Makoto: *RELAX NG home page.* http://relaxng.org/, 1 2011. – [Online; Stand 19. September 2011]

[26] *Kapitel* Host-Konfiguration (DHCP). In: Peterson, L.L. ; Davie, B.S. ; Shafir, A.: *Computernetze: Eine systemorientierte Einführung.* Dpunkt.verl., 2007. – ISBN 9783898644914, S. 261, f.

[27] *Kapitel* Beschränkungen von Switches. In: Peterson, L.L. ; Davie, B.S. ; Shafir, A.: *Computernetze: Eine systemorientierte Einführung.* Dpunkt.verl., 2007. – ISBN 9783898644914, S. 194

[28] *Kapitel* Übersetzung von Netzadressen (NAT). In: Peterson, L.L. ; Davie, B.S. ; Shafir, A.: *Computernetze: Eine systemorientierte Einführung.* Dpunkt.verl., 2007. – ISBN 9783898644914, S. 331, f.

[29] *Kapitel* Multicast. In: Peterson, L.L. ; Davie, B.S. ; Shafir, A.: *Computernetze: Eine systemorientierte Einführung.* Dpunkt.verl., 2007. – ISBN 9783898644914, S. 333, f.

[30] Pilgrim, Mark: *Dive Into Python 3.* http://www.diveintopython3.org : Apress, 2009. – [Online; Stand 13. September 2011]

[31] Project, Virtual S.: *Introduction - VirtualSquare.* http://wiki.virtualsquare.org/wiki/index.php/Introduction&oldid=1512#The_Virtual_Square, 6 2011. – [Online; Stand 13. September 2011]

[32] Project, Virtual S.: *VDE Components - vde_plug - wirefilter.* http://wiki.virtualsquare.org/wiki/index.php/VDE&oldid=1502#wirefilter, 1 2011. – [Online; Stand 13. September 2011]

[33] qemu-buch.de (Hrsg.): *QEMU and Slirp.* http://qemu-buch.de/cgi-bin/moin.cgi/QemuAndSlirp: qemu-buch.de, 7 2010. – [Online; Stand 13. September 2011]

[34] *Kapitel* Authentifizierungsmethoden. In: Riedel, S.: *SSH kurz & gut.* http://books.google.de/books?id=1u118Uq1KD8C : O'Reilly Vlg. GmbH & Co., 2004. – ISBN 9783897212695, S. 102

[35] Shalom, H: *Linux Memory Consumption*. http://www.rt-embedded.com/blog/archives/linux-memory-consumption/: Real-Time Embedded Blog, 4 2010. – [Online; Stand 13. September 2011]

[36] Srivastava, Manoj: *hosts(5) - Linux man page*. 6 2002

[37] VNUML-Team: *All News*. http://neweb.dit.upm.es/vnumlwiki/index.php/All_News: Departamento de Ingeniería de Sistemas Telemáticos (DIT) of the Universidad Politécnica de Madrid (UPM), 11 2009. – [Online; Stand 13. September 2011]

[38] VNUML-Team: *File Release Notes and Changelog: root_fs_tutorial-0.6.0*. http://sourceforge.net/project/shownotes.php?group_id=113582&release_id=684261, 5 2009. – [Online; Stand 13. September 2011]

[39] Warner, James C.: *top(1) - Linux man page*. 9 2002

[40] Warsaw, Berry: *PEP 202 – List Comprehensions*. http://www.python.org/dev/peps/pep-0202/, 10 2008. – [Online; Stand 13. September 2011]

[41] Wikipedia: *Extensible Markup Language*. http://de.wikipedia.org/w/index.php?title=Extensible_Markup_Language&oldid=92522511, 2011. – [Online; Stand 23. August 2011]

[42] Wikipedia: *Initrd*. http://en.wikipedia.org/w/index.php?title=Initrd&oldid=447436668, 2011. – [Online; Stand 19. September 2011]

[43] Wikipedia: *MAC-Adresse*. http://de.wikipedia.org/w/index.php?title=MAC-Adresse&oldid=92364228, 2011. – [Online; Stand 19. September 2011]

[44] Wikipedia: *Programmbibliothek*. http://de.wikipedia.org/w/index.php?title=Programmbibliothek&oldid=93090635, 2011. – [Online; Stand 13. September 2011]

[45] Wikipedia: *Ring network*. http://en.wikipedia.org/w/index.php?title=Ring_network&oldid=446248214, 2011. – [Online; Stand 26. August 2011]

[46] Wikipedia: *UTF-8*. http://de.wikipedia.org/w/index.php?title=UTF-8&oldid=92937351, 2011. – [Online; Stand 15. September 2011]

[47] Ylonen, Tatu ; Campbell, Aaron ; Beck, Bob ; Friedl, Markus ; Provos, Niels ; Raadt, Theo de ; Song, Dug: *ssh(1) - OpenBSD Reference Manual*. 9 2011

[48] Ylonen, Tatu ; Campbell, Aaron ; Beck, Bob ; Friedl, Markus ; Provos, Niels ; Raadt, Theo de ; Song, Dug: *ssh_config(5) - OpenBSD Programmer's Manual*. 9 2011

Abbildungsverzeichnis

Quelltextverzeichnis