Testen von verteilten Systemen in heterogenen Netzwerken mit virtuellen Maschinen am Beispiel von MPI

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Informatiker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik

eingereicht von Wolfgang Schnerring
geboren am 20.04.1981 in Tübingen

Betreuer: Dipl-Inf. Christian Kauhaus
Prof. Dr.-Ing. Dietmar Fey

Jena, 25. Mai 2007

Kurzfassung

Verteilte Systeme sind in zunehmendem Maße mit komplexen Netzwerkkonfigurationen konfrontiert, beispielsweise mit mehreren Netzwerkschnittstellen pro Knoten, NAT und gemischter IPv4/IPv6-Adressierung. Um Unterstützung dafür korrekt zu implementieren, ist regelmäßiges und gründliches Testen unerlässlich. Allerdings ist der Aufwand, derart vielfältige Teststellungen in Hardware vorzuhalten, so hoch, dass Testen nur sehr selten erfolgen kann und oftmals ganz unterbleibt. Gegenstand dieser Diplomarbeit ist die Entwicklung eines *Virtual Test Environment* (VTE), das (Multi-)Cluster-Umgebungen mit Hilfe von virtuellen Maschinen nachbildet und somit den erforderlichen Aufwand so weit senkt, dass kontinuierliches, entwicklungsbegleitendes Testen praktikabel wird.

Der Quellcode des Virtual Test Environment befindet sich auf der beigefügten CD-ROM sowie unter `https://cluster.inf-ra.uni-jena.de/svn/mcm/wosc-diplom/vte/`.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Kürzel	Beschreibung
IP	Internet Protocol
MAC	Media Access Control
MPI	Message Passing Interface
NAT	Network Address Translation
NFS	Network File System
SSH	Secure Shell
TCP	Transmission Control Protocol
VM	Virtuelle Maschine
VPN	Virtual Private Network
VTE	Virtual Test Environment

1 Einleitung

Verteilte Systeme sind inhärent komplex. Daher können vermeintlich lokale Änderungen weitreichende Auswirkungen sogar auf weit entfernte Teile des Systems haben, die im Vorhinein praktisch nicht absehbar sind. Um solche Effekte erkennen und eingrenzen zu können, ist es nötig, das System kontinuierlich Tests zu unterziehen, die sicherstellen, dass es sich so verhält wie erwartet. Natürlich können Tests keinen vollständigen Beweis der Korrektheit erbringen, aber sie sind ein entscheidendes Mittel, um die Unwägbarkeit in einem gewissen Rahmen einzuschränken.

Neben der inhärenten Komplexität kommt hinzu, dass die Netzwerkstrukturen, die verteilten Systemen zu Grunde liegen, immer komplexer werden. Heutige Cluster oder gar Multi-Domain-Cluster (im Folgenden kurz Multicluster) weisen vielfältige Netzwerkkonfigurationen auf, etwa *Network Address Translation* (NAT) oder gemischte IPv4/IPv6-Adressierung. Da Anwendungsentwickler erwarten, dass Programmierumgebungen wie die Message-Passing-Bibliothek MPI diese Komplexität kapseln und eine einheitliche Schnittstelle anbieten, sind diese Bibliotheken nun mit komplexen, oftmals heterogenen Netzwerkumgebungen konfrontiert. Daher enthalten praktisch alle MPI-Implementierungen größere Mengen an Verbindungslogik, um die Kommunikation zwischen den Rechnern herzustellen, Daemons zu starten, Adressen zu bestimmen, Netzwerkschnittstellen auszuwählen und einiges mehr.

Um diese Funktionalität korrekt zu implementieren, ist regelmäßiges, gründliches Testen essenziell. Dazu ist es erforderlich, die Software tatsächlich mit den unterschiedlichen Netzwerktopologien zu konfrontieren. Aber selbst wenn nur ein Teilausschnitt aller möglichen zu unterstützenden Netzwerkkonfigurationen zum Test herangezogen wird, ist es mit zumeist unerschwinglichem materiellen wie personellen Aufwand verbunden, diese in Hardware als Testumgebung aufzubauen. Deshalb sind solche Tests gegenwärtig nicht praktikabel. Dieser Mangel an systematischem Testen führt häufig zu unentdeckten Fehlern, die nur schwer zu lokalisieren und zu beheben sind.

Gegenstand dieser Arbeit ist daher der Entwurf und die Implementierung eines *Virtual Test Environment* (VTE), das (Multi-)Cluster-Umgebungen mit Hilfe von virtuellen Maschinen nachbildet. Es soll in kurzer Zeit anhand von einfachen Beschreibungen automatisiert verschiedene Netzwerkkonfigurationen erzeugen, in

denen dann verteilt arbeitende Software auf ihre korrekte Funktionsweise überprüft werden kann. VTE ist ein Steuerprogramm, das virtuelle Maschinen erzeugt, diese mit einem virtuellen Netzwerk verbindet und die zu testende Software in dieser virtuellen Umgebung ausführt. Mit VTE soll der Aufwand für das Testen von verteilten Systemen so weit gesenkt werden, dass kontinuierliches, entwicklungsbegleitendes Testen umsetzbar wird. Obwohl VTE schwerpunktmäßig im Hinblick auf die Arbeit des Lehrstuhls an Open MPI entwickelt wurde, ist sein Einsatz in keiner Weise darauf beschränkt. VTE eignet sich im Gegenteil für das Testen von allen verteilten Systemen, die mit einer TCP/IP-Netzwerkumgebung interagieren.

Diese Arbeit gliedert sich wie folgt: Kapitel 2 stellt Grundlagen des Testens und virtueller Maschinen vor. Kapitel 3 erläutert den Entwurf und die Implementierung von VTE im Detail. Kapitel 4 illustriert die Testmöglichkeiten mit VTE anhand von Fallbeispielen und untersucht seine Leistungsfähigkeit. Kapitel 5 fasst zusammen und betrachtet mögliche Weiterentwickungen von VTE.

2 Grundlagen

VTE ist eine Testumgebung, die auf virtuellen Maschinen basiert. Um das Verständnis der anschließenden Kapitel zu erleichtern, werden in den zwei folgenden Abschnitten zunächst die zu Grunde liegenden Konzepte des Testens sowie die verfügbaren Technologien zur Virtualisierung vorgestellt.

2.1 Testen

Testen ist ein zentraler Bestandteil von Software-Qualitätssicherung. Neben Code-Reviews und der oftmals unpraktikablen theoretischen Verifikation stellt es das wesentliche Mittel zur Überprüfung der Produktqualität dar [1].

Damit Testen einen effektiven Beitrag zur Qualitätssicherung leisten kann, muss es regelmäßig und entwicklungsbegleitend stattfinden. Es ist wesentlich einfacher, die Ursache für einen Fehler einzugrenzen, wenn die Menge der dafür in Frage kommenden Änderungen an der Software überschaubar ist. Insbesondere in komplexen Systemen, in denen der Zusammenhang von Ursache und Wirkung oftmals nur schwer aufzudecken ist, ist es daher empfehlenswert, in kleinen Schritten zyklisch vorzugehen, und die Tests in kurzen Abständen laufen zu lassen. Wenn dann ein Fehler auftritt, kann die Menge der Änderungen, die dafür verantwortlich sein könnten, besser überblickt werden [2, 3].

Eine solche kontinuierliche Rückkopplung durch entwicklungsbegleitendes Testen kann nur entstehen, wenn der Aufwand für das Testen gering ist. „Aufwand“ ist in diesem Zusammenhang kein eng definierter Begriff, sondern kann verschiedene Formen annehmen, beispielsweise materieller, personeller oder zeitlicher Aufwand, unter Umständen auch der kognitive Aufwand für den Entwickler. Der Grundsatz ist daher: Die Hürde zum Testen muss niedrig liegen. Es muss einfach sein, Tests zu erstellen, und einfach sein, die Tests auszuführen. Wenn stets erst eine umständliche Prozedur oder gar eigenhändiges Ausprobieren erforderlich ist, dann besteht die Gefahr, dass das Testen für den Entwickler zu mühselig ist, und deswegen nicht durchgeführt wird. Deshalb sollten die Tests vollautomatisch ablaufen, sodass die aktuelle Information über das Verhalten der Software jederzeit auf Knopfdruck verfügbar ist. Das Ergebnis der Tests sollte ein klares „bestanden“ oder „nicht bestanden“ sein, nicht etwa eine lange Liste von Messwerten, die erst

noch interpretiert werden müssen, um herauszufinden, ob das System korrekt arbeitet oder nicht. Um häufiges Testen zu ermöglichen, darf die Ausführung der Tests nicht zu lange dauern, da das Testen sonst die Konzentration des Entwicklers zu stark unterbricht und damit den Entwicklungsfluss erheblich stört.

Um dies zu ermöglichen, sind Werkzeuge nötig, die den Entwickler bei der Formulierung und Ausführung von Tests unterstützen. Auf der Ebene von Unit-Tests gibt es eine Vielzahl von solchen Werkzeugen, etwa die xUnit-Testframeworks [4] oder Mock Objects [5]. Aber Unit-Tests reichen nicht aus, um die Funktionalität eines Systems zu überprüfen, denn diese ergibt sich insgesamt erst aus dem Zusammenspiel aller Komponenten. Nur weil diese einzeln wie gefordert funktionieren, ist eine korrekte Integration nicht garantiert. Insbesondere bei verteilten Systemen, die in hohem Maß mit ihrer Umgebung interagieren, ist es unerlässlich, funktionale Tests in einer möglichst realistischen Umgebung durchzuführen.

Dazu gibt es, gerade im MPI-Umfeld, Werkzeuge, die funktionales Testen unterstützen, besonders im Hinblick auf einige der Problembereiche, die sich durch Nebenläufigkeit ergeben [6]. Zur Erkennung von Deadlocks beispielsweise klinken sich Programme wie Umpire [7] und Marmot [8] in die Profiling-Schnittstelle von MPI ein und protokollieren während der Ausführung des Programms zusätzliche Informationen, anhand derer sie dann entsprechende Analysen vornehmen können, um damit kritische Stellen im Programmablauf zu identifizieren. Clémençon et al. [9] beschreiben ein Verfahren, um Race Conditions aufzudecken, indem die über MPI verschickten Nachrichten mit Zeitinformationen annotiert werden, aus denen sich dann ein exakter Ablaufgraph gewinnen lässt. Anhand dessen ist es dann nicht nur möglich, automatisiert typische Muster zu erkennen, die auf eine Race Condition hindeuten, sondern ein einmal aufgezeichneter Programmablauf kann damit bei Bedarf in der gleichen Reihenfolge wiederholt werden.

Allerdings beziehen sich diese Werkzeuge nur auf *Anwendungen*, die mit Hilfe von MPI entwickelt wurden, und verlassen sich auf eine funktionierende MPI-Bibliothek. Zum Testen der MPI-Implementierung selbst gibt es derzeit hingegen kaum Hilfsmittel. Es gibt zwar Testprogramme [10], die überprüfen, ob eine Implementierung den MPI-Standard [11] erfüllt, aber da MPI sehr netzwerknah arbeitet und somit seine korrekte Funktionsweise wesentlich von der jeweils vorliegenden Netzwerkkonfiguration abhängt, ist es erforderlich, eine MPI-Implementierung mit unterschiedlichen Netzwerkkonfigurationen zu konfrontieren, um sie gründlich zu testen.

2.2 Virtualisierung

Virtualisierung bezeichnet die Abstraktion der Nutzung von Hardware-Ressourcen. Sie zieht eine Schicht ein, die die physischen Gegebenheiten der Hardware von

ihrer logischen (virtuellen) Benutzung trennt. Insbesondere können mehrere physische Ressourcen zu einer virtuellen zusammengefasst werden, oder umgekehrt eine einzige physische Ressource in viele virtuelle aufgeteilt werden, wobei es für jeden Nutzer so aussieht, als sei er der einzige, der die Ressource nutzt. Ein RAID-Controller (*Redundant Array of Independent Disks*) beispielsweise fasst mehrere Festplatten zusammen, präsentiert sie dem Betriebssystem aber als eine einzige. Umgekehrt erhält bei der gängigen Form der Speicheraddressierung (*Virtual Memory*) jeder Prozess seinen eigenen vollständigen Adressraum, der vom Betriebssystem dann in den physisch vorhandenen Speicher abgebildet wird.

Für die Simulation eines Clusters benötigt man Systemvirtualisierung, also dass auf einem physischen Rechner, dem *Wirt*, mehrere Betriebssysteme als sogenannte *virtuelle Maschinen* (VM) gleichzeitig laufen können. Jede VM erhält dabei ihre eigene Sicht auf die Hardware, die isoliert ist von allen anderen VMs. Für das Betriebssystem innerhalb der VM sieht es so aus, als laufe es ganz normal alleine auf dem Rechner.

Zur Realisierung von virtuellen Maschinen gibt es die folgenden grundlegenden Ansätze [12]:

2.2.1 Emulation

Die größtmögliche Trennung zwischen der VM und dem Wirt erreicht man, indem die Hardware vollständig emuliert wird. Dazu werden alle Komponenten der „Hardware“ der virtuellen Maschine nachgebildet und vom Emulator auf die physische Hardware übersetzt. Durch Emulation ist es möglich, prinzipiell beliebige Software auszuführen, selbst wenn die physische Hardware überhaupt nicht ihren Anforderungen entspricht, beispielsweise weil die Software eine ganz andere Befehlssatzarchitektur benötigt als physisch vorhanden ist. Allerdings ist die Emulation mit erheblichem Aufwand verbunden, der die Leistungsfähigkeit solcher virtueller Maschinen stark einschränkt.

2.2.2 Native Virtualisierung

Bei nativer Virtualisierung wird eine Zwischenschicht eingeführt, die zwischen der Hardware und dem VM-Betriebssystem vermittelt, der sogenannte Hypervisor oder Virtual Machine Monitor. Der Hypervisor übernimmt die Kontrolle der gesamten Hardware, stellt aber gegenüber dem Betriebssystem die gleichen Schnittstellen zur Verfügung, sodass für das Betriebssystem kein Unterschied erkennbar ist, ob es direkt auf der Hardware oder über den Hypervisor läuft. Erstmals eingeführt wurde diese Technik 1966 von IBM mit dem Betriebssystem CP/CMS [13].

Im Gegensatz zur Emulation steht für die virtuellen Maschinen nur die Systemarchitektur zur Verfügung, die tatsächlich physisch vorhanden ist, es kann also in den VMs nur Software verwendet werden, die für die physische Rechnerarchitektur geeignet ist. Diese Einschränkung ermöglicht jedoch, dass ein Großteil der Maschinenbefehle direkt auf der Hardware ausgeführt werden kann, ohne dass der Hypervisor beteiligt werden muss – eine fundamentale Effizienzsteigerung gegenüber der Emulation.

Um die Transparenz gegenüber dem Betriebssystem zu wahren, muss der Hypervisor allerdings in der Lage sein, alle *kritischen* Maschinenbefehle zunächst selbst zu behandeln [14]. Kritische Befehle sind Befehle, die in der Lage sind, die Isolation der VM zu durchbrechen, beispielsweise also Speicherzugriff oder I/O-Befehle. Am einfachsten gelingt ihre Behandlung, wenn diese kritischen Befehle in der Befehlssatzarchitektur privilegiert sind: Da nur der Hypervisor im höchstprivilegierten Modus läuft und die Betriebssysteme der VMs auf einen niedrigeren Level zurückgestuft sind, wird eine Processor-Trap ausgelöst, wenn ein VM-Betriebssystem einen kritischen Befehl ausführen will, und der Hypervisor kann die Trap dann wie gewünscht behandeln.

Insbesondere in der x86-Architektur gibt es aber kritische Befehle, die nicht privilegiert sind und deshalb für native Virtualisierung auf andere Weise behandelt werden müssen [15]. Eine Möglichkeit dazu ist dynamische Rekompilation, dass also der Binärcode der virtualisierten Software zur Laufzeit verändert wird, um die kritischen Befehle auf den Hypervisor umzuleiten. Das ist jedoch mit zusätzlichem Aufwand verbunden, der die Leistungsfähigkeit von nativ virtualisierten x86-Systemen schmälert.

2.2.3 Paravirtualisierung

Wie bei der nativen Virtualisierung kommt bei der Paravirtualisierung ein Hypervisor zum Einsatz, allerdings bietet er dem Betriebssystem keine transparente Schnittstelle zur Hardware, sondern einen eigenen Satz von Befehlen, die das Betriebssystem nutzen muss, wenn es auf die Hardware zugreifen möchte. Damit ein Betriebssystem auf einem solchen Hypervisor laufen kann, muss es also zunächst portiert werden. Allerdings bietet dieser Ansatz gute Performance auch auf Architekturen wie x86, die sich nicht gut für native Virtualisierung eignen, da auf diese Weise kritische Befehle nicht aufwändig extra abgefangen werden müssen.

In ihren neueren x86-Prozessoren haben Intel und AMD jeweils spezielle Unterstützung für Virtualisierung eingearbeitet (*Intel VT* bzw. *AMD-V*), die einen zusätzlichen Modus bereitstellt, der speziell für einen Hypervisor gedacht ist. Dadurch ist es möglich, auch auf x86 vollständige native Virtualisierung zu erreichen und unmodifizierte Gast-Betriebssysteme zu starten. Allerdings ist die Verbreitung solcher

Prozessoren noch eher gering, und zudem ist die Performance von Paravirtualisierung derzeit noch größer als die von nativer x86-Virtualisierung [16].

2.2.4 Betriebssystem-basierte Virtualisierung

Ein ganz anderer Ansatz ist die Betriebssystem- oder Kernel-basierte Virtualisierung, bei der das Betriebssystem ganz normal direkt auf der Hardware läuft und lediglich intern isolierte Zonen zur Verfügung stellt, in denen Programme getrennt voneinander laufen können. Prinzipiell betrachtet handelt es sich dabei um die Fortsetzung von Techniken wie *Virtual Memory*, der in allen modernen Betriebssystemen zu finden ist und jedem Prozess seinen eigenen Adressraum für den Hauptspeicher darbietet, der von dem aller anderen Prozesse vollständig isoliert ist. Kernel-basierte Virtualisierung umfasst nun nicht nur den Hauptspeicher, sondern auch Dateisystem, Netzwerk und Hardwarezugriff.

Da auf dem Wirtsrechner insgesamt nur ein Kernel läuft, ist der Mehraufwand für solche virtuelle Maschinen sehr gering, und man erreicht eine sehr gute Auslastung der Hardware, da kein Mehraufwand etwa für weitere Kernels anfällt [17]. Gleichzeitig kann in den VMs aber nur genau das Betriebssystem verwendet werden, dessen Kernel auf dem Wirt läuft, und allenfalls das Userland kann in gewissen Grenzen variieren.

Für die Realisierung virtueller Maschinen gibt es vielfältige Möglichkeiten, die jeweils eigene Stärken und Schwächen haben. Es gilt nun, eine geeignete Technik auszuwählen und in eine Architektur für eine Testumgebung einzubetten. Ausgehend von den oben vorgestellten grundlegenden Konzepten für automatisiertes Testen soll dadurch der Entwickler beim Testen von verteilten Systemen unterstützt werden.

3 Architektur und Implementierung

Ziel dieser Arbeit ist es, entwicklungsbegleitendes Testen von verteilten Systemen zu ermöglichen. Das dazu entwickelte *Virtual Test Environment* (VTE) erlaubt es dem Entwickler, komplexe Netzwerkkonfigurationen auf seinem Arbeitsplatzrechner zu simulieren. So kann er die entwickelte Software ohne großen Aufwand testen. In diesem Kapitel werden die Maßstäbe, die dem Entwurf von VTE zu Grunde liegen, und die daraus resultierende Architektur vorgestellt. Anschließend wird die Implementierung von VTE im Detail erläutert.

3.1 Anforderungen

Die wesentliche Anforderung an eine Testumgebung für verteilte Systeme ist, dass sie die entwicklungsbegleitende Ausführung von funktionalen Tests in einer möglichst realistischen Umgebung ermöglichen soll (siehe Abschnitt 2.1). Daraus ergeben sich die folgenden Anforderungen:

Eine Testumgebung für Netzwerkkonfigurationen, die regelmäßig zum Einsatz kommen soll, muss es dem Entwickler ermöglichen, diese Konfigurationen auf einfache Weise zu beschreiben – es ist nicht praktikabel, dafür mehrere Stunden im Serverraum zubringen zu müssen und Kabel zu verlegen. Daher muss die Testumgebung eine prägnante Beschreibungssprache für Netzwerkkonfigurationen enthalten. Die Beschreibungssprache soll es zudem erlauben, das simulierte Netzwerk dynamisch zu rekonfigurieren, um in einem Testlauf viele unterschiedliche Netzwerkkonfigurationen zur Verfügung zu stellen.

Damit der Entwickler die Tests regelmäßig laufen lassen kann, während er entwickelt, dürfen sie nicht länger als wenige Minuten in Anspruch nehmen. Deshalb ist es wichtig, dass die Tests in der virtuellen Umgebung schnell ablaufen, und dass der Auf- und Abbau der Testumgebung ebenfalls rasch vonstatten geht.

Um zu erfahren, wie sich die zu testende Software verhält, wenn sie mit komplexen Netzwerkumgebungen konfrontiert wird, muss die Testumgebung in der Lage sein, echte Software unverändert auszuführen, und die Umgebung muss gegenüber der Software wie echt erscheinen. Gleichzeitig muss der Entwickler in der Lage sein, die Testumgebung umfassend zu steuern, und alle Vorgänge darin zu beobachten.

Um einen repräsentativen Eindruck vom Verhalten der zu testenden Software gewinnen zu können, sollten möglichst viele Konfigurationen abbildbar sein, die heutzutage in (Multi-)Cluster-Umgebungen auftreten. Dazu muss die Testumgebung in der Lage sein, relativ komplexe Netzwerkumgebungen zu simulieren.

3.2 Verwandte Arbeiten

Es existieren derzeit nur wenige Werkzeuge, die Netzwerkumgebungen so simulieren, dass Software unverändert in ihr ablaufen kann. Xen-OSCAR [18] ist ein Testwerkzeug für das Cluster-Installationsprogramm OSCAR. Es erzeugt einen virtuellen Cluster aus Xen-VMs, der dann als Teststellung für den Installationsprozess dient. Das virtuelle Netzwerk ist jedoch nicht konfigurierbar, es steht nur eine einzige statische Konfiguration zur Verfügung, da für OSCAR die Topologie nicht relevant ist. Ohne konfigurierbares Netzwerk ist die Testumgebung jedoch beispielsweise für das Testen von MPI-Implementierungen ungeeignet.

Ein weiteres System ist MLN [19], ein Verwaltungswerkzeug für virtuelle Maschinen. Es unterstützt Xen oder UML als Virtualisierungstechnik und erlaubt im Gegensatz zu Xen-OSCAR die deklarative Beschreibung einer virtuellen Netzwerkumgebung. MLN unterstützt dabei auch recht komplexe Netzwerkkonfigurationen. Allerdings bietet seine Architektur keine Möglichkeit, die Konfiguration des Netzwerks zur Laufzeit zu verändern. Da MLN zudem nicht auf kurze Laufzeit optimiert ist, ist es für entwicklungsbegleitendes Testen nicht geeignet.

Obwohl die bestehenden Werkzeuge für ihr jeweiliges Einsatzgebiet gut geeignet sind, können sie die für den vorliegenden Anwendungsfall nötigen Anforderungen nicht ausreichend erfüllen.

3.3 Architektur

Damit VTE in der Lage ist, die in Abschnitt 3.1 aufgestellten Kriterien zu erfüllen, wurde eine Architektur entworfen, die die Umsetzung dieser Kriterien unterstützt.

Der grundlegende Aufbau von VTE ist in Abbildung 3.1 auf der nächsten Seite dargestellt. VTE simuliert eine (Multi-)Cluster-Umgebung mit virtuellen Maschinen, die mit einem virtuellen Netzwerk verbunden werden. Der Entwickler formuliert die Konfiguration dieser Simulationsumgebung mit Hilfe einer Konfigurationssprache. Die so spezifizierte Konfiguration wird von einem Steuerprogramm umgesetzt, indem auf dem Wirtsrechner VMs erzeugt werden und das virtuelle Netzwerk eingerichtet wird.

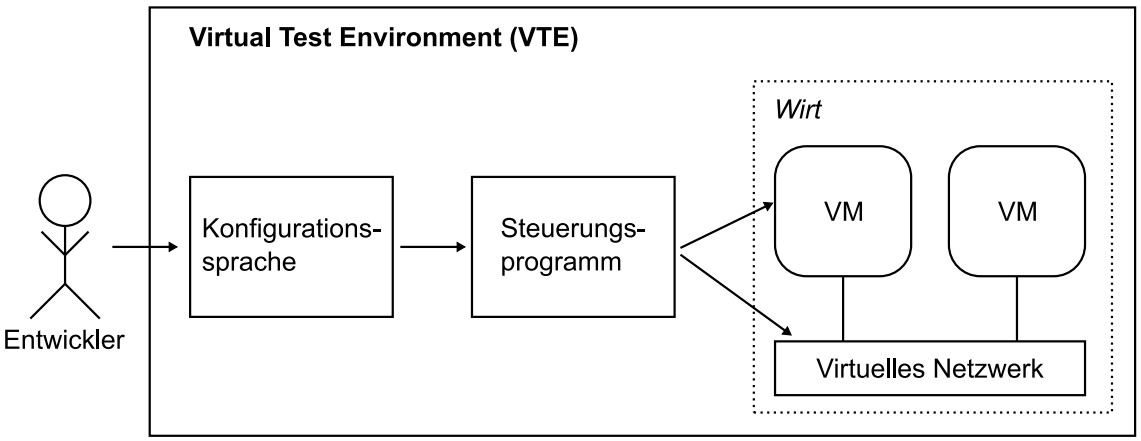


Abbildung 3.1: Die Architektur des *Virtual Test Environment* (VTE).

Als Virtualisierungstechnik kommt OpenVZ zum Einsatz, diese Wahl wird in Abschnitt 3.7.1 erläutert. Das virtuelle Netzwerk wird auf dem Wirtsrechner erstellt, indem die virtuellen Netzwerkschnittstellen der VMs mit Hilfe von Linux-Netzwerk-Werkzeugen entsprechend verbunden werden. Vor der ersten Simulation durchläuft VTE einmalig eine Setup-Phase, in der das Dateisystem für die virtuellen Maschinen angelegt wird.

VTE ist in Ruby implementiert, einer Programmiersprache, die es erlaubt, Sachverhalte auf sehr hohem, abstrakten Niveau auszudrücken. Daher ist kein separates Steuerungsprogramm nötig, sondern die Konfigurationssprache ist direkt ausführbarer Ruby-Code, der die Konfigurationsinformation nicht nur spezifiziert, sondern auch umsetzt, d. h. virtuelle Maschinen erzeugt und das virtuelle Netzwerk anlegt.

Die Umsetzung der Konfiguration erfolgt sofort durch die Ausführung des Ruby-Programms, es ist keine separate Übersetzungsphase nötig. Die Konfigurationssprache erzeugt Ruby-Objekte, die einerseits dem Entwickler zur Interaktion mit der Simulationsumgebung zur Verfügung stehen, und andererseits den Wirtsrechner über Shell-Kommandos steuern, um die Simulationsumgebung zu konfigurieren. Der Entwickler kann die Konfiguration des simulierten Clusters daher sogar interaktiv über die Ruby-Shell manipulieren.

Durch diese Interaktivität der Konfigurationssprache wird eine Änderung der Konfiguration zur Laufzeit prinzipiell möglich. Natürlich müssen die Konfigurationselemente entsprechend implementiert werden, um tatsächlich eine rekonfigurierbare Testumgebung zu erhalten. Aber eine entsprechende Umsetzung ist nur realisierbar, weil die Architektur dies grundsätzlich ermöglicht. Dies gilt für alle in Abschnitt 3.1 aufgeführten Anforderungen: Die Architektur von VTE ist darauf ausgelegt, sie grundsätzlich zu ermöglichen, und die Implementierung kann sie daran anschließend geeignet umsetzen.

3.4 Konfigurationssprache

Um dem Entwickler eine prägnante und ausdrucksstarke Möglichkeit zur Spezifikation von Tests zu geben, wurde für VTE eine *Domain-Specific Language* zur Beschreibung von Netzwerkkonfigurationen entwickelt. Die zur Verfügung stehenden Sprachelemente werden im Folgenden zunächst auf abstrakter Ebene vorgestellt und anschließend anhand von Beispielen erläutert.

3.4.1 Konzepte

Die Konfigurationssprache von VTE beschreibt eine (Multi-)Cluster-Umgebung, die aus Komponenten wie *Rechenknoten*, *Netzwerkverbindung*, *Switch* und *Router* zusammengesetzt ist.

Das zentrale Element der Konfigurationssprache ist die *VirtualMachine*, sie repräsentiert einen Cluster-Knoten, auf dem Software ausgeführt werden kann. Eine VM kann Verzeichnisse des Wirts einblenden, um die zu testende Software innerhalb des simulierten Clusters sichtbar zu machen, und sie kann Shell-Kommandos ausführen und ihre Ergebnisse zurückliefern. Jede VM kann beliebig viele Netzwerkkarten erhalten, die dann das Simulationsnetzwerk bilden. Alle VMs eines physischen Rechners werden in einem *Cluster* zusammengefasst, der das Starten und Stoppen von virtuellen Maschinen ermöglicht.

Ein *Switch* ist der gleichnamigen Netzwerkhardware nachempfunden, er verbindet Rechner auf Ethernet-Ebene zu einem lokalen Netzwerk. Er hat jedoch Funktionen, die über die seines physischen Pendants hinausgehen, insbesondere repräsentiert er ein IP-Subnetz (IPv4 oder IPv6), d. h. wenn VMs mit einem Switch verbunden werden, wird ihnen automatisch eine IP-Adresse aus dem Subnetz des Switch zugewiesen.

Mehrere Switches können mit einem *Router* auf IP-Ebene verbunden werden. Für jeden Switch kann dabei festgelegt werden, ob seine Pakete mit NAT bearbeitet werden sollen. Auf diese Weise können unter anderem auch Subnetze mit privaten IP-Adressen simuliert werden.

Ein Beispiel für eine einfache Netzwerkkonfiguration ist die Angabe

```
cluster = Cluster.new(16)
switch = Switch.new("192.168.5.0/24").connect(cluster.vm)
```

Mit diesen zwei Zeilen wird ein virtueller Cluster mit 16 Knoten konfiguriert, die durch einen Switch verbunden sind. Die Rechner verwenden IP-Adressen aus dem Subnetz 192.168.5.0/24 haben.

Ein Grundzug der Konfigurationssprache ist, sinnvolle Standardwerte zu verwenden, wenn keine expliziten Angaben getroffen werden, dabei aber Ausnahmeregelungen zu ermöglichen. Wenn im obigen Beispiel das Subnetz nicht mit angegeben wäre, also nur Switch.new(:ipv4), würde automatisch ein Subnetz generiert. Dadurch muss der Entwickler derartige Details nicht explizit angeben, wenn sie nicht relevant sind.

3.4.2 Sprachelemente

Die folgende Tabelle stellt die grundlegenden Elemente der Konfigurationssprache mit ihren wichtigsten Funktionen im Überblick dar.

Klasse/Methode	**Beschreibung**
Cluster	Verwaltet eine Menge von VMs.
— new(count, host="localhost")	Erzeugt einen Cluster mit count VMs, optional auf einem anderen Wirtsrechner.
— start(count)	Startet so viele VMs, dass count VMs in diesem Cluster vorhanden sind.
— stop(count)	Stoppt count VMs.
— mount(directory, mountpoint=nil)	Mountet das Wirtsverzeichnis directory unter dem gleichen Pfad, oder dem optionalen mountpoint, in allen VMs in diesem Cluster.
VirtualMachine	Repräsentiert einen Clusterknoten.
— hostname	Der Hostname der VM.
— ipv4[n]	Die IPv4-Adresse der n-ten Netzwerkkarte.
— ipv6[n]	Die IPv6-Adresse der n-ten Netzwerkkarte.
— cmd(command)	Führt das angegebene Shell-Kommando in der VM aus und gibt ein Command-Objekt zurück, mit dem das Ergebnis abgefragt werden kann.
— cmd_raise(command)	Führt das angegebene Shell-Kommando aus und wirft eine Exception, wenn die Ausführung nicht erfolgreich war.
Switch	Verbindet VMs auf Ethernet-Ebene.
— new(network, host="localhost")	Erzeugt einen Switch, der das angegebene Subnetz repräsentiert (z. B. "192.168.0.0/24" oder :ipv6), optional auf einem anderen Wirtsrechner.
— connect(vms, interface=0)	Verbindet die angegebenen VMs mit diesem Switch, optional über die Netzwerkkarte mit der Nummer interface.

(Fortsetzung Sprachelemente)

Klasse/Methode	Beschreibung
— connect_host	Verbindet den Wirtsrechner mit dem Switch (wird z. B. von Router verwendet).
— destroy	Entfernt den Switch vollständig (siehe Abschnitt 3.6.4).
Router	Verbindet Switches auf IP-Ebene.
— new(host="localhost")	Erzeugt einen Router, optional auf einem anderen Wirtsrechner.
— connect(switches, nat=false)	Verbindet die angegebenen Switches via Routing, optional mit NAT.
— destroy	Entfernt den Router vollständig (siehe Abschnitt 3.6.4).
Command	Gibt Zugriff auf die Ergebnisse eines Shell-Kommandos.
— stdout	Der Inhalt des Standard-Ausgabe-Stroms.
— stderr	Der Inhalt des Standard-Fehler-Stroms.
— exit_code	Der Rückgabewert des Kommandos.

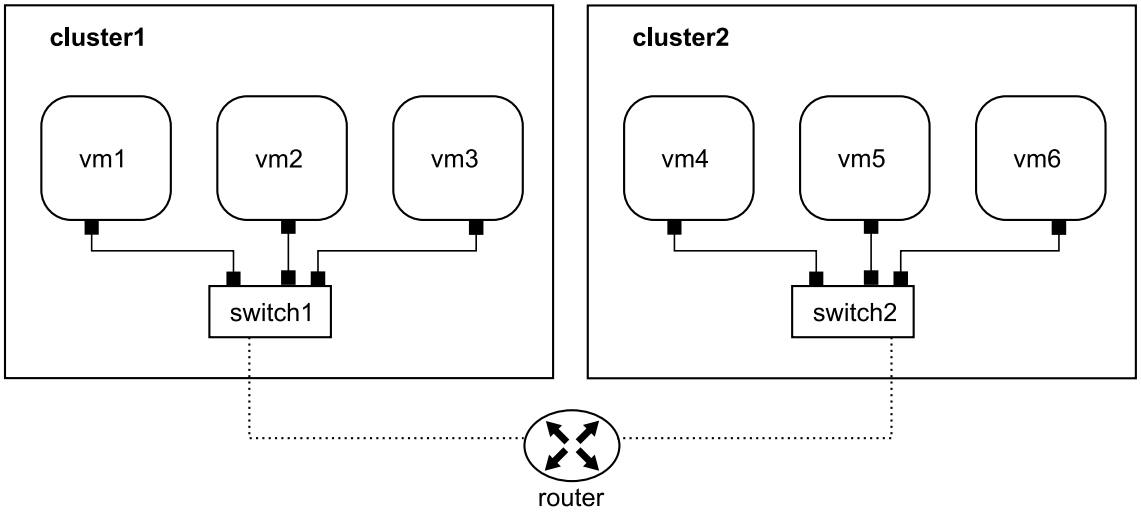
Abbildung 3.2 auf der nächsten Seite zeigt als umfangreicheres Konfigurationsbeispiel eine Multicluster-Umgebung, die aus zwei Clustern mit jeweils drei Knoten besteht, die über einen Router verbunden sind. Auch komplexere Strukturen lassen sich mit VTE prägnant beschreiben, indem die grundlegenden Elemente entsprechend kombiniert werden.

3.5 Virtuelles Netzwerk

VTE bietet verschiedene Netzwerkkomponenten an, aus denen das Simulationsnetzwerk aufgebaut werden kann. Durch ihre Kombination soll VTE in der Lage sein, auch komplexere Netzwerkumgebungen zu modellieren. VTE beschränkt sich dabei auf TCP/IP-Netzwerke, um die Komplexität der Konfigurationssprache möglichst niedrig zu halten, sodass eine einfache Benutzung gewährleistet ist. In den folgenden Abschnitten werden die zur Verfügung stehenden Komponenten sowie ihre Implementierung im Einzelnen vorgestellt. Beispiele für ihren Einsatz finden sich in Abschnitt 4.1.

3.5.1 Switch

Eine *Bridge* verbindet zwei Netzwerksegmente auf Ethernet-Ebene. Bridges mit mehr als zwei Anschlüssen werden üblicherweise als *Switch* bezeichnet. Ein Switch




```
cluster1 = Cluster.new(3)
cluster2 = Cluster.new(3)
switch1 = Switch.new(:ipv4).connect(cluster1.vm)
switch2 = Switch.new(:ipv4).connect(cluster2.vm)
router = Router.new.connect(switch1).connect(switch2)
```

Abbildung 3.2: Beschreibung einer Multicluster-Umgebung mit VTE.

untersucht alle eingehenden Pakete, um Absender und Empfänger festzustellen. Ist ihr bekannt, in welchem Teilnetz sich der Empfänger befindet, schickt sie das Paket nur auf dem zugehörigen Anschluss weiter, andernfalls als Broadcast auf allen Anschlüssen. Um diese Zuordnung herzustellen, unterhält ein Switch ein Verzeichnis darüber, welche MAC-Adresse an welchem seiner Anschlüsse angeschlossen ist [20].

Unter Linux können mit den bridge−utils [21] virtuelle Bridges erzeugt werden, mit denen andere Netzwerkinterfaces auf Ethernet-Ebene zusammengeschaltet werden können – da mehr als zwei Interfaces auf eine Bridge geschaltet werden können, wäre die Bezeichnung virtueller Switch ebenso angebracht. Die virtuelle Bridge verteilt die Pakete in gleicher Weise auf die zusammengeschalteten Interfaces wie eine physische Bridge, lediglich die Kabel zwischen Netzwerkkarte und Bridge entfallen.

Mit Hilfe einer virtuellen Bridge kann beispielsweise ein Rechner mit zwei Netzwerkkarten die Funktion einer physischen Bridge übernehmen und zwei Netzwerke verbinden. Indem die zwei Netzwerkinterfaces auf einer virtuellen Bridge zusammengefasst werden, werden nun alle Pakete, die auf einer Netzwerkkarte eingehen, gegebenenfalls auch auf der anderen Netzwerkkarte wieder ausgegeben. Die überbrückten Netzwerkinterfaces verlieren dabei ihre normale Funktion, stehen also dem Rechner nicht mehr als Verbindung zum Netzwerk zur Verfügung. Um den Rechner selbst in das zusammengefasste Netz zu integrieren, kann eine

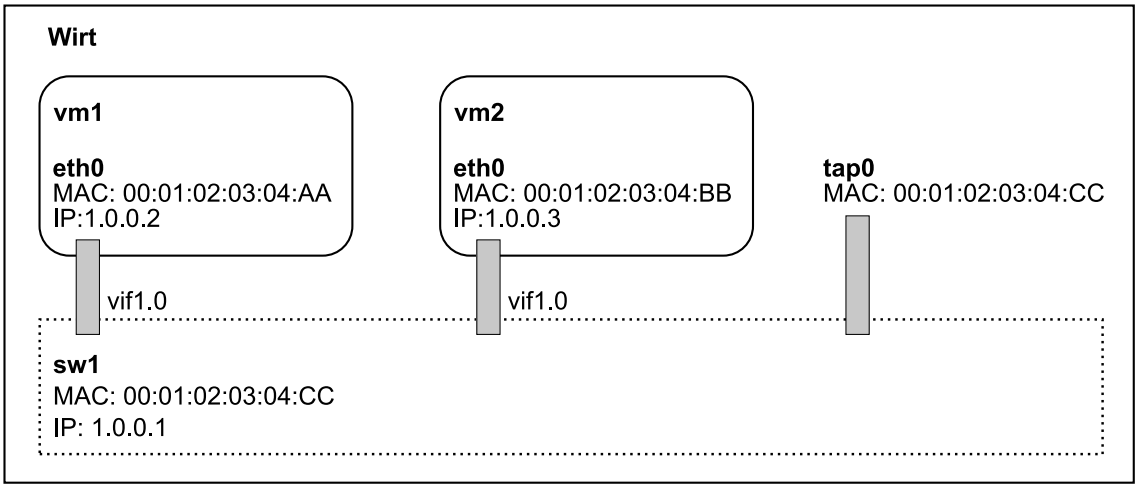


Abbildung 3.3: Virtuelle Bridge.

virtuelle Bridge aber eine Doppelrolle einnehmen: Sie dient nicht nur dazu, Interfaces zusammenzuschalten, sondern sie wird selbst ebenfalls durch ein eigenes Interface repräsentiert, das eigenständig konfiguriert werden und z. B. IP-Adressen erhalten kann.

Um eine Switch-Verbindung zwischen VMs zu simulieren, erzeugt VTE eine Bridge und schaltet darauf, wie in Abbildung 3.3 dargestellt, die wirtsseitigen Enden (vif$y$.x, siehe Abschnitt 3.6.2) der Netzwerkkarten der VMs zusammen. Damit der Wirt an dem virtuellen Netzwerk teilnehmen kann (dies ist insbesondere für Routing erforderlich, siehe den folgenden Abschnitt), kann die Doppelrolle der virtuellen Bridge genutzt werden und ihr eine eigene IP-Adresse zugewiesen werden. Eine Bridge hat jedoch keine eigene MAC-Adresse, sondern kann nur solche von Interfaces annehmen, die auf ihr zusammengeschaltet sind. Die hier zusammengeschalteten Interfaces gehören aber jeweils zu einer VM. Damit der Wirt also an dem Netz teilnehmen kann, muss er eine von allen vorhandenen Adressen verschiedene MAC-Adresse erhalten. Um das zu ermöglichen, erzeugt VTE ein weiteres virtuelles Netzwerkinterface (tap0 in der Abbildung) und schaltet es auf die Bridge. Dieses Interface erfüllt keine weitere Funktion, außer dass auf ihm eine MAC-Adresse konfiguriert wird, die im Anschluss auf die Bridge übertragen werden kann.

3.5.2 Router

Ein *Router* verbindet mehrere IP-Subnetze. Im einfachsten Fall geschieht das dadurch, dass der Router mehrere Netzwerkkarten hat, die mit den verschiedenen Subnetzen verbunden sind. Er schickt eingehende Pakete, die für ein gewisses

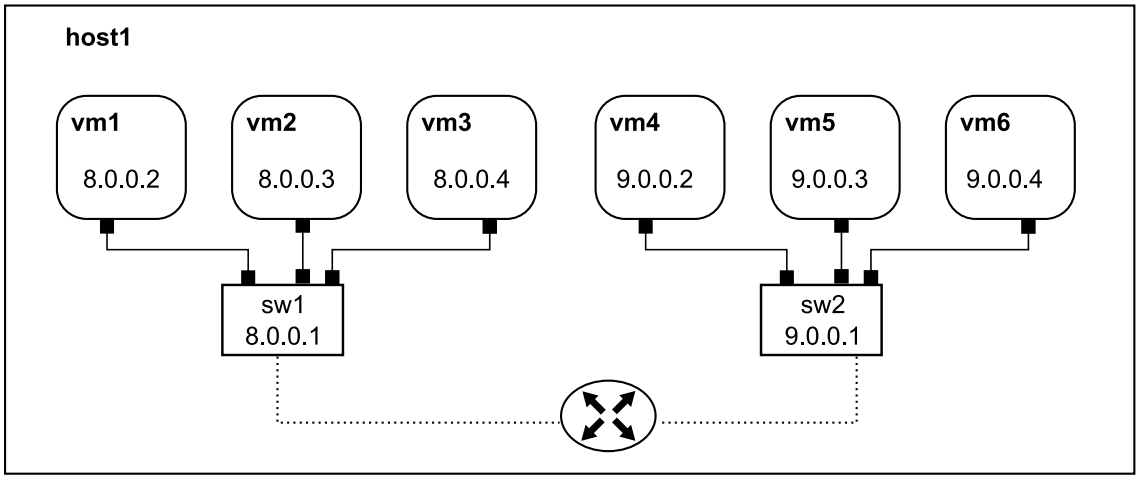


Abbildung 3.4: Routing zwischen zwei Subnetzen auf demselben Wirt.

Subnetz bestimmt sind, über die jeweils zugehörige Netzwerkkarte weiter. Im Allgemeinen muss ein Router jedoch lediglich wissen, wohin ein Paket als *nächstes* geschickt werden muss, um zu seinem Ziel zu gelangen (siehe RFC1812 [22]).

Bei VTE übernimmt der Wirt die Rolle des Routers. Wie im vorigen Abschnitt geschildert, können die Bridge-Interfaces, mit denen VMs zu einem simulierten lokalen Netz verbunden werden, zusätzlich im Wirt als Netzwerkkarte konfiguriert werden und eine IP-Adresse aus dem jeweiligen Subnetz erhalten. Dabei werden auf dem Wirt Routen zu diesen Netzen über das zugehörige Bridge-Interface angelegt. Somit können die VMs, die mit der einen Bridge verbunden sind, den Wirt als Router nutzen, um mit den VMs an der anderen Bridge zu kommunizieren. Dazu werden in den VMs Routen zu den jeweils anderen Subnetzen angelegt, die über den Wirt führen.

Damit der Wirt aber überhaupt als Router fungieren kann, muss im Kernel *IP forwarding* (das Weiterleiten von Paketen zwischen Netzwerkschnittstellen) aktiviert werden. Dazu dienen die *sysctl*-Parameter /proc/sys/net/ipv4/conf/all/forwarding und /proc/sys/net/ipv4/conf/<interface>/forwarding (desgleichen für IPv6). Allerdings verhalten sich diese Parameter bei IPv4 und IPv6 unterschiedlich: Während bei IPv4 sowohl der globale als auch der Interface-spezifische Parameter aktiviert sein müssen, damit *IP forwarding* funktioniert, hat bei IPv6 der Interface-spezifische Parameter darauf keine Auswirkung. Jedoch führt Aktivieren von IPv6-Forwarding auf einem Interface dazu, dass auf diesem Interface die Stateless Autoconfiguration deaktiviert wird, was nicht immer dem gewünschten Verhalten entspricht. Bei beiden IP-Versionen fungiert der globale Parameter all als übergreifender Umschalter, sodass bei einer Änderung seines Wertes auch alle Interface-Parameter diesen Wert erhalten.

In Abbildung 3.4 ist ein Beispiel-Netzwerk dargestellt, in dem es zwei Subnetze

mit jeweils drei VMs gibt. Auf den beiden Bridges sind IP-Adressen konfiguriert, d. h. im Wirt existieren die Routen „9.0.0.0/8 über sw1“ und „8.0.0.0/8 über sw2“. Nun kann auf den VMs 1–3 die Route „9.0.0.0/8 über 8.0.0.1“ und auf den VMs 4–6 die Route „8.0.0.0/8 über 9.0.0.1“ gesetzt werden, um die beiden Subnetze zu verbinden.

3.5.3 NAT

Um ein Subnetz mit privaten IPv4-Adressen (siehe RFC1918 [23]) mit einem öffentlichen Subnetz zu verbinden, kommt üblicherweise *Network Address Translation* (NAT, RFC3022 [24]) zum Einsatz: Der Router modifiziert ausgehende Pakete so, dass sie seine eigene öffentliche Adresse als Absender enthalten, da private Adressen jenseits des Routers keine Gültigkeit besitzen. Antworten gehen daher zurück an den Router, der sie dann an den ursprünglichen Absender weiterleitet. Dazu muss er natürlich eingehende Antworten dem ursprünglichen Absender wieder zuordnen können. Dies wird dadurch erreicht, dass der Router den Quellport der ausgehenden Pakete modifiziert, und intern die modifizierten Quellports den ursprünglichen Absendern zuordnet.

VTE unterstützt die Simulation von NAT und verwendet dazu auf dem Wirt den Linux-Paketfilter iptables [25], der mit dem MASQUERADE-Modul Unterstützung für NAT bietet. NAT findet auf dem ausgehenden Interface statt. Damit ein Subnetz NAT erhalten kann, muss also auf den Bridges aller anderen Subnetze eine entsprechende iptables-Regel konfiguriert werden. Eine solche Einstellung wäre allerdings zu allgemein, denn dadurch würden ja *alle* Pakete auf der ausgehenden Bridge mit NAT versehen, und nicht nur die, die von dem privaten Subnetz stammen, das NAT erhalten soll.

Leider kann iptables in der Regelphase, in der NAT stattfindet, nicht mehr nachvollziehen, von welchem Interface das Paket ursprünglich stammt. Deshalb wird die iptables-Regel zweigeteilt: Auf der Bridge des Subnetzes, das NAT erhalten soll, werden die Pakete als NAT-bedürftig markiert, und auf allen ausgehenden Bridges wird NAT nur für derart markierte Pakete aktiviert. iptables bietet zum Markieren von Paketen das CONNMARK-Modul, das Pakete mit einem numerischen Wert versehen und später zur Weiterverarbeitung anhand dieses Wertes auswählen kann.

3.5.4 Mehrere Wirtsrechner

Wenn mehrere Wirtsrechner an der VTE-Simulation beteiligt sind (siehe dazu Abschnitt 3.7.5), sollen die VMs auch dann via Routing verbunden werden können, wenn sie auf verschiedenen Wirten laufen. Dabei muss natürlich das physische Netz benutzt werden, um einen anderen Wirt zu erreichen. Das simulierte Netz

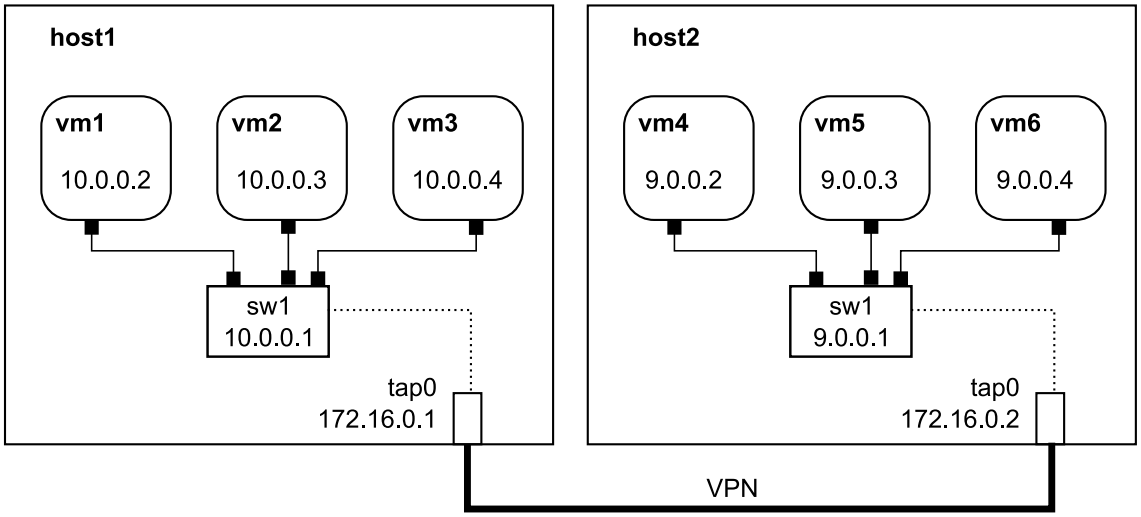


Abbildung 3.5: Routing über mehrere Wirtsrechner.

soll aber vom physischen möglichst isoliert werden, da das physische Netz nicht beeinträchtigt werden soll. Außerdem soll das simulierte Netz unabhängig davon, ob beispielsweise auf dem physischen Netz IPv6 verfügbar ist oder nicht, sowohl IPv4 als auch IPv6 unterstützen. Dazu wird zwischen den Wirten ein VPN-Tunnel aufgebaut, über den die simulierten Netze dann geroutet werden können.

VTE verwendet tinc [26], um ein VPN zu erzeugen. Auf jedem Wirt wird ein tinc-Daemon gestartet, der sich mit den Daemons auf den anderen Wirten verbindet und einen VPN-Tunnel aufbaut. Das Tunnelende stellt tinc als virtuelles Netzwerkinterface tap$x$ zur Verfügung. Das VPN wird von VTE vollkommen automatisch konfiguriert. Dazu werden die Konfigurationsdateien für tinc programmatisch erzeugt, und die Daemons werden ebenfalls automatisch gestartet und beendet.

Jeder Wirt erhält eine IPv4- und IPv6-Adresse aus einem gesonderten Adressbereich zugeordnet. Während die virtuellen Switches für die VMs standardmäßig die Subnetze 1.0.0.0/8, 2.0.0.0/8 bzw. 2001:638:906:aa00/64, :aa01 etc. verwenden, kommen für das VPN 172.16.0.0/24 und 2001:638:906:bbbb/64 zum Einsatz.

Um zwei simulierte Cluster auf verschiedenen Wirten per Routing zu verbinden, werden, wie in Abschnitt 3.5.2 beschrieben, auf allen VMs zunächst Routen in das jeweils andere Netz gesetzt, die über den Wirt führen. Da das andere Netz sich nun aber nicht auf dem Wirt selbst befindet, muss zusätzlich auf dem Wirt eine Route über den anderen Wirt gesetzt werden, um zum gewünschten Netz zu gelangen. Diese Route wird dabei über das VPN geführt, nicht über das physische Netz.

Abbildung 3.5 zeigt eine Beispielkonfiguration mit zwei Clustern auf unterschiedlichen Wirten. Die VMs des ersten Clusters nutzen private IPv4-Adressen und sind über NAT mit der Außenwelt verbunden. Diese Konfiguration lässt sich mit der folgenden Beschreibung erzeugen:

```
cluster1 = Cluster.new(3, "host1")
cluster2 = Cluster.new(3, "host2")
switch1 = Switch.new("10.0.0.0/8", cluster1).connect(cluster1.vm)
switch2 = Switch.new("9.0.0.0/8", cluster2).connect(cluster2.vm)
router = Router.new(cluster1).connect(switch1, nat=true).connect(switch2)
```

In diesem Beispiel würde also auf host1 eine Route „9.0.0.0/8 über 172.16.0.2“ gesetzt, um Pakete von den VMs 1–3 an die VMs 4–6 korrekt zustellen zu können. NAT muss im wirtsübergreifenden Fall allerdings gesondert behandelt werden. Im Beispiel werden die Pakete auf host1 auf dem Interface tap0 mit NAT modifiziert und über das VPN an die VMs auf host2 geschickt. Als Absender ist nun korrekterweise 172.16.0.1 angegeben, diese Adresse ist jedoch den VMs nicht bekannt, weil das VPN ja ausschließlich auf den Wirten existiert. Damit NAT hier dennoch funktioniert, werden die Rückrouten über das VPN zusätzlich in den VMs eingetragen. Im Beispiel erhalten also die VMs 4–6 die Route „172.16.0.0./24 über 9.0.0.1“.

Die vorgestellten Netzwerkkomponenten können vom Entwickler mit Hilfe der Konfigurationssprache relativ frei kombiniert werden, um das Simulationsnetzwerk zu beschreiben. VTE ist dadurch in der Lage, komplexe Netzwerkumgebungen abzubilden, um eine repräsentative Testumgebung bereitstellen zu können.

3.6 Kontrolle der Testumgebung

Gegenüber der zu testenden Software erscheint VTE wie ein echter Cluster, mit richtigen Rechnern und echter Netzwerkhardware, es ist von innen her damit so gut wie unsichtbar. Gleichzeitig hat der Entwickler aber die volle Kontrolle über die Testumgebung, er kann ihre Konfiguration verändern, und von außen detaillierte Beobachtungen und Analysen vornehmen.

3.6.1 Shell-Kommandos

Die Testumgebung kann durch die Ausführung von Shell-Kommandos innerhalb der virtuellen Maschinen gesteuert werden, ihre Ergebnisse (Ausgaben auf stdout und stderr sowie der exit code) können im Anschluss abgefragt werden. Auch VTE selbst verwendet dies zur Konfiguration der VMs.

Die Laufzeiten aller VMs eines Clusters können beispielsweise mit dem folgenden Befehl ausgegeben werden:

```
cluster = Cluster.new(5)
cluster.vm.each { |vm| puts vm.cmd("uptime").stdout }
```

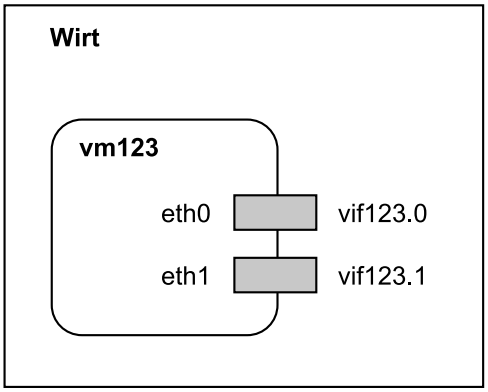


Abbildung 3.6: Virtuelle Netzwerkkarten.

Die Ausführung von Shell-Kommandos ist mit Hilfe des vzctl exec-Mechanismus von OpenVZ implementiert, der es ermöglicht, Kommandos direkt im Kontext einer virtuellen Maschine auszuführen. Alternativ könnten die Kommandos auch mit ssh-Aufrufen in die VMs eingebracht werden, aber dazu wäre ein virtuelles Service-Netzwerk erforderlich. Durch die netzwerklose Kommunikation ist jedoch keinerlei Veränderung der virtuellen Netzwerkkonfiguration nötig, und die Simulationsumgebung kann genau so konfiguriert werden, wie es von den Tests angefordert wird – möglicherweise sogar ganz ohne Netzwerk. VTE versieht zudem alle Kommandos mit einer konfigurierbaren maximalen Laufzeit, sodass etwa ein einzelner hängengebliebener Test nicht den ganzen Testablauf blockiert.

3.6.2 Virtuelle Netzwerkkarten

Ein wesentlicher Bestandteil der Testumgebung ist das virtuelle Netzwerk. OpenVZ bietet zwei verschiedene virtuelle Netzwerktreiber an, venet, der nur auf IP-Ebene arbeitet, und veth, der auf Ethernet-Ebene arbeitet. VTE benötigt Netzwerkverbindungen auf Ethernet-Ebene, deshalb wird veth eingesetzt.

Jede veth-Schnittstelle hat zwei Enden, eines innerhalb der VM und eines auf dem Wirt (siehe Abbildung 3.6). Somit hat VTE die Möglichkeit, die wirtsseitigen Enden vielfältig zu einem virtuellen Netzwerk zusammenzuschalten (siehe Abschnitt 3.5). Außerdem kann der Entwickler Netzwerk-Sniffer oder andere Werkzeuge auf dem Wirtsrechner ausführen, um das virtuelle Netzwerk zu untersuchen.

Die Namen der Schnittstelle sind frei wählbar, VTE benennt das Ende innerhalb der VM eth$x$ und das auf dem Wirt vif$y.x$, wobei x die Nummer der Schnittstelle und y die numerische ID der VM ist. Jede VM kann beliebig viele Netzwerkschnittstellen erhalten. Die MAC-Adressen für die virtuellen Netzwerkkarten werden von VTE unter Einbeziehung der User- und Prozess-ID zufällig generiert. Dadurch können

Parameter	Bedeutung	Voreinstellung
memory_min	Minimaler Arbeitsspeicher	6 MB
memory_max	Maximaler Arbeitsspeicher	32 MB
numproc	Anzahl Prozesse	100
tcp_sockets	Anzahl TCP-Sockets	768
tcp_buffer	Gesamtgröße der TCP-Puffer	10 MB
other_sockets	Anzahl anderer Netzwerk-Sockets	128
other_buffer	Gesamtgröße anderer Netzwerk-Puffer	129 KB

Tabelle 3.1: Konfigurationsparameter für Ressourcenbeschränkungen.

(in gewissem Rahmen) mehrere VTE-Simulationen gleichzeitig auf einem Wirtsrechner ablaufen.

3.6.3 Ressourcenbeschränkungen

OpenVZ bietet sehr feingranulare Möglichkeiten, den Ressourcenverbrauch von VMs zu beschränken. Man kann den verfügbaren Arbeitsspeicher, die Anzahl der Prozesse, Dateihandles und Netzwerksockets, die Größe der Netzwerkpuffer und noch einiges mehr gezielt beschränken. Allerdings sind die Einstellgrößen nicht unabhängig voneinander. Beispielsweise muss der Kernelspeicher groß genug eingestellt werden, um genug Prozessdeskriptoren aufnehmen zu können, sodass die eingestellte Anzahl an Prozessen erreicht werden kann. Die OpenVZ-Dokumentation [27] beschreibt einige Faustregeln, die dabei beachtet werden sollten.

VTE abstrahiert von den detaillierten Einstellgrößen und bietet die in Tabelle 3.1 aufgeführten Konfigurationsparameter zur Anpassung an, falls die eingestellten Standardwerte unzureichend sind. Die übrigen Ressourcenbeschränkungen werden von VTE dann so eingestellt, dass sie mit diesen Angaben zusammenpassen. Beispielsweise kann mit

```
VirtualMachine.config.numproc = 200
cluster = Cluster.new(3)
```

die Maximalzahl erlaubter Prozesse pro VM auf 200 erhöht werden.

Bei allen Angaben bis auf den minimalen Arbeitsspeicher handelt es sich um Maximalwerte, d. h. solange die VMs die Ressourcen nicht anfordern, werden sie auch nicht belegt. Dadurch ist eine sehr effiziente Auslastung des Wirts möglich (siehe dazu Abschnitt 4.2.2).

3.6.4 Isolation

VTE bietet eine vollständig virtuelle Umgebung, die nicht von einem physisch vorhandenen Netzwerk oder einer bestimmten Systemkonfiguration abhängt, nicht einmal davon, mehr als einen Rechner zur Verfügung zu haben. Die Testumgebung ist komplett eigenständig, VTE erzeugt sie aus virtuellen Maschinen und virtuellen Netzwerkkomponenten, auf denen die zu testende Software ausgeführt werden kann.

Im Laufe einer Simulation nimmt VTE allerdings gewisse Veränderungen an der Konfiguration des Wirtsrechners vor, beispielsweise durch Erzeugung von virtuellen Netzwerkkomponenten (siehe Abschnitt 3.5) und das Starten von virtuellen Maschinen. Diese Änderungen betreffen zwar jeweils nur die simulierte Umgebung, insbesondere an der schon vorhandenen Netzwerkkonfiguration wird nichts verändert. Nach Beendigung der Simulation sind diese Konfigurationseinstellungen, temporären Dateien etc. jedoch nicht mehr von Belang und sollten entfernt werden, um den Wirtsrechner in ordentlichem Zustand zurückzulassen. Daher definieren alle Objekte in VTE, die mit dem Wirtsrechner interagieren, *Finalizer*-Methoden, die aufgerufen werden, wenn das Objekt von der Ruby-Laufzeitumgebung gelöscht wird (weil es nicht mehr referenziert wird, oder auch weil der gesamte Prozess beendet wird), und dann entsprechend hinter sich aufräumen.

Für das Starten und Stoppen von VMs (vzctl) sowie für die Konfiguration des Simulationsnetzwerks auf dem Wirt (ip, brctl, iptables, tunctl) werden root-Rechte benötigt. VTE benutzt deshalb für alle derartigen Aufrufe sudo. Aufgaben, die Dateioperationen erfordern, wie etwa das Löschen der Private Area aller VMs nach Abschluss der Simulation, werden von Hilfsprogrammen ausgeführt, die jeweils nur diese eine Aufgabe ausführen und gegen Missbrauch abgesichert sind: Die Private Areas werden standardmäßig in einem temporären Verzeichnis der Form /var/tmp/vte−USERNAME abgelegt, und die Hilfsprogramme führen ihre Operationen nur unterhalb des Verzeichnisses desjenigen Benutzers aus, der sudo aufgerufen hat (dies wird von sudo in der Umgebungsvariable SUDO_USER gespeichert).

Durch die weitreichenden Kontrollmöglichkeiten der virtuellen Umgebung erhält der Entwickler die Möglichkeit, die Testumgebung wie benötigt einzustellen sowie das Verhalten der zu testenden Software genau zu beobachten. Gleichzeitig erscheinen die virtuellen Maschinen der Software gegenüber als so real, dass sie ohne Anpassung direkt in der Testumgebung ausgeführt werden kann.

3.7 Effizienzsteigerung

VTE soll es dem Entwickler ermöglichen, regelmäßig während der Entwicklung Rückmeldungen über das Verhalten der entwickelten Software zu erhalten. Dazu ist es erforderlich, dass die Tests schnell genug ablaufen. In den folgenden Abschnitten werden Mittel zur Steigerung der Leistungsfähigkeit von VTE vorgestellt.

3.7.1 Virtualisierungstechnik

Der wichtigste Faktor für den Zeitbedarf der virtuellen Testumgebung ist die Wahl der Virtualisierungstechnik. VTE benötigt eine möglichst leichtgewichtige Technik, um die Tests ohne großen Zeitaufwand ausführen zu können. Betrachtet man die in Abschnitt 2.2 vorgestellten Virtualisierungstechniken, so ergibt sich ein recht deutliches Bild:

Native Virtualisierung im eigentlichen Sinne ist unter der x86-Architektur nicht möglich. Am nächsten kämen dem die Produkte von VMWare Inc. [28], aber diese sind kommerziell und zudem für das akademische Umfeld ungeeignet, da die Lizenzbestimmungen die Veröffentlichung von Forschungsergebnissen stark einschränken [29].

Paravirtualisierung und Emulation (z. B. mit Xen [30] oder QEMU [31]) wären prinzipiell denkbar, bei Emulation ist jedoch fraglich, ob die Leistungsfähigkeit hoch genug ist, da von massiven Leistungseinbußen auszugehen ist. Beide Ansätze weisen eine starke Entkopplung von Gast und Wirt auf, die es ermöglicht, auch sehr fremde Betriebssysteme als Gast einzusetzen. Dies behindert aber naturgemäß eine einfache Integration von Gast und Wirt. Hinzu kommt, dass pro VM ein vollständiges Betriebssystem gestartet werden muss, was mehr Zeit in Anspruch nimmt, als nur ein neues Userland anzulegen wie bei den Kernel-basierten Techniken: Der Start einer Xen-VM benötigt etwa 20 Sekunden, während eine OpenVZ-VM nach nur einer Sekunde betriebsbereit ist (siehe Abschnitt 4.2.4). Die Möglichkeit, unterschiedliche Betriebssysteme zu verwenden, erfordert also erheblichen Mehraufwand, ist aber für den vorliegenden Anwendungsfall gar nicht erforderlich, da der Schwerpunkt der Simulation auf der Netzwerkumgebung liegt.

Aus diesen Gründen ist Kernel-basierte Virtualisierung für die vorliegende Anwendung am geeignetsten. Unter Linux gibt es mehrere Implementierungen von Kernel-basierter Virtualisierung, jedoch kommt nur OpenVZ [32] ohne Einschränkungen in Frage. User-Mode Linux [33] ist durch seinen im Grunde emulierenden Ansatz nicht leistungsfähig genug (Barham et al. [30] zeigen Leistungseinbußen von 50 bis 80% auf), Linux VServer [34] unterstützt bis dato kein IPv6,

und KVM [35] erfordert hardwareseitige Virtualisierungsunterstützung, die bisher nicht weit verbreitet verfügbar ist.

VTE wurde mit OpenVZ Version 2.6.18-ovz028test19 entwickelt. Frühere Versionen von OpenVZ weisen leider diverse Probleme mit dem virtuellen Netzwerk auf. OpenVZ besteht aus zwei Teilen, einem Patch für den Linux-Kernel und dem Steuerprogramm vzctl, mit dem man VMs starten, stoppen und konfigurieren kann. Eine virtuelle Maschine wird bei OpenVZ mit *Virtual Environment* bezeichnet. Es handelt sich dabei um einen speziellen Ausführungskontext im Kernel, der über ein eigenes Dateisystem, Prozessraum und Netzwerk verfügt und von allen anderen VMs isoliert ist. Jede VM wird durch eine numerische ID gekennzeichnet. Nach außen besteht eine OpenVZ-VM aus der *Private Area*, einem Verzeichnis, das das Dateisystem der VM enthält, und einer Konfigurationsdatei, in der Einstellungen wie virtuelle Netzwerkkarten gespeichert und Ressourcenbeschränkungen festgelegt werden.

3.7.2 Dateisystemerzeugung

Ein wesentlicher Teil des Zeitbedarfs bei der Ausführung von Tests in einer simulierten Umgebung ist die Zeit, die benötigt wird, um die Umgebung aufzubauen. OpenVZ ist als Kernel-basierter Virtualisierer schon von sich aus sehr leichtgewichtig, und da die VMs im bereits laufenden Kernel betrieben werden, ist der Startaufwand sehr gering. Hauptsächlich muss ein neuer init-Prozess erzeugt werden, der dann lediglich noch die benötigten Daemons wie etwa syslogd und sshd startet. Den größten Zeitaufwand bei der Erzeugung einer VM nimmt daher das Aufsetzen des Dateisystems in Anspruch. Das Dateisystem einer VM ist etwa 115 MB groß, allein diese Datenmenge zu erzeugen beansprucht eine nicht unerhebliche Zeit.

Die Erzeugung von VMs kann daher erheblich beschleunigt werden, indem nicht für jede VM ein eigenes Dateisystem angelegt wird. Stattdessen existiert das Dateisystem bei VTE physisch nur einmal und wird für jede neue VM lediglich ein weiteres Mal gemountet, und zwar mit *Copy on Write*-Semantik. Das heißt, das Dateisystem selbst ist schreibgeschützt, und sämtliche Schreibzugriffe werden in ein separates Verzeichnis umgeleitet. Dadurch müssen Dateien erst dann ein weiteres Mal physisch erzeugt werden, wenn auf sie geschrieben werden soll. Der größte Teil der Zugriffe einer VM auf ihr Dateisystem ist aber ohnehin lesend, im Betrieb fallen lediglich einige Logdateien und Ähnliches an.

Die Umsetzung von *Copy on Write* erfolgt mit UnionFS [36], einem Dateisystem, das mehrere Verzeichnisse als eines zusammenfasst, und dabei für jeden Teilbaum festlegen kann, ob er beschreibbar ist oder nicht. Die zusammengefassten Verzeichnisse werden wie Folien übereinander gelegt, und wenn eine Datei in einer Ebene nicht vorhanden ist, wird zunächst in den darunterliegenden Ebenen nach ihr gesucht.

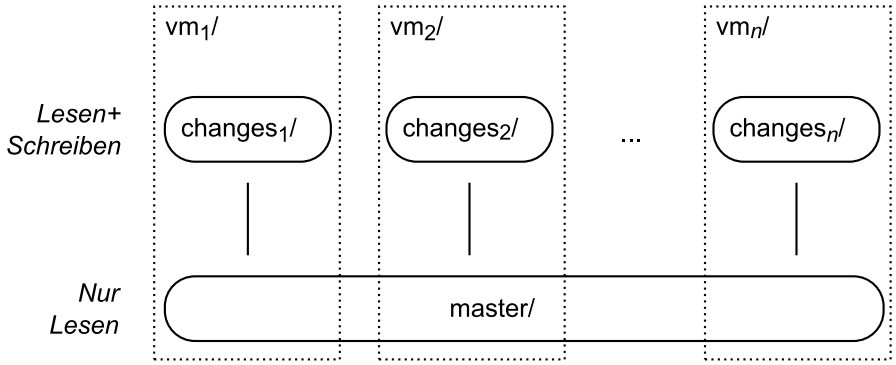


Abbildung 3.7: *Copy on Write* mit UnionFS.

Abbildung 3.7 stellt die Funktionsweise von UnionFS schematisch dar: vm_1 ist die Private Area der virtuellen Maschine Nummer 1, sie ist ein UnionFS-Mount von $changes_1$ (schreibbar) und master (dem eigentlichen VM-Dateisystem, schreibgeschützt). $changes_1$ ist zunächst leer, alle Leseoperationen werden an master durchgereicht. Schreiboperationen werden in $changes_1$ ausgeführt, da master schreibgeschützt ist, somit ist es möglich, master für weitere VMs als Grundlage zu verwenden.

Da in OpenVZ das Dateisystem virtualisiert wird, wurde die Signatur einiger Kernel-Funktionen geändert, um das jeweilige Virtual Environment mit übergeben zu können. Damit UnionFS gegen einen OpenVZ-Kernel kompiliert, musste es an die geänderten Signaturen angepasst werden.

Auch die Erzeugung des Master-Dateisystems nimmt eine gewisse Zeit in Anspruch, deshalb wird es nicht bei jeder Simulation neu erzeugt, sondern nur einmal in der Setup-Phase von VTE (siehe Abschnitt 3.3) angelegt und dann zwischengespeichert. VTE erzeugt das Dateisystem vollautomatisch, als Grundlage dafür dient debootstrap, der automatische Installationsmechanismus von Debian GNU/Linux [37]. Diese Grundinstallation wird noch etwas nachbearbeitet, nicht benötigte Pakete wie PPP oder der Mailserver werden entfernt und andere wie SSH zusätzlich installiert. Außerdem wird das System für den Betrieb unter OpenVZ konfiguriert, dazu werden beispielsweise Konsolen-Logins via getty deaktiviert, da OpenVZ keine Konsole besitzt. Um batch-gesteuert mit dem System arbeiten zu können, werden Passwörter deaktiviert, und der SSH-Host-Key global als bekannt eingestellt. Da alle VMs auf dem gleichen Dateisystem basieren, ist dadurch uneingeschränkte Kommunikation gewährleistet, was beispielsweise für Open MPI wichtig ist. Das fertige Dateisystem wird in einem Tarball zwischengespeichert. Der Tarball erlaubt es auch normalen Benutzern, Device-Files, Dateien, die root gehören, etc. zwischenzuspeichern. Gepackt benötigt das Dateisystem noch etwa 40 MB Speicherplatz.

Das Entpacken des Master-Dateisystems aus dem Tarball dauert beim Start der Si-

mulation immer noch eine gewisse Zeit. Dies legt nahe, das Master-Dateisystem einfach in entpackter Form abzulegen. Leider ist das im Allgemeinen nicht möglich, da OpenVZ nicht damit umgehen kann, wenn die Private Area auf einem NFS-Mount liegt, was bei Home-Verzeichnissen jedoch sehr üblich ist. Deswegen muss VTE das Dateisystem in einem temporären Verzeichnis ablegen, das auf einer lokalen Festplatte liegt. Der Tarball muss pro Simulation jedoch nur einmal entpackt werden, sodass der Zeitaufwand dafür nicht ins Gewicht fällt.

Mittels *Copy on Write* kann VTE eine neue VM nun in weniger als einer Sekunde starten, da jegliches Kopieren entfällt und bei der Erzeugung einer VM nur noch ein Mount-Vorgang erforderlich ist, der wesentlich schneller vonstatten geht. Außerdem hat dies den Vorteil, dass sich der Speicherbedarf bei mehreren VMs auf das reduziert, was in jeder VM tatsächlich an Daten erzeugt wird, da das Dateisystem nur noch einmal physisch auf der Festplatte existieren muss. Typischerweise fallen pro VM nur wenige 100 KB Daten an, der benötigte Festplattenspeicher ist also vernachlässigbar.

3.7.3 Hostmount

Die zu testende Software muss für die Tests zunächst in die simulierte Umgebung eingebracht werden, d. h. von den VMs aus verfügbar sein. Dazu ist es jedoch nicht erforderlich, die Software langwierig in alle VMs zu kopieren, denn OpenVZ sieht für jedes *Virtual Environment* einen Mountpoint im Wirt vor, unter dem das Dateisystem der VM sichtbar gemacht wird. Hier können nun mittels `mount --bind` beliebige Verzeichnisse des Wirts eingeblendet werden, beispielsweise das Arbeitsverzeichnis des Entwicklers. Oftmals (etwa bei Open MPI, vergleiche Abschnitt 4.1.3) spielen die exakten Pfadnamen eine Rolle, deshalb bietet VTE die Möglichkeit, Wirtsverzeichnisse unter beliebigen Pfaden in die VMs einzublenden.

3.7.4 Rekonfigurierbarkeit

Die Aufgabe von VTE ist, Software in vielen unterschiedlichen Netzwerkkonfigurationen zu testen. Deshalb soll es möglich sein, innerhalb eines Testlaufs viele Netzwerkkonfigurationen nacheinander zu simulieren, anstatt jedesmal die komplette Testumgebung herunterfahren zu müssen und in veränderter Konfiguration neu zu starten. Dabei dürfen die verschiedenen Konfigurationen sich aber gegenseitig nicht beeinflussen. Deshalb bietet jede virtuelle Netzwerkkomponente eine Funktion `destroy`, mit der sie komplett entfernt werden kann. Dadurch wird das virtuelle System in den Zustand zurückversetzt, der bestand, bevor diese Komponente erzeugt wurde.

Im folgenden Beispiel wird zunächst ein Netzwerk mit zwei Clustern aufgebaut, die über einen Router mit IPv4 verbunden sind. Nachdem dieses entfernt wurde, werden alle VMs zu einem Cluster zusammengeschlossen, indem sie mit einem neuen Switch verbunden werden, diesmal mit IPv6. Von der ursprünglichen Zwei-Cluster-Konfiguration bleiben keine Rückstände (IPv4-Adressen, Routen oder anderes), die die neue Ein-Cluster-Konfiguration beeinflussen.

```
cluster1 = Cluster.new(3)
cluster2 = Cluster.new(3)
switch1 = Switch.new(:ipv4).connect(cluster1.vm)
switch2 = Switch.new(:ipv4).connect(cluster2.vm)
router = Router.new.connect(switch1).connect(switch2)

switch1.destroy
switch2.destroy
router.destroy

switch_v6 = Switch.new(:ipv6).connect(cluster1.vm + cluster2.vm)
```

Durch diese Rekonfigurierbarkeit der Simulationsumgebung wird der Zeitaufwand für das Testen von verschiedenartigen Netzwerkkonfigurationen gesenkt, da nicht die gesamte Umgebung ab- und wieder aufgebaut werden muss.

3.7.5 Mehrere Wirtsrechner

Für einfache Funktionstests sind virtuelle Maschinen effizient genug, um sie auf einem einzigen physischen Rechner durchzuführen (siehe auch Abschnitt 4.2). Wenn die Tests aber größeren Rechenaufwand erfordern, wird dies unter Umständen nicht mehr ausreichen – schließlich muss der eine Rechner die Leistung für alle virtuellen Maschinen erbringen. Für diesen Fall ist VTE in der Lage, mehrere Wirtsrechner zur Simulation eines virtuellen Multiclusters zu verwenden.

Da VTE zur Steuerung der Simulationsumgebung ausschließlich über Shell-Kommandos mit dem Wirtsrechner interagiert, können weitere Rechner einfach mit in die Simulation eingebunden werden, indem die entsprechenden Kommandos per SSH übermittelt werden. VTE muss dazu auf allen teilnehmenden Rechnern installiert sein. Die Simulation wird zentral von dem Rechner aus gesteuert, von dem sie gestartet wurde, denn dort liegen alle Konfigurationsinformationen für die Simulationsumgebung vor. Die weiteren Rechner werden von dort aus ferngesteuert.

Durch das Einbinden von mehreren physischen Rechnern in die Simulation ist VTE in der Lage, auch Tests mit höheren Leistungsanforderungen schnell durchzuführen, weil die Leistung für mehrere virtuelle Maschinen auf diese Weise nicht nur von einem einzigen Rechner erbracht werden muss.

Mit den vorgestellten Mitteln ist es möglich, die Geschwindigkeit von wesentlichen Operationen von VTE zu erhöhen, etwa indem der Start der Testumgebung

beschleunigt wird. Durch diese Effizienzsteigerungen ist VTE in der Lage, Tests ohne großen Zeitaufwand auszuführen.

Die dargestellte Architektur wurde entwickelt, um die in Abschnitt 3.1 aufgeführten Anforderungen für entwicklungsbegleitendes Testen zu ermöglichen. Bei der Implementierung der einzelnen Aspekte wurden diese Kriterien daran anschließend möglichst konsequent umgesetzt. Das so entwickelte VTE ist ein Werkzeug für Entwickler, das die Hürde zum entwicklungsbegleitenden Testen von verteilten Systemen senken soll. Im folgenden Kapitel wird überprüft, inwiefern sich dies im praktischen Betrieb niederschlägt.

4 Evaluation

Nachdem im vorigen Kapitel die Architektur und Implementierung von VTE vorgestellt wurde, erfolgt in diesem Kapitel die Überprüfung seiner Leistungsfähigkeit im realen Einsatz. In den folgenden Abschnitten wird untersucht, wie gut die Konfigurationssprache von VTE die Formulierung von Tests ermöglicht, wie sich verschiedene Netzwerkkonfigurationen mit VTE abbilden lassen, und ob die Ausführung der Tests schnell genug ist, um entwicklungsbegleitendes Testen zu ermöglichen.

4.1 Testbeispiele

Die Hauptaufgabe von VTE besteht darin, dem Entwickler eine einfache Möglichkeit zur Verfügung zu stellen, komplexe Netzwerkumgebungen zu beschreiben. Dazu wurde die in Abschnitt 3.4 vorgestellte Konfigurationssprache entwickelt. Die Ausdrucksmöglichkeiten von VTE zur Spezifikation von Tests werden in den folgenden Abschnitten anhand von Beispielen für das Verhalten von MPI-Bibliotheken in unterschiedlichen Situationen verdeutlicht. Die Beispiele entstammen der alltäglichen Entwicklungspraxis (Stand April 2007) und illustrieren, wie regelmäßiges Testen mit Hilfe von VTE zur Vermeidung derartiger Probleme beitragen kann.

4.1.1 NAT

Viele Cluster-Installationen nutzen für ihre Knoten private IPv4-Adressen und verbinden sie via NAT mit der Außenwelt. Zwei derartige Cluster direkt zu einem Multicluster zu verbinden ist nicht möglich, denn die privaten Adressen des einen Clusters sind vom anderen aus nicht erreichbar, und umgekehrt. MPI-Bibliotheken sollten diesen Fehlerfall erkennen, dies wird am Beispiel von LAM/MPI und Open MPI mit VTE untersucht.

Zwei Cluster mit je drei Knoten, die private IPv4-Adressen verwenden und mit NAT verbunden sind, können wie folgt modelliert werden:

```
cluster1 = Cluster.new(3)
cluster2 = Cluster.new(3)
switch1 = Switch.new("192.168.5.0/24").connect(cluster1.vm)
switch2 = Switch.new("192.168.6.0/24").connect(cluster2.vm)
router = Router.new.connect(switch1, nat=true).connect(switch2, nat=true)
```

LAM/MPI [38] wird mit dem Kommando lamboot gestartet.

```
hostnames = (cluster1.vm + cluster2.vm).map { |vm| vm.hostname }.join("\n")
cluster1.vm[0].cmd("echo '#{hostnames}' > lamhosts")
cluster1.vm[0].cmd("lamboot lamhosts")
```

lamboot versucht, vom ersten Cluster aus auf allen Knoten den lamd Daemon zu starten, aber da die Knoten des zweiten Clusters wegen ihrer privaten Adressen nicht erreichbar sind, bricht es mit einer Fehlermeldung ab.

```
ERROR: LAM/MPI unexpectedly received the following on stderr:
ssh: connect to host vm3 port 22: Network is unreachable
```

Open MPI [39] startet seinen Daemon orted automatisch, wenn mpiexec aufgerufen wird.

```
hostnames = (cluster1.vm + cluster2.vm).map { |vm| vm.hostname }.join(",")
cluster1.vm[0].cmd("mpiexec -np #{cluster.vm.size} -host #{hostnames} ./ringtest")
```

Open MPI kann ebenfalls einige Knoten nicht erreichen, aber anstatt abzubrechen, bleibt es hängen, bis der Timeout von VTE den Prozess beendet.

```
ssh: connect to host vm3 port 22: Network is unreachable
[vm0:21702] ERROR: A daemon on node vm3 failed to start as expected.
[vm0:21702] ERROR: There may be more information available from
[vm0:21702] ERROR: the remote shell (see above).
Timeout: aborting command "vzctl" with signal 9
```

Das Problem ist, dass Open MPI, nachdem es den Fehler bemerkt hat, an die orted auf *allen* Knoten den Befehl schickt, sich zu beenden – inklusive derjenigen Knoten, die von vornherein nicht erreichbar waren und den Fehler erst ausgelöst haben. Da diese Knoten nie eine Bestätigung zurückschicken, dass sie sich erfolgreich beendet haben, bleibt der Startprozess hängen, da er noch auf sie wartet.

4.1.2 Dual-Stack

Bei der Migration zu IPv6 [40] ist oft eine Dual-Stack-Konfiguration üblich, d. h. beide Adressfamilien sind auf demselben Netzwerkinterface konfiguriert, und die Rechner werden mit einem einzigen Switch verbunden. In VTE wird ein IP-Subnetz üblicherweise durch einen Switch repräsentiert, in diesem Fall existiert aber nur ein Switch mit zwei Subnetzen. Deshalb muss das zweite IP-Subnetz manuell konfiguriert werden, dazu bietet VTE Hilfsfunktionen an, die die Verwaltung von IP-Adressen erleichtern.

```
cluster = Cluster.new(3)
Switch.new(:ipv6).connect(cluster.vm)
net = Switch.generate_network(:ipv4)
cluster.vm.each_with_index do |v, i|
  v.configure_ipv4(net.nth(i+1)) unless v.hostname_short == "vm1"
end
Cluster.update_hostfile
```

Die Konfigurationssprache von VTE ist reines Ruby, dadurch stehen dem Entwickler alle normalen Sprachmittel zur Verfügung. Somit lassen sich beispielsweise Ausnahmen wie „alle Knoten haben IPv4 außer vm1“ gut formulieren.

Open MPI mit der IPv6-Erweiterung [41], die am Lehrstuhl entwickelt wurde, verwendet in dieser Umgebung sowohl IPv4 als auch IPv6, um Verbindungen herzustellen, je nachdem, welche Konnektivität jeweils verfügbar ist. Untersucht man den Datenverkehr auf dem virtuellen Switch mit Hilfe von tcpdump, stellt man beispielsweise fest, dass vm0 sich mit vm2 über IPv4 verbindet (1.0.0.1–1.0.0.3), mit vm1 aber über IPv6 (aa01::1–aa01::2).

```
# tcpdump -i sw1
IP 1.0.0.1.33126 > 1.0.0.3.52353
IP 1.0.0.3.52353 > 1.0.0.1.33126
[...]
2001:638:906:aa01::1.38252 > 2001:638:906:aa01::2.43756
2001:638:906:aa01::2.43756 > 2001:638:906:aa01::1.38252
```

4.1.3 Installationspfad

Wenn Open MPI unter einem anderen Pfad installiert ist, als bei der Kompilation eingestellt wurde, findet es seine Hilfsprogramme und Bibliotheken nicht. Eine Möglichkeit, um den Installationspfad (das *Präfix*) explizit anzugeben, ist ein sogenannter *Application Context*, eine Datei, in der verschiedene Parameter für den mpiexec-Aufruf zusammengefasst werden. So ist es zumindest in der Dokumentation beschrieben. Mit VTE können zwei Knoten mit unterschiedlichen Open MPI-Installationspfaden wie folgt beschrieben werden:

```
cluster = Cluster.new(2)
switch = Switch.new(:ipv4).connect(cluster.vm)
cluster.vm[0].mount("/real/path/to/openmpi", "/usr/local/openmpi")
cluster.vm[1].mount("/real/path/to/openmpi", "/opt/openmpi")
appcontext = "-np 1 -host vm0 --prefix /usr/local/openmpi hostname \
            -np 1 -host vm1 --prefix /opt/openmpi hostname"
cluster.vm[0].cmd("echo '#{appcontext}' > appcontext")
cluster.vm[0].cmd("mpiexec --appfile appcontext date")
```

Leider scheint die --prefix-Einstellung keine Auswirkung zu haben, denn mpiexec terminiert auf vm1 mit der folgenden Fehlermeldung:

```
/usr/local/openmpi/bin/orted: No such file or directory
```

4.1.4 Striping

Wenn zwischen zwei Knoten mehrere Netzwerkverbindungen vorhanden sind, kann Open MPI *Striping* verwenden, um den Durchsatz zu vergrößern. Dabei werden große Nachrichten fragmentiert und parallel über mehrere Netzwerkschnittstellen verschickt. In VTE kann jede VM mehrere Netzwerkschnittstellen erhalten, indem beispielsweise beim Verbinden mit einem Switch eine Schnittstellennummer angegeben wird:

```
cluster = Cluster.new(2)
Switch.new(:ipv4).connect(cluster.vm, 0)
Switch.new(:ipv4).connect(cluster.vm, 1)
```

In diesem Beispiel erhält jede VM zwei virtuelle Netzwerkkarten, eth0 und eth1. Wenn die Nachrichten des Ping-Pong Benchmark aus der Intel MPI-Benchmark-Suite (IMB 3.0, [42]) groß genug werden, verwendet Open MPI Striping, um die Daten über alle verfügbaren Schnittstellen zu verteilen. Daher können mit tcpdump auf beiden virtuellen Switches Verbindungen festgestellt werden.

```
# tcpdump -i sw1
IP 1.0.0.2.56003 > 1.0.0.3.33364
IP 1.0.0.3.33364 > 1.0.0.2.56003
# tcpdump -i sw2
IP 2.0.0.2.54115 > 2.0.0.3.59036
IP 2.0.0.3.59036 > 2.0.0.2.54115
```

Wie die vorgestellten Beispiele zeigen, ist es mit der Konfigurationssprache von VTE gut möglich, in kurzen, einfachen Beschreibungen Testumgebungen für vielfältige Fragestellungen zu formulieren.

4.2 Leistungsfähigkeit

Wie in Abschnitt 3.7 dargestellt wurde, ist ein wichtiger Schwerpunkt von VTE, dass die Ausführung der Tests schnell vonstatten geht. In den folgenden Abschnitten soll die Leistungsfähigkeit von VTE untersucht und geprüft werden, ob der Zeitbedarf für die Ausführung von Tests mit VTE niedrig genug ist, um regelmäßiges Testen zu ermöglichen.

4.2.1 CPU-Mehraufwand

Um die Leistungsfähigkeit der virtuellen Maschinen zu bestimmen, wird zunächst der reine CPU-Mehraufwand ermittelt, den VMs gegenüber dem nicht virtualisierten System aufweisen. Dies erfolgt mit dem Dhrystone-Benchmark, einem synthetischen Benchmark zur Bestimmung der CPU-Leistung. Er besteht aus einer Menge

einzelner Funktionen, die in realen Programmen häufig auftretende Befehlsabläufe nachbilden [43]. Dhrystone umfasst Ganzzahl- und String-Operationen, nicht jedoch Gleitkommaoperationen, und war der verbreitetste CPU-Benchmark, bis er von SPECint [44] abgelöst wurde, der jedoch nicht frei verfügbar ist. Sein Ergebnis wird in Dhrystones pro Sekunde angegeben, der Anzahl Durchläufe durch die innere Schleife, die der Computer in einer Sekunde absolvieren kann. Zur genaueren Ermittlung wird der Benchmark eine einstellbare Anzahl von Durchläufen wiederholt. Auf dem Testrechner (Intel Pentium 4 mit 3 GHz, 2 GB RAM) werden 10^7 Wiederholungen benötigt, damit Dhrystone überhaupt zuverlässig ein Ergebnis angeben kann.

Der CPU-Mehraufwand der virtuellen Maschinen wird bestimmt, indem die gleiche Gesamtzahl von Wiederholungen (10^9) einerseits auf mehrere VMs und andererseits auf ebensoviele geforkte Prozesse aufgeteilt wird. Die Abbildungen 4.1 und 4.2 auf der nächsten Seite zeigen die Gesamtlaufzeit sowie die von allen Prozessen zusammen erreichte Leistung. Wie zu erwarten ist, bleiben beide Werte in etwa gleich, denn einerseits verändert sich der insgesamt zu erledigende Aufwand nicht, er wird nur aufgeteilt. Andererseits findet die Berechnung der Teile „gleichzeitig“ statt, sodass jedem der n Teile nur $\frac{1}{n}$ der CPU zur Verfügung steht.

Der Leistungsunterschied zwischen VMs und normalen Prozessen ist mit durchschnittlich 3% recht gering. Das liegt daran, dass OpenVZ als Kernel-basierte Virtualisierungstechnik den Prozessen in virtuellen Maschinen im Wesentlichen nur einen etwas umfangreicheren Ausführungskontext (das Virtual Environment) zuordnet, es sich ansonsten aber um „normale“ Prozesse handelt, die vom Scheduler ebenso wie nicht virtualisierte Prozesse verwaltet werden.

Natürlich muss dennoch ein einzelner physischer Rechner die Leistung für alle virtuellen Maschinen erbringen, aber solange die Anforderungen der Tests nicht zu rechenintensiv sind, ist es durch den geringen Leistungsverlust an die VMs selbst problemlos möglich, eine Testumgebung mit mehreren Knoten zu simulieren. Falls doch mehr Rechenleistung benötigt wird, ist VTE in der Lage, mehrere Wirtsrechner in die Simulation einzubinden, wie in Abschnitt 3.7.5 dargestellt.

4.2.2 Speicherverbrauch

Die Zahl der virtuellen Maschinen, die gleichzeitig ausgeführt werden können, hängt neben den Kosten hinsichtlich CPU auch von Festplatten- und Hauptspeicherverbrauch ab. Der CPU-Mehraufwand ist bei OpenVZ, wie im vorigen Abschnitt dargestellt, recht gering. An den Festplattenplatz stellt VTE, insbesondere durch *Copy on Write*, keine großen Anforderungen, wie in Abschnitt 3.7.2 erläutert wurde.

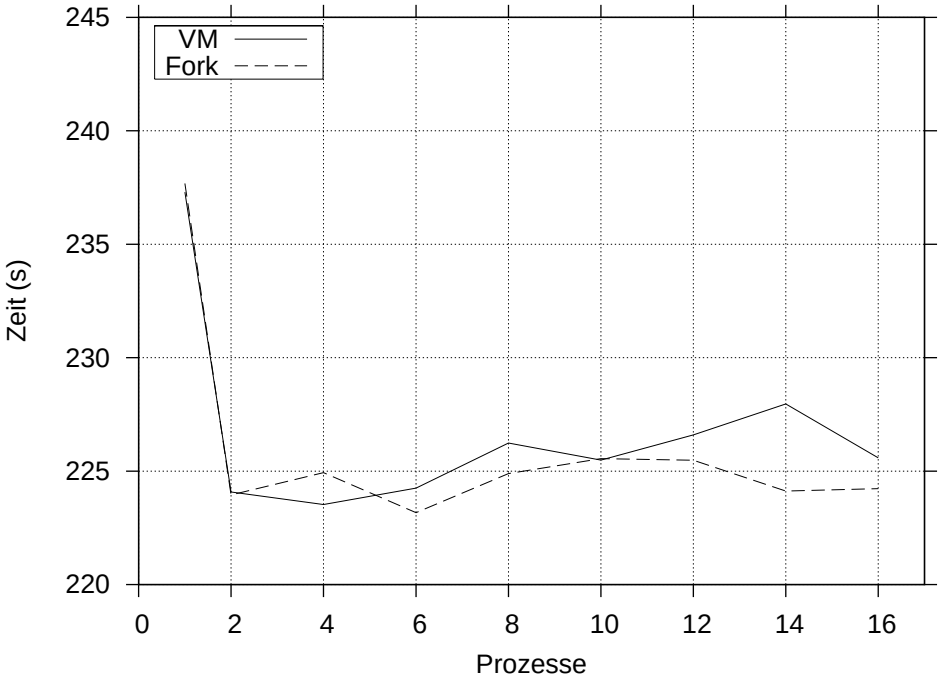


Abbildung 4.1: Gesamtlaufzeit des Dhrystone-Benchmark.

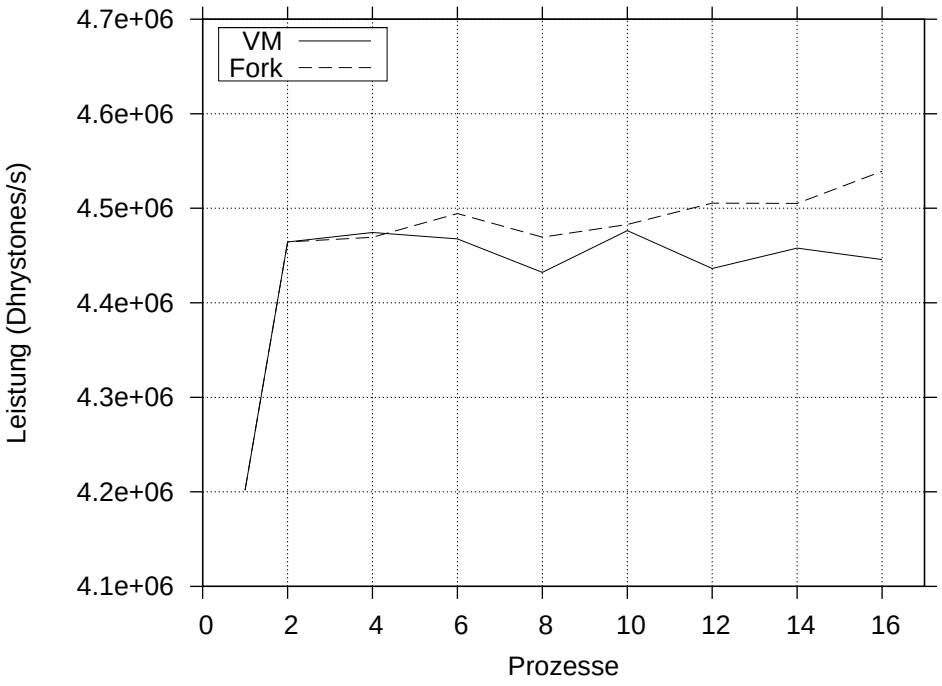


Abbildung 4.2: Von allen Prozessen zusammen erreichte CPU-Leistung (Dhrystones/s).

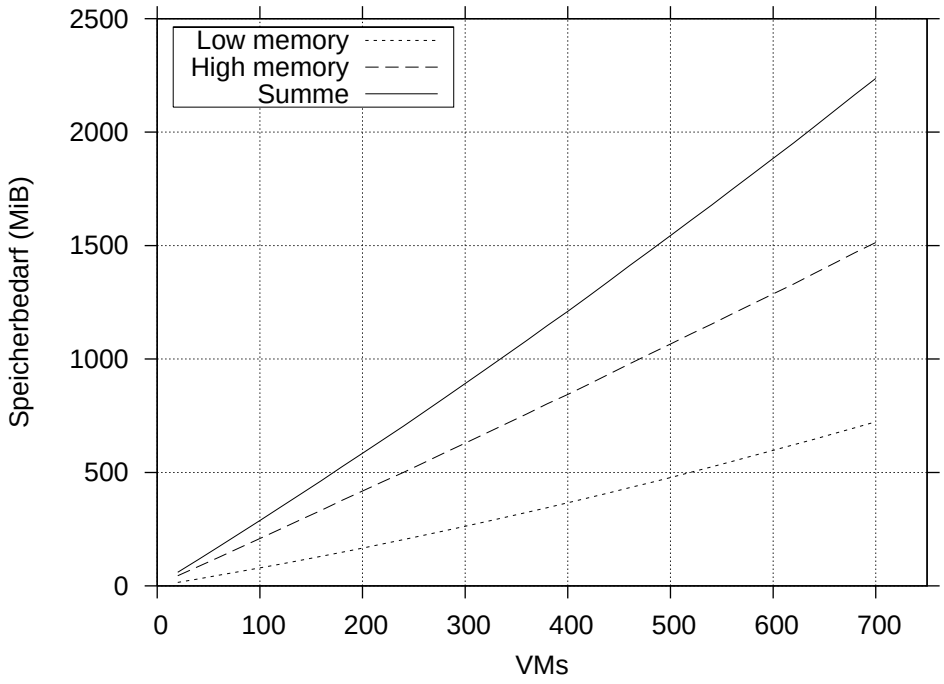


Abbildung 4.3: Speicherverbrauch von virtuellen Maschinen.

Beim Hauptspeicherbedarf ist zumindest unter x86 zwischen *low* und *high memory* zu unterscheiden, da nicht aller Speicher, der physisch vorhanden ist, virtuell adressierbar ist, und daher dem Kernel nicht ständig zur Verfügung steht [45]. Kernel-interne Datenstrukturen wie Netzwerkpuffer, Prozessdeskriptoren, Dateisystemeinträge etc. können nur im *low memory*, dem stets verfügbaren Speicher, abgelegt werden, während Userland-Prozesse ebenso auch im *high memory* abgelegt werden können. Unter x86 teilt der Kernel den Speicher so ein, dass maximal etwas unter 1 GB *low memory* zur Verfügung steht (der genaue Wert hängt u. a. von unterschiedlichen Puffergrößen ab).

Abbildung 4.3 zeigt den Speicherverbrauch beim Test der Open MPI-Laufzeitumgebung ORTE. Diese startet auf allen Knoten per SSH den orted-Daemon, der die MPI-Kommunikation abwickelt. In diesem Fall ist der *low memory* nicht der limitierende Faktor. Bei etwas über 600 Knoten sind die physisch vorhandenen 2 GB RAM ausgelastet und der Testrechner beginnt zu swappen, noch bevor der *low memory* für Netzwerkpuffer etc. aufgebraucht ist. Für Tests, die wenig Anforderungen an die Netzwerkkommunikation und CPU-Leistung stellen, etwa um zu überprüfen, ob überhaupt alle Knoten erreichbar sind, ist VTE also in der Lage, auch größere Teststellungen mit mehreren Hundert Knoten zu simulieren.

4.2.3 Netzwerkleistung

Neben der Leistungsfähigkeit der virtuellen Maschinen selbst ist auch die des virtuellen Netzwerks ein Faktor für die Ausführungsgeschwindigkeit der Tests. Abbildung 4.4 auf der nächsten Seite zeigt die Transferraten, die mit dem Ping-Pong Benchmark aus der Intel MPI-Benchmark-Suite ermittelt wurden. Die Messung wurde zwischen zwei VMs mit virtuellem Netzwerk, zwei Prozessen auf demselben Rechner über Loopback und zwischen zwei physischen Rechnern über Gigabit Ethernet durchgeführt. Das virtuelle Netzwerk erreicht einen wesentlich geringeren Durchsatz als die Verbindung über Loopback, was in Teilen dadurch begründet ist, dass im virtuellen Netzwerk der Netzwerk-Stack komplett bis zur Ethernet-Ebene durchlaufen wird, während der Kernel dies bei der Kommunikation „mit sich selbst“ schon auf der IP-Ebene kurzschließt [46]. Aber selbst im Vergleich mit dem physischen Netzwerk bleibt das virtuelle Netz weit zurück.

Große Leistungsanforderungen müssen aber für die vorliegende Anwendung gar nicht erfüllt werden, denn die Hauptaufgabe sind funktionale Tests, beispielsweise ob überhaupt eine MPI-Kommunikation bei einer gegebenen (komplexen) Netzwerkkonfiguration zustande kommt. Ein gängiger Test dafür ist eine ringförmige Kommunikation, bei der ein Wert von Knoten zu Knoten weitergegeben wird, bis er wieder am Ausgangspunkt ankommt. Da in diesem Fall immer nur je zwei Knoten gleichzeitig kommunizieren, ist die Belastung des Netzwerks eher gering, und das virtuelle Netz bei weitem leistungsfähig genug, wie Abbildung 4.5 auf der nächsten Seite zeigt: Der Ringtest benötigt im virtuellen Netz ungefähr doppelt so viel Zeit wie zwischen geforkten Prozessen, vor allem aber ist der absolute Zeitbedarf klein genug, dass er selbst mit größeren Knotenzahlen (20 Sekunden bei 100 Knoten) gut als routinemäßiger Test einsetzbar ist.

4.2.4 Start- und Stoppzeit

Für den Zeitbedarf der Tests ist nicht nur entscheidend, wie schnell die Tests an sich ablaufen, sondern auch wie lange die Testumgebung zum Auf- und Abbau benötigt. Abbildung 4.6 auf Seite 48 zeigt die Gesamtzeit zum Starten und Stoppen von virtuellen Maschinen. Die Kurven für zwei und drei Rechner zeigen die Zeit, die benötigt wird, um auf jedem der Rechner die jeweilige Anzahl VMs auszuführen, also einen simulierten Multicluster aufzubauen (siehe Abschnitt 3.7.5). Alle VMs wurden stets parallel gestartet und gestoppt, da ein aktueller Rechner bei serieller Abarbeitung bei weitem nicht ausgelastet ist.

OpenVZ benötigt als Kernel-basierte Virtualisierung schon von sich aus wenig Zeit zum Start einer VM, und da die Erzeugung des Dateisystems mittels *Copy on Write* (siehe Abschnitt 3.7.2) nur noch einen Mountvorgang erfordert, kann eine neue

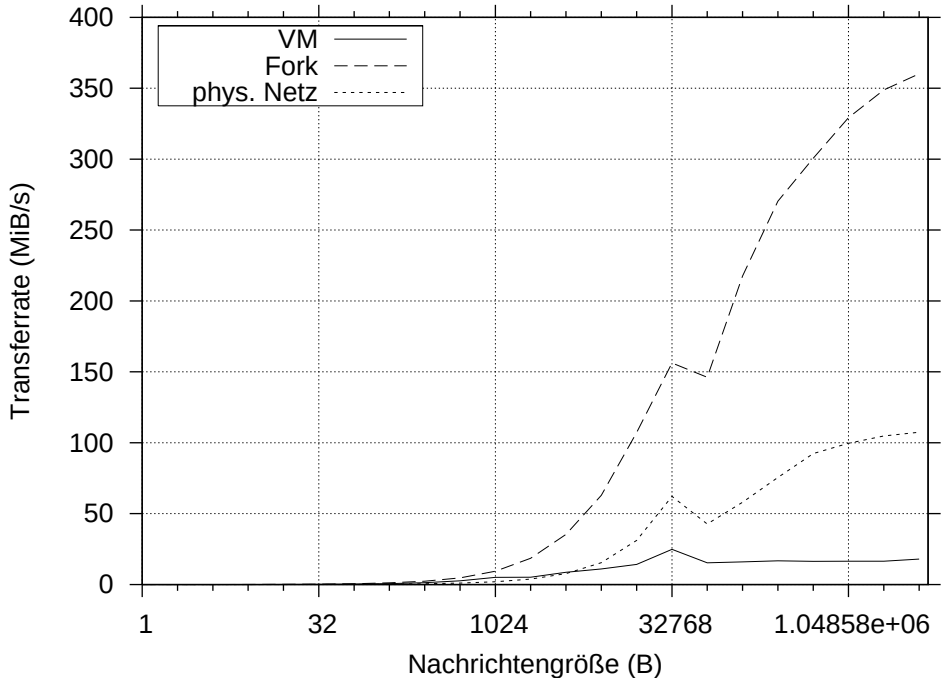


Abbildung 4.4: Ping-Pong Benchmark.

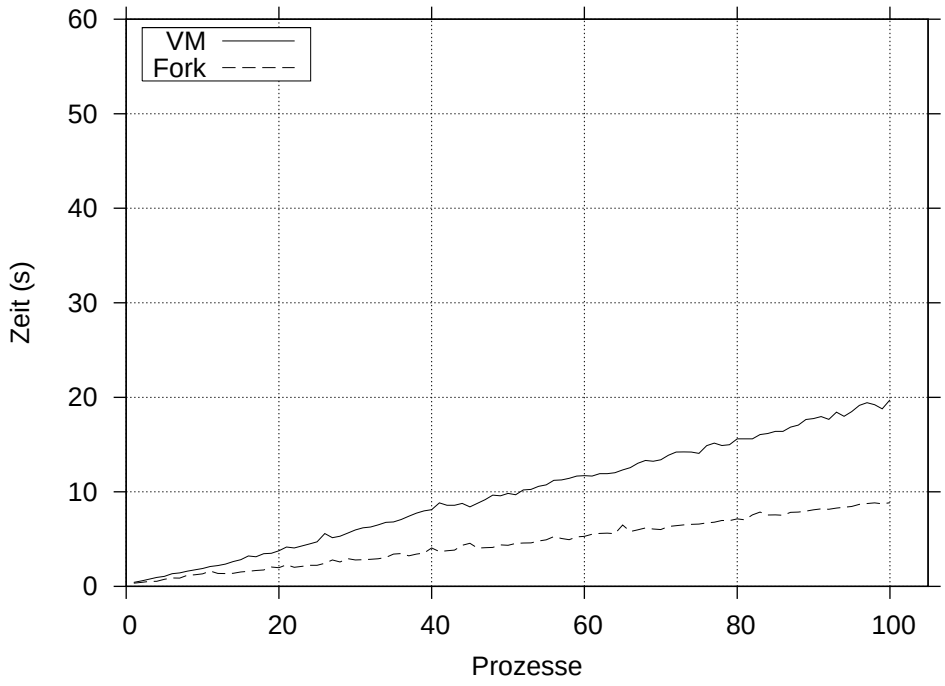


Abbildung 4.5: MPI Ringtest.

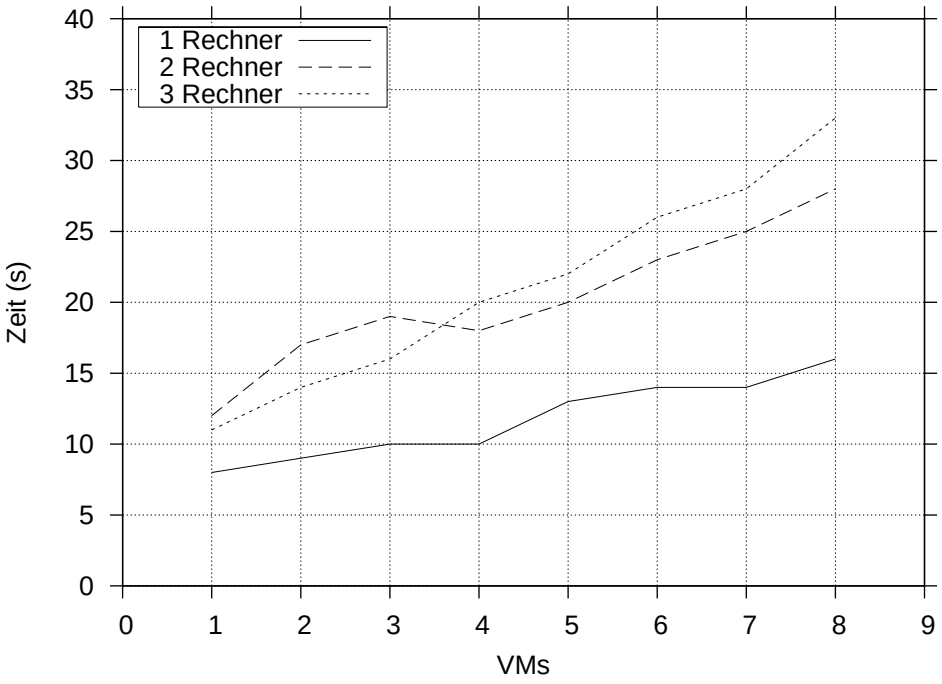


Abbildung 4.6: Zeitbedarf für Start und Stopp von virtuellen Maschinen.

VM in weniger als einer Sekunde gestartet werden. Beim Stoppen hingegen müssen die erzeugten Dateien der VM (Logdateien und andere temporäre Dateien) wieder entfernt werden, was etwas mehr Zeit in Anspruch nimmt.

Wenn sich die VMs auf mehreren physischen Rechnern befinden, wird mehr Zeit benötigt, weil alle Kommandos zur Konfiguration der VMs und des Simulations-Netzwerks via Netzwerk an die anderen Rechner übermittelt werden müssen, was länger dauert, als ein Kommando auf ein und demselben Rechner auszuführen. Trotzdem geht das Starten und Stoppen der VMs insgesamt so rasch, dass es keine Behinderung für die Gesamtlaufzeit der Tests darstellt.

Die vorgestellten Messungen und Anwendungsbeispiele zeigen, dass VTE gut für das entwicklungsbegleitende Testen von verteilten Systemen geeignet ist. Die Konfigurationssprache ist einfach zu verwenden und ausdrucksstark. VTE ist in der Lage, auch komplexere Netzwerkkonfigurationen abzubilden und die zu testende Software somit in unterschiedlichen Umgebungen zu testen. Die Leistungsfähigkeit ist ausreichend und auch der Auf- und Abbau der Testumgebung benötigt nur wenig Zeit. Dadurch ist der Aufwand für Tests mit VTE so gering, dass regelmäßiges Testen praktikabel wird.

5 Zusammenfassung

Das in dieser Arbeit entwickelte Testframework zur Simulation komplexer Netzwerktopologien gibt Entwicklern von verteilten Systemen eine Möglichkeit zu kontinuierlichen Funktionstests. Diese müssen üblicherweise mit physischen Teststellungen durchgeführt werden, dafür ist der Aufwand aber unverhältnismäßig hoch, sodass er nur sporadisch, wenn überhaupt, unternommen werden kann.

Das vorgestellte *Virtual Test Environment* (VTE) ermöglicht die prägnante Beschreibung von (Multi-)Cluster-Umgebungen mit frei variierbaren Netzwerkverbindungen, die dann vollautomatisch und ohne großen Zeitbedarf auf virtuellen Maschinen abgebildet werden. Der Aufwand für Tests wird dadurch so gering, dass er keine Hürde mehr für routinemäßiges Testen darstellt. Dies erlaubt es dem Entwickler, kontinuierlich Rückmeldung über das Verhalten der entwickelten Software zu erhalten, und führt dadurch zu höherer Software-Qualität.

Ein wichtiges, bisher noch nicht detailliert betrachtetes Gebiet ist die Leistungsfähigkeit des virtuellen Netzwerks, vor allem im Hinblick darauf, wie man mit seiner Hilfe die Charakteristika von physischen Netzwerkverbindungen wie etwa Gigabit-Ethernet nachbilden kann. Hier wäre zu erforschen, ob sich ein Zusammenhang der Leistung des zu testenden Systems in der simulierten Umgebung mit der in der realen herstellen lässt. Wenn es möglich wäre, Aussagen zu treffen wie „eine Beschleunigung von 10% in der Simulation führt zu 5–20% Geschwindigkeitszuwachs in realer Umgebung“, könnten auch Performance-Tests von einem kurzen Feedback-Zyklus profitieren, und damit ein großes Potenzial für die Optimierung von verteilten Systemen eröffnen.

Literaturverzeichnis

[1] BALZERT, HELMUT: *Lehrbuch der Softwaretechnik, Teil 2: Softwaremanagement, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[2] BECK, KENT: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company, 1999.

[3] BECK, KENT: *Test Driven Development: By Example*. Addison-Wesley Professional, Boston, 2002.

[4] BECK, KENT und ERICH GAMMA: *The xUnit family of testing frameworks*. `http://www.martinfowler.com/bliki/Xunit.html`, zugegriffen am 15.05.2007.

[5] *Mock Objects*. `http://www.mockobjects.com/`, zugegriffen am 15.05.2007.

[6] GHOSH, S. und A. MATHUR: *Issues in testing distributed component-based systems*. In: *Proceedings of the First International ICSE Workshop, Testing Distributed Component-Based Systems*, 1999.

[7] VETTER, JEFFREY S. und BRONIS R. DE SUPINSKI: *Dynamic Software Testing of MPI Applications with Umpire*. sc, 00:51, 2000.

[8] KRAMMER, BETTINA, KATRIN BIDMON, MATTHIAS S. MÜLLER und MICHAEL M. RESCH: *MARMOT: An MPI Analysis and Checking Tool*. In: JOUBERT, NAGEL, PETERS und WALTER (Herausgeber): *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, Seiten 493–500. Elsevier, Amsterdam, 2004.

[9] CLÉMENÇON, CHRISTIAN, JOSEF FRITSCHER, M. J. MEEHAN und ROLAND RÜHL: *An Implementation of Race Detection and Deterministic Replay with MPI*. In: *Euro-Par '95: Proceedings of the First International Euro-Par Conference on Parallel Processing*, Seiten 155–166, London, UK, 1995. Springer-Verlag.

[10] *MPI Test Suites*. `http://www-unix.mcs.anl.gov/mpi/mpi-test/tsuite.html`, zugegriffen am 15.05.2007.

[11] *MPI: A message-passing interface standard*. `http://www.mpi-forum.org/`, zugegriffen am 15.05.2007.

[12] WIKIPEDIA: *Virtualization – Wikipedia, The Free Encyclopedia*, 2007. `http://en.wikipedia.org/w/index.php?title=Virtualization&oldid=118948618`, zugegriffen am 15.05.2007.

[13] CREASY, ROBERT J.: *The Origin of the VM/370 Time-Sharing System*. IBM Journal of Research and Development, 25(5):483–490, 1981.

[14] POPEK, GERALD J. und ROBERT P. GOLDBERG: *Formal Requirements for Virtualizable Third Generation Architectures*. Commun. ACM, 17(7):412–421, 1974.

[15] LAWTON, KEVIN: *Running Multiple Operating Systems Concurrently on an IA32 PC using Virtualization Techniques*, 1999. `http://www.floobydust.com/virtualization/lawton_1999.txt`, zugegriffen am 15.05.2007.

[16] ADAMS, KEITH und OLE AGESEN: *A Comparison of Software and Hardware Techniques for x86 Virtualization*. SIGOPS Oper. Syst. Rev., 40(5):2–13, 2006.

[17] SOLTESZ, STEPHEN, HERBERT PÖTZL, MARC FIUCZYNSKI, ANDY BAVIER und LARRY PETERSON: *Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors*. In: *Proceedings of EuroSys 2007*, Lisbon, Portugal, March 2007.

[18] VALLEE, GEOFFROY und STEPHEN L. SCOTT: *OSCAR Testing with Xen*. In: *HPCS '06: Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment*, Seite 43, Washington, DC, USA, 2006. IEEE Computer Society.

[19] BEGNUM, KYRRE M.: *Managing large networks of virtual machines*. In *Proc. LISA'06: 20th Large Installation System Administration Conference*, pages 205–214, Washington, D.C., December 2006. USENIX Association.

[20] TANENBAUM, ANDREW S.: *Computernetzwerke*. Pearson Studium, München, 4. überarbeitete Auflage, 2004.

[21] *Linux Bridge Utilities*. `http://linux-net.osdl.org/index.php/Bridge`, zugegriffen am 15.05.2007.

[22] BAKER, F.: *Requirements for IP Version 4 Routers*. RFC 1812 (Proposed Standard), Juni 1995. Updated by RFC 2644.

[23] REKHTER, Y., B. MOSKOWITZ, D. KARRENBERG, G. J. DE GROOT und E. LEAR: *Address Allocation for Private Internets*. RFC 1918 (Best Current Practice), Februar 1996.

[24] SRISURESH, P. und K. EGEVANG: *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022 (Informational), Januar 2001.

[25] *netfilter – firewalling, NAT, and packet mangling for Linux*. `http://www.netfilter.org/`, zugegriffen am 15.05.2007.

[26] *tinc: Virtual Private Network daemon*. `http://www.tinc-vpn.org/`, zugegriffen am 15.05.2007.

[27] *OpenVZ Resource Control Parameters: Consistency Check*. `http://wiki.openvz.org/w/index.php?title=UBC_consistency_check&oldid=2473`, zugegriffen am 15.05.2007.

[28] *VMWare*. `http://www.vmware.com/`, zugegriffen am 15.05.2007.

[29] *VMWare End User License Agreement*. `http://www.vmware.com/download/eula/esx_server.html`, zugegriffen am 15.05.2007.

[30] BARHAM, PAUL T., BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIMOTHY L. HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT und ANDREW WARFIELD: *Xen and the Art of Virtualization*. In: *SOSP*, Seiten 164–177, 2003.

[31] *QEMU: Open Source Processor Emulator*. `http://fabrice.bellard.free.fr/qemu/`, zugegriffen am 15.05.2007.

[32] *OpenVZ*. `http://openvz.org/`, zugegriffen am 15.05.2007.

[33] *User-Mode Linux*. `http://user-mode-linux.sourceforge.net/`, zugegriffen am 15.05.2007.

[34] *Linux VServer*. `http://linux-vserver.org/`, zugegriffen am 15.05.2007.

[35] *KVM: Kernel-based Virtual Machine*. `http://kvm.qumranet.com/`, zugegriffen am 15.05.2007.

[36] *UnionFS: A Stackable Unification File System*. `http://unionfs.filesystems.org/`, zugegriffen am 15.05.2007.

[37] *Debian GNU/Linux*. `http://www.debian.org/`, zugegriffen am 15.05.2007.

[38] SQUYRES, JEFFREY M. und ANDREW LUMSDAINE: *A Component Architecture for LAM/MPI*. In: *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Nummer 2840 in *Lecture Notes in Computer Science*, Seiten 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[39] GABRIEL, EDGAR, GRAHAM E. FAGG, and GEORGE BOSILCA: *Open MPI: Goals, concept, and design of a next generation MPI implementation*. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[40] DEERING, S. and R. HINDEN: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard), December 1998.

[41] KAUHAUS, CHRISTIAN, ADRIAN KNOTH, THOMAS PEISELT, and DIETMAR FEY: *Efficient message passing on multi-clusters: An IPv6 extension to Open MPI*. In *Proceedings of KiCC'07, Chemnitzer Informatik Berichte*, February 2007.

[42] *Intel MPI benchmarks*. `http://www3.intel.com/cd/software/products/asmo-na/eng/307696.htm#mpibenchmarks`, zugegriffen am 15.05.2007.

[43] WEICKER, REINHOLD P.: *Dhrystone: a synthetic systems programming benchmark*. Commun. ACM, 27(10):1013–1030, 1984.

[44] *Standard performance evaluation corporation (SPEC)*. `http://www.spec.org/`, zugegriffen am 15.05.2007.

[45] LOVE, ROBERT: *Linux Kernel Development*. Sams Publishing, Indianapolis, IN, USA, 2004.

[46] BENVENUTI, CHRISTIAN: *Understanding Linux Network Internals*. O'Reilly, Sebastopol, CA, USA, 2005.

A CD-Inhalt

Dateiname	Beschreibung
Benutzerseite	
vte.rb	Einstiegspunkt in die Bibliothek: require `"vte"`
Rakefile	Startet die Setup-Phase („rake prepare“) und die Selbsttests („rake test“)
bin/	Hilfsprogramme zum Starten und Stoppen von VMs
Hauptklassen	
cluster.rb	Cluster
virtual_machine.rb	VirtualMachine
switch.rb	Switch
router.rb	Router
Hilfsklassen	
command.rb	Ergebnisse eines Shell-Kommandos innerhalb einer VM
host.rb	Schnittstelle zum Wirtsrechner
vmconfig.rb	Kapselt VM-Konfigurationsdaten
vpn.rb	Kontrolliert den tinc VPN-Daemon
Werkzeuge	
popen4.rb	popen-Implementierung, die stdout, stderr und pid zurückliefert
work_queue.rb	Parallele Ausführung einer Aufgabe mit einer fixen Anzahl von Threads
netaddr_ext.rb	Zusätzliche Hilfsmethoden für NetAddr::CIDR
cattr.rb	Hilfsmethoden zur Definition von Klassen-Attributen
Dokumentation	
README	Installationsanleitung
doc/	API-Referenz
Installation	
install/	Patches, die für die Installation benötigt werden
sudo/	sudo-Hilfsprogramme
Selbsttest	
test/	Selbsttests für VTE
sge/	Hilfsbibliothek für die Selbsttests (aus der Arbeitsgruppe Multicluster)

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, 25. Mai 2007 Wolfgang Schnerring