Studienarbeit

Simulation mit VNUML

Michael Monreal
30. August 2007

Universität Koblenz-Landau
Campus Koblenz

Institut für Informatik
AG Rechnernetze und Rechnerarchitektur

Betreuer:
Prof. Dr. Christoph Steigner
Dipl. Inf. Harald Dickel

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Diese Studienarbeit soll als Einführung in das Thema *Netzwerksimulation* dienen und unter anderem auch als Einstiegs-Referenz für zukünftige Besucher der Rechnernetze-Veranstaltungen an der Universität Koblenz nutzbar sein. Als Quellen dienten deshalb hauptsächlich verschiedene frühere Seminar- und Studienarbeiten unserer Universität sowie die offiziell zugänglichen Informationen des VNUML-Projekts.

1.2 Aufbau

Die Ausarbeitung beginnt mit den Grundlagen zu UML und VNUML und beschreibt dann die Installation, Konfiguration und das Arbeiten mit dem Netzwerksimulator sowie oft genutzter Tools. Im Anschluss daran werden konkrete Anwendungsfelder vorgestellt: der simulierte Einsatz des Paketfilter *iptables* zur Realisierung von Firewalls und NAT, verschiedene Netzwerkdienste und zuguterletzt simuliertes Routing mit der *quagga*-Suite.

Der Leser ist dabei angehalten, die beschriebenen Abläufe jeweils auch selbst praktisch auszuprobieren. Natürlich kann bei der Fülle der Themen nie sehr tief in die Materie eingestiegen werden, einige Grundlagen und Hintergrundwissen zu den Themen sollten also bereits durch die zugehörigen Vorlesungen vorhanden sein. Durch eigenes Ausprobieren kann das Verständnis der Materie dann noch vertieft werden.

Begleitend zu dieser Ausarbeitung entstand ein passendes Dateisystem, in dem die benötigte Software bereits installiert ist. Dort sind auch die folgenden beschriebenen Beispiel-Szenarien vorkonfiguriert zum Ausprobieren hinterlegt. Genauere Informationen zu diesem Dateisystem finden sich im Anhang C.

Desweiteren wurden auch der von der Universität Koblenz zur Verfügung gestellte Offline-Installer, sowie das VMware-Image mit diesem neuen Dateisystem und den Beispielszenarien ausgestattet. Mehr zu den verschiedenen Nutzungsmöglichkeiten im Kapitel *Installation*.

1.3 Konventionen

Kommandos für die Linux-Shell sind im folgenden jeweils mit einem "$" oder "#" gekennzeichnet. Diese Zeichen gehören nicht zum Kommando selbst, sondern sollen Aufschluss über die benötigten Berechtigungen geben:

- mit "$"-Zeichen: Kommando kann mit User-Rechten ausgeführt werden
- mit "#"-Zeichen: Kommando erfordert Superuser-Rechte (Root-Rechte)

Um Root-Rechte zu erhalten, nutzt man vor der Eingabe der beschriebenen Kommandos den Befehle *su* und bestätigt mit dem Root-Passwort. Macht dies (z.B. bei Ubuntu) Probleme, so sollte *sudo -s* Abhilfe schaffen. Ein paar weitere Informationen zur Linux-Shell finden sich zudem im Anhang B. Die dort beschriebenen Kommandos sollten zur Nutzung von VNUML ausreichen. Tiefergehende Linux-Kenntnisse sind nicht erforderlich.

Backslash-Zeichen am Ende einer Kommandozeile sind ebenfalls zu missachten, die nächste Zeile soll in dem Fall noch Teil der vorangegangenen Zeile sein.

2 Grundlagen und Begriffe

2.1 Was ist... UML?

Hinter dem Akronym *UML* versteckt sich der Name *User-mode Linux*[1]. User-mode Linux ist ein Virtualisierungswerkzeug, welches es erlaubt, ein oder mehrere Linux-Gastsysteme auf einem normalen Linux-System auszuführen. Jedes der Gastsysteme stellt dabei auf dem Wirtsystem nur einen normalen Benutzerprozess dar [Wik07].

User-mode Linux ist bereits seit 1999 in Entwicklung und erstmals als Patch für den Linux-Kernel 2.2.x verfügbar [Keu05]. Seit Version 2.6.x ist UML fester Bestandteil des Linux-Kernels [Wik07] und sollte deshalb heute auf jeder halbwegs aktuellen Linux-Installation nutzbar sein.

Die Anwendungsbereiche von User-mode Linux sind breit gefächert und erstrecken sich vom Bereitstellen von Testumgebungen oder Honeypots über virtuelles Hosting bis hin zu Netzwerksimulationen. Während in den erst genannten Feldern neuere Virtualisierungslösungen wie OpenVZ oder Xen performancebedingt heute die bessere Wahl sind [Mon07], ist UML gerade wegen seiner Einfachheit nach wie vor die erste Wahl für Netzwerksimulationen mit Linux.

Für die Realisierung der Netzwerkfunktionalität stehen bei User-mode Linux verschiedene Möglichkeiten zur Verfügung, am verbreitesten ist jedoch der TUN/TAP-Ansatz. Dieser erlaubt die Kommunikation des Wirtssystems mit den virtuellen Maschinen über die Gerätedateien `/dev/tunX` und `/dev/tapX`. TAP ist dabei für Ethernet-Frames auf Layer 2 und TUN für IP-Pakete auf Layer 3 verantwortlich. Darüber hinaus kommt die Bridging-Funktionalität von Linux zum Einsatz, um virtuell voneinander abgegrenzte Netze zu realisieren.

Nähre Informationen zur Virtualisierungstechnik und der genauen Funktionsweise von User-mode Linux finden sich zum Beispiel in der Arbeit von Tim Keupen [Keu05].

2.2 Was ist... VNUML?

VNUML, kurz für *Virtual Network User Mode Linux*[2], ist ein Framework zur Netzwerksimulation, welches auf User-mode Linux basiert. Mit VNUML ist es möglich, die von User Mode Linux erzeugten virtuellen Maschinen zu Netzen zusammenzufassen. Sieht man von kleineren Einschränkungen bedingt durch die Virtualisierung ab, verhält sich jede VM wie ein beliebiger Knoten in einer realen Netztopologie [Keu05].

Die Arbeit an VNUML begann als Teil des Euro6IX Forschungsprojektes, bei dem IPv6 IX-Szenarien mit Hilfe von Linux und der Quagga Routing-Suite simuliert werden sollten. Nach einiger Zeit der internen Entwicklung an der Technischen Universität Madrid wurde das VNUML-Projekt im Jahr 2004 öffentlich und wird seitdem über das Open-Source-Portal SourceForge[3] weiterentwickelt. Der Fokus ist dabei deutlich breiter geworden: VNUML ist schon lange nicht mehr nur auf das Testen von IPv6-Routing ausgelegt, sondern bietet heute einen für viele Zwecke nutzbaren Netzwerksimulator.

[1] http://user-mode-linux.sf.net/
[2] http://www.dit.upm.es/vnumlwiki/
[3] http://sourceforge.net/projects/vnuml

An der Universität Koblenz wird VNUML seit längerem zur Simulation von Netzwerken eingesetzt und kommt hier besonders in den Rechnernetze-Veranstaltungen zum Zuge.

2.3 Aufbau von VNUML

Das VNUML-System teilt sich in zwei Bestandteile: zum einen wird eine XML-basierte Beschreibungssprache angeboten, mit deren Hilfe sich die Simulationsszenarien definieren lassen. Zum anderen existiert ein passender Parser, welcher die Szenariendefinitionen ausliest und realiert [Lat07]. Die Konfiguration des zugrunde liegenden User-mode Linux wird dem Benutzer dabei quasi komplett abgenommen.

Die Arbeit mit VNUML teilt sich dabei in drei Phasen [FG07]:

1. *Design*: Die erste Phase geschieht noch komplett "auf Papier" oder im Kopf. Hier legt man fest, was simuliert werden soll, wie viele virtuelle Maschinen benötigt werden und wie die Netztopologie aussehen soll.

2. *Implementation* In der zweiten Phase kommt dann die Beschreibungssprache zum Zuge. Hier setzt der Benutzer sein vorhandenes Design in der VNUML-Sprache um.

3. *Ausführung*: Die nun erstellte Szenarienbeschreibung kann nun dem VNUML-Parser übergeben werden. Der Parser prüft zunächst, ob es sich bei der Datei um ein valides XML-Dokument handelt, und ob dieses den Regeln der VNUML-Sprache genüge tut. Ist dies der Fall, beginnt der Parser damit, die Definitionen der virtuellen Netzwerke auszulesen und erstellt die nötigen Interfaces und Bridges. Dann werden die Gastsysteme nach der gegebenen Beschreibung vorbereitet, gebootet und an die virtuellen Netzwerkschnittstellen angeschlossen. Im letzten Schritt werden eventuelle Kommandos auf den laufenden VMs ausgeführt.

Neben dem Parser und der Szenarienbeschreibung sind noch zwei weitere Dinge nötig: ein speziell auf die UML-Architektur angepasster Linux-Kernel und ein Dateisystem-Image für die virtuellen Maschinen. Diese werden in der Beschreibungsdatei den VMs zugeordnet. Dabei können wahlweise für jede VM die gleichen oder unterschiedliche Images genutzt werden.

Das Dateisystem-Image wird während der Ausführungsphase nicht etwa direkt genutzt oder für jede VM kopiert, sondern es werden sogenannte COW-Files (Copy-On-Write) angelegt. Die VMs lesen dabei das Original-Filesystem, schreiben jedoch nur in das COW-File, welches nur die Änderungen zum Original enthält [ELP06].

Anmerkung: Es ist zwar ohne weiteres möglich, mittels COW-Files größere Änderungen (z.B. Software Installationen) zu betreiben, dies sollte allerdings schon allein aus Performancegründen vermieden werden. Anhang D zeigt deshalb einen besseren Weg auf, um dauerhaft größere Änderungen an einem Dateisystem-Image vorzunehmen.

3 Installation

Dieses Kapitel beschreibt die grundlegenden Möglichkeiten bei der Einrichtung eines VNUML-Systems, speziell in der Version 1.7. Bei älteren oder neueren Versionen kann es zu leichten Abweichungen kommen. Es wird auf bereits vorgefertigte Kernel- und Dateisystem-Images zurückgegriffen, detailierte Informationen zur Erstellung eigener Images finden sich in den Aufzeichnungen von Matthias Korn [Kor05] und den Anhängen C und D.

3.1 Debian Pakte

Um VNUML auf einem Debian- oder Ubuntu-System nutzen zu können, werden die folgenden Basispakete benötigt:

```
ncurses-5.5 readline-5 libxml2 expat vlan uml-utilities bridge-utils
libxml-dom-perl libnetwork-ipv4addr-perl libexception-class-perl
libxml-checker-perl libnetaddr-ip-perl libterm-readkey-perl liberror-perl
libnet-pcap-perl libclass-data-inheritable-perl libdevel-stacktrace-perl
libpcap0.7 libxml-regexp-perl libxml-regexp-perl
```

Das VNUML-Projekt selbst liefert die fehlenden Pakete[4]. Um diese über den Paketmanager installieren zu können, muss man nur den folgenden Eintrag in die Datei */etc/apt/sources.list* übernehmen:

```
deb http://jungla.dit.upm.es/~vnuml/debian binary/
```

Dann kann man alles zusammen automatisch installieren lassen:

```
# apt-get update
# apt-get install vnuml
```

Anmerkung: Leider gibt es nach dem Erscheinen der VNUML-Version 1.8 keine Möglichkeit mehr, die Version 1.7 über `apt-get` zu installieren. Deshalb sind die zusätzlichen manuellen Schritte notwendig:

```
$ wget -c http://jungla.dit.upm.es/~vnuml/debian/binary/vnuml_1.7.3-1_all.deb
# dpkg -i vnuml_1.7.3-1_all.deb && rm vnuml_1.7.3-1_all.deb
```

3.2 Offline-Installer

Der Offline-Installer ist eine Entwicklung der Universität Koblenz und bietet sich besonders für Distributionen an, die nicht auf dem Debian-Paketsystem basieren. Die Installation läuft wie folgt ab:

```
$ wget -c http://www.uni-koblenz.de/~vnuml/VNUML-Offline-1.7.2-1.sh
# sh VNUML-Offline-1.7.4-1.sh
```

Anmerkung: Der Offline-Installer ist nicht immer direkt in der neusten VNUML-Version verfügbar. Eine für VNUML 1.7 angepasste Version steht jedoch zur Verfügung.

[4] http://jungla.dit.upm.es/ vnuml/debian/binary/

3.3 Live-DVD

Wer VNUML nicht selbst auf der Festplatte installieren will, kann auf die Live-DVD[5] zurückgreifen. Die Version 2.1 dieser DVD basiert ebenfalls auf der hier zugrundegelegten VNUML-Version 1.7. Nachdem man das ISO-Image gebrannt hat, muss man nur noch im BIOS das Booten von DVD aktivieren und den Rechner mit der DVD im Laufwerk neu starten. Diese Methode hat Performancenachteile, es wird allerdings nichts an der Festplatte verändert.

3.4 VMware Image

Eine Alternative zur Live-DVD bietet das VMware-Image. VMware ist eine Virtualisierungssoftware, welche das Starten anderer Betriebssysteme unter Windows erlaubt. Das vorgefertigte Image enthält ein komplettes Ubuntu Linux-System mit VNUML 1.7, sowie den hier vorgestellten Szenarien. Bei der Installation wird alles so eingerichtet, dass im Folgenden ein einfacher Dateiaustausch mit den virtuellen Maschinen möglich ist. Nähere Informationen dazu finden sich in der Ausarbeitung von Patrick Schütz [Sch07].

3.5 Allgemeine Anpassungen

Egal, welcher der oben genannten Wege gewählt wurde: es verbleiben einige kleine optionale Anpassungen am Host-System, welche nach jedem Neustart erneut vorgenommen werden müssen:

3.5.1 SSH Schlüsselpaar

Für die Verbindung zu den virtuellen Maschinen per SSH ist für den Benutzer auf dem Host (meist Root) ein SSH Schlüsselpaar erforderlich. Um dieses zu generieren, gibt man auf dem Host folgendes ein:

```
# ssh-keygen -t rsa
```

3.5.2 Starten von VMs in X11 Termianls

Sollen die VMs automatisch in X11 Terminalfenster starten, muss sichergestellt sein, dass die benötigten Berechtigungen vorliegen. Einerseits kann man VNUML mit Root-Rechten und dem zusätzlichen Parameter *-u root* starten (ansonsten würde VNUML die Root-Rechte weitgehend ablegen und den Benutzer `vnuml` zur Ausführung nutzen). Die Alternative besteht darin, dem `vnuml`-Benutzer Zugriff auf das X11-Display zu geben:

```
# xhost +local:vnuml
```

3.5.3 Internetanbindung für die VMs

Sollen die VMs später ans Internet angebunden sein, so sind NAT und IP-Forwarding auf dem Hostrechner zu aktivieren:

```
# iptables -t nat -A POSTROUTING -o ath0 -j MASQUERADE
# sh -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'
```

[5] http://downloads.sourceforge.net/vnuml/vnuml-live-2.1.iso

4 Konfiguration

4.1 Die Szenarienbeschreibungsdatei

Die Konfiguration von VNUML geschieht nicht etwa global, sondern jeweils gesondert für jedes einzelne Simulations-Szenario. Szenarien werden jeweils in einer eigenen XML-Datei beschrieben. VNUML stellt extra dafür eine eigene XML-Sprache bereit, welche mit jeder neuen VNUML-Version auch geringfügige Änderungen erfährt. Dies bedeutet, dass Szenarien mit der Sprachversion 1.5 nur mit der dazu passenden VNUML-Version lauffähig sind, nicht jedoch mit einer älteren oder neueren Version. Da die grundlegenden Sprachelemente jedoch gleich bleiben, sind die Unterschiede in der Regel eher geringfügig [Chm06], sodass alte Szenarien schnell an neue VNUML-Sprachversionen angepasst werden können. Eine sehr einfache Szenario-Datei könnte beispielsweise so aussehen:

1 Beispiel der VNUML-Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
<vnuml>
  <global>
    <version>1.7</version>
    <simulation_name>simple</simulation_name>
    <ssh_version>2</ssh_version>
    <ssh_key>~/.ssh/identity.pub</ssh_key>
    <automac/>
    <vm_defaults>
      <filesystem type="cow">/vnuml/filesystems/debian.ext3</filesystem>
      <kernel>/vnuml/kernels/linux-2.6.20.1-uml0</kernel>
    </vm_defaults>
  </global>
  <net name="Net0" mode="uml_switch" />
  <vm name="uml1">
    <if id="1" net="Net0">
      <ipv4>10.0.0.1</ipv4>
    </if>
  </vm>
  <vm name="uml1">
    <if id="1" net="Net0">
      <ipv4>10.0.0.2</ipv4>
    </if>
  </vm>
</vnuml>
```

Mit ein wenig Mühe lassen sich hier bereits die Funktionen der einzelnen Elemente erahnen, eine detailierte Beschreibung folgt jedoch im Anschluss.

4.2 Die XML-Sprache

Wie in jeder XML-Sprache bietet auch die VNUML-Sprache eine Reihe von Elementen (Tags), welche im Baukastenprinzip miteinander verknüpft werden

können, um die Szenarienbeschreibungsdatei zu bilden. Einige dieser Elemente sind zwingend erforderlich, andere hingegen optional [MÖ5]. Hier sollen nun die wichtigsten Tags zur Erstellung eigener Szenarien *in der VNUML-Version 1.7* genauer vorgestellt werden. Es existieren weitere Tags, diese werden allerdings nicht zwangsweise benötigt und finden dementsprechend selten Verwendung. Die komplette Referenz findet sich im VNUML-Wiki[6].

4.3 Metadaten

Ein VNUML-Szenario beginnt jeweils mit dem selben Metadaten-Header:

```
<?xml version="1.0"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
```

Angegeben ist hier das DTD-File zur Validierung der XML-Sprache. Achtung: Der Pfad ist dabei abhängig von der jeweiligen VNUML-Installation, sollte man also VNUML-Szenarien von Dritten nutzen, sollte man stets diesen Pfad prfüfen!

4.4 Kommentare

Wie für XML üblich, werden Kommentare mit den Sequenzen <!- - und - -> eingeschlossen. Kommentare können an jeder Stelle der Datei stehen und werden vom XML-Parser einfach ignoriert.

4.5 <vnuml> ... </vnuml>

Das `<vnuml>`-Tag folgt dem XML-Header und schließt den Rest der Datei ein. Gültige Unterelemente sind `<global>, <net>`, `<vm>` und `<host>`, sowie deren eigene Unterelemente. Außerhalb von `<vnuml>` und `</vnuml>` darf es keine weiteren Elemente geben.

4.6 <global> ... </global>

Dieses Tag beinhaltet eine Reihe von weiteren Tags zur Definition allgemeingültiger Variablen für die betreffende Simulation. Gültige Unterelemente sind `<version>`, `<simulation_name>`, `<ssh_version>`, `<ssh_key>`, `<automac>`, `<netconfig>`, `<vm_mgmt>`, `<tun_device>` und `<vm_defaults>`.

4.6.1 <version> ... </version>

Gibt an, für welche VNUML-Version (hier immer 1.7) das Szenario entworfen wurde. Achtung: Andere VNUML-Versionen lehnen das Szenario ab, selbst wenn es von der XML-Sprache her kompatibel wäre. Beispiel:

```
<version>1.7</version>
```

[6] http://www.dit.upm.es/vnumlwiki/index.php/Reference_1.7

4.6.2 <simulation_name> ... </simulation_name>

Definiert den Namen der Simulation. Dies ist auch der Name, unter dem VNUML Temporärdateien für die VMs anlegt. Achtung: Es ist zu einem Zeitpunkt jeweils nur eine Instanz eines Szenarios mit demselben Namen ausführbar.

4.6.3 <ssh_version> ... </ssh_version>

Beinhaltet Informationen zur verwendeten SSH-Version (1 oder 2). Achtung: Je nach verwendetem Dateisystem kann es hier zu Problemen kommen, für das mitgelieferte Dateisystem empfehle ich deshalb immer Version 2. Beispiel:

```
<ssh_version>2</ssh_version>
```

4.6.4 <ssh_key> ... </ssh_key>

Gibt den Pfad an, wo die zugehörigen SSH-Schlüssel abgelegt sind.

```
<ssh_key>~/.ssh/identity.pub</ssh_key>
```

4.7 <net>

Mit dem <net>-Tag werden einzelne Netze definiert, an die man später eine oder mehrere VMs "anschließen" kann. Neben dem im Folgenden genutzten Modus *uml_switch* ist auch *virtual_bridge* verfügbar.

```
<net name="Net1" mode="uml_switch" />
```

4.8 <vm> ... </vm>

Dieses Tag kann mehrfach vorkommen und beinhaltet jeweils die Definition einer einzelnen VM. Gültige Unterelemente sind `<filesystem>`, `<mem>`, `<kernel>`, `<shell>`, `<basedir>`, `<mng_if>`, `<console>`, `<xterm>`, `<if>`, `<route>`, `<forwarding>`, `<user>`, `<filetree>` und `<exec>`.

4.8.1 <filesystem> ... </filesystem>

Gibt den Pfad der Dateisystemvorlage an, welche für die VM genutzt werden soll. Kann optional auch in `<global>` für alle VMs gleich definiert werden.

4.8.2 <mem> ... </mem>

Gibt an, wie viel Megabyte RAM der VM zur Verfügung stehen soll. Kann optional auch in `<global>` für alle VMs gleich definiert werden.

4.8.3 <kernel> ... </kernel>

Gibt den Pfad des Kernels an, der für die VM genutzt werden soll. Kann optional auch in `<global>` für alle VMs gleich definiert werden.

4.8.4 <console id="0">xterm</console>

Startet jede VM automatisch in einem X-Term. Siehe dazu *Starten von VMs in X11 Termianls* im Kapitel *Installation*.

4.8.5 <if> ... </if>

Das `<if>`-Tag definiert ein Netzwerkinterface der VM, pro gewünschtem Interface legt man also ein solches Tag an. Jedes Interface hat eine eindeutige *id* und ist über *net* an eines der zuvor definierten Netze gebunden. Innerhalb dieses Tags kann man auch direkt IP-Adressen festlegen. Beispiel:

```
<if id="1" net="Net1"><ipv4>10.0.0.1</ipv4></if>
```

4.8.6 <route> ... </route>

Dieses Tag erlaubt die Einrichtung von statischen Routen. Beispiel:

```
<route type="ipv4" gw="10.0.0.1">10.0.0.2/24</route>
```

4.8.7 <forwarding type="ip" />

Aktiviert IP-Forwarding. Damit wird die VM zum IP-Router.

4.8.8 <filetree>

Erlaubt das Einbinden von Verzeichnissen des Hostdateisystems in die VMs. Dies ist praktisch, um beispielsweise Konfigurationsdateien vom Host auf die VMs zu verteilen. Beispiel:

```
<filetree root="/root/config" when="always">uml1/config</filetree>
```

4.8.9 <exec>

Erlaubt das Ausführen beliebiger Kommandos auf einer oder mehrerer VMs. Dazu wird dem Kommando ein Sequenz-Name zugeordnet. Führt man dann eine bestimmte Sequenz aus, werden alle `<exec>`-Tags dieser Sequenz auf den betreffenden VMs abgearbeitet. Beispiel:

```
<exec seq="mount" type="verbatim">/bin/mount.sh</exec>
```

4.9 <host>

Über das <host>-Tag lässt sich der Hostrechner selbst in die Simulation integrieren. Der Host erhält dann auch eine oder mehrere zusätzliche, simulationsinterne IP-Addressen. Interfaces und Routen können ähnlich wie bei `<vm>` definiert werden. Beispiel:

```
<host>
  <hostif net="Net0"><ipv4>10.0.0.10</ipv4></hostif>
</host>
```

5 Benutzung

Nachdem das Konzept hinter VNUML und der Definitionssprache nun vorgestellt sind, wollen wir jetzt zum praktischen Teil übergehen. Als Beispiel-Szenario dient hier *basic.xml*: zwei VMs (*uml1* und *uml2*), verbunden durch eine dritte (*uml0*), welche als Router dienen soll (vgl. Abb. 1).

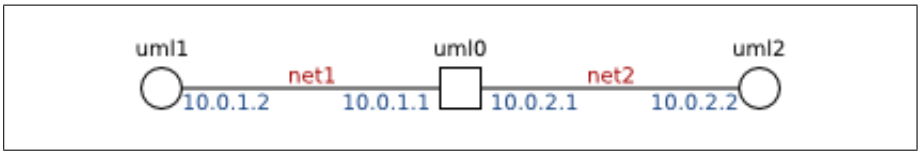

Abbildung 1: Topologie des Szenarios *basic.xml*

Dieses Kapitel soll die Nutzung des VNUML-Parsers nach und nach anhand von Beispielen beschreiben. Eine genauere Beschreibung der einzelnen Parameter ist über `vnumlparser.pl -help` oder das VNUML-Handbuch[7] erhältlich.

5.1 Starten der Simulation

Um das Szenario zu starten, benutzt man das Skript *vnumlparser.pl*:

```
# vnumlparser.pl -t basic.xml -vB
```

Dieser Aufruf legt nun für jede virtuelle Maschine ein COW-File an und richtet die benötigten Netzwerkinterfaces und -Bridges ein. Falls bei diesem Vorgang kein Fehler auftritt, starten im Anschluss die UML Kernels.

Anmerkung: In den neueren Versionen unterstützt VNUML teilweise auch die Ausführung ohne Root-Rechte. In diesem Modus hat man bisweilen allerdings mit einigen Einschränkungen zu kämpfen: es werden keine Management Interfaces unterstützt, man kann also nicht ohne weiteres jede VM ansprechen, ohne das zuvor im simulierten Netz das Routing zwischen der gewünschten VM und dem Host gesichert worden ist. Wegen dieser Einschränkung, und da die Sicherheit des Hostsystems ohnehin eher zu vernachlässigen ist, wird hier einfachheitshalber der Root-User benutzt. Nähere Informationen dazu finden sich im Vergleich der VNUML Versionen 1.5 und 1.6 [Chm06]

Selbst mit dem einfachen *basic.xml*-Szenario lassen sich bereits einige interessante grundlegende Funktionen austesten:

5.2 SSH (Secure Shell)

Secure Shell ermöglicht den Login in entfernte Rechner (oder hier: die VMs) über eine gesicherte Verbindung [Bor05]:

```
$ ssh -2 root@192.168.0.2
```

[7] http://www.dit.upm.es/vnumlwiki/index.php/Usermanual_1.7

Anmerkung: An dieser Stelle ist die IP des Management-Interfaces (192.168.0.2) für *uml0* zu wählen. Über die interne IP (10.0.1.1) wäre *uml0* nicht zu erreichen, da der Hostrechner selbst nicht Teil des simulierten Netzes ist. Um die anderen VMs zu erreichen, muss man also auch deren Management-Interfaces nutzen, oder aber aus der bereits verbundenen VM eine weitere SSH-Verbindung aufbauen.

Falls an dieser Stelle Probleme auftreten, sollte man sich vergewissern, dass man auf dem Host-Rechner ein SSH-Schlüsselpaar generiert hat (vgl. Kapitel *Installation*).

5.3 Interface Konfiguration

Um auf einer VM die vorhandenen Interfaces und die daran gebundenden Werte (wie IP- oder MAC-Adressen) zu sehen, gibt es den Befehl `ifconfig`:

```
uml1:~# ifconfig
```

5.4 Ping

Ein Ping besteht aus den ICMP Nachrichten *Echo Request* (als Anfrage) und *Echo Reply* (als Antwort):

```
uml1:~# ping 10.0.1.1
```

Ein Ping von *uml1* oder *uml2* nach *uml0* funktioniert jeweils, ein Ping von *uml1* nach *uml2* oder umgekehrt jedoch nicht. Dies liegt daran, dass zwischen den beiden VMs keine direkte Verbindung besteht und somit das Zielnetz nicht bekannt ist. Die Lösung dafür ist IP-Forwarding und das Anlegen statischer Routen:

5.5 Statisches Routing und IP-Forwarding

```
uml0:~# echo 1 > /proc/sys/net/ipv4/ip_forward
uml1:~# route add default gw 10.0.1.1
uml2:~# route add default gw 10.0.2.1
```

Beide VMs schicken nun alle Pakete zunächst an das Gateway *uml0*, welches zu beiden Netzen Anschluss hat und dank IP-Forwarding die Pakete weiterleiten kann. Um Informationen über die aktuelle Routing-Konfiguration auf einem System zu erhalten, kann man auf die Befehle `route` oder `netstat -r` zurückgreifen. Die Ausgabe ist jeweils die gleiche.

In größeren Szenarien könnte eine `route`-Ausgabe so aussehen:

```
Kernel IP routing table
Destination     Gateway   Genmask         Flags  Metric  Ref  Use  Iface
192.168.0.1     *         255.255.255.0   UH     0       0    0    eth0
10.0.0.0        *         0.0.0.0         U      0       0    0    eth1
default         10.0.0.3  0.0.0.0         UG     0       0    0    eth1
```

Hier kann der Benutzer sehen, welche Zielrechner bzw -Netze (`Destination`) nicht direkt, sondern nur über ein bestimmtes `Gateway` erreichbar sind. Im Beispiel ist ein Default-Gateway eingetragen, d.h. alle Pakete, die nicht an den

Host 192.168.1.144 oder das Netz 10.0.0.0 gerichtet sind, laufen über das Gateway mit der IP-Adresse 10.0.0.3. Interessant ist auch die Spalte `Flags`, und zwar besonders die Attributwerte U, H und G:

U	Die Route ist aktiv und nutzbar
H	Das Ziel ist kein Netz, sondern ein Hostrechner
G	Es wird der angegebene Gateway benutzt

VNUML bietet zur Einrichtung von statischen Routen und IP-Forwarding die bereits beschriebenen Tags <*route*> und <*forwarding*>, welche in der XML-Datei zur Automatisierung des Prozesses genutzt werden können.

Im Beispielszenario ist eine Zweite Möglichkeit implementiert; die oben beschriebenen Kommandos können wahlweise auch über die <*exec*>-Sequenz *routing* ausgeführt werden:

```
# vnumlparser.pl -x routing@basic.xml -v
```

In größeren Szenarien macht man der Übersichtlichkeit wegen gerne Gebrauch von <*route*> und <*forwarding*> Tag, <*exec*>-Sequenzen werden hier dann aber oft zusätzlich für andere Aufgaben verwendet.

5.6 Traceroute oder Tracepath

Zeigt an, über welchen Pfad die Pakete im Netzwerk von A nach B laufen:

```
uml1:~# traceroute 10.0.2.2
```

Dies geschieht durch sukzessives Inkrementieren des vorherigen TTL Wertes von 1 an, jeweils dann, wenn einer der passierten Hosts den TTL Wert auf 0 dekrementiert und folglich mit einer "Time Exceeded" ICMP-Nachricht antwortet.

5.7 TCP Dump und Netcat

Will man den TCP-Verkehr auf einem bestimmten Knotenpunkt überwachen, so bietet sich `tcpdump` an. Um beispielsweise auf *uml0* einen Ping von *uml1* nach *uml2* sichtbar zu machen:

```
uml0:~# tcpdump -i eth1
```

Alternativ zum Ping bietet sich `netcat` an. Mittels `netcat` kann man Daten über TCP- oder UDP-Verbindungen senden und empfangen. Will man beispielsweise Nachrichten von *uml2* nach *uml1* schicken:

```
uml1:~# netcat -l -p 20200
uml2:~# netcat 10.0.1.2 20200
```

Nachdem die Verbindung aufgebaut ist, sollten die Nachrichten von *uml2* bei *uml1* erscheinen. Gleichzeitig sollten die passierenden Pakete auf *uml0* angezeigt werden.

5.8 ARP

Über ARP (Address Resolution Protocol) kann man die Hardware-Adresse (MAC-Adresse) des Interfaces hinter einer bekannten IP-Adresse herausfinden:

```
uml0:~# arp 10.0.2.2
Address         HWtype  HWaddress           Flags Mask  Iface
10.0.2.2        ether   FE:FD:00:00:03:01   C           eth2
```

In diesem Fall kann *uml0* über das Interface eth2 die IP 10.0.2.2 erreichen, hinter der sich die MAC-Adresse FE:FD:00:00:03:01 verbirgt.

5.9 Beenden der Simulation

Um ein Simulations-Szenario zu beenden, dienen die *-d* und *-P* Optionen:

```
# vnumlparser.pl -d basic.xml -vB
```

In diesem Fall werden die gestarteten VMs heruntergefahren und virtuelle Netzwerk- und Bridge-Interfaces gelöscht. Alle temporären Dateien und COW-Files bleiben intakt: das Szenario startet beim nächsten Mal also im gleichen Zustand, in dem es verlassen wurde.

```
# vnumlparser.pl -P basic.xml -vB
```

Hier hingegen werden zusätzlich alle Temporärdateien gelöscht, beim erneuten Start des Szenarios beginnt man also wieder von Null.

6 Paketfilter: Firewalling und NAT

Natürlich sind nicht alle Netze so einfach gestrickt wie das erste Beispiel-Szenario. Oft ist nötig, die ein- und ausgehenden Pakete genau zu betrachten, um gegebenenfalls die Weiterleitung unterbinden oder ändern zu können. Dies geschieht unter Linux mit Hilfe des eingebauten Paketfilters Netfilter bzw `iptables`.

Dabei ist Netfilter das eigentliche Kernel-Framework zur Filterung und Manipulation von Netzwerkpaketen, `iptables` hingegen ist der Name des Kommandozeilentools zur Konfiguration von Netfilter [Hor06]. In der Praxis werden die beiden Namen oft synonym verwendet. Netfilter gruppiert seine Verarbeitungsregeln in drei Tabellen ("tables"):

- *filter*: Filterung von Paketen
- *mangle*: Manipulation von Paketen
- *nat*: Network Address Translation

In den Tabellen befinden sich Ketten ("chains"), welche Sequenzen von Verarbeitungsregeln darstellen. Die Ketten bestimmen, wann ein Paket geprüft wird. Es werden alle Verarbeitungsregeln innerhalb einer Kette sequentiell abgearbeitet, bis die Kette durchlaufen ist oder ein Treffer vorliegt [Hor06]. Es gibt fünf Ketten, wobei nicht jede Kette für jede Tabelle gültig ist (vgl. Abb. 2):

- *PREROUTING*: alle eingehende Pakete (nat, mangle)
- *INPUT*: Pakete, die an lokale Prozesse gehen (filter, managle)
- *FORWARD*: Pakete, die weitergeleitet werden sollen (filter, managle)
- *OUTPUT*: Pakete, die von lokalen Prozessen kommen (filter, nat, mangle)
- *POSTROUTING*: alle ausgehenden Pakete (nat, mangle)

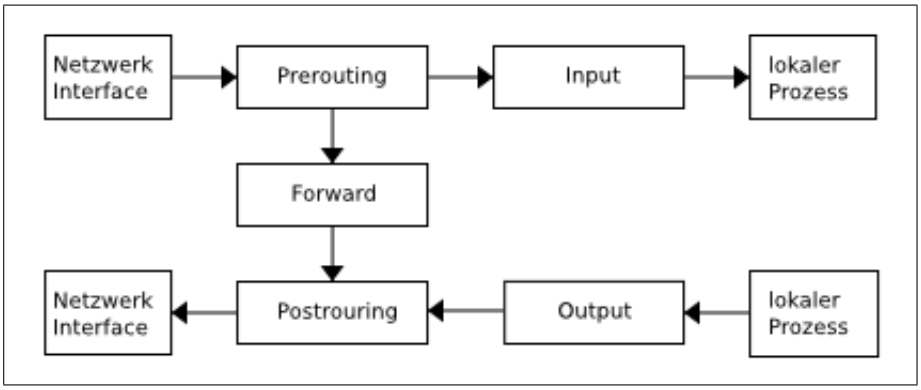

Abbildung 2: IP-Tables Übersicht nach [Dic05]

Verarbeitungsregeln bestehen aus Mustern ("matches") und Zielen ("targets"). Muster bestimmen, auf welche Pakete eine Regel angewendet wird. Mögliche Muster sind demzufolge eine bestimmte IP-, Port- oder MAC-Adresse, ein

bestimmtes Protokoll oder ein Interface. Ziele bestimmen hingegen, was mit einem Paket geschehen soll. Genauere Informationen zu den einzelnen Tabellen, Ketten, Mustern und Zielen bietet Tassilo Horn [Hor06]. Einige oft gebrauchte `iptables`-Kommandos:

Kommando	**Wirkung**
iptables –list –table filter	Filter Tabelle zeigen
iptables –flush –table nat	NAT Tabelle löschen
man iptables	Handbuch zu iptables anzeigen

Das Beispiel-Szenario *netfilter.xml* (vgl. Abb. 3) erlaubt einfaches Experimentieren mit `iptables`. Zum Einsatz kommt sowohl die Funktionalität als Firewall, als auch für NAT bzw PAT.

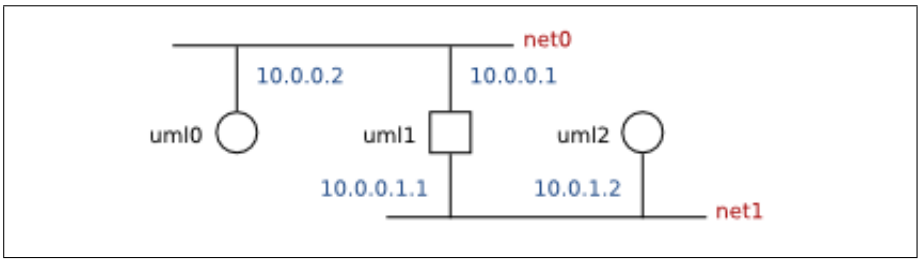

Abbildung 3: Topologie des Szenarios *netfilter.xml*

6.1 Firewall

Eine *Firewall* hat die Aufgabe, Pakete, die den Rechner passieren, zu kontrollieren und gegebenenfalls abzulehnen oder zumindest zu protokollieren. Die Firewall-Funktionalität von Netfilter versteckt sich hinter der Tabelle *filter*. Diese Tabelle erlaubt den Zugriff auf Pakete über die Ketten INPUT, FORWARD und OUTPUT.

Beispiel für das *netfilter.xml*-Szenario: *uml0* soll keine direkten Verbindungen (z.B. Pings) mehr zu *uml1* aufbauen können, wohl aber zu *uml2*.

```
uml1:~# iptables -t filter -A INPUT -d 10.0.1.1 -j DROP
```

6.2 NAT (Network Adress Translation)

Network Adress Translation ist ein Verfahren, um die Quell- oder Zieladresse eines Pakets zu ändern. Je nachdem, ob sich NAT auf die Quelle oder das Ziel bezieht, spricht man auch von Source NAT (SNAT) oder Destination NAT (DNAT). Eine Sonderform von SNAT ist MASQUERADE (Anpassung der Source-Adresse an eine dynamische IP). NAT arbeitet bei Netfilter mit den Ketten PREROUTING, OUTPUT und POSTROUTING.

Bei den NAT-Adressen unterscheidet man nach Cisco zwei Kriterien [Dic05]. Die erste Unterscheidung ist die nach dem Standort des Gerätes bzw Interfaces, in *Inside* oder *Outside*:

- *Inside*: gehört zu einem Interface, das sich innen befindet
- *Outside*: gehört zu einem Interface, das sich außen befindet

Die zweite Unterscheidung bezieht sich hingegen auf den Gültigkeitsbereich der Adresse: *Local* oder *Global*:

- *Local*: gültig im lokalen Netz
- *Global*: gültig außerhalb des lokalen Netzes

Mit der Hilfe von NAT ist es nun möglich, beispielsweise Adressen aus dem Outside-Bereich auf Adressen des Inside-Bereichs zu mappen, sodass diese auch von innen erreichbar sind. Mit NAT wird auch seit Jahren der Knappheit der IPv4-Adressen entgegengewirkt: indem man Netze mittels eines NAT-Routers vom Internet abschottet, verbraucht man weniger global gültige Adressen, da in den verschiedenen Inside-Bereichen auch dieselben Adressbereiche genutzt werden können [Bre07].

Beispiel für das *netfilter.xml*-Szenario: *uml2* soll für *uml0* unter der IP 10.0.2.1 erreichbar sein.

```
uml1:~# iptables -t nat -A PREROUTING -i eth1 -d 10.0.2.1 \
                 -j DNAT --to 10.0.2.2
```

6.3 PAT (Port Adress Translation)

Bei Port Adress Translation handelt es sich um einen Sonderfall von NAT, bei dem die Portadressen von Paketen geändert werden. PAT erlaubt es beispielsweise, hinter einer einzigen öffentlichen IP mehrere physische Server zu betreiben, auf denen jeweils andere Dienste laufen. Der öffentliche Rechner kann Anfragen auf einen bestimmten Port dann gezielt weiterleiten. Von außen sieht es dann so aus, als ob dieser Rechner selbst die Anfragen beantwortet, in Wahrheit stehen jedoch andere Server dahinter.

Beispiel für das *netfilter.xml*-Szenario: Alle TCP-Pakete, die *uml1* Port 80 empfängt, sollen an Port 8080 auf *uml2* weitergeleitet werden.

```
uml1:~# iptables -t nat -A PREROUTING -p tcp -dport 80 \
                 -j DNAT --to 10.0.2.2:8080
```

Komplexeres Paketfilter-Szenario: Ein paar komplexere Funktionen von `iptables` kann man sich beispielsweise am Szenario *netfilter2.xml*, basierend auf der Arbeit von Tassilo Horn citeHor06 verdeutlichen. Hier wird VNUML zur Simulation eines typischen WG-Netzwerks genutzt. Der Hostrechner ist dabei als Gateway zum Internet in das Szenario integriert, das eigentliche lokale Netz wird von vier virtuellen Maschinen nachgebildet (vgl. Abb. 4 auf S.20).

Der Hostrechner hat eine Verbindung zum Internet und soll diese dem VNUML-Netz zugänglich machen:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
# route add -net 10.0.0.0 netmask 255.255.255.0 gw 10.0.1.1
```

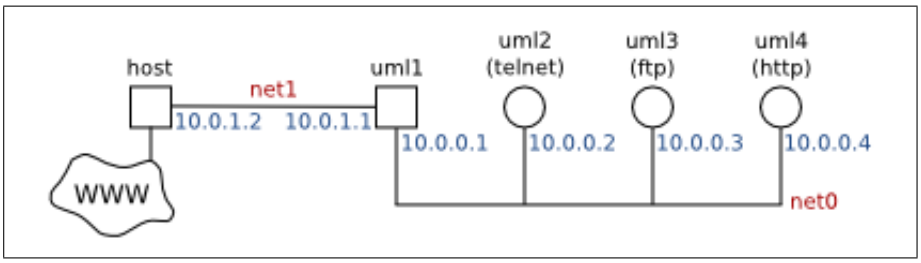

Abbildung 4: Topologie des Szenarios *netfilter2.xml*

Die VM *uml1* soll nun als Router für den Rest des simulierten Netzes dienen. Es müssen also IP-Forwarding und NAT aktiviert werden:

```
uml1:~# echo 1 > /proc/sys/net/ipv4/ip_forward
uml1:~# route add default gw 10.0.1.2
uml1:~# iptables -t nat -A POSTROUTING -j SNAT --to-source 10.0.1.1
```

Die anderen VMs bieten jeweils einen Netzwerkdienst (Telnet, FTP oder HTTP) an. Telnet-Anfragen von außen sollen von *uml1* an *uml2* weitergeleitet werden:

```
uml1:~# iptables -t nat -A PREROUTING -p tcp --dport 23 \
                 -j DNAT --to-destination 10.0.0.2
```

FTP-Anfragen von außen sollen von *uml1* an *uml3* weitergeleitet werden:

```
uml1:~# iptables -t nat -A PREROUTING -p tcp --dport 21 \
                 -j DNAT --to-destination 10.0.0.3
```

HTTP-Anfragen auf Port 80 sollen an den Port 8080 von *uml4* umgeleitet werden:

```
uml1:~# iptables -t nat -A PREROUTING -p tcp --dport 80 \
                 -j DNAT --to-destination 10.0.0.4:8080
```

Alle direkten Anfragen auf *uml1*, außer Verbindungen zu SSH, sollen von der Firewall geblockt und protokolliert werden:

```
uml1:~# iptables -t filter -A INPUT -p tcp -m tcp --dport ! 22 \
                 -j LOG --log-prefix "NOT SSH: "
uml1:~# iptables -t filter -A INPUT -p tcp -m tcp --dport ! 22 \
                 -j REJECT --reject-with icmp-host-prohibited
uml1:~# iptables -t filter -P INPUT ACCEPT
```

Im Beispielszenario kann die wünschenswerte Endkonfiguration auch durch das Aktivieren der Sequenz *netfilter* erzielt werden.

7 Netzwerkdienste: Server und Daemons

In diesem Kapitel werden einige grundlegende Netzwerkdienste vorgestellt: die Domain Name Services (DNS), das Dynamic Host Configuration Protocol (DHCP), die Network Information Services (NIS), sowie spezielle Services zum Datentransfer (HTTP, FTP) und Datenfreigabe (NFS, SMB). Den Abschluss macht die Vorstellung von SNMP, welches das Überwachen und Steuern größerer Netze erlaubt.

Für jeden der folgenden Netzwerkdienste liegt jeweils ein kleines Beispielszenario vor. Diese Szenarien enthalten zur besseren Verständlichkeit jeweils nur zwei virtuelle Maschinen, eine vorkonfiguriert als Server, eine als Client (vgl. Abb. 5). Hier bietet es sich jeweils an, die einzelnen Protokolle mittels Tools wie `tcpdump` oder `wireshark` genauer unter die Lupe zu nehmen.

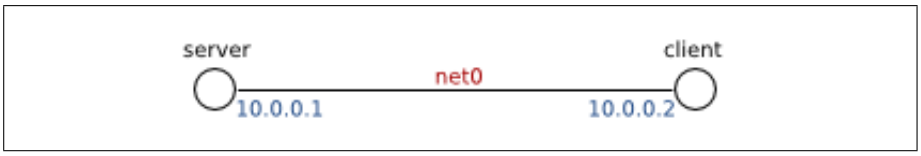

Abbildung 5: Topologie der Service-Szenarien

Ausgehend von den mitgelieferten Konfigurationsdateien sollte es dann aber auch nicht allzu schwer sein, selbst komplexere Szenarien zu entwerfen, in denen mehrere VMs die Dienste nutzen können.

7.1 DNS (Domain Name Services)

Die *Domain Name Services* bilden einen Dienst, der IP-Adressen in einen von Menschen besser handhabbaren Domänennamen übersetzen. Ein DNS-Server ist dabei Teil einer Hierarchie und verwaltet eine Datenbank, in der Namen einer IP-Adresse eindeutig zugeordnet sind. Will ein Benutzer einen anderen Rechner über dessen Domänennamen erreichen, so stellt der Client erst eine Anfrage an seinen DNS-Server. Dieser kennt hoffentlich den gewünschten Namen und liefert dem Client dessen IP-Adresse zurück. Alternativ kann es auch vorkommen, dass der Server die Anfrage an einen anderen DNS-Server weiter reicht.

Die Kommunikation zwischen Client und Server geschieht in der Regel über UDP Port 53, für größere Nachrichten wird allerdings TCP genutzt [Lin06]. Um Informationen vom DNS-Server abzufragen, kommen die Tools `dig` und `nslookup` zum Einsatz.

Beispiel für das *dns.xml*-Szenario: Nach dem Start des Szenarios versucht man von *client* aus, den Host *test.vnuml* zu erreichen:

```
client:~# ping test.vnuml
```

Die Anfrage scheitert, da der Name *test.vnuml* nicht bekannt ist. Nun kopiert man die Konfigurationsdateien und startet den DNS-Server (siehe Exec-Sequenz *server*). Auf *client* trägt man dann dessen IP in die */etc/resolf.conf* ein (siehe Exec-Sequenz *client*). Wiederholt man dann die Anfrage, kann *client* den Namen

mit Hilfe des DNS-Servers auflösen und *test.vnuml* erreichen. Die Konfiguration des Servers wird unter */etc/bind/* abgelegt und kann dort angepasst werden.

Eine komplexere DNS-Struktur mit mehreren Sites und sieben DNS-Servern steht im Szenario *ppss06.xml* bereit. Hintergrundinformationen dazu finden sich in der Dokumentation aus dem Projektpraktikum [ELP06].

7.2 DHCP (Dynamic Host Configuration Protocol)

Das *Dynamic Host Configuration Protocol* hat die Aufgabe, Rechner, welche an ein Netzwerk angeschlossen werden, automatisch mit IP-Adresse und Subnetz-Maske zu versorgen. Da die vorhandenen Adressen vom DHCP-Server verwaltet werden und dieser jede Adresse nur einmal vergibt, können problematische Adressüberschneidungen ausgeschlossen werden. Desweiteren können die Clients über zusätzlichen Parameter beispielsweise mit den Adressen von Gateways und DNS-Servern versorgt werden.

Beispiel für das *dhcp.xml*-Szenario: Nach dem Start des Szenarios und der Exec-Sequenz *server* ist *client* zwar über das Management-Interface von außen erreichbar, hat jedoch noch keine simulationsinterne IP (siehe XML-Datei). Der folgende Aufruf bewirkt die Anfrage beim DHCP-Server:

```
client:~# dhcpcd eth1
```

Die VM *client* sollte nun die Daten vom DHCP-Server übermittelt bekommen haben. Um zu testen, ob tatsächlich eine IP zugewiesen wurde, kann man sich die Ausgabe von `ifconfig` ansehen.

Die Einstellungen wurden so gewählt, dass die Lease-Zeit nur eine Minute lang ist. Wenn man also auf einem der beiden Rechner den Netzwerkverkehr mitlauscht, wird man nach kurzer Zeit die Aushandlung der Verlängerung der Lease-Zeit mitverfolgen können.

7.3 NIS (Network Information Services)

Die *Network Information Services* sind historisch bedingt auch bekannt unter dem Namen *Yellow Pages*. NIS ermöglicht es, einheitliche Benutzeraccounts für das gesamte Netzwerk zu realisieren. NIS nutzt dazu Remote Procedure Calls (RPC) zwischen dem Server (`ypserv`) und dem Client (`ypbind`). Die Informationen, welche im Netzwerk von NIS bereit gestellt werden, sind in sogenannten *Maps* gespeichert. Diese enthalten jeweils Schlüssel/Wert-Paare wie beispielsweise Benutzername/Passwort [MO06].

Beispiel für das *nis.xml*-Szenario: Nach dem Start des Szenarios kopiert man die Konfigurationsdateien und startet NIS auf *server* und *client*:

```
# vnumlparser.pl -x server@nis.xml -v
# vnumlparser.pl -x client@nis.xml -v
```

Die Exec-Sequenz für den Server legt zusätzlich einen Benutzer *testuser* an, der nicht auf *client* existiert und generiert eine neue *passwd*-Map. Zum Testen:

```
client:~# login testuser
```

Um eine Reihe von möglichen Benutzeraccounts zu sehen, kann man folgendes Kommando nutzen:

```
client:~# ypcat passwd
```

7.4 Datentransfer

Für den Transfer von Dateien in Netzwerken haben sich besonders zwei Protokolle durchgesetzt: besonders für die Ablage großer Dateien wird nach wie vor oft FTP eingesetzt. Die Übertragung von Webseiten hingegen geschieht normalerweise über HTTP. Mit VNUML ist es leicht möglich, dies auch in einer Testumgebung nachzubilden.

7.4.1 HTTP (Hypertext Transfer Protocol)

Das *Hypertext Transfer Protocol* legt bekanntermaßen die Grundlage für das World Wide Web und ist somit vielleicht der bekannteste aller Netzwerkdienste. Aufgabe von HTTP war klassischerweise das Ausliefern von HTML-Documenten vom Server an den Client. Heute werden jedoch auch andere Inhalte (Bilder, Multimedia) über HTTP übertragen. HTTP setzt auf TCP auf und läuft standartmäßig über den Port 80 [Fis06]. Eine Besonderheit des Protokolls ist seine Zustandlosigkeit, d.h. der Server behält keine Informationen über frühere Anfragen eines Clients, die Verbindung wird mit jeder Anfrage neu aufgebaut. Es gibt eine Vielzahl an HTTP-Servern, wobei der hier genutzte *Apache* nach wie vor einer der Bekanntesten ist.

Beispiel für das *http.xml*-Szenario: Nach dem Start des Szenarios kopiert man die Konfigurationsdateien und startet den Apache-Server auf *server*:

```
# vnumlparser.pl -x server@http.xml -v
```

Dann kann man sich von der VM *client* aus die hinteregte Webseite mit dem Textmode-Browser `lynx` anzeigen lassen:

```
client:~# lynx 10.0.0.1
```

Alternativ dazu kann man auch einen X11-Server starten und einen graphischen Browser wie `dillo` benutzen. Genaue Informationen dazu finden sich in Anhang E.

7.4.2 FTP (File Transfer Protocol)

Das *File Transfer Protocol* dient der Übertragung beliebiger Dateien. FTP basiert ebenfalls auf TCP. Für den Betrieb sind im Wesentlichen zwei verschiedene Modi vorgesehen [Sla07]:

- *Active Mode:* Der Client verbindet von einem zufällig gewählten Port N größer als 1023 zum Port 21 des Servers. Dann lauscht der Client auf N+1 und sendet das Kommando PORT N+1 an den FTP server. Der Server startet dann eine Verbindung von Port 20 zu diesem Port N+1.

- *Passive Mode:* Der Client öffnet zwei Ports N und N+1 größer als 1023. Mit dem ersten Port verbindet der Client zu Port 21 auf dem Server und schickt das Kommando PASV. Der Server öffnet dann einen zufällig gewählten Port P größer als 1023 und sendet dessen Nummer mittels des Kommandos Port P zurück an den Client. Der Client startet dann die Verbindung von seinem Port N+1 zum Port P des Servers.

Die Nutzung der verschiedenen Ports ist zugleich ein großer Kritikpunkt an FTP, da Probleme bei der Nutzung durch Firewalls oder NAT auftreten. Auch das Versenden unverschlüsselter Passwörter ist nicht wünschenswert. Neue Protokolle wie SFTP und TFTP versuchen heute, diese Lücken zu schließen.

Für FTP gibt es eine Vielzahl an Servern. Für den Testbetrieb in einer virtuellen Maschine sollte dieser allerdings ressourcensparend und leicht zu konfigurieren sein. Christian Oellig und Iwan Diel [OD06] empfehlen deshalb PureFTP.

Beispiel für das *ftp.xml*-Szenario: Nach dem Start des Szenarios startet man den FTP-Server auf *server*:

```
# vnumlparser.pl -x server@ftp.xml -v
```

Nutzt man diese vorgefertigte Exec-Sequenz, so wird automatisch ein FTP-User *test* angelegt, dem man im Terminal noch ein Passwort geben muss. Dann kann man sich von der VM *client* aus mit dem folgenden Befehl auf den FTP-Server verbinden:

```
client:~# ftp 10.0.0.1
```

Eine Verbindung im Passive Mode erreicht man hingegen mit:

```
client:~# ftp -p 10.0.0.1
```

Hier bietet es sich an, den Paketaustausch genauer zu überwachen und beispielsweise die Unterschiede zwischen Active und Passive Mode genauer anzusehen.

7.5 Datenfreigabe

Für die Freigabe von Dateien in Netzwerken eigent sich prinzipiell bereits FTP. Für die Freigabe von beispielsweise Benutzerverzeichnissen in lokalen Netzen greift man aber eher auf NFS (in der Unix Welt) oder SMB (bei Windows) zurück. Auch dies kann man mit VNUML ohne Probleme nachbilden.

7.5.1 NFS (Network File System)

In Unix-Systemen gibt es seit jeher die Möglichkeit, entfernte Verzeichnisse und Ressourcen über sogenannte Netzwerkdateisysteme in den lokalen Verzeichnisbaum "einzuhängen" oder im Unix-Jargon "mounten". Dies geschieht im Prinzip analog zum Mounten von CDs oder anderer Wechselmedien.

Ein bekanntes Unix-Netzwerkdateisystem ist das von Sun Microsystems entwickelte *Network File System*. Auch NFS nutzt, ähnlich wie NIS, RPC-Mechanismen. Auf dem NFS-Server kann man Verzeichnisse an bestimmte Benutzer, Gruppen oder auch IPs freigeben, die Clients können diese dann wie jedes andere Dateisystem in ihr eigenes System mounten [MO06].

Beispiel für das *nfs.xml*-Szenario: Nach dem Start des Szenarios kopiert man die Konfigurationsdatei */etc/exports* und startet den NFS-Server auf *server*:

```
# vnumlparser.pl -x server@nfs.xml -v
```

Dann kann man von der VM *client* aus das NFS-Share vom Server mounten:

```
client:~# mount -t nfs 10.0.0.1:/public/nfs /mnt/nfs
client:~# cd /mnt/nfs && ls
```

Alternativ dazu existiert auch eine Exec-Sequenz *client*, welche zunächst einen entsprechenden Eintrag in die */etc/fstab* schreibt. Das Mounten kann dann etwas schneller mittels des folgenden Kommandos geschehen:

```
client:~# mount /mnt/nfs
```

7.5.2 SMB (Server Message Block)

Da in der Windows-Welt Dateisysteme und Laufwerke etwas anders genutzt werden als unter Unix, gibt es hier auch eine andere Möglichkeit, Netzwerkfreigaben zu realisieren. Was sich hinter der "Netzwerkumgebung" in einem Windows-System verbirgt, ist nichts anderes als das *Server Message Block*-Protokoll. Um dies auch unter Linux bzw unter VNUML nutzen zu können, benötigt man lediglich das *Samba*-Paket.

Die Konfiguration von Samba gestaltet sich ähnlich wie bei NFS. Auch hier existiert eine Konfigurationsdatei, in der bestimmte Verzeichnisse auf dem Server unter bestimmten Kriterien einem entfernten Nutzer zugänglich gemacht werden können. Eine Nutzung, wie über die Windows Netzwerkumgebung, ist auf Textmode-Ebene schwerlich möglich. Um jedoch SMB-Server und -Shares im lokalen Netz zu finden, kann das Kommando `smbtree` genutzt werden. Ansonsten können Samba-Shares ähnlich wie NFS-Shares gemountet werden.

Beispiel für das *samba.xml*-Szenario: Nach dem Start des Szenarios kopiert man die Konfigurationsdatei */etc/samba/smb.conf* und startet den Samba-Server auf *server*:

```
# vnumlparser.pl -x server@smb.xml -v
```

Dann kann man von der VM *client* aus das SMB-Share vom Server mounten:

```
client:~# mount -t smbfs //10.0.0.1/public /mnt/smb
client:~# cd /mnt/smb && ls
```

Alternativ dazu existiert auch hier wieder eine Exec-Sequenz *client*, welche zunächst einen entsprechenden Eintrag in die */etc/fstab* schreibt und dann das Mounten mittels des folgenden Kommandos erlaubt:

```
client:~# mount /mnt/smb
```

Komplexeres Service-Szenario: Um das Zusammenspiel all dieser Netzwerkdienste zu testen, entstand im Projektpraktikum des Sommersemsters 2006 ein größeres Beispiel-Szenario. Dieses ist nun in einer für VNUML 1.7 angepassten Form als *ppss06.xml* verfügbar. Zum Starten:

```
# vnumlparser.pl -t ppss06.xml -vB
# vnumlparser.pl -x config@ppss06.xml
```

Das Szenario besteht aus drei separaten Netzen (10.0.0.0/24, 10.0.1.0/24 und 10.0.2.0/24). Diese sind jeweils untereinander mit einem Router verbunden, auf denen das Routing-Protokoll RIP arbeitet. Jedes der Netze beherbergt mehrere virtuelle Rechner, welche teilweise auch als Server für Dienste wie DNS, DHCP, HTTP, FTP, NIS oder NFS dienten (vgl. Abb. 6).

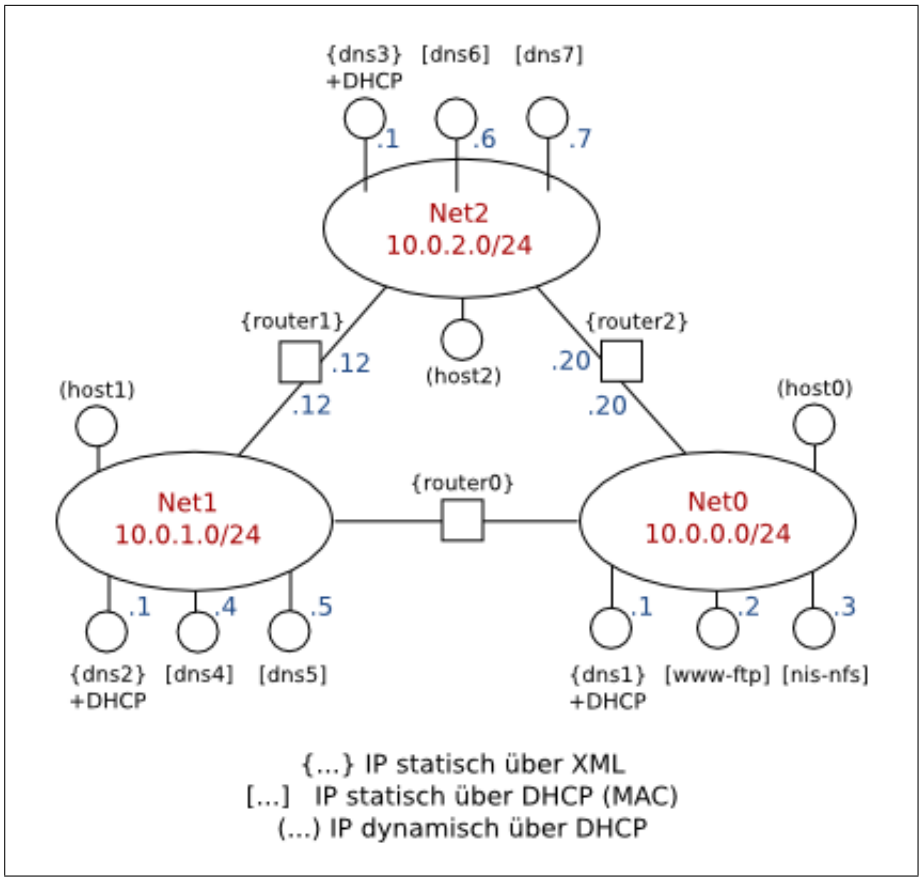

Abbildung 6: Topologie des Szenarios *ppss06.xml*

Die vorhandene DNS-Hierarchie teilt das Szenario in zwei Sites auf. Der Server *dns1* dient dabei nicht nur als Server für das Netz 10.0.0.0/24, sondern auch als Master-Server für das gesammte Szenario. *dns2* und *dns3* regeln die DNS-Anfragen in den eigenen lokalen Netzen und verzweigen jeweils noch einmal auf zwei weitere DNS-Server (vgl. Abb. 7 auf S.27).

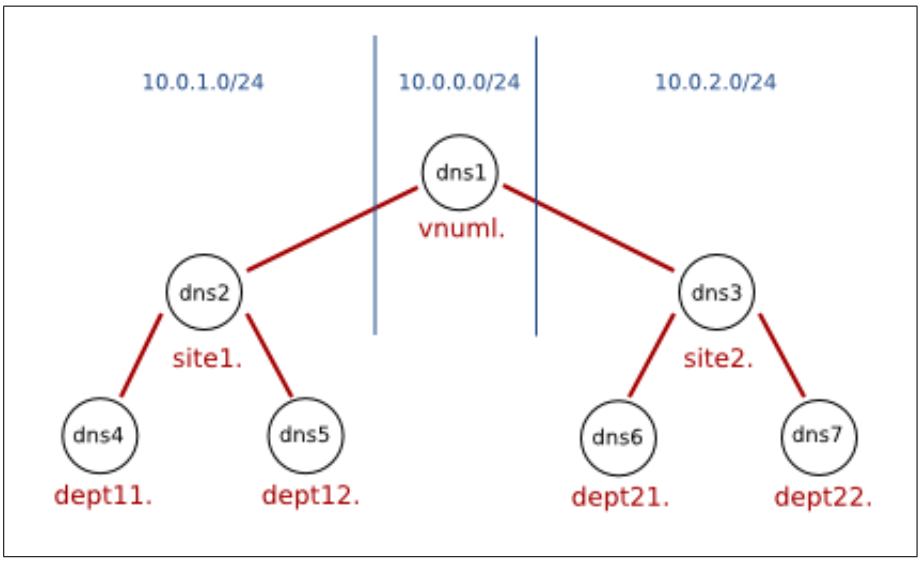

Abbildung 7: DNS-Hirarchie des Szenarios *ppss06.xml*

Zum Testen der Services stehen beispielsweise die folgenden Möglichkeiten zur Verfügung:

DNS: Mit dem Starten der *config*-Sequenz werden die sieben DNS-Server automatisch gestartet. Es ist dann möglich, einen Rechner über dessen Namen zu erreichen: `ping nis.vnuml`

DHCP: Auch die DHCP-Server starten durch die *config*-Sequenz. Die VMs, welche DHCP nutzen, beziehen über den Aufruf von `dhcpcd ethX` automatisch ihre IPs. Zu beachten ist, dass die Server jeweils eine feste IP zu ihrer MAC-Adresse zugeordnet haben, die drei Hosts jedoch einfach auf freie IPs aus dem Pool zurückgreifen.

FTP: Von einer beliebigen Host VM erfolgt der Zugriff auf den FTP-Server durch das Kommando `ftp ftp.vnuml`. Ein Beispielnutzer *user1* mit dem Passwort *xxxx* ermöglicht den Login.

Web: Von einer beliebigen Host VM erfolgt der Zugriff auf den Web-Server durch das Kommando `lynx www.vnuml`.

NFS: Von einer Host VM aus dem 10.0.0.0/16 Netz kann man den freigegebenen Pfad durch das folgende Kommando mounten:

```
mount -t nfs nfs.vnuml:/public/nfs /mnt/nfs
```

NIS: NIS ist nur für den *host0* verfügbar. Um NIS hier zu nutzen, startet man erst den NIS-Client mittels `/etc/init.d/nis start` und loggt dann den Beispielbenutzer ein: `login nisuser`.

8 Routing

Bislang wurde Routing nur durch das Anlegen von statischen Routen erreicht. Für die kleinen Simulationsbeispiele ist dies ausreichend, wesentlich interessanter ist jedoch der Einsatz von Routing-Daemons. Diese Daemons arbeiten untereinander mit speziellen dezentralen Routing-Protokollen und tauschen gewisse Informationen über das Netzwerk aus. Mit Hilfe dieser Daten soll es dann möglich sein, im betreffenden Netzwerk automatisch die Routing-Tabellen aufzustellen und konvergent zu halten.

Mit VNUML können auch komplexere Netze nachgebildet werden, in denen der Einsatz von Routing-Protokollen Sinn macht. Dazu kommt zum Beispiel die Quagga-Suite[8] zum Einsatz.

8.1 Die Quagga-Suite

Quagga ist ein Softwareprojekt, welches aus dem GNU Zebra Projekt hervorgegangen ist. Das Ziel von Quagga ist die Bereitstellung einer Reihe von freien Routing Daemons für Linux und andere Unix-Ähnliche Systeme. [Ish04].

Unterstützt werden die gängigen Routing-Protokolle: RIP v1 und v2 (`ripd`), OSPF v2 (`ospfd`) und BGPv4+ (`bgpd`). Neben diesen wird auch Unterstützung für die IPv6-basierten Varianten RIPng (`ripngd`) und OSPFv3 (`ospf6d`) geboten.

Der Einsatz von Quagga gestaltet sich recht einfach. Neben dem gewünschten Routing-Daemon muss lediglich zunächst der *zebra*-Daemon gestartet werden. Dieser sorgt unter anderem dafür, dass neu erhaltene Routinginformationen in die Weiterleitungstabelle des Systems übernommen werden. Nach dem Start eines Daemons kann man mittels `telnet` auf diesen zugreifen:

```
$ telnet localhost zebra          // bzw ripd / ospfd / bgpd
```

Hier können aktuelle Statusinformationen abgerufen und Konfigurationen eingegeben werden. Komfortabler lassen sich die Einstellungen jedoch über eine Konfigurationsdatei vornehmen, welche dem Daemon beim Start mit dem *-f* Parameter mitgegeben werden kann. Die Liste der möglichen Einstellungen ist recht lang, eine Übersicht findet sich auf der Quagga-Homepage[9].

Beim Einsatz von Quagga unter VNUML bietet es sich an, die benötigten Konfigurationsdateien (*ripd.conf* und *zebra.conf* am Beispiel RIP) in einem separaten Verzeichnis abzulegen und dieses per <*basedir*> und <*filetree*> in das Szenario einzubinden. Die Daemons lassen sich dann sehr einfach über ein Exec-Tag starten:

```
<filetree when="start" root="/conf">router1</filetree>
<exec seq="start" type="verbatim">zebra -f /conf/zebra.conf -d</exec>
<exec seq="start" type="verbatim">ripd -f /conf/ripd.conf -d</exec>
```

Hier wird das Verzeichnis *router1/* im Dateisystem der virtuellen Maschine als */conf/* eingebunden und die Daemons greifen über den *-f* Parameter auf die bereitgestellten Dateien zurück.

[8] http://www.quagga.net/
[9] http://quagga.net/docs/quagga.html

8.2 RIP (Routing Information Protocol)

Das RIP-Protokoll gehört zur Familie der Interior Gateway Protokolle und basiert auf dem Distanz-Vector Ansatz [Per05].

Ein RIP-Router kennt nur seine direkt angrenzenden Nachbarn. Diesen schickt er in periodischen Abständen eine Kopie seiner eigenen Routing-Tabelle (*Advertisement*). Auf diesem Wege können die Router nach und nach neue Erreichbarkeiten erlernen. Probleme bei RIP stellen die vergleichsweise hohen Konvergenzzeiten und das Count-to-Infinity dar. Als Beispiel-Szenario für RIP dient *rip_basic.xml* (vgl. Abb. 8).

Das Szenario beherbergt zwei Router *R1* und *R2*, welche jeweils an eins der Netze *Net1* oder *Net2* angeschlossen sind. Ein dritter Router *R3* hat Anschluss an beide Netze. Es existiert ein Exec-Tag *start*, welches die benötigten Konfigurationsdateien *zebra.conf* und *ripd.conf* auf die VMs kopiert und dann die Daemons startet. Durch den Austausch der Routinginformationen lernt *R1* die Route zu *Net2* und *R2* die Route zu *Net1*.

Tiefergehende Erklärungen und komplexere Beispiele zu RIP unter VNUML finden sich im Paper *RIP Network Laboratory* [KÖ5] von Andreas Köbler.

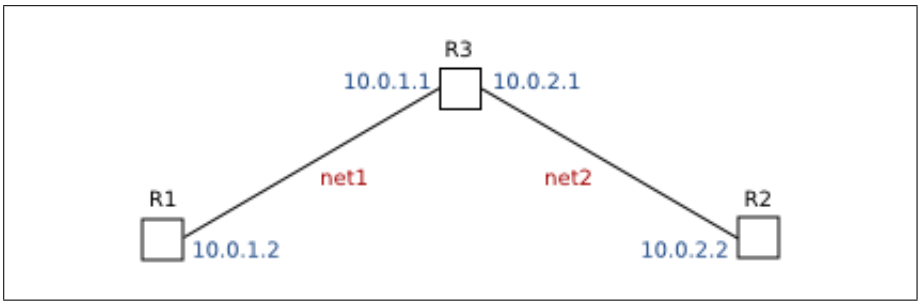

Abbildung 8: Topologie der Szenarien *rip_basic.xml* und *ospf_basic.xml*

8.3 OSPF (Open Shortest Path First)

Das OSPF-Protokoll gehört wie RIP zur Familie der Interior Gateway Protokolle, basiert jedoch auf dem Link-State Ansatz. Gerade in größeren Netzen wird heute vorwiegend OSPF eingesetzt [AB05].

Bei OSPF erlangt jeder teilnehmende Router das Wissen über die gesammte Topologie des Netzwerks. Dies geschieht über das sogenannte *fluten*: jeder Router sendet eine Nachricht mit den angrenzenden Netzen an alle anderen Router. Mittels dieser Informationen kann dann jeder Router einen Erreichbarkeitsgraphen erstellen und die optimale Route zu jedem anderen Knoten im Netz berechnen. Dazu werden auch die Kosten der einzelnen Pfade in Betracht genommen. Als Beispiel-Szenario für OSPF dient *ospf_basic.xml* (vgl. Abb. 8).

Das Szenario ist analog zum RIP-Beispiel aufgebaut. Auch hier existiert wieder ein Exec-Tag *start*, welches die benötigten Konfigurationsdateien *zebra.conf* und *ospfd.conf* auf die VMs kopiert und dann die Daemons startet. Durch den Austausch der Routinginformationen lernt *R1* die Route zu *Net2* und *R2* die Route zu *Net1*.

Tiefergehende Erklärungen und komplexere Beispiele zu OSPF unter VNUML finden sich im Paper *OSPF unter VNUML* [RÖ5] von Thorsten Römer.

8.4 BGP (Boarder Gateway Protocol)

Das Boarder Gateway Protocol ist das momentan gebräuchliche Protokoll zum Routing zwischen autonomen Systemen und stellt somit das Protokoll dar, welches das Zusammenspiel des Internet ermöglicht.

BGP ist ein Pfad-Vektor-Protokoll. Im Vergleich zu Distanz-Vektor-Protokollen wie RIP pflegt BGP bei seinen Erreichbarkeitsinformationen ein Attribut namens *AS_PATH*, welches den genauen Pfad in ein entferntes autonomes System beinhaltet. Durch Auswertung dieses Attributs kann es hier nicht zu Problemen mit Routing-Schleifen kommen.

Das BGP-Protokoll läuft auf TCP-Basis und bietet die vier Nachrichtentypen *OPEN*, *UPDATE*, *NOTIFICATION* und *KEEPALIVE*. Als Beispiel-Szenario für BGP dient *bgp_basic.xml* (vgl. Abb. 9).

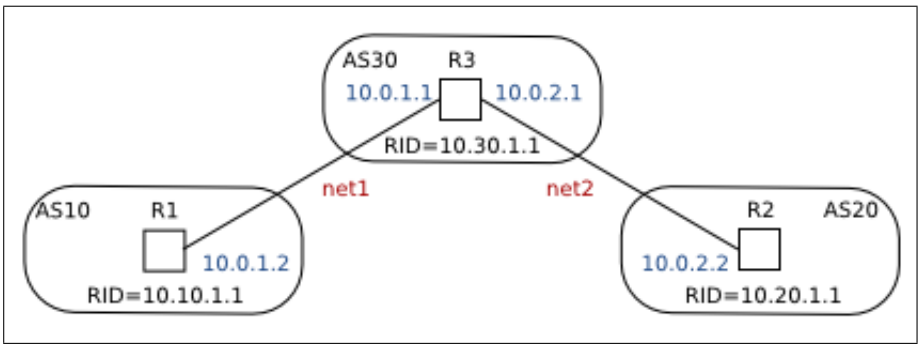

Abbildung 9: Topologie des Szenarios *bgp_basic.xml*

Dieses Szenario ähnelt vom Aufbau her den vorangegangenen Beispielen, die Router befinden sich jetzt jedoch in den abgegrenzten autonomen Systemen AS10, AS20 und AS30. Für die Verbindung unter den BGP-Routern werden jeweils Loopback Interfaces genutzt. Durch den Austausch der Routinginformationen lernt *R1* die Route zu *Net2* und *R2* die Route zu *Net1*.

Tiefergehende Erklärungen und komplexere Beispiele zum BGP unter VNUML finden sich im Paper *BGP unter VNUML* [Bil06] von Daniel Bildhauer.

Komplexeres BGP-Szenario: Ein größeres Beispielszenario für BGP steht mit einer Version von *alpen.xml* bereit (vgl. Abb. 10).

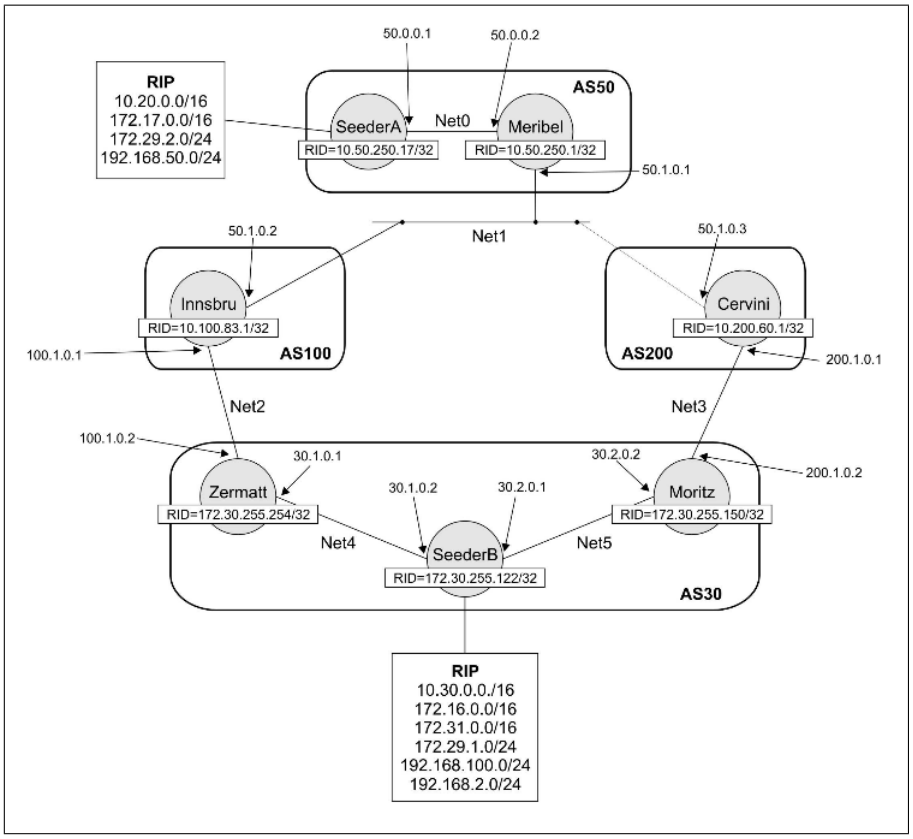

Abbildung 10: Topologie des Szenarios *alpen.xml* nach [Dic05]

Dieses Szenario aus der Rechnernetze 2 Übung wurde für VNUML 1.7 und das neue Dateisystem angepasst. Es beinhaltet vier miteinander verbundene autonome Systeme (AS30, AS50, AS100 und AS200), welche untereinander das Routing über BGP abstimmen. Innerhalb von AS30 und AS50 wird RIP genutzt und einige RIP-Präfixe werden über die Netze verteilt. Das Szenario macht zudem Gebrauch von diversen Filterregeln.

In diesem Szenario gibt es für den Interessierten viele Dinge zu erkunden. Interessant ist bereits ein Blick auf die langsam wachsenden Routing-Tabellen nach dem Start des Szenarios. Als Anhaltspunkt für weitere mögliche Problemstellungen können beispielsweise die Fragen aus der Rechnernetze 2 Übung [Dic05] dienen.

9 Abschließendes Wort und Formales

Die Simulation von Netzwerken ist gerade für angehende Informatiker eine schöne Möglichkeit, sich mit der oftmals nicht trivialen Materie spielerisch vertraut zu machen.

Gerade VNUML hat hier unschätzbaren Wert, da dem Benutzer die virtuellen Maschinen mit einem realen Betriebssystem zugänglich gemacht werden und somit fast realistische Bedingungen geschaffen werden.

Der Leser sollte nun die prinzipiellen Möglichkeiten von VNUML kennen und selbst in der Lage sein, komplexere Szenarien anhand eigener Fragestellungen zu erarbeiten. Besonders der Umgang mit NAT und Routing konnte wegen ihrer hohen Komplexität nur Ansatzweise beschrieben werden. Allerdings sollten die hier aufgezeigten Mittel, in Kombination mit den Beispielen, eine gute Grundlage für eine Vertiefung in diese Themen sein.

Ich finde es wichtig, dass dieser praktische Teil in der Lehre auch in Zukunft nicht zu kurz kommen wird und hoffe, dass diese Ausarbeitung den interessierten Lesern den Einstieg in das Thema etwas erleichtern konnte.

Erklärung nach §10 Abs. 6 Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Veröffentlichung Ich erkläre mich damit einverstanden, dass diese Studienarbeit in digitaler oder ausgedruckter Form von der Universität Koblenz öffentlich zugänglich gemacht wird. Dies schließt die Auslage von Exemplaren in der Universitätsbibliothek ein.

A FAQ: Frequently Asked Questions

An dieser Stelle sollen Lösungen für häufig auftretende Probleme gegeben werden. Einige weitere Fragestellungen werden auf den FAQ-Seiten des VNUML-Wikis[10] oder der Universität Koblenz[11] behandelt.

A.1 Einige Programme in den VMs laufen nicht

Dies kann natürlich verschiedene Ursachen haben, es ist allerdings gut möglich, dass das Programm Thread Local Storage (TLS) nutzt, welches vom UML-Kernel oft noch nicht richtig unterstützt wird. In diesem Fall hat oft das Setzen der folgenden Umgebungsvariable Abhilfe schaffen können:

```
export LD_ASSUME_KERNEL=2.4.1
```

Alternativ könnte auch das Verschieben des TLS-Ordners helfen:

```
# mv /lib/tls /lib/tls.bak
```

A.2 Die VMs starten nicht mit der X-Term Option

Zuerst muss man sicherstellen, dass die benötigten Berechtigungen vorliegen. Dazu gibt es zwei Möglichkeiten: Entweder, man startet VNUML mit Root-Rechten und dem zusätzlichen Parameter *-u root*. Dies bewirkt, dass VNUML nicht versucht, die Root-Rechte weitgehend abzulegen und intern den User *vnuml* zu nutzen. Die Alternative besteht darin, dem VNUML-Benutzer Zugriff auf das X11-Display zu geben:

```
# xhost +local:vnuml
```

A.3 Mit den VMs ins Internet

Man muss lediglich NAT und IP-Forwarding auf dem Hostrechner aktivieren:

```
# iptables -t nat -A POSTROUTING -o ath0 -j MASQUERADE
# sh -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'
```

A.4 Probleme beim Routing

Treten unerklärliche Routingprobleme auf, so hängt dies oft mit dem RP-Filter von Linux zusammen. Um diesen zu deaktivieren, gibt man folgendes auf den Router-VMs ein:

```
for f in /proc/sys/net/ipv4/conf/*/rp_filter; do echo 0 > $f; done
```

[10] http://www.dit.upm.es/vnumlwiki/index.php/FAQ
[11] http://www.uni-koblenz.de/vnuml/error_list.de.php

A.5 Update der VNUML-Installation

Nutzt man die Debian Pakete, so geht ein Update der VNUML-Version absolut transparent vor sich. Anders ist es jedoch, wenn man auf dem System zuvor den Offline-Installer oder eine manuelle Installation genutzt hat. In diesen Fällen sollte man zunächst die alte Installation so gut es geht entfernen:

```
# rm -rf /usr/local/bin/vnumlparser*
# rm -rf /usr/local/man/man1/vnumlparser.1.gz
# rm -rf /usr/local/lib/vnuml
# rm -rf /usr/local/share/vnuml
# rm -rf /usr/local/usr
```

Diese Vorgehensweise sollte unbedenklich sein, der Benutzer möge sich allerdings erst selbst vergewissern, dass dabei keine noch benötigte Dateien versehentlich (Szenarien!) mit gelöscht werden. Um auch die Bibliotheken und Perl-Module des Offline-Installers los zu werden:

```
# rm -rf /usr/local/lib/*
```

Achtung: Hier ist Vorsicht geboten, falls man außer dem Offline Installer noch andere Programme in den `/usr/local`-Präfix installiert hat!

Nachdem die alten Komponenten entfernt worden sind, kann man `apt` wie im Kapitel *Installation* beschrieben einrichten. Bei der eigentlichen Installation sollte man dann auch die Pakete `bridge-utils` und `uml-utilities` neu installieren:

```
# apt-get --reinstall install bridge-utils uml-utilities vnuml
```

A.6 Probleme mit SSH

Besonders in großen Szenarien kann es vorkommen, dass eine VM zwar ordnungsgemäß bootet, aber dann aus unerfindlichen gründen den SSH-Server nicht startet. Das kann in etwa so aussehen:

```
zermatt sshd is ready (socket style): 192.168.0.2 (mng_if)
moritz sshd is not ready (socket style)
host> rm -f /root/.vnuml/LOCK
main::mode_x (2022): UMLs are booted but not ready yet. Wait a while and retry...
```

Falls man die VM in einem XTerm gestartet hat und sich einloggen kann, lässt sich der Fehler leicht beheben:

```
# /etc/init.d/ssh start
```

B Linux Shell Kommandos

Da eine virtuelle Maschine in VNUML ein komplettes Linux-System enthält, stehen hier auch die gleichen Konsolenbefehle und Programme zur Verfügung. Für Linux-Neulinge kann dies auf den ersten Blick etwas komplex und verwirrend erscheinen. Die folgende Tabelle fasst deshalb die wichtigsten Shell-Befehle zusammen.

Befehl	**Beschreibung**
./Programm	Startet das *Programm* (oder Skript)
su	Abfrage des Root-Passworts und Gewähren von Root-Rechten
man *Befehl*	Zeigt eine Hilfe zu *Befehl*
pwd	Zeigt die aktuelle Stelle im Verzeichnisbaum
ls	Zeigt den Inhalt des aktuellen Verzeichnisses
cd */Verzeichnis*	Springt im Verzeichnisbaum nach *Verzeichnis* (von der Wurzel aus)
cd *Verzeichnis*	Springt im Verzeichnisbaum nach *Verzeichnis* (vom akt. Verzeichnis aus)
cd ..	Springt eine Ebene höher
cp *Quelle Ziel*	Kopiert von *Quelle* nach *Ziel*
mv *Quelle Ziel*	Verschiebt von *Quelle* nach *Ziel*
rm -rf *Dateiname*	Löscht *Dateiname* (oder Verzeichnis)
cat *Datei1 DateiN*	Zeigt die Dateien 1 bis N an
wget -c *URL*	Lädt die Datei hinter *URL* herunter
tar -xvzf *Archiv*	Entpackt das ein .tar.gz Archiv
tar -xvjf *Archiv*	Entpackt das ein .tar.bz2 Archiv
unzip *Archiv*	Entpackt das ein .zip Archiv
chown *Benutzer*:*Gruppe Datei*	Setzt Eigner für *Datei*
chmod *Berechtigungen Datei*	Setzt *Berechtigungen* für *Datei*

Tabelle 1: Linux Shell Befehlsreferenz für Host und VMs

Die *Berechtigungen* für Dateien und Ordner werden dabei durch das Bit-Tripel *(user, group, others)* gebildet:

user	Rechte des eigenen Benutzers
group	Rechte der eigenen Gruppe
others	Rechte für sonstige Benutzer

Jedes einzelne dieser Bits entspricht der Summe der Bitwerte der gewünschten Berechtigungen:

Berechtigung	**Bitwert**
r/read/lesen	4
w/write/schreiben	2
x/execute/ausführen	1

Beispiel: chmod 644 *Datei* (Eigner: lesen/schreiben, alle anderen: nur lesen)

C Erstellung eines UML Kernels

Der bereitgestellte UML Kernel, den die virtuellen Maschinen in den Simulations-Beispielen nutzen, wurde extra für diesen Zweck von Hand erstellt. Die dazu nötige Prozedur sei hier nur kurz beschrieben:

```
$ wget -c http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.1.tar.bz2
$ tar -xvjf linux-2.6.20.1.tar.bz2
$ cd linux-2.6.20.1
$ cat ../linux-2.6.16-xterm.patch | patch -p1
$ export VER=2.6.20.1-uml0
$ export CC=gcc-3.4
$ make menuconfig ARCH=um
$ make ARCH=um
$ make modules ARCH=um
$ strip linux
```

Als Basis dient also ein gewöhnlicher Linux Kernel. Der erwähnte Patch entstammt dem offiziellen VNUML-Kernel[12] und dient rein optional dem Ändern des Fenstertitels eines XTerms, was nur sinnvoll ist, falls das genutzte Szenario die <xterm>-Option nutzt. Da der GNU C-Compiler in der Version 4.x oftmals Probleme mit UML hat, wird hier explizit eine 3.x Version verwendet. Ansonsten entspricht das Übersetzen des Kernels der gewöhnlichen Vorgehensweise, jeweils ergänzt um den Zusatz *ARCH=um*.

Natürlich muss zuvor sicher gestellt sein, dass Pakete wie *gcc* und *make* verfügbar sind. Die meisten Linux-Distributionen bieten dazu ein passendes Meta-Paket, für Debian ist dies beispielsweise *linux-kernel-devel*.

Zur Weitergabe eines Kernels empfiehlt es sich, den Kernel, die zugehörige Konfigurationsdatei und die Module wie folgt zu archivieren:

```
$ mkdir -p $HOME/uml-kernel-$VER
$ cp linux $HOME/uml-kernel-$VER/linux-$VER
$ make modules_install ARCH=um \
  INSTALL_MOD\_PATH=$HOME/uml-kernel-$VER/
$ cp .config $HOME/uml-kernel/config-$VER
$ cd $HOME/
$ tar -cvjf uml-kernel-$VER.tar.bz2 uml-kernel-$VER/
```

[12] http://downloads.sourceforge.net/vnuml/linux-2.6.18.1-bb2-xt-1m.tar.bz2

D Erstellung eines VNUML Dateisystems

Wie der zuvor beschriebene UML Kernel, so wurde auch das mitgelieferte Dateisystem passend für diesen Zweck von Hand erstellt [MG07]. Die hierzu entwickelte Vorgehensweise wurde mittlerweile auch vom VNUML-Projekt in das offizielle Wiki übernommen[13].

Zunächst wird ein leeres Dateisystem-Image angelegt (hier: ext3 mit 700MB):

```
$ dd if=/dev/zero of=/usr/share/vnuml/filesystems/debian.ext3 \
  bs=1024k seek=700 count=0
$ mkfs.ext3 -j /usr/share/vnuml/filesystems/debian.ext3
$ tune2fs -c 0 -i 0 /usr/share/vnuml/filesystems/debian.ext3
```

Dieses Image wird nun gemountet und die Debian-Installation gestartet:

```
# mkdir /mnt/debian
# mount -o loop /usr/share/vnuml/filesystems/debian.ext3 /mnt/debian
# debootstrap etch /mnt/debian/ ftp://ftp.de.debian.org/debian/
```

Nachdem das Debian Basisystem installiert ist, kann man auf dieses über den *chroot*-Mechanismus zugreifen (auch nützlich für spätere Modifikationen):

```
# mount -t proc none /mnt/debian/proc
# chroot /mnt/debian
# env -update
```

Man arbeitet nun auf einer Shell innerhalb des neu angelegten Dateisystems. Erster Schritt ist die Anpassung der Datei */etc/apt/sources.list*:

```
deb http://ftp.de.debian.org/debian etch main contrib non-free
```

Dann eine Auffrischung des Paketmanager-Cache:

```
# apt-get update
```

Je nach Wunsch kann man nun weitere Pakete installieren. Für das mitgelieferte Dateisystem (Achtung: openssh-server sollte in jedem Fall installiert werden, da VNUML sonst keine Verbindung zu den VMs herstellen kann!):

```
# apt-get install apache2.2-common apache2-mpm-prefork apache2-utils \
  bzip2 dnsutils ethereal-common fdutils file fileutils finger ftp ftpd \
  iproute ipsec-tools iputils-tracepath ipv6calc ipvsadm less lpr lsof \
  lynx modutils ncurses-term  nfs-common openssh-client openssh-server \
  openssl perl perl-modules pidentd portmap ppp procmail python \
  python-central python-newt quagga racoon radvd setserial sharutils \
  shellutils squid squid-common ssh ssl-cert strace syslinux tcpdump \
  tcsh telnet tethereal textutils time tshark ucf vlan whois \
  wireshark-common sudo vim iptables build-essencial make psmisc
```

Das System läuft auf Englisch, es sei denn man installiert noch die Lokalisierungspakete und wechselt auf deutsch:

[13] http://www.dit.upm.es/vnumlwiki/index.php/Create-rootfs

```
# apt-get install locales console-data
# dpkg-reconfigure locales

------------------------------
  [*] de_DE ISO-8859-1
  [*] de_DE.UTF-8 UTF-8
  [*] de_DE@euro ISO-8859-15
------------------------------
```

Nach der Installation kann man etwas Platz sparen, indem man den Cache des Paketmanagers wieder löscht:

```
# apt-get clean
```

Um eventuellen späteren Problemen mit TLS aus dem Wege zu gehen:

```
# mv /lib/tls /lib/tls.bak
```

Nun sollte man die Zeitzone einstellen:

```
# ln -sf /usr/share/zoneinfo/Europe/Berlin /etc/localtime
```

Sowie zumindest das Localhost-Alias in der */etc/hosts*:

```
127.0.0.1 localhost
::1             localhost

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

Optional kann man in der */etc/hostname* den Hostname ändern. In jedem Fall muss aber dann in der */etc/network/interfaces* ein Loopback-Interface angelegt werden:

```
iface lo inet loopback
auto lo
```

Nun die Mountpoints in der */etc/fstab* anpassen:

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

Falls die VMs später ein Root-Passwort haben sollen, muss dieses jetzt angelegt werden:

```
# passwd
```

Mit *passwd -d root* kann man das Passwort später auch wieder entfernen. Eventuell gewünschte User können auch angelegt werden:

```
# addgroup --gid 1000 test
# adduser --id 1000 --gid 1000 test
```

Bis jetzt ähnelt die Installation einer normalen Debian-Installation mit UML-Anpassungen. Speziell für VNUML sind noch folgende Änderungen von Nöten:

```
# mkdir /mnt/vnuml
# ln -s /mnt/vnuml/umlboot /etc/rc2.d/S11umlboot
```

Einträge in der */etc/securetty* für *tty0* und *ssh* sowie in */etc/inittab*, damit auf Console 0 ein Login erscheint:

```
0:2345:respawn:/sbin/getty 38400 tty0
```

Nun kann man die chroot-Umgebung wieder verlassen, die letzten Schritte werden vom Host-Rechner aus durchgeführt:

```
# exit
```

Ein oft vergessener Schritt ist das Anlegen der UML Device-Nodes */dev/ubdX*. Dies geschieht am einfachsten über ein Skript:

```
$ cd /mnt/debian/dev
$ wget http://www.theshore.net/~caker/uml/makeUBDdev.sh
$ chmod a+x makeUBDdev.sh
$ ./makeUBDdev.sh && rm makeUBDdev.sh
```

Zuletzt muss man die zum später genutzten Kernel passenden Module in das Dateisystem kopieren. Ausgehend von Anhang B also:

```
# cp -R $HOME/uml-kernel-2.6.20.1-uml0/lib/modules/2.6.20.1-uml0 \
   /mnt/debian/lib/modules/
```

Dann einfach unmounten:

```
# cd /mnt
# umount /mnt/debian/proc/
# umount /mnt/debian
```

Jetzt sollte das Image unter VNUML problemlos laufen. Falls man nachher noch Pakete installieren will, nutzt man einfach erneut den *chroot*-Mechanismus wie hier beschrieben und von dort aus *apt-get*, um die fehlenden Pakete zu installieren oder beliebige Änderungen vorzunehmen.

E Graphische Oberfläche (X11) unter (VN)UML

Um graphische Anwendungen auf einer virtuellen Maschine zu starten, müssen einige Vorbereitungen getroffen werden. So muss auf dem Host ein entsprechender X11-Server (*xnest*) installiert und gestartet sein [Fis06]:

```
# apt-get install xnest
$ Xnest :2 -ac &
```

Wobei :2 stellvertretend für eine freie Display-Nummer steht. Im mitgelieferten Dateisystem sind die benötigten Pakete für die VMs bereits installiert. Kommt ein Dateisystem ohne diese Anpassung zur Verwendung, kann man dieses leicht über das in Anhang C vorgestellte *chroot*-Verfahren nachrüsten. Dazu im gemounteten Dateisystem der VM:

```
# apt-get install xterm matchbox dillo leafpad wireshark
```

Nachdem alle Vorkehrungen getroffen sind, wird auf der laufenden VM die Umgebungsvariable DISPLAY auf das freigegebene Xnest-Display des Hosts umgelenkt. Danach kann man beliebige X11-Programme starten, hier beispielsweise die Matchbox-Umgebung:

```
# export DISPLAY=IP_DES_HOSTRECHNERS:2
# matchbox-session &
```

In Matchbox sind im mitgelieferten Dateisystem unter anderem die Programme *xterm* (Terminal), *dillo* (Webbrowser), *leafpad* (Texteditor) und *wireshark* (Sniffer) verfügbar. Diese können jedoch auch ohne Matchbox gestartet werden. Die hier gewählten Programme laufen unter UML noch recht flüssig, von leistungshungrigeren Applikationen sollte man allerdings eher Abstand halten.

Abbildungsverzeichnis

Literatur

[AB05] Suat Algin and Lars Blumenstengel. OSPF. 2005.

[Bil06] Daniel Bildhauer. BGP mit VNUML. 2006.

[Bor05] Thorsten Bormer. Secure Shell (SSH). 2005.

[Bre07] Holger Breitbach. VNUML und NAT. 2007.

[Chm06] Thomas Chmielowiec. VNUML - Version 1.5 und 1.6. 2006.

[Dic05] Harald Dickel. Foliensatz zur Rechnernetze 2 übung. 2005.

[ELP06] Uli Eckstein, Rufus Linke, and Martin Pätzold. COW-Dateisystem, Softwareinstallation, DHCP, DNS und Routing in VNUML. 2006.

[FG07] Casey T. Deccio Fermín Galán, David Fernández. VNUML User Manual. 2007. http://www.dit.upm.es/vnumlwiki/index.php/Usermanual, aufgerufen am 28.07.2007.

[Fis06] Martin Fischer. Web- , FTP- und X-Server unter VNUML. 2006.

[Hor06] Tassilo Horn. Netfilter. 2006.

[Ish04] Kunihiro Ishiguro. Quagga: A routing software for TCP/IP networks. 2004.

[KÖ5] Andreas Köbler. RIP Network Laboratory. 2005.

[Keu05] Tim Keupen. User-mode Linux. 2005.

[Kor05] Matthias Korn. Eigener Kernel und eigenes Dateisystem für User-Mode-Linux. 2005.

[Lat07] Christian Latsch. Einfürung in VNUML. 2007.

[Lin06] Rufus Linke. Domain Name System in VNUML. 2006.

[MÖ5] Markus Müller. VNUML-Einführung in die Simulation von Rechnernetzen. 2005.

[MG07] Michael Monreal and Fermín Galán. How to create a VNUML-compatible root filesystem from scratch using debootstrap. 2007. http://www.dit.upm.es/vnumlwiki/index.php/Create-rootfs/, aufgerufen am 01.08.2007.

[MO06] Michael Monreal and Tomasz Oliwa. Benutzerverwaltung in GNU/Linux Netzwerken mit NIS/NFS. 2006.

[Mon07] Michael Monreal. Xen. 2007.

[OD06] Christian Oellig and Iwan Diel. FTP-Server unter VNUML. 2006.

[Per05] Christian Perscheid. RIP-Konfiguration mittels VNUML-Simulator. 2005.

[RÖ5] Torsten Römer. OSPF unter VNUML. 2005.

[Sch07] Patrick Schütz. VNUML unter VMware. 2007.

[Sla07] Slacksite. Active FTP vs. Passive FTP, a Definitive Explanation. 2007. http://slacksite.com/other/ftp.html, aufgerufen am 01.08.2007.

[Wik07] Wikipedia. User-mode Linux. 2007. http://en.wikipedia.org/wiki/User-mode_Linux, aufgerufen am 11.06.2007.