User-Mode Linux als Testplattform für Netzwerkstacks

Markus Hennig
Astaro AG
76133 Karlsruhe
mhennig@astaro.com

Bernhard Thurm
Institut für Telematik, Universität Karlsruhe
76131 Karlsruhe
thurm@tm.uni-karlsruhe.de

Kurzfassung

Die Nutzung virtueller Netzwerk-Szenarien unter User-Mode Linux (UML) stellt eine effiziente und kostengünstige Alternative zu dedizierten Testaufbauten dar – bisher für praxisnahe Protokolltests unabdingbare Voraussetzung. Im Folgenden wird ein spezielles, im Rahmen einer Forschungsarbeit am Institut für Telematik entstandenes virtuelles Netzwerk auf Basis von UML näher vorgestellt. Ziel war dabei die Simulation von IP-Routern unter besonderer Berücksichtigung der Protokolle OSPF (Open Shortest Path First), MPLS (Multi Protocol Label Switching) und LDP (Label Distribution Protocol).

1 Einleitung

Die sprunghafte Neu- und Weiterentwicklung von Kommunikationsprotokollen scheint bedingt durch neue Anwendungsszenarien, Dienstgüte- und Bandbreitenanforderungen auch in Zukunft anzudauern. Eine wesentliche Aufgabe im Laufe einer praktischen Umsetzung neuer Protokolle stellt dabei die Überprüfung von Korrektheit und Leistungsfähigkeit dar. Grundsätzliche Aussagen hierzu sind zwar schon während der Designphase durch Nutzung geeigneter Simulationswerkzeuge zu gewinnen, ein Problem stellen jedoch inhärente Unterschiede zwischen Simulationsmodell und späterer praktischer Implementierung dar. Vor dem eigentlichen Einsatz sind daher umfangreiche praktische Tests unabdingbar, die unter Nutzung dedizierter Hardware durchgeführt werden und einen immensen geräte- und kostentechnischen Aufwand erfordern.

Eine effiziente und kostengünstige Alternative zu dedizierten Testszenarien unter Linux stellt die Verwendung *virtueller* Systeme mittels UML (User-Mode Linux) dar. Hierbei integriert ein einzelner Rechner *(Host)* verschiedene weitere (weitgehend unveränderte, aber mit dem zu testenden Protokoll ausgestattete) Betriebssysteminstanzen. Der Zugriff auf die tatsächliche Hardware des Hosts erfolgt dabei über virtuelle Geräteeinträge (z.B. Netzwerkadapter). Virtuelle Systeme innerhalb des Hosts sind sowohl nach Außen als auch untereinander nicht von herkömmlichen (realen) Systemen zu unterscheiden.

Im Folgenden wird die Realisierung eines derartigen virtuellen Testnetzwerkes näher beschrieben. Spezieller Schwerpunkt in der Entwicklung lag dabei auf der Simulation von IP-Routern unter besonderer Berücksichtigung der Protokolle OSPF (Open Shortest Path First), MPLS (Multi Protocol Label Switching) und LDP (Label Distribution Protocol).

Der Aufbau der Arbeit gliedert sich wie folgt. Nach einer Vorstellung von UML und MPLS in Kapitel 2, wird in Kapitel 3 zunächst der Simulationsaufbau übersichtsartig beschrieben. Im

darauf folgenden Kapitel 4 werden dann das aufgebaute Szenario und die darin durchgeführten Tests erläutert. Kapitel 5 fasst die wichtigsten Punkte der Arbeit zusammen.

2 Grundlagen

Im Folgenden wird in Abschnitt 2.1 zunächst User-Mode Linux (UML) vorgestellt, bevor Abschnitt 2.2 die grundlegenden Eigenschaften des Multiprotocol Label Switching (MPLS) beschreibt.

2.1 User-Mode Linux

Mit der virtuellen Plattform *User-Mode* [2] wurde ab Ende 1999 von Jeff Dike für den damaligen Entwicklungskernel 2.3.XX ein Patch geschaffen, der es erlaubt, den Kernel als Programm im Anwenderadressraum auszuführen. Damit sind multiple Instanzen von Linux-Kerneln auf einem Host-System möglich, die unabhängig voneinander und von der Host-Hardware jeweils ein vollwertiges Betriebssystem zur Verfügung stellen. Für Prozesse im *User-Mode Linux* (UML) ist diese virtuelle Plattform völlig transparent, das heisst die reale Plattform des Host-Systems steht auch diesen Prozessen zur Verfügung (Ausnahmen sind hier direkte Hardwarezugriffe bzw. einige spezielle Systemfunktionen und IOCTLs).

Schon früh in der Entwicklung von User-Mode Linux wurden verschiedene Netzwerkschnittstellen geschaffen, die den direkten Zugriff auf die Netzwerkanschlüsse des Host-Systems erlauben. Parallel dazu können auch virtuelle Netzwerkschnittstellen definiert werden, die nur den User-Mode Linux Instanzen zur Verfügung stehen und die *virtuelle* Vernetzung der Instanzen ermöglichen.

Benutzt wird hierbei vorrangig die */proc/net/tun*-Schnittstelle des Kernels, über die Datenpakete mit Schicht-2-Informationen zwischen Applikationen ausgetauscht werden können. Dafür wird im Host-System eine *tap*-Netzwerkschnittstelle erzeugt und über den Hilfsprozess *uml_net* eine Punkt-zu-Punkt-Abbildung auf eine virtuelle Netzwerkschnittstelle in UML generiert. Die virtuelle Netzwerkschnittstelle in UML erscheint als Ethernet-Schnittstelle und kann wie gewohnt konfiguriert und benutzt werden. Dem UML-Kernel kann beim Starten für jede virtuelle Netzwerkschnittstelle eine MAC-Adresse als Kommandozeilenargument übergeben werden. Für rein virtuelle Netze zwischen UML-Instanzen entfällt die Verbindung zum Host-System, stattdessen wird über *uml_net* eine *tap*-vergleichbare Verbindung von der UML-Instanz zum virtuellen Switch *uml_switch* hergestellt. Äquivalent zu einem realen Switch arbeitet auch der virtuelle Switch auf Schicht 2 des ISO/OSI-Referenzmodells. Beide Mechanismen werden für das vorgestellte Simulationsnetzwerk verwendet.

2.1.1 Speicherverwaltung

Der einer User-Mode Linux Instanz zugewiesene Speicher wird als Datei im Dateisystem des Hosts abgelegt. Dies ermöglicht die Nutzung eines größeren virtuellen Speichers als tatsächlich physisch im Host verfügbar und das Abbilden dieser Datei auf verschiedene physikalische oder logische Dateisysteme. Besonders das sog. *tmpfs*[1] spielt hierbei eine große Rolle, da es einen Bereich des realen Speichers des Host-Systems als zusätzliche Partition in das Dateisystem des Hosts einbindet. Dadurch kann es zum Beispiel von UML als sehr schneller Speicher genutzt werden.

UML bietet einen *Separate Kernel Address Space*-Mode (SKAS) an, in dem UML die Verwaltung seiner Prozesse selbst übernimmt und für das Host-System nur als einzelner Prozess

[1] tmpfs ist ein ausschließlich im Speicher gehaltenes Dateisystem, welches oberhalb der virtuellen Dateisystemschicht angesiedelt ist und das bisher verwendete *RAM-Disk*-Prinzip ablösen soll.

(*Usermode-Kernel*) sichtbar ist. Für diesen Zweck wird die Schnittstelle */proc/mm* im Host-System benutzt. Im Gegensatz dazu verwaltet der Host-Kernel im *Tracing Thread*-Mode[2] auch alle Prozesse der UML Instanzen, was weniger performant und durch eine fehlende Trennung der Adressräume auch weniger sicher ist.

Die virtuelle Festplatte der UML-Instanz wird ebenfalls als einzelne Datei auf dem Dateisystem des Hosts abgelegt. UML benutzt dabei Partitionen in virtuellen Festplatten, so dass reale Systeme sehr detailgenau simuliert werden können. Mit dem *Change-on-Write* (COW)-System kann ein virtuelles Dateisystem als Read-Only-System benutzt werden. Änderungen auf diesem System während der Laufzeit werden in einer zweiten Datei festgehalten. Dies ermöglicht die effiziente mehrfache Verwendung eines virtuellen Dateisystems für mehrere UML-Instanzen. Da alle Dateioperationen in UML auch als Dateioperationen auf der realen Festplatte des Host-Systems stattfinden müssen, sind diese für eine hohe Leistung des Gesamtsystems möglichst zu vermeiden. Dies impliziert möglichst den Verzicht auf SWAP-Partitionen und SWAP-Dateien in UML bei gleichzeitig reichhaltiger (physischer) Speicherausstattung des Host-Systems (und somit der einzelnen Instanzen).

Als Konsolen-Geräte unterstützt UML *xterm*, *tty's*, serielle Anschlüsse des Host-Systems sowie *Pseudo Terminals* (*pty's*).

2.1.2 Hilfsprozesse

Um alle Funktionen von UML nutzen zu können, müssen auf dem Host-System verschiedene Hilfsprozesse installiert sein. Die folgenden so genannten *UML utilities* werden einmalig installiert und von allen laufenden UML-Instanzen benutzt:

- **uml_mconsole:** Dient dazu, eine UML-Instanz anzuhalten, neu zu starten, Statusinformationen abzufragen oder die Konfiguration ohne einloggen zu ändern. Dieses Hilfsprogramm wird bei Bedarf aufgerufen und anschließend wieder beendet.
- **uml_net:** Dieses Daemon-Programm wird im Hintergrund ausgeführt und stellt die Verbindung zwischen virtuellen Ethernet-Schnittstellen der UML-Instanzen und *tap*-Schnittstellen des Host-Systems her.
- **uml_switch:** Wird ebenfalls im Hintergrund ausgeführt und arbeitet als virtueller Switch für die virtuellen Netzwerkschnittstellen aller UML-Instanzen

2.1.3 Vorteile von UML

Neben der komfortablen Möglichkeit, den Linux-Kernel als Anwenderadressraum-Prozess einfach zu debuggen, bietet sich User-Mode Linux auch als kostengünstiger Ersatz für reale Rechnerhardware an. Eine UML-Instanz im SKAS-Mode mit *tmpfs* als Dateisystem für den virtuellen Speicher und moderaten Festplattenzugriffen hat unter Vollast weniger als 20% Leistungsverlust gegenüber der realen Hardware im Host-System.

Im Gegensatz zum frei-verfügbaren UML emuliert das kommerzielle VMware [12] eine vollständige Hardware-PC-Architektur, woraus sich zunächst höhere Anforderungen bzgl. Speicherausstattung und Rechenleistung des Host-Systems ergeben. Zudem liegt derzeit der Preis je VMWare Server-Lizenz, welche zum Aufbau umfangreicher Netztopologien zwingend benötigt wird, bei mehreren Tausend US-Dollar. Die einfache Integration von Debug- und Testmöglichkeiten (etwa in uml_switch) ist zudem aufgrund des nicht verfügbaren Quellcodes nur eingeschränkt möglich.

Konfigurationen mit verschiedenen Softwareinstallationen auf verschiedenen Hardwareausprägungen (Anzahl der Netzwerkschnittstellen, Größe der Festplatte) lassen sich durch den virtu-

[2] mittels der *ptrace*-Funktion

ellen Charakter von UML schnell und einfach erzeugen. In der Linux-Entwicklerszene ist UML deshalb sehr beliebt und wird u.a. für die Entwicklung von *netfilter*[3] benutzt.

Die Entwicklung von User-Mode Linux geht, getragen von einer aktiven Community, kontinuierlich weiter. Besonders die Leistung und die Hardwareabstraktion werden ständig verbessert. User-Mode Linux wird als auswählbare Plattform in den nächsten stabilen Linux-Kernel (Version 2.6) aufgenommen.

2.2 Multiprotocol Label Switching

Multiprotocol Label Switching (MPLS) wurde Ende der 90er Jahre entwickelt und zielt auf eine beschleunigte/konfigurierbare Paketweiterleitung vornehmlich in Backbone-Netzen. Während Mitte der 90er Jahre noch eine Reihe verschiedener Hersteller wie Cisco, IBM und Toshiba eigenständige Lösungen propagierten, wurde MPLS mittlerweile im Rahmen der IETF (Internet Engineering Task Force) einer weitgehenden Standardisierung unterzogen [11].

Die grundlegende Idee von MPLS besteht darin, das IP-basierte Routing auf Schicht 3 im Kern eines Netzwerkes durch schnelleres Switching auf Schicht 2 zu ersetzen. Hierzu kommen sog. *Labels* fester Länge (normalerweise 20 bit) zum Einsatz, die zwischen den Paketköpfen der Schicht 2 und 3 eingefügt werden. Am Eingang einer MPLS-Domäne werden eingehende (IP-)Pakete durch sog. *Label Edge Router (LERs)* klassifiziert und in verschiedene Äquivalenzklassen (Functional Equivalence Classes, FECs) eingeteilt. Diese Klassen werden wiederum auf Pfade im Netzwerk abgebildet. Innerhalb der MPLS-Domäne leiten *Label Switch Router (LSRs)* die Pakete nur noch anhand des angefügten Labels weiter, ohne die Informationen der höheren Schichten zu betrachten. Während des Vorgangs der Paketweiterleitung wird dabei das alte (eingehende) Label durch ein oder mehrere neue (ausgehende) Label ersetzt *(Label Switching)* – der resultierende Pfad wird auch als Label Switched Path (LSP) bezeichnet. Zuständig für die Verarbeitung der MPLS-Label ist die sog. MPLS-Weiterleitungskomponente, welche auf einem *Label Swapping Algorithmus* zum Setzen und Austauschen der Label von eingehenden und ausgehenden Paketen basiert.

Am Rand einer MPLS-Domäne (Egress) werden sämtliche Labels entfernt und das Paket anhand des konventionellen IP-Routings weitergeleitet. Das Aufsetzen der Äquivalenzklassen, Pfade und Label-Abbildungen erfolgt durch die so genannte MPLS-Kontrollkomponente. Entweder manuell oder durch Nutzung spezieller Signalisierungsprotokolle, wie dem LDP (Label Distribution Protocol), die in Verbindung mit Routingprotokollen (wie OSPF oder BGP) die notwendigen Wege dynamisch bestimmen, wird die von der MPLS-Weiterleitungskomponente verwendete Weiterleitungstabelle aufgebaut und gepflegt.

Seit Anfang des Jahres 2000 existiert das von James R. Leu ins Leben gerufene *MPLS for Linux*-Projekt, das mittels eines Patches für den aktuellen Linux-Kernel 2.4 MPLS-Funktionen implementiert [8]. Der Patch umfasst ca. 5600 Zeilen Code und implementiert MPLS vollständig wie in RFC 3031 [11] und RFC 3032 [10] spezifiziert.

3 Grundlegender Aufbau der Simulationsumgebung

Im Rahmen der Forschungsarbeiten am Institut für Telematik wurde mittels UML-Instanzen ein virtuelles Testnetz aufgebaut, das Tests des MPLS-Netzwerk-Stacks unter Linux ermöglicht.

Das simulierte Netz wurde mit dem virtuellen Switch *uml_switch* vernetzt und besitzt die in Abbildung 1 dargestellte Topologie bzw. die dargestellten IP-Adressen. Jeder LSR hat für Administrationszwecke eine Netzwerkverbindung zum Host-System, die über *TUN/TAP*-Schnittstellen realisiert ist. Der Router *LSR1* hat als Ingress-Router eine zusätzliche Schnittstelle zur MPLS-Domäne und der Egress-Router *LSR7* zwei Schnittstellen, über die die MPLS-Domäne verlassen

[3] [http://www.netfilter.org/]

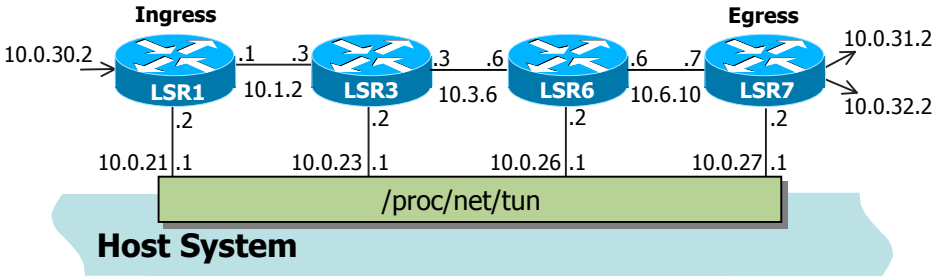

Abbildung 1: Topologie des simulieren Netzes

werden kann. Da LDP in RFC 3036 nicht für *Multipath* spezifiziert ist, wurde die dargestellte einfache Struktur gewählt. Alle Netze haben die Netzmaske 255.255.255.0.

Zusätzlich zu LDP kommt OSPF als Routingprotokoll zum Einsatz, sowie eine nach einem aktuellen Draft *(draft-ietf-mpls-lsp-ping-01.txt)* [6] der IETF implementierte Funktion *mplsping*. Diese gliedert sich in den Anwenderadressraumteil *mplsping* und *mplspong_daemon* sowie den Kernel-Modulen *mplspong* und *mpls_queue*. Der freie Routingdaemon *Zebra* [5] mit Plugins für OSPF und LDP wird ebenfalls auf allen UML-Instanzen ausgeführt.

Die so geschaffene Simulationsumgebung ist unabhängig von eventuell vorhandenen Limitierungen realer Netzwerkschnittstellen und den Verbindungen zwischen diesen, verhält sich aber ansonsten exakt wie ein reales Netzwerk, inkl. sämtlicher protokolltechnischer Abläufe unter Verwendung des bestehenden Linux-Protokollstacks. Im Gegensatz zu dem in dieser Arbeit verfolgten Ansatz, jeden einzelnen LER/LSR durch Nutzung von UML als vollständige und eigenständige Rechnerinstanz zu simulieren und so eine einfache Integration bestehender Software und Protokolle zu ermöglichen, implementiert der MPLS-Simulator des *Lime Projects* [4] die verwendeten LSRs als Kernel-Module unter Verwendung einer speziellen Programmierschnittstelle (API). Der *Network Simulator NS2* [9] abstrahiert das virtuelle Netzwerk noch weiter und simuliert es ausschließlich durch die Interpretation einer eigenen Beschreibungssprache.

4 Konfiguration der Simulationsumgebung und durchgeführte Tests

Im Folgenden wird in Abschnitt 4.1 der Aufbau des Host-Systems und in Abschnitt 4.2 der Aufbau der UML-Instanzen beschrieben. Nach der in Abschnitt 4.3 beschriebenen Initialisierung der Simulationsumgebung wird in Abschnitt 4.4 ein funktionaler Test und in Abschnitt 4.5 ein Leistungstest dargestellt.

4.1 Hostsystem der Simulationsumgebung

Das Host-System für die UML-Instanzen basiert auf einem Rechner mit einer 900MHz Pentium III CPU und 256 MB RAM. Auf einer IDE/UDMA-Festplatte bietet eine 7GB große *ext3*-Partition ausreichend Platz für die installierte Linux-Distribution *Gentoo 1.4* und die virtuellen Festplatten der UML-Instanzen. Ein Linux-Kernel der Version 2.4.20 mit SKAS-Patch[4] wird als Betriebssystem benutzt.

Neben verschiedenen Compilern und Werkzeugen ist auch die grafische Oberfläche *X Windows* und das Konsolen-Multiplexer Programm *screen*[5] installiert.

[4] [http://prdownloads.sourceforge.net/user-mode-linux/host-skas3.patch]
[5] [http://www.math.fu-berlin.de/∼guckes/screen/]

Die in Abschnitt 2.1 erläuterten UML-Utilities sind in der Version *20030202* installiert, zudem *Ethereal* [1] in der Version 0.9.9 als komfortables und weit entwickeltes Werkzeug zur Netzverkehrsanalyse mit grafischer Benutzeroberfläche.

Als Basis für den User-Mode Linux Kernel wird der Quellcode der Version 2.4.19 verwendet. Folgende Patches wurden in dieser Reihenfolge angewandt:

1. **User-Mode Linux:** *uml-patch-2.4.19-51.bz2* [3]
2. **MPLS for Linux:** *CVS Branchtag: mpls-linux_1_172* [8] mit den Schnittstellen *mpls_in*, *mpls_out* und *mpls_labelspace* im *proc*-Dateisystem
3. **MPLS-Ping Funktion:** der aus der Implementierung von *mplsping* resultierende Patch für den Linux-Kernel.

Der UML-Kernel wurde auf dem oben beschriebenen Host-System mit *GCC 3.2.2* und *glibc 2.3.2* mit dem Aufruf `make clean dep linux modules ARCH=um` kompiliert.

4.2 User-Mode Linux Instanz

Grundlage für jede UML-Instanz ist eine auf der virtuellen Festplatte installierte Linux-Distribution. *Vector Linux* [7] als kleine, aber vielseitig einsetzbare Distribution, wurde als Basissystem ausgewählt. Sie enthält neben *X Windows* alle notwendigen Programme. Komplett installiert belegt die verwendete Version 2.5 153MB.

Da mehrere identische UML-Instanzen benötigt werden, wird das COW-System von UML benutzt. *Vector Linux* wird im Read-Only-Teil installiert, die spezifischen Konfigurationen und Dateiänderungen werden während der Laufzeit im beschreibbaren Teil abgelegt.

Die Konfiguration für die UML-Instanzen und den darin ausgeführten Programmen werden mittels dem Concurrent Versions System (CVS)[6] verwaltet. Editieren und Vergleichen von Konfigurationsdateien verschiedener UML-Instanzen wird damit stark vereinfacht. Zusätzlich lassen sich die Konfigurationen über so genannte *CVS-Branches* leicht austauschen und Änderungen verfolgen. Folgende Einstellungen werden mit CVS verwaltet:

- **start_conf.source:** für alle UML-Instanzen wird in dieser Konfigurationsdatei der verwendete Kernel, verwendetes RAM, die Konsoleneinstellungen und die Parameter für die Anwendung *nice*[7] angegeben.

Folgende Konfigurationsdateien sind spezifisch für jede UML-Instanz:

- **HOSTNAME:** der Rechnername der UML-Instanz wird in dieser Konfigurationsdatei hinterlegt.
- **rc.inet1:** die lokale Netzwerkschnittstelle zum Host-System wird in dieser Datei gespeichert. Das Skript *rc.inet1* wird beim Starten der UML-Instanz ausgeführt. Dies ermöglicht das Einbinden von individuellen Befehlen für jede UML-Instanz.
- **zebra.conf:** die Konfigurationsdatei für Zebra legt die IPv4-Adressen für alle benutzten Netzwerkschnittstellen fest.
- **ospf.conf:** die OSPF-Konfiguration für jeden LSR im Testnetz wird in dieser Konfigurationsdatei festgehalten.
- **mplsd.conf:** mit den LDP-Einstellungen für das Zebra-Plugin *mplsd* sind alle spezifischen Konfigurationen für die UML-Instanzen abgeschlossen.

[6] [http://www.cvshome.org]

[7] Mit dem Befehl *nice* lässt sich Scheduler-Priorität eines Programms verändern.

4.3 Initialisieren des Simulationsnetzwerkes

Die UML-Instanzen des Testnetzes werden mit Hilfe des Skriptes *image1.sh* gestartet und gestoppt. Das Skript testet, ob einen virtueller Switch gestartet ist, das TUN/TAP Kernel-Modul geladen ist, ausreichend tmpfs-RAM zur Verfügung steht und *netfilter*-Module geladen sind, die die Netzwerkverbindung zum Host-System beeinträchtigen könnten. Zusätzlich stellt das Skript Funktionen zum Erzeugen der Kommandozeilenparameter für den UML-Kernel aus den UML-spezifischen Konfigurationsdateien und der globalen Konfigurationsdatei *start_conf.source* zur Verfügung. UML-Instanzen können mit *image1.sh* einmalig oder in einer Endlosschleife gestartet bzw. gestoppt werden.

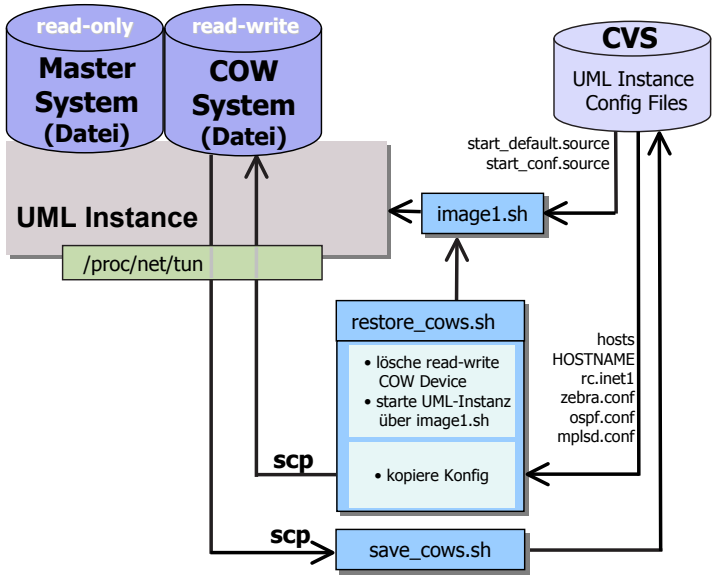

Abbildung 2: Verarbeitung von Konfigurationsdaten einer UML-Instanz mit *image1.sh*, *restore_cows.sh* und *save_cows.sh*

In Abbildung 2 ist folgender Ablauf verdeutlicht:

1. Im ersten Schritt wird die Read-Write Datei des COW-Systems gelöscht.
2. Mit dem Skript *image1.sh* wird eine Master-UML-Instanz mit fester IP Adresse und leerem Read-Write File gestartet.
3. Nachdem die Master-Instanz gestartet ist, werden die spezifischen Konfigurationsdateien mit Hilfe von *secure copy* (scp) kopiert, so dass diese beim nächsten Neustart der UML-Instanz aktiv sind. Abschließend wird die Ausführung der UML-Master-Instanz mit dem Befehl `halt` beendet.

Falls erforderlich, können die spezifische Konfigurationsdateien mit dem Skript *save_cows.sh* aus den UML-Instanzen wieder in das CVS-System gesichert werden. Unter Verwendung der oben genannten Skripte kann das Testnetzwerk in weniger als 30 Minuten aufgebaut werden.

Abbildung 3 zeigt eine mit Hilfe von CVS und den erläuterten Skripten konfigurierte UML-Instanz, die über den virtuellen Switch *uml_switch* mit dem simulierten Testnetz verbunden ist. Schematisch angedeutet sind die verwendeten Netzwerkprotokolle *OSPF*, *LDP*, *ICMP* und *MPLS-Ping*. Die Schnittstelle */proc/net/tun* wird als Netzwerkverbindung zum Host-System benutzt. Über */proc/mm* greift der Kernel der UML-Instanz auf den für ihn reservierten Speicher (tmpfs-RAM) zu.

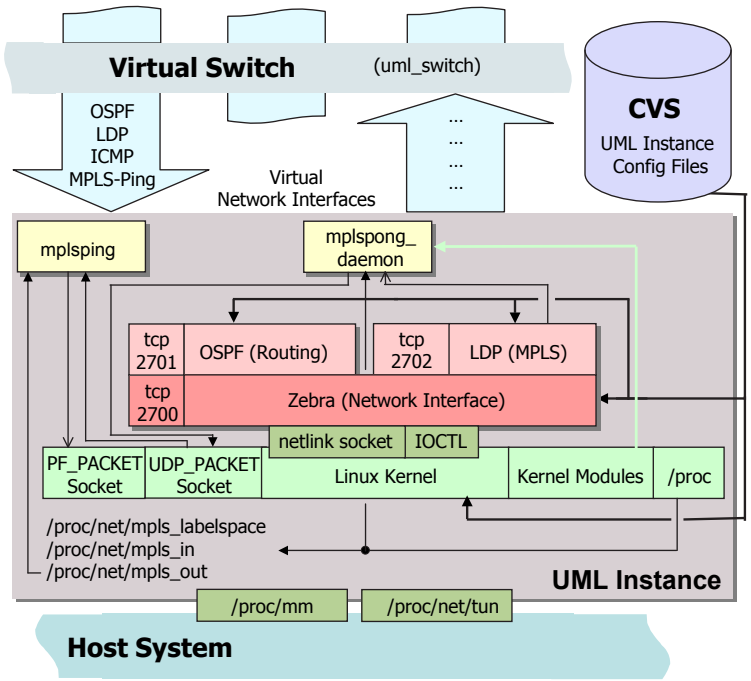

Abbildung 3: Verbindungen einer UML-Instanz mit dem virtuellen Switch des Testnetzes

Die zu testende Anwendung *mplsping* versendet über einen *PF_PACKET*-Socket `echo request`-Pakete und empfängt über einen UDP-Socket `echo reply`-Antwortpakete. Über die */proc*-Schnittstelle werden Label- und Netzwerkschnittstellen-Informationen aus *mpls_out* für das Erzeugen der `echo request`-Pakete abgefragt. Die Anwendung *mplspong_daemon* liest über die Kernel-Module *mplspong* und *mpls_queue* bereitgestellte `echo request`-Pakete und erzeugt mit Informationen aus dem LDP-Plugin für Zebra das `echo reply`-Antwortpaket. Ein UDP-Socket wird für den Versand des Antwortpaketes benutzt.

Mit der in *Vector Linux* enthaltenen Anwendung *telnet* kann auf die Administrationsschnittstelle des Routing-Daemon Zebra (TCP-Port 2700) und der Plugins für OSPF (TCP-Port 2701) und LDP (TCP-Port 2702) zugegriffen werden.

4.4 Funktionstest des simulierten MPLS-Netzwerkes

Mit *mplsping* werden zwei MPLS-Ping-Pakete mit Label `10016` und FEC `10.6.10.0/24` versendet. Die maximale Wartezeit für `echo reply`-Pakete wird auf 3 Sekunden gesetzt:

```
LER-1:~$ /tmp/mplsping -c 2 -L 10016 -T 10.6.10.0/24 -w 3
Target FEC: 10.6.10.0/24
MPLSPING 10.1.2.3 (127.178.79.210):
sent mplsping #01 with label 10016 (FEC 10.6.10.0/24) to 10.1.2.3
(TTL=255)
53 bytes from 10.3.6.6 seq=1 time=14.967(13.646)ms [egress]
sent mplsping #02 with label 10016 (FEC 10.6.10.0/24) to 10.1.2.3
(TTL=255)
53 bytes from 10.3.6.6 seq=2 time=15.48(13.765)ms [egress]
--- 127.178.79.210 mplsping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
```

Die IP-Zieladresse im UDP-Paket wird, wie im betrachteten Draft der IETF vorgeschrieben, auf einen zufälligen Wert aus dem 127.0.0.0/8-Netz gesetzt. LSR6 antwortet mit dem *Return Code* 3 (*Replying router is an egress for the FEC*). Die angegebenen Paketlaufzeiten sind analog zum ICMP-Ping berechnet und sowohl als Roundtrip-Zeit als auch (in Klammern) als One-Way-Zeit angegeben.

Der erfolgreiche Test von *mplsping* zeigt, dass die durch UML-Instanzen aufgebaute virtuelle MPLS-Simulationsumgebung mit dem virtuellen Switch *uml_switch* funktionsfähig ist und korrekt arbeitet.

Zusätzliche MPLS-Statusinformationen können auf den jeweiligen LSRs über */proc/net/mpls_in* und */proc/net/mpls_out* sowie die *telnet*-Schnittstellen von *Zebra* abgefragt werden. Mit den Debug-Ausgaben von *mplsping* und *mplspong_daemon* kann zusätzlich leicht die korrekte Funktion der MPLS-Kontrollkomponente (in diesem Testbeispiel der Routingdaemon Zebra mit den Plugins für OSPF und LDP) überprüft werden.

4.5 Leistungsmessung des Host-Systems mit einem simulierten Netzwerk

Mit einer Messung wurde die Leistungsfähigkeit des simulierten Netzwerkes bestimmt. Wichtig für das simulierte Testnetzwerk ist dabei, dass bedingt durch die Simulation selbst keine Änderungen im Verhalten des Netzes entstehen. Da das Testnetzwerk mit vier LSRs zu klein für einen aussagekräftigen Leistungstest ist, wurde ein größeres MPLS-Netzwerk mit sieben LSRs benutzt.

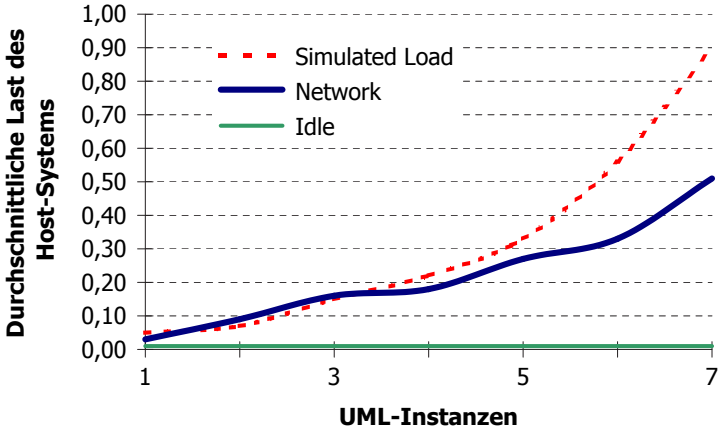

Abbildung 4: Durchschnittliche Last auf dem Host-System

Abbildung 4 zeigt die durchschnittliche Last des in Abschnitt 4 beschriebenen Host-Systems in Abhängigkeit von der Anzahl der ausgeführten UML-Instanzen. Die Kurve mit der Bezeichnung *Idle* entspricht der erzeugten Last von UML-Instanzen, die nur *crond*, *syslogd* und *sshd* ausführen. Im Test *Network* sind zusätzlich der Routing-Daemon *Zebra* mit den Plugins für OSPF und LDP sowie ein Hilfsskript, welches regelmäßig ICMP-Ping Pakete versendet, gestartet.

Bei dem mit *Simulated Load* gekennzeichneten Test wurde zusätzlich in jeder UML-Instanz ein *top*-Prozess mit einem Intervall von 1 Sekunde als Lastsimulator gestartet. In allen drei Tests war auf dem Host-System der virtuelle Switch *uml_switch* aktiv.

Die maximal erreichte *Last* von 0.9 besagt, dass im betrachteten Zeitintervall (5 Minuten) durchschnittlich 0.9 Prozesse in der Ausführungswarteschlange des Host-System-Kernels auf Ausführung gewartet haben. Dies bedeutet, dass das Host-System auch im *Simulated Load*-Test

nicht zu 100% ausgelastet war. Da auch eine Last von 5 und mehr eine effiziente Netzsimulation zulässt (vorläufige Tests legen eine Verwendbarkeit des Simulationsszenarios für mehrere Dutzend UML-Instanzen nahe), stellt die entwickelte Testumgebung eine äußerst effiziente Möglichkeit zum Aufbau umfangreicher, praxisnaher Szenarien dar.

5 Zusammenfassung

Die in dieser Arbeit vorgestellte neuartige Simulationsumgebung für MPLS-Netzwerke verdeutlicht die Eignung von User-Mode Linux (UML) für komplexe Netztopologien unter Integration verschiedenster Protokolle (MPLS, OSPF, LDP). Wie das beschriebene Beispiel des *mplsping* zeigt, machen die umfangreichen Analysemöglichkeiten die Simulationsumgebung zur optimalen Plattform für funktionale und leistungsbezogene Tests neuer Protokollimplementierungen. Ein besonderer Vorteil liegt dabei in der Eigenständigkeit der *virtuellen* Instanzen, die als vollwertige Rechner- und Netzwerkknoten von „echten“ Systemen praktisch nicht zu unterscheiden sind und den schnellen und kostengünstigen Aufbau neuer virtueller Netzwerktopologien ermöglichen.

Als Ansatzpunkte für zukünftige Arbeiten ist die Erweiterung der Simulationsumgebung um zusätzliche Protokolle (z.B. RSVP-TE, BGP) und Managementfunktionalitäten der einzelnen UML-Instanzen (SNMP) vorgesehen.

Literatur

[1] G. Combs. *Ethereal*. 2003. Weitere Informationen unter http://www.ethereal.com/.

[2] J. Dike. A User-Mode Port of the Linux Kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference (ALS) 2000*, Atlanta, Georgia, USA, 10.–14. October 2000.

[3] J. Dike. *User Mode Linux Project*. 2003. Weitere Informationen unter http://user–mode–linux.sourceforge.net/index.html.

[4] A. Gadgil and A. Karandikar. *LIME - A Linux based MPLS Emulator*. 2003. Weitere Informationen unter http://www.ee.iitb.ac.in/uma/ mpls/.

[5] K. Ishiguro. *Zebra*. 2003. Weitere Informationen unter http://www.zebra.org.

[6] K. Kompella, P. Pan, D. Cooper, et al. Detecting MPLS Data Plane Liveliness. Internet-Draft, Oct. 2002.

[7] R. Lange. *Vector Linux*. 2003. Weitere Informationen unter http://www.vectorlinux.com.

[8] J. Leu. *MPLS for Linux Project*. 2003. Weitere Informationen unter http://mpls-linux.sourceforge.net/.

[9] *The Network Simulator - NS 2*. 2003. Weitere Informationen unter http://www.isi.edu/nsnam/ns/.

[10] E. Rosen, D. Tappan, G. Fedorkow, et al. *MPLS Label Stack Encoding*. 2001. RFC 3032.

[11] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture*. 2001. RFC 3031.

[12] VMware. *VMware GSX Server 2.5*. 2003. Weitere Informationen unter http://www.vmware.com/.