Bedienungsanleitung zum Simulationsscript zimulator.pl

Marcel Jakobs

22. Februar 2010

Inhaltsverzeichnis

1 Einleitung

Das Perl-Script „zimulator.pl“ ermöglicht die Autmatisierung von Simulationen des RIP-Protokolls unter VNUML. Die Konvergenzeigenschaften der Simulationen können dabei automatisch ausgewertet werden. Dabei kann der Ablauf der Simulation über eine Konfigurationsdatei sehr genau beschrieben werden. Die Messungen der Konvergenzeigenschaften werden durch die Analyse von automatisch generierten TCP-Dumps nach der eigentlichen Simulation durchgeführt. Daneben bietet das Script noch Optionen zum schnellen Erzeugen von VNUML-Konfigurationsdateien mittels einer eigenen Konfigurationsdatei oder per Zufall nur über die Angabe der Anzahl der Netze und Router. Darüber hinaus können Netzwerke (als png-Dateien) visualisiert werden und einige Eigenschaften der zugrunde liegenden Graphen berechnet werden.

1.1 Wichtige Hinweise

Die Simulationen werden in der Regel mit root-Rechten ausgeführt. Von daher sollte man stets Vorsicht walten lassen.

Dieses Script wurde nur unter Linux getestet.

Die neuesten Versionen gibt es im bei GitHub: `http://github.com/zimon/zimulator`

Hinweise, Fragen ... können an mich per Email (zimon@uni-koblenz.de) gerichtet werden.

2 Bedienungsanleitung

Mit dem Perl-Programm `zimulator.pl` ist es möglich, Simulationen mit VNUML-Netzwerken zu automatisieren und automatische Konvergenzmessungen durchzuführen.

2.1 Übersicht

Um Simulationen zu automatisieren, werden drei Konfigurationsdateien benötigt:

- Eine *Topologiebeschreibung* in Form einer ZVF-Datei (siehe auch Seite 9).
- Ein *Test-Script*, in dem die einzelnen Simulationsschritte definiert sind, als .cfg-Datei (siehe auch Seite 12).
- Eine *Ausführungsbeschreibung*, die beschreibt, welche Topologien mit welchen Test-Scripten wie oft simuliert werden sollen (siehe auch Seite 16).

Bei allen Dateien werden Leerzeilen und Zeilen, die mit `#` beginnen (Kommentare) ignoriert. Die zugehörige VNUML-XML-Konfigurationsdatei (*Szenario-Datei*) wird automatisch erstellt, falls sie nicht bereits vorhanden ist.

Eine *Simulation* ist eine Kombination aus Topologiebeschreibung und Test-Script. Das Simulieren einer Simulation wird als *Durchlauf* (bzw. Run) bezeichnet. Das Ergebnis eines Durchlaufs (also dessen Konvergenzzeit, gesendete Pakete, ...) wird in einer *Ergebnisdatei* (resultfile) gespeichert. Bei mehreren Durchläufen wird das Ergebnis jeweils an die Datei angehängt, wobei jede Zeile einem Durchlauf entspricht. Die Ergebnisdatei wird zusammen mit tcpdump-Dateien und anderen erstellten Dateien für diese Simulation in einem Ordner (*Simulationsordner*) gespeichert. Der Name des Ordners wird aus den Namen des Test-Scripts und der Topologiebeschreibung zusammengesetzt (ohne Endungen und mit Bindestrich getrennt).

Funktionen des Programms

Im Folgenden werden alle Funktionen von `zimulator.pl` mit ihren Optionen aufgeführt.

Die Datei `zimulator.pl` verarbeitet die übergebenen Argumente und ruft die benötigten Funktionen auf.

Syntax: `./zimulator.pl MODE [FILE] [OPTIONS] [FILE(S)]`

Die Modi des Programms:

`-s FILE` – Simuliert die Simulationen, die in der angegebenen Ausführungsbeschreibung aufgeführt sind.

`-S [-T] [-F] [-o OUTPUT_FILE] TOPOLOGY_FILE(S)` – Berechnet die Konvergenzeigenschaften für alle Durchläufe aller Simulationen einer Topologie.

`-r ZVFFILE [-T] [-F] [-o OUTPUT_FILE] DUMP_FILE(S)` – Berechnet die Konvergenzeigenschaften für die angegebenen tcpdump-Dateien zu einer gegebenen Topologie.

`-a RESULT_FILE(S)` – Errechnet die durchschnittlichen Konvergenzzeiten von allen angegeben Ergebnisdateien.

`-z [-i] TOPOLOGY_FILE(S)` – Analysiert die Topologie des Netzwerks und schreibt die Eigenschaften des Graphen in die ZVF-Datei (es wird auch eine PNG-Datei angelegt). Kommentare in der ZVF-Datei gehen verloren

`-x TOPOLOGY_FILE(S)` – Erzeugt VNUML xml-Dateien für alle angegebenen Topologiebeschreibungen.

`-g TYPE [-o OUTPUT_FILE] [-i] ARGUMENTS` – Generiert einen Graphen des Typs TYPE [1]. Es wird eine ZVF-Datei mit den Eigenschaften des Graphen erzeugt.

`-C TOPOLOGY_FILE(S)` – Überprüft die angegebene ZVF-Datei auf Sytaxfehler.

`-H FILE(S)` – Gibt angegebene tcpdump-Dateien oder Ergebnisdateien in von Menschen lesbarer Form aus.

[1] Die möglichen Typen und ihre Argumente sind in Kapitel 2.6 auf Seite 24 beschrieben.

`-h` – Gibt die Hilfe aus.

Mögliche Optionen:

`-v` – Gibt zusätzliche Informationen aus.

`-c CONFIG_FILE` – Benutzt die angegebene Konfigurationsdatei

`-o OUTPUT_FILE` – Schreibt alle Ausgaben in die angegebe Datei (die Datei „-“ bezeichnet die Standardausgabe).

`-T TIME` – Zieht nur Pakete zur Konvergenzberechnung heran, die vor dem Zeitpunkt TIME versendet wurden (im Modus `-S` wird die Zeitangabe ignoriert, da diese aus der Ergebnisdatei entnommen werden wird).

`-F TIME` – Gibt den Startzeitpunkt für die Konvergenzberechnung an (im Modus `-S` wird die Zeitangabe ignoriert, da diese aus der Ergebnisdatei entnommen werden wird).

`-i` – Generiert ein PNG-Bild der Topologie(n).

2.2 Installation und Konfiguration

Im Folgenden werden die einzelnen Schritte der Installation und Konfiguration von VNUML und der Simulationsumgebung `zimulator.pl` auf Debian-basierten Linuxdistributionen beschrieben. Getestet wurde `zimulator.pl` unter Ubuntu 9.04, Ubuntu 9.10 und Fedora 11.

Installation von VNUML

Als erstes muss das benötigte Paket `bridge-utils` mit dem Befehl aus Quelltext 1 installiert werden.

```
sudo apt-get install bridge-utils
```

Quelltext 1: Installation der bridge-utils

Dieses wird benötigt, um die virtuellen Netze an das Hostsystem weiterleiten zu können damit entsprechende tcpdump-Dateien vom Hostssystem aus arbeiten können.

Als nächstes wird VNUML installiert. Dazu wird mit root-Rechten die Datei `/etc/apt/sources.list` editiert und folgende Zeile aus Quelltext 2 hinzugefügt:

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

Quelltext 2: VNUML Debian Repository

Die drei Befehle aus Quelltext 3 installieren dann VNUML und den UML-Kernel.

```
sudo apt-get update
sudo apt-get install vnuml
sudo apt-get install linux-um
```

Quelltext 3: Installation von VNUML

Nun kann man von der Seite http://www.dit.upm.es/vnumlwiki/index.php/Download ein Dateisystem herunterladen. Im Abschnitt „Root filesystems“ werden zwei Dateisysteme angeboten, wobei sich das kleinere („minimal root_fs“) für größere Simulationen besser eignet, da für jede simulierte Maschine eine Kopie des Dateisystems in den Arbeitsspeicher geladen wird. Um den RMTI-Algorithmus zu verwenden, kann man stattdessen auch das Dateisystem von Frank Bohdanowicz von der Seite http://www.uni-koblenz.de/~bohdan/vnuml/ nutzen. Das Dateisystem wird dann mit root-Rechten unter einem geeigneten Namen in das Verzeichnis `/usr/share/vnuml/filesystems` kopiert. Das Kommando in Quelltext 4 kopiert das minimal root_fs (die Versionsnummer und damit der Dateiname kann sich mit der Zeit ändern) unter dem Namen `mini_fs` in das entsprechende Verzeichnis.

```
sudo cp n3vlr-0.11-vnuml-v0.1.img /usr/share/vnuml/filesystems/mini_fs
```

Quelltext 4: Kopieren des Dateisystems

Schließlich werden noch die Konfigurationsdateien für Zebra und RIP benötigt. Dafür wird ein Verzeichnis `conf` erstellt, in dem die beiden Dateien `ripd.conf` (siehe Quelltext 5) und `zebra.conf` (siehe Quelltext 6) liegen.

```
hostname zebra
password xxxx
enable password xxxx
```

Quelltext 5: Zebra-Konfigurationsdatei

```
hostname ripd
password xxxx
router rip
network 10.0.0.0/8
timers basic 10 30 20
```

Quelltext 6: RIP-Konfigurationsdatei

Installation und Konfiguration von zimulator.pl

Das Programm `zimulator.pl` benötigt einige CPAN-Bibliotheken sowie das Programm GraphViz[2]. Diese Pakete können mit dem Befehl aus Quelltext 7 installiert werden.

```
sudo apt-get install graphviz libgraphviz-perl
```

Quelltext 7: Installieren der Abhängigkeiten von zimulator.pl

[2] http://www.graphviz.org/

Die Bibliothek `Graph` muss über CPAN installiert werden (siehe Quelltext 8), da der Dijkstra-Algorithmus der Ubuntu-Version (Paket libgraph-perl unter Ubuntu 9.04 und Ubuntu 9.10) nicht funktioniert. Ebenso muss die Bibliothek `Test::More` über CPAN installiert werden, wenn man die dem Programm beiliegenden Tests starten möchte.

```
sudo cpan
install Graph
install Test::More
exit
```

Quelltext 8: CPAN-Installation der Abhängigkeiten von zimulator.pl

Beim ersten Start von CPAN werden einige Fragen gestellt. Die meisten Fragen kann man mit ENTER bestätigen. Man sollte jedoch den Kontinent, das Land und den zu verwendenden Server angeben. Es kann auch sein, dass alles automatisch eingerichtet wird.

Das Programm kann man unter http://github.com/zimon/zimulator beziehen (indem man auf den Link „Download Source“ klickt). Im während des Entpackens erstellten Ordner befindet sich dann das Programm mit einigen Beispieldateien.

Nun sollte man zuerst die wichtigsten Konfigurationsparameter des Programms einrichten. Dazu öffnet man in einem Editor die Konfigurationsdatei `.zimulatorrc` (sie befindet sich im gleichen Verzeichnis wie das Programm). Darin werden die wichtigsten Einstellungen festgelegt wie Pfade zu verschiedenen Programmen und Dateien. Die Konfigurationsdatei wird beim Start des Programms automatisch mit den Standardwerten erstellt, falls sie noch nicht existiert.

Eine gängige Konfiguration ist in Quelltext 9 aufgeführt (hier wird das „minimal root_fs“-Image genutzt, welches wie auf Seite 6 beschrieben „mini_fs“ genannt wurde). Der erste Abschnitt legt fest, wie aus ZVF-Dateien die Topologiebeschreibungen generiert werden. Im zweiten Abschnitt werden die Parameter für die VNUML-Aufrufe festgelegt. Der dritte Abschnitt definiert sonstige Variablen zur Steuerung des Programms. Alle Variablen sind mit Kommentaren versehen.

```
## Konfiguration fuer XML-Generierung

# Pfad zur DTD-Datei.
DTDPATH = "/usr/local/share/xml/vnuml/vnuml.dtd"

# Pfad zum oeffentlichen SSh Schluessel
SSH_KEY = "/root/.ssh/id_rsa.pub"

# Addresse des Management-Netzes
MANAGEMENT_NET = "192.168.0.0"

# Netzmaske des Management-Netzes
MANAGEMENT_NETMASK = "24"

# Offset, das auf jede IP aufaddiert wird
MANAGEMENT_NET_OFFSET = "100"

# Attribute fuer das Tag "vm_defaults" (Standard: " exec_mode=\"mconsole\"")
VM_DEFAULTS = "exec_mode="mconsole""
```

```
# Pfad zum VNUML-Dateisystem
FILESYSTEM = "/usr/local/share/vnuml/filesystems/mini_fs"

# Pfad zum VNUML-Kernel
KERNEL = "/usr/local/share/vnuml/kernels/linux"

# Typ der virtuellen Netze (Standard: "virtual_bridge")
NET_MODE = "virtual_bridge"

# Pfad zu Zebra im VNUML-Dateisystem (nur der Pfad ohne / am Ende)
ZEBRA_PATH = "/usr/lib/quagga"

# Pfad zu RIP im VNUML-Dateisystem (nur der Pfad ohne / am Ende)
RIPD_PATH = "/usr/lib/quagga"

# Pfad zu OSPF im VNUML-Dateisystem (nur der Pfad ohne / am Ende)
OSPF_PATH = "/usr/lib/quagga"


## Konfiguration fuer VNUML Steuerung

# Pfad zum VNUML-Programm (nur der Pfad ohne / am Ende)
VNUML_PATH = "/usr/local/bin"

# Parameter zum Starten von VNUML
VNUML_START_PARAMETERS = "-w 300 -Z -B -t"

# Parameter zum Ausfuehren von Tags
VNUML_EXEC_PARAMETERS = "-x"

# Parameter zum Beenden von VNUML
VNUML_STOP_PARAMETERS = "-P"


## Sonstige Konfigurationsvariablen

# Maximale Anzahl der Fehler bis Abbruch der Simulation,
# falls dies nicht in der Aufuehrungsbeschreibung angegeben ist
MAXFAIL_DEFAULT = "5"

# Maximale Anzahl der Durchlaeufe falls dies nicht in der
# Aufuehrungsbeschreibung angegeben ist
MAXRUN_DEFAULT = "15"

# Datei, in die VNUML-Ausgaben gespeichert werden
# "/dev/null/" um keine Logdatei zu verwenden
LOGFILE = "logfile.log"

# 1 um tcpdump-Dateien im RAW-Format zu speichern.
# 0 um sie im HEX-Format zu speichern
RAW_TCPDUMP = "0"
```

```
# 1 um die Namen der Netze in die PNG-Datei zu schreiben
VISUALIZE_NET_NAMES = "1"

# Einstellungen fuer Timeout Timer
TIMEOUT_TIMER = "180"

# Einstellungen fuer Garbagecollection Timer
GARBAGE_TIMER = "120"

# Genutzte Metrik fuer infinity
INFINITY_METRIC = "16"
```

Quelltext 9: Beispiel für die Konfigurationsdatei .zimulatorrc

Beim Aufruf des Programms kann man eine beliebige Datei als Konfigurationsdatei mit dem Parameter `-c` angeben. Alle darin enthaltenen Variablen überschreiben die Standardwerte. Wird keine Konfigurationsdatei explizit angegeben, so wird automatisch die `.zimulatorrc` genutzt, deren Inhalt ebenfalls die Standardwerte überschreiben. Das bedeutet, man braucht in die Konfigurationsdateien nur Variablen zu schreiben, die vom Standard abweichen.

2.3 Simulationen automatisieren

Im Folgenden werden die drei Dateien beschrieben, die für eine Simulation nötig sind.

Topologiebeschreibung mittels ZVF-Dateien

Die Topologie kann durch eine Datei beschrieben werden, die einfacher zu erstellen ist als VNUML-Konfigurationsdateien. Diese Topologiebeschreibung muss die Endung `.zvf` besitzen, um vom Programm erkannt zu werden.

Um aus einer ZVF-Datei eine VNUML-Konfigurationsdatei zu generieren, wird das Programm mit folgenden Parametern aufgerufen:

```
./zimulator.pl -x {DATEI}
```

Es können auch mehrere Dateien gleichzeitig verarbeitet werden.

Das Beispiel aus Quelltext 10 generiert die VNUML-Konfigurationsdateien aus allen ZVF-Dateien im aktuellen Verzeichnis.

```
./zimulator.pl -x *.zvf
```

Quelltext 10: Beispiel für die Generierung einer VNUML-Konfigurationsdatei

Beim Starten einer Simulation werden die XML-Dateien automatisch generiert, falls sie noch nicht existieren.

Syntax der Topologiebeschreibung

Es wurde versucht, die Syntax möglichst einfach zu halten.

Leerzeilen und Zeilen, die mit einem Rautenzeichen (`#`) beginnen werden ignoriert. In der ersten Zeile werden die Netze durch Komma getrennt aufgelistet. Diese müssen mit „net“ beginnen und fortlaufend nummeriert sein. Zum Beispiel *net1*,*net2*,... In den restlichen Zeilen wird jeweils der Name des Routers angegeben. Dieser muss mit *r* beginnen, zum Beispiel *r1* oder *r2*. Nach dem Router kommt ein Leezeichen und dahinter eine Komma-separierte Liste der an diesen Router angeschlossenen Netze.

Beispiel: Dreiecks-Topologie

In der Graphentheorie bezeichnet ein Dreieck den kleinstmöglichen Kreis, der mit einem einfachen Graphen erstellt werden kann. Dieser besteht wie, in Abbildung 1 zu sehen ist, aus drei Knoten, welche mit drei Kanten verbunden sind. Quelltext 11 beschreibt diese Topologie in der ZVF Syntax:

```
net1,net2,net3

r1 net1,net3
r2 net1,net2
r3 net2,net3
```

Quelltext 11: ZVF-Datei eines Dreiecks

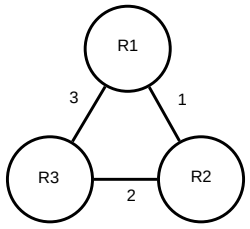

Abbildung 1: Graphische Darstellung der Dreiecks-Topologie

Die globalen Einstellungen für die VNUML-Konfigurationsdatei, der Netztyp und der Aufbau eines einzelnen Routers sowie das Netz, aus dem die IP-Adressen und Netze generiert werden, können über die Konfigurationsdatei (`.zimulatorrc`, siehe auch Seite 7) gesteuert werden.

Zusätzlich kann man die Variablen `$global`, `$net` und `$router` in der Funktion `toXML()` der Datei `modules/Topologie.pm` anpassen, in der die Funktion zum Generieren von Szenario-Dateien definiert ist.

Zur Generierung der IP- und Netzadressen wird standardmäßig das Netz 10.0.0.0/16 herangezogen. Die Netzadressen werden dann entsprechend der eingegebenen Netznummern angelegt: net1 wird zu 10.0.1.0/24 und net12 zu 10.0.12.0/24. Somit sind bis zu 254 Netze möglich. Die Routernummern entsprechen dem letzten Byte der IP-Adresse, wodurch insgesamt maximal 254 Router möglich sind. Durch diese Konvention kann an der IP-Adresse sofort das Netz und der Router abgelesen werden. Demnach würde der Router *r3* im obigen Beispiel die beiden Adressen 10.0.2.3 und 10.0.3.3 erhalten.

Im Folgenden sind noch einige Beispiele für Topologien aufgeführt.

Y-Topologie

Eine gängige Topologie ist die Y-Topologie (siehe Quelltext 12), welche aus vier bis fünf Routern besteht und die ungefähre Form eines Y besitzt. Die Konvergenzzeit beträgt etwa fünf bis zehn Sekunden. (Also sollte man nach dem Start der Routingdaemons mindestens 30 Sekunden warten, wenn man sicher gehen möchte, dass das Netzwerk konvergent ist.)

```
net1,net2,net3,net4,net5

r1 net1,net3
r2 net1,net2
r3 net2,net3,net4
r4 net4,net5
r5 net5
```

Quelltext 12: ZVF-Datei der Y-Topologie

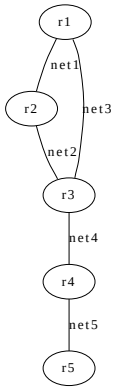

Abbildung 2: Visualisierung der Y-Topologie

Die zugehörige PNG-Datei zu dieser Topologie ist in Abbildung 2 dargestellt.

Square3 Topologie

Eine weitere, oft genutzte Topologie ist Square3 (siehe Quelltext 13). Sie besteht aus einem Quadrat von drei mal drei Routern. Die Konvergenzzeit beträgt etwa fünf bis zehn Sekunden.

```
net1,net2,net3,net4,net5,net6,net7,net8,net9,net10,net11,net12

r1 net1,net3
r2 net1,net2,net4
r3 net2,net5
r4 net3,net6,net8
r5 net4,net6,net7,net9
r6 net5,net7,net10
r7 net8,net11
r8 net9,net11,net12
r9 net10,net12
```

Quelltext 13: ZVF-Datei der Topologie Square3

Abbildung 3 zeigt die Visualisierung der Square3-Topologie.

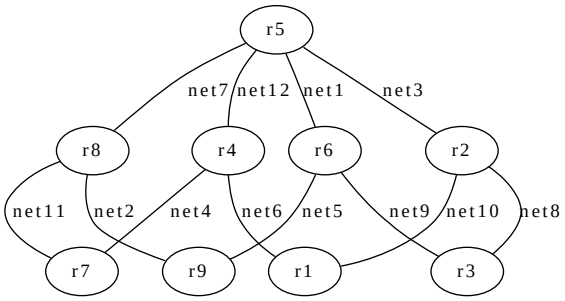

Abbildung 3: Visualisierung der Square3-Topologie

Test-Script

Um automatisiert Simulationen durchzuführen, wird für jedes Test-Script eine Datei mit der Endung `.cfg` erstellt. Diese beschreibt die einzelnen Schritte eines Durchlaufs.

Dabei wird in jede Zeile ein Befehl geschrieben. Die Funktion `runScenario` in der Datei `modules/Simulator.pm` interpretiert die Befehle nacheinander und ruft entsprechende Methoden der Klasse `modules/Scenario.pm` auf.

Beispiele für Test-Scripts befinden sich auf Seite 15.

Im Folgenden werden die möglichen Befehle beschrieben:

Befehlsübersicht:

- start() – Startet VNUML-Szenario, Routingdaemon oder tcpdump-Prozesse.
- stop() – Stoppt VNUML-Szenario, Routingdaemon oder tcpdump-Prozesse.
- gettime() – Hält den aktuellen Zeitpunkt fest zum späteren Messen der Konvergenzzeit.
- sleep(x) – Wartet x Sekunden.
- execute(ROUTER,COMMAND,FILE) – Führt den Befehl COMMAND auf Router ROUTER aus und speichert das Ergebnis mit Zeitpunkt in die Datei FILE (bzw. hängt es an).
- disable() – Deaktiviert einen Router oder ein Device eines Routers (auch zufällige Auswahl ist möglich).
- enable() – Aktiviert den zuletzt deaktivierten Router/Device oder einen angegebenen, falls mehrere deaktiviert wurden.

start() und stop()

Mit dem Startbefehl werden VNUML, die tcpdump-Prozesse und die Routingdaemons gestartet. Für eine genauere Konfiguration können die einzelne Programme separat gestartet werden. Dies wird mit den Parametern angegeben.

Mit dem Parameter `scenario` wird das der Topologie entsprechende VNUML-Szenario gestartet. Zwei Sekunden später wird der Zebra-Daemon auf allen Routern gestartet. Der Parameter `tcpdump` startet für jedes Netz einen tcpdump-Prozess und leitet die Ausgaben in entsprechende Dateien (`<Netzname>.dump` zum Beispiel `net1.dump`) um. Mit dem Parameter `protocol` wird der im globalen Teil definierte Routingdaemon gestartet.

Um alle drei Aktionen in der oben genannten Reihenfolge mit jeweils drei Sekunden Abstand zu starten, kann man den Parameter `all` verwenden.

Der Stop-Befehl hat die gleichen Parameter wie der Start-Befehl. Er stoppt die entsprechenden Dienste bzw. Programme. (`stop(protocol)` ist noch nicht implementiert. Der Routingdaemon wird zusammen mit dem Szenario per `stop(scenario)` gestoppt.)

sleep()

Der Befehl `sleep()` erwartet als Parameter eine ganze Zahl, welche die Sekunden angibt, die gewartet werden sollen bis der nächste Befehl ausgeführt wird.

Dieser Befehl ist vor allem dazu gedacht, abzuwarten bis das Netzwerk konvergent ist, bevor die Simulation beendet wird und die tcpdump-Dateien ausgewertet werden.

Hinweis:

Man sollte in jedem Fall darauf achten, diese Zeit groß genug zu wählen um sicher zu gehen, dass das Netzwerk auch konvergent ist, bevor die Simulation beendet oder die Topologie geändert wird. Am besten probiert man aus, wie lange ein Netzwerk ungefähr braucht um den konvergenten Zustand zu erreichen, sodass die abgewartete Zeit großzügig bemessen zu können.

execute()

Der Befehl `execute()` kann beliebige Befehle auf einem angegebenen Router ausführen und bei Bedarf die Ausgabe des Befehls in eine angegebene Datei auf dem Hostsystem schreiben. Dafür wird der entsprechende Befehl per ssh[3] an den entsprechenden Router gesendet. Als ersten Parameter erwartet `execute()` den Namen des Routers, auf dem der Befehl ausgeführt werden soll. Das zweite Argument gibt den Befehl selber an. Dieser darf jedoch keine Kommata enthalten. Der dritte Parameter ist optional und gibt die Datei an, in der die Ausgabe gespeichert werden soll. Dabei wird auch immer der Zeitpunkt mit in die Datei geschrieben, wann der Befehl ausgeführt wurde. Wird die Datei mehrfach verwendet, so wird die Ausgabe jeweils angehängt.

Der Befehl lässt sich gut verwenden um iptables-Befehle an Router zu senden oder sich zu bestimten Zeitpunkten die Forwardingtabelle eines Routers ausgeben zu lassen.

[3] genauer: OpenSSH – http://www.openssh.com

Das Beispiel in Quelltext 14 führt den Befehl `route -n` auf dem Router *r1* aus und hängt dessen Ausgabe an die Datei `route_r1.txt` an.

```
execute(r1,route -n,route_r1.txt)
```

Quelltext 14: Beispiel für den execute()-Befehl

disable() und enable()

Um einen Router oder ein Device eines Routers zu deaktivieren, kann der Befehl `disable()` verwendet werden. Er dient zur Simulation von Ausfällen. Um einen Router zu deaktivieren ruft man den Befehl mit dem Namen des Routers als Parameter auf. Soll ein zufälliger Router ausfallen, so kann der Parameter `random` genutzt werden. Der Router wird bei jedem Durchlauf neu berechnet, sodass man Aussagen über Konvergenzzeiten treffen kann, wenn ein beliebiger Router ausfällt (vorrausgesetzt es wurden genügend Durchläufe ausgeführt).

Möchte man Router vom Herunterfahren mittels des `random`-Parameters ausschließen, so kann man diese in eckigen Klammern mit Komma getrennt angeben. Im folgenden Beispiel soll ein zufälliger Router deaktiviert werden, die beiden Router *r1* und *r2* sollen davon aber ausgeschlossen werden: `disable(random[r1,r2])`.

Wenn man einen Router deaktiviert, der im entsprechenden Graphen einer Artikulation entspricht, so kann es passieren, dass zwei Teilnetze entstehen, die parallel viel schneller konvergieren als das ursprüngliche große Netz. Da dies die Messergebnisse verfälschen würde, sollten solche Router ausgeschlossen werden (Artikulationen können durch die Analyse der Topologie ausfindig gemacht werden, siehe dazu Kapitel 2.5).

Problematisch ist es auch, wenn Router deaktiviert werden, die an nur ein Netz angeschlossen sind. Da die getesteten Routingalgorithmen netzbasiert arbeiten bleibt das Netzwerk in einem solchen Fall konvergent, da weiterhin alle Netze erreichbar sind. Daher sollten bei der zufälligen Auswahl des zu deaktivierenden Routers solche Router ausgeschlossen werden (Blätter können ebenfalls durch die Analyse der Topologie berechnet werden).

Um ein Device eines Routers zu deaktivieren, kann man die Nummer des Devices als zweiten Parameter angeben. Der erste Parameter gibt dann den Rechner an, auf dem das Device deaktiviert werden soll. Auch hier kann der Parameter `random` in beliebiger Kombination genutzt werden.

Beispiele:

- `disable(r1)`
- `disable(random)`
- `disable(random[r1,r2])`
- `disable(r1,2)`
- `disable(r1,random)`
- `disable(r1,random[1])`
- `disable(random,2)`

- `disable(random[r1,r2],1)`
- `disable(random,random)`
- `disable(random[r1,r2],random)`
- `disable(random,random[1])`
- `disable(random[r1,r2],random[1])`

Um einen deaktivierten Router wieder zu aktivieren, gibt es den Befehl `enable()`. Wird er ohne Parameter aufgerufen, so wird der zuletzt deaktivierte Router (bzw. das zuletzt deaktivierte Device) wieder aktiviert. Möchte man einen anderen Router (bzw. ein anderes Device) wieder aktivieren, so kann man (bis auf `random`) die gleichen Parameter nutzen wie bei `disable()`.

gettime()

Der Befehl `gettime()` speichert die zum Zeitpunkt seines Aufrufs aktuelle Zeit. Diese wird später als Startzeitpunkt zur Berechnung der Konvergenzzeit genutzt. Dieser Befehl kennt keine Parameter. Er kann genutzt werden, um zu messen, wie lange ein Netzwerk benötigt, bis es nach dem Ausfall eines Routers wieder konvergent ist. Dazu führt man den Befehl direkt vor oder nach dem Befehl `disable()` aus.

Im Folgenden werden zwei Beispiele für Test-Scripte aufgeführt.

Konvergenzzeitbestimmung

Als erstes ein recht simples Beispiel, welches lediglich die Konvergenzzeit einer Topologie bestimmt. Es wird ein VNUML-Szenario gestartet und danach die Forwardingtabelle des Routers *r1* mit dem Befehl `execute(r1,route -n,route_r1.txt)` in die Datei `route_r1.txt` geschrieben. Anschließend werden die tcpdump-Prozesse und das Routingprotokoll gestartet und 60 Sekunden abgewartet. Schließlich werden die tcpdump-Prozesse sowie der Durchlauf selbst beendet.

Quelltext 15 zeigt die zugehörige Datei `konvergenzzeit60sec.cfg`.

```
start(scenario)
sleep(3)
execute(r1,route -n,route_r1.txt)
start(tcpdump)
start(protocol)
sleep(60)
execute(r1,route -n,route_r1.txt)
stop(tcpdump)
stop(scenario)
```

Quelltext 15: Test-Script für eine einfache Konvergenzzeitbestimmung

Routerausfall

Dieses Beispiel lässt einen zufälligen Router ausfallen und misst die Zeit, bis das Netzwerk wieder konvergent ist. Nach dem Start des Szenarios, der tcpdump-Prozesse und des Routingdaemons

wird 60 Sekunden gewartet, bis das Netz konvergent ist. Nun wird ein zufälliger Router mit dem Befehl `disable(random)` deaktiviert. Dann wird die aktuelle Zeit mit `gettime()` gespeichert, um sie später als Startzeitpunkt für die Konvergenzzeitberechnung zu nutzen. Schließlich wird noch fünf Minuten gewartet, bis das Netzwerk wieder konvergent ist und dann der Durchlauf beendet.

Die zugehörige Datei `routerausfall60\_300.cfg` ist in Quelltext 16 aufgeführt.

```
start(scenario)
sleep(3)
start(tcpdump)
start(protocol)
sleep(60)
disable(random)
gettime()
sleep(300)
stop(tcpdump)
stop(scenario)
```

Quelltext 16: Test-Script für einen Routerausfall

Ausführungsbeschreibung

Die Ausführungsbeschreibung besitzt die folgende Syntax:

```
TOPOLOGIENAME SIMULATIONSNAME PROTOKOLL DURCHLÄUFE FEHLER
```

wobei der Topologiename der ZVF- bzw. XML-Datei ohne Endung entspricht. Der Simulationsname ist der Name des Test-Scripts (ohne Endung). Das Feld „Protokoll“ ist für zukünftige Zwecke eingeführt worden, um auch andere Protokolle als RIP testen zu können. Gleichzeitig entspricht es dem auszuführenden *execute-Tag* der VNUML-Konfigurationsdatei, welches das entsprechende Protokoll startet. Das Feld „DURCHLÄUFE“ gibt an, wie oft diese Simulation wiederholt werden soll. „FEHLER“ gibt die maximale Anzahl der Fehler an, nach denen mit dieser Simulation abgebrochen werden soll. Ein Fehler entsteht zum Beispiel, wenn ein Dienst nicht startet oder die errechnete Konvergenzzeit negativ ist.

Ruft man das Programm nun mit den Parametern aus Quelltext 17 auf (wobei `simulations.txt` die Ausführungsbeschreibung ist), so werden die Simulationen der Reihe nach abgearbeitet.

```
sudo ./zimulator.pl -s simulations.txt
```

Quelltext 17: Beispiel für das Starten einer Simulation

Nach jedem Durchlauf wird die Datei erneut eingelesen und mit dekrementierter Anzahl von Durchläufen wieder zurückgeschrieben. Somit zeigt die Datei gleichzeitig an, an welcher Stelle

sich die Simulation gerade befindet. Durch dieses Vorgehen ist es auch möglich während der Simulation die Anzahl der Durchläufe oder die zu simulierenden Szenarien zu ändern. Ein weiterer Vorteil dieser Methode ergibt sich daraus, dass bei einem Problem (es kann vorkommen, dass VNUML hängen bleibt) alle an der Simulation beteiligten Programme und Prozesse durch ein spezielles Script gestoppt werden und danach die mit dem abgebrochenen Durchlauf fortgesetzt werden können (siehe auch Kapitel 2.3 auf Seite 17).

Die Funktion `simulate()` in der Datei `modules/Simulator.pm` interpretiert und manipuliert die Ausführungsbeschreibung und ruft für jeden Durchlauf die Funktion `runScenario()` auf, welche das Test-Script parst.

Die Ausführungsbeschreibung `simulations.txt` (siehe Quelltext 18) lässt jede Topologie in jeder Simulation zehn mal laufen. Wenn pro Simuation mehr als fünf Fehler auftreten, wird sie abgebrochen.

```
y_scenario konvergenzzeit60sec rip 10 5
square3 konvergenzzeit60sec rip 10 5

y_scenario routerausfall60_300 rip 10 5
square3 routerausfall60_300 rip 10 5
```

Quelltext 18: Beispiel einer Ausführungsbeschreibung

Überwachung und Neustart der Simulationen bei Problemen

Da es aus verschiedenen Gründen immer wieder vorkommen kann, dass VNUML hängen bleibt, wurde eine Reihe von Perl-Scripts geschrieben, welche die Simulationen überwachen und alle beteiligten Prozesse bei Problemen stoppen können. Danach werden die Simulationen genau an der abgebrochenen Stelle wieder aufgenommen.

Um diese Scripts zu benutzen, muss die Ausgabe von `zimulator.pl` in eine Datei umgeleitet werden (siehe Quelltext 19 für den entsprechenden Befehl), die regelmäßig geprüft wird.

```
./zimulator.pl -sv simulations.txt >> simulations.out
```

Quelltext 19: Starten von zimulator.pl mit Umleitung der Ausgabe

Danach startet man das Script `checkRunning.pl`, welches regelmäßig prüft, ob die Datei `simulations.out` größer wurde. Ist der Datei fünf Minuten lang nichts angehängt worden, so geht das Script von einem Problem aus und ruft das Script `restart.pl` auf. `restart.pl` ruft das Script `killsimulation.pl` auf, welches `zimulator.pl`, `vnumlparser.pl` sowie alle laufenden uml- und tcpdump-Prozesse beendet. Die LOCK-Datei im VNUML-Verzeichnis des Users wird gelöscht, damit VNUML wieder starten kann. Daraufhin wird der Ausführungsbeschreibung der Name des aktuellen Szenarios entnommen und dieses mit `vnumlparser.pl` gestoppt. Danach wird `zimulator.pl` durch `restart.pl` über den Aufruf in Quelltext 19 erneut gestartet. Es wird die abgebrochene

Simulation wiederholt und danach normal weiter gearbeitet. `checkRunning` läuft die ganze Zeit über weiter und überwacht weiterhin die Datei `simulations.out` auf Veränderungen.

2.4 Auswertung der Ergebnisse

Alle während einer Simulation generierten Dateien werden in einem eigenen Verzeichnis (dem Simulationsordner) gepeichert, welches wie folgt genannt wird: `simulationsname-topologiename`.

Dort wird auch die Ergebnisdatei abgelegt, die den Namen der Topologie mit der Endung `.txt` erhält. Aus den tcpdump-Dateien eines Durchlaufs kann errechnet werden, wie lange die Konvergenzzeit war (von Anfang an, ab oder bis zu einem bestimmten Zeitpunkt). Des Weiteren wird die Anzahl der versendeten Routingpakete in diesem Zeitraum und der dadurch erzeugte Traffic ausgewertet.

Aus diesen Daten werden automatisch Statistiken erzeugt (durchschnittliche Anzahl der Pakete pro Netz, Netz mit dem meisten/wenigsten Traffic, ...). Diese Statistiken werden zusammen mit den Topologieeigenschaften in die Ergebnisdatei geschrieben, welche pro Durchlauf die folgenden Werte jeweils mit einem Leerzeichen getrennt enthält. In Klammern steht der jeweilige Schlüssel, der im `resultHash` für den entsprechenden Wert verwendet wird (siehe Seite 31):

1. Durchmesser der getesteten Topologie (DIAMETER)
2. Anzahl der Knoten der Topologie (VERTICES)
3. Anzahl der Blätter (LEAVES)
4. Anzahl der inneren Knoten (INNERVERTICES)
5. Anzahl der Kanten (EDGES)
6. Clusterkoeffizient (CC)
7. Konvergenzzeit des Durchlaufs (TIMETOCONVERGENCE)
8. Anzahl der Routingpakete (TOTALPACKETCOUNT)
9. Trafficvolumen der Routingpakete (TOTALTRAFFIC)
10. Anzahl der durchschnittlichen Pakete pro Netz (AVERAGEPACKETCOUNT)
11. Durchschnittlicher Updatetraffic pro Netz (AVERAGETRAFFIC)
12. Anzahl der Pakete im Netz, über das die wenigsten Pakete versendet wurden (LEASTPACKETSNETCOUNT)
13. Nummer des Netzes, über das die wenigsten Pakete versendet wurden (LEASTPACKETSNET)
14. Anzahl der Pakete im Netz, über welches die meisten Pakete versendet wurden (MOSTPACKETSNETCOUNT)
15. Nummer des Netzes, über das die meisten Pakete versendet wurden (MOSTPACKETSNET)
16. Traffic des Netzes mit dem geringesten Trafficaufkommen (MINTRAFFICNET)
17. Nummer des Netzes mit dem geringsten Trafficaufkommen (MINTRAFFIC)
18. Traffic des Netzes mit dem höchsten Trafficaufkommen (MAXTRAFFICNET)

19. Nummer des Netzes mit dem höchsten Trafficaufkommen (MAXTRAFFIC)
20. Router oder Device, das ausgefallen ist – 0 wenn es keinen Ausfall gab (FAIL)
21. Zeit des Router- oder Deviceausfalls (FAILURETIME)
22. Name der getesteten Topologiedatei (TOPOLOGIENAME)
23. Name des getesteten Protokolls (PROTOCOL)
24. Nummer des Durchlaufs (INTERNRUNCOUNT)
25. Erster Zeitstempel (FIRSTTIMESTAMP)
26. Letzter Zeitstempel (LASTTIMESTAMP)

Wird eine Simulation mehrmals mit den gleichen Parametern simuliert, so können am Ende mit dem Modus `-a` die maximalen, minimalen und durchschnittlichen Werte errechnet werden. Die Ausgabe hat eine ähnliche Form wie die Ergebnisdatei. Auch hier wurde in Klammern der jeweilige Schlüssel des `avgResult`-Hashes angegeben.

1. Durchmesser der getesteten Topologie (DIAMETER)
2. Anzahl der Knoten der Topologie (VERTICES)
3. Anzahl der Blätter (LEAVES)
4. Anzahl der inneren Knoten (INNERVERTICES)
5. Anzahl der Kanten (EDGES)
6. Clusterkoeffizient (CC)
7. Längste Konvergenzzeit eines Durchlaufs (LONGESTTIME)
8. Kürzeste Konvergenzzeit eines Durchlaufs (SHORTESTTIME)
9. Durchschnittliche Konvergenzzeit aller Durchläufe (AVERAGETIME)
10. Anzahl der Pakete im Durchlauf mit den meisten Paketen (Maximum von Feld TOTALPACKETCOUNT der Ergebnisdatei) (MAXTOTALPACKETS)
11. Anzahl der Pakete im Durchlauf mit den wenigsten Paketen (Minimum von Feld TOTALPACKETCOUNT der Ergebnisdatei) (MINTOTALPACKETS)
12. Durchschnittliche Anzahl der gesendeten Packete pro Durchlauf (AVERAGETOTALPACKETS)
13. Durchschnitt der maximalen Paketanzahlen pro Netz (Durchschnitt von Feld MOSTPACKETSNETCOUNT der Ergebnisdatei) (MAXAVERAGEPACKETS)
14. Durchschnitt der minimalen Paketanzahlen pro Netz (Durchschnitt von Feld LEASTPACKETSNETCOUNT der Ergebnisdatei) (MINAVERAGEPACKETS)
15. Durchschnitt der Pakete pro Netz (AVGAVERAGEPACKETS)
16. Höchster Gesamttraffic (MAXTOTALTRAFFIC)
17. Geringster Gesamttraffic (MINTOTALTRAFFIC)
18. Durchschnittlicher Gesamttraffic (AVERAGETOTALTRAFFIC)
19. Durchschnittlicher Traffic pro Netz (AVGAVERAGETRAFFIC)
20. Name der getesteten Topologie (TOPOLOGIENAME)

Es besteht weiterhin die Möglichkeit, tcpdump-Dateien im Nachhinein erneut auszuwerten. Dafür gibt es die Option `-r`. Mit der Option `-T` kann ein Zeitpunkt angegeben werden, bis zu dem die Dumps ausgewertet werden sollen. Alle Pakete, die zu einem späteren Zeitpunkt versendet worden sind werden ignoriert. Das bietet die Möglichkeit, eine Simulation mit einem Routerausfall zu testen und danach die Konvergenzzeiten von Anfang an bis zum ersten konvergenten Zustand berechnen zu lassen. Somit sind weniger Simulationen notwendig.

Mit der Option `-F` lässt sich ein Zeitpunkt angeben, der in der Berechnung als Startzeitpunkt für die Konvergenzzeit verwendet wird. Alle vorher gesendeten Pakete werden ignoriert.

Um mehrere Durchläufe gleichzeitig zu analysieren gibt es die Option `-S` bei der statt der tcpdump-Dateien die Topologiedatei übergeben werden. Dabei werden die tcpdump-Dateien aller mit dieser Topologie simulierten Durchläufe automatisch gesucht und ausgewertet.

Beispiele:

`./zimulator -r beispiel.zvf simulation-topologie/rip_run_1/net*` – Wertet die Dumps des ersten Durchlaufs der Simulation „simulation-topologie“ aus.

`./zimulator -r beispiel.zvf -F 123456789 simulation-topologie/rip_run_1/net*` – Wertet die Dumps des ersten Durchlaufs der Simulation „simulation-topologie“ ab dem Zeitpunkt 123456789 aus.

`./zimulator -r beispiel.zvf -T 123456789 simulation-topologie/rip_run_1/net*` – Wertet die Dumps des ersten Durchlaufs der Simulation „simulation-topologie“ bis zum Zeitpunkt 123456789 aus.

`./zimulator -S beispiel.zvf` – Wertet die Dumps aller Durchläufe aller Simulationen der topologie „beispiel“ aus und überschreibt die Ergebnisdateien.

`./zimulator -S -o newresultfile.txt beispiel.zvf` – Wertet die Dumps aller Durchläufe aller Simulationen der Topologie „beispiel“ aus und generiert die Ergebnisdatei „newresultfile.txt“.

`./zimulator -S -F 0 -o newresultfile.txt beispiel.zvf` – Wertet die Dumps aller Durchläufe aller Simulationen der Topologie „beispiel“ aus, wobei der Startzeitpunkt der Auswertung der Ergebnisdatei entnommen wird, und generiert die Ergebnisdatei `newresultfile.txt`.

`./zimulator -S -T 0 -o newresultfile.txt beispiel.zvf` – Wertet die Dumps aller Durchläufe aller Simulationen der Topologie „beispiel“ aus, wobei der Endzeitpunkt der Auswertung der Ergebnisdatei entnommen wird, und generiert die Ergebnisdatei `newresultfile.txt`.

tcpdump

Beim Starten der tcpdump-Prozesse wird zuerst ein ifconfig ausgeführt, um alle Netze ausfindig zu machen. Für jedes Netz (also jedes Device, das mit „net“ anfängt) wird nun ein tcpdump-Prozess gestartet, der mit dem Namen des Netzes und der Endung `.dump` gespeichert wird. Die dafür benötigten Funktionen sind in der Datei `modules/Scenario.pm` definiert. Für jeden

Durchlauf werden die Dumps nach der Simulation in ein eigenes Verzeichnis verschoben. Wird eine Simulation mit gleicher Topologie und gleichem Protokoll noch einmal mit der gleichen Durchlauf-Nummer ausgeführt, so wird eine fortlaufende Zahl angehängt.

Beispiel:

```
rip_run_1
rip_run_2
rip_run_1_1
rip_run_1_2
```

Somit werden die Dumps niemals versehentlich überschrieben. Dies ist oft sehr praktisch, wenn die Simulation abgebrochen wird oder im Nachhinein weitere Durchläufe ausgeführt werden.

Die Dumps können entweder im RAW-Format gespeichert werden oder im Hexadezimalformat ohne den Link-Header (dies entspricht der Option `-x` von `tcpdump`). Welches Format genutzt werden soll, kann in der Datei `.zimulatorrc` (siehe auch Seite 7) angegeben werden.

Berechnung der Konvergenzeigenschaften

Die Berechnung der Konvergenzzeit benötigt neben den tcpdump-Dateien aller Netze die Topologiebeschreibung des Netzwerks. Für jeden Router werden die tcpdump-Pakete der an den Router angeschlossenen Netze chronologisch sortiert. Für jeden Router wird eine Forwardingtabelle anhand der Dumps aufgebaut. Der Zeitpunkt des letzten Pakets, welches die Forwardingtabelle eines Routers verändert hat, wird als Konvergenzzeitpunkt gespeichert. Das erste RIP-Paket aller Dumps wird als Startzeitpunkt der Konvergenzmessung herangezogen. Die Differenz beider Zeitstempel bildet die Konvergenzzeit.

Die Anzahl der Pakete sowie der Traffic werden aus den RIP-Paketen der tcpdump-Dateien berechnet, die zwischen den beiden während der Konvergenzzeitmessung erzeugten Zeitstempeln liegen. Die tcpdump-Dateien geben die Länge der Pakete ohne Header aus. Daher ist auf diese Länge noch der IP-Header von 20 Byte [4] und der UDP-Header von 8 Byte [3] aufzuaddieren.

Statistische Analyse der Messergebnisse

Die Ergebnisse der Simulationen können mit dem Programm gnuplot[4] graphisch dargestellt werden.

Wichtig ist, dass die Werte durch Leerzeichen voneinander getrennt sind. Um eine solche Datei zu visualisieren, wird noch eine Plotdatei benötigt, die Durchmesser, Knotenanzahl, Kantenanzahl oder Clusterkoeffizient auf der x-Achse gegen Konvergenzzeit, Paketanzahl oder Traffic auf der y-Achse darstellt. Eine minimale Plotdatei, welche den Durchmesser und die Konvergenzzeit

[4] http://gnuplot.info

berücksichtigt, ist in Quelltext 20 dargestellt. Dadurch wird für jeden Messwert die erste Spalte auf der x-Achse und die fünfte Spalte auf der y-Achse aufgetragen.

```
plot "messungen.txt" using 1:5
```

Quelltext 20: Beispiel einer minimalen Plotdatei

Für dreidimensionale Graphen wird die Funktion splot verwendet. Im nächsten Beispiel (Quelltext 21) wird die Paketanzahl in Zusammenhang mit Knotenanzahl und Kantenanzahl gebracht.

```
splot "messungen.txt" using 2:3:6
```

Quelltext 21: Beispiel einer minimalen 3d-Plotdatei

Methode der kleinsten Quadrate

`gnuplot` beherrscht auch die Methode der kleinsten Quadrate. Jedoch funktioniert sie nur für den eindimensionalen Fall richtig. Um diese Funktion zu verwenden, definiert man zuerst eine Funktion, die die Messwerte wahrscheinlich am besten approximiert, und lässt `gnuplot` dann mit der Funktion `fit` die Variablen berechnen. Das Beispiel aus Quelltext 22 zeigt die Approximation des linearen Zusammenhangs von Durchmesser und Konvergenzzeit. Dort sind a und b die Variablen, die berechnet werden, während x der Durchmesser und *f(x)* die Konvergenzzeit ist.

```
f(x)=a*x+b
fit f(x) 'resultfile.txt' using 1:5 via a,b

plot f(x)
```

Quelltext 22: Methode der kleinsten Quadrate mit gnuplot

Für die berechnung von Funktionen mit mehreren unabhängigen Variablen (multidimensionale Funktionen), eignet sich das Programm maxima[5] besser. Es liefert jedoch nur die Werte der approximierten Variablen sowie die Varianz. Diese Werte können jedoch für die Funktion in gnuplot genutzt werden.

Hat man maxima gestartet, so kann man die Ergebnisdatei über den Befehl `read_matrix` einlesen und als Matrix speichern. Wenn die Datei viele Zeilen hat, so kann es vorteilhaft sein, vorher die nicht benötigten Spalten zu löschen sowie die Werte auf Ganzzahlen zu runden. Als nächstes lädt man das Paket `lsquares` mit dem Befehl `load(lsquares)$`, welches Funktionen für die Berechnung der kleinsten Quadrate zur Verfügung stellt. Der nächste Befehl `lsquares_estimates` erwartet als ersten Parameter das Array, in dem die Messwerte liegen. Der zweite Parameter definiert Variablen für die einzelnen Spalten der Matrix. Im dritten Parameter wird die Funktion festgelegt, welche der Approximation zugrunde gelegt werden soll, während der vierte und letzte

[5] http://maxima.sourceforge.net

Parameter die zu berechnenden Variablen nochmals definiert. Das Ergebnis wird im Beispiel in die Variable `a` geschrieben. Das Beispiel in Quelltext 23 approximiert das Trafficvolumen über die Anzahl der Knoten und Kanten mit einem zweidimensionalen Polynom 2. Grades. Es wird für jede Variable der errechnete Wert ausgegeben und gleichzeitig in der Variable `a` gespeichert.

```
M : read_matrix("resultfile.txt");
load(lsquares)$
a : lsquares_estimates ( M, [r,s,t,u,v,w,x,y,z],
  z = A*s^2 + B*s + C*v^2 + D*v + E, [A,B,C,D,E] );
```

Quelltext 23: Methode der kleinsten Quadrate mit maxima

Um die Varianz zu berechnen, wird die Funktion `lsquares_residual_mse` genutzt. Die ersten drei Parameter der Funktion sind identisch mit denen der Funktion `lsquares_estimates`. Als vierten Parameter gibt man `first(a)` an. Ein Beispiel ist in Quelltext 24 angegeben. Die Ausgabe ist die Varianz der approximierenden Funktion mit den in `lsquares_estimates` errechneten Variablenwerten für die in der Matrix gespeicherten Werte.

```
lsquares_residual_mse ( M, [r,s,t,u,v,w,x,y,z],
  z = A*s^2 + B*s + C*v^2 + D*v + E, first(a) );
```

Quelltext 24: Berechnung der Varianz mit maxima

Quellen: [2], [1]

Die in dieser Arbeit genutzten Plotdateien haben noch einige weitere Einstellungen, um zum Beispiel die Achsen zu beschriften oder die Legende anders zu positionieren. Sie sind auf der beiliegenden CD enthalten oder online unter http://github.com/zimon/zimulator zu finden.

2.5 Graphentheoretische Analyse von Topologien

Mit Hilfe der CPAN-Bibliothek „Graph“ ist das Programm in der Lage, Graphen aus Topologiebeschreibungen zu erstellen und graphentheoretisch zu analysieren. Es können ZVF-Dateien eingelesen und in einen Graphen umgewandelt werden und aus Graphen wieder ZVF-Dateien erstellt werden. Zudem können einige Eigenschaften der Graphen bestimmt werden, die einen praktischen Nutzen bei der Analyse von Routingprotokollen nahe legen.

Alle Funktionen und Datenstrukturen für die Analyse und Darstellung von Graphen sind in der Datei `modules/GraphTools.pm` definiert (in der auch die Bibliothek `Graph` eingebunden ist).

Um eine ZVF-Datei zu analysieren ruft man das Programm mit folgender Syntax auf:

```
./zimulator.pl -z {DATEI}
```

Dabei werden die folgenden graphentheoretischen Eigenschaften berechnet:

- Durchmesser d
- Anzahl der Knoten n
- Anzahl der inneren Knoten n_i
- Blätter
- Anzahl der Kanten m
- Artikulationen
- Clusterkoeffizient cc
- Anzahl der Kreise k (zyklomatische Zahl)

Es können also auch mehrere Dateien gleichzeitig analysiert werden. Das Ergebnis der Analyse wird auf der Konsole ausgegeben sowie als Kommentar in die ZVF-Datei geschrieben. Alle bisherigen Kommentare gehen dabei jedoch verloren.

Die verwendeten Module haben einige Einschränkungen bei der Verwendung von Hypergraphen. Diese können zum Beispiel nicht mittels GraphViz dargestellt werden.

Hinweis: Das Programm wurde noch nicht mit Hyperkanten, deren Knotenanzahl kleiner als 2 ist, getestet.

Visualisierung von Graphen

Bei der Analyse der Graphen wird automatisch eine PNG-Datei mit einem Bild des Graphen erstellt. Dieses wird durch das Programm GraphViz[6] berechnet. Es können die Namen der Netze an die Kanten geschrieben werden. Dies kann man in der Datei `modules/Constants.pm` festlegen.

2.6 Generierung von Topologien

Mit dem Programm können über die Option `-g` symmetrische und zufällige Topologien generiert werden. Dabei wird automatisch eine ZVF-Datei erstellt und beide Dateien unter einem automatisch generierten oder optional angegebenen Namen gespeichert. Eine graphische Darstellung der Topologie als PNG-Datei kann durch die zusätzliche Angabe des Parameters `-i` generiert werden.

Die generierten Graphen werden automatisch analysiert und deren Eigenschaften als Kommentare in die ZVF-Datei geschrieben. Die generelle Syntax zum Generieren von Topologien ist:

```
./zimulator.pl -g TYPE ARGUMENTS [NAME]
```

In Tabelle 1 werden die möglichen Typen von Topologien mit ihren Argumenten aufgelistet:

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

Name	Argumente	Beschreibung
Row	N	Eine Reihe aus N Routern
circle	N	Ein Kreis aus N Routern
star2	N	Ein Router, an den N-1 andere Router direkt angeschlossen sind
star	D R	Ein Router, an den R Reihen der Länge D/2 angeschlossen sind
square	N	Ein Quadrat aus N x N Routern
crown	N	Eine CrownN-Topologie
circlex_rowy	X Y	Ein Kreis aus X Routern verbunden mit einer Reihe aus Y Routern
random	N M	Eine zufällige Topologie aus N Routern verbunden mit M Netzen

Tabelle 1: Tabelle der generierbaren Topologieklassen

Zufällige Topologien

Die Funktionen zur Generierung zufälliger Graphen sind im CPAN-Modul Graph implementiert, welches in der Datei `modules/GraphTools.pm` eingebunden ist. Es werden keine Netze mit Hyperkanten oder Multikanten erstellt.

Die Aufrufsyntax ist:

```
./zimulator.pl -g random N M [NAME]
```

wobei N die Anzahl der Knoten angibt, M die Anzahl der Kanten und mit NAME der Name der zu erzeugendenen Dateien angegeben wird. Der Befehl aus Quelltext 25 erstellt eine zufällige Topologie mit dem automatisch generierten Namen „random9_15“ bestehend aus neun Knoten und 15 Kanten. Durch diesen Aufruf könnten zum Beispiel die Dateien `random9_15.png` (siehe Abbildung 4) und `random9_15.zvf` (siehe Quelltext 26) erzeugt werden.

```
./zimulator.pl -g random 9 15
```

Quelltext 25: Beispiel zur Generierung einer zufälligen Topologie

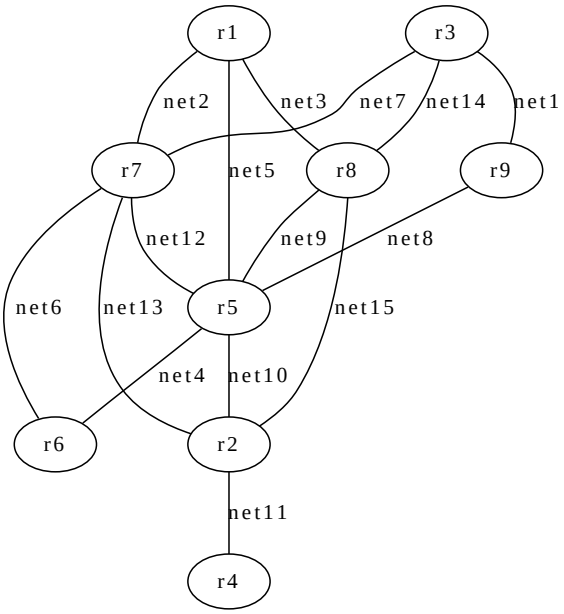

Abbildung 4: Visualisierung der erzeugten Zufallstopologie random9_15

```
# random9_15
#
# diameter:                       3
# number of cycles:               7
# clustering coefficient:         0.416666666666667
# leaves:                         r4
# articulations:                  r2

net1,net2,net3,net4,net5,net6,net7,net8,net9,net10,net11,net12,net13,net14
,net15

r1 net2,net3,net5
r2 net10,net11,net13,net15
r3 net1,net7,net14
r4 net11
r5 net4,net5,net8,net9,net10,net12
r6 net4,net6
r7 net2,net6,net7,net12,net13
r8 net3,net9,net14,net15
r9 net1,net8
```

Quelltext 26: ZVF-Datei der erzeugten Zufallstopologie

3 Informationen für Entwickler

Das Programm `zimulator.pl` wurde objektorientiert in der Programmiersprache Perl geschrieben. Ein UML-Klassendiagramm mit einigen zusätzlichen Informationen ist in Abbildung 5 dargestellt. Im Folgenden werden die einzelnen Klassen und Module beschrieben.

Configuration

Die Klasse `Configuration` speichert alle wichtigen Konstanten, die auch über eine Konfigurationsdatei definiert werden können. Sie besitzt keine fest definierten Variablen. Stattdessen wird die gesamte Klasse als Hash genutzt, in dem beliebige Optionen mit den zugehörigen Werten gespeichert werden können. Dafür sind auch Getter- und Settermethoden implementiert. Ohne Angabe einer Konfigurationsdatei werden Standardwerte geladen. Wird beim Erzeugen eines Configuration-Objekts ein File-Objekt mit einer Konfigurationsdatei übergeben, so werden die Standardwerte mit den Werten der Konfigurationsdatei überschrieben. Dabei brauchen in der Konfigurationsdatei nur die Optionen definiert zu werden, die von den Standardwerten abweichen.

Die Klasse ist als *Singleton Pattern* implementiert, sodass es nur eine Instanz gibt. Diese Instanz kann von jedem Punkt im Programm über die statische Funktion `instance()` bezogen werden[7].

Utilities

Die Datei `Utilities.pm` ist ein Perl-Modul, welches statische Funktionen zur Verfügung stellt, die häufig gebraucht werden. Dies sind zum einen die Funktionen `makeTimeStamp()` und `makeTime()`, welche aus einem Zeitstring, wie ihn tcpdump liefert, einen Zeitstempel generieren und umgekehrt aus einem Zeitstempel den entsprechenden String erstellen. `getTime()` und `getTimeStamp()` geben die aktuelle Systemzeit zum Zeitpunkt des Aufrufs als Zeitstempel oder Zeitstring zurück.

Die Funktionen `getSimulationDirectory()` und `getResultFiles()` durchsuchen das aktuelle Verzeichnis nach Simulationsordnern und darin nach Ergebnisdateien. Sie liefern jeweils eine Liste der gefundenen Verzeichnisse bzw. Dateien zurück.

Die Funktion `remove()` bekommt zwei Listenreferenzen übergeben, und löscht dann jedes Vorkommen eines Elementes der zweiten Liste aus der ersten Liste. Diese Funktion wird vor allem genutzt, um Ausnahmen für zufällige Routerausfälle umzusetzen. (Die Liste der Router, die nicht ausfallen dürfen, wird von der Liste aller Router abgezogen.)

[7] Solange das Modul `Configuration` geladen wurde.

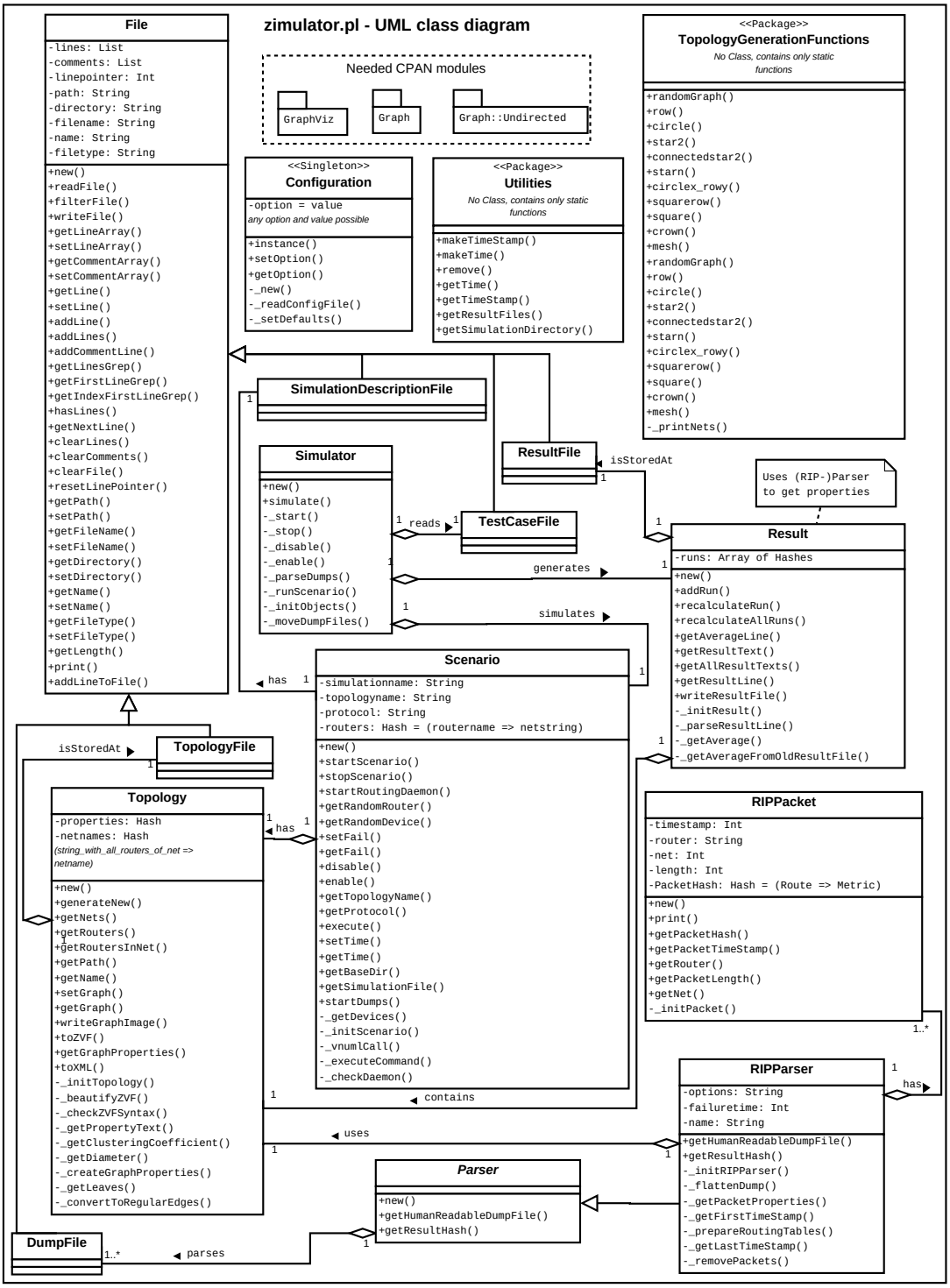

Abbildung 5: UML-Klassendiagramm

File

Die Klasse `File` repräsentiert eine Datei. Da bei allen im Programm genutzten Dateien Leerzeilen und Kommentarzeilen (welche mit einem `#` beginnen) ignoriert werden, werden zwei Listen mit Zeilen verwaltet. Die Liste `lines` beinhaltet die Nutzdaten, während in der Liste `comments` Kommentare und Leerzeilen gespeichert werden. Die Klasse `File` besitzt Methoden, um Zeilen zurückzugeben, anzuhängen und zu ändern. Es können auch reguläre Ausdrücke genutzt werden, um die erste Zeile, den Index der ersten Zeile oder alle Zeilen, die den regulären Ausdruck erfüllen, zurückzugeben. Die beiden Listen können auch komplett ausgegeben oder überschrieben werden.

Des Weiteren wurde ein *Iterator Pattern* implementiert. Dabei repräsentiert die Variable `linepointer` den Index der aktuellen Zeile. Die Methode `hasLines()` prüft, ob der `linepointer` bereits auf die letzte Zeile der Liste lines zeigt. Ist dies der Fall, so wird 0 zurückgegeben, ansonsten die Anzahl der restlichen Zeilen. Die Funktion `getNextLine()` gibt die aktuelle Zeile zurück und inkrementiert den `linepointer`, wenn er noch nicht auf der letzten Zeile steht. Mit der Methode `resetLinePointer()` wird der `linepointer` auf 0 gesetzt.

Der Pfad zur Datei wird in der Klasse in die Variablen `path` (kompletter Pfad mit Dateiname), `directory` (der Pfad ohne Dateiname), `filename` (der Dateiname mit Suffix), `name` (der Dateiname ohne Suffix) und `filetype` (nur das Suffix) aufgespalten. Wird einer der Werte geändert, so werden alle anderen automatisch angepasst.

Die statische Funktion `addLineToFile()` fügt eine Zeile zu der angegebenen Datei hinzu. Sie erwartet den Pfad zur Datei sowie den anzuhängenden String als Argumente. Diese Funktion kann auch ohne ein `File`-Objekt aufgerufen werden.

Von der Klasse `File` sind die Klassen `TopologyFile`, `ResultFile`, `TestCaseFile`, `ExecutionDescriptionFile` und `DumpFile` abgeleitet, welche einen Syntaxcheck für den jeweiligen Dateityp implementieren.

Topology

Die Klasse `Topology` repräsentiert eine Topologie (also einen Graphen), welcher aus einem `TopologyFile`-Objekt gelesen werden kann. Mit den Methoden `toZVF()` und `toXML()` kann der Graph wieder in ein `TopologyFile`-Objekt oder eine VNUML-XML-Datei überführt werden. Für die Datenstruktur des Graphen wird dabei das Perl-Modul `Graph` verwendet. Daneben besitzt die Klasse auch Methoden zur Berechnung von Grapheneigenschaften wie Durchmesser, Anzahl der Blätter oder Artikulationen. Für diese Berechnungen wird zum Teil auch auf Funktionen des Perl-Moduls `Graph` zurückgegriffen. Des Weiteren existiert die private Funktion `_writeGraphImage()`, die mit dem Programm GraphViz und dem entsprechenden Perl-Modul `GraphViz`[8] den Graphen visualisieren und als PNG-Datei abspeichern kann. Diese Funktion wird von der Funktion `toZVF()` aufgerufen, wenn die Konfigurationsvariable `CREATEGRAPHIMAGE` gesetzt ist. Mit Hilfe des Moduls `TopologyGenerationFunctions` können auch verschiedene Topologien erstellt werden wie zum Beispiel Row, Circle, Star, ... (siehe auch Abschnitt „TopologyGenerationFunctions“ auf Seite 30).

[8] http://search.cpan.org/~lbrocard/GraphViz-2.04/lib/GraphViz.pm

TopologyGenerationFunctions

Das Perl-Modul `TopologyGenerationFunctions` stellt statische Funktionen zur Verfügung, um verschiedene Topologien zu generieren. Neben vielen Topologieklassen können auch Zufallstopologien erstellt werden, bei denen entweder die Anzahl der Knoten und Kanten oder die Anzahl der Knoten sowie der Clusterkoeffizient vorgegeben sind. Eine Liste aller generierbaren Topologieklassen ist in der Bedienungsanleitung auf Seite 24 angegeben.

RIPPacket

Die Klasse `RIPPacket` entspricht einem RIP-Paket. Es werden nur die zur Berechnung der Konvergenzeigenschaften notwendigen Attribute gespeichert. Dies sind der sendende Router, der Zeitpunkt, an dem das Paket gesendet wurde, das Netz, über das das Paket gesendet wurde, die Gesamtlänge des Pakets (in Bytes) sowie ein Hash, welches jeder Route des Pakets die entsprechende Metrik zuordnet.

Die Klasse `RIPPacket` wird von der Klasse `RIPParser` benötigt. Sie kann bei der Implementierung anderer Protokollparser durch entsprechende Klassen oder Datentypen (wie zum Beispiel Hashes) ersetzt werden.

Parser

Die abstrakte Klasse `Parser` dient vor allem zur Berechnung der Konvergenzeigenschaften. Sie wird jedoch immer genutzt, wenn die gesendeten Pakete eines Routingprotokolls geparst werden müssen. Zur Erzeugung der Klasse müssen die `DumpFile`-Objekte des zu berechnenden Durchlaufs sowie die zugehörige Topologie in Form eines `Topology`-Objekts übergeben werden. Daneben wird noch der Zeitpunkt eines etwaigen Router- oder Deviceausfalls sowie eine Option zum Parsen übergeben. Je nach Parseroption wird die Konvergenzzeit bis zum oder ab dem Routerausfall berechnet. Um die Auswertung verschiedener Routingalgorithmen zu ermöglichen, muss jeweils eine Spezialisierung dieser Klasse für das entsprechende Protokoll implementiert werden. Für RIP wurde in dieser Arbeit die abgeleitete Klasse `RIPParser` implementiert.

Die beiden Methoden `getResultHash()` und `printHumanReadableDumpFile()` müssen bei jeder `Parser`-Spezialisierung implementiert sein und dienen der Berechnung der Konvergenzeigenschaften sowie der Erzeugung einer Datei in von Menschen lesbaren Format. `getResultHash()` gibt ein Hash zurück, welches alle Konvergenzeigenschaften des berechneten Durchlaufs enthält. Dieses `resultHash` wird im Abschnitt „Result“ auf Seite 31 beschrieben. Das `File`-Objekt, welches von `printHumanReadableDumpFile()` zurückgegeben wird, enthält alle wichtigen Informationen der einzelnen Pakete, die chronologisch sortiert sind.

RIPParser

Die von `Parser` abgeleitete Klasse `RIPParser` implementiert die Funktionenen `getResultHash()` und `printHumanReadableDumpFile()` für das RIP-Protokoll.

Zum Generieren der `RIPPacket`-Objekte werden die `DumpFile`-Objekte mit den privaten Funktionen `_flattenDump()` und `_getPacketProperties()` genutzt. Aus den `DumpFile`-Objekten wird mittels dieser Funktionen eine Liste von `RIPPacket`-Objekten erstellt, die chronologisch sortiert sind.

Zur Berechnung der Konvergenzeigenschaften werden die privaten Funktionen `_getFirstTimeStamp()` und `_getLastTimeStamp()` implementiert, welche den Anfangs- und Endzeitpunkt der Konvergenzzeit berechnen. Danach werden mittels der privaten Funktion `_removePackets()` alle Pakete zurückgeliefert, welche innerhalb der beiden errechneten Zeitpunkte versendet wurden, um daraus die Anzahl der Pakete sowie den Updatetraffic zu berechnen.

Result

Die Klasse `Result` verwaltet die Ergebnisse einer Simulation. Ein solches Objekt kann entweder durch ein `Simulator`-Objekt erzeugt werden, um eine gerade beendete Simulation auszuwerten, oder vom Programm selbst, um die Konvergenzeigenschaften einer Simulation (oder eines Durchlaufs) neu zu berechnen. Zur Berechnung von Konvergenzeigenschaften wird kurzzeitig ein Parser-Objekt erzeugt, welches die Berechnungen für das jeweilige Protokoll durchführt.

Zu jedem Durchlauf wird ein `resultHash` gespeichert, welches von der entsprechenden `Parser`-Spezialisierung geliefert wird.

Die Funktion `_getAverage()` generiert aus mehreren Durchläufen ein `averageHash`, welches ähnlich wie das `resultHash` die Konvergenzeigenschaften einer Simulation enthält. Hier wird jedoch der Durchschnitt aus mehreren Durchläufen errechnet. Die Funktion kann auch mit verschiedenen Topologien innerhalb der dem `Result`-Objekt übergebenen `resultFile`-Objekt umgehen. In diesem Fall werden alle Topologien zusammengefasst. Die Funktion `getAverageLine()` gibt die Durchschnittswerte in je einer Zeile pro Topologie in einem von gnuplot lesbaren Format zurück.

Die Hashes werden in der Bedienungsanleitung auf Seite 18 genauer beschrieben.

Scenario

Die Klasse `Scenario` beinhaltet alle Informationen einer Messung und Methoden, um diese mittels VNUML durchzuführen. Dazu gehören ein `Topology`-Objekt und ein `SimulationDescriptionFile`-Objekt.

Ein `Scenario`-Objekt enthält alle Methoden, die während der Online-Phase eines Durchlaufs das VNUML-Netzwerk sowie die tcpdump-Prozesse steuern. Mit der privaten Methode `_getDevices()`

werden alle Devices des VNUML-Netzwerks mittels des UNIX-Befehls `ifconfig`[9] und regulären Ausdrücken herausgesucht, damit die Funktion `startDumps()` für jedes Device einen tcpdump-Prozess starten kann.

Des Weiteren steht für jeden Befehl, der in Test-Scripten genutzt werden kann, eine Funktion zur Verfügung, die diesen Befehl implementiert. Diese Funktionen werden durch ein `Simulator`-Objekt aufgerufen, welches das aktuelle Test-Script parst.

Simulator

Eine Simulation wird durch die Klasse `Simulator` repräsentiert.

Diese bekommt ein `ExecutionDescription`-Objekt übergeben, in dem aufgeführt ist, welche Topologien mit welchen Test-Scripten wie oft simuliert werden sollen. Es wird ein `Scenario`-Objekt mit der entsprechenden Topologie und dem Test-Script in Form eines `TestCaseFile`-Objekts erzeugt, um einen Durchlauf zu starten. Das Test-Script wird Zeile für Zeile gelesen und die jeweiligen Befehle durch Methodenaufrufe des `Scenario`-Objekts ausgeführt. Nach beendeter Simulation wird ein `Result`-Objekt erzeugt. Darin werden mit Hilfe eines `RIPParser`-Objektes die tcpdump-Dateien (in Form von `DumpFile`-Objekten) geparst und in `RIPPacket`-Objekte überführt. Die Konvergenzzeit wird schließlich im `RIPParser`-Objekt anhand einer chronologisch sortierten Liste der `RIPPacket`-Ojekte ermittelt.

Die Klasse `Simulator` ist so angelegt, dass Simulation und Auswertung später auch durch Threads parallelisiert werden können. Für diese Arbeit war eine Parallelisierung jedoch nicht erforderlich, sodass sie auch noch nicht implementiert wurde.

[9] Manual für ifconfig: http://linux.die.net/man/8/ifconfig

4 Verzeichnisse

Abbildungsverzeichnis

Listings

Literatur

[1] *Gnuplot Manual - Fit Command*. Online erreichbar unter http://www.gnuplot.info/docs/node82.html. Abgerufen am 08.01.2010.

[2] *Maxima Manual - lsquares Packet*. Online erreichbar unter http://maxima.sourceforge.net/docs/manual/en/maxima_62.html. Abgerufen am 08.01.2010.

[3] Postel, Jonathan B.: *RFC768 - User Datagram Protocol (UDP)*. IETF, August 1980.

[4] Postel, Jonathan B.: *RFC791 - Internet Protocol*. IETF, September 1981.