Universität Koblenz-Landau

Abteilung Koblenz
Fachbereich Informatik

Prof. Dr. Christoph Steigner
Dipl.-Inf. Harald Dickel

Diplomarbeit
AG Rechnernetze und Rechnerarchitektur

Implementation und Simulation von RIP-MTI

25. April 2005

Tobias Koch
<tkoch@uni-koblenz.de>
http://www.uni-koblenz.de/∼tkoch/

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Routingprotokoll RIP, welches bereits in Vorstufen des heutigen Internet zum Einsatz kam, birgt aufgrund seiner Wurzeln im Distance-Vector-Algorithmus einige Nachteile, die sich in der maximalen Größe eines durch RIP verwalteten Netzes und den Konvergenzeigenschaften widerspiegeln.

Die negativen Eigenschaften aufgreifend wurde in [Sch99] gezeigt, daß diese durch wenige Ergänzungen am Algorithmus gänzlich eliminiert werden können. Das dort vorgestellte Verfahren speichert Topologieinformationen über das Netz, in dem es eingesetzt wird. Somit entstand RIP-MTI, welches laut [Sch99] die Lücke zwischen RIP und anderen Routingprotokollen wie OSPF schließt.

Aufbauend darauf untersuchte [Kle01] mit Hilfe eines eigens dafür in Java entwickelten, diskreten Netzwerksimulator RIP-MTI näher. Ergebnis der Arbeit war, daß RIP-MTI nicht in allen Situationen die Unzulänglichkeiten von RIP ausschalten kann.

Die Motivation für die vorliegende Arbeit ist daher, Möglichkeiten zur näheren und zwar primär realitätsnäheren Untersuchung von RIP-MTI zu schaffen, damit zum einen die Ergebnisse aus [Kle01] überprüft und ihre Auswirkungen im Praxiseinsatz von RIP-MTI aufgezeigt werden können.

Folgende Fragen stehen in diesem Rahmen zur Diskussion:

- Welche Erweiterungen sind an RIP vorzunehmen? / Wie funktioniert RIP-MTI?
- In welchen Umgebungen kann die Arbeit von RIP-MTI in beliebig großen Netzen auch in den Laboren der Universität durchgeführt werden?

- Wie ist RIP-MTI zu integrieren/implementieren?
- Wie können diese Untersuchungen durchgeführt werden?

Diese Arbeit soll als Leitfaden für nachfolgende Untersuchungen verstanden werden. Neben den vorgestellten Programmen beinhaltet sie erste Ansätze, die es weiter auszubauen gilt.

Kapitel 2

Distance-Vector-Routing

Einen Grundbaustein für diese Arbeit bildet die Familie der *Distance-Vector-Routing-Protokolle* (im Folgenden kurz *DVR* genannt). Diese Algorithmen wurden bereits im ARPANET zur Routenberechnung eingesetzt. Anhand eines Vertreters soll nun zunächst kurz auf die Arbeitsweise, Funktionalität sowie auf die Grenzen eingegangen werden. Für eine weiterführende und detailliertere Erklärung der folgenden Abschnitte sei auf [Mal98] und [PD04] verwiesen.

2.1 Der Algorithmus

Routing bezeichnet die Aufgabe, einen *Pfad* zwischen einem Sender und einem zu erreichenden Ziel zu finden. Im Sinne des IP-Modells läßt sich dies als eine Folge von Routern zwischen Ursprungs- und Zielnetzwerk beschreiben. Wie gelangt nun ein Router an die zur Bestimmung notwendigen Informationen?

Eine Kategorisierung der verschiedenen Routingprotokolle findet meist anhand der Art und des Umfangs der ausgetauschten Information statt. DVR ist dadurch geprägt, daß nur eine geringe Menge an Daten ausgetauscht wird. So schickt jede am Routing teilnehmende Entität einen Vektor bestehend aus den von ihr aus erreichbaren Zielen und den damit verbundenen Kosten (*Metrik*) an seine Nachbarn (vgl. Tabelle 2.1).

Initial sind dabei nur die direkt angeschlossenen Ziele/Netzwerke enthalten. Weitere werden durch die von den Nachbarn erhaltenen Nachrichten hinzugefügt. Der Algorithmus läßt sich folgendermaßen beschreiben:

1. Verwalte eine Tabelle mit einem Eintrag für jedes mögliche Ziel. Speichere zu

Ziel	Metrik
141.26.65.0/24	1
141.26.66.0/24	1
141.26.69.0/23	1
141.26.0.0/22	3
141.26.11.32/27	4
141.26.70.0/24	2
141.26.201.128/23	8

Tabelle 2.1: Beispiel Distance-Vector (IP-Modell)

jedem Ziel N die Distanz/Metrik[1] M sowie den ersten Router R des Pfades.

2. Sende in regelmäßigen Zeitintervallen ein Update an alle Nachbarn. Dieses beinhaltet eine Menge von Tupeln der Form $(Ziel, Metrik)$. Zu jedem Eintrag der Routingtabelle existiere ein entsprechendes Tupel.

3. Bei Ankunft eines Updates von Nachbar R' berechne die Metrik M' für jeden Eintrag der Nachricht wie folgt:

$$M' = m_i^{R'} + M_{R'} \quad \text{mit} \quad \begin{array}{ll} m_i^{R'} & \text{Metrik zum Router } R' \\ M_{R'} & \text{die von } R' \text{ mitgeteilte Metrik} \end{array}$$

   Ist M' für ein Ziel N kleiner als der bisher in der Routingtabelle eingetragene Wert, so ersetze den Eintrag für N mit dem Tupel (N, M', R'). Ist R' der Router, der derzeit bereits für N genutzt wird, also $R' = R$, so übernimm M' in jedem Fall.

Dieses Vorgehen stützt sich bisher auf die Annahme einer statischen Topologie. Das Erkennen von und Reagieren auf Änderungen dieser hängt von der jeweiligen Implementierung des DV-Algorithmus ab. Für RIP wird dies im folgenden Abschnitt kurz erläutert.

2.2 RIP - Routing Information Protocol

RIP ist die bekannteste und häufigst eingesetzte Implementierung des Distance-Vector-Algorithmus, was wohl zu einem Großteil durch deren Verwendung im Pro-

[1] Distanz und Metrik sind nicht synonym zu setzen, werden in vielen Anwendungsfällen aber so betrachtet.

gramm *routed* der Berkeley-Unix-Distribution und der *Xerox Network Systems Architektur* begründet ist.

Wie bereits angesprochen, stellt sich nun, nachdem die grundlegende Arbeitsweise erklärt wurde, die Frage, wie Topologieänderungen in RIP gehandhabt werden. Zur Bestimmung der geeigneten Route zu einem Ziel wird das Minimum der Kosten über alle Nachbarn untersucht. Ändert sich allerdings die Topologie, so auch die Menge der Nachbarn[2], was eine Neuberechnung der Routen zur Folge haben muß.

Um Änderungen, wie z.B. den Ausfall eines Routers, zu erkennen, müssen Timer eingeführt werden. RIP-Router senden im Abstand von 30 Sekunden Updatenachrichten an ihre Nachbarn. Trifft für eine gewisse Zeitspanne keine Nachricht von einem Nachbar ein, so kann davon ausgegangen werden, daß entweder der Router oder das dazwischenliegende Netzwerk ausgefallen ist. Die zugehörigen Routen werden als ungültig markiert. In RIP wurden 180 Sekunden für dieses Intervall gewählt, um gelegentliche Paketverluste im Netzwerk einzubeziehen.

Bisher ist allerdings noch offen, wie die Unerreichbarkeit eines Netzes, was einer Metrik von ∞ entspricht, dargestellt wird. Um die Entscheidung für die Wahl dieser Größe zu erklären, muß zuerst ein Problem des Distance-Vector-Routings vorgestellt werden: *counting-to-infinity*, kurz *CTI*.

Es sei das einfache Netzwerk aus Abbildung 2.1 bestehend aus 3 Routern und 4 Netzen gegeben. Aufbauend darauf soll nun der Aufbau der Routingtabellen und das Entstehen eines CTI-Ereignisses gezeigt werden.

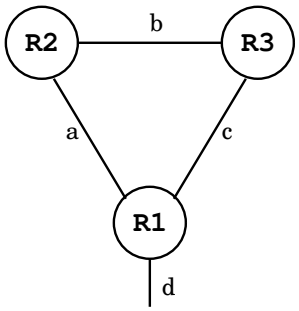


Abbildung 2.1:

Initial besitzen die Router die in den Tabellen 2.2(a)-2.2(c) dargestellten Informationen. Nur die lokal angeschlossenen Netze sind bekannt. Nach Austausch der Routingtabellen mit den Nachbarn stabilisieren sich die Tabellen auf die in 2.2(d)-2.2(f) angegebenen Inhalte. Abhängig vom Ablauf des Nachrichtenaustauschs sind auch andere Konfigurationen denkbar, die sich aber nicht grundlegend von der angegebenen unterscheiden.

Ausgehend von dieser Konfiguration falle nun die Verbindung zum Netzwerk d am Router $R1$ aus. $R1$ teilt dies $R2$ und $R3$ durch Übermittlung seines neuen Distanzvektors mit, der nun den Eintrag (d, ∞) beinhaltet. Noch vor Eintreffen dieser Nachricht bei $R2$ sendet dieser sein halbminütliches Update an $R3$, welches bei diesem allerdings erst nach der Nachricht von $R1$ eintrifft (vgl Abb. 2.2(a)).

Aufgrund dieser zeitlichen Abfolge verarbeitet $R3$ eine nicht mehr gültige Infor-

[2] Zumindest für eine nichtleere Teilmenge der Knoten.

Ziel	Metrik	via
a	1	-
c	1	-
d	1	-

(a) R1

Ziel	Metrik	via
a	1	-
b	1	-

(b) R2

Ziel	Metrik	via
b	1	-
c	1	-

(c) R3

Ziel	Metrik	via
a	1	-
c	1	-
d	1	-
b	2	R2

(d) R1

Ziel	Metrik	via
a	1	-
b	1	-
c	2	R3
d	2	R1

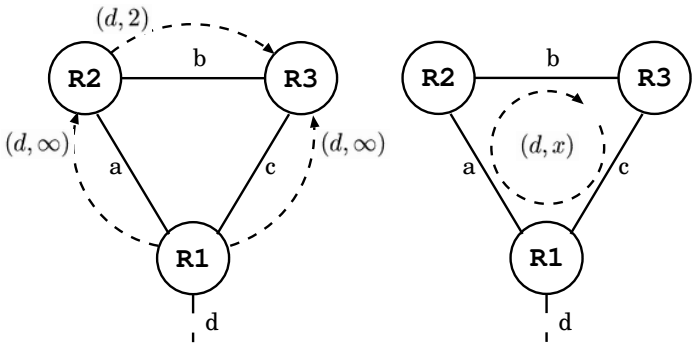
(e) R2

Ziel	Metrik	via
b	1	-
c	1	-
a	2	R2
d	2	R1

(f) R3

Tabelle 2.2: Routingtabellen zu Abb. 2.1

mation von $R2$, daß Netzwerk d mit einer Metrik von 3 über $R2$ erreicht werden kann. Wie im Algorithmus vorgeschrieben, wird dies an $R1$ propagiert, welcher nun wieder die Erreichbarkeit von d feststellt. Wie in Abbildung 2.2(b) und Tabelle 2.3 dargestellt, kreist diese Nachricht nun im Netz, wobei die Metrik mit jeder Verarbeitung inkrementiert wird. Dieses Verhalten wird, wie bereits zuvor erwähnt, *counting-to-infinity* genannt.



(a) Ausfall von d (b) CTI

Abbildung 2.2: Entstehung eines CTI-Ereignisses

Da diese Erhöhung der Metrik zumindest in der Theorie bis ins Unendliche fortschreitet und somit die Konvergenz der Routinginformationen ebenso lange dauert,

Router \ Zeit	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
$R1$	∞	∞	∞	4	4	4	7	7	7	10
$R2$	2	∞	∞	∞	5	5	5	8	8	8
$R3$	2	∞	3	3	3	6	6	6	9	9

Tabelle 2.3: Verbreitung der Information zur Erreichbarkeit von d

wurde in der Praxis der Zahlenwert 16 gewählt, um ∞ bzw. die Nichterreichbarkeit zu repräsentieren. Diese Zahl mußte als Kompromiss zwischen Netzwerkgröße und Konvergenzgeschwindigkeit gewählt werden. Als Folge ergibt sich ein maximaler Netzwerkdurchmesser von 15.

2.3 Erweiterungen

2.3.1 Split horizon

Die oben geschilderte Problematik beruht u.a. auf einer gegenseitigen Fehlinformation der Teilnehmer. Unter *split horizon* versteht man daher eine Änderung der Strategie, wie mit neu gewonnenen bzw. erlernten Informationen umgegangen werden soll. So ist es im Allgemeinen nicht nützlich, dem Nachbarn, von dem man eine Information erhalten hat, diese erneut mitzuteilen. In der einfachsten Form des split horizon werden daher in einem Routingupdate an einen Nachbarn jene Routen, die von diesem mitgeteilt bzw. gelernt wurden, vernachlässigt. Dies läßt sich auch als "*Belehre nicht den Lehrer*"-Prinzip ausdrücken.

In einer weiteren Form, *split horizon with poisoned reverse* genannt, sind diese Routen zwar wieder enthalten, werden aber mit einer Metrik von ∞ versehen, sofern sie vom Empfänger der Nachricht gelernt wurden. Dieser Ansatz ist sicherer als das einfache split horizon. Besitzen zwei Nachbarn Routen, die auf den jeweils anderen verweisen, so werden diese Schleifen durch eine Unerreichbarkeitsnachricht sofort gelöst. Bei split horizon ohne poisoned reverse ist erst eine Zeitüberschreitung abzuwarten. Ein Nachteil ist allerdings, daß dieser Ansatz zu enorm großen Updatenachrichten führen kann, was eine signifikante Bandbreitennutzung zur Folge hat.

Es kann gezeigt werden, daß split horizon with poisoned reverse Routingschleifen bestehend es zwei Routern verhindert. Für das Beispiel aus Abbildung 2.1 sind diese Maßnahmen also nicht ausreichend.

2.3.2 Triggered updates

Triggered updates sollen die Konvergenzgeschwindigkeit nach einer Topologieänderung verbessern. Sobald sich die Metrik einer Route ändert, ist sofort bzw. protokollabhängig nach einer kurzen Verzögerung eine Updatenachricht an die jeweiligen Nachbarn zu senden. Diese Regel resultiert in einer Kaskade von Nachrichten, sobald eine Änderung auftritt. Unter der Bedingung, daß während einer solchen Kaskade keine weiteren Änderungen im System stattfinden, läßt sich zeigen, daß counting-to-infinity nicht mehr auftritt.

Allerdings läßt sich diese Anforderung nicht erfüllen, da während einer solche Kaskade auch normale Updatenachrichten ausgetauscht werden und von der Kaskade noch nicht erreichte Router weiterhin falsche Informationen verbreiten.

Zusammenfassend sei somit festgehalten, daß die vorgestellten Maßnahmen counting-to-infinity-Ereignisse nicht verhindern können, aber diese mit einer geringeren Wahrscheinlichkeit auftreten. Alle Implementationen nutzen triggered updates für ungültige Routen und können diese für neue oder geänderte Routen verwenden. Ebenso wird üblicherweise split horizon mit optionalem poisoned reverse verwendet.

Kapitel 3

RIP-MTI

In [Sch99] wurde ein Algorithmus vorgestellt, der das counting-to-infinity-Problem gänzlich eliminieren sollte. Hierzu wird aus den empfangenen Updatenachrichten ein grobes Abbild der Netzwerktopologie geschaffen. Insofern grob, als daß festgestellt wird, ob ein Zyklus zwischen zwei angrenzenden Subnetzen besteht. Diese Vorgehensweise spiegelt sich in der Namensgebung wider: *RIP with minimal topology information - RIP-MTI.*

Vorab sei darauf hingewiesen, daß dieses Kapitel dem grundsätzlichen Verständnis von RIP-MTI dienen soll. Für einen tiefergehenden Einblick ist [Sch99] heranzuziehen.

3.1 Grundlagen

Notwendige Bedinungung zur Entstehung eines CTI-Ereignisses ist die Existenz einer Schleife in der Netzwerktopologie. In Abbildung 2.2(b) besteht diese zwischen den Knoten $R1, R2$ und $R3$. Eine Routingschleife (*routing loop*) bezeichnet nun eine Route bzw. einen Weg, der einen Knoten mehrmals passiert. Das Beispiel in Abbildung 3.1 zeigt solch einen Weg in Bezug auf $R1$.

Das Beispiel zeigt ebenso einen Spezialfall für Routingschleifen, der den Kern von RIP-MTI darstellt. Auf dem eingetragenen Weg liegt nicht nur zweimal $R1$, sondern auch $R3$, der gleichzeitig Ausgangspunkt des Weges ist. Weg ist hier aus Netzwerksicht natürlich mit Route gleichzusetzen.

Solch eine Route wird als *source loop* bezeichnet, da hier explizit der Ursprungsknoten mehrmals passiert wird. Schmid zeigte nun durch Widerspruch, daß sofern

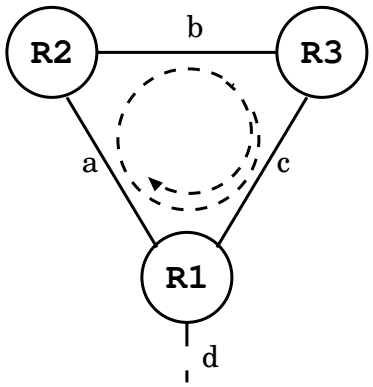
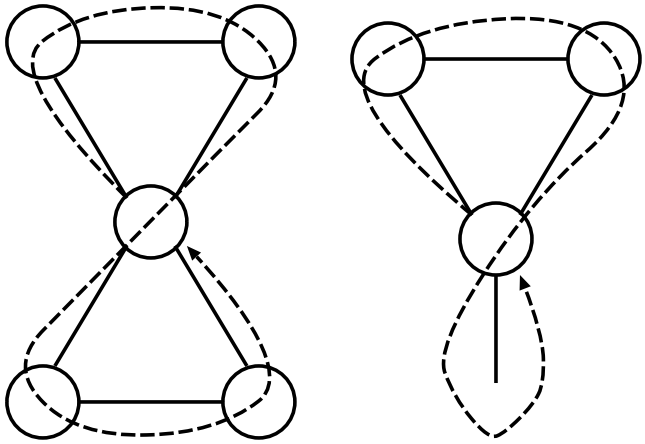


Abbildung 3.1: Routing loop

source loops durch jeden Router lokal vermieden werden, keine routing loops im Netzwerk entstehen. Daraus ergibt sich eine einfache Implikation: existieren keine source loops, so können CTI-Ereignisse nicht entstehen.

Unter der Bedingung, daß auf jedem beteiligten Router split horizon verwendet wird, konnte gezeigt werden, daß nur zwei mögliche source loops betrachtet werden müssen: X- und Y-Kombinationen.



(a) X-Kombination (b) Y-Kombination

Abbildung 3.2: Source loops

3.2 X-Kombinationen

Wie in Abbildung 3.2(a) ersichtlich, ergeben sich X-Kombinationen aus dem Zusammenfügen zweier Kreise. Betrachten wir nun zum weiteren Verständnis das Beispiel aus Abbildung 3.3.

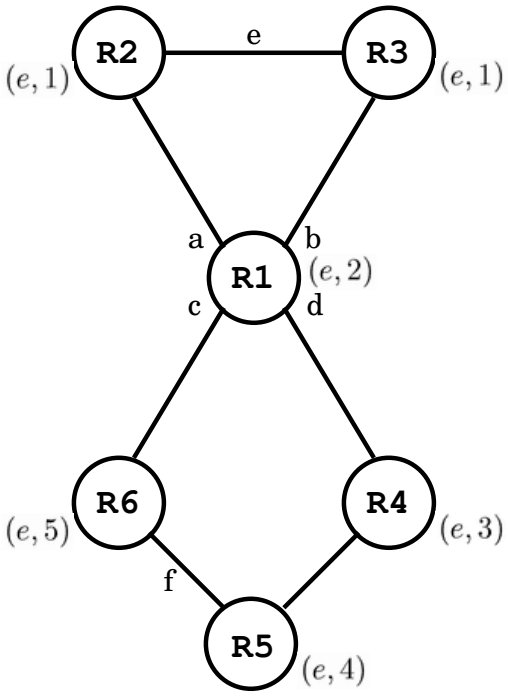
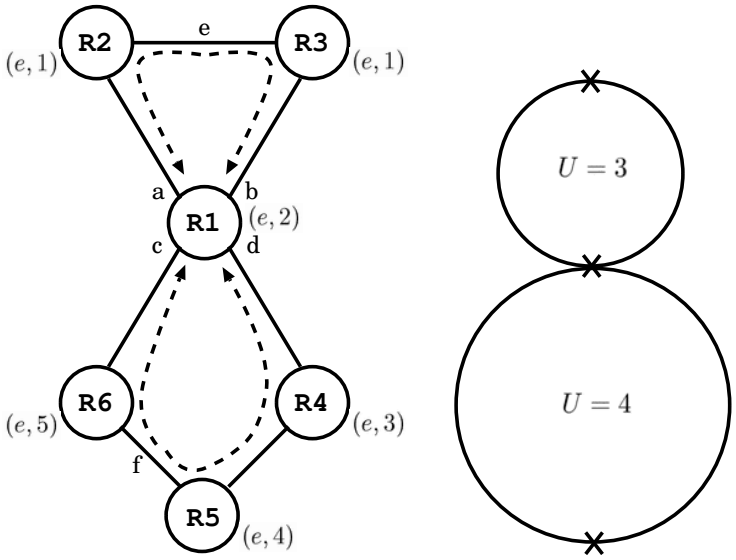


Abbildung 3.3: Beispiel X-Kombination

In dieser Topologie können zwei Teilnetze unterschieden werden: zum einen jenes aus den Routern $R1$, $R2$ und $R3$, sowie das aus $R1$, $R4$, $R5$ und $R6$. Den Mittelpunkt bildet der Knoten $R1$, der im Folgenden auch als *source router* bezeichnet wird.

Betrachten wir das Netzwerk nun aus Sicht von $R1$. Um das Subnetz e zu erreichen, kann $R1$ einen Weg über a oder b wählen. Alternative Wege über c oder d sind nicht brauchbar, da diese wieder, um e zu erreichen, $R1$ passieren müssten. Woher kann $R1$ aber diese Information gewinnen, da $R1$ nur seine direkten Nachbarn kennt, aber kein Wissen über den Aufbau des Netzes hat?

$R1$ weiß, daß zwischen den beiden Netzen a und b eine Kreisverbindung besteht. Grund dafür ist, daß die Erreichbarkeit von Netz e sowohl von $R2$, also über Netz a, als auch von $R3$ über b mitgeteilt wird. Gleichermaßen verhält es sich mit c und d. Hier kann beispielsweise durch die Erreichbarkeit von Netz f auf einen Kreis geschlossen werden. Dieser Prozess ist in Abbildung 3.4(a) dargestellt.



(a) Schleifenerkennung (b) Abstrahierte Sicht

Abbildung 3.4: Beispiel X-Kombination

Aufgrund der RIP-Nachrichten kann aber nicht nur auf die Existenz einer solchen Schleife geschlossen werden, sondern auch ihr Umfang bestimmt werden. Betrachten wir dies im vorliegenden Beispiel anhand des Netzes f. f ist von $R1$ aus gesehen über c mit einer Metrik m_c^f von 2 zu erreichen. Wird der Weg über das Netz d gewählt, beträgt die Metrik m_d^f 3. Der Umfang berechnet sich nun als

$$m_c^f + m_d^f - 1 = m_{c,d}^{R1,f,R1},$$

wobei m_c^f die Metrik zum Erreichen von f über das Netz c bezeichnet.

Analog berechnet sich für das Netz zwischen a und b ein Umfang von $m_{a,b}^{R1,e,R1} = 3$. Der Umfang ist in diesem Zusammenhang also als die Summe der Metriken/Kosten beim Beschreiten des Weges von $R1$ nach $R1$ zu sehen.

Abbildung 3.4(b) zeigt somit eine Darstellung dieser Ergebnisse in abstrahierter Form.

Es kommt nun die Frage auf, inwiefern sich diese Resultate im Rahmen der Zielsetzung verwenden lassen. Ziel ist es, source loops zu erkennen und zu vermeiden. Im Alltag entspricht dies in etwa dem Ziel: "*Der Weg von Frankfurt/Main nach*

Hamburg soll nicht über München und wieder Frankfurt/Main führen.[1]" Oder anders formuliert: "*Die Route Frankfurt/M. - München - Frankfurt/M. - Hamburg ist keine Alternative für die Route Frankfurt/M. - Hamburg.*"

Bezogen auf das vorliegende Beispiel heißt das: wird eine Route ermittelt, deren Metrik nach e und wieder zurück nach $R1$ kleiner ist als die Summe der beiden Netzumfänge, so liegt keine X-Kombination vor.

Angenommen es kommt zu einem Aufeinanderfolgen von Nachrichten, so daß $R6$ die Information von $R1$ erhält, e sei nicht zu erreichen. Weiterhin gibt aber $R5$ die Nachricht an $R6$, e ist mit Metrik 4 erreichbar, was $R5$ nun wiederum $R1$ mitteilen kann. $R1$ kann die Gültigkeit dieser Information nun ausschließen, denn

$$m_{a,b}^{R1,e,R1} + m_{c,d}^{R1,f,R1} = 3 + 4 = 7$$
$$\not> 7 = 5 + 2 = m_{b,c}^{R1,e,R1}$$

Die von $R6$ vorgeschlagene Route ist somit nicht kürzer als die Kombination der beiden Kreise. Eine X-Kombination ist nicht auszuschließen, da die beiden Zyklen Teil des Weges sein könnten.

3.3 Y-Kombinationen

Ein bereits erwähntes mögliches Szenario einer Y-Kombination ist in Abbildung 3.5 dargestellt. Das Netzwerk c ist direkt an $R1$ angeschlossen. Diese Information wird von $R1$ auch an $R2$ und $R3$ weitergeben, die das Netz c somit im Standardfall mit einer Metrik von 2 erreichen können.

Bei einem Ausfall von der Netzwerkverbindung von $R1$ nach c könnte es nun dazu kommen, daß $R2$ $R1$ die Erreichbarkeit von c mit einer Metrik von 4 mitteilt. Dies wurde bereits in Abschnitt 2.2 erörtert. $R1$ muß nun feststellen, ob die announcierte Route einen source loop enthält.

$R1$ ist auf die gleiche Weise wie bei X-Kombinationen bekannt, daß zwischen den Netzen a und b eine Verbindung besteht. Dieser Kreis $m_{a,b}^{R1,d,R1}$ besitzt einen Umfang von 3.

[1] Gewünschte Zwischenstopps sind zu vernachlässigen.

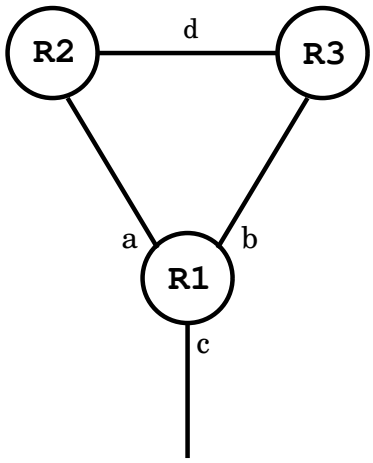


Abbildung 3.5: Beispiel Y-Kombination

Es gilt daher:

$$m_{a,b}^{R1,d,R1} = 3$$
$$\not> 3 = 4 - 1 = m_a^c - m_c^c$$

Dies bedeutet, die Differenz der von $R2$ announcierten Route nach c und der direkten Route von $R1$ nach c ist nicht kürzer als der Kreisweg zwischen den Netzen a und b an $R1$. Daher kann nicht ausgeschlossen werden, daß dieser Kreis Teil der Strecke über $R2$ nach c ist. Genau dies entspricht einer Y-Kombination, die nun nicht auszuschließen ist. Die Route über $R2$ stellt daher keinen Alternativweg dar.

Wäre die Ungleichung erfüllt, läge keine Y-Kombination vor und die Route wäre zu akzeptieren. Sie ist somit in allgemeiner Form ausschlaggebendes Kriterium bei der Prüfung auf Y-Kombinationen.

3.4 Der Algorithmus

Nachdem mit den bisherigen Schilderungen zunächst nur ansatzweise und beispielhaft die Arbeit von RIP-MTI erläutert wurde, soll nun der für diese Verfahrensweise notwendige Algorithmus vorgestellt werden.

Wie bereits erwähnt, handelt es sich bei RIP-MTI um eine Erweiterung von RIP, dessen Funktionsweise bereits in 2.1 auf Seite 3 vorgestellt wurde. Durch RIP-MTI sind keine grundlegenden Änderungen an dieser Vorgehensweise nötig. Es werden vielmehr einige Überprüfungen hinzugefügt.

Im folgenden werden diese Ergänzungen nun vereinfacht dargestellt. Eine detaillierte Beschreibung wird in Kapitel 5 nachgeholt.

3.4.1 Abbild der Netzwerktopologie

Zunächst müssen Informationen über die Topologie des Netzwerks gesammelt werden. Router R, der hier und im folgenden repräsentativ für die Knoten des Netzes steht, speichert diese Information beim Eintreffen der Erreichbarkeitsnachrichten in zwei Tabellen:

$\texttt{mincyc}_{i,j}$: In $\texttt{mincyc}_{i,j}$ wird die Länge des kleinsten Kreises zwischen den Interfaces i und j von R abgelegt. Trifft sowohl am Interface i als auch an j eine Erreichbarkeitsmeldung für das Netz d ein, so gilt:

$$\texttt{mincyc}_{i,j} = m_i^d + m_j^d - 1$$

Wird ein kürzerer Kreis festgestellt, wird der alte Wert überschrieben. Es ist sofort zu sehen, daß mincyc symmetrisch aufgebaut ist, da $\texttt{mincyc}_{i,j} = \texttt{mincyc}_{j,i}$.

$\texttt{minm}_i$: $\texttt{minm}_i$ speichert den kleinsten Kreis zwischen Interface i und einem beliebigen anderen Interface von R.

$$\texttt{minm}_i = \min(\texttt{mincyc}_{i,j}) \quad \forall j \in Interface(R), j \neq i$$

3.4.2 Test auf Source loops

Parallel dazu wird die Routingtabelle entsprechend RIP aufgebaut. Fällt nun die eingetragene Route zu einem Netz aus und es trifft eine Alternativroute ein, so wird auf mögliche X- und Y-Kombinationen geprüft.

X-Kombination: Führte die alte Route nach d über das Interface i von R und die neue soll über das Interface j führen, so muß folgende Ungleichung erfüllt sein, um eine X-Kombination auszuschließen:

$$\texttt{minm}_i + \texttt{minm}_j > m_{i,j}^{R,d,R} \quad \text{mit } m_{i,j}^{R,d,R} = m_i^d + m_j^d - 1$$

Y-Kombination: Entsprechend gilt bei Y-Kombinationen:

$$\texttt{minm}_k > m_k^{R,d} - m_l^{R,d} \quad \text{mit} \begin{cases} k = i, l = j & \text{falls } m_i^d > m_j^d \\ k = j, l = i & \text{falls } m_i^d < m_j^d \end{cases}$$

Die höhere Metrik m_k^d bestimmt somit die Wahl des minm -Eintrags. Ist $m_i^d = m_j^d$, so sind keine Y-Kombinationen möglich.

3.4.3 Funktionalität

Mit diesen Ergänzungen übernimmt RIP-MTI alle Eigenschaften von RIP. Auch RIP-MTI wählt immer die kürzeste Route zum Ziel. Lediglich im Falle eines Ausfalls einer vorhanden Route verweigert RIP-MTI die Aufnahme bestimmter Alternativrouten in die Routingtabelle.

Kapitel 4

Netzwerksimulatoren

Nachdem nun die theoretischen Grundlagen von RIP-MTI dargelegt wurden, stellt sich die Frage, wie der Einsatz in der Praxis näher untersucht werden kann. Schon in [Kle01] wurde ein in Java implementierter Simulator für RIP-MTI vorgestellt. Es lassen sich damit zwar beliebige Netze nachstellen, aber aufgrund der Architektur und der Möglichkeiten des Simulators ist der Abstraktionsgrad zur Realität sehr hoch. Ein Vergleich zu anderen Routingalgorithmen ist ebenfalls nicht durchführbar.

Da es eines der weiterführenden Ziele dieser Arbeit ist, eine Grundlage für die Untersuchung von Rechnernetzen und diverser Routingprotokolle zu schaffen, bestand zunächst die Aufgabe eine geeignete Umgebung für dieses Unterfangen zu identifzieren.

4.1 ns-2

Bei *ns-2*[1] handelt es sich nach eigener Definition um einen diskreten Ereignis-Simulator. Die Anwendung besteht aus zwei Kompenten:

1. einem in C++ geschriebenen objektorientierten Simulator sowie
2. einem Interpreter für Benutzereingaben, welcher in OTcl, einer objektorientierten Erweiterung zu Tcl, verfasst wurde.

Die compilierte C++-Ebene dient einer höheren Effizienz und kürzeren Ausführungszeit der Simulation. Durch Skripte in OTcl kann der Benutzer wieder-

[1] Erhältich auf [ns2].

um Netzwerktopologie, Applikationen usw. definieren. Die dort definierten Objekte können durch eine Tcl/C++-Schnittstelle, die als verbindendes Element dient, bereits kompilierten C++-Objekten zugeordnet werden, deren Verhalten in C++ definiert ist.

Der Ablauf einer Simulation hängt vom zeitlichen Auftreten von Ereignissen ab. Diese werden durch einen Scheduler verwaltet. Eine sortierte/geordnete Datenstruktur, welche im Standardfall eine einfache verkettete Liste darstellt, dient der Speicherung der Ereignisse, die nacheinander ausgeführt werden.

Diese Eigenschaften ähneln stark dem in [Kle01] vorgestellten Simulator, dem ebenfalls eine diskrete Ereignissimulation zu Grunde liegt.

Eine der bekannten Einschränkungen von ns-2 liegt beispielsweise im Modell für one-way TCP. Die verwendete Fenstergröße wird nicht dynamisch angepasst, Segment- und ACK-Nummerierung findet in Paketeinheiten statt und zum Verbindungsauf-/abbau wird nicht SYN/FIN verwendet.

4.2 netkit

Netkit ist eine Sammlung diverser Open-Source-Programme, die die Emulation von Netzwerken ermöglichen. Dabei stützt sich Netkit primär auf *User Mode Linux* (s. [Dik]). *UML* ist eine Erweiterung des Linux-Kernels, die es ermöglicht, den Kernel als normalen Userprozess auf einem Linux-Hostsystem auszuführen.

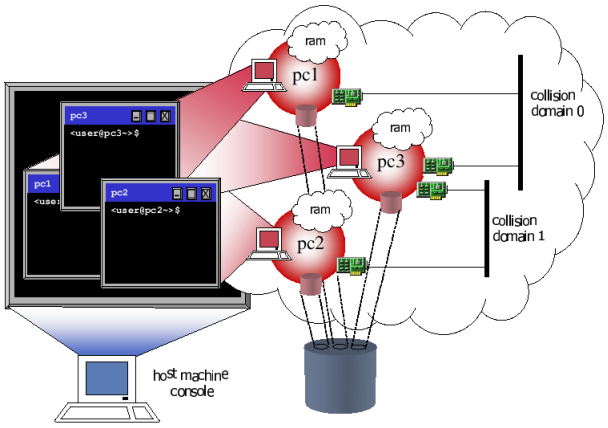


Abbildung 4.1: Aufbau virtuelle Maschinen [net]

Somit ist es möglich, mehrere virtuelle Linux-Instanzen auf einem physischen System zu starten. Jeder Instanz werden individuell Speicher und weitere Ressourcen

zugeordnet. Das Dateisystem wird in einer einer Datei auf dem Hostsystem abgebildet (siehe Abb. 4.1).

Das Starten der Instanzen erfolgt über Python-Skripte, die allerdings nur eine minimale Konfiguration der virtuellen Maschinen vornehmen. Lediglich die Zuweisung von Hostnamen und Zugehörigkeit der Netzwerkschnittstellen zu virtuellen Kollisionsdomänen wird vorgenommen (vgl. Eingabe 4.1). Die Konfiguration von IP-Adressen usw. muß innerhalb der virtuellen Instanzen selbst geschehen.

Ein-/Ausgabe 4.1 Starten einer virtuellen Maschine mit Netkit

```
tkoch@becks:~$ vstart r1 --eth0=A --eth1=B --new
```

Der Vorteil dieser Architektur ist, daß der komplette Linux-Netzwerkstapel untersucht werden kann. Diese Untersuchungen lassen sich sehr realitätsnah durchführen, da keinerlei Änderungen an Applikationen oder dem Kernel selbst vorgenommen werden. Zwar ist diese Lösung auf Linux beschränkt, was aber aufgrund der großen Verbreitung von Linux im Netzwerkbereich ausgeglichen wird. Weiterhin ist es nicht notwendig, neue Applikationen/Algorithmen innerhalb des Rahmens einer Simulationsumgebung zu entwickeln, da jede Linux-Applikation auch innerhalb einer UML-Umgebung lauffähig ist.

4.3 VNUML

VNUML wurde an der Universität von Madrid entwickelt und baut ebenso wie Netkit auf User Mode Linux auf (vgl. [vnu]). Daraus ergibt sich grundsätzlich eine gleiche Mächtigkeit der beiden Systeme.

Im Bezug auf die Konfiguration der virtuellen Instanzen sowie auf die Ausführung von Simulationen bietet VNUML allerdings einen entscheidenden Vorteil: der Aufbau eines Simulationsszenarios mit der Konfiguration aller beteiligter virtueller Instanzen wird in XML beschrieben. Ein Parser baut die Simulation anhand dieser Beschreibung auf und verwaltet sie. Dem Benutzer bleiben somit UML-spezifische Programme und andere Konfigurationsschritte verborgen. Ebenfalls lassen sich aufgrund dieser VNUML-Sprache sehr einfach automatisch komplexe Netzwerke entwerfen.

Diese Synthese der Vorteile von User Mode Linux und einer Beschreibungssprache in XML führte dazu, VNUML für die weiteren Betrachtungen als geeignete Simulationsumgebung auszuwählen. Um einen tieferen Einblick in diese Anwendung zu

gewinnen, die als Grundlage für spätere Arbeiten verwendet werden kann, wird im weiteren nun die Installation und Konfiguration von VNUML beschrieben.

4.3.1 Installation

VNUML befindet sich noch in reger Entwicklung. Dieser Abschnitt erläutert die Installtion von VNUML 1.5 unter Debian Sarge (3.1). Aber auch unter anderen Linux-Distributionen ist VNUML problemlos einsetzbar.

Von VNUML werden zunächst diverse Perl-Module benötigt. Diese können zwar bei bestehender Internetverbindung auch automatisch im Installationsprozeß eingerichtet werden, es empfiehlt sich aber, diese zunächst in den Paketen der Distribution zu suchen, da sie somit auch der Wartung durch den jeweiligen Paketmanager unterstehen. Tabelle 4.1 zeigt eine Gegenüberstellung der benötigten Module zu vorhanden Debian-Paketen.

Perl-Modul	Debian-Paket
XML-Parser-2.31	`libxml-parser-perl`
XML-RegExp-0.03	`libxml-regexp-perl`
libxml-perl-0.07	`libxml-perl`
XML-DOM-1.42	`libxml-dom-perl`
XML-Checker-0.13	`libxml-checker-perl`
TermReadKey-2.21	`libterm-readkey-perl`
Math-Base85-0.2	
Net-IPv4Addr-0.10	`libnetwork-ipv4addr-perl`
Net-IPv6Addr-0.2	
Exception-Class-1.20	`libexception-class-perl`

Tabelle 4.1: Perl-Module

Weiterhin werden die XML2-Bibliothek sowie `expat2` zum Parsen der XML-Dateien benötigt. Auch diese Pakete sind in jeder gängigen Distribution enthalten.

Eingabe 4.2 zeigt die notwendigen Schritte zur Installation. Die Notwendigkeit zur Erzeugung des Verzeichnisses unter `/usr/local/share` beruht auf einem Fehler im Installationsprozeß, der damit einfach umgangen werden kann. Nach dem Aufruf von `make install` kommt es ggf. noch zur Konfiguration von CPAN, um während der Installation fehlende Perl-Module einzurichten[2]. Hier ist vor allem auf die Angabe eines HTTP-/FTP-Proxys zu achten.

[2]Es sollte zuvor mittels `apt-get install ftp ncftp gnupg unzip lynx` sichergestellt werden, daß alle notwendigen Programme, die nicht zur Standardinstallation gehören, vorhanden sind.

Ein-/Ausgabe 4.2 Installation VNUML

```
becks:~# apt-get install libxml2 expat libxml-parser-perl
             libxml-regexp-perl libxml-perl libxml-dom-perl
             libxml-checker-perl libterm-readkey-perl
             libnetwork-ipv4addr-perl libexception-class-perl
...
becks:~/vnuml-1.5.0-1# ./configure --without-libxml --without-expat
becks:~/vnuml-1.5.0-1# make
becks:~/vnuml-1.5.0-1# mkdir -p /usr/local/share/doc/vnuml
becks:~/vnuml-1.5.0-1# make install
```

UML-Kernel und Root-Dateisystem[3] für die virtuellen Maschinen sollten nun nach `/usr/local/share/vnuml/kernels` bzw. `/usr/local/share/vnuml/filesystems` kopiert werden.

Damit ist die Installation abgeschlossen. Zur Verwendung werden noch einige UML-Werkzeuge benötigt, die mittels `apt-get install uml-utilities bridge-utils` installiert werden.

4.3.2 Konfiguration

Wie bereits erwähnt, erfolgt die Konfiguration eines Simulationsszenarios mittels XML. Ausgabe 4.1 zeigt beispielhaft Auszüge einer solchen Konfiguration, die in einen globalen Abschnitt, die Einrichtung der Netzwerke und der virtuellen Maschinen unterteilt ist. Genauer wird hierauf in Kapitel 6 eingegangen.

[3] Ein ebenfalls für später vorgestellte Beispiele verwendbarer Kernel und Root-Dateisystem sind auf der beiliegenden DVD in `vnuml/kernels` und `vnuml/filesystems` zu finden.

Konfiguration 4.1 Beispiel XML-Konfiguration VNUML

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

<vnuml>
   <global>
      <version>1.5</version>
      <simulation_name>rip_mti</simulation_name>
      <ssh_key>/root/.ssh/identity.pub</ssh_key>
      <automac offset="0"/>
      <ip_offset>192</ip_offset>
      <host_mapping/>
      <shell>/bin/sh</shell>
   </global>

   <!-- Networks -->
   <net name="0"/>
   <net name="1"/>
   <net name="2"/>
   <net name="3"/>

   <!-- Nodes -->
   ...
   <vm name="R1">
      <filesystem type="cow">/usr/local/share/vnuml/filesystems/root
      </filesystem>
      <kernel>/usr/local/share/vnuml/kernels/linux-2.6.10-1m</kernel>
      <boot>
        <con0>pts</con0>
      </boot>
      <if id="1" net="0">
         <ipv4 mask="255.255.255.0">10.0.0.2</ipv4>
      </if>
      <if id="2" net="1">
         <ipv4 mask="255.255.255.0">10.0.1.1</ipv4>
      </if>
      <if id="3" net="2">
         <ipv4 mask="255.255.255.0">10.0.2.1</ipv4>
      </if>
   </vm>
   ...
</vnuml>
```

Kapitel 5

Quagga - ein Routing Daemon

Mit VNUML ist eine Auswahl für die Simulationsumgebung getroffen. Es bleibt zu klären, wie RIP-MTI innerhalb dieser Umgebung getestet werden kann. Die Wahl fällt hier auf *Quagga* [qua]. Quagga ist eine Routing Software Suite und inoffizieller Nachfolger von *Zebra*, welches nicht mehr weitergeführt wird.

5.1 Architektur

Herzstück von Quagga ist weiterhin der Zebra-Daemon, der als Abstrakstionsschicht zum Unix-Kernel und den Quagga Clients fungiert. Diese Clients kommunizieren über eine eigene API mit dem Zebra-Daemon und sie sind es, die die eigentliche Routingfunktionalität mit sich bringen. Aktuelle Clients sind:

- `bgpd` - BGPv4+
- `isisd` - ISIS
- `ospfd` - OSPFv2
- `ospf6d` - OSPFv3 (IPv6)
- `ripd` - RIP V1 und V2
- `ripngd` - RIP v3 (IPv6)

Über eine Netzwerkkonsole sind diese Daemons ansprechbar. Die Konsole ist der anderer Routing Software angepasst und erinnert z.B. an Cisco IOS.

Diese Architektur macht es leicht, weitere Clients hinzuzufügen.

5.2 Quagga und RIP-MTI

Um Quagga um die RIP-MTI-Erweiterungen zu ergänzen, stehen zwei alternative Wege zur Auswahl:

1. Implementation eines neuen Quagga Clients auf Grundlage von `ripd` mit den gewünschen Eigenschaften.
2. Erweiterung des vorhandenden `ripd` um die MTI-Funktionalität.

Da mit `ripd` nun bereits ein RIP-Daemon existiert, der ständiger Wartung und Pflege unterliegt, hätte die erste Option zum einen unnötigen Mehraufwand auch nach der Fertigstellung bedeutet, zum anderen erhebliches Fehlerpotential in sich geborgen. In dieser Arbeit wird daher der zweite Ansatz verfolgt. Die Erweiterungen sollten dabei mit möglichst wenig Eingriff in den Originalquellcode erfolgen, um Anpassungen an neue Versionen schnell zu ermöglichen.

Die Anpassungen fußen dabei auf den Vorschlägen in [Sch99], die allerdings nicht so einfach wie dargestellt übernommen werden können. Quagga besitzt eine gänzlich andere Struktur als die dort aufgezeigten Routing Daemons.

Die Quellen von `ripd` befinden sich im ebenfalls so benannten Unterverzeichnis `ripd/` der Quagga-Suite. Die Hauptlast der RIP-Funktionalität liegt in `ripd.c`. Die MTI-Erweiterungen sind in `rip_mti.c` und der zugehörigen Headerdatei `rip_mti.h` untergebracht.

5.2.1 Strukturen und Konstanten

Hinzugefügte Strukturen und Konstanten werden in `rip_mti.h` definiert (vgl. Codesegment 5.1). `RIP_MTI_INFINITY` repräsentiert die größt mögliche Metrik zweier zusammengesetzter Routen. Im Standardfall mit `RIP_METRIC_INFINITY` = 16 ist dies also 31. Durch `RIP_MTI_ALIVE` wird wiederum die maximale Gültigkeitsdauer eines Eintrags in den Tabellen `minm` und `mincyc` festgelegt. Dieser Wert ergibt sich aus der Gültigkeitsdauer einer Route und der Zeit, bis diese endgültig aus der Routingtabelle gelöscht wird.

Die Werte in den Tabellen `minm` bzw. `mincyc` sind vom Typ `rip_mti_entry` (s. Codesegment 5.1), der einen Metrikwert und einen Zeitstempel mit dem letzten Bestätigungszeitpunkt enthält. `minm` und `mincyc` selbst sind vom selbsterklärenden Typ `vector`, der von Quagga plattformunabhängig zur Verfügung gestellt wird[1].

[1] Definiert in `libs/vector.h`.

Ein-/Ausgabe 5.1 Inhalt von `ripd/`

```
tkoch@becks:~/src/quagga$ ls -l ripd/
total 364
-rw-r--r--  1 tkoch tkoch  24306 Sep 30  2004 ChangeLog
-rw-r--r--  1 tkoch tkoch    619 Oct 18 17:19 Makefile.am
-rw-r--r--  1 tkoch tkoch  15719 Oct 19 17:13 Makefile.in
-rw-r--r--  1 tkoch tkoch  16715 Sep 30  2004 RIPv2-MIB.txt
-rw-r--r--  1 tkoch tkoch   7349 Sep 30  2004 rip_debug.c
-rw-r--r--  1 tkoch tkoch   1795 Sep 30  2004 rip_debug.h
-rw-r--r--  1 tkoch tkoch  51167 Sep 30  2004 rip_interface.c
-rw-r--r--  1 tkoch tkoch   6586 Feb 16 17:58 rip_main.c
-rw-r--r--  1 tkoch tkoch   6393 Feb 20 11:19 rip_mti.c
-rw-r--r--  1 tkoch tkoch    644 Feb 19 18:04 rip_mti.h
-rw-r--r--  1 tkoch tkoch  10777 Sep 30  2004 rip_offset.c
-rw-r--r--  1 tkoch tkoch   4643 Sep 30  2004 rip_peer.c
-rw-r--r--  1 tkoch tkoch  27418 Sep 30  2004 rip_routemap.c
-rw-r--r--  1 tkoch tkoch  14489 Sep 30  2004 rip_snmp.c
-rw-r--r--  1 tkoch tkoch  19041 Sep 30  2004 rip_zebra.c
-rw-r--r--  1 tkoch tkoch 105562 Feb 20 11:02 ripd.c
-rw-r--r--  1 tkoch tkoch    410 Sep 30  2004 ripd.conf.sample
-rw-r--r--  1 tkoch tkoch  11149 Oct 12 09:33 ripd.h
```

Codesegment 5.1 Konstanten `rip_mti.h`

```
#define RIP_MTI_INFINITY   RIP_METRIC_INFINITY * 2 - 1
#define RIP_MTI_ALIVE      RIP_TIMEOUT_TIMER_DEFAULT \\
                                + RIP_GARBAGE_TIMER_DEFAULT

struct rip_mti_entry {
  u_int32_t metric;
  time_t timer;
};
```

Mit diesem Datentyp dynamischer Größe lässt sich aufgrund diverser Zugriffsfunktionen, die von der eigentlichen Implementierung abstrahieren, komfortabel arbeiten.

Codesegment 5.2 zeigt die wichtigsten aus `ripd` verwendeten Strukturen. Die Struktur `rte` repräsentiert jeweils einen Eintrag der Erreichbarkeitsmeldung in einer RIP-Nachricht. `rip_info` wiederum beinhaltet diese und weitere Informationen im Bezug auf jeden Eintrag der Routing-Tabelle des Hosts.

Trifft beispielsweise eine RIP-Nachricht ein, die über das Ziel 10.0.1.0/24 benachrichtigt, so wird beim Bearbeiten der Nachricht in `ripd` - dieser Vorgang wird später noch genauer beschrieben - die Struktur `rte` verwendet. Ist dieses Ziel bereits in der Routingtabelle vorhanden, werden aus der Routingtabelle ebenfalls die Daten in eine Struktur `rip_info` übertragen und können so miteinander verglichen werden.

`rip_info` muß für die Implentierung von RIP-MTI um ein Datum erweitert werden. `oldmetric` speichert den letzten Metrikwert der Route kleiner als `RIP_METRIC_INFINITY`, sobald die Route als nicht erreichbar (= `RIP_METRIC_INFINITY`) deklariert wird.

5.2.2 Funktionen

rip_mti_init()

Diese Prozedur dient lediglich der Initialisierung der beiden Vektoren `minm` und `mincyc` . Es wird jeweils Speicher für einen Zeiger auf ein Element initialisiert. Dabei ist zu beachten, daß noch keine Elemente vom Typ `rip_mti_entry` hinterlegt werden, sondern nur ein Nullzeiger vorzufinden ist.

rip_mti_table_lookup()

Eine Hilfsfunktion, die in [Sch99] nicht zu finden ist, aber nun aufgrund der Implentation von `minm` und `mincyc` notwendig ist, ist `rip_mti_table_lookup()`. Sie dient dem vereinfachten Zugriff auf die beiden RIP-MTI-Tabellen. Die jeweils zu betrachtende Tabelle sowie ein Positionsindex in dieser werden als Argument übergeben. Rückgabewert ist das jeweilige Element an dieser Stelle und somit vom Typ `struct rip_mti_entry`.

Wie in Codesegment 5.3 ersichtlich, übernimmt diese Funktion auch eine Aufgabe, die nach [Sch99] für `rip_mti_init()` vorgesehen ist. Existiert an der angegebenen Position noch kein Eintrag, so sorgt zunächst die Vektorzugriffsfunkti-

Codesegment 5.2 Strukturen aus `ripd.h`

```
/* RIP routing table entry which belong to rip_packet. */
struct rte
{
  u_int16_t family; /* Address family of this route. */
  u_int16_t tag; /* Route Tag which included in RIP2 packet. */
  struct in_addr prefix; /* Prefix of rip route. */
  struct in_addr mask; /* Netmask of rip route. */
  struct in_addr nexthop; /* Next hop of rip route. */
  u_int32_t metric; /* Metric value of rip route. */
};
...
/* RIP route information. */
struct rip_info
{
  /* This route's type. */
  int type;

  /* Sub type. */
  int sub_type;

  /* RIP nexthop. */
  struct in_addr nexthop;
  struct in_addr from;

  /* Which interface does this route come from. */
  unsigned int ifindex;

  /* Metric of this route. */
  u_int32_t metric;

  /* Old metric used by RIP MTI */
  u_int32_t oldmetric;

  ,,,
};
```

Codesegment 5.3 Hilfsfunktion `rip_mti_table_lookup()`

```
struct rip_mti_entry *rip_mti_table_lookup(vector table,
                                           unsigned int pos) {

  struct rip_mti_entry *rme;

  if ((rme = vector_lookup_ensure(table, pos)) == NULL) {

    /* allocate new rme struct */
    rme = XMALLOC(MTYPE_RIP_MTI_ENTRY, sizeof(struct rip_mti_entry));
    if (rme == NULL) {
      zlog_err("rip_mti_table_lookup: could not allocate...");
    }
    /* malloc successful
       initialize entry as described in mti_init */
    else {
      rme->metric = RIP_MTI_INFINITY;
      rme->timer = time(0);
      vector_set_index(table, pos, rme);
    }
  }

  return rme;
}
```

on `vector_lookup_ensure()` dafür, daß bis zur angefragten Position entsprechend Speicherplatz innerhalb des Vektor zur Verfügung gestellt wird. Danach wird über ein von Quagga zur Verfügung gestelltes Makro[2] Speicherplatz für einen neuen Eintrag alloziiert. Ist dies geschehen, erfolgt die Initialisierung mit einem Wert von `RIP_MTI_INFINITY` und Speicherung in der angefragten RIP-MTI-Tabelle über die Funktion `vector_set_index()`.

rip_mti_minm()

Diese Methode birgt zwei Aufgaben in sich. Zum einen liefert sie den Metrikwert in `minm` zu einem als Argument übergebenen Interface. Dabei ist zu erwähnen, daß Interfaces in Quagga primär durch einen eindeutigen Intergerindex identifiziert werden. Zwar wird auch mit jedem Interface ein Name assoziiert, aber innerhalb von RIP-MTI bedeutet es nur Mehraufwand, diesen zu verwenden, da wiederum von dem Namen auf den Interfaceindex abgebildet werden muß.

Codesegment 5.4 Zugriff auf `minm` und Aufbau Netztopologie in `rip_mti_minm`

```
u_int32_t rip_mti_minm(unsigned int ifindex, int combi) {

  struct rip_mti_entry *rme;

   if ((rme = rip_mti_table_lookup(minm, ifindex)) != NULL) {
    if (combi <= rme->metric) {
      rme->metric = combi;
      rme->timer = time(0);
    }
    return rme->metric;
  }

  return RIP_MTI_INFINITY;
}
```

Zum anderen sammelt `rip_mti_minm()` Informationen über die Topologie des Netzwerks. Denn das zweite Argument `int combi` repräsentiert die Kombination zweier Routen zwischen dem angebenen Interface und einem beliebigen anderen. `rip_mti_minm()` vergleicht nun diesen Wert mit dem bisher für dieses Interface abgelegten Wert, ersetzt diesen, sofern `combi` kleiner ist, und bildet somit das Minimum für jedes Interface (vgl. Definition von `minm` auf Seite 15).

[2] Definiert in `libs/memory.h`.

rip_mti_routeok()

Mit `rip_mti_routeok()` nähern wir uns dem Herzstück von RIP-MTI, da es sich hierbei um eine Wrapper-Funktion handelt, die aus `ripd.c` aufgerufen wird und diverse Bedingungen überprüft. Der Funktion werden ein Element vom Typ `struct rip_info` mit Daten über die bisherige Route, sowie der korrespondierende Routing Table Entry (`struct rte`) und der Interfaceindex der neuen Route zum gleichen Ziel übergeben. Der Rückgabewert von 0 oder 1 zeigt an, ob die neue Route akzeptiert werden soll.

Codesegment 5.5 Wrapper-Funktion `rip_mti_routeok()`

```
int rip_mti_routeok(struct rip_info *oldroute, struct rte *rte,
                    struct interface *ifp) {

  u_int32_t routemetric;

  /* preconditions */
  if ((oldroute->ifindex == ifp->ifindex)
      || (rte->metric >= RIP_METRIC_INFINITY)
      || (oldroute->metric > RIP_METRIC_INFINITY))
    return 1;

  routemetric = (oldroute->metric < RIP_METRIC_INFINITY) ?
                        oldroute->metric : oldroute->oldmetric;
  return rip_mti_cycleok(ifp->ifindex, oldroute->ifindex,
                     rte->metric, routemetric,
                     (oldroute->metric == RIP_METRIC_INFINITY));
}
```

Die Funktion stellt sicher, daß

- das Interface, über das alte und neue Route empfangen wurden, verschieden sind.
- die Metrik der neuen Route kleiner `RIP_METRIC_INFINITY` ist.
- die Metrik der alten Router kleiner oder gleich `RIP_METRIC_INFINITY` ist.

Ist eine der Bedingungen nicht erfüllt, erfolgt keine weitere Bearbeitung durch RIP-MTI und die Route wird seitens RIP-MTI akzeptiert.

Desweiteren wird überprüft, ob die bisherige Route zu diesem Ziel noch verwendet wird (< `RIP_METRIC_INFINITY`). Anderenfalls wird die Route direkt mit Aufruf der Kernfunktion von RIP-MTI als alt deklariert und die alte Metrik der Route (s. Abschnitt 5.2.1 auf Seite 26) im weiteren verwendet.

rip_mti_cycleok()

Die bereits in Codesegment 5.5 auf der vorherigen Seite erwähnte Funktion `rip_mti_cycleok()` bildet den Kern von RIP-MTI ab. Sie untersucht nun effektiv zwei Routen auf X- und Y-Kombinationen und entscheidet, ob die neue Route von `ripd` akzeptiert werden soll.

Dafür werden als Parameter die Interfaceindices und Metriken der neuen und alten Route sowie der boolsche Wert `isold` übergeben. Letzterer gibt, wie bereits erwähnt, an, ob die alte Route nicht mehr erreichbar ist. Ist dies Fall wird nur überprüft, ob die neue Alternativroute zu akzeptieren ist; anderenfalls nutzt `rip_mti_cycleok()` die Informationen lediglich dazu, ein Abbild der Netzwerktopologie zu schaffen.

Diese Aufgaben werden in der Funktion etwas vereinfacht angegangen, so daß sie gerade mit dem Hintergrundwissen aus Kapitel 3 nicht intuitiv nachvollziehbar sein mögen. Daher soll hier detailliert auf die Arbeitsweise eingegangen werden.

Die in `rip_mti_cycleok()` verwendeten Variablen sind (vgl. Codesegement 5.6 auf der nächsten Seite):

`minmA, minmB`: Hierin wird der Metrikwert aus `minm` zu den beiden Interfaces abgelegt.

`combi` bezeichnet die Länge der Kombination beider Routen.

`yminm`: Diese Variable dient der Untersuchung auf Y-Kombinationen. Sie wird mit 2 initialisiert, was für den Fall, daß noch kein Zyklus am betrachteten Interface festgestellt wurde, der kürzesten Route von und wieder zum Interface entspricht.

`pos` bestimmt die Position innerhalb von `mincyc` im Bezug auf die angegebenen Interfaces. Die Berechnung macht sich, wie bereits geschildert, den symmetrischen Aufbau von `mincyc` zu Nutze.

Betrachten wir nun zunächst den Fall `isold == FALSE`, was bedeutet, daß nur Informationen über die Topologie gesammelt werden. Codesegment 5.7 auf Seite 33 zeigt den entsprechenden Auszug aus `rip_mti_cycleok()`.

Codesegment 5.6 Deklarationen und erste Berechnungen in `rip_mti_cycleok()`

```
int rip_mti_cycleok(unsigned int ifindexA, unsigned int ifindexB,
               u_int32_t metricA, u_int32_t metricB, int isold) {

  u_int32_t minmA, minmB, combi;
  u_int32_t yminm = 2;
  int pos;

  struct rip_mti_entry *rme;

  combi = metricA + metricB - 1; /** route combination */
  minmA = minmB = RIP_MTI_INFINITY;

  rip_mti_timeout();

  if (ifindexA > ifindexB)
    pos = ((ifindexA * (ifindexA - 1) / 2) + ifindexB);
  else
    pos = ((ifindexB * (ifindexB - 1) / 2) + ifindexA);
...
```

Codesegment 5.7 Aufbau der Topologieinformationen in `rip_mti_cycleok()`

```
1     if ((rme = rip_mti_table_lookup(mincyc, pos)) != NULL) {
    ...
        if (metricA > metricB) {
          minmA = rip_mti_minm(ifindexA, combi);
5
          if (minmA != RIP_MTI_INFINITY)
            yminm = minmA;

          /** y combination found if condition is true */
10        if (yminm <= metricA - metricB)
            return 1;

          minmB = rip_mti_minm(ifindexB, combi);
        }
15
        if (metricA < metricB) {
          minmB = rip_mti_minm(ifindexB, combi);

          if (minmB != RIP_MTI_INFINITY)
20          yminm = minmB;

          /** y combination found if condition is true */
          if (yminm <= metricB - metricA)
            return 1;
25
          minmA = rip_mti_minm(ifindexA, combi);
        }

        if (metricA == metricB) {
30        minmA = rip_mti_minm(ifindexA, combi);
          minmB = rip_mti_minm(ifindexB, combi);
        }

        /** check for x combination */
35      if (minmA + minmB <= combi)
          return 1;

        /** neither y- nor x-combination found */
        rme->metric = combi;
40      rme->timer = time(0);
      }
```

Zunächst werden die beiden Routen auf Y-Kombinationen untersucht. Da nach Abschnitt 3.4.2 auf Seite 15 die Route mit der höheren Metrik dabei ausschlaggebend ist, wird dies in Zeile 3 bzw. 16 überprüft. Im weiteren sei exemplarisch nun `metricA` > `metricB`, da beide Fälle analog behandelt werden.

In den Zeilen 4-7 wird ermittelt, ob in `minm` bereits eine Zyklusinformation zu diesem Interface gespeichert ist. Ist dies nicht der Fall und somit kein Eintrag vorhanden, so legt `rip_mti_minm()` diesen an und liefert `RIP_METRIC_INFINITY` zurück. `ymin` wird dann nicht überschrieben und trägt weiter den Initialisierungswert von 2, anderenfalls den korrespondieren Metrikwert aus `minm` .

Ist die aus Abschnitt 3.4.2 bekannte Ungleichung nicht erfüllt bzw. hier deren Umkehrung (Zeile 10) nicht erfüllt, liegt eine Y-Kombination vor. Zunächst mag dabei nun irritierend sein, daß die Route dennoch akzeptiert wird, da ein wahrer Wert zurückgeben wird (Zeile 11). Dies ist seitens RIP-MTI korrekt, da die alte Route noch ihre Gültigkeit besitzt und noch seitens `ripd` zu entscheiden ist, wie mit der neuen Route verfahren wird. Ausschlaggebend für RIP-MTI ist aber, daß keinerlei Informationen über diese beiden Routen in `mincyc` abgelegt werden. `mincyc` repräsentiert aber gerade den kleinsten, gültigen Zyklus zwischen zwei Interfaces. Somit arbeitet `rip_mti_cycleok()` korrekt, da die erkannte Y-Kombination kein gültiger Zyklus ist und Informationen darüber nicht gespeichert werden.

Sollte keine Y-Kombination vorliegen, so erfolgt in Zeile 35 mit der bekannten Ungleichung bzw. wieder deren Umkehrung der Test auf eine X-Kombination. Auch hier gilt wieder, daß im Falle einer X-Kombination keine Gültigkeitsinformation zu diesem Zyklus gespeichert wird.

Ist kein Source Loop festgestellt worden, wird der Wert in `mincyc` für die beiden Interfaces angepasst und der Timer zurückgesetzt (Zeilen 39,40).

Codesegment 5.8 Entscheidung über Alternativrouten in `rip_mti_cycleok()`

```
1     if ((rme = rip_mti_table_lookup(mincyc, pos)) != NULL) {

        if (combi > rme->metric)
          return 1;
5
        if (isold)
          return (combi == rme->metric);

        if (metricA > metricB) {
10      ...
```

Wird nun die bisherige Route zu einem Ziel ungültig (`isold == TRUE`), nutzt RIP-MTI das wie eben beschrieben gesammelte Wissen, um zu entscheiden, ob eine Alternativroute verwendbar ist.

Dieses Wissen verbirgt sich im `mincyc` -Eintrag `rme` (vgl. Codesegment 5.8 auf der vorherigen Seite). Die Kombination der Bedingungen in den Zeilen 3 und 7 zeigen, daß RIP-MTI die neue Route akzeptiert, sofern die Metrik der Kombination der alten und neuen Route größer oder gleich dem bisher kleinsten, gültigen Kreis zwischen den beiden betreffenden Interfaces ist. Anders ausgedrückt, die Metrik der Kombination zweier gültiger Route kann nicht kleiner sein als die in `mincyc` gespeicherte kleinste, gültige Kombination zu den betroffenen Interfaces.

In Zeile 3 ist desweiteren zu erkennen, daß die Bedingung `combi` > `rme->metric` unabhängig von `isold` ist. Diese Prüfung wird zuerst vorgenommen, da zum einen ein längerer Zyklus als der in `mincyc` abgelegte Wert immer von RIP-MTI als gültige Alternative angesehen wird, zum anderen muß dieser Fall nicht weiter betrachtet werden, da ihm keine neue Informationen über die Netzwerktopologie zu entnehmen sind.

rip_mti_timeout()

Wie die Routing-Tabellen-Einträge in `ripd` besitzen auch die Werte in `minm` und `mincyc` eine zeitlich begrenzte Gültigkeit, sofern sie nicht kontinuierlich bestätigt werden. Quagga verwendet hierfür eigene Threads für die Einträge, die deren Gültigkeit überwachen. Da auf die Elemente in `minm` und `mincyc` nur sehr selten und nur innerhalb von `rip_mti.c` zugegriffen wird, ist der Ansatz von Quagga hier nicht notwendig.

`rip_mti.c` verwendet die Funktion `rip_mti_timeout()` für diese Überprüfung. Die Funktion wird nur in `rip_mti_cycleok()` aufgerufen (vgl. Codesegment 5.6 auf Seite 32), da nur von dieser selbst oder von dort aufgerufenen Funktionen auf Einträge in `minm` und `mincyc` zugegriffen wird.

Sollten zukünftige Änderungen dies nicht mehr gewährleisten, so ist auf eventuell hinzuzufügende Aufrufe von `rip_mti_timeout()` zu achten.

Zunächst betrachtet `rip_mti_timeout()` (s. Codesegment 5.9) jeden Eintrag in `minm` . Jedes Element darin speichert den Zeitpunkt der letzten Änderung/Bestätigung im Strukturelement `timer`. Ist die Differenz zwischen dem aktuellen Zeitpunkt (`now`) und dem in `timer` abgelegten Wert größer oder gleich `RIP_MTI_ALIVE`, so ist der Wert ungültig und wird mit `RIP_MTI_INFINITY` überschrieben.

Gleiches Vorgehen wird auch bei `mincyc` angewandt. Das Problem hierbei ist aber,

Codesegment 5.9 Gültigkeitsprüfung in `rip_mti_timeout()`

```
1   void rip_mti_timeout(void) {

      unsigned int i, j, pos;
      u_int32_t minmA, minmB;
5
      time_t now = time(0);
      struct rip_mti_entry *entry;

      /* Timeout minm table */
10    for (i = 0; i < vector_max(minm); i++)
        if ((entry = vector_lookup(minm, i)) != NULL)
          if (now - entry->timer >= RIP_MTI_ALIVE) {
            entry->metric = RIP_MTI_INFINITY;
          }
15

      /* mincyc table */
      for (i = 0; i < vector_max(minm); i++) {
        if ((entry = vector_lookup(minm, i)) != NULL) {
20        minmA = entry->metric;
          for (j = 0; j < i; j++) {
            if ((entry = vector_lookup(minm, j)) != NULL) {
              minmB = entry->metric;
              pos = (((i + 1) * i) / 2) + j;
25            if ((entry = vector_lookup(mincyc, pos)) != NULL) {

                /* timeout */
                if (now - entry->timer >= RIP_MTI_ALIVE) {
                  entry->metric = RIP_MTI_INFINITY;
30              }

                /* x combinations
                   now - entry->timer < RIP_MTI_ALIVE */
                else if (minmA + minmB <= entry->metric) {
35                entry->metric = RIP_MTI_INFINITY;
                }
              }
            }
          }
40      }
      }
    }
```

daß für jedes Paar von Interfaces die Position des Eintrags in `mincyc` ermittelt werden muß. Zwar lassen sich die Interfaces über Quagga abfragen, dies ist aber mit einigen Änderungen an Quagga selbst verbunden, die vermieden werden können. Einfacher ist es, die Indices der Interfaces aus `minm` zu erhalten. Jeder Eintrag dort entspricht mit seiner Position genau einem Interface. Daher werden zwei verschachtelte Schleifen für jeden Eintrag in `minm` durchlaufen (vgl. Zeilen 18,21 in Codesegement 5.9), der zugehörige Eintrag in `mincyc` ermittelt und die Zeitstempel verglichen.

Zusätzlich stellt `rip_mti_timeout()` sicher, daß keine X-Kombinationen in `mincyc` abgelegt sind (Zeilen 34,35). Für jeden Eintrag in `mincyc` werden dafür die Metrikwerte aus `minm` der beiden beteiligten Interfaces ermittelt und in den Variablen `minmA` und `minmB` abgelegt. Analog zur Beschreibung in 3.4.2 auf Seite 15 liegt eine X-Kombination vor, wenn die Summe von `minmA` und `minmB` kleiner oder gleich dem korrespondieren Wert in `mincyc` ist. Auch dann wird der entsprechende Eintrag als ungültig deklariert.

Letztere Überprüfung fand in [Kle01] nicht statt.

5.2.3 Änderungen an Quagga und `ripd`

Nachdem nun dargelegt wurde, welche Neuerungen eingeführt wurden, ist noch offen, welche Änderungen an Quagga und `ripd` selbst vorzunehmen sind. In Codesegement 5.2 auf Seite 27 wurde bereits auf die Anpassungen in `ripd.h` eingegangen.

ripd.c

In `ripd.c` ist im Bezug auf RIP-MTI nur die Funktion `rip_rte_process` zu betrachten, die jeden Eintrag einer empfangenen RIP-Nachricht bearbeitet. Die Funktion addiert zur mitgeteilten Metrik bereits die mit dem Interface verbundene Metrik und überprüft u.a., ob bereits eine Route zu dem angegebenen Ziel existiert. Ist dies der Fall werden Information über die bisherige Route in der Variablen `rinfo` vom Typ `struct rip_info` abgelegt.

Codesegement 5.10 auf der nächsten Seite zeigt den angepassten Bereich aus `rip_rte_process()`. Als zweite Kondition stellt die Funktion in Zeile 12 fest, ob die neu mitgeteilte Metrik kleiner ist als die aktuell verwendete. Diese Prüfung wird um den Aufruf von `rip_mti_routeok()` erweitert und mit der ursprünglichen Bedingung über logisches UND verknüpft, wodurch RIP-MTI die Route ablehnen oder die Entscheidung von den Metriken abhängig machen kann.

Codesegment 5.10 Änderungen in `rip_rte_process()`

```
...
/* Next, compare the metrics.  If the datagram is from the same
   router as the existing route, and the new metric is different
   than the old one; or, if the new metric is lower than the old
   one, or if the tag has been changed; or if there is a route
   with a lower administrave distance; or an update of the
   distance on the actual route; do the following actions: */
if ((same && rinfo->metric != rte->metric)
  /* RIP MTI
     check mti tables */
    || (rip_mti_routeok(rinfo, rte, ifp)
        && (rte->metric < rinfo->metric))
    ...
  {
    /* - Adopt the route from the datagram.  That is, put the
       new metric in, and adjust the next hop address (if
       necessary). */
    oldmetric = rinfo->metric;
    rinfo->metric = rte->metric;
...
    /* - If the new metric is infinity, start the deletion
       process (described above); */
    if (rinfo->metric == RIP_METRIC_INFINITY)
      {
        /* If the new metric is infinity, the deletion process
           begins for the route, which is no longer used for
           routing packets.  Note that the deletion process is
           started only when the metric is first set to
           infinity.  If the metric was already infinity, then a
           new deletion process is not started. */
        if (oldmetric != RIP_METRIC_INFINITY)
          {

          /* RIP MTI
          store old metric since route is not available any more */
          rinfo->oldmetric = oldmetric;
...
```

Wird die neue Route akzeptiert, speichert `ripd` die Metrik der alten Route zunächst in der *lokalen* Variable `oldmetric` ab. Dies ist für RIP-MTI dann interessant, wenn die neue Route vom gleichen Nachbarn wie die alte Route aber mit unterschiedlicher Metrik stammt (vgl. Codesegment 5.10 Zeile 8). Dies schließt den Fall ein, daß die Route als unerreichbar gekennzeichnet wird, was in Zeile 26 überprüft wird. War das Ziel aber bisher erreichbar, so muß diese letzte Metrik ungleich `RIP_METRIC_INFINITY` im Element `oldmetric` des Routing-Tabellen-Eintrags abgelegt werden. Dies geschieht im Code der Zeilen 34 bis 39.

rip_main.c

Desweiteren ist die `main`-Funktion von `ripd` um den Aufruf von `rip_mti_init()` zu erweitern, um auch dort alle Datenstrukturen zu initialisieren.

Damit sind alle für die Funktionälitat notwendigen Änderungen beschrieben. Die nachfolgenden Anpassungen ergeben sich als Seiteneffekte aus der Implentierung.

Makefile.am

Damit auch die RIP-MTI-Dateien von `make` während des Compilierens beachtet werden, muß die Vorlage für `automake` angepasst werden. Hierbei sind nur die Quellen `rip_mti.c` und die Headerdatei `rip_mti.h` ebenfalls anzugeben (vgl. Codesegment 5.11).

Codesegment 5.11 Makefile.am

```
librip_a_SOURCES = \
        ripd.c rip_zebra.c rip_interface.c rip_debug.c rip_snmp.c \
        rip_routemap.c rip_peer.c rip_offset.c rip_mti.c

noinst_HEADERS = \
        ripd.h rip_debug.h rip_mti.h
```

libs/memory.h

In der von allen Quagga-Clients verwendeten Headerdatei `memory.h` werden alle benutzten *Speichertypen* aufgelistet, damit Quagga den alloziierten Speicher entsprechend markieren kann. Da in `rip_mti.c` auch die von Quagga bereitgestellten

Methoden zur Speicherverwaltung genutzt werden, ist hier nur ein *Tag* für den für die Struktur `rip_mti_entry` verwendeten Speicher hinzuzufügen. Dieses trägt den Namen `MTYPE_RIP_MTI_ENTRY`.

5.3 Installation

Quagga läßt sich sehr komfortabel über den unter Unix bekannten Weg mittels `configure` und `make` installieren. Standardmäßig werden die Programme unter `/usr/local` abgelegt. Zu beachten ist, daß der Benutzer und die Gruppe `quagga` manuell erstellt werden müssen, da Quagga unter diesen IDs ausgeführt wird.

Für die Verwendung im nachfolgenden Beispiel wurde Quagga mit den Optionen

- `--sysconfdir=/usr/local/etc/quagga` und
- `--localstatedir=/usr/local/var/quagga`

kompiliert.

Kapitel 6

Simulationen mit VNUML und Quagga

Nachdem die Komponenten bisher nur einzeln betrachtet wurden, ist es an der Zeit, diese in einen gemeinsamen Kontext zu rücken. Kapitel 4 zeigte den Aufbau von VNUML und die Möglichkeit, damit Netze zu simulieren. Im weiteren wurde dargestellt, wie der unter Linux einsetzbare Routing Daemon Quagga um die Funktionalität von RIP-MTI erweitert werden kann. Dieses Kapitel beschäftigt sich nun mit der Frage, wie die schon in Beispielen aufgezeigten Netze zum Einsatz von RIP-MTI mittels VNUML näher untersucht werden können.

6.1 Vorbereitungen

6.1.1 Dateisystem und Kernel

Die durch VNUML zur Verfügung gestellten virtuellen Maschinen benötigen ein Root-Dateisystem. Dies wird unter UML in einer einzigen Datei auf dem Hostsystem abgelegt und kann durch Programme wie `dbootstrap` oder `rootstrap` erstellt werden[1]. Wie bereits erwähnt wird diese Datei z.B. in `/usr/local/share/vnuml/filesystems` auf der Hostmaschine abgelegt.

Nach dem Erstellen eines neuen Dateisystems muß auch Quagga mit RIP-MTI in diesem installiert werden. Ein-/Ausgabe 6.1 auf der nächsten Seite zeigt den grundsätzlichen Ablauf dieser Prozedur, nachdem Quagga bereits auf dem Hostsy-

[1] Ein einsatzfähiges Root-Dateisystem findet sich auf der beiliegenden DVD im Verzeichnis `vnuml/filesystems`.

Ein-/Ausgabe 6.1 Installation von Quagga im Root-Dateisystem

```
becks:~# mount -o loop /usr/local/share/vnuml/filesystems/root /mnt
becks:~# cp -a /usr/local/src/quagga /mnt/usr/local/src
becks:~# chroot /mnt
becks:/# cd /usr/local/src/quagga
becks:/usr/local/src/quagga# make install
becks:/usr/local/src/quagga# exit
```

stem im Verzeichnis `/usr/local/src/quagga` kompiliert wurde.

Zunächst wird das verwendete Dateisystem mittels eines Loop-Device wie eine gewöhnliche Partition unter `/mnt` eingehängt und das Quagga-Verzeichnis dorthin kopiert. `chroot` ändert das Rootverzeichnis der Shell auf `/mnt`, so daß sich alle weiteren Operationen unterhalb von `/mnt` ereignen und die Verwendung als Root-Dateisystem nachahmt. Danach ist das bereits kompilierte Quagga nur noch mittels `make` zu installieren.

Für die Ausführung von Quagga ist es ebenfalls notwendig, daß der Benutzer und die Gruppe `quagga` existieren. Diese müssen ggf. mittels `useradd` im Root-Dateisystem angelegt werden.

Weiterhin sollte der von VNUML verwendete UML-Kernel[2] ebenfalls nach `/usr/local/share/vnuml/kernels` kopiert werden.

6.1.2 SSH-Schlüssel

Der Datenaustausch zwischen VNUML auf dem Hostsystem und den virtuellen Instanzen findet primär über das SSH-Protokoll statt. Damit es beim Ausführen des VNUML-Parsers nicht notwendig ist, bei jedem SSH-Verbindungsaufbau das Rootpasswort[3] einzugeben, sollte auf die Schlüsselauthentifizerung zurückgegriffen werden.

In Ein-/Ausgabe 6.2 auf der nächsten Seite ist zu sehen, wie ein privater und öffentlicher RSA-Schlüssel zu erstellen sind. Ein Passphrase ist nicht zu verwenden, da dieser anderenfalls bei der SSH-Authentifizierung abgefragt wird. Das Schlüsselpaar wird mit den Namen `id_rsa` und `id_rsa.pub` unter `/root/.ssh` abgelegt.

[2] Zu finden auf der beigefügten DVD unter `vnuml/kernels`.
[3] Das Rootpasswort des beigefügten Root-Dateisystems lautet `xxx`.

Ein-/Ausgabe 6.2 Erzeugung RSA-Schlüssel

```
becks:~# ssh-keygen -t rsa -b 1024
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
b8:87:b5:3e:c6:79:c8:60:b9:a8:4c:29:17:85:c5:7b root@becks
```

6.2 Konfiguration

6.2.1 VNUML

Nächster Schritt ist die Beschreibung des zu simulierenden Netzwerks mit XML. Kapitel 4 zeigte bereits eine solche Konfigurationsdatei, aus welcher VNUML alle Daten entnimmt, um das Netzwerk aufzubauen und die Simulation zu verwalten. Als Beispiel soll nun das bereits bekannte Netzwerk aus Abbildung 6.1 dienen.

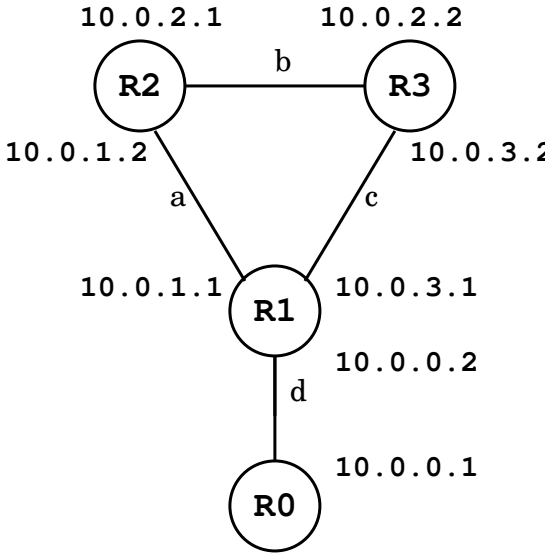


Abbildung 6.1: Beispielnetzwerk mit Zuordnung der IP-Adressen

Konfigurationssegment 6.1 zeigt den Teil der globalen Einstellugnen, die für alle virtuellen Maschinen gelten. Die verwendeten Tags werden im folgenden nur aus-

Konfiguration 6.1 Globale Konfiguration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/local/share/xml/vnuml/vnuml.dtd">
<vnuml>
   <global>
      <version>1.5</version>
      <simulation_name>rip_mti</simulation_name>
      <ssh_key version="2">/root/.ssh/id_rsa.pub</ssh_key>
      <automac offset="0"/>
      <ip_offset>192</ip_offset>
      <host_mapping/>
      <shell>/bin/sh</shell>
   </global>
```

zugsweise[4] erläutert:

`<version>`: Dieses Element bezieht sich auf die verwendete VNUML-Sprachversion und dient der Prüfung durch den VNUML-Parser.

`<simulation_name>`: Der Name jeder Simulation muß eindeutig gewählt werden, um diese korrekt zu identifzieren.

`<ssh_key>`: Diese Angabe zeigt absolut auf den verwendeten SSH-Schlüssel, der zuvor erzeugt wurde, um komfortabel auf die virtuellen Maschinen zuzugreifen. Der öffentliche Schlüsselteil wird während des Netzwerkaufbaus auf die virtuellen Maschinen kopiert.

Das Attribut `version` gibt die verwendete SSH-Version an. 2 sollte hier bevorzugt eingesetzt werden.

`<ip_offset>`: Zur Kommunikation zwischen Hostsystem und den virtuellen Maschinen erstellt VNUML nur dafür genutzte Interfaces. Das Tag `<ip_offset>` beschreibt zum einen durch das Attribut `prefix` das zu verwendende Subnetz, zum anderen mit dem Inhalt des Tags die Start-IP-Adresse.

Bsp.: `<ip_offset prefix=''192.100''>0<ip_offset>`

Gerade beim parallen Ausführen mehrerer Simulationen muß dies gesetzt werden.

[4] Die komplette Sprachreferenz findet sich auf der DVD unter `vnuml/html/doc/1.5/reference/`.

Konfiguration 6.2 Einstellung der verwendeten Netze

```
<!-- Networks -->
<net name="a"/>
<net name="b"/>
<net name="c"/>
<net name="d"/>
```

Der Konfigurationsauszug 6.2 zeigt die Definition der verwendeten virtuellen Netze. Mit Hilfe des `<net>`-Tags werden die Netze benannt, so daß eine Zuordnung von virtuellen Interfaces an virtuelle Netze stattfinden kann. Eine Konfiguration von Netzadressen usw. erfolgt an dieser Stelle noch nicht.

Ein weiteres erwähnenswertes Attribut von `<net>` ist `mode`. Ist dieses nicht oder auf `virtual_bridge` gesetzt, nutzt VNUML eine virtuelle Bridge, um die Netze aufzubauen. Dafür greift VNUML auf das Programm `brctl` zurück, das die Konfiguration direkt im Kernel des Hostsystems vornimmt. Daher muß die Simulation standardmäßig auch mit Root-Rechten ausgeführt werden.

Ist `mode=''uml_switch''` gesetzt, so kommt der `uml_switch`-Daemon zum Einsatz. Dieser sorgt über UNIX Sockets für eine Weiterleitung der Pakete zwischen den virtuellen Maschinen. `uml_switch` erlaubt die Ausführung als regulärer Benutzer.

Nachdem nun allgemeine und Netzwerkeinstellungen durchgeführt sind, bleibt noch die Konfiguration der virtuellen Maschinen. Dies ist in Konfigurationssegment 6.3 auf der nächsten Seite dargestellt.

`<vm>` beschreibt eine virtuelle Maschine. Im Attribut `name` wird der eindeutige Name festgelegt.

`<filesystem>`: Das Tag `<filesystem>` gibt das Root-Dateisystem der virtuellen Maschine an. In diesem Fall handelt es sich um die bereits in 6.1.1 beschriebene Imagedatei.

Beim Zugriff stehen verschiedene Optionen zur Auswahl. `cow` steht für *copy-on-write* und bedeutet, daß nur von der virtuellen Maschine vorgenomme Änderungen am Image in einer eigenen Datei gespeichert werden. Dies spart primär Plattenplatz.

`<kernel>`: Der zu bootende UML-Kernel wird hier angegeben.

`<if>`: Mittels `<if>` werden die Interfaces der virtuellen Maschine konfiguriert. Dabei erfolgt mittels `id` die Namensgebung und durch `net` die Zuordnung zu einem zuvor definierten virtuellen Netz.

Konfiguration 6.3 Virtuelle Maschine zu Beispiel 6.1

```
<vm name="R1">
   <filesystem type="cow">\
     /usr/local/share/vnuml/filesystems/root</filesystem>
   <kernel>/usr/local/share/vnuml/kernels/linux-2.6.10-1m\
   </kernel>
   <boot>
     <con0>pts</con0>
   </boot>
   <if id="1" net="d">
      <ipv4 mask="255.255.255.0">10.0.0.2</ipv4>
   </if>
   <if id="2" net="a">
      <ipv4 mask="255.255.255.0">10.0.1.1</ipv4>
   </if>
   <if id="3" net="c">
      <ipv4 mask="255.255.255.0">10.0.3.1</ipv4>
   </if>
   <filetree when="start" root="/usr/local/etc/quagga">\
     /usr/local/share/vnuml/rip-mti/conf/R1</filetree>
   <exec seq="start" type="verbatim">hostname</exec>
   <exec seq="start" type="verbatim">mount none /host \
     -t hostfs -o /data/R1</exec>
   <exec seq="start" type="verbatim">rm -f /host/ripd.log\
   </exec>
   <exec seq="start" type="verbatim">/usr/local/sbin/zebra -d\
   </exec>
   <exec seq="start" type="verbatim">/usr/local/sbin/ripd -d\
   </exec>
   <exec seq="stop" type="verbatim">hostname</exec>
   <exec seq="stop" type="verbatim">killall zebra</exec>
   <exec seq="stop" type="verbatim">killall ripd</exec>
   <exec seq="stop" type="verbatim">umount /host</exec>
</vm>
```

Innerhalb von `<if>` wird das Tag `<ipv4>` für die Konfiguration von IP-Adresse und Subnetzmaske eingesetzt. Analog kann auch das Tag `<ipv6>` verwendet werden.

`<filetree>`: Dieses Tag bewirkt, daß die Verzeichnisstruktur unterhalb des angebenen Verzeichnisses auf dem Hostsystem in das durch `root` angebene Verzeichnis innerhalb der virtuellen Maschine kopiert wird.

Im vorliegenden Fall ist dies nötig, um die Konfigurationsdateien des RIP- und Quagga-Daemons, die auf dem Hostsystem erstellt werden - hierauf wird im nächsten Abschnitt eingangen -, den virtuellen Maschinen zur Verfügung zu stellen.

`<exec>`: Durch `<exec>` können Befehlssequenzen definiert werden, die innerhalb der virtuellen Maschine vom Hostsystem aus angestossen werden können.

Im vorliegenden Beispiel werden zum Starten der Simulation folgende Befehle durchgeführt:

1. Das Verzeichnis `/data/R1` auf dem Hostsystem wird innerhalb der virtuellen Maschine im Verzeichnis `/host` eingehängt. Dies dient dem Zweck, Logdateien u.ä. direkt auf dem Hostsystem und nicht im Image des Rootdateisystems abzulegen.
2. Ein eventuell vorhandenes Logfile des RIP-Daemons wird gelöscht.
3. Der Zebra-Daemon wird gestartet.
4. `ripd` wird gestartet.

Analog werrden beim Stoppen der Simulation die beiden Daemons beendet und das Verzeichnis auf dem Hostsystem wieder freigegeben.

Diese Konfigurationspunkte sind für alle vier beteiligten Knoten durchzuführen.

6.2.2 Quagga und ripd

Neben VNUML müssen auch die in der Simulation eingesetzten Daemons konfiguriert werden. Wie in Konfiguration 6.3 ersichtlich werden die entsprechenden Dateien in diesem Beispiel im Verzeichnis

```
/usr/local/share/vnuml/rip-mti/conf/<Name der VM>
```

Konfiguration 6.4 Zebra-Daemon `zebra.conf`

```
! -*- zebra -*-
! RIP-MTI example
!
hostname R1
password zebra
enable password zebra
```

abgelegt. Dort sind für jede virtuelle Maschine die Dateien `zebra.conf` und `ripd.conf` zu erstellen.

In Auszug 6.4 ist die sehr übersichtliche Konfiguration des Zebra-Daemons angegeben. Es ist ausreichend, einen Hostnamen sowie ein Passwort für den Zugriff auf den Daemon zu defininieren.

Die etwas umfangreichere Konfigurationsdatei von `ripd` ist in Konfiguration 6.5 aufgeführt. Analog zu `zebra.conf` müssen erst Hostname und Passwort angegeben werden.

`router rip` aktiviert den RIP-Daemon. Das Element `network` beschreibt, für welche Netze und somit Interfaces RIP aktiviert werden soll. Durch `neighbor` werden wiederum die zugehörigen RIP-Nachbarn deklariert.

Das Element `redistribute` nimmt Einfluß auf die Routing-Informationen, die durch `ripd` propagiert werden. Neben den durch RIP selbst gelernten Routen werden von jeder Maschine auch die statischen Routen sowie die Routen zu direkt angeschlossenen Netzen weitergegeben. In anderen Anwendungsfällen sind hier z.B. Optionen wie `redistribute bgp` denkbar.

Abschließend wird dem Daemon mitgeteilt, in welche Logdatei die entsprechenden Einträge geschrieben werden sollen.

Das für RIP-MTI notwendige split-horizon ist standardmäßig in `ripd` aktiviert. Ein eventuelle Abschaltung ist über den Eintrag `no ip split-horizon` möglich.

Diese Konfigurationen sind natürlich wieder für alle eingesetzten virtuellen Maschinen zu wiederholen.

Konfiguration 6.5 RIP-Daemon `ripd.conf`

```
! -*- rip -*-
! RIP-MTI example
!
hostname R1
password zebra
!
debug rip events
debug rip packet
!
router rip
 network 10.0.0.0/24
 neighbor 10.0.0.1
 network 10.0.1.0/24
 neighbor 10.0.1.2
 network 10.0.3.0/24
 neighbor 10.0.3.2

redistribute static
redistribute connected

log file /host/ripd.log
```

6.3 Ausführung

Da nun alle Konfigurationsschritte abgeschlossen sind, kann mit dem Start der Simulation begonnen werden. Kern des weiteren Vorgehens ist das Programm `vnumlparser.pl`, welches die Schnittstelle zwischen der VNUML-Sprache und UML darstellt.

Die folgenden Kommandozeilenoptionen dienen der Verwaltung des simulierten Netzes. Ihr Einsatz wird später durch Beispiele weiter erläutert.

`-t foobar.xml`: Dies erstellt die in der Konfigurationsdatei `foobar.xml` beschriebenen Netze, startet die virtuellen Maschinen und konfiguriert diese inkl. ihrer Interfaces.

Außer den durch den Initvorgang gestarteten Prozessen befinden sich keine weiteren Anwendungen in der Ausführung.

`-x seq@foobar.xml`: Durch die Option `-x` ist es möglich, die in XML mit den Namen `seq` definierte Kommandosequenz auszuführen.

`-d foobar.xml`: Beendet die Simulation und somit alle virtuellen Maschinen.

Die Verwendung dieser Optionen in einer Simulation ist in Ein-/Ausgabe 6.3 beispielhaft dargestellt. Der Ablauf ist der folgende:

1. Aufbau der Netzwerktopologie und Starten aller virtueller Maschinen (Zeile 1).
2. Nun kann mittels SSH auf die Instanzen über ihren Namen zugegriffen werden. Beispielsweise wird hier die Routingtabelle des Knotens $R2$ vor Start der Simulation angezeigt.

   Nur die direkt angeschlossenen Netze sind eingetragen. Die Route zu Netz `192.168.3.6/30` gehört zum Managementinterface, welches von VNUML verwendet wird.
3. Im Anschluß kann auf dem Hostsystem die Simulation gestartet werden (Zeile 12). VNUML führt damit auf allen virtuellen Instanzen die mit `<exec seq=''start''>` definierten Kommandos aus. Somit werden auch der Zebra- und RIP-Daemon gestartet, die sofort die Arbeit aufnehmen.
4. Dies läßt sich direkt auf $R2$ an der Änderung der Routingtabelle ablesen. Auch alle anderen Netze des Beispiels sind nun enthalten. Ein Traceroute verdeutlicht den Weg von $R2$ zu $R0$.

Ein-/Ausgabe 6.3 Ablauf Ausführung der Simulation

```
becks:~# vnumlparser.pl -t ripmti.xml -v -B
...
becks:~# ssh R2
R2:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Met .. Iface
192.168.3.8     0.0.0.0         255.255.255.252 U     0   .. eth0
10.0.1.0        0.0.0.0         255.255.255.0   U     0   .. eth1
10.0.2.0        0.0.0.0         255.255.255.0   U     0   .. eth2
R2:~# logout
Connection to R2 closed.
becks:~# vnumlparser.pl -x start@ripmti.xml -v
...
becks:~# ssh R2
R2:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Met .. Iface
192.168.3.0     10.0.1.1        255.255.255.252 UG    3   .. eth1
10.0.0.0        10.0.1.1        255.255.255.0   UG    2   .. eth1
10.0.1.0        0.0.0.0         255.255.255.0   U     0   .. eth1
10.0.2.0        0.0.0.0         255.255.255.0   U     0   .. eth2
10.0.3.0        10.0.1.1        255.255.255.0   UG    2   .. eth1
R2:~# traceroute 10.0.0.1
traceroute to 10.0.0.1 (10.0.0.1), 30 hops max, 38 byte packets
 1  10.0.1.1 (10.0.1.1)  1.082 ms  0.928 ms  0.820 ms
 2  10.0.0.1 (10.0.0.1)  1.318 ms  1.254 ms  1.281 ms
R2:~# logout
Connection to R2 closed.
becks:~# vnumlparser.pl -x stop@ripmti.xml -v
...
becks:~# vnumlparser.pl -d ripmti.xml -v
```

5. In Zeile 29 werden alle mit `stop` benannten Kommandos angestossen, was dem Beenden der Routing-Daemons entspricht.

6. Schließlich wird die Simulation durch Herunterfahren der virtuellen Maschinen komplett beendet (Zeile 31).

Ein-/Ausgabe 6.4 Auszug aus `/data/R1/ripd.log` nach der Simulation

```
1   RIP: RECV packet from 10.0.3.2 port 520 on eth3
    RIP: RECV RESPONSE version 2 packet size 64
    RIP:   10.0.2.0/24 -> 0.0.0.0 family 2 tag 0 metric 1
    RIP: rip_mti_routeok: Checking new route for 10.0.2.0/24
5   RIP: rip_mti_routeok: Old route from interface eth2 (2) \
                                                    with metric 2
    RIP: rip_mti_routeok: New route from interface eth3 (3) \
                                                    with metric 2
    RIP: rip_mti_lookup_table: allocating memory for new entry \
10                                                  at index 5
    RIP: rip_mti_lookup_table: allocating successful
    RIP: rip_mti_lookup_table: allocating memory for new entry \
                                                    at index 3
    RIP: rip_mti_lookup_table: allocating successful
15  RIP: rip_mti_lookup_table: allocating memory for new entry \
                                                    at index 2
    RIP: rip_mti_lookup_table: allocating successful
```

In den Logdateien, die auf dem Hostsystem unter `/data` abgelegt wurden, können weitere Information über die Arbeit des RIP-Daemons eingesehen werden. Ausgabe 6.4 zeigt den Empfang eines RIP-Pakets von $R3$ (IP `10.0.3.2`) stammend auf dem Knoten $R1$ (vgl. Zeile 1).

Das Paket beinhaltet die Information über die Erreichbarkeit des Netzes `10.0.2.0/24`. $R3$ teilt mit, dieses mit einer Metrik von 1 zu erreichen (Zeile 3). Darauffolgend wird der Aufruf von `rip_mti_routeok()` aufgezeichnet. Die von RIP-MTI zu vergleichenden Routen werden ebenfalls angegeben.

Eine bereits zuvor erlernte Route für das Netz führt mit einer Metric von 2 über das Interface eth2 (Zeilen 5/6). Die neue Route hingegen zeigt mit gleicher Metrik auf das Interface eth3 (Zeilen 7/8). RIP-MTI stellt mit diesen Daten einen Zyklus zwischen eth2 und eth3 fest, da das Netz `10.0.2.0/24` über beide Interfaces erreicht werden kann. Die Zeilen 8-13 zeigen, wie entsprechend Einträge in `mincyc` an Position 5 und in `minm` an den Positionen 3 und 2 angelegt werden.

Ein-/Ausgabe 6.5 Interaktion mit `ripd`

```
becks:~# vnumlparser.pl -x start@ripmti.xml -v
...
becks:~# telnet R2 2602
Trying 192.168.3.10...
Connected to R2.
Escape character is '^]'.

Hello, this is quagga
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password:
R2>
R2> show ip rip
Codes: R - RIP, C - connected, S - Static, O - OSPF, B - BGP
Sub-codes:
  (n) - normal, (s) - static, (d) - default, (r) - redistribute,
  (i) - interface

     Network            Next Hop     Metric From       Tag Time
R(n) 10.0.0.0/24        10.0.1.1          2 10.0.1.1     0 02:39
C(i) 10.0.1.0/24        0.0.0.0           1 self         0
C(i) 10.0.2.0/24        0.0.0.0           1 self         0
R(n) 10.0.3.0/24        10.0.1.1          2 10.0.1.1     0 02:39
R2> exit
Connection closed by foreign host.
becks:~#
```

Nach dem Starten von `ripd` ist es ebenfalls möglich, direkt mit diesem Kontakt aufzunehmen. Ausgabe 6.5 auf der vorherigen Seite zeigt, wie dies durchzuführen ist. `ripd` bindet sich auf jeder virtuellen Maschine an den Port 2602. Dieser ist auch vom Hostsystem aus per `telnet` oder auch `netcat` erreichbar. Das verlangte Passwort entspricht jenem, das in der Konfigurationsdatei `ripd.conf` gesetzt wurde. Im vorliegenden Beispiel also „zebra“.

Innerhalb der `ripd`-Konsole stehen zahlreiche Befehle zur Verfügung. Die einfache Eingabe eines `?` listet diese während einer Sitzung auf. In Ausgabe 6.5 werden per `show ip rip` direkt vom RIP-Daemon die aktuellen Routeninformationen abgefragt. Dies liefert im Gegensatz zu `route` innerhalb der virtuellen Instanz (vgl. Ausgabe 6.3) auch einige der nur `ripd` bekannten Daten.

So zeigt Zeile 23, daß die Route für Netz `10.0.0.0/24` per RIP und von Nachbar `10.0.1.1` also $R1$ mitgeteilt wurde. Weiter ist in der letzten Spalte die ablaufende Gültigkeitszeit der Route dargestellt.

Damit sind einige Möglichkeiten von VNUML und auch Quagga dargestellt. Neben dem Einsatz des RIP-Daemons allein sind auch weitere Szenarien beispielsweise mit BGP oder der Kombination verschiedener Routingprotokolle möglich.

Kapitel 7

Ausblick

Mit VNUML und den MTI-Erweiterungen am RIP-Daemon von Quagga sind Werkzeuge vorgestellt, die es gemäß den Zielen dieser Arbeit erlauben, RIP-MTI weiter und auch praxisnäher zu untersuchen. Es ist nicht nur möglich, RIP-MTI alleine sondern auch seine Arbeit in Netzen mit unterschiedlichen Routingprotokollen zu beobachten und miteinander zu vergleichen.

Die Erkennung und möglicherweise auch Provokation von Counting-to-Infinity-Ereignissen muß noch tiefer beleuchtet werden, da diese zum Großteil vom zeitlichen Aufeinanderfolgen bestimmter Ereignisse abhängig sind. Der in [Kle01] vorgestellte Simulator konnte diese diskret betrachten, was in einer realitätsnahen Simulationsumgebung nun nicht mehr trivial ist. Es ist hier denkbar, entweder geringfügige Änderungen an der Nachrichtenverarbeitung in `ripd` vorzunehmen oder z.B. durch zusätzliche Anwendungen gefälschte RIP-Nachrichten zu bestimmten Zeiten in Umlauf zu bringen.

Aufgrund der verwendeten Architektur mit User-Mode-Linux handelt es sich weiterhin nur um eine Annäherung an die Realität. Weiterführende Arbeiten sollten dies bedenken und auch aufgreifen. So ist beispielsweise denkbar, die in dieser Architektur eingesetzten virtuellen Netze zu untersuchen. Latenzen können in diesen bisher nicht abgebildet werden, spielen aber gerade im Bezug auf die Eigenschaften von RIP eine große Rolle.

Weiterhin sollten die in [Kle01] gewonnenen Ergebnisse mit den hier vorgestellten Werkzeugen verifiziert werden. Wie bereits erwähnt werden in [Kle01] nicht alle Implementationsvorschläge aus [Sch99] übernommen. Das in [Kle01] vorgestellte Beispielnetz für die Unkorrektheit von RIP-MTI sollte hier als Grundlage für Simulationen mit VNUML dienen.

Aber auch Quagga bietet noch vielseitige Möglichkeiten. Durch den modularen Aufbau ist es möglich, neue Routingalgorithmen schnell hinzuzufügen. Dies kann zum einen eine Anwendung in der Lehre finden, um den Studierenden das Thema Routing aus Sicht eines Entwicklers näher zu bringen. Natürlich ist aber auch zum anderen der Einsatz im Bereich der Forschung und somit der Neuentwicklung zukünftiger Routingalgorithmen möglich. Durch die vollständige Unterstützung von IPv6 sind auch hier bereits alle Weichen in Quagga gestellt.

Literaturverzeichnis

[AJ03] Eitan Altmann and Tania Jimenez. NS simulator for beginners. http://www-sop.inria.fr/maestro/personnel/Eitan.Altman/COURS-NS/n3.pdf, 2003.

[Dik] Jeff Dike. The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/.

[Kle01] Thomas Kleemann. RIPeval - Evaluierung und Weiterentwicklung des RIP-MTI-Algorithmus. Diplomarbeit, Universität Koblenz-Landau, 2001.

[Mal98] G. Malkin. RFC 2453: RIP version 2, November 1998. See also STD0056. Obsoletes RFC1388, RFC1723. Status: STANDARD.

[net] Netkit - computer network emulation toolkit. http://netkit.org/.

[ns2] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[PD04] Larry L. Peterson and Bruce S. Davie. *Computernetze*. Dpunkt Verlag, 3rd edition, 2004.

[qua] Quagga Software Routing Suite. http://quagga.net/.

[Sch99] Andreas J. Schmid. RIP-MTI: Minimum-effort loop-free distance vector routing algorithm. Diplomarbeit, Universität Koblenz-Landau, 1999.

[vnu] VNUML Project home page. http://jungla.dit.upm.es/~vnuml/.

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Molsberg, 25. April 2005