Universität Koblenz-Landau
Abteilung Koblenz

Institut für Informatik
Arbeitsgruppe Steigner

September 2007

Diplomarbeit

Zentrale Betrachtung von Routing-Informationen zur Analyse des Konvergenzverhaltens verschiedener RIP-Algorithmen und Unterstützung des Generierens von Testfällen.

Erarbeitet von

Stefan Lange
stlange@uni-koblenz.de

Betreut von

Prof. Dr. Ch. Steigner
Dipl. Ing. Harald Dickel

Koblenz, den 13. September 2007

Erklärung

Ich, Stefan Lange (Student der Informatik an der Universität Koblenz-Landau, Matrikelnummer 202120827), versichere an Eides statt, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Stefan Lange

Inhaltsverzeichnis

1 Einleitung **2**

1.1 Motivation 2

1.2 Das Routing Information Protocol 2

1.3 Beispiel: Counting to Infinity 3

1.4 RIP with Minimal Topology Information 6

2 Anforderungen **8**

2.1 Änderungen der Routing Tabelle bestimmen 9

2.2 Datenhaltung und Datenanalyse 10

2.3 CTI-Erkennung 10

2.4 Datenrepräsentation 10

2.5 Speichern / Laden von Daten 10

3 Konzeption **11**

3.1 Quagga 11

3.2 RIP_XT 11

3.3 konzeptioneller Aufbau 12

4 Umsetzung **15**

4.1 Der RTE_Process 15

4.1.1 Analyse des RTE_Prozesses 15

4.1.2 Ermitteln der RT_Änderung 18

4.1.3 Der RTE_Process und MTI 22

4.2 SLServer und SL_Client 24

4.2.1 SLServer 24

4.2.2 SL_Client 25

4.2.3 Entwurf des Kommunikationsprotokolls 25

4.2.4 Implementation des Kommunikationsprotokolls 32

4.3 Die Datenorganisation 41

4.4 Der Networkanalyser 45

4.4.1 Pfadanalyse und CTI-Erkennung 46

4.5 Zeiten und Zeitbestimmung 54

4.5.1 Zeitbestimmung der Updateerzeugung 54
4.5.2 Der CTI-Timer 55
4.6 Die erweiterte XT_Client-GUI 62
4.6.1 MyNetworkTable 62
4.6.2 Der Verlaufsgraph 63
4.7 Toolbox 65
4.7.1 Speichern von Simulationsdaten (scenario data files) 65
4.7.2 Laden von Simulationsdaten 68
4.7.3 Speichern in CSV-Dateien 69

5 Der GraphObserver 71
5.1 Nachbarschaftsverhältnisse 71
5.2 Pfadsuche 72

6 Simulation und Anwendungsbeispiel 76
6.1 Voraussetzungen 76
6.2 Automatisches Laden einer Topologie 76
6.3 Aufbau der XT_Peer-GUI 77
6.4 Switched-Loop Szenario 79
6.5 Speichern und Laden von Szenariodaten 82
6.6 Switched-Loop Szenario mit Alternativ-Route 83
6.7 Beobachtung 87

7 Ausblick 89

Literaturverzeichnis 90

A Anhang 91
A.1 Quellcodes 91
A.2 Anleitung zur Patch-Erstellung 92
A.3 Anleitung zum Patchen 92
A.4 Mounten von Filesystemen 93

1 Einleitung

1.1 Motivation

Der an der Universität Koblenz-Landau entwickelte RIP-MTI-Algorithmus (RIP-MTI = RIP with Minimal Topology Information) stellt eine Modifikation des Routingalgorithmus RIP dar, die es dem RIP-Algorithmus ermöglichen soll, die Häufigkeit des Auftretens des Counting-to-Infinity (CTI) Problems zu reduzieren. Um die Korrektheit und Zuverlässigkeit dieses Algorithmus nachweisen, aber auch Schwächen aufdecken zu können, bedarf es der Möglichkeit, das Verhalten des Algorithmus zu testen.

Ziel dieser Arbeit ist die Nutzbarmachung der von, unter *Virtual Network User Mode Linux* (kurz VNUML) laufenden RIP-Routern dezentral verwalteten Routing-Informationen, um die Entstehung von CTIs zentral protokollieren und analysieren zu können. Zu diesem Zweck soll eine Software entwickelt werden, die Informationen zur Netzkonfiguration, zu Erreichbarkeiten und Update-Aufkommen sammelt, verwaltet und analysiert, um auf aus dem daraus resultierenden Modell Simulationen unter Verwendung verschiedener Ausprägungen des RIP-Algorithmus durchführen, Stärken und Schwächen dieser RIP-Varianten aufdecken und gegenüberstellen zu können. Dadurch kann dann ermöglicht werden, neben den bereits bekannten problematischen Netztopologien weitere für die einzelnen RIP-Ausprägungen problematische Topologien zu ermitteln.

1.2 Das Routing Information Protocol

Das Routing Information Protokoll RIP basiert auf dem Distanz Vektor Algorithmus. Dabei verwaltet jeder RIP-Router eine Tabelle, auf deren Grundlage er die kürzeste Strecke zu einem Zielnetz ermitteln kann. Die Aktualisierung dieser Tabelle erfolgt dabei durch Informationen, die zwischen direkt verbundenen Nachbarn ausgetauscht werden. Jeder RIP-Router, sofern er nicht direkt mit einem Zielnetzwerk verbunden ist, bekommt somit alle Erreichbarkeitsinformationen nur aus zweiter Hand. Diese Informationen werden standardmäßig alle 30 Sekunden [LLP03] in Form von periodischen Updates oder mittels eines, durch ein eingehendes Update hervorgerufenen, getriggerten Updates zwischen direkt benachbarten RIP-Routern ausgetauscht. RIP hat aus diesen Gründen die Eigenschaft, schnell auf „gute“ Nachrichten, also Nachrichten, die konsistente Informationen über die Erreichbarkeit eines Netzwerkes beinhalten, reagieren zu können, aber sehr langsam auf „schlechte“ Nachrichten [LLP03]. „Schlechte “Nachrichten sind in diesem Zusammenhang Nachrichten, die Informationen enthalten, die nicht der aktuellen Erreichbarkeit eines Netzwerkes entsprechen. Bei ungünstigen Updatekonstellationen kann das sog. Counting-To-Infinity Problem, kurz CTI auftreten. Das Beispiel aus Abschnitt 1.3 soll die Entstehung eines CTIs exemplarisch veranschaulichen. Um Herr über diese Art von Problematik zu werden, ist es notwendig, das Zählen bis ins Unendliche auf eine maximale Anzahl zu reduzieren. Es muss also ein Wert für RIP_METRIC_INFINITY definiert werden. Dieser Wert wurde auf 16 festgelegt. Mit dieser Beschränkung ist RIP ausschließlich in Netzwerken einsetzbar, in denen kein Pfad über mehr als 15 Hops geht.

Folgende Begriffe werden im Zusammenhang des RIP in dieser Arbeit verwendet:

- triggered Updates: Triggered-Updates stellen Updates dar, die durch Änderungen an der Routing-Tabelle eines Routers hervorgerufen werden. Dieser Router sendet allen direkten Nachbarroutern ein Update, das genau diese Änderung enthält. Um die Netzlast gerade in Netzen geringer Bandbreite gering zu halten wird nicht direkt nach jeder Änderung ein Triggered-Update gesendet. In [Mal98] ist deshalb folgendes für Triggered Updates festgelegt. Ein Triggered-Update wird nicht sofort gesendet, sondern obliegt einem Timer, der einen zufälligen Wert zwischen 1 und 5 Sekunden besitzt. Sollte sich die Erreichbarkeitsinformation für dieses Netz innerhalb dieses Timers nochmals ändern, wird der Timer gestoppt, das alte Triggered-Update verworfen und ein neuer Timer für das neue Triggered-Update gestartet. So ist es möglich, die Netzlast in einem Netz bei häufigen Änderungen von Erreichbarkeiten relativ gering zu halten.
- periodic Updates: Updates, die periodisch an RIP-Nachbarn gesendet werden. Dabei enthalten diese Updates alle Informationen, die der sendende Router über die Erreichbarkeit von Netzen kennt.
- Update-Time: Timer für periodische Updates
- Timeout-Timer: Nach Ablauf dieses Timer wird der betreffende Eintrag in der Routing-Tabelle als unerreichbar ausgezeichnet.
- Garbadge-Collection-Timer: Wurde ein Eintrag der Routing-Tabelle aufgrund des Ablaufens des Timeout-Timers als nicht mehr erreichbar ausgezeichnet, verbleibt dieser Eintrag für die Zeit des Garbadge-Collection-Timers noch in der Routing-Tabelle.

1.3 Beispiel: Counting to Infinity

Ein Beispiel soll an dieser Stelle noch einmal die Problematik beim Auftreten eines CTIs und den Aufwand bei der Beobachtung eines solchen CTIs verdeutlichen. In Abbildung 1 ist die sog. Y-Konfiguration dargestellt. Als Ausgangssituation Abbildung 1 (a) sei das Netz konvergent. D.h. allen Routern ist bekannt, wie jedes Netz erreichbar ist. In Tabelle 1 sind die Einträge zu allen Routern in der Ausgangssituation dargestellt.
Zum nächsten Zeitpunkt bleiben Updates von R4 über die Erreichbarkeit von Net5 aus. Das führt dazu, dass der *timeout-timer* [Qua] von R3 für diesen Pfad abläuft, ohne dass ein Update von R4 eintrifft. Das führt dazu, dass die Route zu Net5 in R3 als nicht erreichbar markiert wird und den Metrikwert RIP_METRIC_INFINITY (16) erhält. Sogleich setzt R3 seine Nachbarn R1 und R2 über diese Neuerung in Kenntnis, indem er entsprechende Updates an diese Router sendet. R1, der bislang R2 noch nicht über die Erreichbarkeit des Netzes Net5 über sich in Kenntnis gesetzt hat, sendet ein Update mit den Erreichbarkeitsinformationen an R2. Diese Updatefolge führt unweigerlich dazu, dass R2, dem nun 2 Updates angeboten werden, sich für das Update entscheidet, über das er das Netz erreichen kann und trägt die Information, die er von R1 bekommen hat in seine Routing-Tabelle ein.

Sogleich benachrichtigt R2 den Router R3 darüber, dass er über das Wissen verfügt, wie das Netz Net5 über ihn erreicht werden kann. R3 ersetzt den bislang mit

RIP_METRIC_INFINITY markierten Eintrag für Net5 durch die Informationen, die er von R2 erhalten hat. Zu diesem Zeitpunkt besitzt R1 aber bereits die Information, dass Net5 über R3 nicht mehr erreichbar sein kann und benachrichtigt R2 darüber. Dieser benachrichtigt wiederum R3, der seinerseits vorher R1 schon mit der falschen Information, dass Net5 über ihn erreichbar ist, versorgt hat. Zu diesem Zeitpunkt liegt ein CTI vor, da die Router R1, R2 und R3 stets ein falsches Update über die Erreichbarkeit von Net5 erhalten und diese Information an ihren jeweiligen Nachbarn weiterreichen. Selbst wenn das Update, das die Router über die nicht mehr verfügbare Erreichbarkeit aufklärt, auf der Strecke bleibt, zählen sich die Metriken für das Net5 kontinuirlich hoch, da die Information mit der höheren Metrik stets von dem Router kommt, von dem auch das vorhergehende Update kam und wird somit durch die neue Information ersetzt. Diese Inkonsistenz bleibt so lange erhalten, bis jeder Router für sich für diesen Pfad den Wert RIP_METRIC_INFINITY, also den Wert 16 eingetragen hat.

Dest.	**Metric**	**next Hop**
Net1	1	self
Net2	2	R2
Net3	1	self
Net4	2	R3
Net5	3	R3

(a) RT von R1

Dest.	**Metric**	**next Hop**
Net1	1	self
Net2	1	self
Net3	2	R1
Net4	2	R3
Net5	3	R3

(b) RT von R2

Dest.	**Metric**	**next Hop**
Net1	2	R1
Net2	1	self
Net3	1	self
Net4	1	self
Net5	2	R4

(c) RT von R3

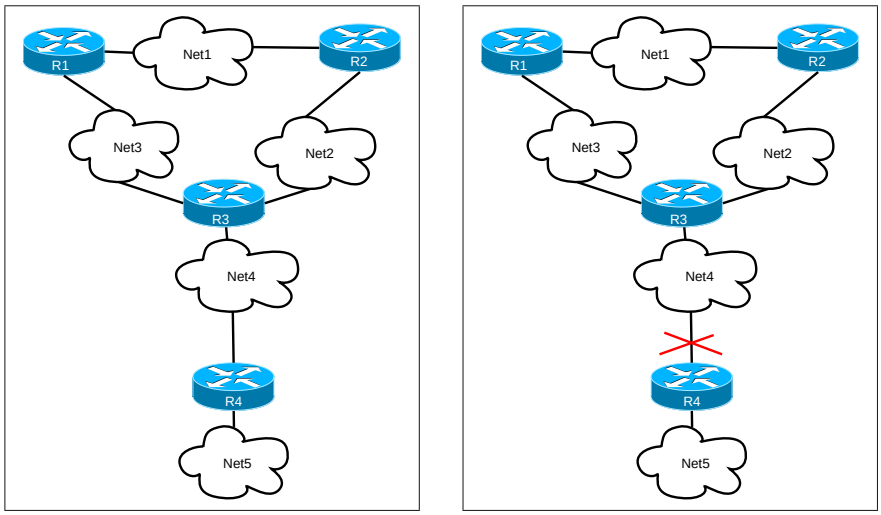
Dest.	**Metric**	**next Hop**
Net1	3	R3
Net2	2	R3
Net3	2	R3
Net4	1	self
Net5	1	self

(d) RT von R4

Tabelle 1: Routing Tabellen der Router R1 - R4 zur Ausgangssituation

Tabelle 2 zeigt die Routing-Tabelle von Router R3 zu den Zeitpunkten t bis t+5, wobei durch die Zeitpunkte die Zeitpunkte bezeichnet werden, zu denen die Änderung in der Routing-Tabelle für die Erreichbarkeit von Net5 vorgenommen wurden. Ausgehend von Tabelle 2 (a), in der zu erkennen ist, dass R3 zu diesem Zeitpunkt über das Wissen verfügt, dass Net5 nicht mehr erreichbar ist. Zum Zeitpunkt t+1 erhält er die Nachricht von R2, dass dieser einen Weg zu Net5 kennt. Zum Zeitpunkt t+2 erhält er wiederum von R2 ein Update für einen Pfad zu Net5 und übernimmt die neue Metrik. Bei t+3 hat sich die Metrik bereits auf 11 und zu t+4 auf 14 hochgezählt. Erst zum Zeitpunkt t+5 erreicht R1 eine Nachricht, dass Net5 nicht mehr über R2 erreichbar ist, da R2 nun bei einer Metrik von 16 angelangt ist und dies auch R3 wissen lässt.
Unter Einsatz des RIP-MTI wird dieses Phänomen unterbunden. Durch zusätzliche Überprüfung, ob ein Update angenommen werden darf oder ob dieses Update Resultat eines bestehenden Zyklus ist, kann R3, der in diesem Beispiel den Source-Router[1] darstellt, die

[1]Ein Source-Router beschreibt den Router innerhalb eines Zyklus, der dem ausgefallenen Netz am nächsten



(a) Ausgangssituation Konvergenz (b) Updates von R4 bleiben aus

Abbildung 1: Beispiel Y-Konfiguration

Dest.	Metric	next Hop
Net1	1	self
Net2	2	R2
Net3	1	self
Net4	2	R3
Net5	***16***	-

(a) RT von R3 zum Zeitpunk t

Dest.	Metric	next Hop
Net1	1	self
Net2	1	self
Net3	2	R1
Net4	2	R3
Net5	***5***	R2

(b) RT von R3 zum Zeitpunk t+1

Dest.	Metric	next Hop
Net1	2	R1
Net2	1	self
Net3	1	self
Net4	1	self
Net5	***8***	R2

(c) RT von R3 zum Zeitpunk t+2

Dest.	Metric	next Hop
Net1	3	R3
Net2	2	R3
Net3	2	R3
Net4	1	self
Net5	***11***	R2

(d) RT von R3 zum Zeitpunk t+3

Dest.	Metric	next Hop
Net1	2	R1
Net2	1	self
Net3	1	self
Net4	1	self
Net5	***14***	R2

(e) RT von R3 zum Zeitpunk t+4

Dest.	Metric	next Hop
Net1	3	R3
Net2	2	R3
Net3	2	R3
Net4	1	self
Net5	***16***	-

(f) RT von R3 zum Zeitpunk t+5

Tabelle 2: Routing Tabellen von Router R3 während eines CTI

Entstehung eines CTI unterbinden, indem er Updates von R2 ignoriert.

1.4 RIP with Minimal Topology Information

Zur Vermeidung des Counting-To-Infinity Verhaltens wurde in der Diplomarbeit von A. Schmid [Sch99] eine Erweiterung des RIP-Algorthmus vorgeschlagen, die das eigentliche Protokoll nicht verändert und somit die Kompatibilität mit dem herkömmlichen RIP gewährleistet. Dies ermöglicht einen Einsatz des MTI in einem Netzwerk mit herkömmlichen RIP-Routern. Dabei sammelt ein RIP-MTI-Router neben den Erreichbarkeitsinformationen eines Netzwerkes zusätzlich Informationen über die Netztopologie und erstellt eine Art Historie über Metriken, die sich für die Erreichbarkeiten in dieser Netztopologie ergeben. Wird dem Algorithmus nach Verlust eines Pfades ein neuer Weg zum Zielnetzwerk angeboten, kann der RIP-MTI-Router testen, ob dieses Update aus einem Zyklus resultiert und kann somit entscheiden, ob diese Information in die Routing-Tabelle übernommen werden darf oder nicht. Somit ist ein Source-Router in der Lage, einen Routing-Loop aufzubrechen und einen entstehenden CTI zu unterbinden.
Zu diesem Zweck arbeitet der MTI auf zwei Tabellen, um Informationen über die Netztopologie zu verwalten. Die Einträge in den Tabellen werden mit Hilfe der Interface-Indizes, über die Erreichbarkeits-Informationen im Router eintreffen, angesprochen, bzw. berechnet. Da im Kapitel 6 die Berechnung der Werte angesprochen wird, soll hier kurz auf den Aufbau und die Bedeutung dieser Tabellen eingegangen werden. Für eine detaillierte Beschreibung und die Herleitung verweise ich auf [Koc05] und [Sch99].

Folgende zwei Tabellen werden dabei verwaltet, die wie folgt in [Koc05] beschrieben werden:

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

$$mincyc_{i,j} = m_i^d + m_j^d – 1$$

Wird ein kürzerer Kreis festgestellt, wird der alte Wert überschrieben.“

minm : „$minm_i$ speichert den kleinsten Kreis zwischen Interface i und einem beliebigen anderen Interface von R.“

$$minm_i = min(mincyc_{i,j}) \qquad \forall j \in Interface(R), j \neq i.$$

m_i^d und m_j^d beschreiben dabei die Metriken mit denen das Zielnetz d über Interface i bzw. j erreicht werden kann.

ist.

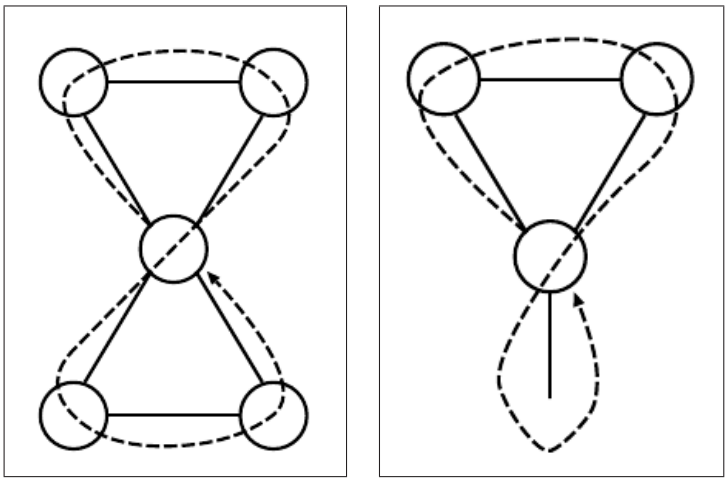
Wird einem Router R nach einem Ausfall eine Alternativ-Route angeboten, so testet der MTI, ob dieses Update aus einem Zyklus resultiert. Dazu wird auf zwei mögliche Zyklen, bzw. Kombinationen[2] (vergl. Abb. 1.4[3]) getestet[4]:

X-Kombination: Führt die alte Route nach d über das Interface i von Router R und die neue führt über Interface j, so muss folgende Ungleichung erfüllt sein, um eine X-Kombination ausschließen zu können.

$$minm_i \quad + \quad minm_j \quad > \quad m_{i,j}^{R,d,R}, \qquad mit \quad m_{i,j}^{R,d,R} = m_i^d + m_j^d - 1$$

Y-Kombination: Soll die Y-Kombination ausgeschlossen werden, so muss die folgende Ungleichung erfüllt sein.

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



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

Abbildung 2: mögliche Kombinationen

Mit Hilfe dieser Berechnung ist ein Source-Router in der Lage Source-Loops[5] zu erkennen und aufzubrechen. Somit ist es möglich die Entstehung von CTIs unter Einsatz des MTI auf Source-Routern zu unterbinden.

[2] Die Ungleichungen wurden aus [Koc05] übernommen

[3] Quelle der Abbildung [Koc05]

[4] „Unter der Bedingung, dass auf jedem beteiligten Router split horizon verwendet wird, konnte gezeigt werden, dass nur zwei mögliche Kombinationen betrachtet werden müssen.“[Koc05]

[5] Der Begriff Source-Loop beschreibt Pfade auf denen ein Update seinen Ursprung zweimal durchlaufen kann.

2 Anforderungen

In Abschnitt 1.3 wurde ein Szenario vorgestellt, welches die Entstehung eines CTIs erklärte. Aufgabe dieser Arbeit ist es, eine solche CTI-Entstehung beobachten zu können. Bisher war es dazu notwendig, die Routing-Tabellen auf den Routern selbst zu beobachten, bei denen davon ausgegangen wird, dass sie von einem CTI betroffen sind. In Abschnitt 1.3 konnte leicht vorhergesehen werden, welche Router involviert sind. In größeren Topologien, in denen man u.U. komplexere Schleifenkonstrukte vorfindet, ist dagegen nicht einfach zu ermitteln, welche Router von der Existenz eines CTI betroffen sein können. Das ist dem Umstand zu verdanken, dass ein Router, der nicht unmittelbar Bestandteil eines Zyklus ist, dennoch inkonsistente Informationen zur Erreichbarkeit eines Netzes bekommt, die aus einem vorherrschenden CTI in einem entfernten Zyklus hervorgehen. Abbildung 3 zeigt eine solche Konstellation.

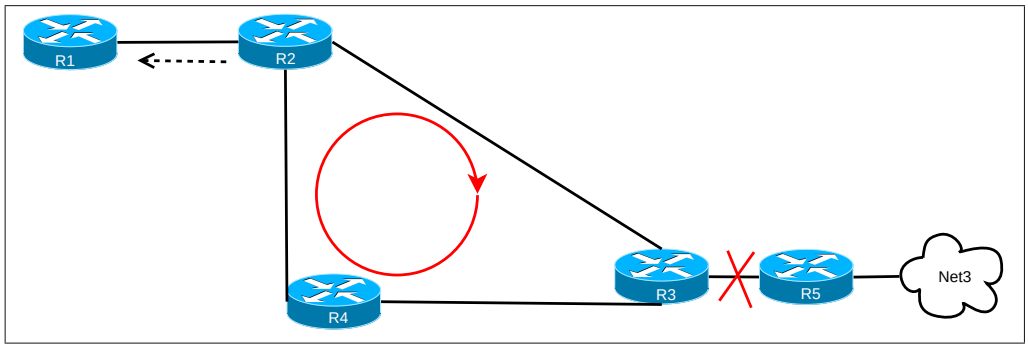


Abbildung 3: entfernter Zyklus

In Abbildung 3 wurden der Übersichtlichkeit halber keine Netze aufgeführt. Nur das Netz Net3 ist zu sehen. Die Erreichbarkeit dieses Netzes ist durch ausbleibende Updates des Routers R5 nicht mehr gegeben. Tritt in diesem Fall ein CTI auf, wird jedes Mal, wenn sich in R2 die Metrik für Net3 ändert auch R1 über die neue Metrik benachrichtigt. Dies führt dazu, dass auch R1 für die Dauer des CTI fehlerhafte Informationen für die Erreichbarkeit von Net3 führt. Dieses Beispiel könnte ebenfalls über mehrere Zyklen hinaus ausgebaut werden, woraus sich die Komplexität ergibt.
Dieses Beispiel zugrunde gelegt, kann man sich vorstellen, mit welchem Aufwand das Beobachten eines CTIs einhergehen kann, wenn stets für zahlreiche Router überprüft werden muss, ob ein CTI vorliegt oder nicht.
Zudem ist nicht sichergestellt, dass die Routing-Tabelle zum richtigen Zeitpunkt betrachtet wird, da nur grob abgeschätzt werden kann, wann ein Update zu einer Änderung der Routing-Tabelle führt.
Ein weiterer Nachteil, der sich ergibt, wenn die Beobachtung nur anhand der Routing-Tabellen der einzelnen Router erfolgt, liegt in der Natur der Sache. Betrachtet man die Einträge einer Routing-Tabelle, so sieht man immer nur die Pfadinformationen, die dem neuesten Stand entsprechen. Es gibt keine Möglichkeit, aufgrund fehlender Verlaufsspeicherung, die Netzentwicklung nachvollziehen zu können. Zu diesem Zweck wäre es notwendig, die Routingtabelle sukzessive, etwa durch ein Skript abzurufen und in eine Datei umzuleiten. Damit hätte man alle Daten gespeichert. Betrachtet man sich jedoch Abbildung 4, so ist zu erkennen, dass die

Auswertung dieser Daten mit einem großen Aufwand in der Nachbearbeitung einhergehen muss, da ein großer Overhead durch irrelevante Zusatzdaten entsteht.
Die folgenden Abschnitte beschäftigen sich damit, wie Änderungen an den Routing-Tabellen

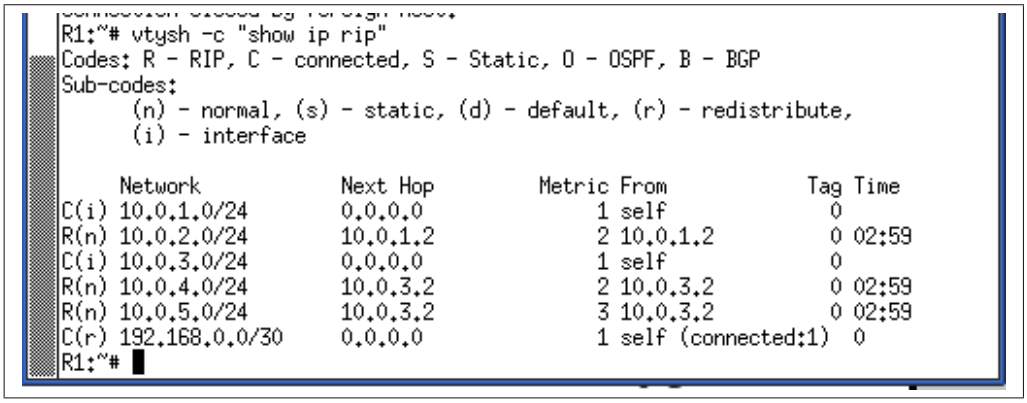


Abbildung 4: Ausgabe des Befehls „show ip rip“

von RIP-Routern effizient abgerufen und zentral verwaltet werden können, um Analysen zum Vorhandensein und zur Entstehung von CTIs durchführen zu können. Als Basis dient dabei VNUML und die VNUML-Routing-Suite Quagga. Dabei ergeben sich Anforderungen, die in den folgenden Abschnitten besprochen werden.
Folgende Schlüsselwörter und Prioritäten werden dabei verwendet:

- **muss**: Diese Funktionalität muss die resultierende Software zwingend erfüllen.
- **soll**: Diese Funktionalität muss die resultierende Software erfüllen.
- **sollte**: Diese Funktionalität soll ebenfalls durch die Software erfüllt werden, dabei spielt die Qualität des Ergebnises eher eine untergeordnete Rolle.
- **ist es wünschenswert**: Diese Funktionalität ist nicht erforderlich, aber hilfreich.

2.1 Änderungen der Routing Tabelle bestimmen

Essentiell für die Betrachtung von Routinginformationen ist, ob Updates, die bei einem Router eintreffen, angenommen werden oder nicht. Interessant sind dann nur die Updates, die angenommen werden und eine Änderung in der Routing-Tabelle hervorrufen. Es ergibt sich somit die Anforderung, dass immer dann, wenn an der Routing-Tabelle eines Routers Änderungen vorgenommen werden, diese Änderungen ebenfalls an einen zentralen Server geschickt werden **müssen**, von dem diese Daten für jeden einzelnen Router gesammelt und verwaltet werden.

2.2 Datenhaltung und Datenanalyse

Angesichts der Anzahl von Routern und den Informationen, die jeder Router für jedes von ihm gekannte Netz im Laufe seiner Betriebszeit verarbeitet, **muss** auch die Verwaltung dieser Daten auf der Seite des zetralen Servers passend organisiert werden. Die Datenspeicherung und Verwaltung **soll** somit auf Serverseite so realisiert werden, dass Analysen zur Bestimmung des Auftretens von CTIs direkt auf den vorliegenden Daten zeitnah vorgenommen werden können.

2.3 CTI-Erkennung

Zum Zwecke der Unterstützung der automatisierten CTI-Provokation **soll** ein Ansatz für eine ebenfalls automatisierte CTI-Erkennung entwickelt werden. Diese **soll** einen Anhaltspunkt für die automatisierte Generierung von Testfällen liefern, indem sie Zeitpunkte bestimmt, zu denen eine Abfolge von Updates das Auftreten von CTIs begünstigt.

2.4 Datenrepräsentation

Neben dem Sammeln von Daten **sollte** auch die Möglichkeit gegeben werden direkt das Routerwissen abzufragen. Dabei **sollte** es möglich sein, einen schnellen Überblick über die Situation zu bekommen, indem die Daten übersichtlich dargestellt werden.

2.5 Speichern / Laden von Daten

Um weiterführende Analysen oder Daten mehrerer Simulationen miteinander vergleichen zu können, **ist es wünschenswert**, dass die Simulationsdaten gespeichert werden können. Zum einen **sollten** dann die Daten verschiedener Simulationen direkt durch die Software gegenüberstellt, aber auch durch dritte Software, wie etwa Tabellenkalkulations-Programmen, eingelesen und verarbeitet werden können.

3 Konzeption

In diesem Abschnitt werden die konzeptionellen Überlegungen beschrieben, die sich mit der Frage beschäftigen, wie die Anforderungen auf Grundlage der gegebenen Software realisiert werden können. Die zugrundeliegende Software bildet die unter VNUML verwendete Quagga-Routing Suite und die XT-Software aus der Arbeit von Daniel Pähler [Pä06]. Dabei wird zunächst die XT-Software beschrieben. Darauf aufbauend werden Überlegungen getroffen, wie die Anforderungen auf dieser Grundlage konzipiert und realisiert werden können.

3.1 Quagga

Bei Quagga handelt es sich um eine Routing-Software-Suite, die verschiedene Routing-Protokolle implementiert. Verfügbare Routing-Protokolle sind u.a. RIP, OSPF und BGP-4. Quagga ist dabei modular aufgebaut, sodass die einzelnen Routing-Protokolle als eigenständige Daemons realisiert sind. Diese Daemons nutzen den zugrunde liegenden Zebra-Daemon, der eine Abstraktionsschicht zwischen Unix-Kernel, der die Routing-Tabellen verwaltet, und den Routing-Daemons, die dem Kernel über den Zebra-Daemon gelernte Routen mitteilen, darstellt. Durch den modularen Aufbau der Quagga-Routing-Suite können die einzelnen Routing-Daemons unabhängig von einander entwickelt werden. Bei der MTI-Implementierung entschied sich Tobias Koch in seiner Diplomarbeit [Koc05], den Quagga-RIP-Daemon um den MTI-Algorithmus zu erweitern, um die Neuentwicklung eines eigenständigen Quagga-Daemon und das damit einhergehende Fehlerpotential zu vermeiden. Die im folgenden Abschnitt beschriebene XT-Erweiterung aus [Pä06] beschränkt sich aus diesem Grund auch auf den Quagga-RIP-Daemon.

3.2 RIP_XT

Ausgangspunkt für diese Arbeit ist die aus der Diplomarbeit von Daniel Pähler [Pä06] hervorgegangene Software `RIP_XT`. Dabei handelt es sich um einen um ein Server-Modul erweiterten RIP-Daemon, der in der Lage ist von „außen“ Befehle entgegenzunehmen. Der Hintergrund dabei ist es, das Trigger-Verhalten von RIP-Routern so zu steuern, dass sie Updates in einer bestimmten Reihenfolge verteilen, wodurch CTIs provoziert werden können. Auf Hostseite befindet sich für diesen Zweck ein `XT_Client`, durch den die Updatetriggerung durch Steuerbefehle, die über eine TCP-Verbindung zwischen Host, d.h. `XT_Client` und VNUML-Maschine respektive `XT_Server`, vorgenommen wird. Eine schematische Darstellung ist in Abbildung 5 zu sehen.

Der `XT_Client` ist in Java implementiert, da Java nach Abwägung in [Pä06] alle Anforderungen erfüllt, einen Client zu realisieren, der unter anderem auch eine grafische Oberfläche zur übersichtlichen Benutzerinteraktion zur Verfügung stellt.
Auf diese Software baut ebenfalls die Diplomarbeit von Tim Keupen [Keu07] auf, so dass es sich anbietet, ein entsprechendes Programmpaket zu realisieren, das alle drei Arbeiten beinhaltet.
Im nächsten Kapitel wird beschrieben, wie die zugrundeliegende Software zum einen auf Quagga-Seite und zum anderen auf Seiten des `XT_Clients` verwendet und angepasst werden

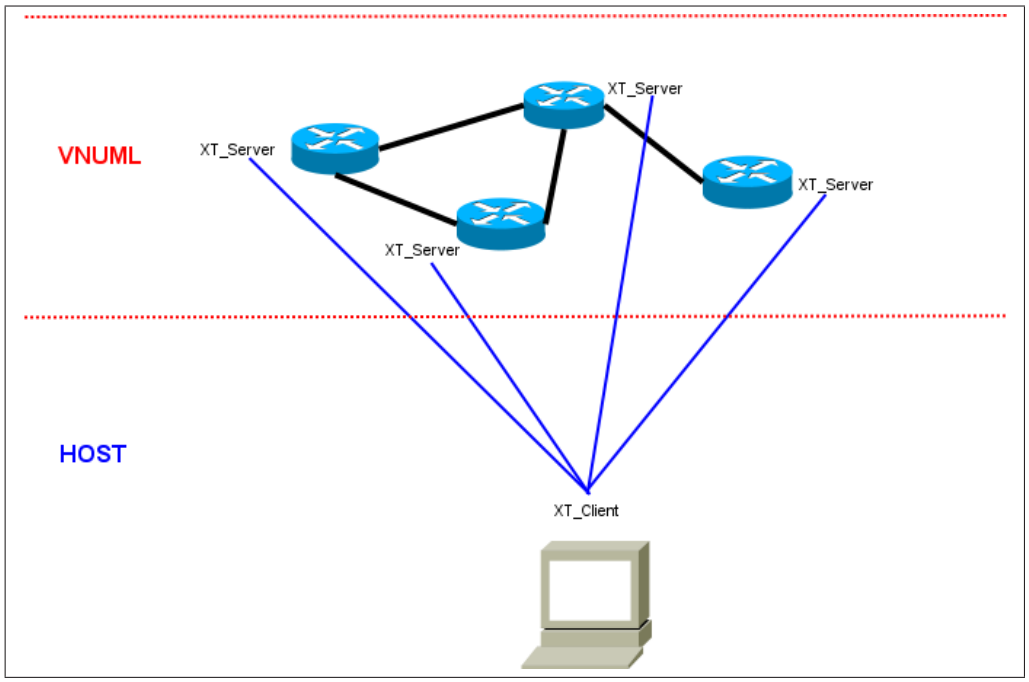


Abbildung 5: Schematische Darstellung einer XT-Session

kann.

3.3 konzeptioneller Aufbau

Auf Grundlage der beschriebenen Funktionalität der XT-Software bietet es sich an die Steuerfunktionalität dieser Software dahingehend auszubauen, dass weitere Verbindungen zwischen den VNUML-Maschinen und dem Host aufgebaut werden können. Diese zusätzliche Verbindung soll dabei ausschließlich als Datenkanal dienen. Die Abbildung 6 verbildlicht diese Überlegung.

Die RIP-Daemons auf jeder VNUML-Maschine werden also um einen Client ergänzt, der die Aufgabe besitzt, Routing-Informationen an einen zentralen Server auf Hostseite zu versenden. Der Client soll im folgenden als `SL_Client` und der Server `SLServer` bezeichnet werden. Das Verhältnis zwischen RIP-Daemon und `SL_Client` ist im Flussdiagramm in Abbildung 7 konzeptionell dargestellt.

Sobald ein RIP-Daemon ein Update über die Erreichbarkeit eines bestimmten Netzwerkes von einem seiner direkten Nachbarn erhält, prüft dieser, ob diese Erreichbarkeitsinformationen als Eintrag in die Routing-Tabelle übernommen werden können. Dieser in der RIP-Implementation der Quagga-Routing-Suite durch die Funktion `rip_rte_process` durchgeführte Entscheidungsprozess, der im folgenden `RTE_Process` genannt wird, löst dann eine zusätzliche Funktion aus, die veranlasst, dass diese Informationen im Falle einer Änderung der Routing-Tabelle ebenfalls an den `SLServer` verschickt werden. Ebenso

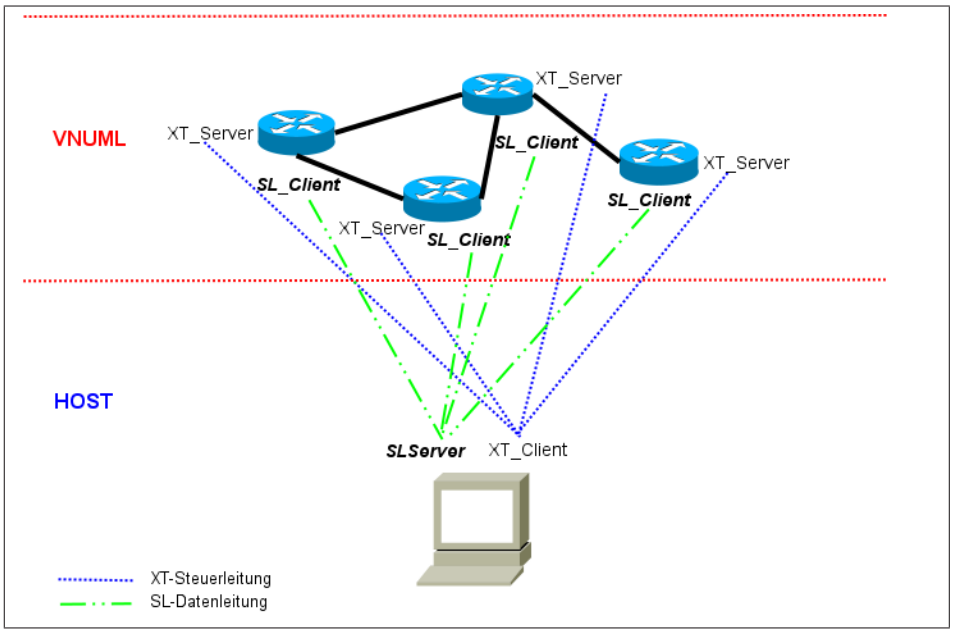


Abbildung 6: Schematische Darstellung einer XT-Session mit SL-Datenverbindung

müssen Änderungen von Einträgen durch das Ablaufen des Timeout-Timers einer Route berücksichtigt werden.

Auf Hostseite empfängt der `SLServer` diese Änderungs-Informationen und führt dann gemäß des Flussdiagramms aus Abbildung 8 die Organisation und Analyse dieser Daten durch. Die Bedingung, ob weitere Nachrichten eintreffen oder nicht, wird durch den `SL_Client` entschieden und dem `SLServer` mitgeteilt. Die detaillierte Erläuterung des Protokolls erfolgt in Abschnitt 4.2.

Im Sinne der Realisierung dieser Funktionalitäten sind folgende Softwareergänzungen auf VNUML- und XT-Client-Seite vorzunehmen:

- Implementation des `SL_Clients` als einen Endpunkt auf VNUML-Seite zur Realisierung der Datenverbindung
- Implementation des `SLServers` als Endpunkt auf Host-Seite zur Realisierung der Datenverbindung
- Anpassung relevanter Funktionen des Quagga RIP-Daemon um die Veranlassung des Datenexports
- Anpassung des `RIP_XT`-Protokolls um weitere Steuerbefehle zur Koordination des Aufbaus, Abbaus und Überwachung der Datenverbindung
- Implementation einer geeigneten Datenorganisation auf Host-Seite
- Implementation eines Analysewerkzeugs auf Host-Seite zur Analyse von Erreichbarkeitsinformation über Router-Grenzen hinaus.

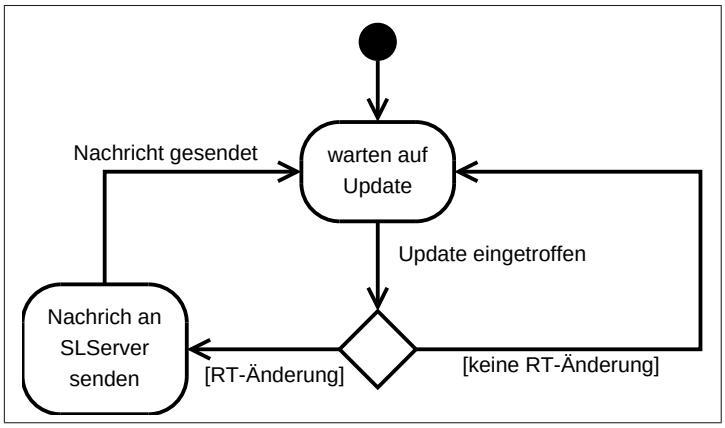


Abbildung 7: Flussdiagramm des erweiterten RIP RTE-Prozesses

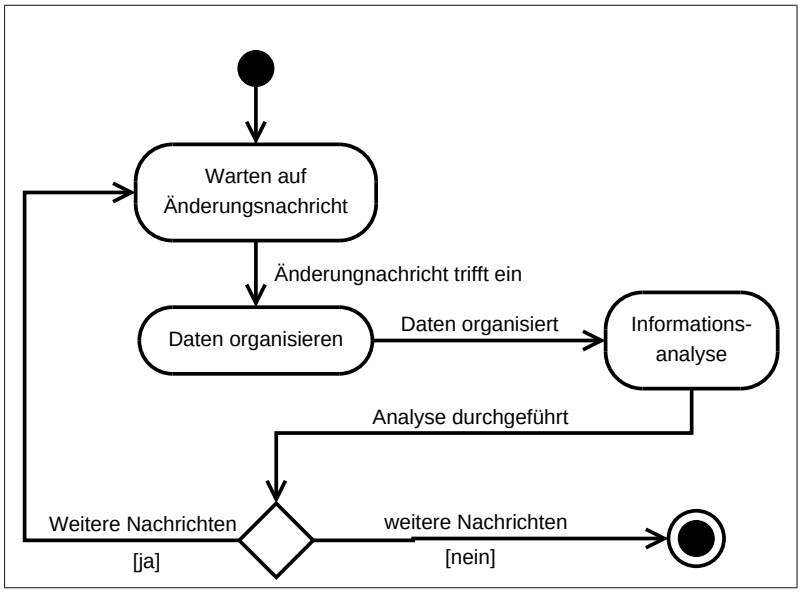


Abbildung 8: Flussdiagramm der Nachricht-Verarbeitung des SLServers

4 Umsetzung

In diesem Kapitel wird die Umsetzung der in Abschnitt 3.3 beschriebenen Funktionalitäten erläutert. Dazu wird zunächst untersucht, wie und an welchen Stellen im Quelltext der Implementation des Quagga RIP-Daemon die Entscheidung gefällt wird, ob Updates in der Routing Tabelle berücksichtigt werden oder unter welchen Umständen sich Einträge in der Routing-Tabelle ändern können und welche zusätzlichen Informationen für die gewünschte zentrale Betrachtung relevant sind. Zu diesem Zweck werden die jeweiligen Code-Stellen des Quagga-RIP betrachtet und die notwendigen Ergänzungen erläutert.
Um diese Änderungen der Routing-Tabelle eines RIP-Daemon zentral zu speichern und zu verarbeiten, wird des weiteren das Design und die Implementierung des für die Nachrichtenweiterleitung zuständigen `SL_Clients` und die des `SLServers` beschrieben.
Die Implementierung der Analysen, die anhand der zentral gesammelten Erreichbarkeitsinformationen durchgeführt werden, wird in Abschnitt 4.4 vorgestellt.

4.1 Der RTE_Process

Der `RTE_Process` ist der Bestandteil des RIP, der bei einem eingehenden Update die bestehende Routing-Tabelle des RIP-Routers betrachtet und die Entscheidung fällt, ob der entsprechende Eintrag der Tabelle geändert, ein neuer Eintrag hinzugefügt, der Timer eines bestehenden Eintrags zurückgesetzt oder der „deletion Process„ , der einen Eintrag nach Ablauf eines Timers aus der Routing-Tabelle entfernt, für einen Eintrag gestartet werden soll.
Der `RTE_Process` wird durch die Funktion `rip_rte_process` der ripd.c realisiert.
Da an dieser Stelle geprüft wird, ob sich die Routing-Tabelle ändert, bietet es sich an, auch hier die Schnittstelle zwischen `RTE_Process` und dem Versand der Updateinformation zum `SLServer` zu realisieren. Dabei wird jedes Mal, wenn diese Funktion aufgerufen und eine Änderung an der Routing-Tabelle vorgenommen wird, auch eine entsprechende Nachricht zum Versand an den `SLServer` generiert. Das Vorgehen wird in den folgenden Abschnitten beschrieben.

4.1.1 Analyse des RTE_Prozesses

Der `RTE_Process` ist der Bestandteil des RIP-Daemon, der eingehende Updates auf ihre Eignung testet und ggf. eine Änderung des entsprechenden Eintrags in der Routing-Tabelle vornimmt oder einen Eintrag, falls noch keiner für das Netzwerk, auf das sich das Update bezieht, vorhanden ist, hinzufügt. Der schematische Ablauf des `RTE_Process` ist in Abbildung 9 dargestellt.

Trifft ein Update beim Router ein, wird durch den `RTE_Process` geprüft, ob in der Routing-Tabelle bereits ein Eintrag über die Erreichbarkeit für dieses Netzwerk vorhanden ist. Ist dies nicht der Fall, wird ein neuer Eintrag in die Routing-Tabelle entsprechend der Informationen, die das Update enthält, hinzugefügt, sofern die Information nicht die Metrik RIP_METRIC_INFINITY bekannt gibt. Im anderen Fall existiert bereits ein Eintrag in der Routing-Tabelle. Das Update wird auf seine RIP-Integrität geprüft. RIP-Integrität

bedeutet, dass das Update alle Bedingungen, die im [Mal98] beschrieben sind, erfüllt, um in die Routing-Tabelle übernommen zu werden. Im Falle der Integritätserfüllung wird die Metrik, die learnedFrom- und die nextHop-Adresse des Updates in dem Eintrag der Routing-Tabelle angepasst. In diesen beiden Fällen, in denen das Update zum jeweiligen Eintrag bzw. Update der Routing-Tabelle führt, wird ein Triggered-Update eingeleitet, das alle anderen RIP-Nachbarn über diese neue Erreichbarkeitsinformation benachrichtigt und danach der Timeout-Timer für diesen Eintrag in der Routing-Tabelle gestartet wird.
Erfüllt das Update diese Integritätsbedingung nicht, wird es vom RIP verworfen und die alte Erreichbarkeitsinformation bleibt bestehen.

Der Quagga RIP-Daemon ist so implementiert, dass der Timeout Timer einer Route erst gesetzt wird, nachdem der Response-Process zum Senden der triggered Updates angestoßen wurde. D.h. erst zu diesem Zeitpunkt liegen alle Informationen vollständig in der Routing-Tabelle vor. Dies eröffnet zweierlei Möglichkeiten die Informationsübertragung an den zentralen Server zu etablieren.

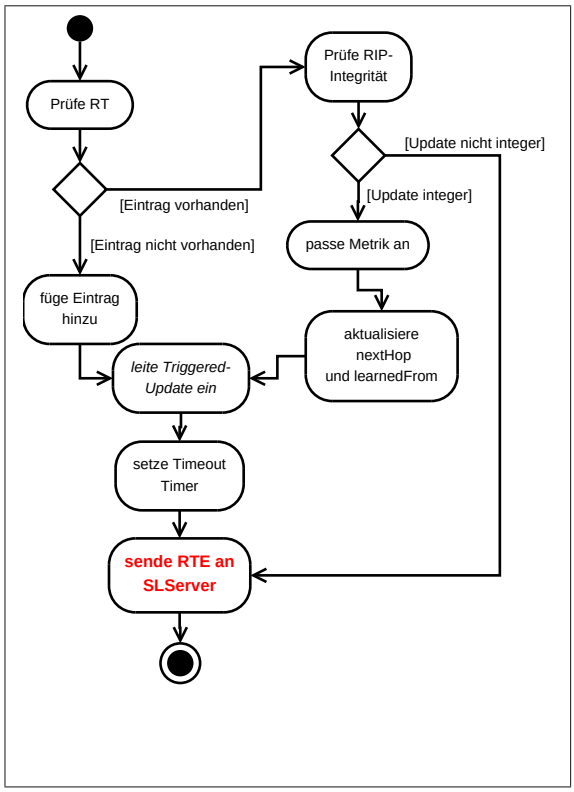


Abbildung 9: RTE Prozess des RIP-Daemon

Die erste Möglichkeit würde eine Veränderung der Anweisungsabfolge im `RTE_Process` bedeuten, indem der Response-Prozess erst angestoßen wird, nachdem der Timeout-Timer gestartet und die Informationsübertragung zum `SLServer` vorgenommen wurde. Dies würde ebenfalls einen Eingriff in die Implementation des Quagga-RIP bedeuten, der die Arbeitsweise des `RTE_Process` verändert. Da in dieser Arbeit ausschließlich eine Ergänzung

und nicht die Veränderung der Implementation vorgenommen werden soll, wird dieser Ansatz daher nicht weiter betrachtet.
Die zweite Möglichkeit besteht darin, die Informationsübertragung zum `SLServer` erst anzustoßen, nachdem alle RIP-Anweisungen im `RTE_Process` durchgeführt wurden, wie es auch in Abbildung 9 dargestellt ist.
In diesem Zusammenhang sei vorweggenommen, dass die Reihenfolge, in der die Informationen beim `SLServer` und den RIP-Nachbarn angelangen, eine elementare Rolle für die Analyse der Pfadinformationen spielt. Dabei muss die Information zuerst beim `SLServer` vorliegen, bevor sie bei den benachbarten RIP-Routern eintrifft. Zu diesem Zweck soll noch einmal die Arbeitsweise des Quagga in Bezug auf die Thread-Verwaltung aufgegriffen werden, die in [zeb] beschrieben und recht ausführlich in [Pä06] erläutert wurde.
Quagga-Threads werden im Innern des jeweiligen Quagga-Daemon verwaltet. Das führt dazu, dass ein Quagga-Daemon und seine Threads aus Sicht des Betriebssystems als ein einzelner, abgeschlossener Prozess erscheinen.
Threads die auf ihre Ausführung warten befinden sich in einer von vier Queues. Dabei wird eine dieser Queues, die Event-Queue abgearbeitet, bis kein Thread in dieser Queue mehr vorhanden ist. Threads, die in einer der Queues Read-Queue oder Write-Queue warten, werden in die leere Event-Queue verschoben, sobald die Datei oder der Socket, von dem sie lesen oder in den sie schreiben bereit ist und wiederum nacheinander abgearbeitet. Threads, die sich in der Timer-Queue befinden, warten dort bis der Timer abgelaufen ist und werden danach in die Event-Queue zu ihrer Ausführung verschoben.
Stößt der `RTE_Prozess` den Thread für ein Triggered-Update an, wird dieser direkt in die Event-Queue eingefügt, da der Thread weder auf einen Socket noch auf eine Datei noch auf einen Timer warten muss. In der Ausführung dieses Thread wird ein neuer Thread erzeugt, der in die Timer-Queue eingehängt wird. Der Timer ist dabei der, der in [Mal98] definiert wird. Der Timerwert dieses Timers liegt zwischen 1 und 5 Sekunden. Nach Ablauf dieses Timers werden per Multicast alle benachbarten Router über die Neuerung informiert. Ist Split Horizon aktiviert, wird der Router vom Response-Process ausgespart, von dem das Update gelernt wurde.

Um die zuvor erwähnte Ordnung

"`SLServer` ist informiert → Router Nachbarn sind informiert„

zu gewährleisten, ist es nun notwendig auszuschließen, dass diese Ordnung trotz der Abfolge

"Triggerung anstoßen → `SLServer` informieren„

nicht gestört wird.

Quagga-Threads unterliegen nicht dem präemptivem Scheduling, d.h. ein Quagga-Thread, der einmal mit seiner Ausführung begonnen hat wird nicht vom Dispatcher unterbrochen, stattdessen gibt der Thread die Betriebsmittel erst bei seiner Terminierung wieder ab. Dieses kooperative oder non-preemtive Scheduling hat zur Folge, dass der `RTE_Process` bis zu seiner Terminierung nicht unterbrochen wird, so dass das Senden der RT-Änderungsnachricht

an den `SLServer` in seiner Ausführung vor der des Response-Prozesses, der Triggerung des Updateversands an die Nachbarrouter, erfolgt.

Ein weiterer Grund für diese Designentscheidung ist die Tatsache, dass der `RTE_Process` durch die zwar sehr geringe aber dennoch vorhandene zeitliche Verzögerung für die Datenübertragung zwischen `SL_Client` und `SLServer` nicht in Mitleidenschaft gezogen wird. Da die Datenübertragung als letzte Anweisung im `RTE_Process` erfolgt, bleiben die Anweisungen des `RTE_Process` und somit Änderungen an der Routing-Tabelle aus zeitlicher Sicht unbeeinflusst.

Betrachtet man sich Abbildung 9 stellt man fest, dass in jedem Fall, auch bei Abweisung eines RIP-Updates, eine Nachricht an den zentralen Server gesendet wird. Diese Designentscheidung geht damit einher, dass neben den Erreichbarkeits-Informationen ebenfalls Informationen, die durch den Aufruf des MTI -falls vorhanden- erzeugt werden, an den zentralen Server geschickt werden sollen, damit die Routing-Informationen mit denen, aus der MTI-Analyse hervorgehenden Informationen in Zusammenhang gebracht werden können. Unter welchen Bedingungen die Funktion des MTI zum Einsatz kommt, wird in Abschnitt 4.1.3 geschildert. In welcher Form die MTI-Informationen und Änderungen in der Routing-Tabelle in Beziehung gebracht werden, wird in Abschnitt 4.3 beschrieben.

4.1.2 Ermitteln der RT_Änderung

Neben dem in Abschnitt 4.1.1 beschriebenen Fall, dass sich Einträge der Routing-Tabelle, aufgrund eingehender RIP-Updates ändern können und somit den Nachrichtenversand an den `SLServer` anstoßen, soll ebenfalls der Fall betrachtet werden, in dem sich ein Eintrag der Routing-Tabelle wegen des Ablaufens des Timeout-Timers verändert. Läuft der Timeout-Timer für eine Route ab, so wird die Metrik für diese Route auf RIP_METRIC_INFINITY gesetzt, was eine Unerreichbarkeit des Ziels über diese Route signalisiert. Auch in diesem Fall soll eine Nachricht über diese Änderung an den `SLServer` versendet (vergl. Listing 3) werden.

Es ergeben sich somit insgesamt folgende Auslöser für den Versand von Nachrichten an den SLServer:

1. es kommt aufgrund eines Updates zu einer Änderung der Routing-Tabelle
2. es kommt aufgrund eines Updates zu keiner Änderung der Routing-Tabelle
3. es kommt zu einem MTI-Check
4. der Timeout-Timer für eine Route läuft ab und die Metrik dieser Route wird auf RIP_METRIC_INFINITY gesetzt
5. es werden alle Routing-Informationen angefordert

Fall 1, 2 und 3 ergeben sich aus der in Abschnitt 4.1.1 erläuterten Designentscheidung, dass bei einem eingehenden RIP-Update eine Nachricht an den SLServer gesendet wird, auch wenn sich der Eintrag in der Routing-Tabelle nicht ändert, damit mögliche MTI-Informationen

mit den sich ergebenden Routing-Informationen in Beziehung gebracht werden können. In den Fällen 1 und 2 wird die Anweisungsfolge aus Listing 1 abgearbeitet. Diese stellt die Ergänzung und zugleich die letzte Anweisungsfolge der Funktion[6] `rip_rte_process` dar.

```
722 ...
723     sprintf(slc_state->send_buffer,"%s!-!",
724                         (char *) lookup(rip_slc_msg, RIP_SLC_MSG_UPDATE));
725     xt_sl_get_rte(rp);
726     xt_sl_notify_peer();
727 }
```

Listing 1: Anweisungsfolge im RTE_Process

Dabei wird der Typ[7] der Nachricht in den Sendepuffer geschrieben um danach die Informationen, auf die sich das RIP-Update bezog aus der Routing-Tabelle zu ermitteln und in den Sendepuffer zu schreiben. Das Auslesen der Routing-Informationen erfolgt durch die Funktion `xt_sl_get_rte` aus Listing 2. Diese bekommt den Pointer auf den betrachteten Eintrag übergeben, anhand dessen die relevanten Informationen aus der Struktur ausgelesen werden. Die Funktion arbeitet dabei analog zu dem Makro `DEFUN`, das im ripd definiert ist, um die bei Ausführen des "show ip rip „ -Befehls aktuelle Routing-Tabelle auf der Konsole auszugeben. Die Funktion *xt_sl_get_rte* hingegen speichert im Gegensatz zum rip-eigenen `DEFUN`-Makro, das alle Informationen aus der Routing-Tabelle ausliest und formatiert, nur die Information in den Sendepuffer (z.B. Zeile 4198-4201), die ihr mit dem Übergabeparameter *route_node* zur Verfügung stehen. D.h. genau die Informationen für einen Eintrag der Routing-Tabelle.

```
4184 int xt_sl_get_rte(struct route_node *np){
4185    struct rip_info *rinfo;
4186    int MAX = RIP_SLC_BUFSIZE;
4187    if ((rinfo = np->info) != NULL){
4188       int len;
4189       xt_sl_get_sim_time(); //prints the current timestamp into the sendbuffer
4190       len = sprintf(slc_state->tmp_buffer, "%c(%s)!-!%s/%d!-!",
4191             zebra_route_char(rinfo->type),
4192             rip_route_type_print(rinfo->sub_type),
4193             inet_ntoa (np->p.u.prefix4), np->p.prefixlen);
4194       size_t str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4195       strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4196       len = 24 - len;
4197       if (rinfo->nexthop.s_addr){
4198          sprintf (slc_state->tmp_buffer, "%s!-!%d!-!", inet_ntoa (rinfo->nexthop),
4199                rinfo->metric);
4200          str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4201          strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4202       }
4203       else{
4204          sprintf(slc_state->tmp_buffer, "0.0.0.0!-!%d!-!", rinfo->metric);
4205          str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4206          strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
```

[6] Wegen des beträchtlichen Umfangs dieser Funktion wird hier nur die Ergänzung aufgezeigt und für die Betrachtung der vollständigen Funktion auf den Quelltext des Quagga RIP verwiesen.

[7] Die Nachrichtentypen werden in Abschnitt 4.2.3 eingehend erläutert

```
4207        }
4208        // Route which exist in kernel routing table.
4209        if ((rinfo->type == ZEBRA_ROUTE_RIP) && (rinfo->sub_type == RIP_ROUTE_RTE)){
4210            sprintf (slc_state->tmp_buffer, "%s!-!", inet_ntoa (rinfo->from));
4211            str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4212            strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4213            sprintf(slc_state->tmp_buffer, "%d!-!" , rinfo->tag);
4214            str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4215            strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4216            xt_sl_get_rip_uptime(rinfo);
4217            str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4218            strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4219        }
4220        else
4221            if (rinfo->metric == RIP_METRIC_INFINITY){
4222                sprintf (slc_state->tmp_buffer, "self!-!");
4223                str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4224                strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4225                sprintf (slc_state->tmp_buffer, "%d!-!", rinfo->tag);
4226                str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4227                strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4228                xt_sl_get_rip_uptime(rinfo);
4229                str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4230                strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4231                }
4232            else{
4233                if (rinfo->external_metric){
4234                    len = sprintf (slc_state->tmp_buffer, "self_(%s:%d)!-!",
4235                    zebra_route_string(rinfo->type),
4236                    rinfo->external_metric);
4237                    str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4238                    strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4239                    len = 16 - len;
4240                }
4241                else{
4242                    sprintf (slc_state->tmp_buffer, "self!-!");
4243                    str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4244                    strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4245                }
4246                sprintf (slc_state->tmp_buffer, "%d!-!", rinfo->tag);
4247                str_len = MAX - strlen(slc_state->tmp_buffer)+1;
4248                strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
4249            }
4250            strncat(slc_state->send_buffer, "\n",2);
4251            slc_state->message_available = 1;
4252            return 0;
4253    }
4254    else {
4255    slc_state->message_available = -1;
4256    return -1;
4257    }
4258
4259 }
```

Listing 2: xt_sl_get_rte der ripd.c

Nach Aufruf dieser Funktion liegt im Sendepuffer folgender String vor:

Type!-!*updateTime*!-!*Code*!-!*Network*!-!*NextHop*!-!*Metric*!-!*learnedFrom*!-!*Tag*!-!*Time*

updateTime, *Code*, *Network*, *NextHop*, *Metric*, *learnedFrom*, *Tag* und *Time* stellen dabei Platzhalter für die jeweiligen Werte dar. Die Reihenfolge der Werte ist analog zu der, die

sich bei Ausgabe der Routing-Tabelle auf der Konsole (vergl. Abbildung 4) durch den Befehl "show ip rip„ ergibt. Nur die ***updateTime*** ist ein zusätzlicher Wert, der dem Zeitpunkt entspricht, zu dem diese Änderungsnachricht generiert wurde. Dieser Zeitstempel wird zur Analyse der Erreichbarkeitsinformationen herangezogen und im Abschnitt 4.5.1 dieser Arbeit eingehend erklärt. Um den Daten-Overhead und somit die Übertragungszeit zwischen `SL_Client` und `SLServer` gering zu halten, werden nur die Werte getrennt durch "!-!„ in den Sendepuffer geschrieben.

Analog dazu wird in Fall 4 nach Ablauf des Timeout-Timers eine entsprechende Nachricht generiert. Die um den Nachrichtenversand ergänzte Funktion `rip_timeout` ist in Listing 3 zu sehen. Die Ergänzungen erfolgten in den Zeilen 195 - 199.

```
165 /* Timeout RIP routes. */
166 static int
167 rip_timeout (struct thread *t)
168 {
169   struct rip_info *rinfo;
170   struct route_node *rn;
171
172   rinfo = THREAD_ARG (t);
173   rinfo->t_timeout = NULL;
174
175   rn = rinfo->rp;
176
177   /* - The garbage-collection timer is set for 120 seconds. */
178   RIP_TIMER_ON (rinfo->t_garbage_collect, rip_garbage_collect,
179                 rip->garbage_time);
180
181   rip_zebra_ipv4_delete ((struct prefix_ipv4 *)&rn->p, &rinfo->nexthop,
182                          rinfo->metric);
183   /* - The metric for the route is set to 16 (infinity).  This causes
184      the route to be removed from service. */
185   rinfo->metric = RIP_METRIC_INFINITY;
186   rinfo->flags &= ~RIP_RTF_FIB;
187
188   /* - The route change flag is to indicate that this entry has been
189      changed. */
190   rinfo->flags |= RIP_RTF_CHANGED;
191
192   /* - The output process is signalled to trigger a response. */
193   rip_event (RIP_TRIGGERED_UPDATE, 0);
194
195   /*notify the sl_server about this timeout*/
196   sprintf(slc_state->send_buffer,"%s!-!",
197                    (char *) lookup(rip_slc_msg, RIP_SLC_MSG_RTE_TIMEOUT));
198   xt_sl_get_rte(rn);
199   xt_sl_notify_peer();
200   return 0;
201 }
```

Listing 3: rip_timeout der ripd.c

Diese Funktion wird nach Ablauf des Timeout-Timers einer Route durch den dazugehörigen Thread ausgeführt. Dieser terminiert nun, aufgrund des in Abschnitt 4.1.1 erläuterten kooperativen Scheduling erst, wenn die Übertragung an den `SLServer` vollständig abgeschlossen wurde. Die Übertragung erfolgt durch die Funktion `xt_sl_notify_peer`, die in Abschnitt

4.2.4 erläutert wird.

Fall 5 dient dem Abgleich der Routing-Informationen zwischen RIP-Router und `SLServer` nach dem Verbindungsaufbau und wird in Abschnitt 4.2.4 beschrieben.

4.1.3 Der RTE_Process und MTI

Da sich die Designentscheidung, die in Abschnitt 4.1.1 beschrieben wurde auch danach richtet, wann der MTI-Mechanismus im `RTE_Process` aufgerufen wird, soll hier kurz darauf eingegangen werden.
Der `RTE_Process` wurde im Zusammenhang mit MTI insofern ergänzt, als dass neben der in [Mal98] beschriebenen Bedingungen weitere überprüft werden müssen (vergl. Abschnitt 1.4), bevor eine Route in die Routing-Tabelle übernommen wird. Zum Zwecke dieser Überprüfungen wird die Funktion `rip_mti_routeok` des MTI-Algorithmus, wie in [Koc05] erläutert aufgerufen. Dieser Funktionsaufruf erfolgt dabei in einer *if*-Abfrage in Form einer *Short-circuit evaluation* [8], wie sie in Listing 4 dargestellt ist.

```
598 ...
599 same = (IPV4_ADDR_SAME (&rinfo->from, &from->sin_addr)
600                && (rinfo->ifindex == ifp->ifindex));
601 ...
602 ...
603
604 /* Next, compare the metrics.  If the datagram is from the same
605 router as the existing route, and the new metric is different
606 than the old one; or, if the new metric is lower than the old
607 one, or if the tag has been changed; or if there is a route
608 with a lower administrave distance; or an update of the
609 distance on the actual route; do the following actions: */
610 if ((same && rinfo->metric != rte->metric)
611    /* RIP MTI
612    check mti tables */
613    || (rip_mti_routeok(rinfo, rte, ifp) && (rte->metric < rinfo->metric))
614    || ((same)
615    && (rinfo->metric == rte->metric)
616    && ntohs (rte->tag) != rinfo->tag)
617    || (rinfo->distance > rip_distance_apply (&rinfotmp))
618    || ((rinfo->distance != rip_distance_apply (rinfo)) && same))
619 {
620 ...
```

Listing 4: Aufruf der `rip_mti_routeok`

Ist der Ausdruck ($same \;\; \&\& \;\; rinfo \rightarrow metric \;\; != rte \rightarrow metric$) erfüllt, ist die Bedingung aufgrund der oder-Verknüpfung mit dem übrigen Ausdruck ebenfalls erfüllt und die Funktion `rip_mti_routeok` wird nicht aufgerufen. Im Umkehrschluss wird die Funktion also nur aufgerufen, wenn sich der Router, bzw. die Interface-Adresse des Routers von dem das

[8] "Bei der Short-Circuit-Evaluation eines logischen Ausdrucks wird ein weiter rechts stehender Teilausdruck nur dann ausgewertet, wenn er für das Ergebnis des Gesamtausdrucks noch von Bedeutung ist. Falls in dem Ausdruck A && B also bereits A falsch ist, wird zwangsläufig immer auch A && B falsch sein, unabhängig von dem Resultat von B. Bei der Short-Circuit-Evaluation wird in diesem Fall B gar nicht mehr ausgewertet. Analoges gilt bei der Anwendung des ODER-Operators.„.[Kru06]

Update stammt oder das Interface über das das Update empfangen wurde, von dem unterscheidet über das das vorherige Update empfangen wurde oder wenn die Metriken des neuen und des alten Updates gleich sind.
Es bietet sich an, neben den Änderungen, die die Routing-Tabelle erfährt auch die MTI-Entscheidungen, die zu einer Änderung führen, zu dokumentieren. Zu diesem Zweck wird bei Eintritt in die Funktion `rip_mti_routeok` dies ebenfalls dem `SLServer` mitgeteilt werden. Sobald die Funktion betreten wird, werden Debug-Ausgaben generiert, die in die ripd.log Datei ausgegeben werden. Im Sinne der zentralen Datensammlung, soll diese Ausgabe ebenfalls als Nachricht an den `SLServer` gesandt werden. Für diesen Zweck werden die Daten, die auch die Debug-Ausgabe enthält in den Sende-Puffer des `SL_Clients` geschrieben (vergl. Listing 5 Zeilen 192 - 211). Nach dem Aufruf der Funktion `rip_mti_routeok` steht folgendes im sende-Buffer:

Net-Address!-!*oldIncomingInface*!-!*oldMetric*!-!*newIncomingIface*!-!*newMetrik*

Die Bezeichner stellen Platzhalter für die jeweiligen Werte dar.

```
175 int rip_mti_routeok(struct rip_info *oldroute, struct rte *rte,
176                                                 struct interface *ifp) {
177
178   u_int32_t routemetric;
179   /* not neccessary since the cost of the interface is already
180      added to the metric in rip_rte_process
181      u_int32_t newmetric = rte->metric + 1; */
182 #ifdef RIP_MTI_DEBUG
183   zlog_debug("rip_mti_routeok: Checking new route for %s/%d",
184              inet_ntoa(rte->prefix), ip_masklen(rte->mask));
185   zlog_debug("rip_mti_routeok: Old route from interface %s (%d) with metric %d",
186              ifindex2ifname(oldroute->ifindex), oldroute->ifindex,
187              oldroute->metric);
188   zlog_debug("rip_mti_routeok: New route from interface %s (%d) with metric %d",
189              ifindex2ifname(ifp->ifindex), ifp->ifindex,
190              rte->metric);
191 #endif
192     /*put this DEBUG-Message through the
193      *sl_sendbuffer to send it to the SL_Server*/
194    sprintf(slc_state->send_buffer,"%s!-!" ,
195                                lookup(rip_slc_msg, RIP_SLC_MSG_MTI_DEBUG_INFO));
196    xt_sl_get_sim_time();
197    sprintf(slc_state->send_buffer,"%s%s/%d",slc_state->send_buffer,
198                                            inet_ntoa(rte->prefix),
199                                            ip_masklen(rte->mask));
200    sprintf(slc_state->tmp_buffer, "%s (%d)!-!%d",ifindex2ifname(oldroute->ifindex),
201                                                  oldroute->ifindex,
202                                                  oldroute->metric);
203    sprintf(slc_state->send_buffer,"%s!-!%s",slc_state->send_buffer,
204                                            slc_state->tmp_buffer);
205    sprintf(slc_state->tmp_buffer, "%s (%d)!-!%d",ifindex2ifname(ifp->ifindex),
206                                                  ifp->ifindex,
207                                                  rte->metric);
208    sprintf(slc_state->send_buffer,"%s!-!%s\n",slc_state->send_buffer,
209                                              slc_state->tmp_buffer);
210    xt_sl_notify_peer();
211 ...
```

Listing 5: Ergänzung der Funktion mti_route_ok

4.2 SLServer und SL_Client

Im folgenden werden der `SLServer` und der `SL_Client` vorgestellt. Die grundlegenden Aufgaben dieser beiden Kommunikationsendpunkte werden in den Abschnitten 4.2.1 und 4.2.2 beschrieben. Der Entwurf des Kommunikationsprotokolls unter der Verwendung des XT-Protokolls als Grundlage folgt in Abschnitt 4.2.3, um dann in Abschnitt 4.2.4 die Implementierung vorzustellen.

4.2.1 SLServer

Der `SLServer` bildet den Kommunikationsendpunkt auf Hostseite. Der Server wird mit dem Programm XT_Peer-GUI gestartet und wartet auf eingehende Daten-Verbindungen. Standardmäßig lauscht der Server auf TCP-Port 5001 auf eingehende Verbindungen. Die Portnummer kann aber über die XT_Peer-GUI geändert werden.
Der Server stellt einen *concurrent-Server* dar, da mit jeder eingehenden Datenverbindung ein neuer `SLServerThread` erzeugt wird, der nur für diese eine Datenverbindung zuständig ist. Letztenendes existieren maximal n `SLServerThreads`, wobei n die Anzahl der um den `XT_Server` und `SL_Client` erweiterten RIP-Daemons in einem VNUML-Szenario ist. Jeder dieser `SLServerThreads` ist also für die Annahme und Verarbeitung der RT-Änderungs-Nachrichten, die ihm von einem RIP-Router zugesandt werden, zuständig. Dieser Sachverhalt ist in Abbildung 10 ersichtlich.

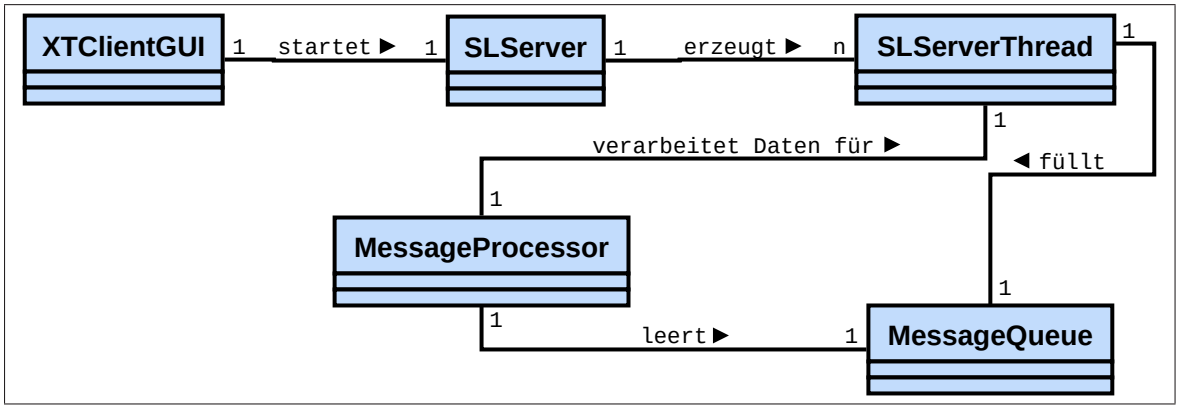


Abbildung 10: UML-Klassendiagramm des SLServer

Mit jedem `SLServerThread` wird ein weiterer Thread gestartet. Dieser Thread ist der sog. `MessageProcessor`, der parallel zu seinem `SLServerThread` und allen anderen Threads arbeitet. Der Grund für die nebenläufige Implementierung besteht darin, dass die Daten, die durch den `SLServerThread` gesammelt werden, so schnell wie möglich für die Analyseschritte anderer Threads verfügbar gemacht werden müssen. Deshalb werden eingehende Änderungsnachrichten sofort in die dafür zuständige Datenstruktur eingefügt. So besteht die Aufgabe des `SLServerThreads` nur darin, die eingegangene Nachricht auf

ihre Vollständigkeit und den Nachrichtentyp zu prüfen und in die `RTList`, die in Abschnitt 4.3 beschrieben wird, des entsprechenden Netzwerkes einzufügen. Die Änderungsnachricht wird über eine Messagequeue an den `Messageprocessor` übergeben. Die Messagequeue dient als Nachrichten-Puffer, da der `Messageprocessor` einige Zeit für die Analyse der Änderungsnachrichten in Anspruch nehmen kann.
Einzelheiten zur Verarbeitung der Änderungsnachrichten und die Verwaltung dieser Nachrichten werden in Abschnitt 4.3 erläutert.

4.2.2 SL_Client

Der `SL_Client` stellt den Kommunikationsendpunkt einer SL-Datenverbindung auf RIP-Router-Seite dar. Ein `SL_Client` wird mit dem Start des RIP-Daemons aktiviert. Dabei wartet er auf die Aufforderung, eine Verbindung zu einem `SLServer` aufzubauen bzw. abzubauen oder die Verfügbarkeit der SL-Datenverbindung zu überprüfen. Der `SL_Client` wird nur über den `XT_Server` angesteuert, der Steuerbefehle zum Verbindungsaufbau, Verbindungsabbau und Verbindungstests an den `SL_Client` weitergibt. Der `XT_Server` erhält diese Befehle vom `XT_Client` und somit von der Host-Seite. Die Hauptaufgabe des `SL_Clients` besteht darin die Änderungen, die sich in der Routing-Tabelle durch den RIP-Daemon ergeben, an den `SLServer` zu versenden.
Wann die Änderungsnachrichten gesendet werden, wurde bereits in Abschnitt 4.1.1 besprochen.

4.2.3 Entwurf des Kommunikationsprotokolls

Wie in 3.2 beschrieben, setzt diese Arbeit neben der Quagga-Routingsuite ebenfalls auf dem aus [Pä06] hervorgegangenen RIP_XT auf. Folgende Überlegung wird nun im Zusammenhang mit der Datenübertragung zwischen RIP_Daemon und Außenwelt angestellt. Die zugrundeliegende XT_Software soll dabei nicht unberührt bleiben, da der `XT_Client` die Möglichkeit bietet, Steuerbefehle an die RIP-Daemon-Seite zu senden. Das XT-Protokoll, das in Abbildung[9] 11 dargestellt ist, soll dabei nicht grundlegend verändert, sondern nur angepasst werden.
Das XT-Protokoll benutzt dabei Steuerbefehle, die in Klartext übermittelt werden. Dies bedeutet zwar einen Datenoverhead gegenüber der Verwendung von Binärdaten, wurde aber von [Pä06] als Entwicklungs- und Benutzerfreundlicher eingestuft und deshalb für die Realisierung des XT-Kommunikationsprotokolls verwendet.

Der Aufbau der XT-Verbindung und somit der Steuerverbindung bleibt unverändert. Nachdem der `XT_Client` den *"HELLO"*-Befehl an den `XT_Server` gesendet hat und die Bestätigung *"ACCEPT"* vom `XT_Server` geschickt und durch den `XT_Client` empfangen wurde, ist die XT-Steuerverbindung hergestellt. Anpassungen wurden lediglich für *ServerBefehl* und *BefehlAntwort* vorgenommen. Nachfolgend ist die zugrundeliegende Protokollsyntax in Listing 4.2.3 aus [Pä06] in EBNF nochmals dargestellt.

[9] Quelle der Abbildung [Pä06]

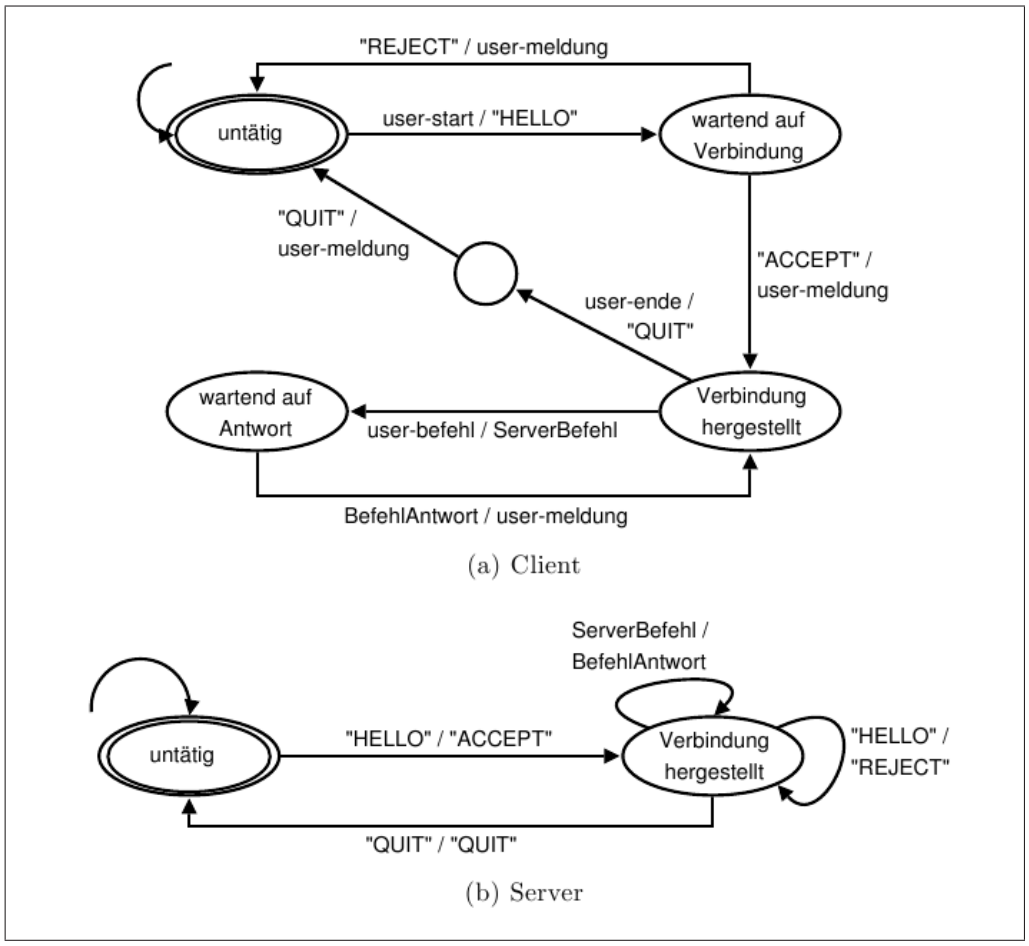


Abbildung 11: Zustandsautomat des XT-Protokolls

Die ergänzten Befehle für das Verbindungsmanagement der SL-Datenverbindung sind in der EBNF-Syntax-Darstellung aus Listing 4.2.3 fett markiert.

Für das Protokoll der SL-Datenverbindung fiel die Entscheidung ebenfalls auf die Klartextübermittlung der Daten. Zwar bedeutet dies auch einen gewissen Overhead an Daten, soll aber aus Debug-Gründen dennoch Anwendung finden.
Zum Zwecke des Aufbaus einer SL-Datenverbindung wird mit dem Verbindungsbefehl *„CONNECT_SLCLI“* auch die gewünschte Portnummer an den `XT_Server` gesandt, um bekannt zu geben, auf welchem Port der `SLServer` lauscht. Das Feld für Portnummern ist bei TCP 16 Bit groß und kann somit 65535 verschiedene Ports adressieren.
Wurde die SL-Datenverbindung erfolgreich hergestellt, wird dies mit einem *“SLCLI_CONNECTED“* vom `XT_Server` quittiert. Im Falle eines Fehlers beim Verbindungsaufbau wird vom `XT_Server` ein *“SLCLI_CONNECTION_ERROR“* zurückgeschickt.

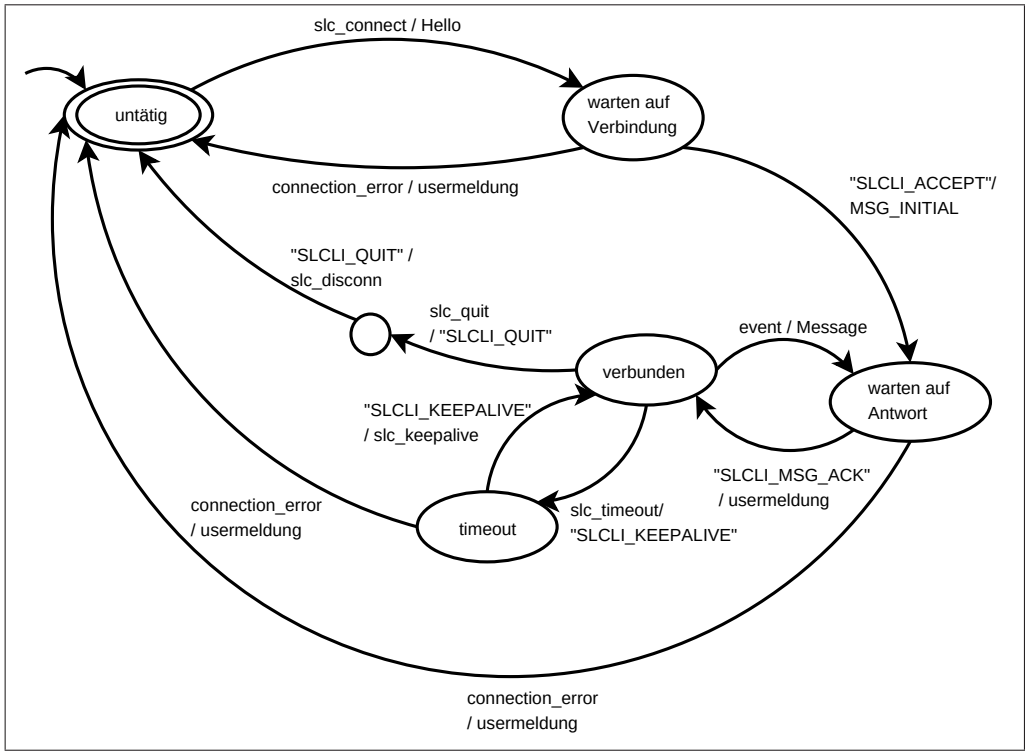
ClientNachricht	=	(ServerBefehl | ClientVerbindungsMeldung) [“\n“];
ServerBefehl	=	(InterfaceSelektor Befehl)| **SLClientVerbindungsMeldung** ;
InterfaceSelektor	=	(“IF“ InterfaceName) | “IF_ALL“;
InterfaceName	=	Zeichen {Zeichen} (* z.B. “eth0“ *);
Befehl	=	“MODE_AUTO“ | “MODE_MANUAL“
		| “TRIGGER_UPDATE“ | “SHOW_STATUS“
		| “SHOW_IP“;
ClientVerbindungsMeldung	=	“HELLO“ | “QUIT“;
ServerNachricht	=	(BefehlAntwort | ServerVerbindungsMeldung)“\n“;
BefehlAntwort	=	({InterfaceInfo} “DONE“) | (“ERROR“ ErrorCode)|
		SLServerVerbindungsMeldung;
InterfaceInfo	=	“IF“ InterfaceName (InterfaceStatus | InterfaceIP) “\n“;
InterfaceStatus	=	„MODE_AUTO“ | “MODE_MANUAL“;
InterfaceIP	=	IP-Adresse (* in “dotted decimal“-Notation *)
		/Zahl (* mit n $\in \mathbb{N}$, $1 \leq n \leq 32$ *);
ErrorCode	=	Zahl (* mit n $\in \mathbb{N}$, $100 \leq n \leq 999$ *);
ServerVerbindungsMeldung	=	“ACCEPT“ | “REJECT“ | “QUIT“;
SLClientVerbindungsMeldung	=	“CONNECT_SLCLI:“ Portnumber
		| “DISCONNECT_SLCLI“ | “SLCLI_CONN_TIMEOUT“;
Portnumber	=	Zahl (* mit n $\in \mathbb{N}$, $1 \leq n \leq 65535$ *);
SLServerVerbindungsMeldung	=	“SLCLI_CONNECTED“ | “SLCLI_DISCONNECTED“
		| “SLCLI_CONNECTION_ERROR“;

Listing 4.2.3: EBNF der erweiterten XT-Protokoll-Syntax

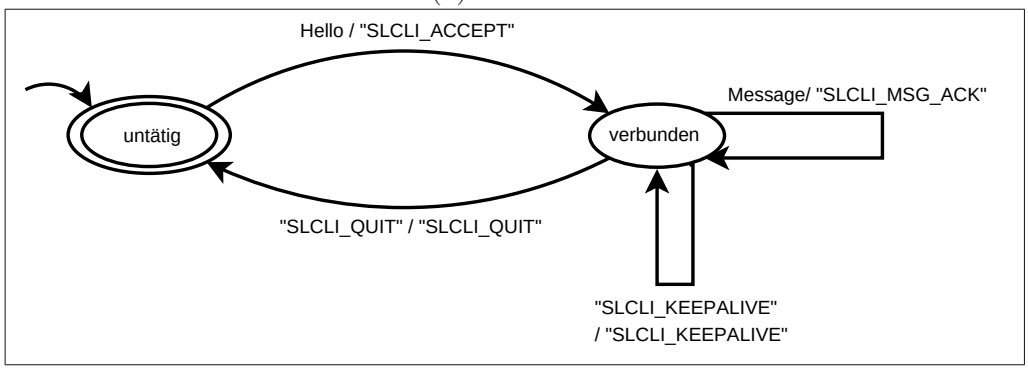
Die Kommunikation zwischen `SL_Client` und `SLServer` wird durch die in Abbildung 12(a) und 12(b) dargestellten Automaten realisiert. Dabei wird für jede SL-Datenverbindung eine TCP-Verbindung verwendet, um eine zuverlässige Datenübertragung zu gewährleisten, da die folgenden Punkte für die spätere Analyse der Erreichbarkeitsinformationen garantiert werden müssen:

1. alle gesendeten Daten kommen vollständig an
2. die Daten kommen in der richtigen Reihenfolge an
3. es gibt keine Duplikate

Eine SL-Datenverbindung wird erstellt, indem der `SLServer` von einem `SL_Client` eine *“SLCLI_HELLO“*-Nachricht empfängt. Die Werte für *UpdateTime* und *RMI_Value* werden dabei als Parameter mitgeliefert. Der Parameter *UpdateTime* gibt dabei den Timer-Wert an, der in der Konfiguration des RIP-Routers vorliegt und die Zeitdauer angibt, in der periodische Updates unaufgefordert an die Nachbarrouter versandt werden. An diesem Timer orientiert sich der Timeout-Timer des `SLServerThreads`. Dieser Timeout Timer dient dazu, bei ausbleibenden Änderungsnachrichten über die Dauer des Update-Timers hinweg zu prüfen, ob die Datenverbindung zwischen `SL_Client` und `SLServer` noch vorhanden ist. Der Wert für diesen Timeout-Timer beträgt $updatetime + \frac{updatetime}{10}$.
Der zweite Parameter, die *RMI_Value* gibt dabei den Wert an, den die RIP-Konstante



(a) SL_Client



(b) SLServer

Abbildung 12: kommunizierende Automaten des SL-Protokolls

Client_Nachricht	=	Hello | Message | ((“SLCLI_KEEPALIVE““| “SLCLI_QUIT““)\n“);
Hello	=	“SLCLI_HELLO“ “!-!“ UpdateTime “!-!“ RMI_Value “\n“;
Server_Nachricht	=	(“SLCLI_ACCEPT“| “SLCLI_KEEPALIVE“|“SLCLI_MSG_ACK“| “SLCLI_QUIT“)“\n“;
UpdateTime	=	Zahl (* n ∈ ℕ\{0}*);
RMI_Value	=	Zahl (* n ∈ ℕ\{0}*);
Message	=	MSG_Update | MSG_Initial | MSG_RTE_Timeout | MSG_MTI_Debug_Info;
MSG_Update	=	“MSG_UPDATE“RT_Info;
MSG_Initial	=	“MSG_INITIAL_MSG“RT_Info;
MSG_RTE_Timeout	=	“MSG_RTE_TIMEOUT“RT_Info;
MSG_MTI_Debug_Info	=	“MSG_MTI_DEBUG_INFO““!-!“Uptime “!-!“Network “!-!“ Old_If_Index “!-!“Old_Metric “!-!“New_If_Index “!-!“New_Metric “\n“;
RT_Info	=	“!-!“Uptime “!-!“Network “!-!“NextHop “!-!“Metric “!-!“LearnedFrom “!-!“Tag “!-!“Time “\n“;
New_If_Index	=	InterfaceName If_Index;
Old_If_Index	=	InterfaceName If_Index;
If_Index	=	Zahl(*n ∈ ℕ*);
Old_Metric	=	Metric;
New_Metric	=	Metric;
Metric	=	Zahl(*n ∈ ℕ\{0}*);
Network	=	IP-Adresse (* in “dotted decimal“-Notation *) /Zahl (* mit n ∈ ℕ, 1 ≤ n ≤ 32 *);
LearnedFrom	=	IP-Adresse (* in “dotted decimal“-Notation *);
NextHop	=	IP-Adresse (* in “dotted decimal“-Notation *);

Listing 4.2.3: EBNF der SL-Protokoll-Syntax

RIP_METRIC_INFINITY besitzt. Standardmäßig ist diese Konstante mit dem Wert 16 versehen, kann aber für Testzwecke im Quelltext des RIP-Daemon verändert werden. Da die CTI-Erkennung des `SLServers` u.a. auf dem Wert dieser Konstante aufbaut, ist es notwendig, diesen Wert gleichermaßen dem `SLServer` bekannt zu geben.

Der `SLServer` nimmt generell jeden Verbindungswunsch eines `SL_Clients` an. Die Prüfung, ob bereits eine SL-Datenverbindung besteht wird durch den jeweiligen `SL_Client` vorgenommen. Den Verbindungswunsch beantwortet der `SLServer` mit einem “SLCLI_ACCEPT“, woraufhin der `SLServer` in den Zustand “verbunden“ übergeht. In diesem Zustand ist er in der Lage, RT-Änderungsnachrichten vom `SL_Client` zu empfangen und diese zu verarbeiten. Sobald die RT-Änderungsnachricht vom `SLServer` entgegengenommen und in der vorgesehenen Datenstruktur abgelegt wurde, quittiert der `SLServer` den Empfang der Nachricht mit einem „SLCLI_MSG_ACK“. Ein einfaches Entgegennehmen der Nachricht, ohne eine Bestätigung währe auch denkbar gewesen. Wie aber in Abschnitt 4.1.1 beschrieben, wird versucht eine Ordnung auf der Abfolge des Eintreffens der RT-Änderungsnachrichten beim `SLServer` inklusive der dortigen Bereitstellung der übermittelten Daten für die spätere Analyse und des Eintreffens von triggered Updates bei den Nachbarroutern zu bewerkstelligen, damit eine Analyse auf den Erreichbarkeitsinformationen durchgeführt werden kann. Zu diesem Zweck wird dabei zum einen die Art des Scheduling quagga-eigener Threads und die Integration des Nachrichtenversands an den `SLServer` ausgenutzt -dieser Nachrichtenversand wurde, wie beschrieben nicht als eigener Thread, sonders als Bestandteil des `RTE_Processes` bzw. `Timeout`-Prozesses realisiert-, zum anderen die Designentscheidung, dass der Eingang

der RT-Änderungsnachricht erst bestätigt wird, nachdem der `SLServer` diese in der dafür vorgesehenen Datenstruktur abgelegt hat. TCP sorgt zwar für eine zuverlässige Übertragung, liefert aber kein Indiz dafür, ob die übertragenen Daten auf den Schichten oberhalb der Transportschicht des ISO/OSI-Referenzmodells verarbeitet wurden.

Die Designentscheidung bringt zwei Effekte mit sich:

- Wird eine RT-Änderungsnachricht an den `SLServer` versendet, wartet der `RTE_Process` bzw. `Timeout`-Prozess, bis eine Bestätigung über die Verarbeitung der Nachricht vom `SLServer` eingeht. Dies garantiert die angesprochene beabsichtigte Ordnung auf der Sendefolge.
- Kommte es aufgrund einer nicht mehr vorhandenen SL-Datenleitung dazu, dass Bestätigungen vom `SLServer` beim `SL_Client` ausbleiben, wartet der `RTE_Process` bzw. `Timeout`-Prozess vergeblich und terminiert für diese Zeitspanne nicht. Somit kommt es, aufgrund des in Abschnitt 4.1 beschriebenen kooperativen Scheduling, zum Aushungern anderer RIP-Prozesse.

Es ist an dieser Stelle also ein Kompromiss notwendig:

1. Das Sammeln der Simulationsdaten steht im Vordergrund.
2. Die CTI-Erkennung baut auf den gesammelten Daten auf und wird durch nicht eintreffende Daten verfälscht.
3. Eine Unterbrechung der SL-Datenleitung geht möglicherweise mit einer gleichzeitigen Unterbrechung der XT-Steuerleitung einher.

Aus diesen drei Gründen ergibt sich folgender Schluss:
Das Aushungern von RIP-Threads geht mit einer fehlenden SL-Datenleitung oder/und einer Unterbrechung der XT-Steuerleitung einher. Ist die SL-Datenleitung unterbrochen, kann das Sammeln der notwendigen Daten nicht fortgesetzt werden und die CTI-Erkennung wird verfälscht. Eine Unterbrechung der XT-Steuerleitung verhindert das automatisierte Triggern von Updates, sodass die möglicherweise dennoch gesammelten Daten unbrauchbar sind. Ein korrektes Arbeiten des RIP ist somit irrelevant.
Das Abbrechen der XT-Steuerleitung oder der SL-Datenleitung stellt einen Sonderfall dar, der während der zahlreichen Simulationen, die während dieser und der Arbeit von Tim Keupen [Keu07] nie auftrat. Diese genannten Gründe rechtfertigen die Designentscheidung.

Um eine Möglichkeit zu bieten, die Verfügbarkeit der SL-Datenleitung zu überprüfen, wird eine serverseitiger Timeout-Mechanismus verwendet. Nach dem Ablaufen des Timers werden Nachrichten zwischen `SL_Client` und `SLServer` ausgetauscht, um die Datenleitung auf ihre Verfügbarkeit hin zu testen. Aufgrund der Tatsache, dass ein RIP-Router nach einer bestimmten Zeit periodische Updates sendet, treffen bei den Nachbarroutern ebenfalls nach Ablauf dieser Zeit Updates ein. Die wiederum lösen das Senden einer Nachricht an den `SLServer` aus, sodass nach einer bestimmten Periode immer eine Nachricht beim

SLServer erwartet wird. Es wird dabei davon ausgegangen, dass die Nachbarrouter eines RIP-Router ähnlichen Konfigurationen im Bezug auf die Update Time für periodische Updates unterliegen. Sollte es aufgrund des Ausbleibens eines periodischen Updates und somit auch des Ausbleibens einer RT-Änderungsnachricht zu einem Timeout kommen, wird der XT_Client damit beauftragt, dem XT_Server eine *"SLCLI_CONN_TIMEOUT"*-Nachricht zu senden. Der XT_Server wird dann wiederum den SL_Cient damit beauftragen, eine *"SLCLI_KEEPALIVE"*-Nachricht an den SLServer zu senden, der diese dann ebenfalls mit einer *"SLCLI_KEEPALIVE"*-Nachricht quittiert. Dies signalisiert, dass die SL-Datenverbindung nach wie vor besteht. Sollte die SL-Datenverbindung unterbrochen sein, wird dies dem XT_Client durch eine *"SLCI_CONNECTION_ERROR"*-Nachricht mitgeteilt und der SLServerThread wird beendet, damit ggf. eine neue SL-Datenverbindung aufgebaut werden kann. Durch eine entsprechende *user-meldung* wird die verloren gegangene SL-Datenverbindung mitgeteilt.

Auf SL_Client-Seite wird der Verbindungsaufbau initiiert, indem der XT_Server vom XT_Client die Aufforderung zum Aufbau einer SL-Datenverbindung erhält. Sodann wird der SL_Client durch *slc_connect* aufgefordert eine *Hello*-Nachricht an den SLServer zu versenden.
Wird der Verbindungswunsch durch den SLServer mit einem *"SLCLI_ACCEPT"* bestätigt, wird daraufhin vom SL_CLient der gesamte Inhalt der Routing-Tabelle des Routers in Form einer *MSG_INTINIAL*-Nachricht an den SLServer versendet, damit die Daten, die im SLServer mit denen des RIP-Routers übereinstimmen. Im Zustand *verbunden* werden Aufforderungen eine SL-Datenverbindung aufzubauen abgewiesen. Durch ein *event*, d.h. einer Veränderung der Routing-Tabelle werden durch den SL_Client entsprechende Änderungs-Nachrichten in Form von *Message*-Nachrichten an den SLServer gesendet. Kommt es aufgrund eines auf SLServer-Seite aufgetretenen Timeouts zu der *slc_timeout*-Benachrichtigung durch den XT_Server, wird eine *"SLCLI_KEEPALIVE"*-Nachricht an den SLServer geschickt. Wird diese ebenfalls durch eine *"SLCLI_KEEPALIVE"*-Nachricht vom SLServer bestätigt, wird der Nachrichtenverkehr im Zustand *verbunden* wieder aufgenommen. Im anderen Fall unterbricht der SL_Client die Datenverbindung und wartet im Zustand *untätig* auf die Aufforderung eine neue Datenverbindung aufzubauen.
Die SL-Datenverbindung wird durch das Senden einer *"SLCLI_QUIT"*-Nachricht an den SLServer und dessen Bestätigung ebenfalls durch eine *"SLCLI_QUIT"*-Nachricht abgebaut. Die Aufforderung *slc_quit* zum Abbau der SL-Datenverbinung geschieht wiederum durch den XT_Server.
Die Protokoll-Syntax der SL-Datenverbindung ist in Listing 4.2.3 dargestellt. Hier wurden die Regeln für die Auflösung einiger Nichtterminalsymbole der Übersichtlichkeit halber durch Kommentare ersetzt.

4.2.4 Implementation des Kommunikationsprotokolls

Im folgenden soll die Umsetzung des Kommunikationsprotokolls beschrieben werden. Dabei werden die notwendigen Erweiterungen des XT-Protokolls und die Kernfunktionen bzw. -Methoden, die das SL-Protokoll realisieren, beschrieben.

Die Modifikationen am XT-Protokoll wurden durch die entsprechenden Anpassungen in der `xt_clint.h` aus Listing 6 in den Zeilen 60 - 65 und Zeilen 75 -94 vorgenommen.

In Listing 7 ist das Messages-Interface auf `XT_Client`-Seite dargestellt. Dieses Interface wurde insofern erweitert, als dass es nun auch die Definition der Nachrichtentypen für den `SLServer` beinhaltet. Neben den bekannten Nachrichten, die zur Kommunikation zwischen `XT_Client` und `XT_Server` dienen, sind nun 14 weitere Nachrichtentypen ergänzt worden. Fünf (Zeile 17-22) davon stellen Steuernachrichten dar, die den Verbindungsstatus zwischen `SL_Client` und `SL_Server` beschreiben und zwischen `XT_Client` und `XT_Server` ausgetauscht werden, wie es oben bereits erläutert wurde. Fünf Befehle (Zeile 23-27) werden zum Zwecke des Verbindungaufbaus und -Abbaus und der Verbindungsüberwachung zwischen `SLServer` und `SL_Client` verwendet. Die restlichen Befehle (ab Zeile 28) werden zur Identifikation der Nachrichtentypen verwendet, die zwischen `SL_Client` und `SLServer` ausgetauscht werden.

```
57
58 ...
59 ...
60 #define RIP_XT_MSG_CONNECT_SLCLI                14
61 #define RIP_XT_MSG_SLCLI_CONNECTED              15
62 #define RIP_XT_MSG_SLCLI_CONNECTION_ERROR       16
63 #define RIP_XT_MSG_SLCLI_DISCONNECT             17
64 #define RIP_XT_MSG_SLCLI_DISCONNECTED           18
65 #define RIP_XT_MSG_SLCLI_CONN_TIMEOUT           19
66 /**
67  * @brief Since the communication is based on
68  * human-readable text, this array is not only used
69  * for logging purposes, but for mapping protocol
70  * messages to message numbers and vice versa.
71  * (I.e., both keys and strings must be unique)
72  */
73 static const struct message rip_xt_msg[] =
74 {
75     {RIP_XT_MSG_HELLO,                  "HELLO"},
76     {RIP_XT_MSG_QUIT,                   "QUIT"},
77     {RIP_XT_MSG_IFSEL_SINGLE,           "IF"},
78     {RIP_XT_MSG_IFSEL_ALL,              "IF_ALL"},
79     {RIP_XT_MSG_MODE_AUTO,              "MODE_AUTO"},
80     {RIP_XT_MSG_MODE_MANUAL,            "MODE_MANUAL"},
81     {RIP_XT_MSG_TRIGGER_UPDATE,         "TRIGGER_UPDATE"},
82     {RIP_XT_MSG_SHOW_STATUS,            "SHOW_STATUS"},
83     {RIP_XT_MSG_SHOW_IP,                "SHOW_IP"},
84     {RIP_XT_MSG_ACCEPT,                 "ACCEPT"},
85     {RIP_XT_MSG_REJECT,                 "REJECT"},
86     {RIP_XT_MSG_DONE,                   "DONE"},
87     {RIP_XT_MSG_ERROR,                  "ERROR"},
88     {RIP_XT_MSG_CONNECT_SLCLI,          "CONNECT_SLCLI"},
89     {RIP_XT_MSG_SLCLI_CONNECTED,        "SLCLI_CONNECTED"},
90     {RIP_XT_MSG_SLCLI_CONNECTION_ERROR, "SLCLI_CONNECTION_ERROR"},
91     {RIP_XT_MSG_SLCLI_DISCONNECT,       "SLCLI_DISCONNECT"},
```

```
92      {RIP_XT_MSG_SLCLI_DISCONNECTED,      "SLCLI_DISCONNECTED"},
93      {RIP_XT_MSG_SLCLI_CONN_TIMEOUT,      "SLCLI_CONN_TIMEOUT"},
94      {0,                                  NULL}
95  };
```

Listing 6: Anpassung der Datei rip_xt.h

```
1  package backend;
2
3  public interface Messages {
4      public static final String HELLO                          = "HELLO";
5      public static final String QUIT                           = "QUIT";
6      public static final String IFSEL_SINGLE                   = "IF";
7      public static final String IFSEL_ALL                      = "IF_ALL";
8      public static final String MODE_AUTO                      = "MODE_AUTO";
9      public static final String MODE_MANUAL                    = "MODE_MANUAL";
10     public static final String TRIGGER_UPDATE                 = "TRIGGER_UPDATE";
11     public static final String SHOW_STATUS                    = "SHOW_STATUS";
12     public static final String SHOW_IP                        = "SHOW_IP";
13     public static final String ACCEPT                         = "ACCEPT";
14     public static final String REJECT                         = "REJECT";
15     public static final String DONE                           = "DONE";
16     public static final String ERROR                          = "ERROR";
17     public static final String CONNECT_SLCLI                   = "CONNECT_SLCLI_PORT";
18     public static final String SLCLI_CONNECTED                 = "SLCLI_CONNECTED";
19     public static final String SLCLI_MSG_DISCONNECT            = "SLCLI_DISCONNECT";
20     public static final String SLCLI_MSG_CONNECTION_ERROR
21                                                    = "SLCLI_CONNECTION_ERROR";
22     public static final String SLCLI_MSG_CONN_TIMEOUT          = "SLCLI_CONN_TIMEOUT";
23     public static final String SLCLI_MSG_HELLO                 = "SLCLI_HELLO";
24     public static final String SLCLI_MSG_ACCEPT                = "SLCLI_ACCEPT";
25     public static final String SLCLI_MSG_ACK                   = "SLCLI_MSG_ACK";
26     public static final String SLCLI_MSG_KEEPALIVE             = "SLCLI_KEEPALIVE";
27     public static final String SLCLI_MSG_QUIT                  = "SLCLI_QUIT";
28     public static final String SLCLI_MSG_UPDATE                = "MSG_UPDATE";
29     public static final String SLCLI_MSG_RTE_TIMEOUT           = "MSG_RTE_TIMEOUT";
30     public static final String SLCLI_MSG_INITIAL_MSG           = "MSG_INITIAL_MSG";
31     public static final String SLCLI_MSG_MTI_DEBUG_INFO        = "MSG_MTI_DEBUG_INFO";
32 }
```

Listing 7: Anpassung des Interfaces Messages.java

Folgende vier Nachrichtentypen werden unterschieden:

- MSG_UPDATE: Nachrichten, die durch eine Änderung an der Routing-Tabelle eines RIP-Routers erzeugt wurden. Dabei rühren die Änderungen an der Routing-Tabelle von einem vom Router empfangenen Update über Erreichbarkeitsinformationen her. Unter einem Update der Routing-Tabelle wird im folgenden entweder eine Änderung an einem bestehenden Eintrag der Routing-Tabelle oder das Hinzufügen eines Eintrags in die Routing-Tabelle verstanden. Auch im Falle des Abweisens eines RIP-Updates wird eine solche Nachricht (vergl. Abschnitt 4.1.1) erzeugt.
- MSG_INITIAL_MSG: Wird eine Verbindung zwischen einem `SL_Client` und einem `SLServer` hergestellt, werden dem `SLServer` alle im RIP-Router verfügbaren Daten zugesandt. Somit kann sichergestellt werden, dass die Informationen auf Router-Seite und die Informationen, die auf Seiten des `SLServers` vorliegen, konsistent sind und beide Seiten die gleichen Informationen besitzen.

- MSG_RTE_TIMEOUT: Kommt es bei einer Erreichbarkeitsinformation in einer Routing-Tabelle zum Ablaufen des *TimeoutTimers*, so wird diese Erreichbarkeitsinformation in der Routing-Tabelle des RIP-Routers auf den Wert RIP_METRIC_INFINITY gesetzt (vergl. [Qua] und [Mal98]) und der *GarbageCollection Timer* wird für diese Route gestartet. In diesem Fall wird dem `SLServer` diese Änderung bekannt gegeben.
- MSG_MTI_DEBUG_INFO: Wird aufgrund eines Updates die Funktion *rip_mti_routeok* des MTI aufgerufen, wird in dieser Nachricht die DEBUG-Information an den `SLServer` gesendet, die auch in der Datei ripd.log auf RIP-Router Seite zu finden ist.

In Listing 8 ist die Nachrichtendefinition in der Header-Datei `sl_client.h` auf Seiten des `SL_Clients` zu sehen.

```
1 #ifndef _ZEBRA_RIP_SLCLIENT_H
2 #define _ZEBRA_RIP_SLCLIENT_H
3
4 #include <zebra.h>
5 #include "thread.h"
6 #include "vector.h"
7 #include "log.h"
8
9 /**
10  * @brief Various constants
11  */
12 #define RIP_SLC_BUFSIZE  2048
13
14 /**
15  * @brief The messages which are used for communication
16  *        between the SL_client and the XT_Peer.
17  */
18 #define RIP_SLC_MSG_QUIT              1
19 #define RIP_SLC_MSG_UPDATE            2
20 #define RIP_SLC_MSG_RTE_TIMEOUT       3
21 #define RIP_SLC_MSG_ACK               4
22 #define RIP_SLC_MSG_INITIAL           5
23 #define RIP_SLC_MSG_MTI_DEBUG_INFO    6
24 #define RIP_SLC_MSG_KEEPALIVE         7
25
26 /**
27  * @brief Since the communication is based on
28  * human-readable text, this array is not only used
29  * for logging purposes, but for mapping protocol
30  * messages to message numbers and vice versa.
31  * (I.e., both keys and strings must be unique)
32  */
33 static const struct message rip_slc_msg[] =
34 {
35     {RIP_SLC_MSG_QUIT,                "SLCLI_QUIT"},
36     {RIP_SLC_MSG_HELLO,               "SLCLI_HELLO"},
37     {RIP_SLC_MSG_ACCEPT,              "SLCLI_ACCEPT"},
38     {RIP_SLC_MSG_ACK,                 "SLCLI_MSG_ACK"},
39     {RIP_SLC_MSG_KEEPALIVE,           "SLCLI_KEEPALIVE"},
40     {RIP_SLC_MSG_UPDATE ,             "MSG_UPDATE"},
41     {RIP_SLC_MSG_RTE_TIMEOUT,         "MSG_RTE_TIMEOUT"},
42     {RIP_SLC_MSG_INITIAL,             "MSG_INITIAL_MSG"},
43     {RIP_SLC_MSG_MTI_DEBUG_INFO,      "MSG_MTI_DEBUG_INFO"},
44     {0,                               NULL}
45 };
```

Listing 8: Protokolldefinition in der sl_client.h

```
42 /**
43  * @brief Stores everything about the SL_Client
44  *       which must be globally available.
45  */
46 struct rip_slc_state {
47     /*is set to 1 if the SL_Connection is established*/
48     int connected;
49     int mySocket;
50
51     /*is set to one if there is a message to send*/
52     int message_available;
53
54     /*the port the SL_Server is listening on*/
55     unsigned short serverPort;
56
57     /* the SL_Servers address*/
58     char *server_Address;
59
60     /* buffer is used for incoming and outgoing messages. */
61     char *tmp_buffer;
62     char *send_buffer;
63     char *recv_buffer;
64
65     /*some variable that are used to determine the update-generation time*/
66     struct tm *uptime;
67     struct timeval uptime_usec;
68     long sim_time_sec;
69
70     unsigned long update_time;
71     int rmi_value;
72 };
73 /* Function declarations */
74 ...
75
76 #endif /* _ZEBRA_RIP_SLCLIENT_H */
```

Listing 9: Das struct rip_slc_state der sl_client.h

In Listing 9 ist das *struct rip_slc_state* dargestellt. Dieses *struct* speichert alle Informationen, die den Verbindungsstatus und den Nachrichtenversand zwischen `SL_Client` und `SLServer` unterstützen. In Zeile 48 ist der Status der Verbindung definiert. Befindet sich der `SL_Client` in einer Verbindungs-Beziehung mit einem `SLServer`, so ist der Wert der *connected*-Variable 1. Diese Variable wird dazu verwendet abzufragen, ob eine Verbindung mit einem `SLServer` aufgebaut werden darf oder ob bereits eine Verbindung besteht. Im Falle einer bestehenden Verbindung wird der `XT_Server`, der einen SL-Verbindungswunsch empfangen hat, dazu angehalten, die Steuernachricht *"MSG_SLC_CONNECTION_ERROR"* (vergl. Listing 6) an den `XT_Client` zu senden, der diesen Verbindungswunsch geäußert hat. Somit ist sichergestellt, dass ein `SL_Client`, der mit einem `SLServerThread` eine Verbindung unterhält, keine neue Verbindung aufbaut.
Die Variable *mySocket* enthält Informationen über den Socket, der für die Verbindung verwendet wird. In Zeile 55 wird in der Variable *serverPort* gespeichert, auf welchem Port der `SLServer` auf eingehende Daten-Verbindungen lauscht. Der Wert für diese Variable wird bei dem Verbindungsbefehl *CONNECT_SLCLI* als Argument mitgesandt und durch den

XT_Server gesetzt. In der Variable *server_Address* ist die IP-Adresse des Servers festgehalten. Der Wert wird ebenfalls durch den XT_Server gesetzt und entspricht der IP-Adresse des verbundenen XT_Clients.
Die in Zeile 52 definierte Variable *message_available* dient der Überprüfung, ob der Sendepuffer aus Zeile 62 Daten enthält, die gesendet werden sollen.
In den darauffolgenden Zeilen 61 - 63 werden diverse Puffer definiert. Der *send_buffer* ist der Puffer, aus dem die zu sendenden Nachrichten ausgelesen werden. Der *tmp_buffer* dient der Erzeugung der Nachrichten. Der *recv_buffer* dient dem Empfang von Bestätigungsnachrichten für Verbindungsaufbau, Verbindungsabbau, *"SLCLI_KEEPALIVE"* und *"SLCLI_MSG_ACK"*-Nachrichten.
Die Variabeln *uptime*, *uptime_usec* und *sim_time_sec* werden für die in Abschnitt 4.5.1 beschriebene Erzeugung von Zeitstempeln bei der Generierung von Änderungsnachrichten verwendet.
Die Variable *update-time* enthält den Wert des update-Timers für periodische Updates, wie er in der Konfiguration des RIP-Routers festgelegt ist. Wie bereits beschrieben, wird anhand dieser Variablen der Timeout-Wert für die SL-Datenverbindung ermittelt.
Die Variable *rmi_value* beinhaltet den Wert der RIP_METRIC_INFINITY-Konstante.

Wie bereits erwähnt, dient die XT-Verbindung als Steuerleitung zum Aufbau, Abbau und Überprüfung einer SL-Datenverbindung. Damit dies möglich ist, muss der XT_Server und der XT_Client um die Fähigkeit erweitert werden, die jeweiligen SL-Verbindungsnachrichten verarbeiten zu können. In Listing 10 ist die Ergänzung der Funktion *parse_message* des XT_Servers, die zur Verarbeitung von XT-Steuerbefehlen dient, aufgezeigt.

```
429 ...
430 ...
431 case RIP_XT_CONSTATE_ESTABLISHED:{
432   struct rip_xt_ifstate * interface = NULL;
433   switch (token){
434     case RIP_XT_MSG_CONNECT_SLCLI:{
435       /*
436        * to establish the sl_connection parse the incomming buffer again to
437        * obtain the port the sl_server is listening on
438        */
439       unsigned short serverPort = atoi(get_next_token(NULL));
440       if(xt_sl_establish_con(inet_ntoa(xt_state->connected_addr->sin_addr),
441                                                                 serverPort)<0){
442          send_strings(xt_state->connected_socket, rip_xt_lookup_msg(
443                                          RIP_XT_MSG_SLCLI_CONNECTION_ERROR),"\n");
444          zlog_warn("XT: SL_Client Connection cannot be established!");
445          break;
446       }
447     zlog_info("XT: Client has send CONNECT_SLCLI");
448     send_strings(xt_state->connected_socket, rip_xt_lookup_msg(
449                                                RIP_XT_MSG_SLCLI_CONNECTED),"\n");
450     if(xt_sl_send_initial()<0){
451        zlog_warn("XT: SL_Client can not send initial message!");
452     }
453     break;}
454
455    case RIP_XT_MSG_SLCLI_DISCONNECT:
456     if(slc_state->connected >=0){
457       xt_sl_release_con();
458       zlog_info("XT: Client has send DISCONNECT_SLCLI");
459     }
```

```
460     else{
461       zlog_info("XT: Client has send DISCONECT_SLCLI,
462 ___________________________________________but SLCLIE already disconnected!");
463     }
464     break;
465
466   case RIP_XT_MSG_SLCLI_CONN_TIMEOUT:
467     if(slc_state->connected >= 0){
468       if(xt_sl_send_keepalive()<0){
469           send_strings(xt_state->connected_socket, rip_xt_lookup_msg(
470                                                 RIP_XT_MSG_SLCLI_CONNECTION_ERROR), "\n");
471           zlog_warn("XT: SL_Client Connection corrupted!");
472       }
473       else{
474         send_strings(xt_state->connected_socket, rip_xt_lookup_msg(
475                                                                 RIP_XT_MSG_DONE), "\n");
476       }
477     }
478     break;
479 ...
```

Listing 10: Erweiterung der Funktion parse_message des XT_Servers

Befindet sich der `XT_Server` entsprechend des Automaten aus Abbildung 11 im Zustand „Verbindung hergestellt“, kann er Serverbefehle verarbeiten. Befindet sich der `XT_Server` in diesem Zustand und empfängt er den Serverbefehl *“CONNECT_SLCLI“*, wird der SL-Verbindungsaufbau durch Aufruf der Funktion *xt_sl_establish_con* in Zeile 440 unter Übergabe der IP-Adresse und des Serverports, der aus dem Serverbefehl in Zeile 439 extrahiert wurde, als Parameter eingeleitet. Im Erfolgsfall wird die entsprechende Bestätigung (vergl. Zeile 448) an den `XT_Client` zurückgesandt und im Fehlerfall die Nachricht *“SLCLI_CONNECTION_ERROR“*. Die Funktion *xt_sl_establish_con* des `SL_Clients` ist in Listing 11 zu sehen.

```
52 /**
53  * This function sets up the connection.
54  * @param ipAddress the IPAddress of the Server
55  * @return 0 if the connection could be established, -1 otherwise
56  */
57 int xt_sl_establish_con(char *ip_Address, unsigned short serverPort){
58   /*checks if the connection is already established*/
59   if(slc_state->connected == 1){
60     zlog_warn("sl_client: client already connected!");
61     return -1;
62   }
63   /*create a socket*/
64   if(xt_sl_create_socket()<0)
65     return -1;
66
67   /*set up the connection*/
68   struct sockaddr_in server_Address;
69   slc_state->serverPort = serverPort;
70   memset(&server_Address, 0, sizeof(server_Address));
71   server_Address.sin_family      = AF_INET;
72   server_Address.sin_addr.s_addr = inet_addr(ip_Address);
73   server_Address.sin_port        = htons(slc_state->serverPort);
74   slc_state->server_Address = inet_ntoa(server_Address.sin_addr);
75
76   if (connect(slc_state->mySocket, (struct sockaddr *) &server_Address,
```

```
77                                                  sizeof(server_Address)) < 0){
78       slc_state->connected = -1;
79       zlog_err("sl_client: Connecting to %s failed: %s",
80       inet_ntoa(server_Address.sin_addr),safe_strerror(errno));
81       return -1;
82     }
83     /*Print the HELLO-message into the send_buffer*/
84     sprintf(slc_state->send_buffer,"%s!-!%d!-!%d\n",
85                                     lookup(rip_slc_msg, RIP_SLC_MSG_HELLO),
86                                slc_state->update_time, slc_state->rmi_value);
87     /*Send the HELLO-message*/
88     if(xt_sl_send() < 0)
89        return -1;
90     /*receive the ACCEPT-message from the server*/
91     if(xt_sl_receive() < 0)
92        return -1;
93     if(strcmp(slc_state->recv_buffer,lookup(rip_slc_msg, RIP_SLC_MSG_ACCEPT)) < 0)
94        return -1;
95     slc_state->connected = 1;
96     zlog_info("SLCLI: Client successfully connected to %s",
97                                                          slc_state->server_Address);
98     return 0;
99 }
```

Listing 11: Die Funktion xt_sl_establish_con der sl_client.c

Analog dazu läuft der Verbindungsabbau ab Zeile 455 in Listing 10 ab.
Die Verarbeitung der Timeout-Nachricht wird in Zeile 466 vorgenommen und durch die Funktion `xt_sl_send_keepalive` aus Listing 12 realisiert.

```
102 /* This method is called due to a timeout, to check the connection.
103  * @return 0 if the connection is available, -1 otherwise
104  */
105 int xt_sl_send_keepalive(){
106     sprintf(slc_state->send_buffer,"%s\n",
107                                     lookup(rip_slc_msg, RIP_SLC_MSG_KEEPALIVE));
108     if(xt_sl_send() < 0){
109        slc_state->connected = -1;
110        return -1;
111     }
112     if(xt_sl_receive() < 0){
113        slc_state->connected = -1;
114        return -1;
115     }
116     if(strcmp(slc_state->recv_buffer,
117                                lookup(rip_slc_msg, RIP_SLC_MSG_KEEPALIVE)) < 0){
118       slc_state->connected = -1;
119       return -1;
120     }
121     return 0;
122 }
```

Listing 12: Die Funktion `xt_sl_send_keepalive` der sl_client.c

Die notwendigen Anpassungen auf `XT_Client`-Seite umfassen die Ergänzung der Klasse `XTServer`, der ein logisches Äquivalent zum `XT_Server` auf VNUML-Seite darstellt und eine Seite der Kommunikation zwischen `XT_Server` und `XT_Client` realisiert, stellen die Methoden *connectSLCLI* in Listing 13 und *diconSLCLI* in Listing 14 dar. Diese Methoden leiten den Verbindungsauf- bzw. Abbau ein. Die Methode `isSLCLIConnected` aus Listing 15

übernimmt das Senden von Timeout-Nachrichten.

```
348 /**
349  * Notifies the XT_Server to urge the SL_Client to connect.
350  * @param serverPort The port the SL-Server is listening on.
351  * @throws IOException
352  * @throws XTServerException
353  */
354  public void connectSLCLI(String serverPort)throws IOException, XTServerException{
355    send(Messages.CONNECT_SLCLI.replace("PORT",serverPort));
356    String received = receive();
357    if(received.trim().equals(Messages.SLCLI_CONNECTED)){
358       if(gui.DEBUG) System.out.println(Messages.SLCLI_CONNECTED);
359       if(gui.DEBUG) System.out.println("Server setup");
360    }
361    else{
362      System.err.println(received.trim());
363      System.err.println("SLCLient can't connect to the SLServer");
364    }
365 }
```

Listing 13: Die Methode connectSLCLI der Klasse XTServer

```
348 /**
349  * Notifies the corresponding XT_Server to urge the sl_cli to disconnect.
350  */
351 public void disconnSLCLI(){
352   if(!socket.isClosed())
353      send(Messages.SLCLI_MSG_DISCONNECT);
354   }
```

Listing 14: Die Methode disconnSLCLI der Klasse XTServer

```
348 /**
349  * Calls the XT_Server to urge the SL_Client to send a keepalive-message to
350  * the SLServer if possible.
351  * @return true if keepalive could be sent, false otherwise
352  */
353 public boolean isSLCLIConnected(){
354    try{
355       send(Messages.SLCLI_MSG_CONN_TIMEOUT);
356       String received = receive();
357       return received.trim().equals(Messages.DONE);
358    }
359    catch(XTServerException xtsExcept){
360      return false;
361    }
362 }
```

Listing 15: Methode isSLCLIConnected der Klasse XTServer

Der Versand der RT-Änderungsnachrichten wird durch die Funktion `xt_sl_notify_peer` aus Listing 16 realisiert. Dabei wird der Inhalt des Sende-Puffers durch die Funktion `xt_sl_send` an den `SLServer` übertagen. Im Erfolgsfall wird nach dem Senden auf die Bestätigung des `SLServers` über den Empfang und die Bereitschaft weitere Nachrichten empfangen zu können in Form einer „SLCLI_MSG_ACK„-Nachricht, gewartet.

```
139 /*
140  * sends the send_buffer's content through the established tcp-connection
141  * @return 0 the send_buffer is successfully send, -1 otherwise
142  */
143 int xt_sl_notify_peer(){
144     /*if the connection is not established return without sending*/
145         if (slc_state->connected < 0){
146                 zlog_warn("SL_CLI: not connected!");
147                 return -1;
148         }
149     /*check if there is a message in the send_buffer*/
150         if (slc_state->message_available >= 0){
151                 if (xt_sl_send() < 0){
152                         zlog_err("SL_CLI: Send operation to %s failed: %s",
153                                         slc_state->server_Address,
154                                         safe_strerror(errno));
155             return -1;
156                 }
157                 //wait for acknowledgement
158                 if(xt_sl_receive() < 0){
159                         zlog_err("SL_CLI: No ACK received, %s",
160                         safe_strerror(errno));
161                         slc_state->connected = -1;
162                         return -1;
163
164                 }
165             if(strcmp(slc_state->recv_buffer,
166                             (char *) lookup(rip_slc_msg, RIP_SLC_MSG_ACK)) < 0)
167                 zlog_warn("SL_CLI: unrecognized message received, ACK expected.");
168         }
169         return 0;
170 }
```

Listing 16: Die Funktion xt_sl_notify_peer der sl_client.c

4.3 Die Datenorganisation

In den vorangegangenen Kapiteln wurde mehrfach erwähnt, dass jeder `SL_Client` bei jeglicher Veränderung der Routing-Tabelle des jeweiligen RIP-Routers diese Änderung auch an den `SLServer` schickt. Diese Tatsache führt zu der Überlegung, wie sich diese Daten am günstigsten verwalten lassen, damit diese auch sofort einer Analyse unterworfen werden können. Dazu wird die Objektorientiertheit der Programmiersprache JAVA und die Eigenschaft der Eindeutigkeit programmiertechnischer Objekte[10] ausgenutzt. Das `java.util`-Package stellt eine Reihe von Collections zur Verfügung, die es ermöglichen, eine Kombination aus Datenstrukturen so zu implementieren, dass die vorliegende Netztopologie logisch nachgebildet werden kann. Dieser Abschnitt beschreibt das Konzept und dient als Grundlage für die folgenden Kapitel.

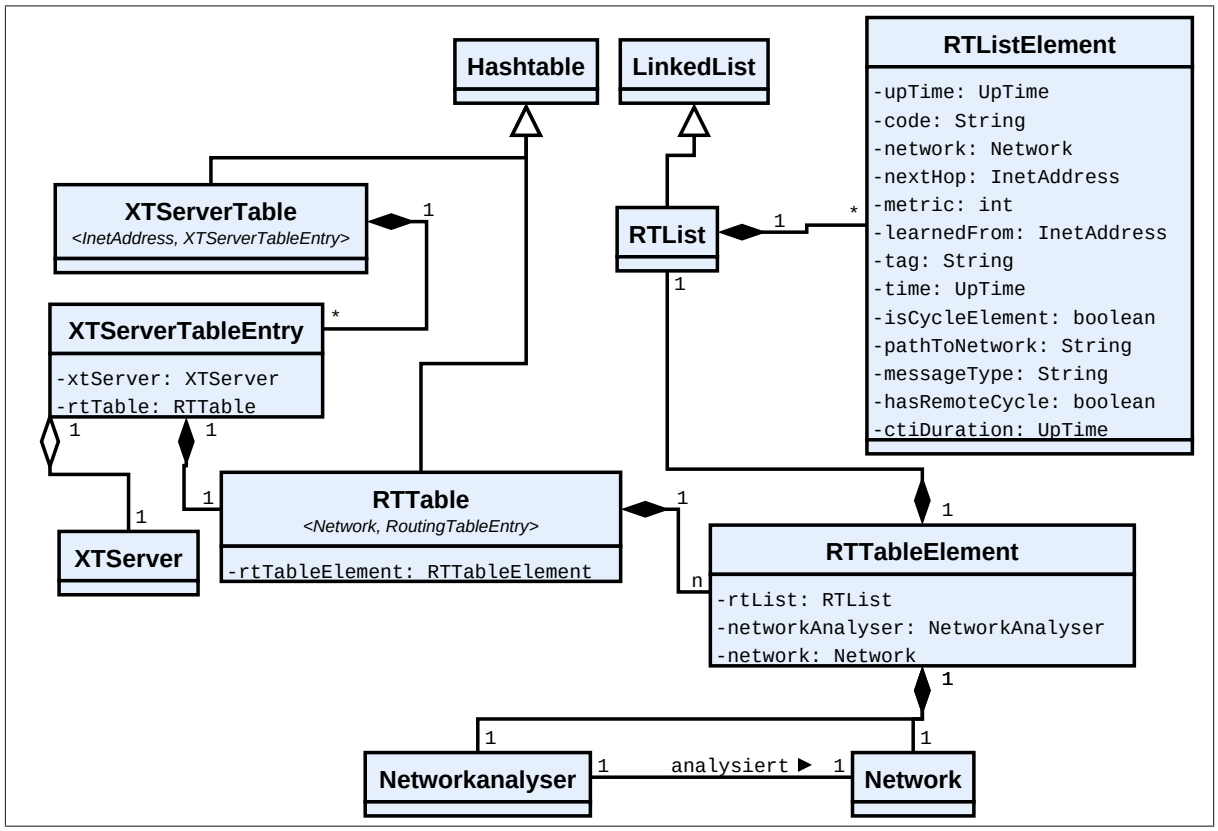


Abbildung 13: Klassendiagramm der Datenorganisation

In Abbildung 13 ist die Umsetzung der Datenstruktur als Klassendiagramm dargestellt. Der Übersichtlichkeit halber wurden einige Klassen im Diagramm nur als Attribute der Klassen

[10] Die Identität und die damit einhergehende Eindeutigkeit programmiertechnischer Objekte ist durch die eindeutige Adresse im Speicher gegeben, an der das Objekt im Speicher abgelegt ist.

angegeben, mit denen sie in direkter Beziehung stehen und auf die explizite Angabe dieser Klassen im Diagramm verzichtet.

Ausgegangen wird von der `XTServerTable`, einer Spezialisierung der `Hashtable`. Diese `XTServerTable` enthält als Einträge `XTServerTableEntries`, die wiederum aus einer `RTTable` und einem `XTServer` bestehen. Ein `XTServer` stellt dabei ein logisches Äquivalent zu einem VNUML RIP-Router dar.
Als Schlüssel für den Zugriff auf die Elemente der `XTServerTable` dient die Menge der Interface-Adressen des jeweiligen `XTServer`, sodass man unter Verwendung aller Interface-Adressen eines RIP-Routers auf ein und den selben Eintrag der `XTServerTable` gelangt und somit alle Daten zu diesem RIP-Router erreichen kann.

In der `RTTable` werden für einen `XTServer`, also dem entsprechenden RIP-Router auf VNUML-Seite, Informationen in Form eines `RTTableElements` verwaltet, wobei zu jedem Netzwerk, das der korrespondierende RIP-Router kennt, dies ist durch die Multiplizität von n in Abbildung 13 gekennzeichnet, ein solches `RTTableElement` existiert. Die `RTTableElemente` zu den betreffenden Netzwerken können über die Netzwerk-Adresse in Form von IP-Adresse/Präfixlänge des jeweiligen Netzwerkes als Schlüssel in der `RTTable` angesprochen werden.

Zur Speicherung der Pfadinformationen wird eine Spezialisierung einer `LinkedList` verwendet. Diese `RTList` ist Bestandteil des `RTTableElements`. Weitere Elemente, die in einem `RTTableElement` vereinigt werden, ist ein Objekt der Klasse `Network`, das das Netzwerk, auf das sich die Pfad-Informationen beziehen repräsentiert und ein Objekt der Klasse `Networkanalyser`, das für eben dieses Netzwerk die Erreichbarkeitsinformationen analysiert, wie es im Abschnitt 4.4 beschrieben wird.

Die Elemente einer `RTList` sind Objekte der Klasse `RTListElement`. Dies sind Objekte, die einen Eintrag in der Routing-Tabelle auf Routerseite repräsentieren. Dabei stellen die Attribute dieses Objektes, neben einigen Zusatzinformationen, genau die Informationen eines Routing-Tabellen-Eintrags dar.

Pfadinformationen:

- **code**: Die Art und Weise, wie dieser Eintrag generiert wurde.
- **network**: Das Zielnetzwerk, für das dieser Eintrag gilt.
- **nextHop**: Der nächste Hop, über den ein Paket zum Zielnetzwerk geschickt wird.
- **metric**: Die Distanz zum Zielnetzwerk.
- **learnedFrom**: Von welchem Nachbarrouter wurde diese Information mitgeteilt.
- **time**: Wert des Timers, der für diesen Eintrag gerade läuft.

Zusatzinformationen:

- **uptime**: Der Zeitstempel, zu dem diese Änderung in der Routing-Tabelle vorgenommen wurde.
- **isCycleElement**: Gibt an, ob dieses Update in Zusammenhang mit anderen Updates in anderen RIP-Routern eine Schleife verursacht.
- **hasRemoteCycle**: Gibt an, ob dieses Update durch eine Schleife erzeugt wurde, wobei der Router selbst nicht Element dieser Schleife ist.
- **pathToNetwork**: Gibt den Pfad wieder, über den dieses Update gelernt wurde.
- **messageType**: Gibt den Typ der RT-Änderungsnachricht an, die durch dieses Element repräsentiert wird.[11]
- **ctiDurationTime**: Gibt die Zeitspanne an, die ein CTI bis zum Eintreffen dieses Updates vorlag.

Diese Zusatzinformationen werden für Analysezwecke verwendet und werden erst nach Eintreffen der Information durch den `MessageProcessor` und den `Networkanalyser` berechnet.

Der beschriebene Aufbau der Datenorganisation und die Informationen, die in den `RTListElement`-Objekten vorliegen, können nun verwendet werden, um den Ursprung dieser Information zu ermitteln und etwaige Existenzen von Schleifen zu bestimmen. Dazu können durch die Attribute eines `RTListElement`-Objektes wiederum die `RTListElement`-Objekte ermittelt werden, die die Erreichbarkeitsinformation zu einem bestimmten Netzwerk eines anderen Routers repräsentieren. Diese Pfadanalyse wird im Abschnitt 4.4.1 eingehend beschrieben.

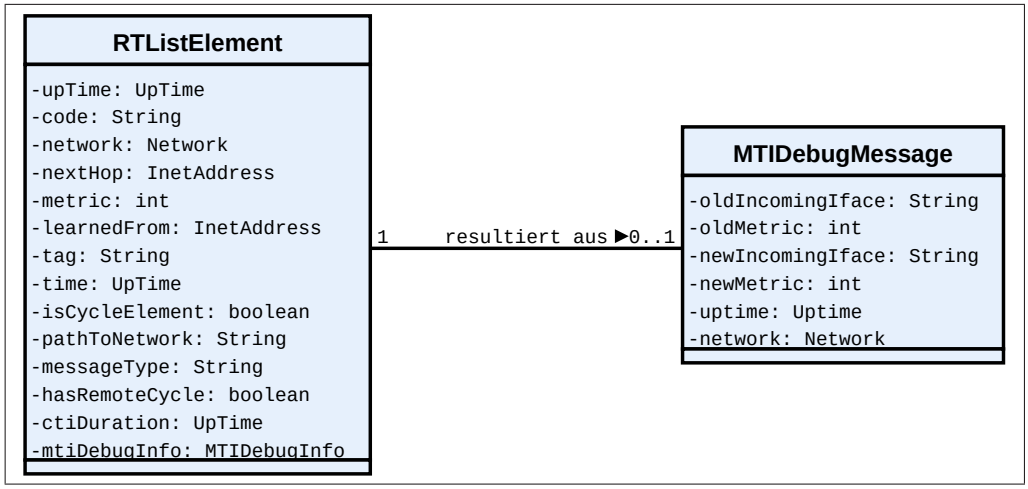


Abbildung 14: Beziehung RTListElement und MTIDebugInfo

[11] Die Nachrichtentypen werden in Abschnitt 4.2.4 beschrieben.

Im Zusammenhang mit der Betrachtung vom RIP mit der MTI-Erweiterung besteht eine weitere Beziehung, die im UML-Diagramm in Abbildung 13 nicht berücksichtigt wurde. Eine RT-Änderungsnachricht kann in Beziehung mit einem MTI-Check stehen. Wie in Abschnitt 4.1.3 beschrieben wurde, wird jedes mal, wenn der MTI mittels der Funktion `rip_mti_routeok` aufgerufen wird, eine MTI_Debug_Message generiert und an den `SLServer` geschickt. Auf diese MTI_Debug_Message folgt dann die dazugehörige UPDATE_MSG aus der vom `SLServer` ein `RTListElement` erzeugt wird. Die zuvor empfangene MTI_Debug_Message wird in einer Hashtable zwischengespeichert, bis das passende `RTListELement` vorliegt. Das `RTListElement` bekommt dann die jeweilige MTI_Debug_Message als Attribut übergeben. Die Zugehörigkeit einer MTI_Debug_Message zu einem RTListElement wird anhand des Netzwerkes, auf das sich diese beiden Informationen beziehen, geprüft.
Ob eine RT-Änderung aus einem MTI-Check hervorgeht ist abhängig von der in [Koc05] beschriebenen Implementierung des MTI im Quagga-RIP. Damit die Funktion `rip_mti_routeok` aufgerufen wird, muss die Bedingung, dass das Update vom selben Router stammt, wie das vorige und dass sich die Metriken der Updates unterscheiden, wie in Abschnitt 4.1.3 erläutert, nicht erfüllt sein. Erst unter dieser Voraussetzung wird aufgrund der *short-circuit-evaluation* die Funktion rip_mti_routeok aufgerufen. In Abbildung 14 ist die Beziehung zwischen `RTListElement` und `MTI_Debug_Message` zu sehen. Ein `RTListElement` besitzt also eine `MTI_Debug_Message`, falls diese vorhanden ist.

4.4 Der Networkanalyser

Die Klasse `Networkanalyser` bildet das Herzstück für die Analyse von Pfadinformationen und damit für die Erkennung von CTIs. Mit jedem Eintreffen eines Updates, bzw. einer RT-Änderungsnachricht, wird der `Networkanalyser` durch den `MessageProcessor` aus Abschnitt 4.2.1 aufgerufen und damit beauftragt, die Pfadinformationen, die zu diesem Zeitpunk in allen relevanten Routern vorliegen, auf die Existenz von Schleifen zu untersuchen. Ebenso wird die Bestimmung des Andauerns von CTIs festgestellt. Der `Networkanalyser` führt für jede Simulation Informationen darüber, wieviele CTIs in einer Simulation aufgetreten sind und berechnet die mittlere Dauer aller aufgetretenen CTIs.

In den folgenden Abschnitten wird die Funktionsweise zur Schleifenermittlung (Abschnitt 4.4.1) und Zeitbestimmung (Abschnitt 4.5.2) beschrieben.
Aus dem UML-Klassendiagramm in Abbildung 13 geht hervor, dass für jedes Netzwerk für einen Router jeweils ein `Networkanalyser`-Objekt die Analyse der Pfadinformationen vornimmt. Der `Netzwerkanalyser` ist dabei so konzipiert, dass er für das ihm zugeordnete Netzwerk ermittelt, wie es um die Pfadentwicklung anhand der Erreichbarkeitinformationen steht. Die Erreichbarkeitsinformationen für ein Netzwerk und somit auch der `Netzwerkanalyser` befinden sich dabei in einem Zustandsraum, dessen Zustandsübergangsgraph in Abbildung 15 dargestellt ist.

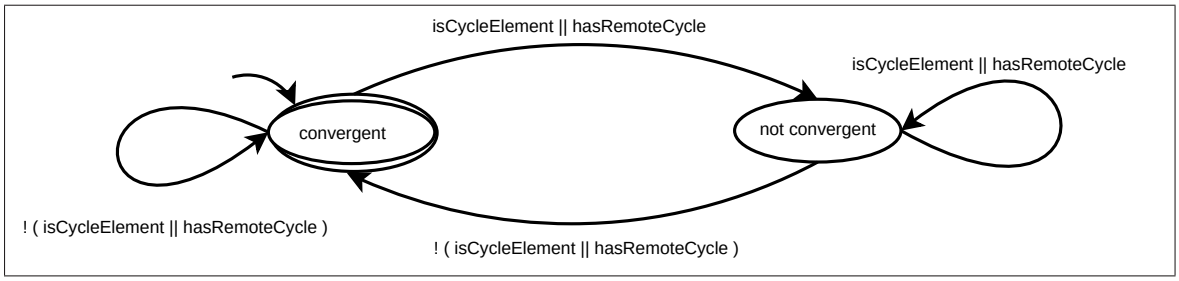


Abbildung 15: Statechart des Networkanalysers

Nach der Erzeugung eines `Networkanalyser`-Objektes beim Eintreffen der ersten Erreichbarkeitsinformation für das korrespondierende Netzwerk befindet sich der `Networkanalyser` im Zustand *convergent*. D.h. bis zu diesem Moment wird angenommen, dass die Informationen, die das Update mit sich bringt, konsistent sind und das Netzwerk tatsächlich mittels dieser Erreichbarkeitsinformationen zu erreichen ist. Der Zustand wechselt in *not convergent*, wenn sich durch die im Abschnitt 4.4.1 besprochene Pfadanalyse herausstellt, dass der Pfad, der durch dieses Update und die Erreichbarkeitsinformationen aller anderen Router zu diesem Zeitpunk für dieses Netzwerk eine Schleife bilden. Eine Schleife bildet sich, wenn entweder der Router selbst Bestandteil dieser Schleife ist oder sich ein anderer Router auf dem Weg durch das Netz in einer Schleife befindet. Im zweiten Fall handelt es sich um einen im folgenden genannten *remote Cycle*. Der `Networkanalyser` befindet sich im Zustand *convergent*, wenn der Pfad sowohl keinen Zyklus als auch keinen *remote Cycle* enthält. Die Attribute *isCycleElement* und *hasRemoteCycle* sind dabei Eigenschaften der betrachteten Erreichbar-

keitsinformation. Der Aufbau einer solchen Erreichbarkeitsinformation, dem `RTListElement`, wird ausführlich in Abschnitt 4.3 beschrieben.

4.4.1 Pfadanalyse und CTI-Erkennung

Die Pfadanalyse gehört zu den elementaren Funktionen, um die Existenz eines CTIs zu ermitteln. Das Konzept baut dabei auf der Tatsache auf, dass ein CTI nur dann entstehen kann, wenn ein Update irgendwann wieder bei seinem Ursprung angelangt. Mit anderen Worten ist die Voraussetzung für das Auftreten eines CTIs die Existenz einer Schleife in der Topologie, was auch durch [Sch99] und [Koc05] bestätigt wird.

Aufbauend auf der in Kapitel 4.3 beschriebenen Konzeption der Datenhaltung lässt sich nun die Überlegung anstellen, ob sich die gespeicherten Pfadinformationen und der Zugriff auf die `XTServerTable` durch die Interface-Adressen der einzelnen Router dazu verwenden lässt, den Pfad, den ein Update genommen hat, zu ermitteln.
In Abbildung 16 ist die Überlegung vereinfacht dargestellt.
Die Abbildung stellt die in Kapitel 4.3 beschriebene Datenstruktur noch einmal vereinfacht dar, indem hier nur die `XTServerTable`, `RTTable` und `RTList` sowie die `RTListElemente` in direktem Zusammenhang betrachtet werden. Die kapselnden Objekte wie `XTServerTableEntry` und `RTTableElement` werden der Übersichtlichkeit halber nicht berücksichtigt, was sich auf die abstrakte Betrachtung der Pfadanalyse nicht auswirkt.
Die in Abschnitt 4.1.1 besprochene Ordnung der Abfolge, in der die Übertragung der RT-Änderungsnachricht an den `SLServer` und die Triggerung der Updates an die Nachbarrouter erfolgt, kommt hier zum Tragen. Wird eine RT-Änderungsnachricht empfangen und analysiert, muss sichergestellt sein, dass die entsprechende RT-Änderungsnachricht des triggernden RIP-Routers bereits zentral im `SLServer` vorliegt und zur korrekten Analyse herangezogen werden kann.
In Abbildung 16 und 17 ist somit folgendes dargestellt: Die `XTServerTable` enthält für jeden RIP-Router eine `RTTable`. Diese enthält ihrerseits Erreichbarkeitsinformationen zu jedem Netzwerk, das ein Router kennt. Diese Erreichbarkeitsinformationen werden in Form von `RTListElementen` in eine `RTList` abgelegt. Die `RTList` zu einem Netzwerk kann unter Verwendung der Adresse eines Netzwerken in der Form IP-Adresse/Präfixlänge als Schlüssel in `der RTTable` angesprochen werden. Das letzte Element einer `RTList` stellt dabei die Routinginformation dar, über die der korrespondierende Router aktuell verfügt und in seiner Routing-Tabelle eingetragen ist.

Die Zeitpunkte t_1, t_2 und t_3 stellen die Zeitpunkte dar, zu denen die durch die `RTListElemente` dargestellten Updates in der Routing-Tabelle der RIPRouter R_1, R_2 bzw. R_3 vorgenommen wurden. Dabei gilt in diesem Beispiel $t_2 < t_1 < t_3$. D. h. das Update auf Router R_2 wurde zu einem früheren Zeitpunkt vorgenommen als das Update zum Zeitpunkt t_1 auf Router R_1 und das zum Zeitpunkt t_3 insgesamt letzte Update wurde auf Router R_3 vorgenommen.
Der Zeitpunkt t_3 dient in diesem Beispiel als Referenzpunkt zur Pfadanalyse. Ausgehend von diesem Zeitpunkt erfolgt eine Analyse des Pfades, die aufbauend auf den zugrunde

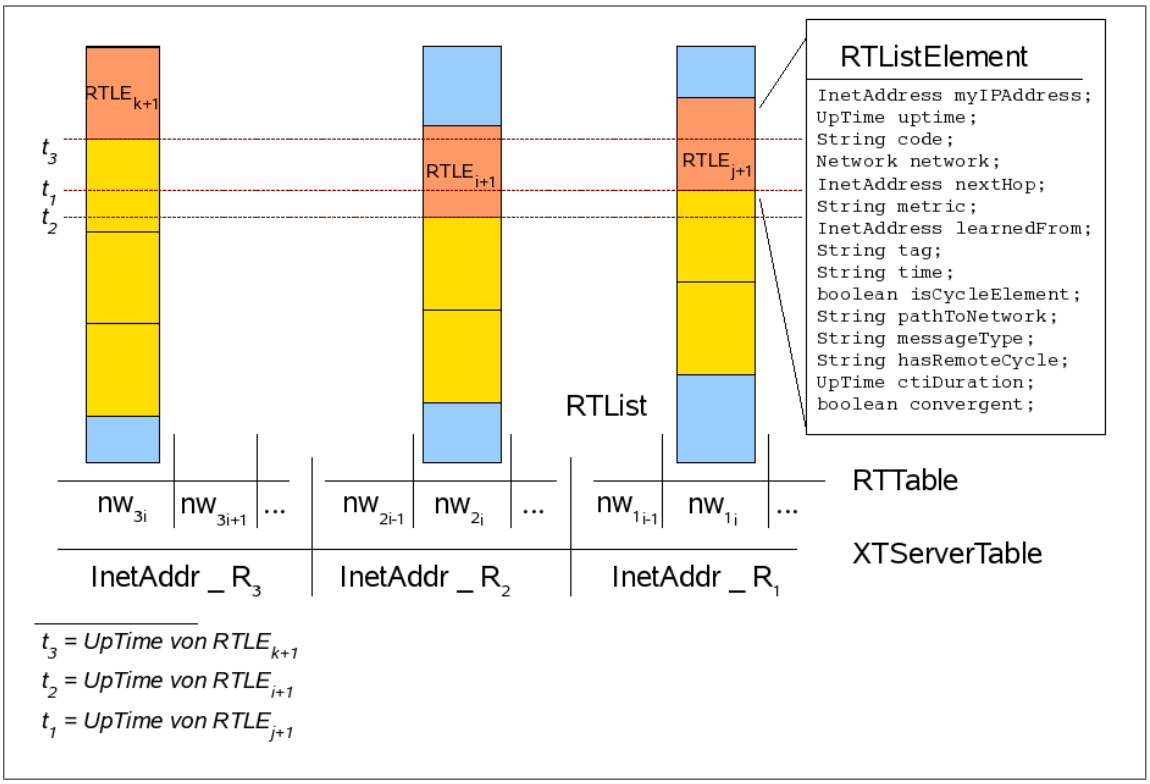


Abbildung 16: Pfadanalyse

liegenden Informationen, die die Router R_1, R_2 und R_3 bis zum Zeitpunkt t_3 besitzen, ermittelt, ob durch diese Informationen eine Schleife erzeugt wird. Dabei wird, ähnlich wie beim Routingprozess, den ein Paket auf dem Weg bis zum Zielnetz durchläuft, in den einzelnen Routern überprüft, über welchen *nextHop* das Paket bis zu einem bestimmten Ziel gelangt. Allerdings wird hier die *learnedFrom*-Adresse betrachtet, da es zu ermitteln gilt, welchen Weg das Update genommen hat. In Abbildung 17 ist der Sachverhalt dargestellt, in dem man wieder beim Ausgangsobjekt, d.h. der `RTList` von der gestartet wurde, angelangt, wenn man den Pfad, der durch das Wissen der Router R_3, R_2 und R_1 zum Zeitpunkt t_3 entsteht, verfolgt. In diesem Beispiel wird von den Informationen, die R_3 zur Verfügung stehen der Pfad über R_2 genommen. Der besitzt bis zu diesem Zeitpunkt die Informationen, die er zum Zeitpunkt t_2 in der Routingtabelle enthielt. Anhand dieser Informationen geht es dann weiter über R_1. Von dort aus wieder mit den Informationen, die er bis zum Zeitpunkt t_3 besitzt, nämlich die, die zum Zeitpunk t_1 in die Routingtabelle dieses Router eingetragen wurden.

In diesem Fall gelangt man durch den Eintrag *learnedFrom* im `RTListElement` wieder zum Ausgangsobjekt, also der `RTList` bei der die Analyse gestartet wurde. Diese Situation ist in Abbildung 17 dargestellt.
Diese Analyse wird für jede, beim `SLServer` eingehende Information durchgeführt.

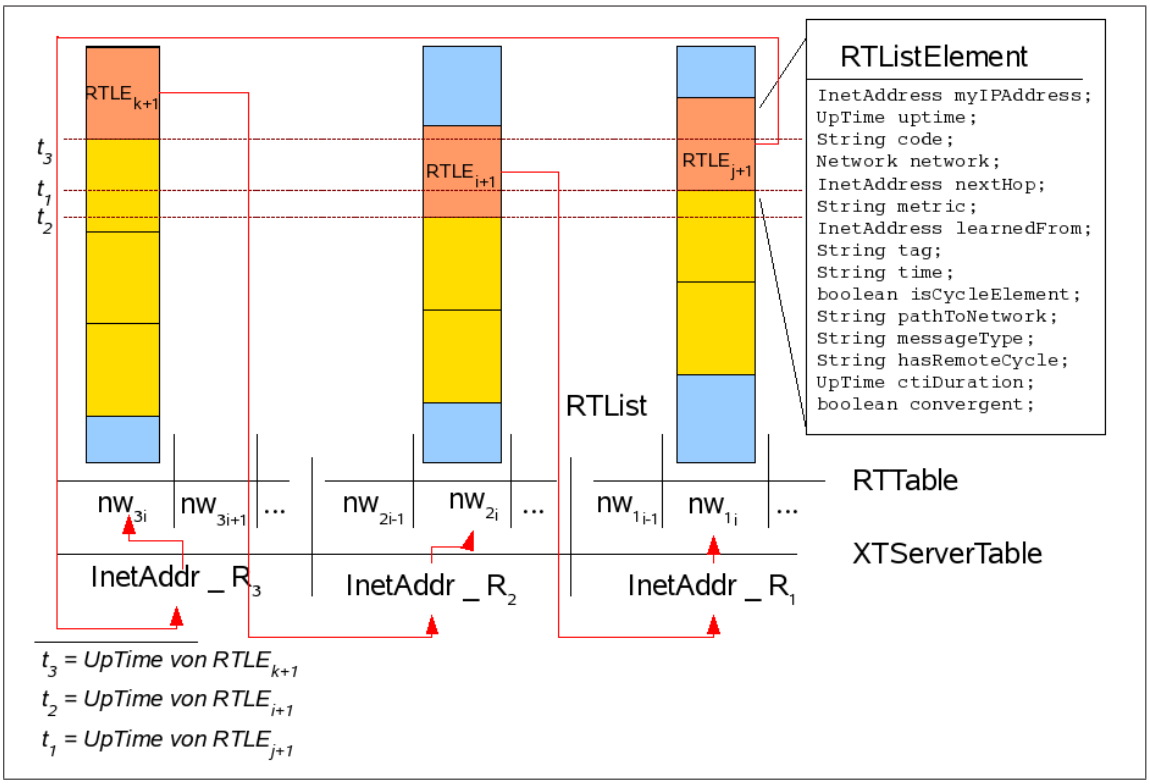


Abbildung 17: Pfadanalyse mit Schleifenfindung

Durch Abbildung 18 soll der beschriebene Vorgang der Pfadanalyse noch einmal verdeutlicht werden. Dabei beschreibt der Zeitpunk t den spätesten Referenzzeitpunkt.

Die Kernmethoden, die diesen Ansatz implementieren, werden im folgenden beschrieben.
In Listing 17 ist der Algorithmus dargestellt, durch den die Pfadanalyse durchgeführt wird. Diese Methode bekommt als Argument das `RTListElement`, von dem ausgehend die Analyse startet. In der Zeile 50 wird der Startknoten bestimmt. Dazu wird der `XTServer` als Abbild des RIP-Routers herangezogen. Die globale Variable *pathToNetwork* dient dazu, den Weg, den das Update genommen hat, textuell zu repräsentieren. Der *pathVector* aus Zeile 54 wird verwendet, um festzustellen, ob auf dem Weg durch das Netz ein entfernter Zyklus -*remote cycle*- vorliegt. In der `do-while`-Schleife erfolgt dann die „Durchwanderung“ des Netzes zurück zum Ursprung des Updates, das das `RTListELement` repräsentiert.
Zunächst wird dabei und in jedem weiteren Durchlauf durch die Methode `destinationArrived`, die in Listing 18 aufgezeigt ist, geprüft, ob das Zielnetz erreicht wurde. Ein Zielnetz ist genau dann erreicht, wenn die *learnedFrom*-Adresse gerade die Loopback-Adresse[12] des betrachteten XTServers ist.

[12]Ist das Zielnetz direkt mit einem RIP-Router verbunden, wird dies durch den learnedFrom-Wert *self* in der Routing-Tabelle ausgezeichnet. Ein RTListElement verwendet zu diesem Zweck die Loopback-Adresse 127.0.0.1.

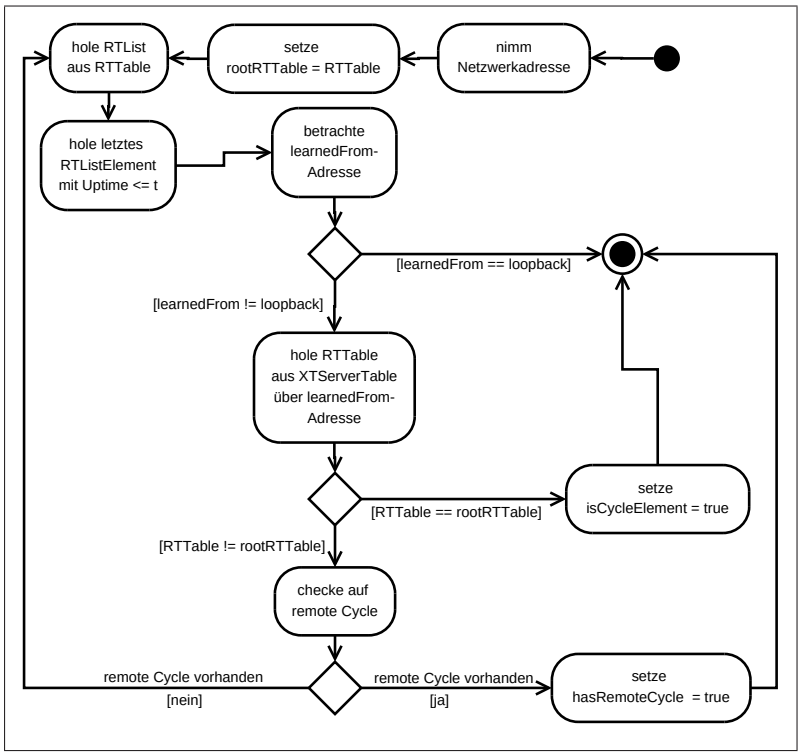


Abbildung 18: Flussdiagramm der Pfadanalyse

```
39 /**
40  * This method determines if the given RTListelement, containing the latest
41  * rip-update, results in a cycle.
42  *
43  * @param rtListElement
44  * @return true if the given RTListelement contains a rip-update which causes
45  *  a cycle, else false
46  */
47 public boolean hasNetworkCycle (RTListElement rtListElement){
48    RTListElement thisServersElement = rtListElement;
49    RTListElement otherServersElement = thisServersElement;
50    XTServer rootNode = (XTServer)xtServerTable
51                   .get(thisServersElement.getMyIPAddress()).getXtServer();
52    XTServer curNode = (XTServer)xtServerTable
53                   .get(otherServersElement.getMyIPAddress()).getXtServer();
54    pathToNetwork = rootNode.getName();
55    this.pathVector = new Vector<XTServer>();
56    boolean isCycle = false;
57
58    do{
59       if(destinationArrived(otherServersElement, network)){
60          isCycle = false;
61          break;
62       }
63       else{
64          otherServersElement = getNextProperElement(otherServersElement,
65                rootNode, thisServersElement.getUptime(), network);
66          curNode = (XTServer)xtServerTable.get(otherServersElement
67             .getMyIPAddress()).getXtServer();
68
69          /* There is a loop, if the current XTServerTableEntry equals the one
70           * we started from.
71           */
72          isCycle = curNode.equals(rootNode);
73          this.pathVector.add(curNode);
74          pathToNetwork += "->"+curNode.getName();
75       }
76
77    }while(!curNode.equals(rootNode)&&!hasRemoteCycles());
78
79    if(isCycle||hasRemoteCycles()){
80       thisServersElement.setIsCycleElement(true);
81       setConvergence(false);
82    }
83    else{
84       setConvergence(true);
85    }
86    setMetricDiff(thisServersElement);
87    return isCycle;
88 }
```

Listing 17: hasNetworkCycle der Kalsse Networkanalyser

In diesem Fall liegt keine Schleife vor und die `do-while`-Schleife wird abgebrochen.
Im anderen Fall wird anhand des Attributs *learnedFrom* im aktuellen `RTListElement` der passende Eintrag aus der `XTServerTable` geholt. Dies wird durch die Methode `getNextProperElement` aus Listing 19 bewerkstelligt. Dabei gilt als „nächtes passendes Element“ ein `RTListElement`, das bist zum Zeitpunkt t auf dem zur `curNode` korrespondierenden RIPRouter generiert wurde. t bezeichnet dabei den Zeitpunk, zu dem das `RTListElement` erzeugt wurde, das als Argument an diese Funktion übergeben wurde.

```
91 /**
92  * this method checks if the destination network is arrived.
93  * @param rtListElement the element to check for
94  * @param network the destination network
95  * @return
96  */
97 private boolean destinationArrived(RTListElement rtListElement, Network network){
98    String networkString = network.getNetworkAsString();
99    if (rtListElement.getLearnedFrom().isLoopbackAddress()){
100      /*
101       * If the update corresponding to this RTListElement is learnd by
102       * the rip-router itself, the network is directly connected to this
103       * rip-route. So the destination is arrived and there is no loop.
104       */
105      return true;
106   }
107   else if (xtServerTable.get(rtListElement.getLearnedFrom())
108            .getRtTable().get(networkString) == null
109            ||xtServerTable.get(rtListElement.getLearnedFrom())
110            .getRtTable().get(networkString).getRtList().isEmpty()
111            ||
112            xtServerTable.get(rtListElement.getLearnedFrom())
113            .getRtTable().get(networkString).getRtList().size() < 1){
114      /*
115       * If a RTList of a rip-router has no entry of a network, there is
116       * also no loop. This case will not happen in a normal run, only if
117       * an update does not arrive the XT_Peer.
118       */
119      pathToNetwork = "null:␣"+pathToNetwork+"␣->"+
120      xtServerTable.get(rtListElement.getLearnedFrom()).getXtServer()
121      .getName();
122      return true;
123   }
124
125   return false;
126 }
```

Listing 18: destinationArrived der Kalsse Networkanalyser

Eine Referenz des dazugehörigen `XT_Servers` wird in der Variablen *curNode* gespeichert. Es liegt bei einer Konstellation genau dann eine Schleife vor, wenn man wieder am Ausgangspunkt angekommen ist. Das ist der Fall, wenn das `XT_Server`-Objekt, bei dem die Analyse begonnen hat, noch einmal erreicht wurde, also genau dann, wenn $curNode == rootNode$.
An die *pathToNetwork*-Variablen wird in jedem Schleifendurchlauf der Name des besuchten `XT_Server`-Objektes angehängt, um den Weg nachvollziehen zu können. Die `do-while`-Schleife wird unterbrochen, wenn entweder das Ausgangsobjekt erreicht oder ein entfernter Zyklus, ein „*remoteCycle*“vorliegt. Die Existenz eines *remoteCycles* wird durch die Methode `hasRemoteCycle` aus Listing 20 überprüft. Die Methode überprüft dabei, ob in der globalen Variable *pathVector* zwei identische `XT_Server`-Objekte enthalten sind. In diesem Fall ist die *rootNode* indirekt von diesem entfernten Zyklus betroffen.
Wurde die `do-while`-Schleife beendet, wird im Falle der Existenz eines Zyklus oder eines entfernten Zyklus das Attribut *isCycleElement* des untersuchten `RTListElements` auf *true* gesetzt. Im Falle eines entfernten Zyklus wird zudem noch das Attribut *hasRemoteCycle* auf *true* gesetzt, um zu markieren, dass dieses Element selbst kein Bestandteil einer Schleife, sondern indirekt von einer entfernten Schleife betroffen ist.

```
130 /**
131  * This Method looks for the next proper RTListElement according to the given
132  * UpTime.
133  * @param rtListElement the starting element
134  * @param rootNode
135  * @param uptime
136  * @param network
137  * @return RTListElement the found RTListElement
138  */
139 private RTListElement getNextProperElement(RTListElement rtListElement,
140                                            XTServer rootNode,
141                                            UpTime uptime, Network network){
142     String networkString = network.getNetworkAsString();
143     RTListElement anchor = xtServerTable.get(rtListElement
144           .getLearnedFrom()).getRtTable().get(networkString).getRtList()
145           .getFirst();
146
147     rtListElement = xtServerTable.get(rtListElement.getLearnedFrom())
148                     .getRtTable().get(networkString).getRtList().getLast();
149      XTServer curNode = (XTServer)xtServerTable.get(rtListElement
150           .getMyIPAddress()).getXtServer();
151
152     RTListElement tmpLE = rtListElement;
153     /*
154      * If another rip-router has a later Update, we have to look for a proper
155      * one, because we need the knowledge of all routers involved since the
156      * time the starting element has been generated.
157      */
158     while(uptime.getTime() <= tmpLE.getUptime().getTime()
159           && !tmpLE.equals(anchor) && !curNode.equals(rootNode)){
160           tmpLE = tmpLE.getPrev();
161     }
162     return tmpLE;
163 }
```

Listing 19: getNextProperElement der Kalsse Networkanalyser

Anhand der Pfadbestimmung lässt sich somit nach jedem Eintreffen neuer Pfadinformationen bestimmen, ob die Voraussetzung für die Entstehung eines CTI gegeben ist. Als Voraussetzung wird die Existenz einer durch ein bestimmtes Update hervorgerufenen Schleife betrachtet.

```
130 /**
131  * Checks if there exists a remote cycle in this path
132  * @return true if ther exists a remote cycle for this path, false otherwise
133  */
134 public boolean hasRemoteCycles(){
135     XTServer xtServer;
136     if(this.pathVector.isEmpty()){
137        /* if the pathvector is empty there a loop can not exist.*/
138        return false;
139     }
140     try{
141        Vector pVector = (Vector)this.pathVector.clone();
142
143        for (int i = 0; i <= this.pathVector.size()-1; i++){
144           //look at the i-th object
145           xtServer = (XTServer)pVector.get(i);
146           // remove it from the vector-clone
```

```
147             pVector.remove(i);
148             /* check if the pathvector contains this object twice
149             * if it is so ther must be a loop*/
150             if(pVector.contains(xtServer)){
151                 return true;
152             }
153             // get a new copy of the pathVector to assure consistency
154             pVector = (Vector)this.pathVector.clone();
155         }
156     }
157     catch(Exception e){
158         System.err.println(e);
159     }
160     return false;
161 }
```

Listing 20: hasRemoteCycles der Kalsse Networkanalyser

4.5 Zeiten und Zeitbestimmung

In den folgenden zwei Abschnitten wird beschrieben, welche Überlegungen angestellt wurden, um den kritischen Aspekt der Zeit in geeigneter und zuverlässiger Weise so zu berücksichtigen, dass man die Zeit als Anhaltspunkt zur Analyse des Konvergenzverhaltens, wie sie in Abschnitt 4.4.1 beschrieben wurde, heranziehen kann. Die Zeit wird deswegen als kritischer Aspekt betrachtet, da sich die Analyse ganzheitlich auf die Zeitpunkte stützt, zu denen die einzelnen Router Erreichbarkeitsinformationen besitzen. Aus diesem Grund muss die Betrachtung der Zeit sorgfältig überlegt sein, damit keine verfälschten Ergebnisse und somit fehlerhafte Beobachtungen zu Stande kommen.
Auch die Zeiträume des Andauerns von CTIs müssen betrachtet werden, da die Existenz eines CTIs durch die Anfangs- und Endpunkte seines Auftretens von anderen CTIs abgegrenzt wird. Die Zeitnahme und statistische Gegenüberstellung zeitlicher Aspekte von Simulationen wird ebenfalls durch die Ermittlung der Zeiten bzw. Zeitdauern von CTIs ermöglicht.

4.5.1 Zeitbestimmung der Updateerzeugung

Zur Betrachtung der Zeit, zu der ein Router Änderungen an seiner Routing-Tabelle vornimmt, ist es notwendig, diese Zeit auch auf Routerseite zu bestimmen. Eine Bestimmung auf Hostseite, etwa der Zeitpunkt des Eintreffens einer Änderungsmeldung vom Router ist nicht verwertbar. Der Grund dafür ist die Anzahl der Threads und damit die Nebenläufigkeit und die damit einhergehende, nicht abschätzbare Zeitverzögerung bei der Ausführung der `SLServerThreads`. Um aber bestimmen zu können, ob eine Erreichbarkeitsinformation einen CTI auslösen kann, ist eine unbedingte Ordnung auf der Abfolge der Entstehung dieser Änderungen unabdingbar. Die Bestimmung des Änderungszeitpunktes muss daher so zeitnah wie möglich zur Änderung und unabhängig vom Scheduling der Threads festgehalten werden.
In Abschnitt 4.4.1 wurde vorgestellt, wie durch den Networkanalyser festgestellt wird, ob ein Update einen Zyklus durchwandert hat. Diese Prüfung erfolgte unter Berücksichtigung des Wissens, das die involvierten Router bis zum Zeitpunkt der RT-Änderung besitzen, die ebenfalls die Analyse durch den Netzwerkanalyser auslöst. Der Zeitpunkt, zu dem die RT-Änderung vorgenommen wurde, wird im `RTListElment` in dem Attribut upTime festgehalten und stellt eine relative Zeit zum Start einer Simulation dar. Da diese Zeitpunkte wegen der oben genannten Gründe auf Seiten der Router bestimmt werden und somit auch auf Seiten einer VNUML-Maschine, bleibt zu klären, ob man davon ausgehen kann, dass die Uhren der virtuellen Maschinen synchron zur Uhr des Hostsystems und somit auch untereinander synchron laufen.
Dazu ist es nötig, die Architektur von User Mode Linux, kurz UML, zu betrachten. User Mode Linux ist so konzipiert, dass eine UML-Maschine als Prozess auf dem Hostsystem ausgeführt wird. Dabei besitzt eine UML-Maschine keinerlei direkten Zugriff auf die Hardware des Hostsystems, stattdessen kommuniziert ein UML-Kernel mit dem Kernel des Hostsystems [Tea06]. Ebenso steht es mit dem Zugriff auf die hardware-clock. Nach [gen07] bedient sich jede UML-Maschine der Zeiteinstellung des Hostsystems und synchronisiert sich auch mit die-

sem. Dies macht den Einsatz von Technologien wie NTP[13] in diesem Umfeld nicht notwendig.

Die Zeitbestimmung selbst geschieht so nah wie möglich zu der Änderung der Routing-Tabelle des betreffenden Routers. Direkt nachdem die Änderung vorgenommen wurde, wird die Funktion `xt_sl_get_rte` wie in Abschnitt 4.1.2 beschrieben aufgerufen. Die erste Aktion, die in dieser Funktion durchgeführt wird, ist der Aufruf der Funktion `xt_sl_get_sim_time` (vergl. Abschn. 4.1.2 Listing 2 Zeile 4189) durch die der Zeitstempel generiert wird. Die Funktion selbst ist in Listing 21 gezeigt.

```
194 /*
195  * generates the timestamp on that the rt has changed and the message was
196  * generated
197  */
198 long xt_sl_get_sim_time(){
199     gettimeofday(&slc_state->uptime_usec, NULL);
200     slc_state->sim_time_sec = slc_state->uptime_usec.tv_sec;
201     slc_state->uptime = gmtime(&slc_state->sim_time_sec);
202     sprintf(slc_state->tmp_buffer,"%d:%d:%d:%d!-!",slc_state->uptime->tm_hour,
203                                                 slc_state->uptime->tm_min,
204                                                 slc_state->uptime->tm_sec,
205                                                 slc_state->uptime_usec.tv_usec/1000);
206     size_t str_len = RIP_SLC_BUFSIZE - strlen(slc_state->tmp_buffer)+1;
207     strncat(slc_state->send_buffer, slc_state->tmp_buffer, str_len);
208     return slc_state->sim_time_sec;
209 }
```

Listing 21: xt_sl_get_sim_time der SL_Client.c

Die Zeit wird durch die Verwendung der `sys/time.h` ermittelt. Diese Header-Datei definiert eine `timeval-Structure`, die u.a. die Zeiteinheit Mikrosekunden zur Verfügung stellt. Um eine möglichst genaue Zeitbestimmung zu ermöglichen, reicht es aber, aufgrund der Eigenschaft von „triggered Updates“[14] und der automatisierten Triggerung aus [Keu07], die sich mindestens im Zehntelsekunden-Bereich bewegt, eine Betrachtung auf die Millisekunden-Ebene zu beschränken. Deshalb wird der gelieferte Wert in Mikrosekunden Listing 21 in Zeile 205 durch 1000 dividiert. Die ermittelte Zeit wird im Format hh:mm:ss:msmsms (vergl. Abschnitt 4.1.2) in den Sendepuffer geschrieben und mit den Daten der RT-Änderung an den `SLServer` gesendet.

4.5.2 Der CTI-Timer

Der CTI-Timer beschreibt einen Mechanismus, der zur Bestimmung von Anfangs- und Endpunkten der Dauer eines vorliegenden CTI eingesetzt wird, um somit Beginn und Ende eines CTI-Auftretens zu ermitteln und um somit einzelne CTI-Auftreten voneinander abzugrenzen. Die Bestimmung der Zeit des Vorliegens eines CTI soll ebenfalls für die Ermittlung statistischer Mittelwerte verwendet werden. Das Andauern eines CTI wird dabei

[13] NTP = Network Time Protocol. NTP ist ein Standard zur Synchronisierung von Uhren in Computersystemen über paketbasierte Kommunikationsnetze.[wik]

[14] triggered Updates erfahren, wie bereits erwähnt eine zufällige Verzögerung zwischen 1 und 5 Sekunden, um die Netzlast bei häufigen Änderungen gering zu halten.[Mal98]

pro Router für das jeweilige Netz bestimmt.
Trifft ein Update für einen bestimmten Router im `SLServer` ein, bearbeitet der korrespondierende `SLServerThread` diese Update-Informationen mit Hilfe des in Abschnitt 4.4 vorgestellten `Networkanalysers`. Ermittelt der `Networkanalyser` einen Zyklus, der durch diese Update-Informationen und dem Wissen aller beteiligten Router gebildet wird, besteht die Gefahr, dass ein CTI entsteht.
In diesem Fall wird der CTI-Timer gestartet und stoppt die Zeit, die der CTI besteht.
Die Methode `checkTimerStart` aus Listing 22 bekommt als Argument das aktuelle `RTListElement` übergeben und prüft anhand dieses Elements, des zuvor ermittelten Zustands des Netzes (vergl. Abschnitt 4.4) und des CTI-Timers, ob der Timer gestartet werden muss.

```
439 /**
440  * Checks if the CTIDurationTimer has to be started and starts the Timer.
441  *
442  * @param rtListElement
443  */
444 public void checkTimerStart(RTListElement rtListElement){
445     if(!isConvergent() && ctiUptime==CTI_TIMER_STOPPED
446             && rtListElement.getMetricAsInt() != RIP_METRIC_INFINITY){
447         ctiUptime = new UpTime(rtListElement.getUptime().getTime());
448     }
449 }
```

Listing 22: checkTimerStart der Kalsse Networkanalyser

Der Timer wird also genau dann gestartet, wenn der Timer nicht bereits gestartet ist und das Netzwerk, für das die Analyse durchgeführt wird nicht konvergent ist (d.h. dass zuvor ermittelt wurde, dass das `RTListeElement` einen Zyklus verursacht) und die Metrik dieses Updates nicht dem Metrikwert RIP_METRIC_INFINITY entspricht. Sind diese drei Bedingungen erfüllt, wird ein neues `UpTime`-Objekt erzeugt. Der Wert für dieses neue Objekt ist dabei gerade der Zeitpunkt, zu dem das `RTListElement` auf RIP-Routerseite erzeugt wurde, also genau der Zeitpunkt, zu dem eine Änderung an der Routing-Tabelle des korrespondierenden Routers vorgenommen wurde.

Im Gegensatz zum Starten des CTI-Timers gestaltet sich das Stoppen etwas komplizierter.
Ob der CTI-Timer gestoppt werden muss, wird immer dann geprüft, wenn der CTI-Timer läuft, das Zielnetz wieder erreichtbar ist oder ausgeschlossen werden kann, dass weitere Updates auf ein Update, das die Metrik RIP_METRIC_INFINITY bekannt gibt, hervorgerufen durch den CTI, zu erwarten sind. Letzteres wird durch die Methode `moreTicksPossible` aus Listing 24 bestimmt.

```
453 /**
454  * Checks if the CTIDurationTimer has to be stopped and stops the Timer
455  * according to the given rtListElement.
456  *
457  * @param rtListElement
458  */
459 public void checkTimerStop(RTListElement rtListElement){
```

```
460     if(ctiUptime != CTI_TIMER_STOPPED
461        && (isConvergent() || !moreTicksPossible(rtListElement))){
462        if(isConvergent() && rtListElement.getPrev()
463                                        .getMetricAsInt() == RIP_METRIC_INFINITY){
464           correctTimingOverhead(rtListElement);
465           ctiUptime = CTI_TIMER_STOPPED;
466        }
467        else if(isConvergent() && rtListElement.getPrev().getMetricAsInt()
468                                                          != RIP_METRIC_INFINITY){
469           ctiTimes.add(rtListElement.getCtiDuration());
470           ctiUptime = CTI_TIMER_STOPPED;
471           }
472           else{
473              rtListElement.setCtiDuration(getCTIDurationTime(rtListElement));
474              ctiTimes.add(rtListElement.getCtiDuration());
475              ctiUptime = CTI_TIMER_STOPPED;
476              setConvergence(true);
477           }
478     }
479 }
```

Listing 23: checkTimerStop der Kalsse Networkanalyser

Zunächst wird überprüft, ob das Netzwerk konvergent ist oder ob auf dieses `RTListElement` keine weiteren `RTListElemente` folgen können, die ebenfalls noch Elemente des vorliegenden CTI sind und ob der CTI-Timer überhaupt läuft (vergl. Listing 23 Zeile 460-461). Ist diese Bedingung erfüllt, sind drei Fälle zu unterscheiden.

1. Das Netzwerk ist konvergent und das Vorgängerelement besitzt die Metrik RIP_METRIC_INFINITY (dies tritt dann auf, wenn korrekte und falsche Updates alternieren). In diesem Fall muss durch die Methode `correctTimingOverhead` geprüft werden, ob dieses `RTListElement` das letzte nach dem bisher vorherrschenden CTI ist und ggf. das letzte nach dem CTI suchen und den Timer stoppen. Zudem fügt, die Methode `correctTimingOverhead` die ermittelte Zeit des CTI in den CTI-Times-Vektor ein.

2. In diesem Fall ist das Netz konvergent und das Vorgängerelement des aktuellen `RTListElements` besitzt eine Metrik ungleich RIP_METRIC_INFINITY. Dann muss an dieser Stelle nur noch der CTI-Timer gestoppt und diese Zeit in den CTI-Times-Vektor gespeichert werden.

3. In diesem Fall ist das Netzwerk nicht konvergent. Es gilt aber nach der Bedingung aus Zeile 560 aus Listing 23, dass weitere `RTListElemente` in diesem CTI-Vorkommen durch die Bedingung `!moreTicksPossible()` ausgeschlossen werden. In diesem Fall muss der Zustand des `Networkanalysers` auf `convergent` gesetzt, der CTI-Timer gestoppt und die CTI-Dauer in den CTI-Times-Vektor hinzugefügt werden.

Im CTI-Times-Vektor werden alle ermittelten Zeitdauern von aufgetretenen CTIs gespeichert. Dieser Vektor wird zur Ermittlung der durchschnittlichen CTI-Zeiten verwendet.

```
512 /**
513   * Checks if some more CTI-Updates will follow to the given rtListElement.
```

```
514   * @param rtListElement The latest RTListElement.
515   * @return true if it is possible that some cti-updates will follow,
516   * false otherwise
517  */
518 private boolean moreTicksPossible(RTListElement rtListElement){
519   if(rtListElement.getMetricAsInt()== rip_metric_infinity && !isConvergent()){
520     RTListElement prevElement = rtListElement.getPrev();
521     /*  search the next previous element with a metric that does not equal
522     *  rip_metric_infinity*/
523     while(prevElement.getMetricAsInt() == rip_metric_infinity){
524        prevElement = prevElement.getPrev();
525     }
526     if (rtListElement.getMetricAsInt() - prevElement.getMetricAsInt()
527                                                          <= metricDiff){
528        return false;
529     }
530     else return true;
531   }
532   else return true;
533 }
```

Listing 24: Die Methode moreTicksPossible der Kalsse Networkanalyser

Die Methode `moreTicksPossible` aus Listing 24 vergleicht während der Existenz eines CTI die Abstände der Metriken der während des CTI erzeugten `RTListElemente`. Die Methode gibt *false* zurück, wenn gilt:

$$rtListElement_{cur}.getMetric() - rtListElement_{prev}.getMetric() \leq metricDiff \tag{1}$$

wobei stets gilt

$$\begin{aligned} & rtListElement_{cur}.getMetric() > rtListElement_{prev}.getMetric() \\ \wedge \quad & rtListElement_{cur}.getMetric() = RIP\_METRIC\_INFINITY \\ \wedge \quad & \neg convergent \end{aligned}$$

```
248 /**
249  * Caclulates and sets the difference between two metrics (global variable
250  * metricDiff) that are not rip_metric_infinity
251  *
252  * @param rtListElement the current RTListElement.
253  */
254 private void setMetricDiff(RTListElement rtListElement){
255   if(!isConvergent()){
256      if(rtListElement.getMetricAsInt()!=rip_metric_infinity ){
257        RTListElement prevElement = rtListElement.getPrev();
258        while(prevElement.getMetricAsInt() == rip_metric_infinity
259              && prevElement.getNext().isCycleElement()){
260           prevElement = prevElement.getPrev();
261        }
262        if(rtListElement.getMetricAsInt() > prevElement.getMetricAsInt())
263           metricDiff = rtListElement.getMetricAsInt() - prevElement.getMetricAsInt();
264      }
265     }
266   else{
267     metricDiff = 1;
```

```
268     }
269 }
```

Listing 25: setMetricDiff der Kalsse Networkanalyser

Nach Gleichung (1) wird jedes eintreffende Element bei nicht-Konvergenz des Netzwerks darauf überprüft, ob die Metrik den Wert RIP_METRIC_INFINITY besitzt. Ist dies der Fall, wird durch Ermitteln des Metrik-Abstandes zwischen diesem Element und dem Vorgängerelement bestimmt, ob weitere Updates, hervorgerufen durch den CTI möglich sind. Es wird dabei das Vorgängerelement gesucht, dessen Metrik einer Metrik kleiner RIP_METRIC_INIFINITY entspricht.

Der Wert für den Metrik-Abstand `metricDiff` wird, wie in Listing 25 gezeigt, innerhalb einer inkonvergenten Phase immer dann ermittelt, wenn ein RTListElement, das aus der aktuellen RT-Änderungsnachricht hervorging, einen Metrik-Wert kleiner RIP_METRIC_INFINITY besitzt. Dazu wird das nächste Vorgängerelement gesucht, das einen Metrik-Wert kleiner dem des aktuellen besitzt. Wird kein solches Vorgängerelement gefunden, wird der alte, metricDiff-Wert beibehalten. Innerhalb einer konvergenten Phase wurde ein metricDiff-Wert von 1 gewählt.

Der Grundgedanke bei dieser Bestimmung, ob weitere Updates, die durch einen CTI hervorgerufen werden, in einem Router eintreffen können besteht durch folgende Tatsache: Metriken, die durch einen CTI in genau *einem* Zyklus entstehen und einen Wert ungleich RIP_METRIC_INFINITY besitzen, wachsen stetig mit gleich bleibendem Abstand.

So gilt bei einem eintreffenden Update mit der Metrik RIP_METRIC_INFINITY:
Sei m_{rmi} die Metrik des zuletzt empfangenen Updates mit der Metrik RIP_METRIC_INFINITY und m_i die Metrik des Updates davor, die kleiner ist als m_{rmi}, dann gilt:

1. m_{rmi}–$m_i \leq metricDiff$ gdw. es können keine Updates, die durch diesen CTI hervorgerufen werden, in dem Router eintreffen, für den diese Berechnung gilt.
2. m_{rmi}–$m_i > metricDiff$ gdw. es können weitere Updates, die durch diesen CTI hervorgerufen werden, in dem Router eintreffen, für den diese Berechnung gilt.

Dabei errechnet sich $metricDiff$ wie folgt:
Sei m_{i-1-j} die Metrik des ersten Updates, das vor dem Update mit Metrik m_i eintraf, und eine Metrik kleiner m_i besitzt. Dann ergibt sich:

$$metricDiff = m_i - m_{i-1-j}, \qquad mit \quad 0 < j \leq i-1$$

Dabei kann m_{i-1-j} eine Metrik sein, die aus der konvergenten Phase stammt.

Betrachtet man sich die Entwicklung des Metrik-Abstandes unter der zusätzlichen Einschränkung, dass m_{i-1-j} ebenfalls aus einem Update hervorging, dass innerhalb der

inkonvergenten Phase empfangen wurde, dann ergibt sich folgende Beobachtung:

Seien m_i und m_{i-1-j} die Metriken, die beim ersten Eintreffen eines Updates zur Berechnung von $metricDiff_1$ herangezogen werden. Dabei wird der Wert für $metricDiff$ durch das Eintreffen von weiteren k-1 Updates mit Metrikwerten kleiner RIP_METRIC_INFINITY insgesamt k mal berechnet und es ergibt sich:

$metricDiff_k = metricDiff_{k-1} = metricDiff_{k-2} = ... = metricDiff_1 = m_i – m_{i-1-j}$.

Eine Unterbrechung dieses gleichbleibenden Abstands kann nur durch Updates mit kleineren Metriken erfolgen. Dazu sind zwei Fälle zu unterscheiden:

1. Das Zielnetz ist wieder erreichbar mit einer Metrik, die kleiner der Metrik ist, die durch Updates aus dem CTI einem Router mitgeteilt wurden.
2. Es trifft ein Update ein, das eine geringere Metrik besitzt und aus einem CTI eines kleineren Zyklus[15] hervorging (vergl. Abb. 19 Zyklus a).

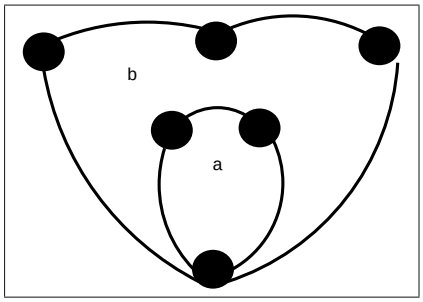


Abbildung 19: verschachtelte Schleifen

Im ersten Fall ist das Zielnetz wieder erreichbar und der Zustand des Networkanalysers ist convergent, woraus sich ergibt, dass der CTI beendet ist.

Im zweiten Fall wird der zuvor aus m_k und m_{k-1} ermittelte $metricDiff_k$-Wert beibehalten. Als Voraussetung gilt:

$RIP\_METRIC\_INFINITY – m_k > m_k – m_{k-1}$,

d.h. es können weitere Updates innerhalb der inkonvergenten Phase eintreffen.
Sei weiter m_{k+1} die Metrik des Updates, das nach dem Update mit Metrik m_k empfangen wurde, wobei gilt $m_{k+1} < m_k$, dann errechnet sich der $metricDiff_k$-Wert aus $m_k - m_{k-1}$ wie oben. Trifft nun ein Update ein, das die Metrik m_{rmi} mit dem Wert RIP_METRIC_INFINITY besitzt, so ergibt sich

$m_{rmi} – m_{k+1} > metricDiff_k$

[15] Als Maß für die Größe eines Zyklus wird die Anzahl der Hops innerhalb des Zyklus herangezogen.

und es können weitere Updates erwartet werden. Danach wächst die Metrik wieder gleichbleibend und der Abstand wird wie oben berechnet.

Zwei Sonderfälle werden durch die Methode `moreTicksPossible` nicht abgedeckt:

1. Sollte es dazu kommen, dass der CTI-Timer nachträglich aufgrund mehrfachen Eintreffens von RT-Änderungsnachrichten mit einer Metrik von RIP_METRIC_INFINITY und einer sich danach ergebenden Konvergenz, obwohl noch weitere Update erwartet werden, die aus einem CTI hervorgehen[16], korrigiert werden muss, wird der CTI-Timer durch die vorgestellte Methode `correctTimingOverhead` nachträglich an der entsprechenden Stelle[17] beendet.
2. Sollte es dazu kommen, dass der CTI-Timer gestoppt wurde, und es trifft dennoch ein Update hervorgerufen durch einen vorherrschenden CTI ein, so wird dieser als zusätzlicher CTI erkannt.

Mit Hilfe dieses Timers werden die Anzahl sowie die Zeiträume für die auftretenden CTIs und die durchschnittliche Zeitdauer von CTIs bestimmt. Speziell wegen Sonderfall 2 ist es notwendig das Ergebnis einer Simulation und die berechneten Mittelwerte für die Zeiten von CTI-Erscheinungen durch den Benutzer zu prüfen.

[16] dies kann eintreten, wenn ein CTI vorzeitig beendet wird.
[17] Das erste Update, das die Metrik RIP_METRIC_INFINITY besitzt

4.6 Die erweiterte XT_Client-GUI

Neben dem Sammeln der Daten und dem Auswerten der Daten zur Bestimmung von CTI-Erscheinungen soll die Möglichkeit gegeben werden, die Daten auch während der Simulation in Echtzeit in Augenschein nehmen zu können. Um den Overhead an geöffneten Fenstern gering zu halten und dadurch eine angenehmes und zielgerichtetes Arbeiten zu ermöglichen, wurde in Betracht gezogen, die `XT_Client`-GUI, die als Frontend aus der Arbeit von [Pä06] hervorging, so zu erweitern, dass neben der ursprünglich gebotenen Anzeigen der Netztopologie auch die Daten, die durch die RIP-Routingprozesse hervorgehen, anzeigen und somit den Konvergenz-Prozess verfolgen zu können.
Die daraus hervorgegangene `XT_Peer`-GUI und die Interaktion derselben wird in Abschnitt 6.3 beschrieben. Dieser Abschnitt hier soll Überblick über die technische Umsetzung der graphischen Bestandteile verschaffen.

Um dem hohen Datenaufkommen Herr zu werden und um diese geeignet darstellen zu können, wurde die ursprüngliche `XT_Client`-GUI in zwei Abschnitte unterteilt. Im ersten Abschnitt ist wie bereits erwähnt der Teil der Netztopologie enthalten. Im zweiten Teil findet die grafische Repräsentation der Daten statt. Dieser Teil enthält dabei für jeden Router, der sich im Netz befindet und über die XT-Erweiterung verfügt, einen Karteireiter. Für jedes Netzwerk, das ein Router kennt, ist in dieser Karteikarte wiederum eine Karteireiter zu finden. Jede Netzwerkkarteikarte besteht dabei ihrerseits aus zwei Teilen. Diese Teile bilden zum einen die Netzwerktabelle, in der alle Erreichbarkeitsinformationen angezeigt werden, die in der `RTListe` durch den dazugehörenden `SLServerThread` gesammelt und verarbeitet werden. Zum Anderen enthält jede dieser Karteikarten eine CTI-Verlaufsgraph, der CTI-Erscheinungen in ihren zeitlichen Eigenschaften in Form einer Verlaufskurve in einem Koordinatensystem darstellt. Die Umsetzung dieser beiden Bestandteile wird in den folgenden beiden Abschnitten erläutert.

4.6.1 MyNetworkTable

Zur Repräsentation der Daten, die im Laufe einer Simulation anfallen, wurde die `JTable` aus dem Package `javax.swing` gewählt. Diese Tabelle soll dabei folgende Aufgaben erfüllen:

- Repräsentation wesentlicher Routing- und Zusatzinformationen, die in einer `RTListe` gespeichert werden.
- Markierung kritischer zeitlicher Zustände im Netzwerk
- Anzeige zusätzlicher Informationen in einem gesonderten Fenster bei Bedarf.

Die erste Funktion wird durch die Klasse `MyTableModel` erfüllt. Diese bildet das Modell der Tabelle, das die Spaltenwerte definiert und die Spalten mit den jeweiligen Daten füllt. Die Daten entsprechen der textuellen Darstellung der Attributwerte eines `RTListElements`. Der Aufbau eines `RTListElements` wurde in Abschnitt 13 vorgestellt. Ein Objekt der Klasse `MyTableModel` wird bei der Erzeugung einer `JTable` im Konstruktor als `defaulTableModel` angegeben.

Die Klasse `MyTableCellRenderer` definiert ihrerseits die Darstellung der einzelnen Zellen der Tabelle. Diese Klasse wird zur farblichen Gestaltung und zum Markieren eingesetzt, um kritische Abschnitte besser erkennbar zu machen. Ein Objekt dieser Klasse wird durch den Aufruf `setDefaulTableCellRenderer` der `JTable` bekannt gegeben.
Die Klasse `MyTableCellEditor` definiert das Verhalten einzelner Zellen einer Tabelle für das Auftreten von MouseEvents. Das Verhalten der Zellen wurde so implementiert, dass es zeilenspezifisch ist. D.h. das Verhalten jeder Zelle einer Zeile ist gleich. Die Reaktion auf einen MouseEvent ist dabei das Öffnen eines zusätzlichen Fensters, in dem Informationen über das Verhalten des MTI enthalten sind. Falls keine derartige Information vorliegt, ist der Fensterinhalt leer und das Fenster öffnet sich nicht. Zeilen, zu denen eine MTI-Information vorliegt, sind durch ein „(i)“ in der Spalte „cause“ gekennzeichnet.

4.6.2 Der Verlaufsgraph

Um einen schnellen Überblick über das Auftreten und das Andauern von CTIs in einer Simulation zu bekommen, wurde die `XT_Peer-GUI` um einen Verlaufs-Graphen erweitert. Dieser Verlaufsgraph erlaubt es, den zeitlichen Verlauf einer Simulation und den darin aufgetretenen CTI auch mehrerer Simulationen, deren Daten gespeichert und geladen wurden, graphisch gegenüber zustellen.

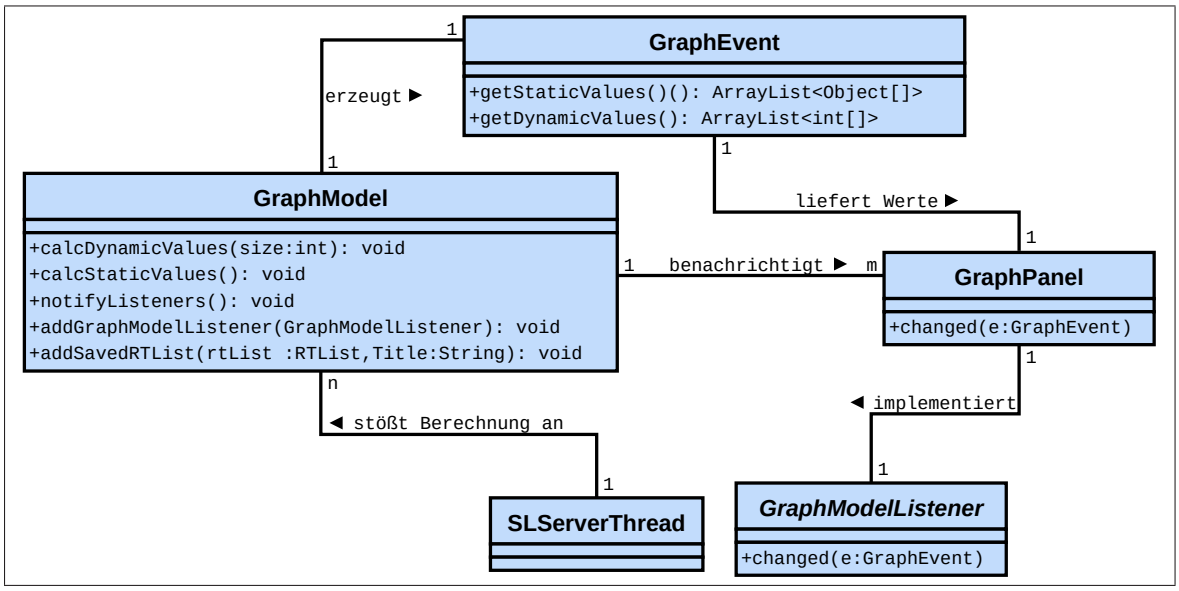


Abbildung 20: UML des Verlaufsgraphen

Der Verlaufsgraph wurde wie auch die Datentabelle nach dem Model-View-Controller Prinzip implementiert. In Abbildung 20 ist die Beziehung zwischen Datenberechnung und Datenrepräsentation dargestellt.
Die Berechnung der Daten, die für die Anzeige des Graphen auf dem Panel benötigt werden,

übernimmt das Model. Das Model ist eine Objekt der Klasse `GraphModel`. Dieses Objekt berechnet ein Array, vielmehr eine Array-List aus Werten, die für das Zeichnen des Verlaufsgraphen benötigt werden. Ebenso erzeugt das Model ein Even-Objekt der Klasse `GraphEvent`. Dieses Event-Objekt wird vom Model an das `GraphPanel` übergeben. Das Panel-Objekt entnimmt dem Event-Objekt die Array-Listen und zeichnet sie.
Bei der Erzeugung des Event-Objektes werden die errechneten Daten-Arrays im Konstruktor übergeben. Das Graph-Model berechnet dabei zwei Arten von Daten:

- dynamische Daten: Daten, die eine aktuell laufende Simulation erzeugt.
- statische Daten: Daten, die aus einer geladenen Simulation herrühren

Die dynamischen Daten werden der `RTListe` einer laufenden Simulation entnommen. Die `RTListe` dieser Simulation wird beim Erzeugen des `GraphModel`-Objekte im Konstruktor übergeben. Das Ermitteln statischer Daten ist optional und wird aus diesem Grund nicht im Konstruktor übergeben. Stattdessen können statische Daten aus geladenen Szenario-Daten mit Hilfe der Methode `addSavedRTList` hinzugefügt werden. Als Argumente für diese Methode wird zum einen die geladene `RTListe` und zum anderen der Name der Simulation, der in der SDF-Datei als Simulations-Titel angegeben ist, übergeben. Bis zu fünf Verlaufsgraphen geladener Szenariodaten können berücksichtigt werden, ohne dass die Übersichtlichkeit des Verlaufsgraphen darunter leidet. Bei mehr als fünf geladenen Szenarien wird die Anzeige durch den Versatz der einzelnen Graphen ungenau und aufgrund fehlender Farbabstufungen unübersichtlich.
Bei Änderung der dynamischen `RTListe` muss das `Graphmodel` die Daten für die Anzeige neu berechnen. Diese Neuberechnung geschieht durch den Aufruf der Methode `calcDynamicValues`. Da Änderungen an der `RTList` durch den entsprechenden `SLServerThread` vorgenommen werden, wird diese Methode für das jeweilige `GraphModel` dort aufgerufen, sobald alle notwendigen Daten berechnet wurden.
Nach dem Berechnen der Daten durch das `GraphModel` benachrichtigt dieses alle Listener, die bei diesem Model registriert sind, in diesem Fall das `GraphPanel`, und das Neuzeichnen des Graphen wird eingeleitet. Das `GraphModel` ist so konzipiert, dass sich mehrere Model-Listener registrieren können. Ein Model-Listener kann über die Methode `addGraphModelListener` am `GraphModel` registriert werden. Ein solcher Listener muss das Interface `GraphModelListener` implementieren, um sicherzustellen, dass die Methode `changed` verfügbar ist, über die die Änderung der Werte über eine `ModelEvent`-Objekt bekannt gegeben wird. Der `GraphModelListener`, in diesem Fall das GraphPanel, liest über das übergebene Event-Objekt die Werte für das Zeichnen der dynamischen und der statischen Verlaufsgraphen aus. Um eine gewisse Übersichtlichkeit zu gewähren, werden die einzelnen Graphen um einen Bildpunkt auf der Ordinate versetzt gezeichnet. Das Ende des dynamischen Graphen befindet sich dabei immer im Zentrum des sichtaberen Bildausschnittes. Die Skala der Ordinate wandert bei der Verschiebung des Bildauschnittes entlang der Abszisse und befindet sich stets links vom aktuellen Bildpunkt des dynamischen Verlaufsgraphen. Die Legende ist immer in der linken oberen Ecke des Bildausschnittes zu finden.

Die Werte der Abszisse sind die Zeiteinheiten in Sekunden der Simulationsdauer. Die Werte der Ordinate bildet die Zeitdauer eines CTI in Sekunden.

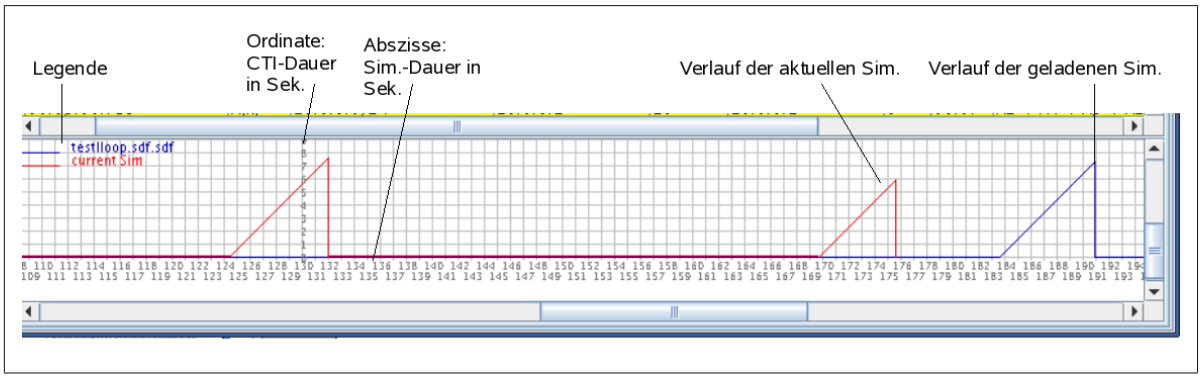


Abbildung 21: CTI-Verlaufsgraph der XT_Peer-GUI

4.7 Toolbox

Die Klasse `Toolbox` stellt eine Sammlung von Methoden zur Verfügung, die es ermöglichen, gesammelte Simulationsdaten zu speichern, zu laden und in einer Tabelle anzeigen zu lassen. Um eine strukturierte Speicherung dieser Daten zu ermöglichen, wurde zum einen die Möglichkeit der Speicherung in „Colon Separated Values„ für die Weiterverarbeitung in dritter Software und zum anderen eine strukturierte Speicherung in eine an XML angelehnte Strukturierung verwendet. Diese xml-ähnlichen Dateien werden mit der Erweiterung „.sdf“ angelegt, wobei sdf = „scenario data file “ bedeutet. In den folgenden Abschnitten wird der Aufbau der Dateien und die Vorgehensweise zur Speicherung erklärt.

4.7.1 Speichern von Simulationsdaten (scenario data files)

Um die strukturierte Speicherung zu gewährleisten, werden folgende Regeln verwendet, die nachfolgend in EBNF in Listing 4.7.1 beschrieben werden.

```
1  package sl_Extensions;
2
3  /**
4   * This interface is used by toolbox for distinct mapping of the tag-names
5   * an sdf-tag holds for saving and loading simulation-data.
6   * @author stefan
7   *
8   */
9  public interface SFDTag {
10
11     public static final String SAVEDSIMULATION = "savedSimulation";
12     public static final String SIMULATION = "simulation";
13     public static final String RIPROUTER = "riprouter";
14     public static final String NET = "net";
15     public static final String UPDATE = "update";
```

```
16     public static final String MTIDEBUGMSG = "mtiDebugMSG";
17     public static final String CTITIME = "ctiTime";
18 }
```

Listing 26: Das Interface SDFTag.java

FileContent	=	"<savedSimulation>" "\n"Body "</savedSimulation>";
Body	=	"<simulation name = ""SimName""">""\n"SimData "<simulation>" "\n";
SimData	=	{RouterData"\n"};
RouterData	=	"<riprouter name = " "RouterName""ipAddress =""IPAddress""">" "\n" NetworkData "</riprouter>" "\n";
NetworkData	=	"<net address = ""NetworkAddress""">""\n"{UpdateData} [Statistics] "</net>"\n";
UpdateData	=	"<update "UpdateAttributes "/>""\n"{MTIDebugMSG};
Statistics	=	"<ctiTime time = ""CTITime""/>""\n";
MTIDebugMSG	=	"<mtiDebugMSG "MTIDebugAttributes "/>"\n";
CTITime	=	Houres":"Minutes":"Seconds":"Milliseconds;
Houres	=	Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 2$ *) Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 4$ *);
Minutes	=	Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 5$ *) Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 9$ *);
Seconds	=	Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 5$ *) Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 9$ *);
Milliseconds	=	Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 9$ *) Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 9$ *) Digit(* with $n \in \mathbb{N}$, $0 \leq n \leq 9$ *);
IPAddress	=	IP-Adresse (* in "dotted decimal"-Notation *);
NetworkAddress	=	IP-Adresse (* in "dotted decimal"-Notation *)/ Number (* with $n \in \mathbb{N}$, $1 \leq n \leq 32$ *);

Listing 4.7.1: EBNF des Aufbaus einer SDF-Datei

Die Zuordnung der Update-Attributbezeichner zu den Werten, die durch die entsprechenden „getter" -Funktionen eines `RTListELement`-Objektes geliefert werden, wird in Tabelle 4 dargestellt. Der Aufbau der einzelnen Attributwerte wird durch die verwendeten Rückgabewerte definiert und werden hier nicht vertieft beschrieben. Zu diesem Zweck verweise ich auf die Quelltextdokumentation.
Analog dazu wird in Tabelle 3 die Zuordnung der Attributwerte eines MTIDebugMessage-Objektes aufgelistet.

Attributbezeichner	**Attributwert**
time	MTIDebugMessage.getTime()
network	MTIDebugMessage.getNetwork()
oldIncomingIface	MTIDebugMessage.getOldIncomingIface()
oldMetric	MTIDebugMessage.getOldMetric()
newIncomingIface	MTIDebugMessage.getNewIncomingIface()
newMetric	MTIDebugMessage.getNewMetric()

Tabelle 3: Zuordnung der Attributwerte eines MTIDebugMSG-Abschnittes

Attributbezeichner	Attributwert
relUpdatetime	rtListElement.getUpTime()
cause	rtListElement.getMessageType()
code	rtListElement.getCode()
network	rtListElement.getNetwork().getNetworkAsString()
nextHop	rtListElement.getNextHop()
metric	rtListElement.getMeric()
learnedFrom	rtListElement.getLearnedFrom()
tag	rtListElement.getTag()
pathToNetwork	rtListElement.getPathToNetwork()
cycleElement	rtListElement.getIsCycleElement().toString()
remoteCycle	rtListElement.getHasRemoteCycle().toString()
ctiDurationTime	rtListElement.getCTIDurationTime().getTime()
isConergent	rtListElement.isConvergent().toString()

Tabelle 4: Zuordnung der Attributwerte eines Update-Abschnittes

```
1 package sl_Extensions;
2
3 /**
4  * This interface is used by toolbox for distinct mapping of the tag-Attributes
5  * an update-tag holds for saving and loading simulation-data.
6  * @author stefan
7  *
8  */
9 public interface TAGAttribute {
10
11    public static final String NAME              = "name";
12    public static final String IPADDRESS         = "ipAddress";
13    public static final String CAUSE             = "cause";
14    public static final String RELUPTMIE         = "relUpdatetime";
15    public static final String CODE              = "code";
16    public static final String NETWORK           = "network";
17    public static final String NEXTHOP           = "nextHop";
18    public static final String METRIC            = "metric";
19    public static final String LEARNEDFROM       = "learnedFrom";
20    public static final String TAG               = "tag";
21    public static final String TIME              = "time";
22    public static final String PATHTONETWORK     = "pathToNetwork";
23    public static final String CYCLEELEMENT      = "cycleElement";
24    public static final String REMOTECYCLE       = "remoteCycle";
25    public static final String CTIDURATIONTIME   = "ctiDurationTime";
26    public static final String ISCONVERGENT      = "isConvergent";
27    public static final String OLDINCIMINGIFACE  = "oldIncomingIface";
28    public static final String OLDMETRIC         = "oldMetric";
29    public static final String NEWINCOMINGIFACE  = "newIncomingIface";
30    public static final String NEWMETRIC         = "newMetric";
31 }
```

Listing 27: Das Interface TAGAttribute.java

Obwohl eine SDF-Datei dem Aufbau einer XML-Datei gleicht, wurde die Verwendung einer Document-Type Definition vermieden. Stattdessen stellen zwei Interfaces den eindeutigen Aufbau einer SDF-Datei sicher. In Listing 26 werden die TAG-Bezeichner definiert, die für

die Erstellung und das Auslesen einer SDF-Datei verwendet werden. Die Attribut-Bezeichner eines Tags werden in Listing 27 definiert.
Bei der Speicherung an sich werden, ausgehend von der `XTServerTable`, alle verfügbaren Daten aller Router und dort wiederum aller Netze iteriert und ausgelesen. Dies erfolgt durch die Methode `generateSDFFile` der Klasse `Toolbox`.

4.7.2 Laden von Simulationsdaten

Das Laden der im sdf-Format gespeicherten Daten erfogt ebenfalls durch die Klasse `Toolbox`. Hierzu wird der `SAXParser` des Packages `javax.xml.parsers` verwendet. Der Parser wird in der Methode `readSavedSimulationFromSDF`, die in Listing 28 zu sehen ist, erzeugt.

```
234 /**
235  * This method initiates the sdf-parser to parse the file of the given
236  * filename
237  * @param sdfFileName the name of the file to parse
238  * @return XTServerTable the XTServerTable containing all loaded
239  * simulation-data
240  * @throws ParserConfigurationException
241  * @throws SAXException
242  * @throws IOException
243  */
244 public XTServerTable readSavedSimulationFromSDF(String sdfFileName)
245         throws ParserConfigurationException, SAXException, IOException
246 {
247   SAXParserFactory factory = SAXParserFactory.newInstance();
248   SAXParser parser = factory.newSAXParser();
249   SDFHandler sdfHandler = new SDFHandler();
250   parser.parse(new File(sdfFileName), sdfHandler);
251   this.simulationTitle = sdfHandler.getSimulationTitle();
252   return sdfHandler.getXTServerTable();
253 }
```

Listing 28: Die Methode readSavedSimulationFromSDF der Klasse Toolbox

In Zeile 248 in Listing 28 wird eine neue Instanz eines SAXParsers erstellt, der dann mit dem `DefaultHandler`, dem `SDFHandler` aufgerufen wird. Der `SDFHandler` erweitert den `DefaultHandler` aus dem Package `org.xml.sax.helpers` so, dass die in Abschnitt 4.7.1 vorgestellten Tags und den darin enthaltenen Informationen, die die Interfaces `SDFTag.java` und `TAGAttribute.java` definieren, verarbeitet werden können.

Beim Laden einer sdf-Datei wird eine eigene Datenstruktur erstellt, die der in Abschnitt 4.3 vorgestellten `XTServerTable` entspricht. Mit jedem Router-Eintrag in der sdf-Datei wird ein Eintrag in der neuen `XTServerTable` eingefügt, wobei das im `riprouter`-Tag vorhandene IPAddress-Attribut den Wert des Schlüssels zum Zugriff auf diesen Eintrag in der neuen `XTServerTable` liefert.
Für jedes `Net`-Tag wird ein Eintrag in der `RTTable` vorgenommen, wobei wiederum die Adress-Attributwerte die Schlüsselfunktion übernehmen. Für die zwischen den öffnenden und schließenden `Net`-Tags befindlichen Update-Informationen wird jeweils ein `RTListElement` erstellt, dass in die korrespondierenden `RTList` eingefügt wird. Die Attributwerte für

die angelegten `RTListElemente` liefern die Attribute der `Update`-Tags. Die MTI-Debug-Informationen liefert das `MTIDebugMessage`-Tag.

Informationen zum Konvergenzverhalten und zum CTI-Auftreten werden in einem für jedes Netzwerk erzeugten `NetworkAnalyser`-Objekt gespeichert, wobei eine Referenz auf dieses Objekt im jeweiligen `RTTableElement` abgelegt wird. Die Werte liefern die Attribute des `Statistics`-Tag.

Für jede geladene Simulation wird eine eigene Datenstruktur erzeugt. Je nach Datenaufkommen wird entsprechend viel Hauptspeicher benötigt, der zur Laufzeit durch die Software reserviert werden muss.
Durch die Option des Java-Interpreters

-Xms<size>
-Xmx<size>

kann beim Starten der Software entsprechend viel Hauptspeicher explizit zugewiesen werden. Die Option `-Xms<size>` legt die initiale Größe und `-Xmx<size>` die maximale Größe des Heap fest.

4.7.3 Speichern in CSV-Dateien

Das Speichern der gesammelten Daten in csv-Dateien wird durch eine Iteration über allen Elementen der `XTServerTable` und den darin enthaltenen Tabellen und Listen realisiert. Das Ansprechen der einzelnen Attributwerte wie im vorangegangenen Abschnitt ist nicht notwendig, da ein `RTLsitElement`-Objekt direkt eine Methode bereit stellt, die einen String mit allen Informationen liefert. Dieser String ist so aufgebaut, dass er direkt in eine CSV-Datei geschrieben werden kann. Durch Übergabe des Delimiters an diese Methode kann bestimmt werden, durch welches Trennzeichen die einzelnen Werte voneinander getrennt werden sollen. Der Header der CSV-Datei wird dabei als Konstante in der Klasse `Toolbox` definiert und beinhaltet die Einträge, die in Tabelle 5 aufgelistet werden.

Für jedes Netzwerk eines Router wird eine csv-Datei angelegt, die den Namen des Routers, gefolgt von der in CIDR-Notation gehaltenen Netzwerkadresse, wobei aus Kompatibilitätsgründen der Slash durch einen Bindestrich ersetzt wird, enthält. All diese Dateien werden in einem gesonderten Ordner abgelegt, der beim Speichern ebenfalls erstellt wird und dessen Name über den Speichern-Dialog festgelegt wird.

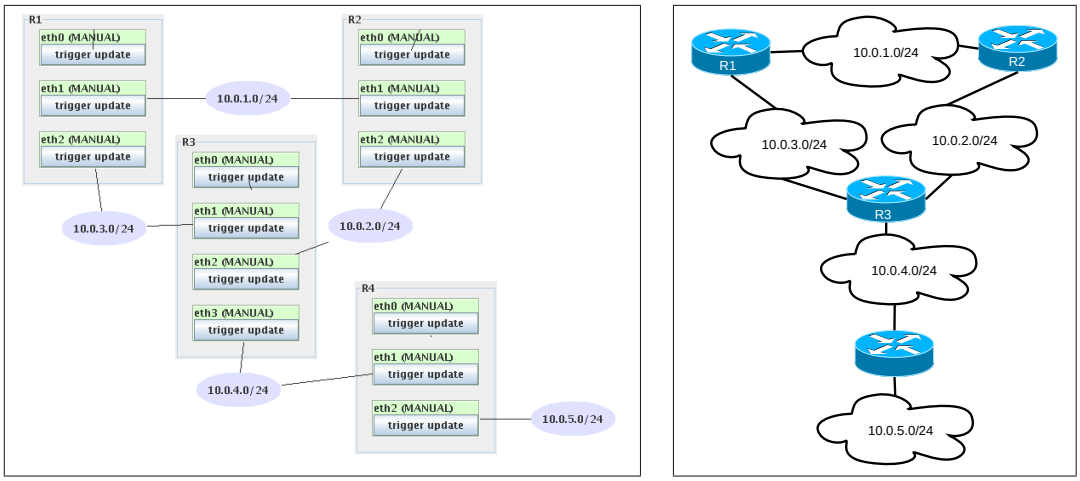
Spaltenbezeichner	**Informationsbeschreibung**
cause	Grund für die Auslösung dieser Nachricht
Simulation-Time	Zeitpunkt der Nachrichtengenerierung
Code	Code, wie diese Route gelernt wurde
Network	Zielnetzwerk
LearnedFrom	IP-Adresse des RIP-Routers von dem diese Route gelernt wurde
Metric	Metric für diese Route
NextHop	Nächster Hop, über den das Netzwerk erreicht werden kann
tag	Tag
Time	Timer für diese Route
Path to network	Pfad bis zum Zielnetzwerk
is cycle-element	Angabe, ob zu diesem Zeitpunkt ein routing-loop vorliegt
has remote-cycle	Angabe, ob ein entfernter routing-loop vorliegt
CTI-Duration Time	Zeitdauer für das Vorliegen eines CTI bis zu diesem Update
(MTI) old incoming Interface	MTI-Information über das alte Interface, über das eine Erreichbarkeitsinformation einging)
(MTI) old Metric	MTI-Information über die Metrik der alten Erreichbarkeitsinformation
(MTI) new incoming Interface	MTI-Information über das neu Interface, über das eine Erreichbarkeitsinformation einging)
(MTI) new Metric	MTI-Information über die Metrik der neuen Erreichbarkeitsinformation

Tabelle 5: Spalten einer CSV-Datei und Informationsgehalt

5 Der GraphObserver

Da diese Arbeit parallel zur Arbeit [Keu07], die sich mit der automatisierten Generierung von Testfällen beschäftigt, entstand, bot es sich an, die Unterstützung des Generierens neben der in Abschnitt 4.4.1 beschriebenen CTI-Erkennung insofern auszudehnen, als das eine Grundlage für das in [Keu07] beschriebenen „intelligenten Generierens“ geschafft wird. Der in diesem Abschnitt beschriebene `GraphObserver` bildet diese Grundlage, indem Objekte und Strukturen zur Verfügung gestellt werden, deren sich diese „intelligente Generierung“ bedient.

Ausgangspunkt für den `GraphObserver` ist die Annahme, dass durch die in [Pä06] geschaffte Abbildung der Szenario-Topologie eines VNUML-Szenarios aus um den `XT_Server` erweiterten RIP-Router die gesamte Topologie bekannt und somit auch sämtliche mögliche Pfade in alle Zielnetzwerke bestimmt werden können. In Abbildung 22(a) ist die graphische Repräsentation der y-Topologie aus Abbildung 22(b) durch die `XT_Client_GUI` dargestellt.



(a) (b)

Abbildung 22: Graphische Repräsentation der Y-Topologie

5.1 Nachbarschaftsverhältnisse

Zu dem o.g. beschriebenen Zweck müssen zunächst die Nachbarschaftsverhältnisse der einzelnen RIP-Router bzw. `XT_Server` ermittelt werden.
Die Klasse `XT_Server` des Packages `backend` wurde zu diesem Zweck um einen Vektor erweitert, der diese Nachbarschaftsverhältnisse beinhaltet. Ein solches Nachbarschaftsverhältnis wird durch ein Objekt `AdjacTupel` beschrieben. Dieses `AdjacTupel` beinhaltet dabei die folgenden Informationen:

- XTServer: Der direkte Nachbar zu dem Besitzer dieses `AdjacTupel`
- NetworkInterface: Das Netzwerkinterface, über das dieser Nachbar direkt zu erreichen ist.

Ein `AdjacTupel` bildet somit eine Relation zwischen Nachbarrouter und Netzwerkinterface eines `XT_Servers`.

In Listing 29 ist der Mechanismus zum Ermitteln der Nachbarschaft zu sehen. Die Methode iteriert über alle Interfaces, die ein `XT_Server` besitzt. Für jedes Interface wird dann das verbundene Netzwerk ermittelt. Ein `Network`-Objekt des Packages `backend` besitzt seinerseits wieder Networkinterfaces, für die der verbundenen `XT_Server` ermittelt wird. Es wird somit jeder `XT_Server` erfasst, der nicht dem suchenden `XT_Server` entspricht.

```
491 /**
492  * Determines the adjacencies of this XTServer (rip-router) and builds-up
493  * an AdjacTupel for each neighbor that is stored into this XTServers
494  * adjacencies-vector
495  * @see AdjacTupel
496  *
497  */
498 public void determineAdjacencies(){
499     if(adjacencies.isEmpty()){
500         NetworkInterface nif;
501         for(Iterator ifIt = this.getInterfaceIterator(); ifIt.hasNext();){
502             nif = (NetworkInterface)ifIt.next();
503             Network network = nif.getConnectedNetwork();
504             for(Iterator nwIfIt = network.getInterfaceIterator();nwIfIt.hasNext();){
505                 NetworkInterface networkInterface = (NetworkInterface)nwIfIt.next();
506                 XTServer xtServer = networkInterface.getServer();
507                 if(!xtServer.equals(this)){
508                     adjacencies.add(new AdjacTupel(nif, xtServer));
509 }}}}}
```

Listing 29: determineAdjacencies der Klasse XTServer

5.2 Pfadsuche

Die Pfadsuche besteht darin, aus dem gegebenen Graphen alle möglichen Pfade zu ermitteln. Es bietet sich demnach an, eine Tiefensuche durchzuführen. Der Algorithmus zur Bestimmung setzt sich aus drei Methoden zusammen. Die Methode `getPossiblePaths` aus Listing 30, die die Suche vorbereitet und aus der die 2 beteiligten Methoden `getPaths` aus Listing 31, die die eigentliche Tiefensuche implementiert und die Methode `getLastPathNodes` aus Listing 32, die die Pfade um die von der Tiefensuche ausgelassenen Endknoten ergänzt.
Die Methode `getPossiblePaths` bekommt als Argument einen Vektor übergeben, der alle `XT_Server` des VNUML-Szenarios enthält. Für alle `XT_Server` wird dann ermittelt, welche Pfade gestartet von diesem `XT_Server` möglich sind, indem das entsprechende `AdjacTupel` für jeden Nachbarn dieses `XT_Servers` als Startpunkt für die Tiefensuche verwendet wird. Der Vektor p stellt dabei einen Pfadvektor dar. Der Vektor *paths* ist ein Vektor aus Pfadvektoren, der alle ermittelten Pfade mit dem `XT_Server` als Wurzel beinhaltet. Die Tiefensuche wird durch die Methode aus Listing 31 `getPaths` realisiert, die ausgehend vom übergebenen `AdjacTupel` wiederum alle möglichen Pfade ermittelt und in den Vektor *paths* einhängt. Die Methode arbeitet dabei so, dass sie über alle Nachbarn des `XT_Servers` iteriert, der im übergebenen `AdjacTupel` angegeben ist und rekursiv alle darauffolgenden Knoten besucht. Dabei erkennt die Methode anhand des der *visitedState*-Variable eines `XT_Servers`,

ob dieser bereits besucht wurde. Wurde der `XT_Server` bereits besucht, wird nicht weiter abgestiegen und der Pfad ist vollständig.

```
447 /**
448  * Determines all possible paths through the net starting from
449  * each node in the given xtServerVector
450  * @param xtServerVector containing the nodes of the net
451  * @return java.util.Vector a vector of path-vectors. A pathvector
452  * consists of a Sequence AdjactTupels.
453  * @see AdjacTupel
454  */
455 public Vector<Vector<AdjacTupel>> getPossiblePaths(Vector xtServerVector){
456    Vector<Vector<AdjacTupel>> paths = new Vector<Vector<AdjacTupel>>();
457    for (Iterator iter = xtServerVector.iterator(); iter.hasNext();){
458       XTServer xtServer = (XTServer)iter.next();
459       /*Determine all possible paths starting from each XTServer*/
460       for(Iterator i = xtServer.getAdjacenciesIterator(); i.hasNext();){
461          /* set the visited state of the strating XTServer to true,
462           * that means that is the root-node*/
463          xtServer.setVisited(true);
464          AdjacTupel adt = (AdjacTupel)i.next();
465          Vector<AdjacTupel> p = new Vector<AdjacTupel>();
466          /* add this XTServert to an new Adjactupel with no interface
467           * to itself at the beginning of the pathvector p
468           */
469          p.add(new AdjacTupel(null, xtServer));
470          paths.add(p);
471          /* run the depth-first search-Method to build up the paths-Vector
472           * that contains all possible paths
473           */
474          getPath(adt, p, paths);
475          /* reset all states for the next iteration-step*/
476          initADT(xtServerVector);
477       }
478    }
479    /* at the end add all missing endnodes of a path*/
480    Vector<Vector<AdjacTupel>> toAdd =  getLastPathNode(paths);
481    paths.addAll(toAdd);
482    return paths;
483 }
```

Listing 30: getPossiblePaths der Klasse GraphObserver

Wird weiter abgestiegen, wird vom Pfad-Vektor *path* eine Kopie erstellt und im Rekursivaufruf übergeben. Der ursprüngliche Pfad-Vektor *path* wird in den Vektor *paths* als möglicher Pfad eingehängt. Wird die Methode verlassen, wird der *visitedState* dieses Knotens wieder auf *false* gesetzt, um zu signalisieren, dass dieser XTServer nicht mehr besucht ist.
Da die Methode `getPaths` nur bis zum letzten Knoten vor dem jeweiligen Startknoten rekursiv aufgerufen wird, ist es notwendig die resultierenden Pfade um den letzten Knoten zu ergänzen. Dies wird durch die Methode aus Listing 32 `getLastPathNodes` vorgenommen. Diese nimmt sich das letzte Element jedes Pfades im Vektor *paths* vor und iteriert über dessen Nachbarschaften. Dabei prüft die Methode für jeden Nachbarn, ob das Netzwerk, mit dem dieser Knoten verbunden ist, mit dem Netzwerk übereinstimmt, das zwischen dem letzten Element des Pfades und dem Vorgängerelement des Pfades liegt. Für alle Nachbarn, für die dies nicht übereinstimmt, bedeutet das, dass diese Nachbarn noch nicht im Pfad berücksichtigt wurden und die Pfadsammlung *paths* wird um diese Pfade ergänzt.

```
489 /**
490  * depth-first search-Method, which determines all possible paths starting
491  * from the given AdjacTupel (AdjacencyTupel).
492  * This is a helper-function for getPossiblePaths
493  * @param adt AdjacTupel to determine all possible Path starting from
494  * @param path Pathvector
495  */
496 private void getPath(AdjacTupel adt, Vector<AdjacTupel> path,
497                              Vector<Vector<AdjacTupel>> paths){
498    /*set the visited-state to true, to mark this node as visited*/
499    adt.getXtServer().setVisited(true);
500    path.add(adt);
501    Vector<AdjacTupel> p;
502    /* get all adjacencies from the XTServer of the given AdjacTupel*/
503    for(Iterator it = adt.getXtServer().getAdjacenciesIterator();
504                                                        it.hasNext();){
505       AdjacTupel a = (AdjacTupel)it.next();
506       /*if the neighbour was not visitet yet went down the tree*/
507       if(!a.getXtServer().isVisited()){
508          /* clone the old path to get all paths starting from this node*/
509          p = (Vector)path.clone();
510          /* add the cloned pathvector to the vector of pathvectors*/
511          paths.add(p);
512          getPath(a, p, paths);
513       }
514    }
515    /* set the visited-state of this node to false, that means this node is no
516    * longer visited
517    */
518    adt.getXtServer().setVisited(false);
519 }
```

Listing 31: getPaths (Tiefensuche) der Klasse GraphObserver

```
423 /**
424  * This method is a helper-method for getAllPossiblePaths. This Method
425  * determines all end-nodes of the paths in the given vector.
426  * @param paths the vector containing all possible paths
427  * @return vector the contains all paths extended be their end-nodes
428  */
429 private Vector<Vector<AdjacTupel>> getLastPathNode(Vector paths){
430    Vector<Vector<AdjacTupel>> toAdd = new Vector<Vector<AdjacTupel>>();
431    for(Iterator iter = paths.iterator(); iter.hasNext();){
432       Vector path = (Vector)iter.next();
433       /*get the last node of the path*/
434       AdjacTupel lastAdt = (AdjacTupel)path.get(path.size()-1);
435       for (Iterator nit = lastAdt.getXtServer().getAdjacenciesIterator();
436                                                          nit.hasNext();){
437          AdjacTupel adt = (AdjacTupel)nit.next();
438          /*check for all neighbours if the the neightbor is already an
439           * element of the path. If not, add it.
440           */
441          if(! adt.getConnIf().getConnectedNetwork().getAddress().equals(lastAdt
442                                .getConnIf().getConnectedNetwork().getAddress())){
443             Vector<AdjacTupel> p = (Vector<AdjacTupel>)path.clone();
444             p.add(adt);
445             toAdd.add(p);
446          }
447       }
448    }
449    return toAdd;
450 }
```

Listing 32: getLastPathNodes der Klasse GraphObserver

Der Pfadvektor, der durch die Methode `getPossiblePaths` zurückgeliefert wird, enthält nun alle möglichen Pfade, mit allen möglichen XTServern bzw. RIP-Routern als Ausgangspunkt. Auf Grundlage dieser Pfadsammlung können nun kritische Pfade d.h. Pfade, die einen Zyklus beinhalten, ermittelt werden, um darauf aufbauend eine entsprechende Konfiguration zur Triggerung der in diesen kritischen Pfaden enthaltenen RIP-Router generieren zu lassen.

6 Simulation und Anwendungsbeispiel

In diesem Abschnitt wird das Frontend und die Benutzerinteraktion mit der erweiterten `XT_Client-GUI`, der nun genannten `XT_Peer-GUI`, erläutert. Zu diesem Zweck wird die Simulation und die Auswertung anhand eines Szenarios vorgenommen werden, das in dieser Form noch nicht betrachtet wurde. Das Ausgangsszenario wird dabei im Abschnitt 6.4, das im folgenden „switched_loop“ genannt wird, erklärt. Eine Erweiterung dieses Szenarios ist im Abschnitt 6.6 beschrieben und erweitert das „switched_loop“ Szenario um eine alternative Route. Die Simulationen werden unter Verwendung des um dem MTI erweiterten RIP durchgeführt.

6.1 Voraussetzungen

Folgende Voraussetzung sind für die Nutzung der in dieser Arbeit entstandenen Software notwendig:

- VNUML ab Version 1.5
- Quagga in der Version 0.99.4 (die Sources sind auf der CD dieser Arbeit zu finden)
- Ein Filesystem, das die um die XT_SL-Erweiterungen und ggf. um den MTI ergänzte Quagga-Software (RIP-Daemon) enthält. Eine Anleitung zum Mounten eines Filesystems und Patchen der Quagga-Sources ist im Anhang A.3 und A.4 zu finden.
- Es müssen Network-Management-Interfaces zwischen Host und VNUML-Maschinen verwendet werden.
- Die Aktivierung des Hostmappings durch das Tag *<host_mapping/>* in der XML-Datei des VNUML-Szenarios, damit eine Assoziation zwischen Hostnamen und IP-Adressen möglich ist.

6.2 Automatisches Laden einer Topologie

Das Programm XTPeer kann zum einen durch den Befehl „java -jar xtpeer.jar“ oder durch das Skript „runXTPeer“ gestartet werden. Die `XT_Peer-GUI` ist in der Lage, anhand einer VNUML-Szenario-Datei sämtliche Verbindungen mit allen verfügbaren `XT_Servern` eines VNUML-Szenarios aufzubauen. Dazu lädt man über *Edit→Open→add XT-Servers from XML-File* die VNUML-Szenario Datei, die auch zum Starten des zu untersuchenden VNUML-Szenarios verwendet wurde. Mit allen verfügbaren `XT_Servern` des Szenarios wird dann eine XT-Verbindung aufgebaut. Sollten sich neben den um den `XT_Server` erweiterten RIP noch andere UML-Maschinen im Szenario befinden, gibt die GUI durch eine Meldung bekannt, dass keine XT-Verbindung zu diesen hergestellt werden kann.
Sobald eine XT-Verbindung hergestellt wurde, wird auch die Verbindung zwischen den `SL_Clients` und dem `SLServer` aufgebaut. Existieren im geladenen Szenario `XT_Server`, die keiner Erweiterung um den `SL_Client` obliegen, antworten diese `XT_Server` auf den

SL-Verbindungswunsch mit einem Fehlercode. Kommt es zu einer Abweisung des SL-Verbindungswunsches wird über einen Dialog angefragt, ob ein weiterer Versuch zur Herstellung der SL-Datenverbindung vorgenommen werden soll. Ein solcher Fehler tritt dann auf, wenn der XT_Peer beendet wurde, ohne die Verbindungen zwischen `XT_Server` und `XT_Client` sowie zwischen `SL_Client` und `SLServer` abzubauen.
`XTServer` können auch manuell hinzugefügt werden. Wurden die Verbindung zu allen `XTServern` hergestellt, muss nun noch die SL-Datenverbindung aufgebaut werden. Dabei werden zwei Fälle unterschieden:

- Es wurden nachträglich `XTServer` über *Edit → Server controls → Add XT-server* zu einem geladenen Szenario hinzugefügt. In diesem Fall müssen alle SL-Datenverbindungen neu aufgebaut werden. Dies geschieht über den Menüpunkt *Scenario → Reinitialize for new Simulation*. Dabei ist zu beachten, dass alle Datenstrukturen neu initialisiert werden.
- `XTServer` werden ausschließlich über *Edit → Server controls → Add XT-server* manuell hinzugefügt. Nachdem alle `XTServer` hinzugefügt wurden, muss nun noch die SL-Datenverbindung zwischen `SL_Clients` und `SLServer` hergestellt werden. Dies muss über *Edit → Server controls → connect SLClients* vorgenommen werden.

Der `SLServer` lauscht standardmäßig auf Port 5001 auf eingehende SL-Datenverbindungen. Soll dieser Port geändert werden, so ist dies vor dem Laden des Szenarios möglich. Die Einstellung dazu kann über *Settings → SLServer – Settings* vorgenommen werden. Ebenso ist es in diesen Einstellungen möglich die „Timeout“-Funktion zur Überprüfung der SL-Datenleitung zu deaktivieren bzw. zu aktivieren. Diese Einstellungen werden nur berücksichtigt, wenn sie vor dem Verbindungsaufbau vorgenommen wurden.

6.3 Aufbau der XT_Peer-GUI

Die `XT_Peer-GUI` stellt die Wurzelklasse des gesamten Programms dar. Aus ihr werden alle beteiligten Threads gestartet. Sie dient darüber hinaus der Darstellung aller relevanten Informationen. In Abbildungs 23 ist diese `XT_Peer-GUI` dargestellt, wie sie sich nach dem Laden eines VNUML-Szenarios präsentiert. Die GUI ist aus drei wesentlichen Bestandteilen aufgebaut. Im oberen Teil befindet sich die Darstellung der `XT_Server`-Topologie, wie sie in [Pä06] beschrieben wurde. Darunter ist der Bereich zur Darstellung der Daten angeordnet. Dieser setzt sich aus Karteireitern für jeden `XT_Server` zusammen. Die Karteikarte eines `XT_Servers` setzt sich ihrerseits wieder aus Karteikarten zusammen. Diese Karteikarten beinhalten eine Tabelle, die Routing-Informationen und zusätzlich errechnete Informationen zu jedem Netzwerk, das ein `XT_Server`, respektive RIP-Router kennt darstellen und einen CTI-Verlaufs-Graphen für dieses Netzwerk.
Liegen Informationen bezüglich des MTI vor, wird dies durch ein *(i)* für Information in der *cause*-Spalte des jeweiligen Eintrags der Netzwerktabelle gekennzeichnet. In diesem Fall können diese MTI-Informationen durch einen Druck mit der Maustaste in dieser Zeile aufgerufen werden. Die MTI-Informationen werden in einem, sich zusätzlich öffnenden Fenster angezeigt.

Abbildung 23: Aufbau der XT_Peer-GUI

6.4 Switched-Loop Szenario

Das folgende Szenario stellt eine Konstellation dar, wie sie in größeren Netzwerken vorgefunden werden kann. Die Topologie ist in Abbildung 24 dargestellt. Die Router `R1`, `R2`, `R3` und `R5` befinden sich dabei im kritischen Abschnitt der Topologie, da sie einen Zyklus bilden. Die Besonderheit an diesem Szenario stellt dabei die Verbindung zwischen den Routern `R1`, `R3` und `R2` dar, die über ein und dasselbe Netzwerk Net2 miteinander verbunden sind. Diese Verbindung wird von VNUML-Seite durch eine virtual Bridge realisiert und ist mit der Verbindung mehrerer Netzwerkknoten über einen Switch zu vergleichen. Die Besonderheit stellt sich mit der Tatsache ein, dass die Router `R1`, `R2` und `R3` über nur ein Interface pro Router miteinander verbunden sind, der MTI sich bei seiner Entscheidung, ob eine Route angenommen wird, aber auf den Interfaceindex, wie in [Koc05] beschrieben und nicht auf die Interface-Adresse des Nachbarn (learned From) stützt von dem das RIP-Update stammt. Die Bedingung zur Prüfung des eingehenden Updates durch den MTI ist in Listing 33 zu sehen. Handelt es sich beim Eingang des Updates um den selben Interface-Index, über das das vormals akzeptierte RIP-Updates einging (vergl. Listing 33 Zeile 216), so wird durch den MTI stets true (1) zurückgegeben und die Route wird akzeptiert.

```
213 /**
214  * preconditions
215  */
216 if ((oldroute->ifindex == ifp->ifindex)
217     || (rte->metric >= RIP_METRIC_INFINITY)
218     || (oldroute->metric > RIP_METRIC_INFINITY))
219     return 1;
```

Listing 33: Vorbedingung des RIP-MTI

Die Trigger-Konfiguration wurde durch den „intelligenten “Konfigurations-Generator aus [Keu07], der den `GraphObserver` aus Abschnitt 5 verwendet, erstellt und baut sich wie in Listing 34 auf. In den Zeilen 15-17 schicken sich alle Router Updates, damit das Netz konvergiert. Danach schickt Router `R4` ab Zeile 18 bis 22 keine Updates über die Erreichbarkeit von Net5[18] mehr, so dass der Timeout Timer für diese Route in `R3` abläuft und der dies den Routern `R1` und `R2` mitteilt. Der kritische Punkt ist in Zeile 23, da `R5` noch die alte, zu diesem Zeitpunkt bereits falsche Erreichbarkeitsinformation besitzt, `R1` und `R2` aber bereits über die Unerreichbarkeit von Net5 über `R3` informiert sein sollten. `R5` schickt sodann die falschen Informationen an `R2` weiter, der diese dankend entgegen nimmt und `R3` und `R1` darüber in Kenntnis setzt. `R1` sollte keine Updates von `R5` erhalten, da dies der Split-Horizon-Mechanismus verbietet. Zu diesem Zeitpunkt entwickelt sich der CTI.

[18] Net5 wird im folgenden synonym zu 10.0.5.0/24 verwendet.

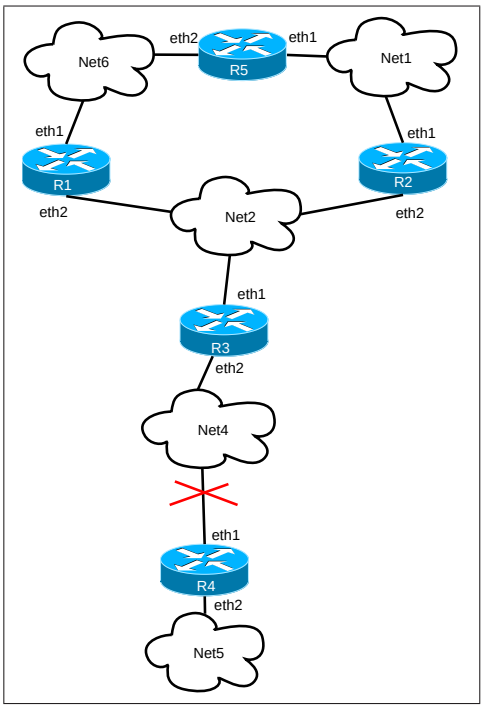


Abbildung 24: Switched-Loop Topologie

```
1 // generated for path: R4(1) eth1 -> R3(2) eth1
2 //-> R1(1) eth1 -> R5(1) eth1 -> R2(1) eth2 -> R3(2)
3 // autogenerated config file
4 Rip-Router r3
5 Rip-Router r2
6 Rip-Router r1
7 Rip-Router r5
8 Rip-Router r4
9
10
11 // update_time in ms
12 periodic_update 3000
13
14 Update-Forecast
15 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(0.0,0.0);
16 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(0.0,0.0);
17 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(0.0,0.0);
18 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(x,x);
19 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(x,x);
20 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(x,x);
21 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(x,x);
22 r3(0.0,0.0); r2(0.0,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(x,x);
23 r3(0.5,0.0); r2(x,0.0); r1(0.0,0.0); r5(0.0,0.0); r4(x,x);
24 r3(0.0,0.0)a; r2(x,0.0)a; r1(0.0,0.0)a; r5(1.0,0.0)a; r4(x,x);
```

Listing 34: Trigger-Konfiguration für das Switched-Loop-Szenario

Betrachtet man sich Abbildung 25, kann man erkennen, dass die „falschen“Updates von Router `R3` abgewiesen werden, solange der Garbage Collection Timer für den RT-Eintrag der Router nach Net5 noch nicht abgelaufen ist. Dies geschieht zu den Zeitpunkten 00:29:298

für ein Update mit der Metrik 6 zum Zeitpunkt 00:32:326 für das der Metrik 9 und um 00:33:055 für das Update, das die Metrik 12 propagiert. Danach ist der Garbage-Collection Timer für den alten Pfad nach Net5 abgelaufen und der Eintrag aus der Routing-Tabelle von Router R3 entfernt. Erst zu diesem Zeitpunkt 00:37:073 wird das letzte „falsche“Update und danach das Update über die Unerreichbarkeit von `R2` angenommen, das als Ergebnis des vorliegenden CTI zwischen `R2`, `R5` und `R1` hervorging.

Bis zu diesem Zeitpunkt 00:00:38:098 gelingt es dem MTI auf `R3` die Inkonsistenz

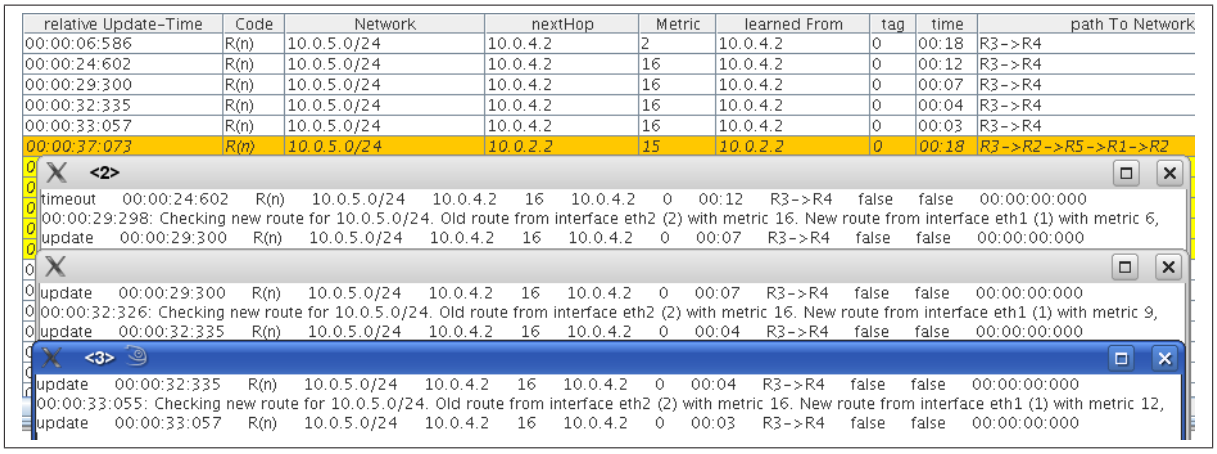


Abbildung 25: Abweisung falscher Erreichbarkeits-Updates von R3 für Netz 10.0.5.0/24

zur Erreichbarkeit von Net5 für sich minimal zu halten. Allerdings trägt dies aufgrund der Verbindung zwischen `R1`, `R2` und `R3` über einen Switch im Zyklus selbst nicht dazu bei, den CTI zu verhindern, da `R3` wegen der vorliegenden direkten Verbindung zwischen `R1` und `R2` übergangen werden kann. Die Erkennung des routing-loops in `R1` wird wohl durch die alleinige Betrachtung des Interface-Index, über das sowohl das korrekte Update von R3, als auch das falsche von R2 eingegangen ist, getrübt, so dass sich auch aufgrund dessen der CTI entwickeln kann.
Ein unangenehmer Effekt tritt allerdings ein, wenn der Garbage-Collection Timer für Net5, hervorgerufen durch den CTI, aktiv ist. Ist in diesem Zeitraum Net5 wieder über `R4` erreichbar, werden Updates über diese Erreichbarkeit vom MTI auf `R3` beispielsweise zum Zeitpunkt 00:44:412 abgewiesen, wie es in Abbildung 26 zu erkennen ist. D.h. Net5 wird von `R3` als nicht erreichbar deklariert, obwohl ein Pfad über `R4` wieder vorhanden ist. Der Wert für den Garbage-Collection Timer ist für diese Simulation auf 12 Sekunden gesetzt, sodass das Netz weiterhin für diese 12 Sekunden nicht erreichbar ist. Im schlechtesten Fall würde ein korrektes Update unter diesen Voraussetzungen während der timeout-timer Phase des in diesem Bespiel vorletzten falschen Updates eintreffen und abgewiesen werden. Das würde dann im schlechtesten Fall zusätzlich zu einer Verzögerung von 18 Sekunden führen. Also insgesamt einer Inkonsistenz von 30 Sekunden.
Die Werte für den Garbage-Collection Timer von 12 bzw. 18 Sekunden für den timeout timer entsprechen einem Zehntel der Standardwerte von 120 Sekunden bzw. 180 Sekunden [Qua]. Aus diesem Grund ist eine Relativierung des negativen Effekts der Beobachtung zu

erläutern. Die zeitliche Abfolge der CTI-Updates ist unabhängig von den Werten der Timer, sodass dieser Effekt unter Verwendung der Standardwerte nie eintreten könnte.
Dazu muss man sich das Verhalten des RIP für Triggered-Updates betrachten. Triggered-Updates stellen Updates dar, die durch Änderungen an der Routing-Tabelle eines Routers hervorgerufen werden. Dieser Router sendet allen direkten Nachbarroutern nun ein Update, das genau diese Änderung enthält. Um die Netzlast gerade in Netzen geringer Bandbreite gering zu halten wird nicht direkt nach jeder Änderung ein Triggered-Update gesendet. In [Mal98] ist deshalb folgendes für Triggered Updates festgelegt. Ein Triggered-Update wird nicht sofort gesendet, sondern obliegt einem Timer, der einen zufälligen Wert zwischen 1 und 5 Sekunden besitzt. Sollte sich die Erreichbarkeitsinformation für dieses Netz innerhalb dieses Timers nochmals ändern, wird der Timer gestoppt, das alte Triggered-Update verworfen und ein neuer Timer für das neue Triggered-Update gestartet. So ist es möglich, die Netzlast in einem Netz bei häufigen Änderungen von Erreichbarkeiten relativ gering zu halten.
Pro Router innerhalb eines Zyklus wird also ein Triggered-Update maximal 5 Sekunden zurückgehalten. Der Zyklus wird maximal $\frac{16-offset}{\#Router}$ durchlaufen. Der $offeset$ beschreibt dabei die Anzahl der Hops, ausgehend vom Router, der das falsche Update in den Zyklus schickt (in diesem Beispiel R5), bis zum Zielnetz. Nach [Keu07] ergibt sich für die maximale Dauer eines CTI folgende Gleichung 1.

$$\begin{aligned} t_{max}^{CTI} &= \frac{n * 5s * (16 - offset)}{n} + ((15 - offset) \mod n) * 5s \\ &= 5s * (16 - offset) + ((15 - offset) \mod n) * 5s \end{aligned} \tag{1}$$

In diesem Beispiel wird der $offset$ durch die Router `R4`, `R3`, `R2` und `R5` gebildet und beträgt 4. n steht für die Anzahl der Hops im Zyklus. Es ergibt sich demnach für die maximale Dauer eines CTI in diesem Beispiel folgender Wert.

$$\begin{aligned} t_{max}^{CTI} &= 5s * (16 - 4) + (11 \mod 3) * 5s \\ &= 5s * 12 + 2 * 5s \\ &= 70s \end{aligned} \tag{2}$$

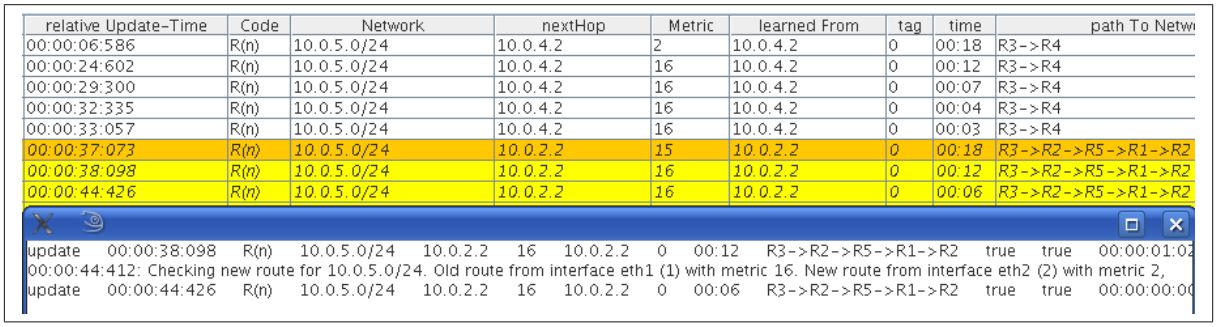
Es ergibt sich somit:

$$t_{max}^{CTI} = 70 < 120 = t_{default}^{garbage-timer} \tag{3}$$

Das beobachtete Verhalten ist unter der Voraussetzung, dass der Garbage-Collection Timer in Router R3 > 70 Sekunden gewählt wird, auszuschließen.

6.5 Speichern und Laden von Szenariodaten

Zum Vergleich verschiedener Simulationen bietet die `XT_Peer-GUI` die Möglichkeit, die gesammelten und zusätzlich ermittelten Daten in einer XML-ähnlichen Datei zu speichern. Über *Edit → Save → Save Scenario Data to File* kann dann der Speicherort und der Name der zu erstellenden Datei ausgewählt werden. Der gewünschte Dateiname wird automatisch um die



relative Update-Time	Code	Network	nextHop	Metric	learned From	tag	time	path To Netw
00:00:06:586	R(n)	10.0.5.0/24	10.0.4.2	2	10.0.4.2	0	00:18	R3->R4
00:00:24:602	R(n)	10.0.5.0/24	10.0.4.2	16	10.0.4.2	0	00:12	R3->R4
00:00:29:300	R(n)	10.0.5.0/24	10.0.4.2	16	10.0.4.2	0	00:07	R3->R4
00:00:32:335	R(n)	10.0.5.0/24	10.0.4.2	16	10.0.4.2	0	00:04	R3->R4
00:00:33:057	R(n)	10.0.5.0/24	10.0.4.2	16	10.0.4.2	0	00:03	R3->R4
00:00:37:073	R(n)	10.0.5.0/24	10.0.2.2	15	10.0.2.2	0	00:18	R3->R2->R5->R1->R2
00:00:38:098	R(n)	10.0.5.0/24	10.0.2.2	16	10.0.2.2	0	00:12	R3->R2->R5->R1->R2
00:00:44:426	R(n)	10.0.5.0/24	10.0.2.2	16	10.0.2.2	0	00:06	R3->R2->R5->R1->R2

update 00:00:38:098 R(n) 10.0.5.0/24 10.0.2.2 16 10.0.2.2 0 00:12 R3->R2->R5->R1->R2 true true 00:00:01:0
00:00:44:412: Checking new route for 10.0.5.0/24. Old route from interface eth1 (1) with metric 16. New route from interface eth2 (2) with metric 2,
update 00:00:44:426 R(n) 10.0.5.0/24 10.0.2.2 16 10.0.2.2 0 00:06 R3->R2->R5->R1->R2 true true 00:00:00:0

Abbildung 26: Abweisung korrekter Erreichbarkeits-Updates in R3 für Netz 10.0.5.0/24

Dateierweiterung *.sdf* = *„scenario data file“* ergänzt.
Das Laden von gespeicherten Szenariodaten geschieht über *Edit* → *Open* → *open saved simulation data*. Es ist zu empfehlen, diese gespeicherten Daten zu laden, bevor das zu simulierende VNUML-Szenario geladen wird (vergl. Abschnitt 6.2), damit die geladenen Szenariodaten ebenfalls im Verlaufsgraph berücksichtigt werden können.

6.6 Switched-Loop Szenario mit Alternativ-Route

Die Beobachtung aus Abschnitt 6.4 lässt die Vermutung aufkommen, dass `R3` unter Einsatz des MTI während des Zeitraums des Garbage-Collection Timers keine Erreichbarkeitsupdates akzeptiert, auch wenn diese korrekt sind. Um diese Annahme zu untermauern, wird in diesem Abschnitt das Szenario aus Abschnitt 6.4 um einen alternativen Pfad erweitert und zwei Simulationsergebnisse besprochen. Die erste Simulation wurde dabei mit MTI und die zweite mit dem herkömmliche RIP ohne MTI-Erweiterung durchgeführt. In Abbildung 27 ist diese erweiterte Topologie dargestellt.
Es besteht in diesem Szenario die Möglichkeit Net5 nach Ausfall der Route `R3`, `R4` über `R6`, mit der Metrik 3 zu erreichen. Die Konfiguration zur Provokation eines CTI zwischen `R2`, `R1` und `R5` bleibt dabei wie in Abschnitt 6.4 bestehen. Die Ergänzungen dieser Konfiguration um die Triggerung des hinzugefügten Router ist in Listing 35 aufgezeigt. Die Router `R3` und `R4` wurden um ein Interface ergänzt, über das der alternative Pfad angeboten wird.

Nachdem `R3` die Unerreichbarkeit von Net5 über `R4` registriert hat, indem `R4` keine Updates über diese Erreichbarkeit an `R3` sendet (vergl. Listing 35 Zeile 19-25), unterrichtet `R3` seine Nachbarn, über diese Unerreichbarkeit und es entwickelt sich ein CTI wie schon in Abschnitt 6.4 beschrieben. Im Unterschied dazu wird nun Router `R3` von `R6` ein alternativer Pfad mit Metrik 3 (vergl. Listing 35 Zeilen 24 und 25) angeboten, die `R3` im günstigsten Fall akzeptieren sollte.
Ein Ergebnis in dieser Simulation ist in Abbildung 28 dargestellt.

Wie in Abbildung 28 zu erkennen ist, kommt es in Router R3 nach der konvergenten Phase zum Zeitpunkt 00:00:40:670 zu einem Timeout für den Pfad nach Netz 10.0.5.0/24,

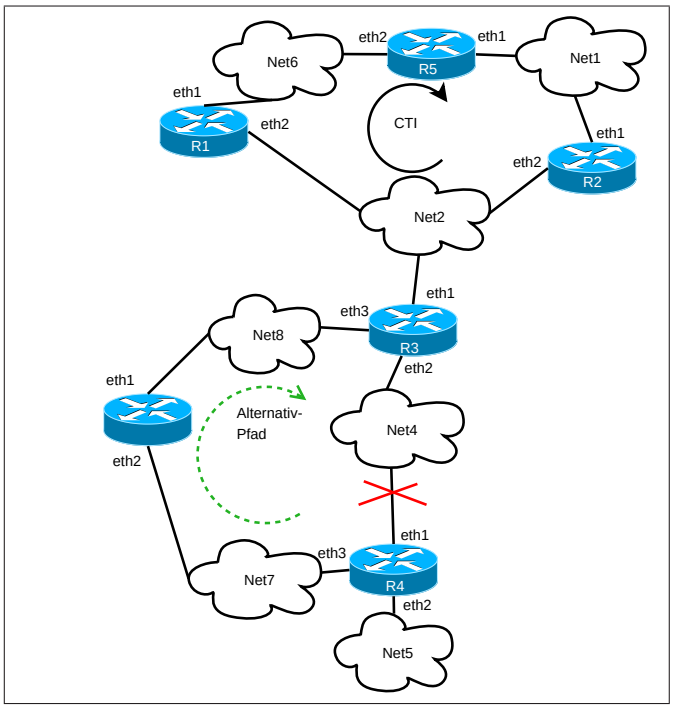


Abbildung 27: Switched-Loop Topologie mit Alternativ-Pfad

sodass die Metrik auf für diesen Pfad auf RIP_METRIC_INFINITY gesetzt wird.

Daraufhin wird R3 von R6 der alternative Pfad in dieses Netz mit einer Metrik von 3 angeboten. Dies geschieht zum Zeitpunkt 00:00:41:742. Betrachtet man sich die MTI-Information, ist zu erkennen, dass dieser Pfad über das Interface eth3 des R3 angenommen wurde und aufgrund des MTI abgelehnt wird. Zu den Zeitpunkten 00:00:44:994 und 00:00:47:159 weist Router R3 aufgrund der MTI-Entscheidung Falsch-Updates von Router R1 über Interface eth1 ab und arbeitet somit erwartungsgemäß.

Betrachtet man Abbildung 29 in der das Verhalten von Router R1 gezeigt ist, erkennt man, dass dieser zum Zeitpunkt 00:00:41:808 das Update über die Unerreichbarkeit von Netz 10.0.5.0/24 von R3 bekommt und den entsprechenden Eintrag in seiner Routing-Tabelle vornimmt. Da zu diesem Zeitpunkt Router R5 noch die alte Erreichbarkeits-Information über Netz 10.0.5.0/24 besitzt und ein entsprechendes Update an R1 gesendet[19] wird, übernimmt `R1` diese Information ebenfalls in seine Routing-Tabelle. Dies geschieht zum Zeitpunkt 00:00:44:683. Kurz darauf wird ihm von `R3` die Unerreichbarkeit von Netz 10.0.5.0/24 abermals mitgeteilt, die R1 aber nicht übernimmt. Dies resultiert in einem Doppeleintrag in der Tabelle aus Abbildung 29. In den darauf folgenden Einträgen ist zu erkennen,

[19]Betrachtet man sich die Spalte „pathToNetwork, erkennt man, dass der vollständige Pfad trotz Unerreichbarkeit aufgelistet wird. Das resultiert daraus, dass bei der Pfadanalyse aus Abschnitt 4.4.1 die Metriken unberücksichtigt bleiben, was wiederum dazu führt, dass Erreichbarkeitsinformationen mit einer Metrik von 16 dennoch in der Pfadanalyse berücksichtigt werden.

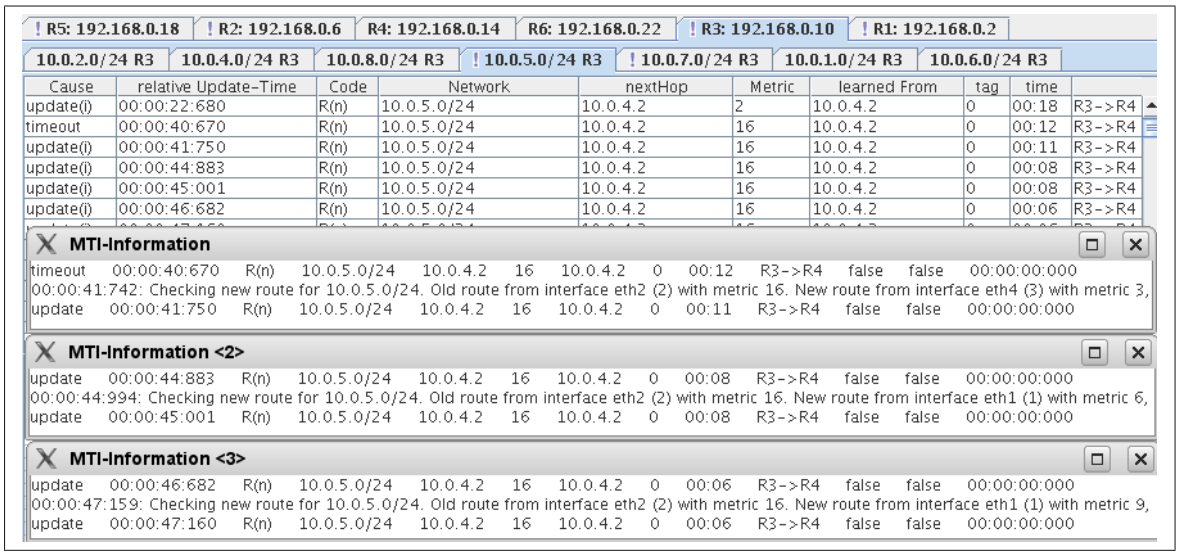


Abbildung 28: Router R3 für Netz 10.0.5.0/24-Abweisung korrekter Erreichbarkeits-Updates von R9

dass sich zwischen `R1`, `R5` und `R2` ein CTI entwickelt und die Metrik in R1 bis zum Wert RIP_METRIC_INFINITY hoch zählt.

Trotz der Tatsache, dass der CTI von `R3` erkannt wird, handelt es sich in diesem Fall insgesamt um eine Verschlechterung, da die Zeitspanne eines CTI nie die Zeitspanne des Garbage-Collection Timers, bei einem Timerwert größer 70 Sekunden übersteigt (vergl. Abschnitt 6.4 Gleichungen 1 bis 3), das Netz 10.0.5.0/24 über `R6` aber für genau diese Zeitspanne als falsch und damit als nicht erreichbar deklariert wird.

In Abbildung 30 ist das Verhalten des herkömmlichen RIP zu sehen. Nachdem der Timeout für den Pfad nach Netz 10.0.5.0/24 über R4 stattfand, nimmt R3 sofort die angebotene Alternative über R6 an. Dieses Verhalten wirkt sich ebenfalls auf die Erreichbarkeits-Informationen der Router R1, R2 und R5 aus. R3 sendet sofort nach Erhalt der alternativen Route entsprechende triggered Updates an R1 und R2. Diese übernehmen diese Information in ihre Routing-Tabellen, was die Entstehung eines CTI verhindert.

Betrachtet man sich über *Scenario → show statistics* die Statistik der beiden durchgeführten Simulationen, so ist in Abbildung 31 zu erkennen, dass der herkömmliche RIP in `R3`, wie bereits bemerkt, die angebotene Route über `R6` nicht ablehnt und diesen Alternativpfad an seine Nachbarn `R2` und `R1` in Form eines *triggered Updates* bekanntgeben kann, so dass der zwischen R1, R2 und R5 mögliche CTI nicht entsteht.

Dies resultiert in einer durchschnittlichen CTI-Dauer von R5 von 0 Sekunden, während die durchschnittliche CTI-Dauer bei der Simulation unter Verwendung des CTI bei 6:465 Sekunden und einem CTI-Aufkommen von 6 lag.
Die Werte ergaben sich nach jeweils 20 Simulationsdurchläufen.

```
1  // generated for path: R4(1) eth1 -> R3(2) eth1
2  //-> R1(1) eth1 -> R5(1) eth1 -> R2(1) eth2 -> R3(2)
3  // autogenerated config file
4  Rip-Router r3
5  Rip-Router r2
6  Rip-Router r1
7  Rip-Router r5
8  Rip-Router r4
9  Rip-Router r6
10
11
12 // update_time in ms
13 periodic_update 3000
14
15 Update-Forecast
16 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(0.0,0.0); r6(x,0.0);
17 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(0.0,0.0); r6(x,0.0);
18 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(0.0,0.0); r6(x,0.0);
19 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(x,x,0.0); r6(x,0.0);
20 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(x,x,0.0); r6(x,0.0);
21 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(x,x,0.0); r6(x,0.0);
22 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(x,x,0.0); r6(x,0.0);
23 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(x,x,0.0); r6(x,0.0);
24 r3(0.0,0.0,x); r2(0.0,0.0); r1(x,0.0); r5(0.0,0.0); r4(x,x,0.0); r6(0.0,0.0);
25 r3(0.0,0.0,x); r2(0.0,0.0)a; r1(x,0.0)a; r5(1.0,0.0)a; r4(x,x,0.5); r6(0.0,0.0);
```

Listing 35: Trigger-Konfiguration für das Switched-Loop-Szenario mit Alternativroute

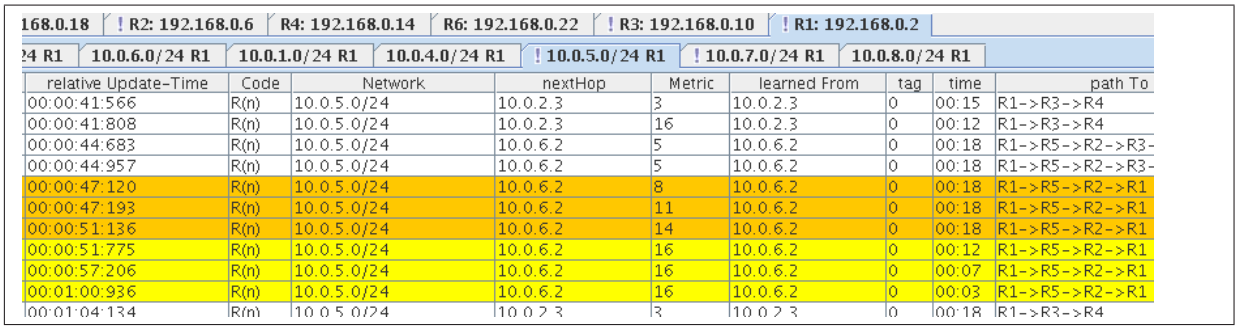


Abbildung 29: Router R2 für Netz 10.0.5.0/24- Entstehung eines CTI

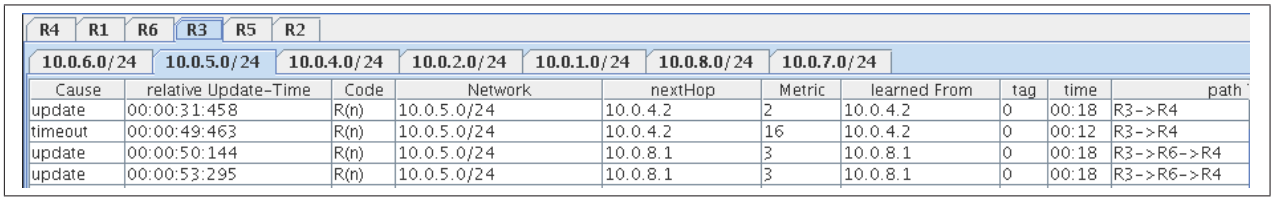


Abbildung 30: Router R3 für Netz 10.0.5.0/24 - Verhalten des herkömmlichen RIP bei angebotener Alternativ-Route von R6

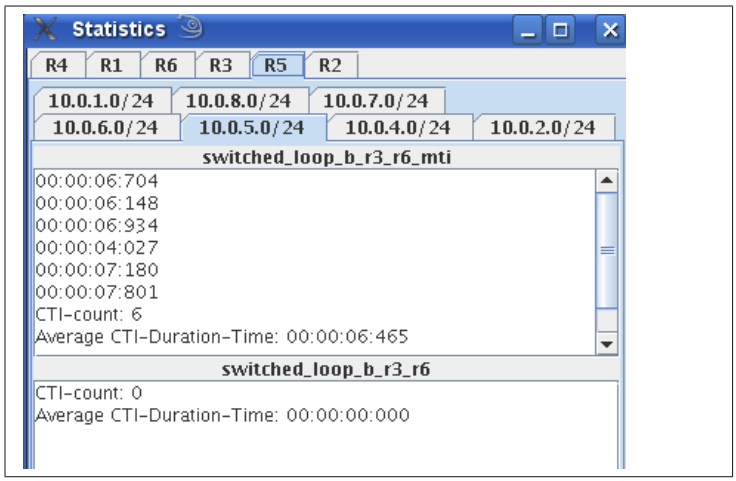


Abbildung 31: Statistiken von Router R5 für Netz 10.0.5.0/24

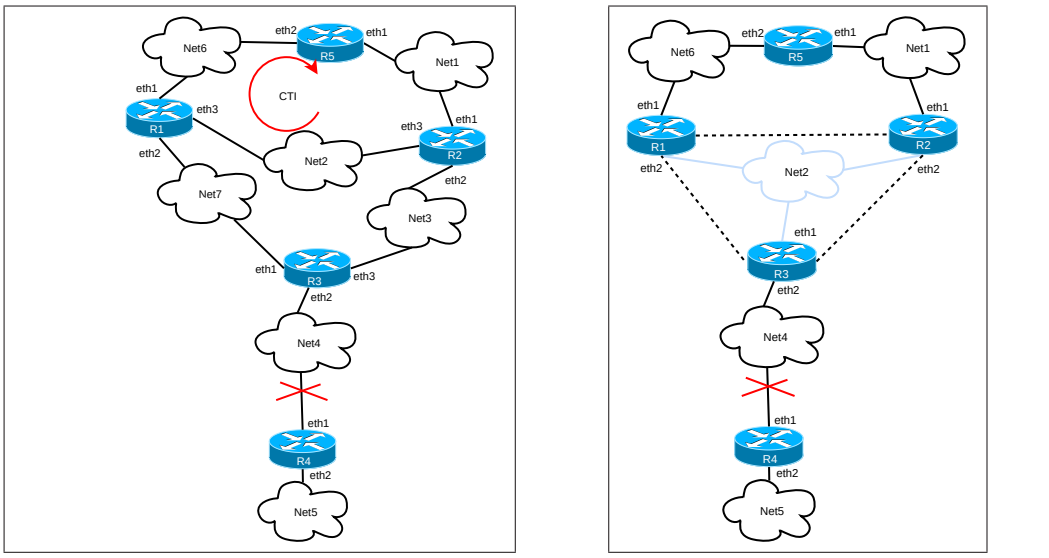
6.7 Beobachtung

Die beobachteten Ergebnisse aus den Simulationen sollen hier noch einmal zusammengefasst werden. Dabei werden denkbare Lösungsansätze aufgezeigt und erläutert.

Aus Beispiel 6.4 ergab sich, dass trotz der Erkennung des Routing-Loop auf Router R3 ein CTI zwischen R1, R5 und R2 entstehen kann, da die Wirksamkeit des MTI auf R3 durch die Verbindung zwischen R1, R2 und R3 über einen Switch übergangen werden kann.

Lösungsansatz: Wie bereits bemerkt orientiert sich der MTI in seiner momentanen Implementierung auf den Interface-Index des eingehenden RIP-Updates. Würde sich die Entscheidung über die Berücksichtigung von Updates z.B. auf die learnedFrom-Adresse stützen, wäre eine Reduzierung auf das Interloop-Szenario, das in [Wol06] unter der Bezeichnung “Verschachtelte Schleifen„ betrachtet, und in [Keu07] noch einmal aufgegriffen wurde, denkbar.
Dazu betrachte man sich Abbildung 32. In Abbildung 32 (a) ist dieses Interloop-Szenario dargestellt. Das durch [Wol06] geschilderte Problem in dieser Topologie stellt sich dabei wie folgt dar: Router R3 kann den Routing-Loop R3-R1-R5-R2-R3 unterbinden. Router R2 allerdings kann, wegen der direkten Verbindung zu R1 weiterhin Erreichbarkeitsupdates von R1 beziehen. Der MTI auf R2 ist aber nicht in der Lage den Routing-Loop zwischen R2, R5 und R1 als Zyklus zu identifizieren, wodurch in diesem “kleineren Zyklus„ ein CTI entstehen kann. Eine solcher CTI ist in Abbildung 33 aus der Sicht von Router R2 aufgezeigt.

In Abbildung 32 (b) ist die Reduzierung des switched-loop-Szenarios auf eben dieses Interloop-Problem dargestellt, die sich ergibt, wenn durch den Source-Router, bzw. den MTI die learnedFrom-Adressen, für die in Listing 33 gezeigte Vorbedingung sowie für den Zugriff auf die Tabellen und die Berechnung der Tabellenwerte durch den MTI betrachtet werden. Dabei könnte dann der MTI des Routers R2 zwischen den Nachbarn R3 und



(a) Interloop-Szenario (b) reduziertes Switched-Loop-Szenario

Abbildung 32: Szenarien

! R2: 192.168.0.6 | ! R3: 192.168.0.10 | R4: 192.168.0.14 | ! R1: 192.168.0.2

0.0.2.0/24 R2 | 10.0.3.0/24 R2 | 10.0.6.0/24 R2 | 10.0.4.0/24 R2 | 10.0.7.0/24 R2 | ! 10.0.5.0/24 R2

ne	Code	Network	nextHop	Metric	learned From	tag	time	path To N
	R(n)	10.0.5.0/24	10.0.3.3	16	10.0.3.3	0	00:12	R2->R3->R4
	R(n)	10.0.5.0/24	10.0.2.1	6	10.0.2.1	0	00:18	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	6	10.0.2.1	0	00:18	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	6	10.0.2.1	0	00:18	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	9	10.0.2.1	0	00:18	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	9	10.0.2.1	0	00:16	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	12	10.0.2.1	0	00:18	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	15	10.0.2.1	0	00:18	R2->R1->R5->R2
	R(n)	10.0.5.0/24	10.0.2.1	16	10.0.2.1	0	00:12	R2->R1->R5->R2

Abbildung 33: Router R2 für Netz 10.0.5.0/24 - CTI aus Sicht von R2

R1 unterscheiden. Eine Lösung des Interloop-Problems unter der Berücksichtigung der learnedFrom-Adressen, würde mit einer Lösung des switched-loop-Problems einhergehen können.

In Beispiel 6.6 wurde festgestellt, dass Router R3 während der Garbadge-Collection-Phase keine Erreichbarkeitsinformationen annimmt, auch wenn diese korrekt sind und ins Zielnetz führen.
Lösungsmöglichkeit: Eine Korrektur der MTI-Implementation sollte an dieser Stelle Abhilfe schaffen können.

7 Ausblick

Mit der XT-Software aus [Pä06] steht eine gute Grundlage zur Verfügung, gezielte Experimente unter Verwendung der Quagga-Routing-Suite und VNUML durchzuführen, um das Konvergenzverhalten des RIP und des RIP-MTI, gerade im Hinblick auf die Enstehung von CTIs durchführen zu können. Mit den Ergänzungen aus dieser Arbeit ist eine direkte zentrale Betrachtung des Konvergenzverhaltens ermöglicht worden. Somit ist es nun einfacher die Zusammenhänge zwischen Updateaufkommen, Updatereihenfolgen und CTI-Entstehung zu erkennen und durch die zeitnahe Analyse kritischer Konstellationen direkt angezeigt und nachvollziehen zu können.
Auch die Betrachtung von Ergebnissen verschiedener Simulationsdurchläufe wird ermöglicht. Mit den gesammelten Daten besteht auch die Möglichkeit, Auswertungen unabhängig von denen der SL-Erweiterungen durchzuführen, sofern die Daten in Form von csv-Dateien gespeichert werden.
Durch die Art der `SL_Client`-Implementierung ist es möglich den Informationsfluss zum `SLServer` nach belieben auszubauen und zu erweitern.
Neben dem Einsatz dieses Werkzeugs zur Evaluation und Verbesserung des MTI-Algorithmus, wäre auch der Einsatz in der Lehre denkbar, da man ohne großen Aufwand schnellen Aufschluss über die Arbeitsweise des RIP-Algorithmus und die Entstehung des CTI-Problems erlangen kann.

Literatur

[gen07] Howto user mode linux,
URL: http://gentoo-wiki.com/howto_user_mode_linux, 2007.

[Keu07] Tim Keupen. Automatische Generierung von Testfällen für den RIP-MTI Algorithmus. Dimplomarbeit, Universität Koblenz-Landau, 2007.

[Koc05] Tobias Koch. Implementation und Simulation von RIP-MTI. Dimplomarbeit, Universität Koblenz-Landau, 2005.

[Kru06] Guido Krueger. Handbuch der JAVA-Programmierung,. Addison-Wesley, 4., aktualisierte Auflage 2006

[LLP03] Bruce S. Davie Larry L. Peterson. *Computernetze, Eine systematische Einführung.* dpunkt.verlag, deutsche Ausgabe der 3. amerikanischen Auflage Edition, 2003.

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

[Pä06] Daniel Pähler. Extern steuerbare Routing-Updates im rip-daemon der Quagga-Programmsuite. Dimplomarbeit, Universität Koblenz-Landau, 2006.

[Qua] Quagga Documantation,
URL: http://www.quagga.net/docs/docs-info.php.

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

[Tea06] User Mode Linux Core Team. User Mode Linux howto,
URL: http://user-mode-linux.sourceforge.net/old/usermodelinux-howto.html, January 2006.

[wik] Network Time Protocol.
URL: http://de.wikipedia.org/wiki/network_time_protocol.

[Wol06] Bernhard Wolf. Untersuchung und Simulation des RIP-MTI-Algorithmus. Studienarbeit, Universität Koblenz-Landau, 2006.

[zeb] The thread mechanism in zebra.
URL: http://wiki.cs.uiuc.edu/cs427/the+thread+mechanism+in+zebra.

A Anhang

A.1 Quellcodes

```
3538 DEFUN (show_ip_rip,
3539        show_ip_rip_cmd,
3540        "show ip rip",
3541        SHOW_STR
3542        IP_STR
3543        "Show RIP routes\n")
3544 {
3545   struct route_node *np;
3546   struct rip_info *rinfo;
3547
3548   if (! rip)
3549     return CMD_SUCCESS;
3550
3551   vty_out (vty, "Codes: R - RIP, C - connected, S - Static, O - OSPF, B - BGP%s"
3552     "Sub-codes:%s"
3553     "      (n) - normal, (s) - static, (d) - default, (r) - redistribute,%s"
3554     "      (i) - interface%s%s"
3555     "    Network            Next Hop         Metric From            Tag Time%s",
3556    VTY_NEWLINE, VTY_NEWLINE,  VTY_NEWLINE, VTY_NEWLINE, VTY_NEWLINE, VTY_NEWLINE);
3557
3558   for (np = route_top (rip->table); np; np = route_next (np))
3559     if ((rinfo = np->info) != NULL)
3560       {
3561         int len;
3562
3563         len = vty_out (vty, "%c(%s) %s/%d",
3564                        /* np->lock, For debugging. */
3565                        zebra_route_char(rinfo->type),
3566                        rip_route_type_print (rinfo->sub_type),
3567                        inet_ntoa (np->p.u.prefix4), np->p.prefixlen);
3568
3569         len = 24 - len;
3570
3571         if (len > 0)
3572           vty_out (vty, "%*s", len, " ");
3573
3574         if (rinfo->nexthop.s_addr)
3575           vty_out (vty, "%-20s %2d ", inet_ntoa (rinfo->nexthop),
3576                    rinfo->metric);
3577         else
3578           vty_out (vty, "0.0.0.0               %2d ", rinfo->metric);
3579
3580         /* Route which exist in kernel routing table. */
3581         if ((rinfo->type == ZEBRA_ROUTE_RIP) &&
3582             (rinfo->sub_type == RIP_ROUTE_RTE))
3583           {
3584             vty_out (vty, "%-15s ", inet_ntoa (rinfo->from));
3585             vty_out (vty, "%3d ", rinfo->tag);
3586             rip_vty_out_uptime (vty, rinfo);
3587           }
3588         else if (rinfo->metric == RIP_METRIC_INFINITY)
3589           {
3590             vty_out (vty, "self            ");
3591             vty_out (vty, "%3d ", rinfo->tag);
3592             rip_vty_out_uptime (vty, rinfo);
3593           }
3594         else
3595           {
3596             if (rinfo->external_metric)
3597               {
```

```
3598                 len = vty_out (vty, "self (%s:%d)",
3599                                    zebra_route_string(rinfo->type),
3600                                    rinfo->external_metric);
3601                 len = 16 - len;
3602                 if (len > 0)
3603                   vty_out (vty, "%*s", len, " ");
3604               }
3605             else
3606               vty_out (vty, "self             ");
3607             vty_out (vty, "%3d", rinfo->tag);
3608           }
3609
3610         vty_out (vty, "%s", VTY_NEWLINE);
3611       }
3612   return CMD_SUCCESS;
3613 }
```

Listing 36: DEFUN-Makro der ripd.c

A.2 Anleitung zur Patch-Erstellung

Um einen Patch aus veränderten Source-Codes zu erstellen, müssen folgende Schritte durchgeführt werden:

- Die Verzeichnisse des Original (z.B. Quagga-old) und der veränderten Quelle (z.B. Quagga-new) müssen in einem Verzeichnis liegen.
- Mit dem Kommandozeilen-Tool *diff* können diese Verzeichnisse miteinander verglichen und die Unterschiede in eine Datei ausgegeben werden. Dazu führt man den Befehl diff wie folgt aus:

  diff -Naur Quagga-old Quagga-new > patchname.patch

  Die Datei patchname.patch stellt dabei die Patch-Datei dar.

A.3 Anleitung zum Patchen

Die in dieser Arbeit und den vorangegangenen Arbeiten entstandenen Erweiterungen des Quagga basieren auf der Quagga-Version 0.99.4. Entsprechend wurden die Patches auch auf dieser Version erstellt. Die folgende Anleitung zum Patchen und Compilieren wurde aus der Diplomarbeit [Pä06] übernommen und angepasst.

```
1   You need the following to install RIP-XT-SL:

          - quagga-0.99.4.tar.gz
          - quagga-0.99.4_MTI_XT_SL.patch or quagga-0.99.4_XT_SL.patch

    1.Unpack the tarball with

          tar -xvzf quagga-0.99.4.tar.gz
```

```
2.Apply the patch with

        patch -p0 < quagga-0.99.4_MTI_XT_SL.patch

(This will give you a ripd with MTI and XT as well as SL capabilities)

OR

        patch -p0 < quagga-0.99.4_XT_SL.patch

(This will give you a ripd with XT and SL, but without MTI)

3.Enter the quagga source directory

        cd quagga-0.99.4

4.Due to changes in the Makefiles, you will need to run

        aclocal && libtoolize --copy --force

next. If either of these programs isn't available on your
system, install "automake" and "libtool" to get them.
If you run into problems, try to update both above mentioned
tools as well as "m4".

5.Finally, you can procede with configure and make as you
would in an unpatched quagga:

EASIEST:
        ./configure && make


CUSTOMIZED:
It is recommended to take a closer look at the output of
"./configure --help" and decide which features are needed and
which aren't.
An example might be:

        ./configure --enable-debug=full --disable-ipv6
                    --disable-ripngd  --disable-ospf6d
      --disable-ospfd --disable-bgpd --enable-user=root

In this configuration, debugging messages are enabled and most
unused daemons as well as IPv6 support are disabled.


6. and finally:
                make install
```

Einse detailliertere Installationsbeschreibung ist auf der CD zu dieser Arbeits enthalten.

A.4 Mounten von Filesystemen

Um die Quagga-Sources direkt im VNUML-Filesystem compilieren und installieren zu können, bietet es sich an, das entsprechende Filesystem zu mounten und dann in der Umgebung dieses Filesystems die Installation

vorzunehmen. Dazu erstellt man ein Verzeichnis in das das Filesystem gemountet werden kann, hier *vnumlmount*. Das jeweilige Filesystem liegt hier in */filesystems/*
Mit root-Rechten muss dann folgendes durchgeführt werden:

- `mount -o loop /filesystems/root_fs_tutorial /vnumlmount`
- `mount -t proc proc /vnumlmount/proc`
- `chroot /vnumlmount /bin/bash`