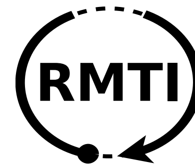
UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik



Improvements to the RMTI network routing daemon implementation and preparation of a public release

Diplomarbeit

zur Erlangung des Grades eines
Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Michael Monreal

203110029

Betreuer
Prof. Dr. Christoph Steigner (Institut für Informatik)
Dipl. Inf. Frank Bohdanowicz (Institut für Informatik)

Koblenz, im November 2010

Formales

Erklärung nach §10 Abs. 6

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Veröffentlichung

Ich erkläre mich damit einverstanden, dass diese Arbeit in digitaler und ausgedruckter Form von der Universität Koblenz-Landau öffentlich zugänglich gemacht wird.

Koblenz, 26. November 2010

Abstract

Routing with Metric based Topology Investigation (RMTI) is an algorithm meant to extend distance-vector routing protocols. It is under research and development at the University of Koblenz-Landau since 1999 and currently implemented on top of the well-known *Routing Information Protocol* (RIP).

RMTI aims to improve both convergency and scalability by making the underlying system insusceptible to *counting-to-infinity*, a problem which plagues all distance-vector routing protocols to this date. RMTI manages to achieve this goal without changing the base protocol, so compatibility is not sacrificed.

Around midyear 2009, the latest implementation of RMTI included a lot of deprecated functionality. Because of this, the first goal of this thesis was the reduction of the codebase to a minimum. Beside a lot of reorganization and a general cleanup, this mainly involved the removal of some no longer needed modes as well as the separation of the formerly mandatory XTPeer test environment.

During the second part, many test series were carried out in order to ensure the correctness of the latest RMTI implementation. A replacement for XTPeer was needed and several new ways of testing were explored.

In conjunction with this thesis, the RMTI source code was finally released to the public under a free software license.

Contents

Abstract **2**

1 Introduction and terminology **5**
1.1 Conventions 6
1.2 Structure of this thesis 7

2 Routing Information Protocol **8**
2.1 Fundametals 8
2.2 Problems with loops 10

3 Routing with Metric-based Topology Investigation **11**
3.1 History 11
3.2 Fundamentals 12
3.3 Detecting simple loops 13
3.4 Decision modes 13

4 Quagga software routing suite **15**
4.1 Overview 15
4.2 Basic usage 17
4.2.1 Configuration files 17
4.2.2 Telnet 17
4.2.3 VTY shell 17
4.3 Structure of the Quagga codebase 18

5 Implementation of RMTI **19**
5.1 Added files 19
5.2 Modified files 19
5.3 New data structures 20
5.4 Application flow 20

6 Modifications as part of this thesis **22**
6.1 Version control 22
6.2 Rebase RMTI on the latest Quagga release 23
6.3 Monotonic clock support 23
6.4 Removal of the XTPeer integration 24
6.4.1 XT-server 25
6.4.2 SL-client 25
6.5 Table output 25
6.6 Removal of deprected modes 26
6.7 Mode selection 27
6.8 Infinity metric 27
6.9 Adapting timers to loop size 27
6.10 Refactoring to meet Quagga coding guidelines 28
6.11 Reduction of code and patch size 28

7 Test environment **30**
7.1 Hybrid testing procedure . . . 31
7.2 Zimulator . . . 31
7.3 Testomato . . . 31
7.3.1 Configuration of Testomato . . . 32
7.3.2 Running Testomato . . . 32
7.4 Generator for counting-to-infinity situations . . . 34

8 Test results and evaluation **37**
8.1 Topology 1: upsilon . . . 38
8.1.1 Running the upsilon topology with host integration . . . 40
8.2 Topology 2: circle . . . 41
8.3 Topology 3: extended upsilon . . . 42
8.4 Final results and log files . . . 42

9 Public release **44**
9.1 License . . . 44
9.2 Website and discussion board . . . 44
9.3 Source code . . . 45
9.4 Binary packages . . . 45
9.5 File system image . . . 46
9.6 Live CD . . . 46

10 Conclusion **47**
10.1 Ongoing work . . . 47
10.2 Future of RMTI . . . 48

A User mode linux performance regression **49**

B Gitorious **51**

C Basic Git usage **53**

D Rebasing with Git **55**

E Quagga coding style **57**

F Quagga coding introduction and examples **59**

G Creating packages for Linux systems **61**

H Patch: RMTI support for Zimulator **64**

I Patch: Counting-to-infinity generator **65**

J Testomato bash script and scenario run files **68**

Listing **74**

List of Figures **74**

References **75**

1 Introduction and terminology

A *network* is a collection of nodes which can communicate with each other. It may consist of a number of *subnets* which have their own addressing scheme or even use totally different technology and communication media.

Subnets need to be interconnected by special nodes called *routers*. A router can have a large number of *interfaces* (normally at least two), each connected to a different subnet. Being part of all the subnets, the router needs to support all addressing schemes and technology used by the individual subnets.

Routers basicly need to handle two jobs: *forwarding* and *routing*.

Forwarding
: The process of taking a packet from one subnet and sending it to another. For this, a special *forwading table* is consulted. The table contains the next hop router for every known subnet as well as a default gateway.

Routing
: The process of building the forwarding tables. There are two basic kinds of routing: *exterior gateway* routing is used in between the large autonomous systems that represent the internet and *interior gateway* routing is used on the inside. As long as not mentioned otherwise, this thesis will always refer to the latter kind.

While routers may also work with manually set *static routes*, the case of dynamic routing using a *routing protocol* is more important. The implementation of such a protocl is not trivial, as the *routing algorithms* need to work in a fully distributed environment and cannot rely on any global knowledge.

This implies that routing is a highly self-organizing mechanism. Routers need to be able to react to changes in the network topology without any human intervention. The goal is to get into a *convergent* state as quickly as possible. This state is reached when newly arriving routing packets no longer result in changes to the routing- and forwarding tables on any router. In turn, it is then possible to reach all destinations from every point in the network.

One of the most interesting challenges a routing protocol has to deal with is the handling of *loops*. When talking about loops, three classes need to be distinguished [3]:

Topology loop
: A physical loop in the network topology.

Routing loop
: A loop in the routing process (routing packets move in circle).

Forwarding loop
: A loop in the forwarding process (data packets move in circle).

The easiest way to prevent loop-related problems is a loop-free network design. This, however, has a big downside: if one link fails, parts of the network are cut

off and no longer reachable. Redundancy improves the reliability of the network as traffic can move around broken links and use alternative paths if necessary. Because of this, topology loops are a deliberate part of most network designs and avoiding them would just create problems and risks of its own.

Routing loops occur when outdated knowledge is passed between routers over a loop in the topology. This kind of loop needs to be prevented because it delays the process of getting into a convergent state and can leave destinations unreachable for a long time. A routing loop can also lead to a forwarding loop. Data traffic circles around the loop, never able to get to the destination. In the worst case, the data packets fill up the whole link, bringing all communication to a halt.

Different routing protocols have their own ways to handle loops. The possible approaches are often determined by the family (distance-vector, link-state or path-vector) of the protocol. Every family has their own strengths and weaknesses [19].

There are some more terms describing connections inside a network which will also be used in the following sections:

Hop
: Travel of one section when moving from source to destination.

Path
: Connection between interfaces of a router over a sequence of hops.

Route
: Path that ends in the subnet beyond a destination router.

Route-combination
: A combination of two routes.

1.1 Conventions

The following highlighting conventions are used throughout this thesis:

- Commands: `foo bar`
- Files, functions, variables: *foo*
- Directories: *foo/*

While most of the console commands shown in the following sections can be executed with normal user rights, superuser rights are required in some cases. To differentiate, the following syntax is used:

- `$cmd`: The command `cmd` can be executed with normal user rights.
- `#cmd`: The command `cmd` needs to be executed with superuser rights.

Note: While `sudo` is the default for superuser execution on some systems, it does not always work as expected. Better use the traditional `su` command to minimize problems.

1.2 Structure of this thesis

The thesis is organized as follows:

Section 2
: introduces the reader to the classic routing protocol RIP. After giving an overview about the basic concept, the section details its shortcomings, focussing on the couting-to-infinity problem.

Section 3
: presents the RMTI algorithm as a solution to the problems mentioned in the previous section.

Section 4
: is about the Quagga routing suite.

Section 5
: describes the current implementation of RMTI based on the RIP daemon from Quagga.

Section 6
: goes into detail about the modifcations done as part of this thesis.

Section 7
: discusses possible replacements for the XTPeer test environment and describes the idea behind the new Testomato framework.

Section 8
: presents the test results of the latest RMTI release using a number of sample topologies.

Section 9
: discusses the measures taken to realize the public release of the RMTI implementation and its source code, going into detail on license and distribution-specific issues.

Section 10
: sums up the thesis and presents some conclusions.

The main part is followed by appendices which cover some practical topics like the Git version control system and Linux package management. Each appendix is referenced in the main part but can be read individually.

Attached to the printed version the reader will find two optical discs. One contains most of the data and code referred to in the text, the other offers a preconfigured live environment.

2 Routing Information Protocol

The *Routing Information Protocol*, better known as *RIP*, is a well-known interior gateway routing protocol. The initial version was developed during the 1960s, based on the Bellman-Ford algorithm. It soon became the first protocol to take care of the routing in the *ARPANET*, the network that would later become the *Internet* [19].

RIP was standardized by the IETF[1] in 1988 (RFC 1058[2]). Since then, the standard has been superseeded by RIP version 2 in 1998 (RFC 2453[3]). There is also a variant called RIPng, which is based on IPv6 instead of IPv4 (RFC 2080[4]).

Nowadays, the Internet is taken care of by the BGP protocol because it has grown far too big for RIP to handle. Even in the interior gateway sector, some newer protocols like Cisco IGRP and OSPF have taken over and are more widely used.

However, RIP is still an interesting protocol: it is very easy to understand and implementations can be really small. Also, it is very easy to set up as almost no configuration is required.

2.1 Fundametals

The RIP protocol belongs to the family of *distance-vector* routing protocols. The basic idea is that every router shares all of its knowledge with all directly connected neighbors. After some time, every node has constructed a list containing all reachable networks. This list does *only* contain information about the next hop for every possible destination network, the complete path is *not* known on a single router.

The packets exchanged between the RIP daemons contain information in the form of the triple *(destination, next hop, metric)*:

Destination
Address of the destination network.

Next hop
Address of the next router on the path to the destination.

Metric
Number of hops to the destination, ranging from 1 to 16.

The number of hops is actually limited to 15. The metric of 16 is often referred to as *infintiy* and reserved for a special purpose: it declares the network as *unreachable*. Furthermore, the protocol differentiate between *triggered updates* and *timed updates*:

[1] IETF = **I**nternet **E**ngineering **T**ask **F**orce - `http://www.ietf.org/`
[2] RFC 1058: `http://www.ietf.org/rfc/rfc1058.txt`
[3] RFC 2453: `http://www.ietf.org/rfc/rfc2453.txt`
[4] RFC 2080: `http://www.ietf.org/rfc/rfc2080.txt`

Triggered updates
Sent after a change of the own routing table to quickly propagate the new information.

Timed updates
Sent periodically based on a fixed timer.

Update packets can contain a maximum of 25 route information triples. The handling of all routing information is controlled by three internal timers:

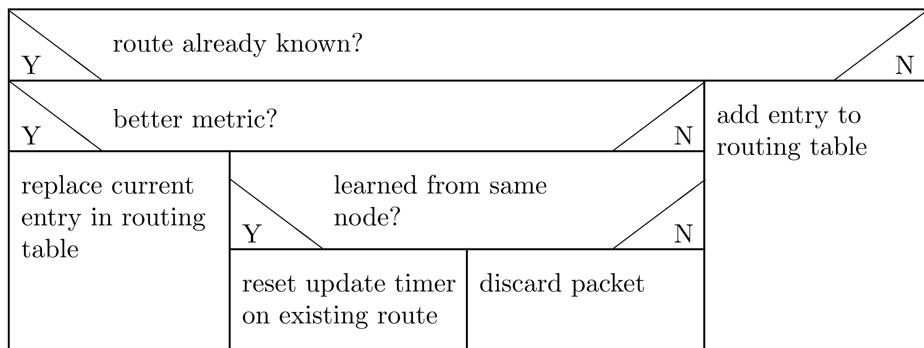
Update timer
Triggers sending of *timed updates* (defaults to 30 seconds).

Timeout timer
Marks routes which have not been confiremd during the update interval as unreachable (defaults to 180 seconds).

Garbage collection timer
Purges unreachable routes from the routing table (defaults to 120 seconds).

After starting up, RIP routers first advertiese themselves to their neighbors. As soon as a router receives a RIP update packet, it checks the information stored in the contained route triples.

Descision process — What happens with the received information depends on a few factors. The basic decision process is visualized in the following Nassi-Schneiderman diagram:



Before adding a new or better route to the routing table, the metric received as part of the update packet is first incremented by one.

Every route needs to be reconfirmed periodically to assure that it is still valid. Because of this, every entry in the routing table is connected to its own timeout timer. As the route gets reconfirmed, this timer is reset to the initial value. If it ever reaches zero, the router assumes that the route is no longer valid. It sets the metric to infinity, sends out triggered updates and starts the garbage collection timer for the route. The route is not deleted completely until this timer runs out. If some update reconfirms the route during this time, the garbage collection timer is stopped and reset. The route is marked valid and the timeout timer is started again.

2.2 Problems with loops

As mentioned in the introduction, the handling of loops is a major concern of a routing protocol. For a small example, imagine a RIP network containing a simple two-hop loop as shown in figure 1.

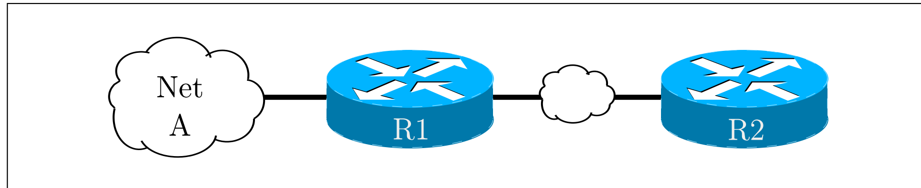


Figure 1: Topology with two-hop loop

Router R1 is connected to a network A and has passed on this knowledge to R2. Now R1 loses the connection to network A, sets the metric to infinity and informs R2. However, this update does not reach R2 in time before R2 sends out a periodic update containing the now invalid route. R1 accepts the route because it looks valid. It sends out the route again. As R2 had originally learned the route from R1, it will accept the new metric now even if it is higher. The same happpens on R1 after the next update from R2 arrives, as the route stored there seemingly originates from R2. The two routers now exchange updates about this invalid route, slowly increasing the metric each time until it finally reaches the infinity value. This problem is known as *counting-to-infinity* or *CTI*.

Apparently, the counting-to-infinity disrupts routing and delays convergency. It also affects the forwarding process: during the duration of the CTI, network nodes continue to send data packets to the seemingly reachable network. Those packets are passed through the loop, slowing down the network and finally taking up all the bandwidth and bring everything to a halt.

To work around the problem, the *split horizon* rule was introduced in a later revision of RIP. The idea is that a router should never send back information about a route to the node it originally learned the route from. Abiding to this rule prevents CTI situations between exactly two neighbors, such as the example described above. However, real loops normally contain more than two routers. In such a case, the split horizern rule does not help to prevent the CTI at all.

Even today, the RIP standard does not offer a better mechanism for handling loops. Counting up to the infinity value is still the only way to regain a consistant state in many cases.

3 Routing with Metric-based Topology Investigation

Routing with Metric-based Topology Investigation (just called *RMTI* below) is a project being worked on by the University of Koblenz-Landau. It aims to solve the aforementioned problems of classic distance-vector protocols in general and the RIP protocol in particular. At the same time, RMTI is meant to be fully backward compatible by not changing or extending the already existing protocol messages in any way.

3.1 History

Being in development since 1999, the RMTI project has evolved from a basic theory to a matured implementation over the years. During this time, it has been worked by many individuals:

1999
: Andreas Schmid [27] introduced the basic concept behind RMTI, called *RIP-MTI* (*Routing Information Protocol with Minimal Topology Information*) at that time.

2001
: Thomas Kleeman [15] evaluated and improved the RIP-MTI concept.

2005
: Tobias Koch [16] implemented an experimental RIP-MTI daemon on top of the RIP daemon shipping with the Quagga routing suite.

2006
: Daniel Phäler [26] and Stefan Lange [17] created a test environment called XTPeer. For this, implementations of the XT-server and SL-client were added to the experimental daemon.

2007
: Tim Keupen [13] further improved XTPeer with different ways to generate counting-to-infinity situations. Using this environment, the RIP-MTI implementation was tested in many exemplary topologies.

2008
: Frank Bohdanowicz [1] evaluated different modes of RIP-MTI and documented various shortcomings. Some of the modes were declared deprecated and noted for possible removal in the future.

2009
: The RIP-MTI project was renamed to RMTI and paper [3] was released and presented at the IARIA conference. A decision for a public release of the RMTI source code was made.

2010
: Another paper [4] based on previous work of Marcel Jakobs [12] was published, showing some of the strong points of the RMTI algorithm compared to plain RIP. Finally, the RMTI code was released to the public as part of this thesis (see section 9).

3.2 Fundamentals

Reduced to the very basics, the idea behind RMTI is to make better use of the information already exchanged by the underlying distance-vector protocol. By not just rejecting or overwriting certain data, RMTI manages to build a broad knowledge base. Being able to access this extra information allows the algorithm to draw conclusions about routing loops in the proximity of the node. In turn, this knowledge allows RMTI to rule out certain incoming routing information and prevent the counting-to-infinity problem this way.

For that matter, RMTI distinguishes between two types of routing loops: *simple loops* and *source loops* (both illustrated in figure 2).

Simple loop
: A path leaving a router on one interface and reentering it on another interface without passing through this same router in between.

Source loop
: A path leaving a router on one interface and reentering it on another interface after passing through this same router in between.

The difference between a simple loop and source loop has one important implication. Assume a destination network located somewhere on a simple loop. The path that leads through this simple loop can be devided into two routes, both connecting the origin router to the destination. If there is a failure on one of these routes, the other route can still be used to reach the destination. Hence, an alternative route to the destination exists. While this is always true for paths forming a simple loop, it is not true for paths forming a source loop.

The realization that only simple loops supply an alternative route to a given destination can be used by a router to decide whether some offered route can safely be accepted or not. In doing so, the router is able to handle situations which would normally lead to counting-to-infinity.

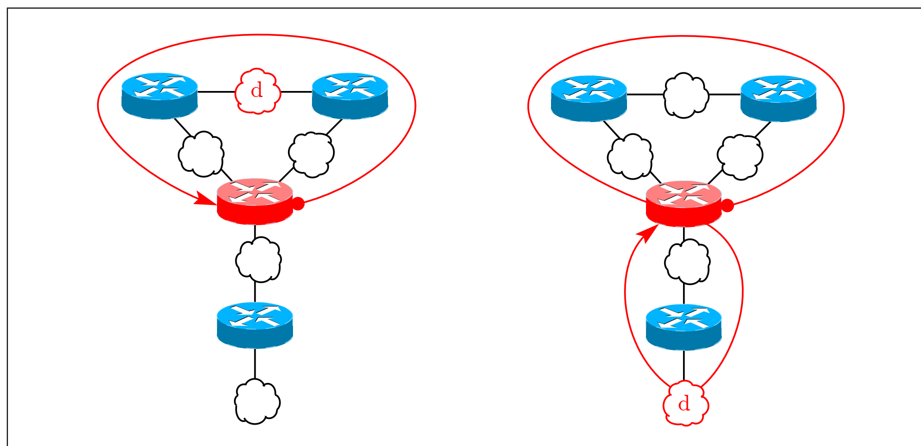


Figure 2: A simple loop and a source loop [3]

3.3 Detecting simple loops

Given a bird's eye view on the network topology, it is not hard to find loops and classify them into simple loops and source loops. However, the routers need to make the same decisions *without* any kind of global knowledge. Because of this, RMTI collects and maintains a larger amount of information compared to plain RIP. This knowledge can later be used to form a decision.

Formally, a simple loop can be defined as a path originating from router i, traversing a subnet d and coming back to router i without crossing it in between. The metric m of a simple loop is referred to as *silm*. Besides this, two more definitions are needed [3].

The *minimal simple loop metric* between two interfaces A and B of router i is

$$msilm^{i}_{A,B} = min\{silm^{i,d,i}_{A,B}\ of\ all\ subnets\ d\}$$

and the *minimal return path metric* on an interface A of router i is

$$mrpm^{i}_{A} = min\{msilm^{i}_{A,B}\ of\ all\ interfaces\ A \neq B\ of\ router\ i\}$$

Based on this groundwork, consider two different paths $P^{i,d}_{A}$ and $P^{i,d}_{B}$ leading to a subnet d, respectively with metrics $m^{i,d}_{A}$ and $m^{i,d}_{A}$. The following *simple loop test* (short *SLT*) can be used to detect a simple loop:

$$m^{i,d}_{A} < mrpm_{A} + m^{i,d}_{B}$$

If this inequation does not hold, the path is *too short* to be a source loop and hence must be a simple loop. Furthermore it has been shown [2] that a combination of this test and the well-known split horizon rule manage to prevent the acceptance of bad routing updates and thus prevent the counting-to-infinity.

3.4 Decision modes

How the result of the simple loop test is used to form a routing decision depends on what implementation of RMTI is used. In order to solve problems with previous versions, the experimental RMTI daemon had grown a number of *modes* to choose from. While section 6.6 will present some significant simplifications in the latest RMTI, it is still important to describe the background and development. In the experimental RMTI daemon, the most important modes are known as *NORMAL*, *STRICT* and *CAREFUL*.

NORMAL

Rejects all routes which do not pass the SLT.

The NORMAL mode represents the initial implementation. It assuems that every route which passes the test is automatically valid. This, however, is not the case: there are routes which pass the test but are invalid nevertheless. The NORMAL mode accepts these routes and is not able to prevent the counting-to-infinity in such a case.

STRICT
: Accepts only routes which pass the SLT.

The STRICT mode inverts the condition in order to ensure the prevention of the counting-to-infinity problem. It disregards the fact that there are also valid routes which do no pass the test. The STRICT mode just rejects these routes as well.

It was later shown that this property makes the STRICT mode unsuitable for serious usage: while the counting-to-infinity is averted successfully, the rejection of valid routes can delay general convergency noticeably [1].

CAREFUL
: Directly accepts only routes which pass the SLT but rejects the rest only temporarily.

The CAREFUL mode extends the STRICT mode with a compromise. For routes which cannot be accepted right away, the rejection is no longer final. It only holds for the critical period in which a counting-to-infinity is likely to happen. In such a case, the CAREFUL mode sends a routing update carrying the infinity metric through the possible loop, starts a timer and then listens for incoming updates of the same route on the same interface. If during this time, the route is only received with a metric of infinity, it can safely be rejected. If, however, the route is received with a valid metric again, the route is actually valid and needs to be accepted.

4 Quagga software routing suite

The current implementation of the RMTI daemon is not built from scratch. It is a modification of the RIP daemon shipping with the popular routing suite *Quagga*[5]. The Quagga project itself is the continuation of the *GNU Zebra*[6] project, which is inactive since 2005. Like Zebra before it, Quagga aims to produce free software implementations of some well-known network routing protocols. To date, Quagga includes implementations of RIP and OSPF (both in IPv4 and IPv6 variants) as well as BGPv4 and ISIS [24].

As all Quagga daemons make use of a common framework, this section first describes the underling concept before the RMTI implementation itself is discussed in the next section.

4.1 Overview

All Quagga daemons require an additional daemon to run. For historical reasons, this daemon is still called the *Zebra core daemon*. The binary is also still called `zebra`. It acts as a hub between the individual routing daemons and the host system. More precisely, its main function is to keep the routing tables of all running protocol-specific Quagga daemons in sync with the single forwarding table of the host system kernel.

All Quagga daemons are divided into various modes, which themselves are anchored to a number of predefined *nodes*:

VIEW_NODE
: This is the initial mode which can be used to query only basic information.

ENABLE_NODE
: This is the privileged command mode which can be used to send control commands to the daemons.

CONFIG_NODE
: This mode allows the user to change settings which are common to all Quagga daemons.

RIP_NODE
: This mode is only available in the RIP daemon and allows you to change settings specific to RIP. Other protocols offer a similar node.

All possible transitions are illustrated in figure 3 on the next page. Starting from the VIEW_NODE, the `enable` command changes into ENABLE_NODE. From there, `configure terminal` changes to CONFIG_NODE where `router rip` changes to RIP_NODE. The `exit` command can normally be used to go back one level. However, in ENABLE_NODE the `disable` command needs to be used to get back to VIEW_NODE, as `exit` drops out of the system here.

[5]Quagga project - `http://quagga.net/`
[6]GNU Zebra project - `http://www.zebra.org/`

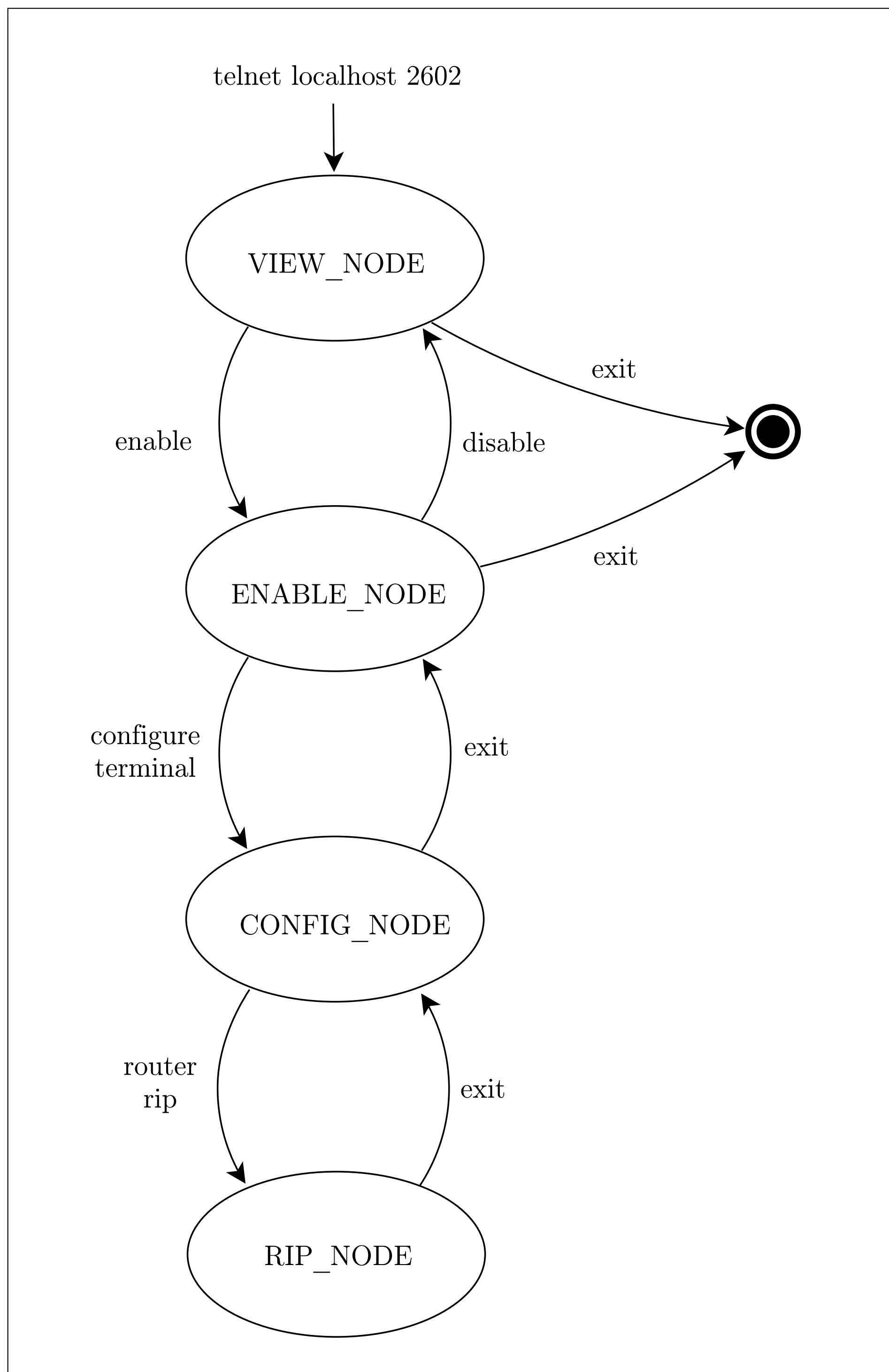


Figure 3: Transitions between modes in Quagga daemons

4.2 Basic usage

There are tree ways to interact with the Quagga daemons: configuration files, Telnet and the VTY shell. For detailed information about the syntax and all possible commands, please refer to the Quagga manual [22] which can be found on the web.

4.2.1 Configuration files

The easiest way to control a Quagga daemon is to provide a small configuration file. This file is parsed during startup and provides the initial settings. Zebra uses the file *zebra.conf* (sample shown in listing 2) and the RIP daemon uses *ripd.conf* (sample shown in listing 1).

```
1 hostname zebra
2 password xxxx
```

Listing 1: Sample of a zebra.conf file

```
1 hostname ripd
2 password xxxx
3
4 router rip
5 network 10.0.0.0/8
6
7 timers basic 10 30 20
```

Listing 2: Sample of a ripd.conf file

Other filenames can also be used. For this, the daemons need to be started with the *-f filename* flag.

4.2.2 Telnet

The Quagga daemon can be accessed by Telnet, which is useful if a running daemon needs to be reconfigured or queried for various information.

The Zebra daemon is running on port 2601 by default:

```
$ telnet localhost 2601
```

The RIP daemon is running on port 2602 by default:

```
$ telnet localhost 2602
```

4.2.3 VTY shell

A third way to control the Quagga daemons is the VTY (virtual teletype terminal) shell tool, `vtysh`. It can be used to send a single command to a daemon:

```
$ vtysh -c "show ip rip"
```

In a similar way, a single line can be used to send a sequence of commands. For this, individual commands are separated using a semicolon:

```
$ vtysh -c "configure terminal; router rip; something"
```

The commands used here are the same as used with Telnet. The `enable` command is not needed as the VTY shell already starts in the privileged command mode (ENABLE_NODE) by default.

4.3 Structure of the Quagga codebase

At the top level, the Quagga codebase is organized into different directories as shown in listing 3.

```
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:13 bgpd
drwxr-xr-x. 4 monreal monreal   4096 2010-07-10 13:25 doc
drwxr-xr-x. 6 monreal monreal   4096 2010-06-19 21:13 isisd
drwxr-xr-x. 4 monreal monreal   4096 2010-07-11 16:39 lib
drwxr-xr-x. 3 monreal monreal   4096 2010-06-19 21:13 m4
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:14 ospf6d
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:13 ospfclient
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:14 ospfd
drwxr-xr-x. 3 monreal monreal   4096 2010-06-19 21:13 pkgsrc
drwxr-xr-x. 3 monreal monreal   4096 2010-07-13 13:08 redhat
drwxr-xr-x. 4 monreal monreal   4096 2010-09-28 15:48 ripd
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:13 ripngd
drwxr-xr-x. 3 monreal monreal   4096 2010-06-19 21:13 solaris
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:13 tests
drwxr-xr-x. 3 monreal monreal   4096 2010-06-19 21:13 tools
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:13 vtysh
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:14 watchquagga
drwxr-xr-x. 4 monreal monreal   4096 2010-06-19 21:14 zebra
```

Listing 3: Structure of the Quagga source code

These directories contain the implementations of routing protocols (*bgpd*, *isisd*, *ospf6d*, *ospfd*, *ripd*, *ripngd*), shared code (*lib*), documentation (*doc*), the build- and test systems (*m4*, *pkgsrc*, *redhat*, *solaris*, *tests*) and related tools (*ospfclient*, *tools*, *vtysh*, *watchquagga*). Finally, the *zebra* directory includes the sources of the Zebra core daemon.

For the work on RMTI, the subdirectories *lib/*, *ripd/* and *zebra/* are the most important:

lib/
: Contains the shared library code that forms *libzebra.so*.

ripd/
: Contains the code of the RIP daemon that forms the executable `ripd`.

zebra/
: Contains the code of the zebra daemon that forms the executable `zebra`.

5 Implementation of RMTI

This section aims to describe the current implementation of RMTI. As already mentioned, the current code is heavily based on the existing Quagga RIP daemon, so most of the modifications and additions can be found in the files inside the *ripd/* directory of the Quagga source tree, shown in listing 4.

```
-rw-rw-r--. 1 monreal monreal  40843 2010-09-17 13:07 ChangeLog
-rw-rw-r--. 1 monreal monreal    653 2010-09-17 13:07 Makefile.am
-rw-rw-r--. 1 monreal monreal 124572 2010-10-12 12:36 ripd.c
-rw-rw-r--. 1 monreal monreal    406 2010-09-17 13:07 ripd.conf.sample
-rw-rw-r--. 1 monreal monreal   7362 2010-09-17 13:07 rip_debug.c
-rw-rw-r--. 1 monreal monreal   1817 2010-09-17 13:07 rip_debug.h
-rw-rw-r--. 1 monreal monreal  12603 2010-09-28 12:59 ripd.h
-rw-rw-r--. 1 monreal monreal  51893 2010-09-17 13:07 rip_interface.c
-rw-rw-r--. 1 monreal monreal   1316 2010-09-17 13:07 rip_interface.h
-rw-rw-r--. 1 monreal monreal   6987 2010-09-30 12:57 rip_main.c
-rw-rw-r--. 1 monreal monreal  28797 2010-10-02 09:53 rip_mti.c
-rw-rw-r--. 1 monreal monreal   3651 2010-10-02 09:53 rip_mti.h
-rw-rw-r--. 1 monreal monreal  11015 2010-09-17 13:07 rip_offset.c
-rw-rw-r--. 1 monreal monreal   4724 2010-09-17 13:07 rip_peer.c
-rw-rw-r--. 1 monreal monreal  28502 2010-09-17 13:07 rip_routemap.c
-rw-rw-r--. 1 monreal monreal  18782 2010-09-17 13:46 rip_snmp.c
-rw-rw-r--. 1 monreal monreal  17526 2010-09-17 13:07 RIPv2-MIB.txt
-rw-rw-r--. 1 monreal monreal  17578 2010-09-17 13:07 rip_zebra.c
```

Listing 4: Files inside the *ripd* directory

5.1 Added files

Compared to a plain Quagga source tree, there are only two additional files inside the *ripd/* directory:

rip_mti.c
: Contains the implementation of the RMTI algorithm.

rip_mti.h
: Contains the declaration of data structures and internal and external function prototypes used for RMTI.

5.2 Modified files

While the majority of the RMTI implementation can be found in *rip_mti.c*, some files containing Quagga RIP code are also modified:

rip_main.c
: Contains the main() function of `ripd` and related startup logic. Initialization code for RMTI was added here.

ripd.c
: The Quagga implementation of the RIP algorithm, containing declarations of important structures like *rip_info* as well as the DEFUNs which define the commands used to control the daemon. The code was modified to call RMTI during the route decision process and new RMTI-specific DEFUNs were added.

ripd.h

Contains the declaration of data structures as well as internal and external function prototypes used for RIP. New RMTI-specific functions, structures and variables were added here.

5.3 New data structures

Compared to a plain RIP implementation, the RMTI concept relies on additional data mainly in two new structures: the *MSILM-table* and the *MRPM-table*, respectively storing the minimal simple loop and minimal return path metrics, introduced in section 3.3. Both are defined in *rip_mti.h*.

MSILM-table (*vector msilm_table*)

Lists pairs of interfaces which correspond to the smalles simple loop between those interfaces.
Answer to: *"Are there any loops between a given pair of interfaces?"*
The table is quadratic and symmetric and consists of vectors of *struct mti_msilm_entry*.

MRPM-table (*vector msilm_table*)

Lists the minimal return path of separate interfaces.
Answer to: *"What is the shortest loop on a specific interface?"*
The table is linear and consists of vectors of *struct mti_mrpm_entry*, referencing the values in the MSILM-table.

At startup, both tables are initialized with *msilm_null_entry*, which contains the *RMTI infinity* value reduced by one. This special infinity value is based on RIP infinity. It is calculated by adding the infinity of a route and a potential alternative route leading to the same subnet, which in turn equals 2*(RIP infinity).

The actual look of both tables is covered later in section 6.5.

5.4 Application flow

The RMTI algorithm is inserted into the RIP routing process in *ripd.c*. The RIP code analyzes every incoming packet and checks if the contained destination is already known. If this is not the case, the route to the new subnet is added to the routing table. If, however, the destination is already known, plain RIP would only check if the new route has a better metric and accept or reject it based on the result of this simple test. With RMTI, a new code path is taken at this point.

The code that is now executed can be devided into two separate parts [3]: the *data collection process*, followed by the *decision process* [3]. Both parts are mainly implemented in *rip_mti.c*.

Data collection

The data collection phase is about building and maintaining the knowledge that RMTI needs in order to make its decisions. If the incoming information is valid, the simple loop test, realized by the *simple_loop_test()* function, is run. If the test is passed, the metric of the underlying simple

loop is calculated. If the new value is smaller than what is already known, the MSILM-table (and in turn the MRPM-table) is updated with the new metric.

Decision

The decision phase makes use of all the extra knowledge stored by the data collection process to identify routing loops. A cascade of tests is run on the route, eventually leading to another simple loop test. There are two different points of exit depening on the outcome of the tests.

After running through the data collection and decision phases, RMTI is able to hand over the route back to the RIP algorithm. If RMTI has identified the route to be invalid, it is directly rejected by RIP. A route that passes all RMTI tests is considered to be valid and thus reinserted into the RIP decision process. Here, the route is rejected if it is worse (higher metric) than the already stored one. If the new route is better (smaller metric), it can replace the stored route.

6 Modifications as part of this thesis

The basic concept behind RMTI and the algorithm itself had proven themselves both in the laboratory as well as in front of the international science community on various conferences on networking by 2009.

As the next big step, it was decided to release the code to the public in order to gather more feedback and encourage real-world deployment. However, the current implementation at that point was not very clean and still carried a few known bugs. Consequently, some cleanups and simplifications needed to be done.

This section describes the modifications done in order to get the implementation into a distributable state.

6.1 Version control

Even in medium-sized software project it is essential to keep track of all changes made to the code. Traditianally, this is done using centralized configuration control systems like the *Concurrent Version System* (CVS) or *Subversion* (SVN). The Quagga project is using a *distributed* system called *Git*, which is also used to manage development of the Linux kernel since its creation as a *BitKeeper* replacement in 2005.

When forking code, using the same configuration control system as the parent project generally makes a lot of sense. In the case at hand, using Git allows to automatically port RMTI to a newer Quagga release in a matter of seconds instead of having to port forth everything manually. Patches developed for RMTI can easily be carried over to Quagga and vice versa.

To make working with a shared Git repositories more comfortable, a *Gitorious*[7] instance was set up for projects of the University of Koblenz-Landau. Gitorious is a web application which allows to create and administer Git repositories organized in project groups using a simple user interface.

The RMTI project[8] contains a repository called *rmti*, which in turn contains various branches. The *master* branch is just a mirror of the upstream Quagga sources including the release tags (*quagga_x_y_z_release*). This allows to easily check out new Quagga versions and rebase the RMTI code on top of them. All RMTI branches are named after their Quagga base version (*r_x_y_z*).

For more information about the topic, consult appendix B, which explains how to access the RMTI code repository on Gitorious and how to work with it. In addition, appendix C provides a reference sheet containing the most commonly needed Git commands.

[7] Gitorious - `http://gitorious.org/`
[8] RMTI project - `http://git.uni-koblenz.de/rmti/pages/Home`

6.2 Rebase RMTI on the latest Quagga release

The initial RMTI implementation was based on Quagga 0.99.4. At some point, it was manually ported to the 0.99.6 release. At the time of this writing, Quagga is already up to 0.99.17.

Fortunately, using the Git version control system for RMTI makes porting to a newer base version a very easy task. Git offeres a tool called *rebase*. The general idea behind the rebase process is shown in figure 4.

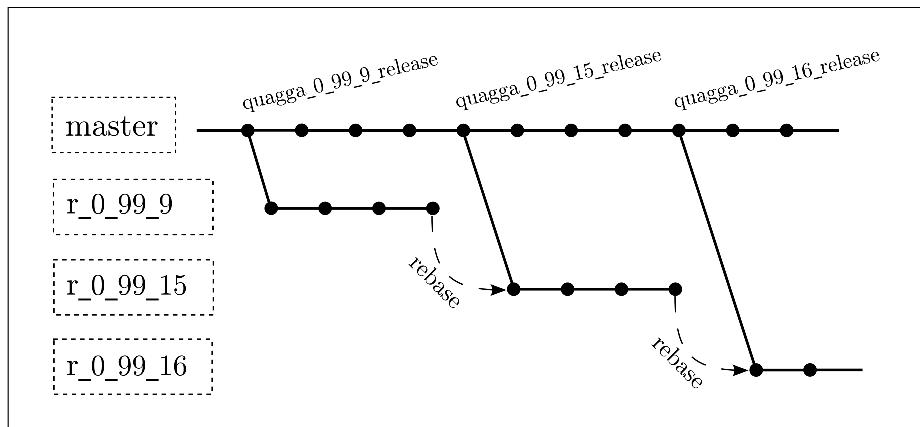


Figure 4: Branch history of the RMTI repository

For more information regarding the rebase operation and a detailed example, please consult appendix D.

6.3 Monotonic clock support

After rebasing the code to Quagga 0.99.16, the RIP daemons did no longer exchanged any information. The problem could not be related to the RMTI patch, as it also affected ordinary Quagga 0.99.16 builds.

After narrowing down the regression range to a single commit it turned out that the problem was caused by the *monotonic clock support*. This optional code path was already introduced in 2006, however a broken makefile caused it to always be disabled until the 0.99.16 release finally fixed the makefile (see `git show 9964fcf` for details).

In any case, the monotonic clock support is only useful for the BGP and OSPF daemons, which both track system time in some way. The RIP daemon does not profit from this as all as RIP timers simply count seconds and work independantly of the system clock.

After filing a bug[9] report in the Quagga bug tracker, some additional testing showed that the problem only occurs inside our virtualized testing environment. This indicates that monotonic clock support is broken in either the User Mode

[9] Quagga bug #592: `https://bugzilla.quagga.net/show_bug.cgi?id=592`

Linux kernel or in combination with μClibc[10], which is used inside the virtual machine file system. Unfortunately, the Quagga developers already stated that supporting either is not a high priority.

As a workaround, it is possible to just revert commit *9964fcf*:

```
$ git revert+ \verb+9964fcf
```

Alternatively, monotonic clock support can be disabled in the build system by running the following command after `./configure`:

```
$ sed -i "/HAVE\_CLOCK\_MONOTONIC/d" config.h
```

This will delete the line containing HAVE_CLOCK_MONOTONIC from *config.h* and result in a working build, matching the behavior of previous releases.

6.4 Removal of the XTPeer integration

Work on the XTPeer framework started in 2006. The goal was to create a test environment able to visualize the current state of the network in realtime and offer the means to remote-control various aspects and settings of the routing daemons. More precisely, XTPeer allows to trigger the counting-to-infinity problem with just a few mouse clicks and the result is shown in a well-arranged metric graph at once.

On the technical side, the system makes use of the client/server approach to allow communication between the routing daemon and a graphical desktop application written in Java.

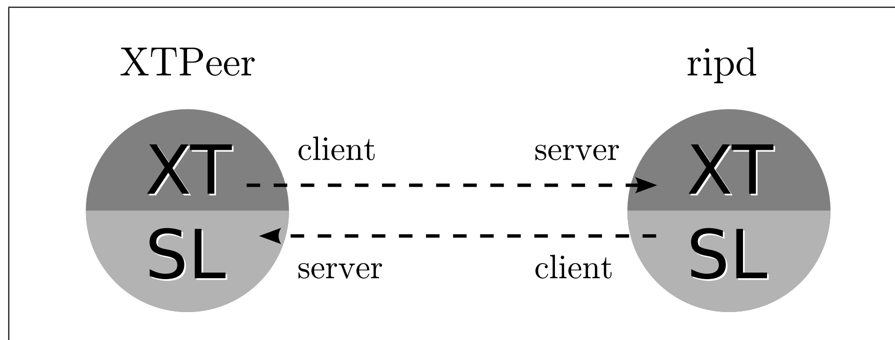


Figure 5: XTPeer client/server architecture

While being very helpful for both testing and presentations, the XTPeer integration repesented a big part of the delta between Quagga RIP and RMTI. There was no way to disable it at build-time and a connection to the XTPeer desktop application was always required at run-time. Removing the code specific to XTPeer allowed to simplify the code and makes it more applicative for productive use.

[10] μClibc = C library for embedded systems - `http://www.uclibc.org/`

6.4.1 XT-server

The XT[11]-server is built into `ripd`. It communicates with the XT-client, which is a part of the XTPeer desktop application. The server listens for control commands and executes them. This basicly substitutes the need to log in using Telnet.

The server had mainly been implemented inside *rip_xt.c*, so removing the integration mainly consisted of taking this file as well as the header file *rip_xt.h* out of the build system. After that, all calls to the XT* functions scattered around the rest of the RMTI code needed to be removed or replaced.

In a few instances, XTPeer used to initialize some variables and counters with values different to the Quagga defaults. These values are now all correctly initialized inside the daemon itself.

6.4.2 SL-client

The SL[12]-client is also built into `ripd`. It was implemented by Stefan Lange as documented in [17].

The client communicates with the SL-server, which is a part of the XTPeer desktop application. The duty of the client is to send status information to the server for further analysis.

Removal of the SL-client was done in a similar way to the XT-server removal. This time, the files *sl_client.c* and the accompanying header *sl_client.h* needed to be removed and all calls to the SL* functions needed to be replaced.

6.5 Table output

With the XTPeer integration removed, there was no longer a way to check the MRPM and MSILM tables within RMTI. The daemon commands `show ip rip mrpm` and `show ip rip msilm` were implemented to resurrect this functionality. They are both available for use in Telnet an VTYSH.

The MRPM table (see listing 5) is diplayed as a simple linear table. Each line starts with an index number and contains the IP-address and name of the interface, as well as the metric for the given prefix and the time since the last update.

```
MTPM-TABLE:

Index  IP-Address   Interface  MRPMetric   MRPMPrefix   Last Update
    1    10.0.1.1        eth1          3   10.0.3.0/24        00:07
    2    10.0.2.1        eth2          3   10.0.6.0/24        00:04
    3     0.0.0.0        eth1          3   10.0.2.0/24        00:04
    4    10.0.7.2        eth3          3   10.0.6.0/24        00:04
```

Listing 5: MRPM Table

[11] XT = e**X**ternally **T**riggered extension
[12] SL = **S**tatus **L**ogging extension by **S**tefan **L**ange

In reality, the MSILM table is implemented as a quadratic table. To make sure the output fits into a normal terminal window, it is drawn as a flat table (see listing 6). Each line represents one cell of the table. The position of the cell in the original table is give by the index numbers, matching interface names and IP-addresses. Additionally, each line provides information about the metric, prefix and time since the last update.

```
MSILM-TABLE:

IndexA    |IndexB    IP-AddressA |IP-AddressB   M-M   MSILIMPrefix   L-Upd
 1(eth1) |  2(eth2)     10.0.1.1|     10.0.2.1    3   10.00.3.0/24   00:03
 2(eth2) |  3(eth1)     10.0.2.1|      0.0.0.0    3   10.00.7.0/24   00:08
 1(eth1) |  4(eth3)     10.0.1.1|     10.0.7.2    5   10.00.4.0/24   00:08
 2(eth2) |  4(eth3)     10.0.2.1|     10.0.7.2    3   10.00.6.0/24   00:08
 3(eth1) |  4(eth3)      0.0.0.0|     10.0.7.2    3   10.00.2.0/24   00:08
```

Listing 6: MSILM Table

Appendix F.3 describes what needs to be done in order to add commands like these to a Quagga daemon and shows a small example.

6.6 Removal of deprected modes

During development of RMTI, various experimental modes were added in order to fix problems of the initial implementation. In 2009, the number of modes had already grown to six, some of which had already been proven either faulty or inefficient:

LISTEN
: This mode was only used for internal testing. The daemon builds and maintains the MSILM and MRPM tables but does not actually use the collected informations for its routing decisions.

STRICT
: This mode does only accept routes that have been verified as valid. Because of this, a lot of valid but not yet veriefied routes are rejected. While the counting-to-infinity can be avoided successfully this way, general convergence time was negatively affected.

CAREFUL_DT
: This mode was one experimental implementation to fix the problems of STRICT, based on timer to delay the acceptance of new routes.

CAREFUL_ESH
: This mode was another experimental implementation to fix the problems of STRICT, making use of the external split horizen.

The inital plan was to keep both of the remaining modes, **NORMAL** and **CAREFUL_RT** (both described in section 3.4). Depending on its place in a given topology, each router was meant to choose the best mode automatically at runtime. Unfortunately, finding a universal criterion for this decision proved to be harder than anticipated. After consideration, it was decided that the NORMAL mode could be removed as well, as the CAREFUL_RT mode can also handle those situations.

6.7 Mode selection

It is now possible to choose the RMTI mode using the configuration file. Because the CAREFUL_RT mode is the only remaining mode in the latest release, the newly added keyword `mti` only takes two values:

- `mti 0` : use plain RIP (default)
- `mti 1` : use RMTI support

To find out which mode is active on a running node, the **`show ip rip mti`** command can be used.

6.8 Infinity metric

In the original Quagga code, the infinity metric is hardcoded to a constant with the default value of 16. There is a preprocessor #define directive which can be used to change the default at compile time. Unfortunately, this does not take care of every single use of the infinity metric: the number 16 is used literally in some places instead of the constant. The first challenge was to identify all of these places. In a second step, the code was refactored to make use of a variable instead of the constant.

The new RMTI now allows changing the infinity metric using a configuration option `infinity n`, which can be set as part of the `router rip` section in *ripd.conf*. While n can be a huge integer number, it does not make sense to set this to a much larger value than what the network actually needs. For networks that are not likely to grow in size, setting the metric to the longest possible path plus one should be sufficient.

Theoretically this value can be changed at runtime. In practice it is highly recommended to only use it within *ripd.conf* to prevent inconsistencies. Note that all RIP nodes inside a topology need to use the same infinity value at all times.

To find out which infinity metric is set on a running node, the **`show ip rip infinity`** command can be used.

6.9 Adapting timers to loop size

Some tests done before this thesis already showed a problem concerning the garbage collection time in networks with large loops, such as the *Bigloop* topology from [13].

In such a case, it is possible that essential data is garbage collected even before an update has passed through the loop once. The result is that the loop detection mechanism does no longer work and a counting-to-infinity can occur.

To circumvent this problem, the following change was done to the RMTI algorithm: instead of blindly using the garbage collection time set by the user in the configuration file, a new value based on the actual loop size is calculated. If the provided value is smaller, the calculated value is used instead (listing 7).

```
unsigned int maxsilm = get_max_loop_metric (rinfo->mti_newindex) * 5;

if (maxsilm > rip->garbage_time)
  rip->garbage_time = maxsilm;
```

Listing 7: Adaption of the garbage collection timer

Additional tests runs revealed that a larger garbage collection time is only one part of the fix. The period of time where routes are not accepted (*careful_value*) needed to be adapted in a similar way (listing 8).

```
if (maxsilm > mti_state->careful_value)
  mti_state->careful_value = maxsilm;
```

Listing 8: Adaption of the execution time

With these modifications in place, the RMTI algorithm manages to reliably precent the occurance of CTIs in topologies that contain larger loops. However, it also means that valid routes can be hold back for an extended period of time, potentially leading to a longer convergency phase.

This new problem is not solved yet. It mostly manifests itself in huge networks with loop sizes way beyond the normal RIP infinity value of 16. The ill effect on the initial convergence time in large topologies has already been mentioned in the thesis [10] of Andreas Garbe.

As a possible solution, a non-linear increase of the *careful_value* could be considered. However, assuring the counting-to-infinity is still prevented reliably could become hard in that case.

6.10 Refactoring to meet Quagga coding guidelines

The Quagga project has agreed on a number of guidelines regarding their coding style. Basicly, these guidelines request code to be written in *GNU style with tabs set to two spaces.* As part of the ongoing process to make the RMTI implemtation available to the public, following these guidelines seemed to be a consequential step.

All code introduced by the University of Koblenz-Landau (*rip_mti.c*) has been re-arranged this way. Modifications of existing code (mostly in *ripd.c*) have been done in a way to minimize deltas, meaning the new coding style is used when the enclosing block already uses it.

More information about the coding style can be found in appendix E.

6.11 Reduction of code and patch size

As mentioned before, RMTI was developed using a Subversion repository during the last few years. The initial patch[13] file I created was huge: 16.2 megabytes.

[13] created using the command: diff -urN quagga-0.99.9 ripmtixtsl

The size can be explained by the fact that a lot of files generated by the build system had been committed to the repository. Also, the code included a lot of useless whitespace changes.

I managed to get the patch down to 146.5 kilobytes before doing much work on the code itself. After all the work done during this thesis, the latest patch[14] is just 73.9 kilobytes.

Only looking at delta sizes does not tell a lot about the evolution of the project itself. To get a better feel, the source code analyzer `cloc`[15] version 1.5 was run to count the lines of C code inside the *ripd/* directory:

Version	file size (KB)	LoC
RMTI (old version)	449.4 KB	10622
Quagga 0.99.9	262.9 KB	7596
Quagga 0.99.16	263.2 KB	7606
RMTI (current version)	312.9 KB	8649

The table shows that only 10 additional lines were indroduced between Quagga 0.99.9 and 0.99.16 release. It is fair to assume that most changes specific to RIP just represented bug fixes and did not add new features.

The actual lines of code for RMTI can easily be calculated as the difference of the respective values of the RMTI version and its Quagga base version: the old RMTI code consisted of 3026 lines, the final code is down to 1043 lines (see figure 6). This equals a reduction by the factor of 2.9.

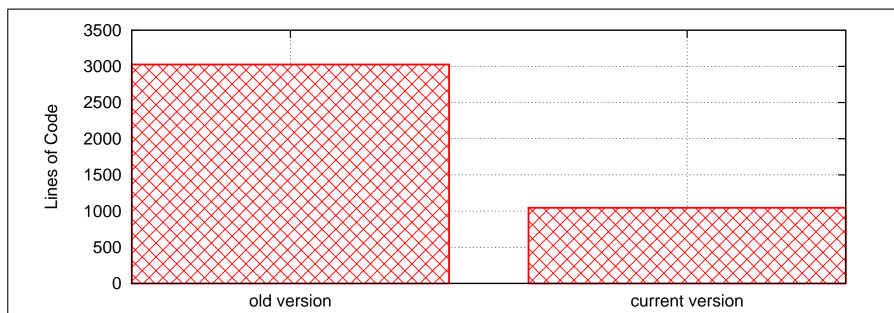


Figure 6: Reduction of RMTI code size

Counting the code in all directories, the difference is 1046. This also shows that RMTI does not touch much code outside the *ripd/* directory.

The removal of deprecated modes and unused codepaths in addition to many simplifications considerably reduced the overall size of the project, making it easier to understand and maintain in the future.

[14]created using the command: git diff q_0_99_16 r_0_99_16
[15]CLOC = count lines of code - `http://cloc.sourceforge.net/`

7 Test environment

In order to validate a routing algorithm, a special test environment is needed. Compared to local code, a distributed algorithm poses a bigger challenge:

- there is no global knowlege and no central place to store a result
- computation and communication needs to be flawless on all nodes to achive the correct result
- testcases only work as long as the environment does not change, however, this is nearly impossible to ensure in a distributed environment
- the number of participating nodes can vary over time: new nodes connect, old nodes disconnect from the network

In the case of a routing algorithm, the network may often fall into some inconsistent state. Still, the algorithm needs to react to the situation and induce convergence as fast as possible. The absence of a central element is a strong point in this case because it can neither hurt scalability nor reliability. However, how can testing be done in such a scenario?

The XTPeer system works around this problem by building up global knowledge itself. It constantly collects information from all participating nodes via dedicated channels and analyses it in a central place. In addition, it needs to know the whole network topology itself. As the XTPeer integration was removed as part of this thesis, an different approach was needed.

The key element of the old test environment was *Virtual Network User Mode Linux* (VNUML), a system that allows to build complex networks using *User Mode Linux* (UML) based virtual machines and software bridges. Detailed information can be found in [25] and [7]. It is important to note that VNUML does not rely on XTPeer, so it can still be used as part of the new test environment.

Before thinking about how to build replacemnts for the other parts, it makes sense to formulate a list of requirements which the test environment needs to satisfy:

- ability to provoke the counting-to-infinity reliably
- ability to copy files to the virtual machines
- ability to copy log files back to the host system
- ability to repeat a given test case for a specified number of times
- be as quick as possible

It is also important to declare what the tests are supposed to proof. Different testing goals could be the correctness of the CTI prevention or changes of properties like convergence time and traffic volume. Apparently, assuring the new implementation still reliably prevents the counting-to-infinity needs to be the focus here. Tests regarding other properties would be nice as well, however those would require comparable test results from the XTPeer-based version to have any meaning.

7.1 Hybrid testing procedure

I first considered a hybrid testing procedure to be feasible. The plan was to replace the new software on one or more selected routers in a topology with the old XTPeer-enabled version.

This approach is problematic because testing the new version together with an older release would not deliver any conclusive results. On the one hand, occurring problems could just be side-effects of minor changes between the two versions and may not happen when using only one version. On the other hand, there is no way to assure that everything would still work the same if only the new version was used.

7.2 Zimulator

The *Zimulator* environment was written by Marcel Jakobs and is described in detail in the related thesis paper [12]. Zimulator runs in two phases, one online and one offline phase. During the online phase, the test topology is set up using VNUML and the scenario is being run. Network traffic is being recorded by `tcpdump` for later use in the offline phase.

As RMTI uses the same packet format as RIP, Zimulator also works for RMTI. However, RIP is limited to the default infinity value of 16, so Zimulator had this value hardcoded as well. To support RMTI, I added support for changing the infinity value at run time using the *-I* flag (full patch in appendix H).

While offering very powerful tools for statistical analysis, the system does not meet the given requirements. It is very useful in cases where properties of a stable codebase are being analyzed but only of limited use for testing experimental code. It is not possible to judge the state of the running simulation before the offline phase is finished. This requires the scenario to aquire a consistent state during the online phase first, which may take a long time. The offline phase on the other hand is known to require a lot of CPU power and may take a long time itself. It is also not possible to cancel a run or copy log files from the virtual machines.

7.3 Testomato

As the Zimulator tool does not allow running a large number of CTI tests in a *passable* time, some alternative was needed. This let to the development of a simple script for test automation and monitoring called *Testomato*. The code is listed in appendix J.

The script itself is written in *Bash* and mainly collects information using command line tools like `vtysh` and `route`. The UML *hostfs* is used to exchange files. Early versions used *SSH* to execute these commands on the virtual machines, sometimes causing large delays and rendering the results unusable. Later, *uml_mconsole* proved to be a much faster and more reliable solution. While it only works unidirectional, all output can be written to a file in the hostfs direcory and later be read by the main script from there.

7.3.1 Configuration of Testomato

Before the tool can be used to start a test run, the VNUML framework needs to be installed. The procedure is descibed in [18]. Additionally, a number of files need to be created:

topology.xml - A normal VNUML topology description file. Visit the VNUML website for detailed information about the XML format [23] or see [18] for a summary of the most important elemets.

scenario.run - A small file which specifies what happens once VNUML has started all virtual machines and set up the networking. Standard bash syntax and console commands can be used here. Additionaly, the following commands are predefined:

- **waitFor** *delay* : Initial delay to allow the network to achieve a convergent state.
- **close** *router interface delay* : Cut connection of given interface on selected router.
- **getRoute** *router network* : Show route information for given network on selected router.
- **open** *router delay* : Reopen all connections on selected router.
- **copyLogs** : Copy all RIP daemon log files to a local directoy.

The second part of appendix J shows a few selected example scenario files.

etc/zebra.conf and **etc/ripd.conf** - Zebra and RIP daemon configuration files used for all virtual machines. All files inside the *etc/* folder are copied to the */mnt/hostfs/* folder on the virtual machines.

7.3.2 Running Testomato

After everything is set up correctly, the tool can be used like this:

```
# testomato -x xmlfile [-r runfile] [-n number of runs]
```

-x ***name.xml*** (mandatory): Name of the VNUML topology file.

-r ***name.run*** (optional): Name of the file containing the operational sequence. If not given, try the same name as used in -x parameter.

-n ***integer*** (optional): Number of runs. If not given, run only once.

Testomato writes a log to the terminal as well as a *test.log* file in a directory named *tests/xml (run) date-time/*. This directory also includes subdirectories for each run, which in turn contain a copy of the *ripd.log* file of every virtual machine.

The approximate duration of a simulation can be estimated based on the following factors:

1. VNUML **startup** : about 2 minutes
2. time for **convergency** : about 15 seconds per run
3. time for **timeout** of route : about 25 seconds per run
4. time to wait for **CTI** : about 40 seconds per run

Based on these numbers, running a test series with 100 iterations takes about 2 hours and 15 minutes while 1000 iterations take about 22 hours and 30 minutes.

Note that item 1 depends heavily on the hardware and current load of the host machine. It may take a bit longer but only happens once per simulation. Items 2 to 4 happen per iteration and require more time if the topology contains larger loops or longer RIP timer values are used.

The following listings include sample output from the tool. The first row shows some of the settings being used, including the infinity value and RIP timers. In listing 9, the RMTI support is disabled. The counting-to-infinity occurs, which can be seen by the steadily increasing metric.

```
Infinity: 24   MTI: 0   Timers: 10 / 40 / 30

=== 1 / 100 ====================

wait for 15 seconds

close interface eth1 on r5 and wait for 25 seconds

r1: R(n) 10.0.6.0/24          10.0.4.2           3 10.0.4.2           0 00:38
r2: R(n) 10.0.6.0/24          10.0.3.1           4 10.0.3.1           0 00:35
r3: R(n) 10.0.6.0/24          10.0.1.1           4 10.0.1.1           0 00:34

r1: R(n) 10.0.6.0/24          10.0.4.2          24 10.0.4.2           0 00:27
r2: R(n) 10.0.6.0/24          10.0.3.1          24 10.0.3.1           0 00:27
r3: R(n) 10.0.6.0/24          10.0.1.1           4 10.0.1.1           0 00:33

r1: R(n) 10.0.6.0/24          10.0.3.2           9 10.0.3.2           0 00:37
r2: R(n) 10.0.6.0/24          10.0.2.2          11 10.0.2.2           0 00:40
r3: R(n) 10.0.6.0/24          10.0.1.1          10 10.0.1.1           0 00:36

r1: R(n) 10.0.6.0/24          10.0.3.2          18 10.0.3.2           0 00:40
r2: R(n) 10.0.6.0/24          10.0.2.2          20 10.0.2.2           0 00:40
r3: R(n) 10.0.6.0/24          10.0.1.1          19 10.0.1.1           0 00:39

r1: R(n) 10.0.6.0/24          10.0.3.2          21 10.0.3.2           0 00:37
r2: R(n) 10.0.6.0/24          10.0.2.2          23 10.0.2.2           0 00:39
r3: R(n) 10.0.6.0/24          10.0.1.1          22 10.0.1.1           0 00:36

r1: R(n) 10.0.6.0/24          10.0.3.2          24 10.0.3.2           0 00:25
r2: R(n) 10.0.6.0/24          10.0.2.2          24 10.0.2.2           0 00:26
r3: R(n) 10.0.6.0/24          10.0.1.1          24 10.0.1.1           0 00:24

opening all interfaces on r5 and wait for 0 seconds

Resetting scenario (restarting daemons)...
```

Listing 9: Testomato output showing a CTI

In contrast, listing 10 shows a similar scene with RMTI support enabled. In this case, the counting-to-infinity is prevented and the metric stays constant.

```
Infinity: 24   MTI: 1   Timers: 10 / 40 / 30

=== 1 / 100 ====================

wait for 15 seconds

close interface eth1 on r5 and wait for 25 seconds

r1: R(n) 10.0.6.0/24          10.0.4.2          3 10.0.4.2         0 00:39
r2: R(n) 10.0.6.0/24          10.0.3.1          4 10.0.3.1         0 00:38
r3: R(n) 10.0.6.0/24          10.0.1.1          4 10.0.1.1         0 00:37

r1: R(n) 10.0.6.0/24          10.0.4.2         24 10.0.4.2         0 00:29
r2: R(n) 10.0.6.0/24          10.0.3.1         24 10.0.3.1         0 00:29
r3: R(n) 10.0.6.0/24          10.0.1.1          4 10.0.1.1         0 00:32

r1: R(n) 10.0.6.0/24          10.0.4.2         24 10.0.4.2         0 00:24
r2: R(n) 10.0.6.0/24          10.0.2.2         24 10.0.2.2         0 00:27
r3: R(n) 10.0.6.0/24          10.0.1.1         24 10.0.1.1         0 00:27

...

r1: R(n) 10.0.6.0/24          10.0.4.2         24 10.0.4.2         0 00:08
r2: R(n) 10.0.6.0/24          10.0.2.2         24 10.0.2.2         0 00:11
r3: R(n) 10.0.6.0/24          10.0.1.1         24 10.0.1.1         0 00:11

opening all interfaces on r5 and wait for 0 seconds

Resetting scenario (restarting daemons)...
```

Listing 10: Testomato output showing no CTI (shortened)

7.4 Generator for counting-to-infinity situations

As described, both Zimulator and Testomato allow provoking counting-to-infinity situations using scenario description files. In both cases, the user has to specify a sequence of commands which close and reopen connections with the right timing to initiate the CTI. In practice, however, it turns out that such a procedure is not very reliable:

- The simulation environment is subject to influences from the outside, such as host system CPU usage.
- RIP updates are sent with a random delay of zero to five seconds, meaning iterations can differ a lot.
- The timings heavily depend on the configured RIP timers (mostly the periodic update timer). After changing the timers, the scenario file needs to be adjusted as well.

Is becomes obvious that the chance of generating a counting-to-infinity situation decreases with growing loop sizes.

To avoid these factors, the idea was to build support for generating a counting-to-infinity situation directly into the RIP daemon. The theory is simple: one of the routers is selected to initiate the CTI. On this router, a special command `cti X Y` has to be executed. The *X* denotes the interface which receives the

information about a no longer reachable network. The Y denotes the interface over which updates don't reach the neighbor.

The generator is very useful for testing purposes and can greatly simplify the creation of scenario description files. Their duty is reduced to just a few things: cutting off one of the networks using the `close` command, showing periodic output about the state of the network using the `getRoute` command and copying the log files at the end of each iteration using the `copyLogs` command.

The current code is based on some unfinished work by Frank Bohdanowicz and exists as a patch against the latest RMTI code. It can be found online[16] and is also listed in appendix I. The following is a short discussion of the essential parts.

The generator introduces a few new variables: The *cti_route_manipulation* variable is set when the generator is activated. The variables *cti_incoming_eth* and *cti_blocking_eth* variables hold the names of the incoming and blocked interface, *cti_incoming_idx* and *cti_blocking_idx* hold the corresponding indices.

```
if (cti_route_manipulation)
{
  unsigned int cti_ifindex = ifname2ifindex (cti_incoming_eth);

  if (!rinfo->cti_flag
      && ifp->ifindex == cti_ifindex && ifp->ifindex == rinfo->ifindex
      && rte->metric >= rip->infinity_metric)
  {
    rinfo->cti_flag = 1;
  }

...
```

Listing 11: CTI patch, part 1

Placed inside the function that analyzes incoming routes (*rip_rte_process*), this code checks if the route is suitable to cause the counting-to-infinity. For this, the route needs to satisfy two requirements:

- the route was received over the specified incoming interface
- the route carries information about some unreachable network, meaning the metric equals RIP infinity

```
if (cti_route_manipulation
    && rinfo->cti_flag
    && ifc->ifp->ifindex == cti_ifindex
    && rinfo->metric >= rip->infinity_metric)
{
  continue;
}
```

Listing 12: CTI patch, part 2

During the output process (*rip_output_process*), updates of the flagged routes are omitted on the blocked interface.

[16]CTI generator patch: `http://git.uni-koblenz.de/share/agrn-share/trees/master`

```
...

  if (rinfo->cti_flag
      && !IPV4_ADDR_SAME(&rinfo->from, &from->sin_addr)
      && ifp->ifindex != cti_ifindex)
  {
    rinfo->cti_flag = 0;
    cti_route_manipulation = 0;
  }
}
```

Listing 13: CTI patch, part 3

Placed directly below the first snippet (in *rip_rte_process*), this code is meant to turn off the generator as soon as the counting-to-infinity is initiated. To meet this condition, same route needs to be received with a new learned-from address via another interface.

While the generator patch has provn itself in simple topologies, it is likely to cause problems in the more compex ones. For example, the current version only works correctly if exactly *one* network is advertised as *not reachable*. If there are multiple networks, the code will try to generate the counting-to-infinity for each of the networks but stop after initiating the first.

8 Test results and evaluation

The tests presented on the following pages have all been carried out using the Testomato script combined with the CTI generator, both described in the previous section.

The chosen test topologies include the simple (see 8.1) and the extended (see 8.3) upsilon topology as well as the circle topology (see 8.2). Each one has its own relevance:

Simple upsilon topology
: The simple upsilon topology is often used as an introduction into the counting-to-infinity problem. It represents the smallest constellation of routers that is affected by the problem even when the split horizon rule (see section 2.2) is respected.

Extended upsilon topology
: The extended upsilon topology is based on the simple upsilon topology but contains a nested loop. As shown in [31], this class of topology is known to cause problems with the original implementation (*NORMAL* mode) of RMTI.

Circle topology
: Compared to the simple upsilon topology, the circle topology has a larger loop. It is an example for a topology that relies on the dynamic timer value adaption described in section 6.9.

In order to *pass* the test, the new RMTI implementation was required to complete at least one thousand iterations of each tested topology with the expected result. To be more precice, the expected results are:

- All iterations executed in a test series with the RMTI algorithm *disabled* (mti=0) should result in a CTI. In the log file, this can be verified by looking for a gradually increasing RIP metric.
- All iterations executed in a test series with the RMTI algorithm *enabled* (mti=1) should not result in a CTI. In the log file, this can be verified by looking for a constant metric equal to RIP infinity.

In many cases, even a simple code change can have undesirable side effects. In order to become aware of such regressions as fast as possible, many tests series have already been carried out during the whole development cycle. Every time a series did not meet the expectations described above, the following steps had to be taken:

1. analysis of the RIP log files of the failed iteration(s)
2. attept to find and fix the problem in the code
3. recompilation and update of file system image
4. restart of the test series and back to step one

All tests have been carried out on a laptop with the following specification:

CPU	Intel(R) Core(TM)2 Duo CPU T7250 @ 2.00 GHz
RAM	2 GB of DDR2
OS	Fedora 13 + latest updates
VNUML	1.8.9
Host kernel	kernel-2.6.31.12-174.2.22.fc12.i686
UML kernel	linux-2.6.18.1-bb2-xt-4m

The kernel has been downgraded to the latest package for Fedora 12 because newer kernels show a bad performance regression. Details about this issue can be found in appendix A.

Each of the sample topologies is visualized using the common elements pictured in figure 7.

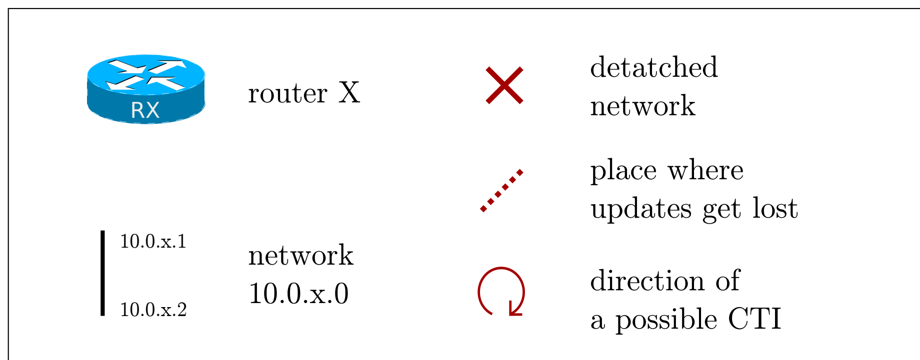


Figure 7: Elements of a topology visualization

The scenario description files mentioned on the following pages can be found on the DVD and Live CD accompanying the printed version and are also listed in the second part of appendix J.

8.1 Topology 1: upsilon

The simple upsilon topology shown in figure 8 consists of five routers. The routers R1, R2 and R3 are connected in a loop. R1 is also connected to R4, which itself is connected to R5.

Using Testomato, the scenario can be started with the following command:

```
$ ./testomato -x y -n 1000
```

The VNUML suite take care of building and setting up the topology. After the routing software is started on all virtual machines, the script waits for a short period of time in order to allow the scenario to obtain a convergent state. The time for this depends on the chosen RIP update timer but usually only takes a few seconds.

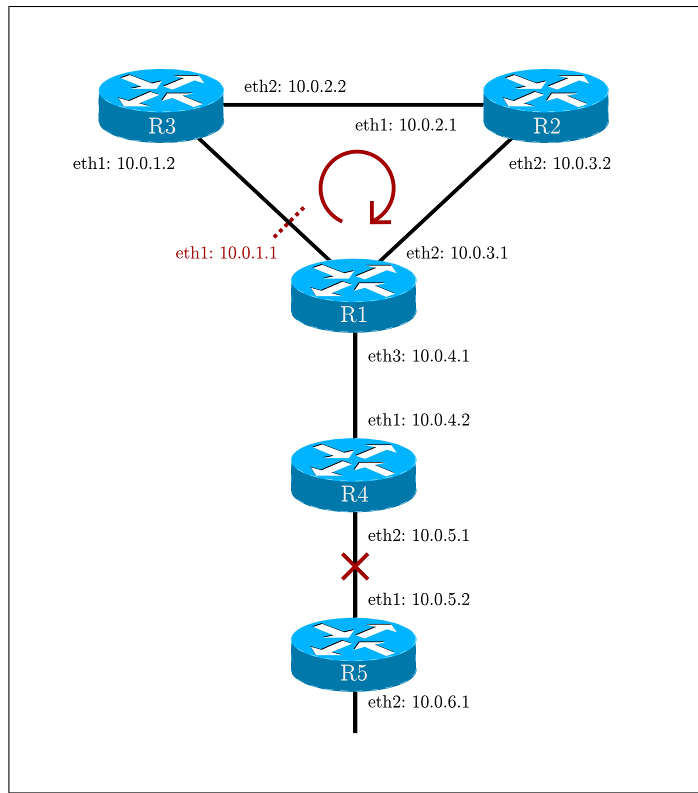


Figure 8: The upsilon topology

As soon as a convergent state can be assumed, Testomato interrupts the connection between routers R4 and R5, effectivly cutting off R5 and the network 10.0.6.0. At this point, packets can no longer reach the 10.0.6.0 network. The first router to notice the problem is R4, as it will no longer receive update packets for this network from R5. After some time, the timeout timer for the route will reach zero and the network is regarded as unreachable on R4 from this point on.

The disturbed reachability of the network 10.0.6.0 is propagated to router R1 through an update containing the infinity metric. In turn, an update is triggered on router R1, passing the infinity metric to its neighbors R2 and R3.

At this point, the CTI generator kicks in: the RIP daemon on R1 is configured to silently drop update packages containing the infinity metric when send to R3. The update packet only reaches R2, which tries to pass it along to R3. Having learned the route to 10.0.6.0 from R1 originally, R3 will assume that its current information is still valid and better than what R2 has to offer. R3 sends an update to R2, overwriting the infinity metric with the outdated but seemingly better value. R2 assumes the offered route is valid and accepts it.

Without RMTI: Router R2 passes on the incorrect metric to R1. Having no means to know that it is not valid, R1 also accepts the route as it has a lower metric and in turn passes it on to R3. The CTI generator does not drop the packet this time because it does not contain the infinity metric. R3 will accept the new information because it originally learned the route from R1. At this point, the counting-to-infinity can not be avoided anymore. The routers will continue to pass on the newly received metrics in the loop between R3, R2 and R1 until the metric reaches the infinity value and finally times out.

With RMTI: Router R2 passes on the incorrect metric to R1. Making use of the extra information that RMTI provides, R1 is aware of the existance of a loop between its interfaces eth1 and eth2. The update is rejected and the correct metric, infinity, is sent back to R2 again.

8.1.1 Running the upsilon topology with host integration

As a variant of the simple upsilon topology, one of the virtual machines can be replaced by the host system. The host system should be placed inside the fork, connecting the 10.0.1.0, 10.0.3.0 and 10.0.4.0 networks because this is the only spot where RMTI can prevent the counting-to-infinity in this topology.

This is especailly useful while working on the code as the daemon can be run from the main working environment. A new version can be tested after a quick recompile without the need to modify the file system images of the virtual machines.

The sample scenario *yh.xml* requires an additional *ripd.conf* file, placed in */tmp* as */tmp/host-ripd.conf*. Once the host system is set up, the scenario can be run by executing the following cammands:

```
# vnumlparser.pl -t yh.xml -u root -vB
# vnumlparser.pl -x start@yh.xml -u root -vB
# vnumlparser.pl -x zebra@yh.xml -u root -vB
# vnumlparser.pl -x rip@yh.xml -u root -vB
```

If the RIP daemon running on the host has been patched to support the *cti* command, all that need to be done now is to manually disconnect network 10.0.5.0 somehow.

Note that the normal CTI generator described before does not work in this case because the interfaces have different names. For testing purposes, the interface names can just be changed from *eth* to *net*.

After initiating the CTI, check the local RIP routing table on the local host periodically. This can be achieved by executing the following command:

```
$ vtysh -c "show ip rip"
```

If running in RMTI mode, the metric for network 10.0.5.0 will change to infinity shortly after the timeout timer reaches zero on R4 and no counting-to-infinity will occour.

8.2 Topology 2: circle

The circle topology shown in figure 9 consists of eight routers. The topology itself is very similar to the upsilon topology but the loop is bigger as it is made out of six routers (R1 to R6) instead of three. The loop can be expanded at will. R1 is connected to a router R7 which is connected to R8.

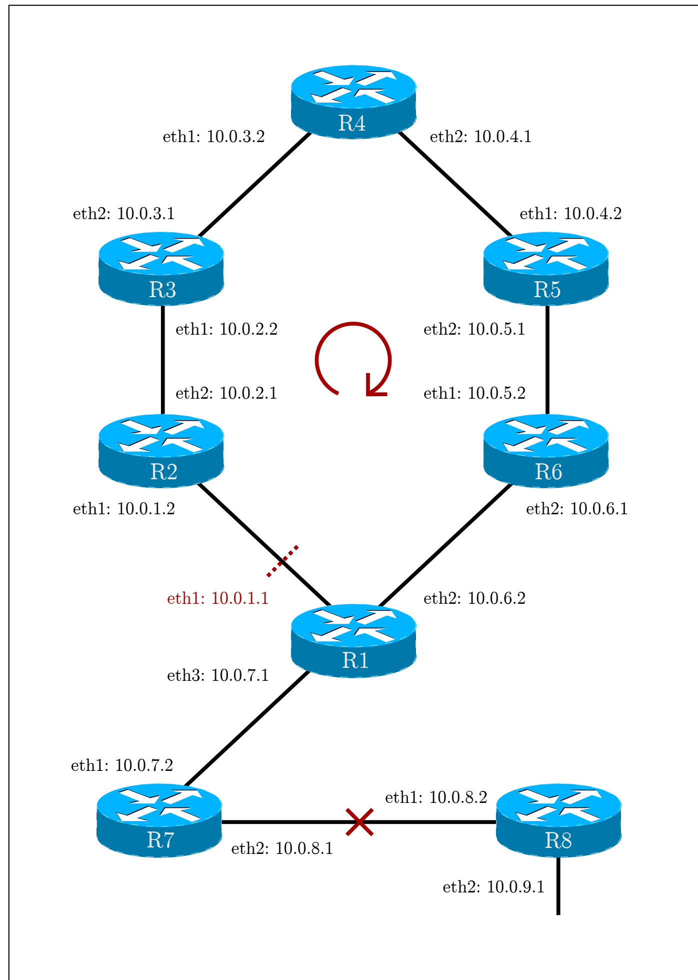


Figure 9: The circle topology

Using Testomato, the scenario can be started with the following command:

```
$ ./testomato -x c6 -n 1000
```

The initiation and development (without RMTI) or the reliable prevention (with RMTI) of the counting-to-infinity problem happens just as described for the the simple upsilon topology. However, the bigger size of the loop can cause problems in cases where the garbage collection time is set to a small value. The bigger the loop, the more likely this problem becomes even with a reasonable timer value.

The problem and its solution have already been described in section 6.9.

Without the fix it is possible that the original route is already garbage-collected on R1 before the incorrect metric arrives through the loop. In this case, RMTI is not able to prevent the CTI: there is no way to tell that the route was originally passed from R7 to R1 anymore. R1 accepts what it thinks is a completely new and valid route. It passes the invalid route to R2, continuing the active couting-to-infinity.

With the fix the configured value of the garbage collection timer is multiplied by the size of the loop (six in this case). Now it is very unlikely that the timer will run out prematurely, so RMTI can still detect the loop and prevent the CTI just as it does in the simple upsilon topology.

8.3 Topology 3: extended upsilon

The extended upsilon topology shown in figure 10 is another variant of the the simple upsilon topology. It contains one extra router R6, which is connected to both the R2 and R3.

This constellation is called a *nested* loop because it contains two interconnected loops. There are three possible ways a CTI can be initiated here:

1. R1-R3-R2-R1 (bottom loop, same as simple upsilon)
2. R1-R3-R6-R2-R1 (big loop)
3. R3-R6-R2-R3 (top loop)

Using Testomato, the scenario can be started with the following command:

```
$ ./testomato -x y+ -n 1000
```

8.4 Final results and log files

As mentioned, all topologies have been tested during development in order to prevent regressions. After development was done, another series of tests with 1000 iterations was carried out for each topology, once with plain RIP and once with RMTI enabled.

In all cases, the counting-to-infinity was successfully prevented when running in RMTI mode. Still, the counting-to-infinity was successfully initiated in all cases when running in plain RIP mode.

The log files of these final test runs can be found in the directory *testomato/tests/* on the DVD accompanying the printed version of this thesis.

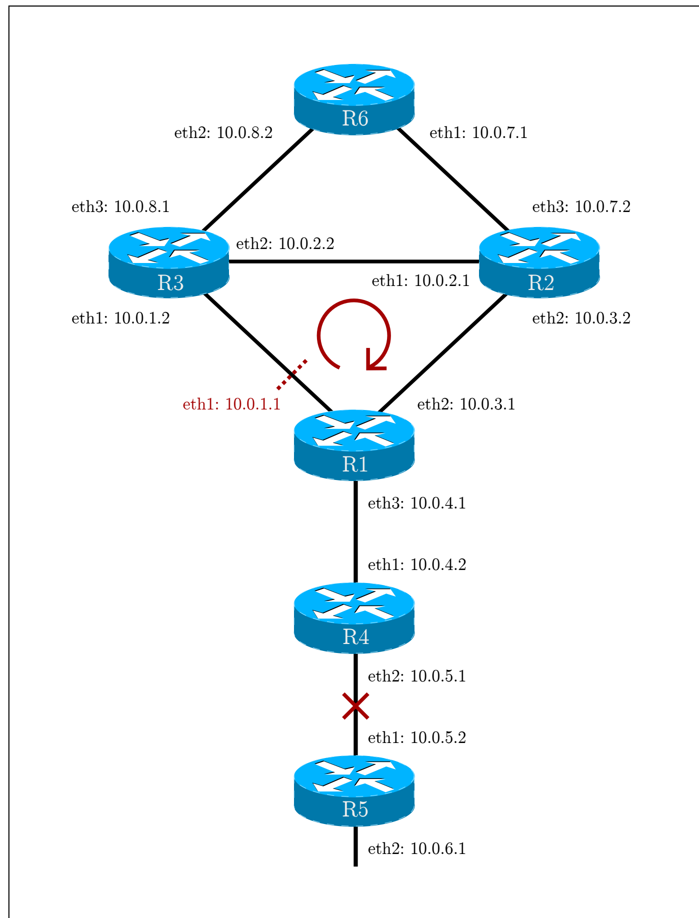


Figure 10: The extended upsilon topology

9 Public release

The current implementation of RMTI is considered to be stable and usable. However, the user base is still very small and real world deployment is scarce. In order to change this, it was decided to release the whole project to the public under a permissive free software license. This section details the process of preparing and assisting the release process.

9.1 License

The Quagga project makes use of of the GNU GPL (GNU General Public License) license, version *2 or later*. As RMTI is heavily based on the Quagga RIP implementation, using the same license is the only viable option. In turn, the RMTI code is now licensed GPL2+ as well.

The license results in a number of legal implications which can be studied in detail in [9]. While the original license text is not easy to understand for a layperson, everyone working on the code should be familiar with the basics:

- The RMTI code can be freely distributed and used in any way with or without the rest of the Quagga code. This includes both source and binary forms.
- Everyone is allowed to create and distribute software based on or derived from RMTI code, as long as they also comply with the GPL2+. More precisely, this requires the release of their modified source code under the same license.
- It is possible to relicense the code to GPL3+. Compatibility of GPL versions is explained in [8].
- Either the whole RMTI codebase or parts of it could be merged into upstream Quagga easily.

9.2 Website and discussion board

For some time, the RMTI website[17] (figure 11) provided a small amount of information related to the development of the algorithm. The page has now been reworked to reflect the latest happenings and offers new content in the *News* and *Downloads* sections.

For public discussion, a bulletin board[18] has been set up (figure 11). The board is based on the popular phpBB software and contains sections for news, general discussion as well as bug reports and feature requests.

A hyperlink pointing to the RMTI website has been added to the Wikipedia entry[19] about the routing information protocol.

[17] RMTI website - `http://userp.uni-koblenz.de/~vnuml/rmti/`
[18] RMTI discussion board - `http://agrn.uni-koblenz.de/forum/`
[19] RIP @ Wikipedia: `http://en.wikipedia.org/wiki/Routing_Information_Protocol`

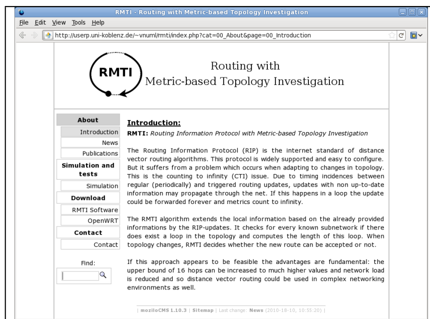


Figure 11: RMTI website

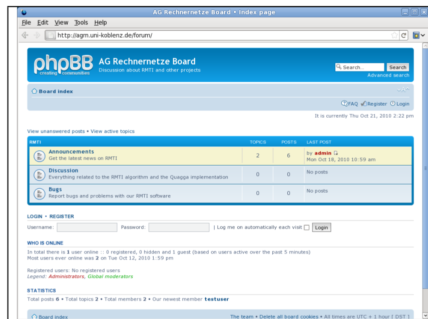


Figure 11: Discussion board

9.3 Source code

The *Downloads* section of the website offers the latest RMTI release packed in a tarball. For people already working on Quagga, it is also available as a patch against the upstream code.

The very latest code can be found in our Gitorious repository[20]. It can be accessed using the standard Git tools. An account is not required for read-only access. However, registered users can create public forks of the code and gain write-access to the main repository after applying for membership in the RMTI group.

Appendix B provides detailed information about getting the code and working with the repository. This information can also be found online on the RMTI project wiki[21].

9.4 Binary packages

The RMTI web page also offers ready-to-use binary packages for various Linux-based systems in the *Downloads* section. Right now Debian/Ubuntu (.deb), Red Hat/Fedora (.rpm) and OpenWRT (.ipk) packages are available.

The packages are based on each distribution's official Quagga package, so they behave exactly like the original and just ship the RMTI-based RIP instead of the unmodified RIP implementation. They are meant to be used inside a virtual machine file system or run a RIP router instance on the host machine.

Detailed notes on how the packages were built can be found in appendix G. Note that all packages currently available are compiled for *x86* (32-bit) only. If package for a *x86-64* (64-bit) system are needed instead, please refer to the compilation guide.

Installing the binary packages is easy. While not strictly required, it is recommended to install the official Quagga package using the package manager first.

[20] RMTI project - `http://git.uni-koblenz.de/rmti`
[21] RMTI wiki - `http://git.uni-koblenz.de/rmti/pages/Home`

This ensures that all dependencies are satisfied. In a second step, the official package is replaced by the custom RMTI-enabled package:

Installing the Debian/Ubuntu package

```
# apt-get install quagga && apt-get remove quagga
# dpkg -i quagga_0.99.16-1rmti_i386.deb
```

Installing the Red Hat/Fedora package

```
# yum install quagga && yum remove quagga
# rpm -ihv quagga-0.99.16-0rmti.fc13.i686.rpm
```

The OpenWRT packages are meant to be installed on modified routers using the free OpenWRT operating system. They are built specifically for the *Linksys WRT54GL*[22] but may also work on similar hardware.

9.5 File system image

For those interested in running RMTI in a virtualized environment, the website offers a minimal file system image. It is created to be used with the VNUML virtualization suite but may also work with other solutions.

9.6 Live CD

Finally, we provide a Live CD image (ISO 9660 format). It is based on the 6.2.1 release of Knoppix[23] and includes a fully working VNUML setup featuring the RMTI file system mentioned above.

To use the image, just write it to a physical disc (700 MB) and reboot. Make sure booting from the optical disc drive is enabled in BIOS. It will then boot the system automatically and does not require installation or any changes to the hard drive.

Alternatively, the image can be run in a virtual machine. One option is to use KVM, a hypervisor integrated in newer Linux kernels:

```
$ qemu-kvm -m 1024 -cdrom knoppix-6_2_1-rmti_0_99_16.iso
```

Other popular virtualization tools like VirtualBox[24] can also be used.

The Live CD contains a copy of the Testomato framework and a few sample scenarios in */home/knoppix/testomato/*.

[22] Linksys WRT54GL - `http://en.wikipedia.org/wiki/Linksys_WRT54G_series#WRT54GL`
[23] Knoppix - `http://www.knoppix.net/`
[24] VirtualBox - `http://www.virtualbox.org/`

10 Conclusion

As demonstrated in various papers like [3] and [4], the RMTI algorithm can be used to improve the aging RIP protocol. By offering a solution to the counting-to-infinity problem, the average convergence time in critical situations can be reduced significantly.

Compared to normal RIP technology, RMTI routers can be deployed in larger networks because they are no longer subject to the artificial limitation of 15 routers per path. As described, the current implementation now supports changing the infinity value using a simple addition to the configuration file. The improved scalability of RMTI helps to narrow the gap betweenn classic distance-vector protocols and current link-state protocols like OSPF.

Being competitive with regard to converge time and scalability is important, but RIP and therewith RMTI still have some more advantages over link-state based protocols. On the one hand, their implementations are comparatively simple and easy to understand. Deployment is easy too, because it requires only a minimal amount of configuration. On the other hand, distance-vector based protocols have a potential to support routing policies, similar to the BGP protocol.

Today, RMTI is finally available to the public as a minimal patch against the current version of the Quagga routing suite. This should make it attractive for third parties to give it a try and hopefully return feedback.

The reorganization of the code and the switch from Subversion to Git has already payed off. It offers a great way to collaborate and makes it very easy to track changes from upstream Quagga.

10.1 Ongoing work

The reduced RMTI codebase has proven to be reliable in simple virtualized networks as demonstrated before. Additional tests are still needed, both in more complex topologies as well as non-virtualized environments. The latter is taken care of by Ansgar Taflinkski, who is currently deploying and testing the new code on OpenWRT-based hardware as part of his bachelor thesis [28].

Unfortunately, the reduced codebase makes testing a lot harder compared to the previous XTPeer-based approach. While simple tests have been carried out successfully using the methods described in section 7, there is currently no easy way to test arbitrarily complex topologies. This is mainly due to the fact that creating a counting-to-infinity situation without any global knowledge is a tricky challenge and should be the focus of subsequent research.

Another regression from the previous testing environment is the absence of a graphical user interface. Maybe parts of the XTPeer desktop application can be used as a base for a possible future replacement which would not rely on the client/server architecture anymore.

Finally, there are various problems with the UML system used by VNUML and thus being part of our test environment. The current performace regression and uncertain future of UML are explained in appendix A. Because of this, Christopher Israel has started to explore [11] alternatives to UML. His work could lead to a replacement of VNUML based on on Qemu[25].

10.2 Future of RMTI

As the basic concept behind RMTI is not bound to the RIP protocol, it should be easy to adapt to other disctance-vector protocols. Quagga already ships a working IPv6 implementation of RIP called RIPng, so adding RMTI support to this could be a future project for example.

In the past, one of the design goals was keeping RMTI compatible to ordinary RIP implementations. The advantage is that modified routers can be deployed together with unmodified RIP routers in the same network in this approach. However, it also limits the amount of possible improvements which could be implemented otherwise. To be fair, compatibility with RIP is already lost as soon as a higher infinity metric is being used, so the theoretic compatibility may not be such a big factor in reality.

At this point, RMTI in its legacy-compatible form is considered to be feature complete. Future research is open to more drastic changes to the protocol.

One of the current experiments is to replace the use of timed updates with neighbor-keepalife messages similar to the *HELLO* messages in OSPF, as described in the paper [14] by Milad Khojasteh. This approach has the potential to greatly decrease update traffic while solving many of the timing-related RIP problems at the same time.

Another project aims to add support for filtering policies to RMTI. Andreas Brandt is currently laying the groundwork for such an extension.

[25]QEMU project - `http://www.qemu.org/`

A User mode linux performance regression

At some point during the testing phase, the performance of the test environment regressed noticeably. The VNUML script did no longer manage to bring up the virtual machines within the given default timeout period of one minute. Shortly after that, other students reported the same problem, independent of the Linux distribution being used. After testing various system components I was able to identify the host kernel as the cause of the problem.

To narrow down the regression range and to find the newest working kernel, I ran the following simple test (measuring the time to boot and directly shut down one virtual machine) in about 20 host kernel versions:

```
# time ./linux-2.6.18.1-bb2-xt-4m ubda=root_fs_tutorial-0.5.2 mem=64M
```

The test was run at least tree times on each host kernel in order to get a representative average value. Figure 12 shows only a selection of the most interesting kernels.

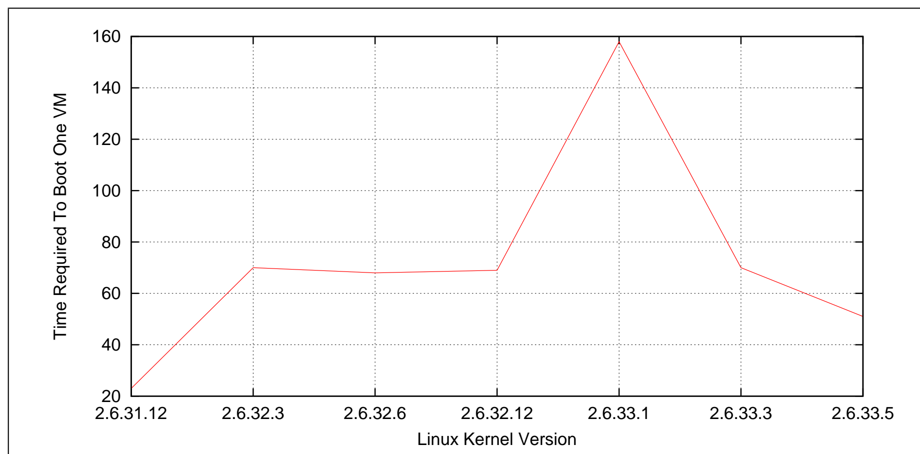


Figure 12: Performance of different host kernel versions

The graph shows that booting a single virtual machine normally only takes about 23 seconds on the given hardware as demonstrated by the 2.6.31.12 kernel. With kernel 2.6.32.x, it suddenly takes about 70 seconds. The spike shown for 2.6.33.1 can be explained by the fact that the kernel in question used a debug configuration, which seems to be a hard hit to performance itself. As debugging has been turned off after that, the 2.6.33.3 kernel performs similar to 2.6.32.x again. The latest kernels in Fedora 13 (2.6.33.5 as of now) takes 51 seconds to complete the test. While this is a slight improvement, it still requires more than twice the time compared to the 2.6.31.x kernels. This means the regression is not fully fixed.

A.1 Workarounds

For now, the only real workaround seems to be keeping an older kernel. This basicly means sticking with an older distribution release or manually downgrading the kernel package on a newer distribution. The exact steps vary from

distribution to distribution. For the Fedora distribution, the following options exist:

Fedora 11
is the last release that works correctly even after installing all updates. Note that this release has already reached its *end of life*, meaning it is no longer supported and may not even be available for download anymore.

Fedora 12
installs a fast 2.6.31.x kernel initially but delivers a slow 2.6.32.x kernel via updates. On a new installation, you can prevent this from happening by adding the line `exclude=kernel*` to the [main] section in */etc/yum.conf*. Similar to the previous release, Fedora 12 is reaching *end of life* at the end of 2010 already.

Fedora 13
does no longer provide a fast 2.6.31.x kernel itself but you can use packages like 2.6.31.12-174.2.3.fc12[26] from Fedora 12. After downloading and installing this kernel using the command listed below, do the same modification to */etc/yum.conf* as described for Fedora 12.

```
# rpm -ihv --oldpackage kernel-2.6.31.12-174.2.22.fc12.*
```

Fedora 14
was released on november 2, 2010 and has not yet been tested. The workaround listed for Fedora 13 is likely to work for this release, too.

A.2 Future

I reported[27] the problem on Red Hat Bugzilla but there has not been any response yet.

Trying to get in touch with the UML maintainer Jeff Dike, I found one of his mailing list posts [6] from early 2010:

> "TBH, I haven't done any real work in UML in the last year or so.
> The world has pretty much passed UML by, and it's beyond my
> capabilities and time to catch up and keep up."

Since then, nobody has stepped up to take over maintainership. Generally, the interest in UML seems to be very low right now because everyone is focussing on newer virtualization solutions like KVM[28]. Unfortunately, this puts the future of UML into question and could necessitate large changes to the current test environment sooner or later.

[26]latest 2.6.31 kernel: `http://koji.fedoraproject.org/koji/buildinfo?buildID=151789`
[27]Red Hat bug #585913: `https://bugzilla.redhat.com/show_bug.cgi?id=585913`
[28]KVM = Linux **K**ernel-based **V**irtual **M**achine - `http://www.linux-kvm.org/`

B Gitorious

The following section aims to describe the special layout of the RMTI Git repository and helps setting up local clones to work with. Further notes on this topic can be found on the RMTI project wiki[29].

The RMTI project is special because it basicly exists as a long-term fork of the Quagga project. The goal was to develop RMTI independantly but still being able to track changes to the Quagga repository in order to benefit from their improvements and fixes. Ultimately, these requirements led to the setup shown in figure 13, which illustrates the interactions between the three separate repositories.

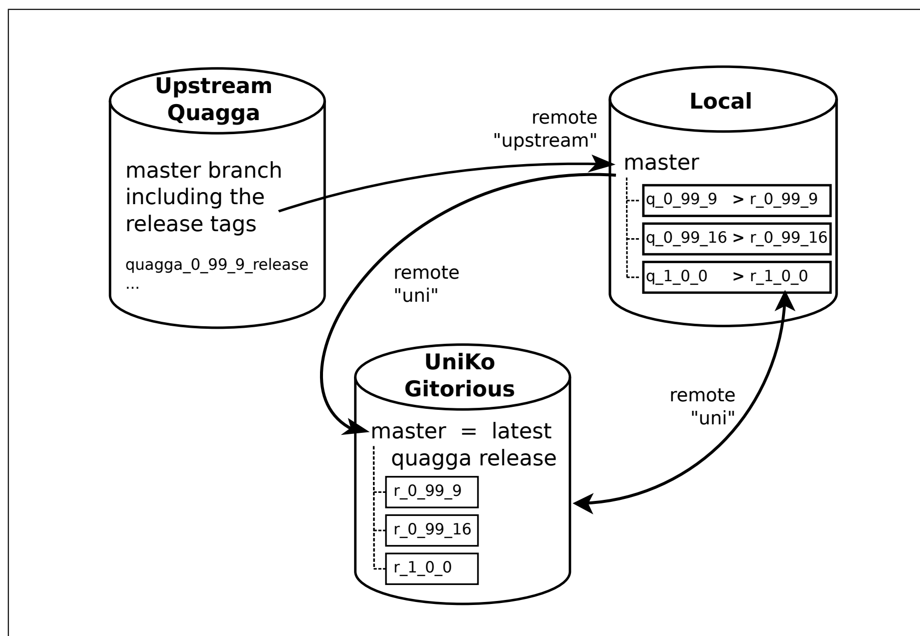


Figure 13: Interaction between the Git repositories

Upstream

The official Quagga Git repository always provides the latest Quagga source code. Releases are tagged with a special string. For example, the 0.99.6 release can be accessed using the *quagga_0_99_9_release* tag.

Local

A local Git repository including a mirror of the Quagga master branch as well as one or more RMTI branches. Each RMTI developer has his own local repository. Direct collaboration between local repositories is possible but not required.

[29]RMTI project - `http://git.uni-koblenz.de/rmti/pages/Home`

Uni

A central Git repository (similar to a Subversion server) which includes a mirror of the Quagga master branch as well as all (current, previous and experimental) RMTI branches. It is hosted on Gitorious[30], our web portal for managing Git projects.

For work on RMTI, only interaction with the *Local* and *Uni* repositories is mandatory.

B.1 Read access (pull)

Getting a copy of the latest RMTI sources just requires a few steps:

```
$ git clone git://git.uni-koblenz.de/rmti/rmti.git && cd rmti
$ git remote rename origin uni
```

Initially, this will clone the *master* branch only. As described above, this branch is just a mirror of the Quagga *master* branch. To get the actual RMTI code, first run `git branch -r` to get a list of all remote branches. Assuming the latest branch is *r_0_99_16*, run the following command to check it out:

```
$ git checkout -tb r_0_99_16 uni/r_0_99_16
```

B.2 Write access (push)

After the initial checkout, all you can do is pull new changes from the server. To be able to push your own changes, you first need to register[31] a new account on the Gitorious portal and also upload your SSH key (*Your dashboard* ⇒ *Manage SSH keys*). Then you can apply for membership in the RMTI group. Once these steps are taken, edit the repository URL as follows:

```
$ git remote set-url uni git@git.uni-koblenz.de:rmti/rmti.git
```

You can now push you own branches using the `git push uni` command.

B.3 Upstream integration

If you need to update the *master* branch to a newer Quagga release, integration of the upstream repository is required. Execute the following sequence:

```
$ git remote add upstream git://code.quagga.net/quagga.git
$ sed -i 's/remote = uni/remote = upstream/' .git/config
$ git pull
```

This will fetch the newest changes of the remote branch and merge them locally. It will also update the list of release tags. To bring the *master* branch on Gitorious up-to-date, use the `git push uni master` command.

[30] Gitorious - `http://gitorious.org/gitorious/`
[31] Register a new account on Gitorious on `https://git.uni-koblenz.de/users/new`

C Basic Git usage

This appendix is meant to be a short reference for the Git version control system.

For anyone who has never worked with a distributed version control system before, it is highly adviced to read at least the first three chapters of the official documentation, the Git Community Book [5].

Getting information

git status	show status of the current tree
git log	show the commit log
git log –oneline	show a summary of the commit log

Committing changes

git add *file*	add given file to the commit list
git add .	add all files to the commit list
git commit	commit staged changes
git commit -m *message*	commit stages changes with given commit message

Undoing changes

git reset --hard HEAD	reset current branch to the last commit
git reset --hard HEADˆ	reset current branch to last commit next to last
git reset --hard HEAD˜*n*	reset current branch to state before *n* commits
git reset --hard *commit*	reset current branch to given commit

Working with branches

git branch	show list of local branches
git branch -r	show list of remote branches
git branch -a	show list of local and remote branches
git checkout *branch*	change to given branch
git checkout -b *branch*	fork current branch to the given name
git merge *branch*	merge given branch into current
git rebase -i *branch*	rebase interactively on given branch

Working with remotes

git pull	merge changes from configured remote
git push	merge changes into configured remote

Rebasing (see appendix D for more information)

git rebase -i *branch*	rebase current tree on given branch
git rebase --continue	continue rebase after conflict resolution
git rebase --abort	abort current rebase and revert tree to previous state

C.1 Graphical Git tools

Some users will be pleased to know that beside the command line tool, various graphical alternatives exist. Tools with a graphical user interface are especially useful to visualize a Git repository, showing individual commits and elements like branches. This can be very helpful when learning Git.

The first of these tools was *gitk*. Being written in tcl/tk, it looks a bit dated on a modern desktop. However, it still offers the largest amount of features.

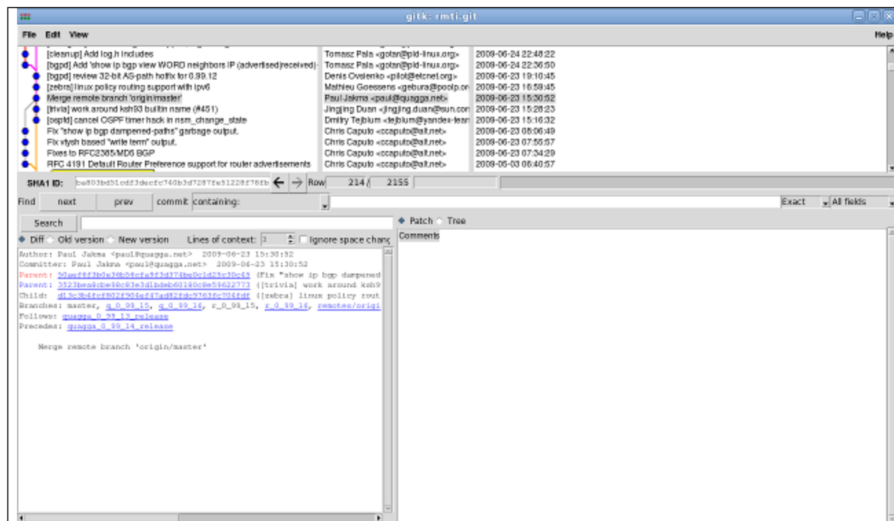


Figure 14: Screenshot of gitk

In the meantime a few more userfriendly tools such as *gitg*[32] and *Giggle*[33] have been released.

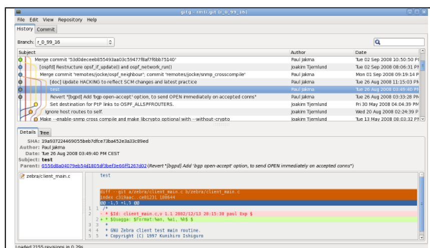


Figure 15: Screenshot of gitg

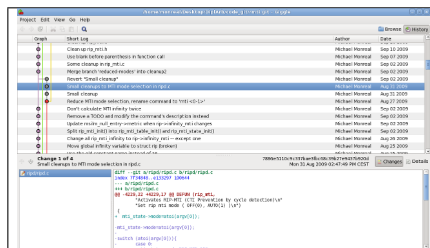


Figure 15: Screenshot of giggle

Many IDEs like *Eclipse* also provide direct access to Git repositories using a plugin[34].

[32] gitg - `http://trac.novowork.com/gitg/`

[33] Giggle - `http://live.gnome.org/giggle/`

[34] Eclipse Team provider for Git - `http://www.eclipse.org/egit/`

D Rebasing with Git

This appendix consists of a short guide which describes the process of rebasing the RMTI source code on a newer Quagga release. Notes on this topic can also be found on the RMTI project wiki[35].

Rebasing can be seen as the process of applying local changes of a forked project to a newer version of the original code. To clarify, imagine the following example:

Some upstream project containing the following series of commits:

`7b7bd - a9e97 - 3e476`

After forking the project, we add three new commits:

`7b7bd - a9e97 - 3e476 - 904cd - b6a8f - 0603e`

In the meantime, the upstream project itself has added three new commits to their repository as well:

`7b7bd - a9e97 - 3e476 - a2ba3 - 46e84 - e6e98`

A rebase now does the following:

1. save the local changes since the fork
2. revert history to the last common commit
3. pull in all the new upstream commits to a specified point
4. re-apply the saved local changes

Eventually, the result should be:

`7b7bd - a9e97 - 3e476 - a2ba3 - 46e84 - e6e98 - dc409 - f8a6b - e3060`

Note that the commit IDs of the re-applied commits have *changed* from `904cd-b6a8f-0603e` to `dc409-f8a6b-e3060`. In other words: the rebase operation has changed history of the forked project. This is an important detail because it has the potential to cause some bad headaches. Basicly, everything is fine if the rebase is only used on a private working branch. As soon as the branch is public, a rebase should *not* be done anymore. Doing it anyway will cause problems for those who have branched from the repository because the common history (as expressed by the commit IDs) will be gone.

This is part of the reason for our special repository layout, described in appendix B. To prevent inconsistencies in the commit history, we create a branch for every base release and only rebase into these new branches.

[35] RMTI project - `http://git.uni-koblenz.de/rmti/pages/Home`

D.1 Rebase to a newer Quagga release

First, make sure that you have set up your Git repository as described in appendix B as we will need the integration of the upstream repository. The example shows a rebase from any previous version to the 0.99.16 release.

At first, look for the name (called *tag*) of the version you want to rebase to:

```
$ git tag
```

You will find the tag *quagga_0_99_16_release*. Check it out as a branch, then create and switch to a fresh RMTI branch which reflects the version of the target release:

```
$ git checkout -b r_0_99_16
$ git checkout quagga_0_99_16_release -b q_0_99_16
$ git checkout r_0_99_16
```

Finally, start the rebase operation:

```
$ git rebase -i q_0_99_16
```

In many cases, the rebase will finish successfully without any additional user input. In other cases, manual conflict resolution may be needed.

D.2 Conflict resolution

If Git is unable to resolve conflicts itself, it will stop the rebase process and ask for your help. In practice, this will only be the case if both branches include changes to the *same* line somewhere.

To find the files which need fixing, run `git status` and look for *Changed but not updated*. Open the files listed here one after another and look for the conflict marker "<<<<<". Below the marker, you will see the upstream version of the code followed by your own version of the code. Either decide to go with one of those versions or add an entirely new piece of code. Look for additional conflicts inside the file and fix them using the same method. Save the file and `git add` it to the index.

After all conflict markers have been removed, continue the rebase process:

```
$ git rebase --continue
```

Alternatively, you can also abort the process at any point. The local repository will be reverted to the state before the rebase was started:

```
$ git rebase --abort
```

Finally, there are two useful aliases from [20] which can be used to automate the conflict resolution process a bit. Just put the following lines into *~/.gitconfig*:

```
[alias]
  fix-unmerged = "!f() { git ls-files --unmerged | cut -f2 | sort -u ; }; vim ‘f‘"
  add-unmerged = "!f() { git ls-files --unmerged | cut -f2 | sort -u ; }; git add ‘f‘"
```

E Quagga coding style

The Quagga project asks to use a special coding style for all new contributions. They describe it as *"GNU style with tabs set to two spaces"*. What does this mean exactly and how should it be used it in practice?

E.1 Tabs versus spaces

Most importantly, everyone working on Quagga (including RMTI) should configure their editor to insert two *space* characters instead of a *tab* character. This simple convention ensures that the code will have the same basic look at different places, thus improving readability. In case of the Vim[36] editor, this can be achived by setting the options in configuration file *~/.vimrc* as shown in listing 14.

```
set tabstop=2
set shiftwidth=2
set expandtab
```

Listing 14: vimrc

Graphical editors often expose these settings in a preferences window. Development environments like Eclipse[37] or Emacs[38] have similar options. The `man` pages usually offer all the information needed.

E.2 GNU coding style

The GNU coding style is described in detail by [21]. A slightly digested guide can also be found in [30]. Please refer to these sources for in-depth information, the following is just a practical example showing the basics.

```
1  static int
2  do_it (int pa, int ra, int me, int ter)
3  {
4    int num = num + 1;
5
6    /* a comment */
7    if (! exp)
8      {
9        ...
10     }
11   else
12     {
13       ...
14     }
15
16   return 0;
17 }
```

Listing 15: Example of the GNU coding style

[36] Vim editor - `http://www.vim.org/`
[37] Eclipse IDE - `http://www.eclipse.org/`
[38] GNU Emacs - `http://www.gnu.org/software/emacs/`

To sum up the most important parts shown in listing 15:

- function type and return value go to a separate line (see lines 1 and 2)
- one blank between operators (see line 4)
- comments go to a separate line, not the line ending (see line 6)
- opening brackets go to a separate line (see line 8)
- all blocks except the top level blocks get indented by two spaces for opening and closing brackets (compare line 3 to line 8)

E.3 Automatic code formatting

To automatically apply the correct coding style to an existing C source file, use the GNU tool `indent` like this:

```
$ indent -nut <file>
```

This command has been used on the file *rip_mti.c* as well as all the RMTI-specific code in other files. Note that none of the Quagga code was reformatted in order to minimize the differences and not jeopardize future rebase operations.

F Quagga coding introduction and examples

This appendix features a series of small examples showing the basic implementation of various functionality within Quagga. It is targeting people not yet familiar with the codebase. All given line numbers are subject to change between releases but should still give a basic idea.

F.1 Debug output

At times it can be very helpful to have some additional debug output in the logfile. Thankfully, the Quagga framework has a very easy way to do this:

```
zlog_debug ("My debug message");
```

Calls of the *zlog_debug* function will result in a new line being added to the respective log file. In the case of `ripd`, the file is called *ripd.log* by default.

It has proven useful to prefix the actual debug message with a short but easy to recognize pattern like *XXX*. This allows using the `grep` command to minimize the log file by filtering out other messages:

```
$ cat ripd.log | grep 'XXX'
```

F.2 Implementing new command line options

All Quagga daemons support parsing information given by command line options during startup. New options for `ripd` can be added in *rip_main.c*. The following example adds the flag *-t*, which just prints a line of text. Other functions can be invoked just as easily.

Around line 52, add a line describing the new flag to the *longopts* array:

```
{ "test", no_argument, NULL, 't' },
```

The first attribute is the name of the option, wrapped in double quotes. The second attribute determines if the specified command line option requires an argument (*required_argument*) or not (*no_argument*). The final attribute is a short notation. Further down in the file, around line 49, *static void usage* contains the description of the command line options:

```
-t, --test          Test option\n\
```

To actually parse the new flag, it needs to be added to *getopt_long*, which can be found around line 210. For options that require arguments, a colon needs to be added behind the short name:

```
opt = getopt_long (argc, argv, "df:i:hA:P:u:g:rvC:t", longopts, 0);
```

Finally, the actual functionality is called inside a switch statement. Add the new code in a new *case* around line 250:

```
case 't':
  printf ("Test\n");
  exit (0);
  break;
```

Listing 16: New command line option, case statement

It is important to note that the RIP data structures cannot be accessed here because they do not exist at this point. In order to use a command line option to set RIP values, a temporary global variable has to be created and used during initialization. Alternatively, consider using configuration file commands as shown below in such a case.

F.3 Implementing new commands

All Quagga daemons support a number of commands which can either be used as part of the configuration file or using Telnet/VTYSH. New commands to control `ripd` can be added in *ripd.c*. The following example adds the command *test* which just prints a line of text. Other functions can be invoked just as easily.

All commands need to be implemented as DEFUN (define function) macros. For the example command, just add:

```
DEFUN (test,
       test_cmd,
       "test",
       "Description of test command\n")
{
  vty_out (vty, "Test", VTY_NEWLINE);
  return CMD_SUCCESS;
}
```

Listing 17: New command, DEFUN

The DEFUN macro includes the name and a command alias, followed by help text and the actual implementation of the command.

To make the command available in Quagga, look for the comment */* Install rip commands */* in *rip_init (void)* and add the command to the *VIEW* and *ENABLE* nodes:

```
install_element (VIEW_NODE, &test_cmd);
install_element (ENABLE_NODE, &test_cmd);
```

The nodes concept of Quagga is described in section 4.1.

F.4 Further information

For in-depth information about the internals of Quagga, the *Zebra Hacking How-To* [29] from 2001 is still a very good resource. While being based on Zebra release 0.91, most parts are still valid for Quagga. The howt-to covers topics like the Zebra thread model, hashing and modifying the routing table.

G Creating packages for Linux systems

In order to make RMTI available to a broad audience, binary packages for some popular Linux distributions like Ubuntu[39] and Fedora[40] are offered for download on the RMTI webpage.

The Quagga packages already shipping with most distributions serve as a good starting point for creating RMTI packages. This appendix briefly describes how these packages can be adapted for RMTI.

Regardless of the target distribution, the first step is creating a clean tape archive containing the modified Quagga codebase. From the RMTI source directory, run:

```
$ autoreconf -i
$ ./configure --prefix=/usr
$ make dist
```

A distributable source archive called *quagga-x.y.z.tar.gz* can now be found in the root of the source tree.

Note that the following sections require you to have access to a system running Ubuntu or Fedora, respectively. If no such system is available, a Live CD will work just as well. It is even possible to boot the Live CD images using a virtual machine hypervisor like KVM[41]:

```
$ qemu-img create swap.img 2G
$ mkswap swap.img
$ qemu-kvm -m 1024 -usbdevice tablet --cdrom live.iso -hda swap.img
```

G.1 Package for Ubuntu and Debian systems

The Debian package has been built on a virtual machine running a Ubuntu 10.04 LTS system and should also work on newer releases as well as the latest Debian *testing* distribution.

```
$ sudo apt-get build-dep quagga
$ apt-get source --download-only quagga
```

You should now find a tarball containing the original source code (*.orig.tar.gz), a description file (*.dsc) as well as a patch file (*.diff.gz) containing the Debian build system and Debian-specific patches in the current directory. To proceed, you only need the last-mentioned item in addition to a tarball containing the RMTI sources.

In this example, `apt-get` deliverd the sources for Quagga 0.99.15 while the RMTI version is based on Quagga 0.99.16. In such a case, the debian build directory needs to be moved to the correct destination first:

[39] Ubuntu - `http://www.ubuntu.com/`
[40] Fedora - `http://www.fedoraproject.org/`
[41] KVM = Linux **K**ernel-based **V**irtual **M**achine - `http://www.linux-kvm.org/`

```
$ tar -xvzf rmti-0.99.16.tar.gz
$ zcat *diff.gz | patch -p0
$ mv quagga-0.99.15/debian quagga-0.99.16/
$ rm *dsc *gz quagga-0.99.15
$ cd quagga-0.99.16
```

Note that apt-get will most likely download outdated or unwanted patches. Those can now be removed from *debian/patches/* and *debian/patches/00list*, respectively.

In any case, add a new entry matching the version of the RMTI tarball to *debian/changelog*. The version string is followed by a dash and an epoch after which 'rmti' can be added to mark the build as RMTI-enabled. The entry should look similar to this:

```
quagga (0.99.16-1rmti) stable; urgency=high

  * New upstream release
  * RMTI

 -- Michael Monreal <monreal@uni-koblenz.de>  Wed, 28 Jul 2010 ↩
    10:45:25 +0200
```

Listing 18: ChangeLog entry for the Debian package

Now the package build process can be started:

```
$ CFLAGS="-g -O2 -std=c99" dpkg-buildpackage -rfakeroot -b
```

The package can be found in the parent directory as soon as the build process has completed successfully.

If the build fails on *quagga.pdf*, take out the references to that file from *debian/rules* and *debian/quagga-doc.docs* and retry.

G.2 Package for Fedora and Red Hat systems

The RPM package has been built on a virtual machine running a Fedora 13 system. It should run on newer releases and may also run on certain versions of Red Hat Enterprise Linux or CentOS.

First make sure that all dependancy packages are installed and set up the environment for RPM development. This can be achieved by running the following sequence of commands:

```
$ su
# yum groupinstall development-tools
# yum install rpmdevtools yum-utils
# yum-builddep quagga
$ rpmdev-setuptree
```

This will create the directory *rpmbuild/* containing the RPM setup tree inside your home directory.

Now you need to get some files from the Fedora package Git repository[42]. You have the choice to take these files from the latest release (currently this is the *f13/master* branch) or a newer version from the testing distribution (currently this is the *f14/master* branch).

First get the *quagga-filter-perl-requires.sh* script as well as any patches you want to include and put them into *rpmbuild/SOURCES/*. Put the RMTI source tarball here as well.

The package specification file *quagga.spec* is needed in *rpmbuild/SPECS/*. Make sure the version tag matches the version of your tarball. Append 'rmti' to the release tag in front of %{?dist}, then comment out any patches you do not want to include and add *-std=c99* to the line containing the CFLAGS. Finally, add a ChangeLog entry at the bottom similar to this:

```
* Wed Jul 28 2010 Michael Monreal <monreal@uni-koblenz.de> - ↩
    0.99.16-0rmti
- New upstream release
- RMTI
```

Listing 19: ChangeLog entry for the RPM package

Now the package build process can be started:

```
$ cd ~/rpmbuild/SPECS/
$ rpmbuild --bb --clean quagga.spec
```

The package can be found in *rpmbuild/RPMS/* as soon as the build process has completed successfully.

[42]http://pkgs.fedoraproject.org/gitweb/?p=quagga.git;a=summary

H Patch: RMTI support for Zimulator

Marcel Jacob originally designed his Zimulator tool for RIP only. Because of this, the class handling RIP packets had the infinity value hardcoded to 16. While RMTI uses the same packet format, it commonly uses a higher infinity metric like 64. In order to allow Zimulator to work with RMTI regardless, I added a way to set the correct metric using the command line switch *-I*.

```
Index: zimulator.pl
===================================================================
--- zimulator.pl        (revision 72)
+++ zimulator.pl        (working copy)
@@ -126,6 +126,7 @@
 $verboseUsageString .= "    -T              : Only take Packets [...]
 $verboseUsageString .= "    -F              : Only take Packets [...]
 $verboseUsageString .= "    -i              : Generates image of graph and ↩
     writes it to png file.\n\n";
+$verboseUsageString .= "    -I inf_metric   : Assume the provided value as ↩
    infinity (for RMTI).\n\n";

 $verboseUsageString .= "Topology Types:\n---------------\n";
 $verboseUsageString .= "When generating topologys there are the following ↩
     types available:\n\n";
@@ -215,7 +216,7 @@

 # get options with Getopt module
 #getopts('s:Sr:aPzxg:CHVhAvc:o:fTF', \%opts);
-getopts('s:Sr:azxg:CHVhAvc:o:fTFi', \%opts);
+getopts('s:Sr:azxg:CHVhAvc:o:fTFiI:', \%opts);

 # prepare all given files that are not catched by getopts
 # TODO: rueckgabewert fuer jede datei pruefen. wenn 0 => nicht hinzufuegen
@@ -247,6 +248,11 @@
 my $configuration = Configuration::instance();
 $configuration->setOption("VERBOSE",$opts{"v"});
 $configuration->setOption("OUTPUT_FILENAME",$opts{"o"});
+if($opts{"I"}){
+  $configuration->setOption("INFINITY_METRIC",$opts{"I"});
+} else {
+  $configuration->setOption("INFINITY_METRIC",16);
+}
 my $verbose = $opts{"v"};
 if($opts{"i"} and ($opts{"g"} or $opts{"z"})){
     $configuration->setOption("CREATEGRAPHIMAGE",1);
Index: modules/RIPParser.pm
===================================================================
--- modules/RIPParser.pm        (revision 72)
+++ modules/RIPParser.pm        (working copy)
@@ -317,6 +317,7 @@
     my $timeoutTime = $config->getOption("TIMEOUT_TIMER")*1000000;
     my $garbageTime = $config->getOption("GARBAGE_TIMER")*1000000;
 #    print "timeout: $timeoutTime, garbage: $garbageTime\n";
+    my $infMetric = $config->getOption("INFINITY_METRIC");

     foreach my $packet (@packets) {
         my $router = $packet->getRouter();
@@ -331,7 +332,7 @@
                 my ($routerNumber) = $r =~ m/^r([0-9]+)/;
                 next if $router =~ /10\.0\.[0-9]+\.$routerNumber/;

-                my ($oldMetric,$oldRouter,$oldTime) = (16,"",0);
+                my ($oldMetric,$oldRouter,$oldTime) = ($infMetric,"",0);
                ($oldMetric, $oldRouter, $oldTime) = @{$routerTables[↩
                    $routerNumber]->{$netIP}} if ref $routerTables[↩
                    $routerNumber]->{$netIP};
                 die "not defined oldmetric" unless defined $oldMetric;
                 die "not defined oldrouter" unless defined $oldRouter;
```

Listing 20: RIP infinity patch for RMTI in Zimulator

I Patch: Counting-to-infinity generator

```
diff --git a/ripd/ripd.c b/ripd/ripd.c
index cdbf892..fd30b5f 100644
--- a/ripd/ripd.c
+++ b/ripd/ripd.c
@@ -60,6 +60,16 @@ struct route_table *rip_neighbor_table;
 /* RIP route changes. */
 long rip_global_route_changes = 0;

+#ifdef ROUTE_MANIPULATION
+int cti_route_manipulation = 0;
+
+char cti_incoming_eth[6] = "eth1";
+char cti_blocking_eth[6] = "eth2";
+
+int cti_incoming_idx = 0;
+int cti_blocking_idx = 0;
+#endif /*ROUTE_MANIPULATION*/
+
 /* RIP queries. */
 long rip_global_queries = 0;

@@ -208,6 +218,20 @@ rip_timeout (struct thread *t)
   rinfo->metric = rip->infinity_metric;
   rinfo->flags &= ~RIP_RTF_FIB;

+#ifdef ROUTE_MANIPULATION
+  if (cti_route_manipulation)
+    {
+      unsigned int cti_ifindex = ifname2ifindex (cti_incoming_eth);
+
+      if (!rinfo->cti_flag
+          && rinfo->ifindex == cti_ifindex)
+        {
+          rinfo->cti_flag = 1;
+          zlog_debug ("CTI (timeout): set cti_flag=%d", rinfo->cti_flag);
+        }
+    }
+#endif /*ROUTE_MANIPULATION*/
+
   /* - The route change flag is to indicate that this entry has been
      changed. */
   rinfo->flags |= RIP_RTF_CHANGED;
@@ -535,6 +559,41 @@ rip_rte_process (struct rte *rte, struct sockaddr_in *↩
    from,

   if (rinfo)
     {
+#ifdef ROUTE_MANIPULATION
+      if (cti_route_manipulation)
+        {
+          unsigned int cti_ifindex = ifname2ifindex (cti_incoming_eth);
+
+          zlog_debug ("CTI (rte): if=%d/%d/%d m=%d c_f=%d",
+                       ifp->ifindex, cti_ifindex, rinfo->ifindex,
+                       rte->metric, rinfo->cti_flag);
+
+          /* Set flag if all interfaces match and metric is infinity. */
+          if (!rinfo->cti_flag
+              && ifp->ifindex == cti_ifindex && ifp->ifindex == rinfo->↩
    ifindex
+              && rte->metric >= rip->infinity_metric)
+            {
+              rinfo->cti_flag = 1;
+              zlog_debug ("CTI (rte): set cti_flag=%d", rinfo->cti_flag);
+            }
+
+          /* Clear flag if route information did not arrive over the CTI's
+             incoming interface and it has a new learned-from address. */
+          if (rinfo->cti_flag
+              && !IPV4_ADDR_SAME(&rinfo->from, &from->sin_addr)
```

```
+               && ifp->ifindex != cti_ifindex)
+             {
+               rinfo->cti_flag = 0;
+               zlog_debug ("CTI (rte): set cti_flag=%d", rinfo->cti_flag);
+
+               /* Turn off CTI generator. */
+               cti_route_manipulation = 0;
+               zlog_debug ("CTI (rte): set cti_route_manipulation=%d",
+                           cti_route_manipulation);
+             }
+         }
+#endif /*ROUTE_MANIPULATION*/
+
        /* Local static route. */
        if (rinfo->type == ZEBRA_ROUTE_RIP
            && ((rinfo->sub_type == RIP_ROUTE_STATIC) ||
@@ -2401,6 +2460,13 @@ rip_output_process (struct connected *ifc, struct ↩
     sockaddr_in *to,
          subnetted = 1;
     }

+#ifdef ROUTE_MANIPULATION
+  unsigned int cti_ifindex;
+
+  if (cti_route_manipulation)
+    cti_ifindex = ifname2ifindex (cti_blocking_eth);
+#endif /*ROUTE_MANIPULATION*/
+
   for (rp = route_top (rip->table); rp; rp = route_next (rp))
     if ((rinfo = rp->info) != NULL)
       {
@@ -2408,6 +2474,17 @@ rip_output_process (struct connected *ifc, struct ↩
     sockaddr_in *to,
         /* that have the same mask as the output "interface". For other ↩
               */
        /* networks, only the classfull version is output. ↩
                             */

+#ifdef ROUTE_MANIPULATION
+  if (cti_route_manipulation
+      && rinfo->cti_flag
+      && ifc->ifp->ifindex == cti_ifindex
+      && rinfo->metric >= rip->infinity_metric)
+    {
+      zlog_debug ("CTI (output): inhibiting update");
+      continue;
+    }
+#endif /*ROUTE_MANIPULATION*/
+
        if (version == RIPv1)
          {
            p = (struct prefix_ipv4 *) &rp->p;
@@ -3588,6 +3665,42 @@ DEFUN (no_rip_distance_source_access_list,
   rip_distance_unset (vty, argv[0], argv[1], argv[2]);
   return CMD_SUCCESS;
 }
+
+#ifdef ROUTE_MANIPULATION
+DEFUN (rip_cti,
+       rip_cti_cmd,
+       "cti <0-255> <0-255>",
+       NO_STR
+       "Interface over which the CTI route comes in\n"
+       "Number of the eth interface\n"
+       "Interface over which the CTI route should be blocked\n"
+       "Number of the eth interface\n")
+{
+  cti_incoming_idx = atoi (argv[0]);
+  cti_blocking_idx = atoi (argv[1]);
+
+  memset(cti_incoming_eth, 0, strlen (cti_incoming_eth));
+  memset(cti_blocking_eth, 0, strlen (cti_blocking_eth));
+
+  sprintf (cti_incoming_eth, "eth%d", atoi (argv [0]));
```

```
+  sprintf (cti_blocking_eth, "eth%d", atoi (argv [1]));
+
+  cti_route_manipulation = 1;
+
+  return CMD_SUCCESS;
+}
+
+DEFUN (rip_no_cti,
+       rip_no_cti_cmd,
+       "no cti",
+       NO_STR
+       "no cti will be generated\n")
+{
+  cti_route_manipulation = 0;
+
+  return CMD_SUCCESS;
+}
+#endif /*ROUTE_MANIPULATION*/

 /* Print out routes update time. */
 static void
@@ -3861,6 +3974,14 @@ config_write_rip (struct vty *vty)
                      VTY_NEWLINE);
        }

+#ifdef ROUTE_MANIPULATION
+  if (cti_route_manipulation)
+    {
+      vty_out (vty, " cti %d %d%s",
+               cti_incoming_idx, cti_blocking_idx, VTY_NEWLINE);
+    }
+#endif /*ROUTE_MANIPULATION*/
+
      /* Redistribute configuration. */
      config_write_rip_redistribute (vty, 1);

@@ -4412,6 +4533,10 @@ rip_init (void)
   install_element (RIP_NODE, &no_rip_distance_source_access_list_cmd);
   install_element (RIP_NODE, &rip_mti_cmd);
   install_element (RIP_NODE, &rip_infinity_cmd);
+#ifdef ROUTE_MANIPULATION
+  install_element (RIP_NODE, &rip_cti_cmd);
+  install_element (RIP_NODE, &rip_no_cti_cmd);
+#endif /*ROUTE_MANIPULATION*/

   /* Debug related init. */
   rip_debug_init ();
diff --git a/ripd/ripd.h b/ripd/ripd.h
index 74a443a..1ede950 100644
--- a/ripd/ripd.h
+++ b/ripd/ripd.h
@@ -22,6 +22,9 @@
 #ifndef _ZEBRA_RIP_H
 #define _ZEBRA_RIP_H

+/* CTI generator. */
+#define ROUTE_MANIPULATION 1
+
 /* RIP version number. */
 #define RIPv1                            1
 #define RIPv2                            2
@@ -205,6 +208,11 @@ struct rip_info
   u_int32_t metric;

+#ifdef ROUTE_MANIPULATION
+  /* Flag to control the occurrence of CTIs. */
+  int cti_flag;
+#endif /*ROUTE_MANIPULATION*/
+
```

Listing 21: CTI generator patch

J Testomato bash script and scenario run files

```
1  #!/bin/bash
2  #
3  # Test-O-Mato: execute a series of automated test runs
4  #
5  # Usage examples:
6  #   testomato -x y.xml
7  #   testomato -x y.xml -n 10
8  #   testomato -x y.xml -r y_104030.run -n 10
9  #
10
11 # settings
12 USER="monreal"         # user owning the log files
13
14 # defaults
15 REPEAT="1"             # default repeat count
16 HOSTFS="/mnt/hostfs"   # hostfs directory on VMs
17 SMLTNS="/root/.vnuml/simulations"
18
19 # files needed on the VMs
20 BIN="bin/*"
21 CFG="etc/*"
22 HLP="helpers/*"
23
24 # commands
25 VNUML="vnumlparser.pl -v -u root"
26
27 # check for root user
28 if [ `whoami` != "root" ]; then
29   echo "Error: Please su to root first!"
30   exit 1
31 fi
32
33 # check for VNUML (and assume it is working)
34 if [ -z `which vnumlparser.pl` ]; then
35   echo "Please install VNUML first!";
36   exit 1
37 fi
38
39 # command line options
40 while getopts ":x:r:n:" opt
41 do
42   case "$opt" in
43     "x")
44       XML=`basename $OPTARG .xml`
45       ;;
46     "r")
47       RUN=`basename $OPTARG .run`
48       ;;
49     "n")
50       REPEAT="$OPTARG"
51       ;;
52     "?")
53       echo "Error: unknown option $OPTARG"
54       ;;
55     ":")
56       echo "Error: no value given for option $OPTARG"
57       ;;
58     *)
59       echo "Error: unknown"
60       ;;
61   esac
62 done
63
64 if [ -z "$RUN" ]; then RUN="$XML"; fi
65 if [ -z "$XML" ]; then
66   echo "Usage: ${0##*/} -x xmlfile [-r runfile] [-n repeat count]"; exit 1;
67 fi
68
69 if [ ! -f "$XML".xml ]; then echo "Error: $XML.xml not found!"; exit 1; fi
70 if [ ! -f "$RUN".run ]; then echo "Error: $RUN.run not found!"; exit 1; fi
```

```
71
72  # check for old scenario files
73  if [ -d $SMLTNS/$XML/ ]; then
74    if [ -n "`ls $SMLTNS/$XML/`" ]; then
75      echo "Error: Scenario $XML already created, please purge it first!"
76      echo -n "Press any key to purge or <ctrl>+<c> to quit: "; read c; echo
77      $VNUML -P $XML.xml
78      exit 0
79    fi
80  fi
81
82  # uml_mconsole requires .uml directory
83  if [ ! -d /root/.uml ]; then
84    mkdir /root/.uml
85  fi
86
87  ###### VNUML ############################################################
88
89  echo "[ 1 ] Starting scenario..."
90  $VNUML -w 100 -t $XML.xml -Z >& /dev/null
91
92  echo "[ 2 ] Copying configs and scripts to the VMs..."
93  for vm in $SMLTNS/$XML/vms/*;
94  do
95    # copy stuff
96    for f in $BIN $CFG $HLP; do
97      cp $f $vm/hostfs/
98      touch $vm/hostfs/ripd.log # initialize log file
99    done
100   # load firewall module
101   uml_mconsole ../..$SMLTNS/$XML/vms/`basename $vm`/run \
102     exec "modprobe iptable_filter" >& /dev/null
103 done
104
105 echo "[ 3 ] Starting daemons..."
106 $VNUML -x start@$XML.xml >& /dev/null
107 $VNUML -x zebra@$XML.xml >& /dev/null
108 $VNUML -x   rip@$XML.xml >& /dev/null
109
110
111 ###### FUNCTIONS ########################################################
112 ## Helper scripts (see helpers/ directory) are used here beacause mconsole
113 ## does neither support quoted commands nor complex chains of commands
114 ## containing characters like ">" an "|". Also, there is no way to send
115 ## output back to the host. Originally, SSH had been used but it turned
116 ## out to be too slow.
117
118 runOn () {
119   uml_mconsole ../..$SMLTNS/$XML/vms/$1/run \
120     exec "$2" >& /dev/null
121 }
122
123 pressKey () {
124   echo; echo -n "Press any key to start test: "; read c; echo
125 }
126
127 waitFor () {
128   echo "wait for $1 seconds"; sleep $1; echo
129 }
130
131 getRoute () {
132   if [ "$1" != "host" ]; then
133     OUT="$SMLTNS/$XML/vms/$1/hostfs/route.out"
134
135     runOn $1 "$HOSTFS/helper_route.sh $1 $2"
136
137     cat $OUT 2> /dev/null
138     while [ $? == 1 ]; do
139       sleep .1
140       cat $OUT 2> /dev/null
141     done
142
143     rm $OUT
144
```

```
145   else # special case for ripd running on host system
146     ROUTE=`vtysh -d ripd -c 'show ip rip' | grep $2`
147     echo "h:  $ROUTE"
148   fi
149 }
150 
151 getMSILM () {
152   OUT="$SMLTNS/$XML/vms/$1/hostfs/msilm.out"
153 
154   echo "$1:"; runOn $1 "$HOSTFS/helper_msilm.sh"
155 
156   while [ ! -f $OUT ]; do sleep .1; done
157   sleep 1; cat $OUT; echo; rm $OUT
158 }
159 
160 getMRPM () {
161   OUT="$SMLTNS/$XML/vms/$1/hostfs/mrpm.out"
162 
163   echo "$1:"; runOn $1 "$HOSTFS/helper_mrpm.sh"
164 
165   while [ ! -f $OUT ]; do sleep .1; done
166   sleep 1; cat $OUT; echo; rm $OUT
167 }
168 
169 close () {
170   echo -n "close interface $2 on $1 and "
171   runOn $1 "iptables -A OUTPUT -o $2 -j DROP"
172   waitFor $3
173 }
174 
175 open () {
176   echo -n "opening all interfaces on $1 and "
177   runOn $1 "iptables -F OUTPUT"
178   waitFor $2
179 }
180 
181 nocti () {
182   echo
183   echo "Stopping CTI provokation..."
184   runOn "$HOSTFS/helper_nocti.sh"
185   echo
186 }
187 
188 reset () {
189   echo -n "Resetting scenario (restarting daemons)... "; echo
190 
191   for vm in $SMLTNS/$XML/vms/*;
192   do
193     VM=`basename $vm`
194     runOn $VM "ip route flush all"
195   done
196 
197   $VNUML -x  stop@$XML.xml >& /dev/null
198   $VNUML -x zebra@$XML.xml >& /dev/null
199   $VNUML -x   rip@$XML.xml >& /dev/null
200 }
201 
202 hold ( ) {
203   echo -n "Finished. Press any key to shut down: "; read c
204 }
205 
206 copyLogs ( ) {
207   RIPLOGS="$DIR/$rep"
208 
209   for vm in $SMLTNS/$XML/vms/*;
210   do
211     mkdir -p "$RIPLOGS"
212     mv $vm/hostfs/ripd.log "$RIPLOGS"/`basename $vm`-ripd.log
213     touch $vm/hostfs/ripd.log
214   done
215 }
216 
217 
218 ##### SIMULATION ###########################################################
```

```
219
220 TIME=`date +%Y%m%d-%H:%M`
221 DIR="tests/$XML ($RUN) $TIME"
222 LOG="$DIR/test.log"
223 FIFO=".fifo"
224
225 mkdir "$DIR"
226
227 if [ -e "$LOG" ]; then rm "$LOG"; fi
228 if [ ! -e $FIFO ]; then mkfifo $FIFO; fi
229
230 exec 3>&1 4>&2            # save fds for stdout (1->3) and stderr (2->4)
231 tee "$LOG" < $FIFO >&3 &  # write $FIFO to $LOG and redirect stdio to 3
232 TEE_PID=$!
233 exec > $FIFO 2>&1 # redirect stdout to $FIFO and stderr to stdout
234
235 echo
236
237 # extract settings
238 RC=`cat etc/ripd.conf | grep -v "\!"`
239
240 I=`cat etc/ripd.conf | grep -v "\!" | grep "infinity" | awk '{ print $2 }'`
241 M=`cat etc/ripd.conf | grep -v "\!" | grep "mti" | awk '{ print $2 }'`
242 T=`cat etc/ripd.conf | grep -v "\!" | grep "timers basic" | grep -v "\!"`
243
244 UT=`echo -e $T | awk '{ print $3 }'`
245 TT=`echo -e $T | awk '{ print $4 }'`
246 GT=`echo -e $T | awk '{ print $5 }'`
247
248 S=5
249 W=`expr $UT + $UT / 2`      # wait 1.5 times the update timer
250 D=`expr $TT - $S - $S - $S` # timeout timer minus 3 sleep times
251
252 echo "Infinity: $I   MTI: $M   Timers: $UT / $TT / $GT"
253
254 # run simulation
255 for rep in `seq 1 $REPEAT`; do
256   echo; echo "=== $rep / $REPEAT ====================="; echo
257
258   source $RUN.run
259
260   if [ $rep -lt $REPEAT ]; then reset; fi
261 done
262
263 exec 1>&3 3>&- 2>&4 4>&-  # restore the original fds for stdout and stderr
264 wait $TEE_PID; rm $FIFO
265
266 chown -R $USER:$USER "$DIR"
267
268
269 ###### QUIT ###############################################################
270
271 $VNUML -P $XML.xml >& /dev/null
272
273 # kill local daemons if running
274 if [ -n "`ps -e | grep ripd`" ]; then killall ripd; fi
275 if [ -n "`ps -e | grep zebra`" ]; then killall zebra; fi
276
277 # kill leftover UML processes
278 sleep 5
279 if [ -n "`ps -e | grep linux`" ]; then killall linux; fi
280
281 exit 0
```

Listing 22: testomato.sh

Sample scenario files to be used *with* the CTI generator:

```
waitFor $W

close r5 eth1 $D

for i in `seq 7`; do
  getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0; echo
  sleep $S
done

open r5 0
copyLogs
```

Listing 23: Upsilon topology (y.run)

```
waitFor $W

close r5 eth1 $D

for i in `seq 7`; do
  getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0; ↩
      getRoute r6 10.0.6.0; echo
  sleep $S
done

open r5 0
copyLogs
```

Listing 24: Extended upsilon topology (y+.run)

```
waitFor $W

close r8 eth1 $D

for i in `seq 7`; do
  getRoute r1 10.0.9.0; getRoute r2 10.0.9.0; getRoute r3 10.0.9.0; ↩
      getRoute r4 10.0.9.0; getRoute r5 10.0.9.0; getRoute r6 10.0.9.0; ↩
      echo
  sleep $S
done

open r8 0
copyLogs
```

Listing 25: Circle topology (c6.run)

Sample scenario files to be used *without* the CTI generator:

```
waitFor $W

close r5 eth1 $D

getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0; echo

sleep $S

close r1 eth2 5

for i in `seq 2`; do
  getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0
  echo
  sleep 7
done

open r1 5

for i in `seq 5`; do
  getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0
  echo
  sleep 2
done

open r5 0
copyLogs
```

Listing 26: Upsilon topology for timers 10/40/30 (m-y_104030.run)

```
waitFor $W

getRoute r1 10.0.6.0
getRoute r2 10.0.6.0
getRoute r3 10.0.6.0
getRoute r5 10.0.6.0
echo

close r5 eth1 20 $D

getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0; getRoute r6 10.0.6.0; echo;

sleep $S

close r6 eth0 0; close r6 eth1 0
close r2 eth3 0
close r3 eth3 0

for i in `seq 2`; do
  getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0; getRoute r6 10.0.6.0; echo
  sleep 7
done

open r6 0
open r2 0
open r3 1

for i in `seq 5`; do
  getRoute r1 10.0.6.0; getRoute r2 10.0.6.0; getRoute r3 10.0.6.0; getRoute r6 10.0.6.0; echo
  sleep 2
done

open r5 0
copyLogs
```

Listing 27: Extended upsilon topology for timers 10/40/30 (m-y+_104030.run)

Listing

1 Sample of a zebra.conf file 17
2 Sample of a ripd.conf file 17
3 Structure of the Quagga source code 18
4 Files inside the *ripd* directory 19
5 MRPM Table 25
6 MSILM Table 26
7 Adaption of the garbage collection timer 28
8 Adaption of the execution time 28
9 Testomato output showing a CTI 33
10 Testomato output showing no CTI (shortened) 34
11 CTI patch, part 1 35
12 CTI patch, part 2 35
13 CTI patch, part 3 36
14 vimrc 57
15 Example of the GNU coding style 57
16 New command line option, case statement 60
17 New command, DEFUN 60
18 ChangeLog entry for the Debian package 62
19 ChangeLog entry for the RPM package 63
20 RIP infinity patch for RMTI in Zimulator 64
21 CTI generator patch 65
22 testomato.sh 68
23 Upsilon topology (y.run) 72
24 Extended upsilon topology (y+.run) 72
25 Circle topology (c6.run) 72
26 Upsilon topology for timers 10/40/30 (m-y_104030.run) 73
27 Extended upsilon topology for timers 10/40/30 (m-y+_104030.run) 73

List of Figures

1 Topology with two-hop loop 10
2 A simple loop and a source loop [3] 12
3 Transitions between modes in Quagga daemons 16
4 Branch history of the RMTI repository 23
5 XTPeer client/server architecture 24
6 Reduction of RMTI code size 29
7 Elements of a topology visualization 38
8 The upsilon topology 39
9 The circle topology 41
10 The extended upsilon topology 43
11 RMTI website 45
11 Discussion board 45
12 Performance of different host kernel versions 49
13 Interaction between the Git repositories 51
14 Screenshot of gitk 54
15 Screenshot of gitg 54
15 Screenshot of giggle 54

References

[1] Frank Bohdanowicz. Weiterentwicklung und Implementierung des RIP-MTI-Routing-Daemons. Diplomarbeit, Universität Koblenz-Landau, 2008.

[2] Frank Bohdanowicz, Harald Dickel, and Christoph Steigner. Adoidance of Routing Loops. 2009. Arbeitsberichte aus dem Fachbereich Informatik Nr. 01/2009.

[3] Frank Bohdanowicz, Harald Dickel, and Christoph Steigner. Routing with Metric-based Topology Investigation. 2009. IARIA Journal.

[4] Frank Bohdanowicz, Marcel Jakobs, and Christoph Steigner. Statistical Convergence Analysis of Routing Algorithms. 2010. ICN Paper.

[5] Scott Chacon. Git Community Book. `http://book.git-scm.com/` - last visited 2010-11-19.

[6] Jeff Dike. Posting on user-mode-linux-devel. `http://www.mail-archive.com/user-mode-linux-devel@lists.sourceforge.net/msg06682.html` - last visited 2010-11-19.

[7] Jeff Dike. *User Mode Linux*. Prentice Hall, 2006.

[8] Free Software Foundation. GPLv3 Discussion Draft FAQ. `http://gplv3.fsf.org/dd3-faq` - last visited 2010-11-19.

[9] Free Software Foundation. GNU General Public License, Version 2. `http://www.gnu.org/licenses/gpl-2.0.html` - last visited 2010-11-17, 1991.

[10] Andreas Garbe. Simulation großer Netzwerke in der VNUML-Umgebung. Diplomarbeit, Universität Koblenz-Landau, September 2010.

[11] Christopher Israel. Diplomarbeit, Universität Koblenz-Landau, 2011.

[12] Marcel Jakobs. Statistische Konvergenzanalyse des RIP Routingprotokolls. Diplomarbeit, Universität Koblenz-Landau, 2010.

[13] Tim Keupen. Generierung von Testfällen für den RIP-MTI Algorithmus. Diplomarbeit, Universität Koblenz-Landau, 2007.

[14] Milad Khojasteh. Erreichbarkeitsbestätigungen in Routing Algorithmen. Bachelorarbeit, Universität Koblenz-Landau, 2010.

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

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

[17] Stefan Lange. Zentrale Betrachtung von Routing-Informationen zur Analyse des Konvergenzverhaltens verschiedener RIP-Algorithmen und Unterstützung des Generierens von Testfällen. Diplomarbeit, Universität Koblenz-Landau, 2007.

[18] Michael Monreal. Simulation mit VNUML. Studienarbeit, Universität Koblenz-Landau, 2007.

[19] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach, 4th Edition*. Morgan Kaufmann, 2007.

[20] Git project. Aliases. `https://git.wiki.kernel.org/index.php/Aliases` - last visited 2010-11-17.

[21] GNU project. GNU Coding Standards. `http://www.gnu.org/prep/standards/standards.html` - last visited 2010-11-17.

[22] Quagga project. Quagga manual. `http://quagga.net/docs/quagga.html` - last visited 2010-11-19.

[23] Quagga project. VNUML language reference. `http://www.dit.upm.es/vnumlwiki/index.php/Reference` - last visited 2010-11-19.

[24] Quagga project. Website. `http://www.quagga.net/` - last visited 2010-11-17.

[25] VNUML project. Website. `http://www.dit.upm.es/vnuml/` - last visited 2010-11-17.

[26] Daniel Pähler. Extern steuerbare Routing-Updates im RIP-Daemon der Quagga-Programmsuite. Diplomarbeit, Universität Koblenz-Landau, 2006.

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

[28] Ansgar Taflinski. Bachelorarbeit, Universität Koblenz-Landau, 2010.

[29] Yon Uriarte. Zebra Hacking How-To. 2001. `http://quagga.net/zhh.html` - last visited 2010-11-17.

[30] Wikipedia. GNU Coding Standards. `http://en.wikipedia.org/wiki/GNU_Coding_Standards` - last visited 2010-11-17.

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