Implementing path-exploration damping in the Quagga Software Routing Suite Version 0.99.10

Mattia Rossi
Centre for Advanced Internet Architectures, Technical Report 081117A
Swinburne University of Technology
Melbourne, Australia
mrossi@swin.edu.au

***Abstract*—Quagga is a software routing suite which provides implementations of various routing protocols for UNIX based platforms. It supports implementations of RIP, OSPF and BGP version 4. Path-exploration damping is one possible approach to filter out the path hunting phenomenon in BGP, in which a single prefix withdrawal event at the original announcement's origin may generate a large volume of subsequent updates as the routing system converges. Implementing path-exploration damping in Quagga, using version 0.99.10, results in a fully operating routing suite which makes use of this technique.**

I. Introduction

Quagga is a collection of daemons, each of them representing a routing protocol and exchanging routing information with peers speaking the same protocol. All of them are hold together by an additional core daemon, the zebra daemon, which installs the learned routes into the kernel and manages static routes. This technical report will explain parts of the Quagga version 0.99.10 BGP implementation (the bgpd daemon), and the necessary changes made in order to get a working routing suite which implements the technique of path-exploration damping. As Quagga is Open Source Software (OSS) written in C it is possible to browse through the source code and try to understand it with the help of a lot of inline comments. Quagga can be found at [1] and includes a slightly outdated but still helpful documentation. It also comes with its own command line interface (CLI) for configuration which is similar to the CLI in Cisco equipment. The latest Quagga release is version 0.99.11, but as this work started prior to that release, version 0.99.10 is used, which is still available form the download section of the Quagga homepage. The path-exploration damping algorithm implemented and described in this report, is based on Geoff Huston's [2] analysis of BGP update messages [3] and is also explained in his ISP column [4]. An implementation is available in form of a Quagga patch at [5]. In short, the intention of the algorithm is to alter the MRAI behaviour such that the MRAI timer is applied on a per-prefix basis, and the MRAI damping period is extended across multiple MRAI intervals for as long as successive updates to a prefix extend the ASPath length. The intention is to use this altered MRAI algorithm to selectively dampen BGP's "path hunt to withdrawal" behaviour, and thereby reduce the BGP update rate without altering the underlying BGP information propagation characteristics.

II. Current Implementation of the MRAI Timer in Quagga

The BGP version 4 standard is described in RFC 4271 [6] which amongst the protocol definitions also suggests the use of an MRAI timer and its standard values for eBGP and iBGP sessions set to 30 seconds and 5 seconds respectively. A current Internet draft exists, which intends to redefine the intervals [7] to lower values, while the path-exploration damping idea yet aims to set the MRAI timer to 0. The algorithm instead introduces a suppression timer which uses the value originally used for the MRAI timer in the BGP configuration.

The MRAI timer defines the minimum time interval between successive advertised updates of a prefix to a peer, where with peer every BGP speaker connected to the sending BGP speaker is intended. In paragraph 9.2.1.1 [8] of RFC 4271 the MRAI timer suggestion is described. It states, that an MRAI timer should be defined on a per peer basis, but applied on a per destination basis and it also explicitly states that the timer should affect updates as well as withdrawals, where RFC 1771 was explicit about MRAI applying to updates and mute about its application to withdrawals. The current Quagga implementation instead deploys a so called "burst" MRAI timer where all updates to a given

peer are held until the next MRAI timer interval expires, at which time all queued updates are announced. Quagga supports this “burst” behaviour, and the practice seems to be quite common also in various other implementations. The cause could not be traced to its origin, but current maintainers of this implementations (including Quagga) guess that this solution has been chosen initially because of its implementation simplicity. Quagga also applies the timer only on updates not on withdrawals, as this part of the implementation has not been updated after obsoleting RFC 1771. The memory and CPU overhead of the path-exploration damping (including per peer per prefix MRAI) implementation is discussed in section VI-A.

III. The Short Version of Path-Exploration Damping

Already in the year 2000 Craig Labovitz et al. [9] demonstrated the problem of BGP convergence due to path hunting. In [4] Geoff Huston proposes to apply a selective heuristic to the BGP update stream in order to attempt to remove these path exploration updates from the stream. This section tries to give a short overview of the whole idea.

A. The Path Hunting Problem

The whole path hunting phenomenon can be explained quickly using Figure 1. When router 1 becomes unreachable, router 5 sends 2 update messages and 3 withdrawals (2 preceding the update messages) of which only the last withdrawal is of any use for the peering routers. This results in unnecessary network traffic, and unnecessary use of resources in routers adjacent to router 5. While the path-exploration damping algorithm tries mainly to address this additional use of resources, it might also have some beneficial effects on the convergence time of routers. Additional thoughts to path-hunting related issues can be found in the BGP stability draft [10].

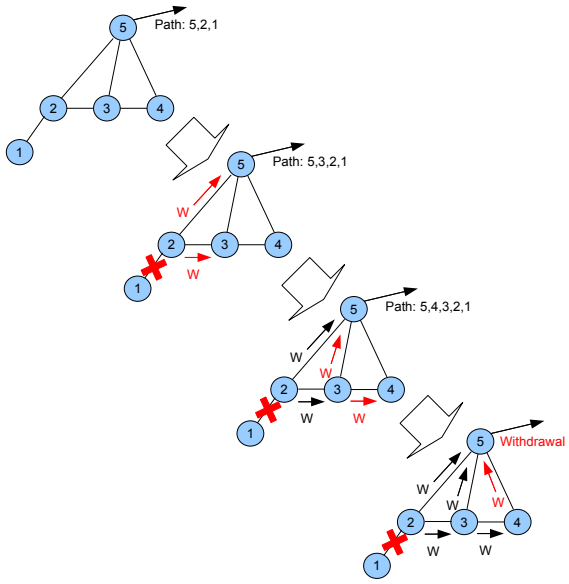

Fig. 1. BGP path hunting problem: The lost reachability of router 1 takes 3 update message before it is properly registered by router 5 (image borrowed from [2])

B. A Solution Proposal

The idea of the algorithm is to simply suppress updates which follow an update of the same prefix within a suppression timer interval if the path the new update is advertising is no shorter than the previous path state. In the case depicted in Figure 1 this could reduce the messages to be sent by router number 5 down to the last withdrawal. Obviously this depends on the settings of the suppression timer, and on the time the various messages from routers 2, 3 and 4 arrive. In the path damping article it is suggested to use a selective suppression timer value which is slightly longer than the commonly used MRAI timer interval.

For the selective update suppression algorithm to function correctly, the per-peer ”burst” behaviour must be replaced by a set of per-prefix timers. When an update reduces the AS Path, or for a withdrawal, the update is immediately propagated and any outstanding per-prefix suppression timer for this prefix is removed. For all other updates, the advertisement of the update is delayed by the suppression timer interval. If there was a timer already active for this prefix, then the per-prefix timer is restarted. Additionally the algorithm would need route flap damping to be turned off, as that algorithm would interfere with the path-exploration damping algorithm, and would be obsoleted anyways through it. Recent studies increasingly discourage using route flap damping at all [11].

IV. Introduction to the Quagga Source Code

In order make the implementation of path-exploration damping more understanding, a short overview of the folder and file structure of Quagga as well as the connections between the most important structs and functions will be given.

Subdirectory	Contents
bgpd	Contains the BGP daemon
doc	Contains documentation files e.g. manpages
isisd	Contains the IS-IS daemon
lib	Contains the core of Quagga: the files and functions common to all daemons
ospf6d	Contains the OSPF daemon for IPv6
ospfd	Contains the OSPF daemon for IPv4
ripd	Contains the RIP daemon
ripngd	Contains the next generation RIP daemon (RIPv2)
tests	Contains some files to test certain Quagga functions
tools	Some additional tools (mostly Perl scripts) for various parts of Quagga
vtysh	The CLI to any Quagga daemon
watchQuagga	Watchdog program to monitor the status of Quagga daemons
zebra	The zebra daemon which controls the kernels forwarding table

TABLE I
Quaggas most important subdirectories

A. Folders and Files

The Quagga project is a typical automake project with its various parts divided into subdirectories which contents are easy to figure out. Table I gives an overview of the most important ones. It can be easily seen, that in order to apply changes to the BGP daemon, it is only necessary to take a closer look to the *bgpd* subdirectory and eventually also to the *lib* subdirectory. It is not necessary to change anything in the *zebra* and *vtysh* subdirectories, as the files containing the connection functions to this important parts of Quagga are actually located in the *bgpd* subdirectories, as the file list in table II shows.

There is actually no need to implement changes to any files in the *lib* directory for path-exploration damping, therefore the listing of the files will be omitted, even if there will be some references to files in that directory in the following sections.

B. Structures

Knowing what all folders and files contain is a start in understanding the Quagga source code, but the most important part is to understand how the various structs and functions play together. It has to be said, that Quagga is a threaded program, and makes extensive use of function pointers, which let you easily loose the overview of the workflow you're following.

As for every BGP implementation, the heart of the BGP speaker is the finite state machine. As this finite state machine keeps the states of every single BGP peering session, it is obvious that it needs to be connected to some structure which reflects such a session. In the case of Quagga this is the *struct peer* defined in the *bgpd.h* header file. The struct where all this information converges, is the *struct bgp* also defined in the same headerfile, which also reflects an instance of the BGP daemon. This bgp struct also contains the pointers to the BGP routing tables, represented by the struct *bgp_table*. There may be present multiple instances of routing tables in one BGP instance, like static routes, aggregated routes and the RIB (Routing Information Base). Every table is constructed as binary tree for quick searching, with structs containing the prefixes and its attributes. The tree nodes are represented by the *struct bgp_node*, which contains the *prefix* struct defined in *lib/prefix.h* — a program wide struct representing a prefix for uniform handling throughout the various protocols Quagga implements.

Additionally the node contains also the incoming and outgoing information for the prefix like the attributes, stored in the *bgp_adj_in* and *bgp_adj_out* structs, which is required by the BGP RFC. The *bgp_adj_out* struct together with the *bgp_node* are the most important structs for the implementation of the path-exploration damping algorithm, and are connected by a one to one relation together with the prefix struct. While the *bgp_table* and *bgp_node* structs are defined in the *bgp_table.h* headerfile, the *bgp_adj_in* and *bgp_adj_out* are defined in *bgp_advertise.h*. The adjacency out struct contains additional pointers to the *bgp_advertise* struct, which adds pointers to structs containing information needed at the moment of sending updates, which do not need to be kept in the adjacency out. The *bgp_advertise* struct is inserted in a double linked list defined in the *bgp_advertise_fifo* struct: the actual output queue. This queue has three instances — update, withdraw and withdraw_low — kept in *struct bgp_synchronize* which is accessed from the peer struct. As BGP is an

Source file	Header file	Description
bgp_advertise.c	bgp_advertise.h	Contains functions to manage advertisement and adjacency information within the BGP daemon. This is one of the files needed to deploy path exploration damping
bgp_aspath.c	bgp_aspath.h	Functions for the most important Autonomous System Path attribute. Since version 0.99.10 of Quagga it supports also 4 Byte AS Numbers
bgp_attr.c	bgp_attr.h	This file contains all the functions related to the attributes of BGP update messages. The attributes are related to the capabilities advertised in a BGP open message. Since version 0.99.10 the AS4Path and AS4Aggregator attributes are also included [12]
bgp_clist.c	bgp_clist.h	This files manage the community and extended community lists [13] [14]
bgp_community.c	bgp_community.h	Functions to handle the community attributes [13]
bgp_damp.c	bgp_damp.h	Handling of Route Flap Damping as described in RFC 2439 [15]
bgp_debug.c	bgp_debug.h	Functions for logging with different debugging levels in the BGP daemon
bgp_dump.c	bgp_dump.h	The binary MRT [16] dump for BGP is created here
bgp_ecommunity.c	bgp_ecommunity.h	BGP extended communities [14]
bgp_filter.c	bgp_filter.h	Route filter functions [17]
bgp_fsm.c	bgp_fsm.h	The BGP finite state machine. This is an other file needed for the implementation of path-exploration damping, as it contains the functions which handle the BGP timers, including the MRAI timer
bgp_main.c		The main function. After running through initialization functions, the main function gets into an infinite loop, which can be stopped only by sending the process a SIG_TERM (15) signal
bgp_mplsvpn.c	bgp_mplsvpn.h	Functions for running BGP over MPLS VPN [18]
bgp_network.c	bgp_network.h	The networking functions. This file manages the TCP connections needed between two BGP speakers
bgp_nexthop.c	bgp_nexthop.h	Functions to check the nexthop reachability if BGP is used in conjunction with the zebra forwarding daemon
bgp_open.c	bgp_open.h	Functions to manage the opening of a BGP connection. In this file it is taken care of all the possible BGP capabilities and their outgoing or incoming advertisement from and to every single peer
bgp_packet.c	bgp_packet.h	The functions in this file take care of the binary BGP packet creation. The FSM mostly calls functions of this file which then call all the other functions which manage BGP attributes, before creating the packet and sending it
bgp_regex.c	bgp_regex.h	Enables the use of regular expressions in the BGP CLI
bgp_route.c	bgp_route.h	Manages the routing tables. As routes/prefixes are mainly handled in this file, it is also needed for the path-exploration damping implementation
bgp_routemap.c		The implementation of Cisco route-maps
bgp_snmp.c	bgp_snmp.h	Controlling the BGP daemon over SNMP
bgp_table.c	bgp_table.h	Contains the BGP routing table structure and provides the functions to perform operations on it
bgp_vty.c	bgp_vty.h	The Command Line Interface client for the BGP daemon. Adds BGP specific commands to the general vtysh
bgp_zebra.c	bgp_zebra.h	The Zebra client for BGP. This client enables to install BGP routes into the kernel through the zebra daemon
bgpd.c	bgpd.h	The main files of the BGP daemon containing the essential *bgp* and *peer* structs

TABLE II
THE FILES OF THE BGPD SUBDIRECTORY

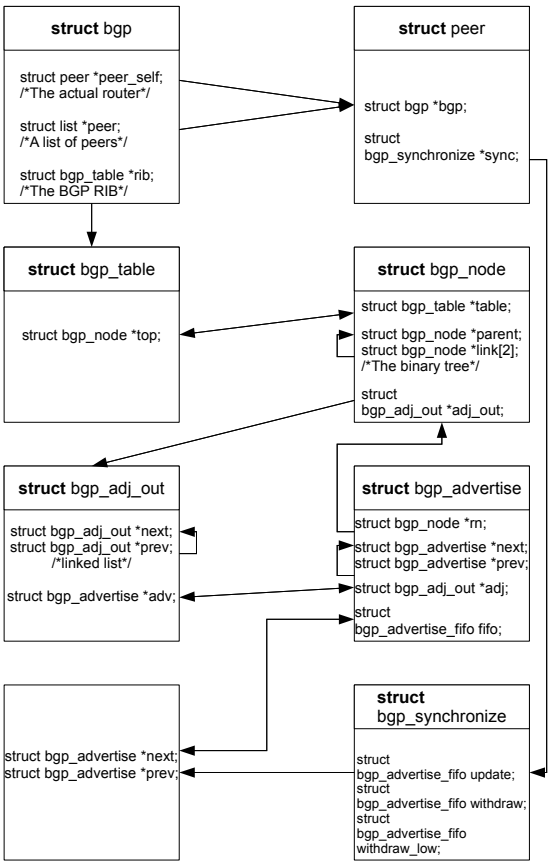

Fig. 2. A simplified diagram of struct interdependencies in bgpd

extensible protocol, there is actually no limit in the amount of attributes an advertisement can carry. In order to minimize memory usage, Quagga uses various hashes and a lot of additional structs to retain the needed information. As this structs have not been changed in the implementation of path-exploration damping, their explanation will be omitted at this point. Figure 2 is a simplified diagram to show the dependencies between the explained structs.

C. Functions

As already explained, Quagga is a heavily threaded program, which allows it to perform extremely well with regard to CPU idle time and responsiveness. The bgpd daemon consists of writing, reading and timing threads, all hold together by the so called thread master, which is executed once for every instance of the daemon. It is practically the main function which ends up in an infinite loop. The thread functions, prototypes and macros are all defined in the *lib/thread.c* and the respective headerfile.

The interesting part is the way BGP advertisements are triggered in the Quagga bgpd. Upon start of the bgpd daemon, after various initializations, like parsing the configuration file, the *main* function ends in the infinite loop, which continuously fetches threads from a list of threads and executes them if necessary. At first, the program tries to create a TCP connection to the various configured peers, using the functions in *bgp_network.c* and on success it installs a reading thread on the connection using the macro *BGP_READ_ON*, which it keeps calling after every read until the connection is closed. It also sends the OPEN messages using a writing thread which is invoked every time something has to be sent by the *BGP_WRITE_ON* macro. It additionally installs various timer threads, one of which triggers the TCP reconnection attempt, in case the first attempt failed. the most important timer thread is the MRAI timer which is set in *bgp_fsm.c*. In bgpd this timer is called route advertisement timer and invokes the *bgp_routeadv_timer* function where the sending of update messages is triggered. To get to this point where updates are sent, first the reading thread needs to be observed. Advertisements are only triggered if there are either changes in the kernel routing table and the zebra daemon is running, or if advertisements arrive from neighboring peers. The second case is the interesting one, as path-hunting only happens for updates coming from other peers. If an advertisement arrives from a remote peer, the reading thread triggers the function *bgp_read* in *bgp_packet.c*, which analyzes the incoming packet and, if it's an update, calls *bgp_update_receive* and a few other functions in *bgp_route.c*. This functions do all the pre-processing for the update: extract the attributes and NLRI (Network Layer Reachability Information) of the update, and apply the configured filters. All updates and withdrawals are stored in a worker queue defined in *lib/workqueue.h*, where a working thread fetches them for further processing as soon as it is available.

The worker thread in bgpd triggers the *bgp_process_main* function which performs the best path selection for updates, applies the changes to the RIB, and passes the updates on to the *bgp_adj_out_set* function in *bgp_advertise.c*, while withdrawals are sent to *bgp_adj_out_unset*. This two functions then put the advertisement into respective update and withdrawal queues. While the withdrawal queue is processed immediately by calling the write thread, the update queue is processed within the *bgp_routeadv_timer*

function.

This shows that Quagga actually doesn't follow properly the MRAI indications in RFC 4271 [8], which suggest to apply the MRAI timer also to withdrawals; but as the path-exploration damping algorithm needs BGP to behave exactly that way for withdrawals, this choice is quite welcome. Figures 3 and 4 show the functions of the advertisement workflow and which structs described in section IV-B are accessed when.

V. Implementation of Path-Exploration-Damping

As already stated, the first changes in the effort of implementing path-exploration damping, has to be made to the MRAI timer implementation. The standard Quagga implementation a per peer "burst" timer, which sends updates queued in the peers advertisement fifo every time the timer thread is fetched calling the *bgp_routeadv_timer* function. This is an easy approach to implement the MRAI timer, but it doesn't quite follow the suggestions in the RFC. With a proper implementation of the MRAI timer as described afterwards, it is not difficult to implement the path-exploration damping algorithm itself.

A. Per Prefix per Peer MRAI Timer

The Solution used to implement the per prefix per peer MRAI timer is the following described in the following steps:

- The first changes are made in function *bgp_adj_out_set* which queues the updates in the output fifo. This changes imply the creation of additional structs and variables in the files *bgp_advertise.c* and *bgp_advertise.h*
- As the information about an earlier update for a certain prefix needs to be retained, the *bgp_adj_out* struct, which represents a prefix which has been sent needs to be altered. The new *mrai_timer* variable simply retains the absolute time — the wall clock time — after which the next possible update for a prefix could be scheduled. The absolute time is provided by an already existing Quagga function.
- A list of update queues per peer is constructed, represented by the struct *bgp_mrai_list*, futher simply called MRAI list. This list reflects the MRAI timer. The maximum amount of queues present in the MRAI list equals the amount of seconds configured in the MRAI timer settings of the BGP configuration. With a 30 second MRAI timer as standard for eBGP sessions this would result in a maximum of 30 queues in the list. Every queue represents one second of the interval. The *mrai_timer* value of a prefix determines in which queue the current update is inserted. This value can never exceed the absolute time value in seconds of the configured MRAI timer added to the current time. The queues are kept sorted in the list containing additionally the timestamp in the variable *update_time*, which indicates at which absolute time the queue should be processed and the contained updates sent. Updates are scheduled only on a per second basis, microseconds are disregarded.
- The second part of the implementation alters the *bgp_routeadv_timer* function in *bgp_fsm.c*.
- The original MRAI timer thread is retained, but it's expiration is changed to every second, and not as originally to the configured MRAI timer (which now is used for the creation of the *bgp_mrai_list*).
- Upon expiration of the timer the first queue of the MRAI list is processed by putting its content into the original bgpd update queue, so that the writing thread function doesn't need to be changed. During this process, the new *mrai_timer* value is set for every *bgp_adj_out* item using the new *set_mrai_timer* function, which follows exactly the MRAI timer calculation described in the RFC. This value determines the queue in which following updates using the same struct (thus carrying the same prefix) are inserted.
- As it is possible, that for certain absolute timevalues the MRAI list does not contain a queue — which means that no updates are scheduled for that second — it is necessary to check the timestamp on the queue, in order not to process it before its expiration time. This also allows the thread to be paused until the absolute timevalue of the next, but only when a queue is present in the list.
- An empty processed queue is deleted from the MRAI list. This way, searching operations on the list are avoided, as the thread always needs to process only the first queue in the list.

B. Implementation of the Damping Algorithm

The damping-algorithm does an ASPath check on the updates to be sent. If the path is longer than a previously sent update, it is queued, otherwise it is sent immediately. Withdrawals are always sent immediately. This way the MRAI timer becomes the update suppression timer, and the actual MRAI timer interval is sset to 0 (send immediately). The following steps describe the implementation

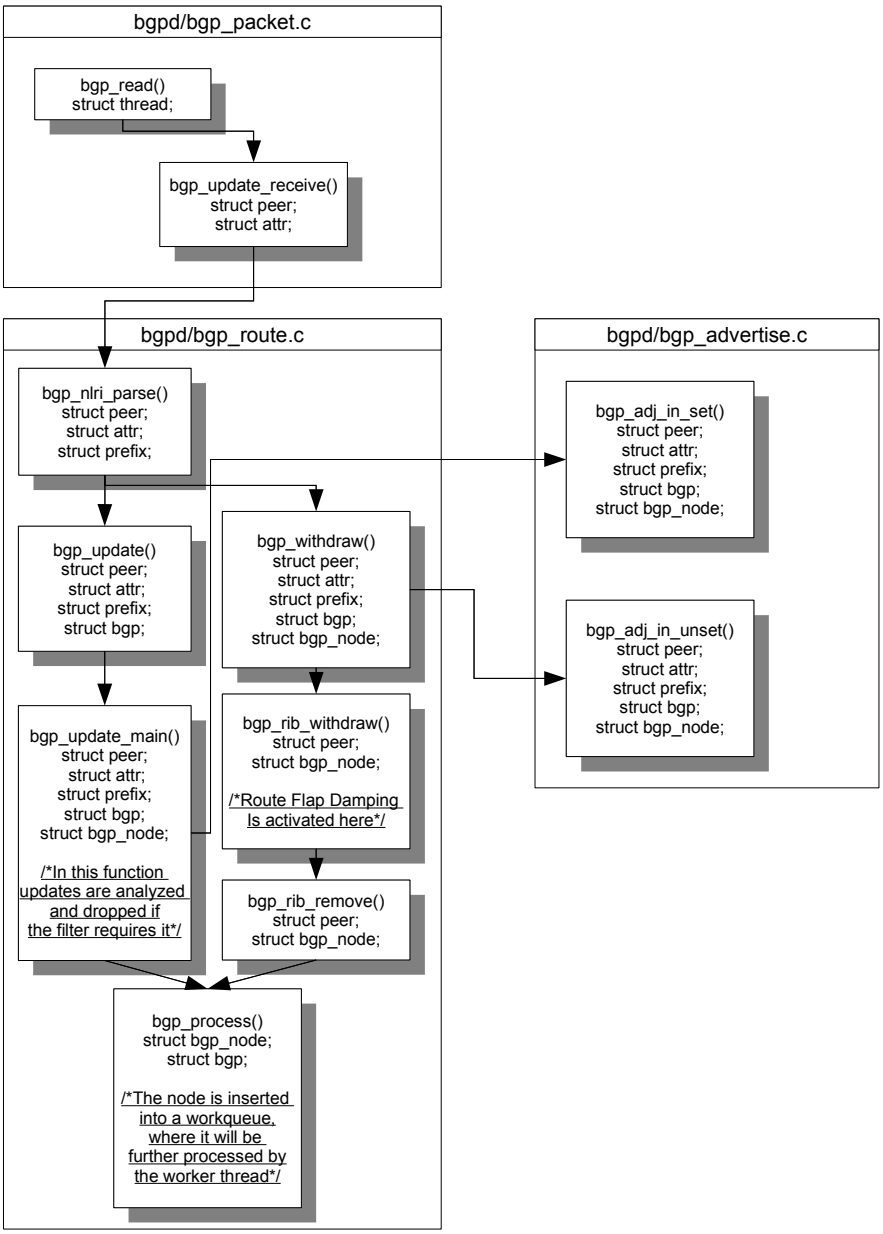

Fig. 3. The workflow of an advertisement reception

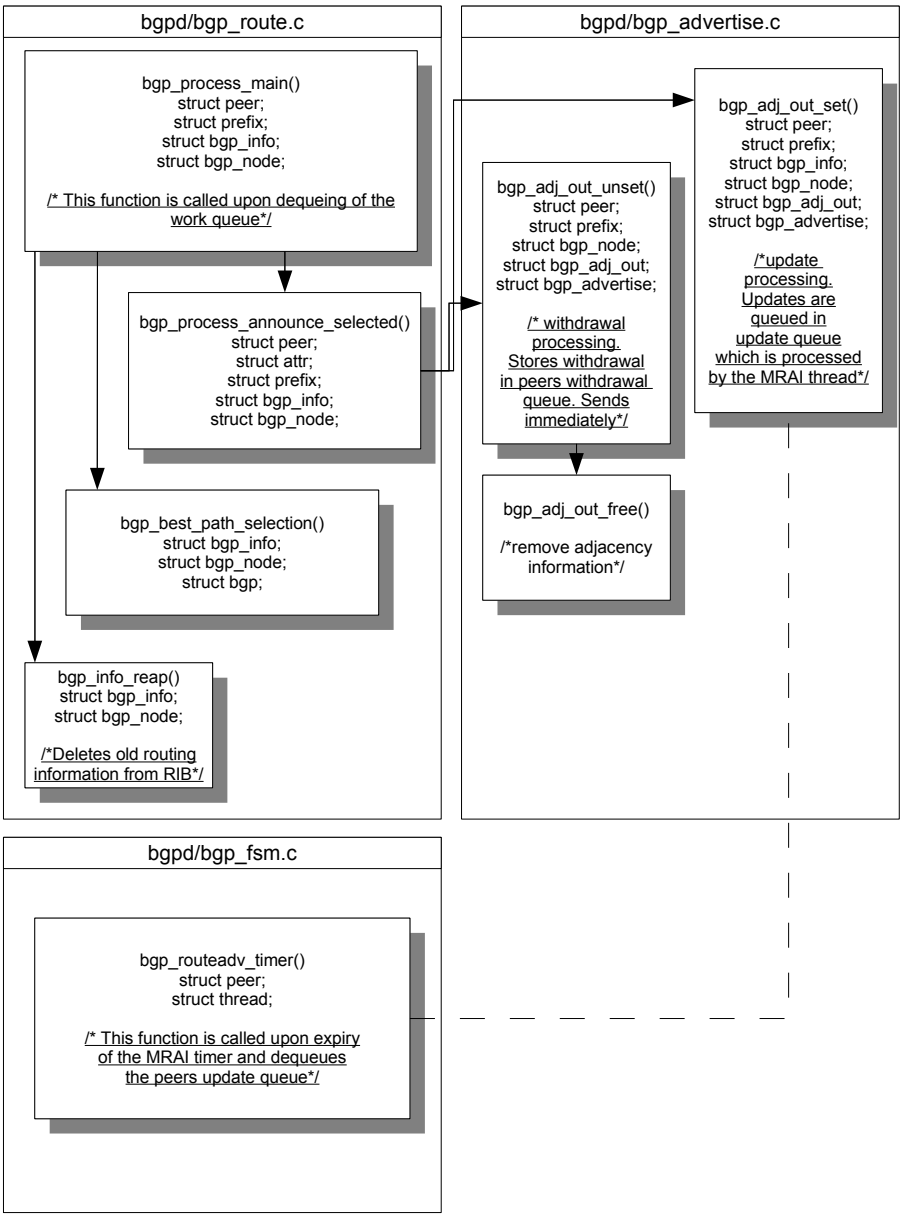

Fig. 4. Further processing of a BGP advertisement in bgpd

in detail:

- The BGP decision process which selects whether to announce the current prefix with it's new updates, or to drop the new information, keeping the RIB entry unchanged and not sending any updates, takes part in the function *bgp_best_selection* in *bgp_route.c*. It returns two instances of the struct *bgp_info*: *old_select* and *new_select* of which the first contains the old or current attribute information, while the second contains the newselected attribute information. The new information must not necessary be the one most recently received. It just contains the information selected in the decision process. If the new information equals the old one, it means that the received update will not be further propagated. It is wrong to say it has been discarded, as it is recorded as backup route in the RIB. If the new information differs from the old one, the incoming update contains a better route, and thus will be propagated further. This is where the first part of the new algorithm takes place.
- In order to keep track of shorter or longer path in the prefix information, a flag *BGP_INFO_SHORTER_PATH* is added to the *bgp_info* struct.
- If a new route is to be advertised, it is first checked if *old_select* contains data. An empty object means, that there was no route for this prefix before, and thus it's a new prefix. A new prefix is handled like a shorter path in the damping algorithm. If there is old information present, then the ASPath of the old and new attribute information is compared. The *BGP_INFO_SHORTER_PATH* flag is set for shorter paths, and unset for longer ones.
- With route information properly set, the new MRAI timer implementation can make use of it. In the function function *bgp_adj_out_set* where the queuing of advertisement takes place, outgoing advertisements can now simply be checked for their *BGP_INFO_SHORTER_PATH* value. If the advertisement carries a shorter path, it is scheduled for sending immediately by changing the MRAI timestamp in the *mrai_timer* value to the current time. If it carries a longer path, it is queued if applicable. It is part of the new MRAI timer implementation, that queuing takes place only for updates for the same prefix within the timer interval, while the shorter/longer path information is not restricted to that.

C. Known Issues

The implementation of the per peer per prefix MRAI timer tries to avoid unnecessary calls of the timer thread by checking for gaps between queues in the MRAI list. Unfortunately in reality it seems to happen quite often that the list ends up completely empty, which results in a continuous per second call of the timer thread.

VI. Testing

The program has been tested quickly on various setups, with single and multiple BGP sessions, to verify the proper implementation of the MRAI timer and path-exploration damping algorithm. Even though there have not been encountered any runtime problems and the testing showed also no errors in the BGP communication, complete certainty can only be achieved with extensive BGP runs, which will also show the effectiveness of the damping algorithm regarding the routing scaling problem and are conducted at the time of writing.

A. CPU and memory usage

The implementation carries with it some barely noticeable overhead in terms of memory and CPU. Additional memory load is caused mainly by the timestamp which is now carried in every update information. The MRAI list and the queues are only additional pointers, so they use very little memory. CPU overhead is only generated by the few comparison operations when queueing updates and by the timer thread expiring every second as described in V-C. There has been made a simple test running an original Quagga and a modified one three times alternating for half an hour, receiving Updates from one BGP speaker, and sending to an other. Every run included the reception and propagation of the whole routing table at the beginning of the session. CPU and memory usage have been measured every second throughout a run. Table III shows the average and peak useage in percents. The tests have been made on a FreeBSD 7.0 system with a 2.4 GHz CPU and 512 MB of RAM.

VII. Conclusions

Thanks to the well structured and cleanly written source code of Quagga, it was possible to implement the path-exploration damping algorithm without much hassle. Many functions and structs already existed, and could be used as needed. The current implementation is not completely optimized, but it is working without producing noticeable overhead. This also shows that the concerns which might have pushed the original authors

	Original	Path-Damping
1st run		
average CPU	3.30794444444	3.3415
max CPU	88.9	88.0
average MEM	11.8920555556	11.8883333333
max MEM	16.0	16.0
2nd run		
average CPU	3.171	3.40688888889
max CPU	88.7	89.6
average MEM	2.98166666667	11.8875
max MEM	15.7	15.9
3rd run		
average CPU	5.93227777778	3.37033333333
max CPU	87.8	88.6
average MEM	10.4531111111	11.9713333333
max MEM	16.1	15.9

TABLE III
AVERAGE AND PEAK CPU AND MEMORY USEAGE IN PERCENTAGE OF AN ORIGINAL AND A MODIFIED QUAGGA INSTANCE

of Quagga towards the current implementation of the MRAI timeras described in section II were unfounded. Even though the Quagga version used in this work is 0.99.10, it will be easy to apply the changes also to the newer 0.99.11 version, as the changes between these two versions do not affect the BGP code used for path damping. The patches to apply path-exploration damping to Quagga 0.99.10 may be downloaded from the "Data gathering and analysis tools" page at [5].

VIII. FUTURE WORK

Future and current work at the time of writing consist in extensive testing of the implementation, with the aim to find out if path-exploration damping has the desired effect of improving BGP. The implementation also needs to be optimized in order to get rid of the known issues. At the time of writing additional changes to Quagga are applied, to create a simulator which can process Quagga logfiles and collect valuable BGP data in test environments in log time.

IX. ACKNOWLEDGMENTS

This work is based on the extensive research work done by Geoff Huston, leading to the development of the path-exploration damping algorithm. The author's extensions to Quagga have been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

REFERENCES

[1] K. Ishiguro, "Quagga Software Routing Suite." [Online]. Available: http://www.quagga.net
[2] G. Huston, "Potaroo.net." [Online]. Available: http://www.potaroo.net
[3] G. Huston, "The BGP Instability Report." [Online]. Available: http://bgp.potaroo.net/index-upd.html
[4] G. Huston, "ISP column: Path Damping." [Online]. Available: http://www.potaroo.net/ispcol/2007-06/dampbgp.html
[5] G. Armitage, G. Huston, and M. Rossi, "Reducing BGP Update Noise." [Online]. Available: http://caia.swin.edu.au/urp/bgp/
[6] Y. Rekhter, T. Li, and S. H. (Editors), "RFC 4271: A Border Gateway Protocol 4 (BGP-4)," RFC 4271 (Draft Standard), 2006, obsoletes RFC 1771. [Online]. Available: http://tools.ietf.org/html/rfc4271
[7] P. Jakma, "Revised Default Values for the BGP 'Minimum Route Advertisement Interval'," draft-jakma-mrai-00.txt (Internet Draft), 2008. [Online]. Available: http://tools.ietf.org/html/draft-jakma-mrai-00.txt
[8] "RFC 4271: 9.2.1.1. Frequency of Route Advertisement." [Online]. Available: http://tools.ietf.org/html/rfc4271#section-9.2.1.1
[9] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed Internet Routing Convergence," in *in Proc. ACM SIGCOMM*, 2000, pp. 175–187.
[10] T. Li and G. Huston, "BGP Stability Improvements," draft-li-bgp-stability-01.txt (Internet Draft), 2007. [Online]. Available: http://tools.ietf.org/html/draft-li-bgp-stability-01.txt
[11] P. Smith and C. Panigl, "RIPE Routing Working Group: Recommendations on Route-flap Damping." [Online]. Available: http://www.ripe.net/docs/ripe-378.html
[12] Q. Vohra and E. Chen, "RFC 4893: BGP Support for Four-octet AS Number Space," RFC 4893 (Proposed Standard), May 2007. [Online]. Available: http://tools.ietf.org/html/rfc4893
[13] R. Chandra, P. Traina, and T. Li, "RFC 1997: BGP Communities Attribute," RFC 1997 (Proposed Standard), 1996. [Online]. Available: http://tools.ietf.org/html/rfc1997
[14] S. Sangli, D. Tappan, and Y. Rekhter, "RFC 4360: BGP Extended Communities Attribute," RFC 4360 (Proposed Standard), 2006. [Online]. Available: http://tools.ietf.org/html/rfc4360
[15] C. Villamizar, R. Chandra, and R. Govindan, "RFC 2439: BGP Route Flap Damping," RFC 2439 (Proposed Standard), 1998. [Online]. Available: http://tools.ietf.org/html/rfc2439
[16] L. Blunk, M. Karir, and C. Labovitz, "MRT routing information export format," draft-ietf-grow-mrt-08.txt (Internet Draft), 2008. [Online]. Available: http://tools.ietf.org/html/draft-ietf-grow-mrt-08.txt
[17] E. Chen and Y. Rekhter, "RFC 5291: Outbound Route Filtering Capability for BGP-4," RFC 5291 (Proposed Standard), 2008. [Online]. Available: http://tools.ietf.org/html/rfc5291
[18] E. Rosen and Y. Rekhter, "RFC 4364: BGP/MPLS IP Virtual Private Networks (VPNs)," RFC 4364 (Proposed Standard), 2006, obsoletes RFC 2547, Updated by RFC 4577 and RFC 4684. [Online]. Available: http://tools.ietf.org/html/rfc4364