

Quagga-Accelerator: An Implementation for Accelerated Processing of Historical BGP Events using Quagga 0.99.13 - version 0.1

Mattia Rossi

Centre for Advanced Internet Architectures, Technical Report 090730C

Swinburne University of Technology

Melbourne, Australia

mrossi@swin.edu.au

Abstract—This technical report describes the *Quagga Accelerator (QA)*, an implementation of a technique for artificially accelerating 'real time' to evaluate historical real-world BGP traffic. The QA is based on the MRT Dump File Manipulation Toolkit (MDFMT) and Quagga version 0.99.13. It replays a live BGP update stream previously recorded in MRT format into Quagga faster than real-time, by accelerating Quagga's internal clock, yet generating outputs which still reflect actual BGP operation. We explain the implementation in detail and show performance results considering the impact of our technique when multiple instances of Quagga run on a single host, and give an outlook of possible implementations of the QA when Quagga instances are distributed across multiple hosts.

I. INTRODUCTION

The idea of a Quagga Accelerator (QA) has been first discussed in [1] (called "accelerated emulator"). The QA is a system consisting of an Update Regenerator (URQA), based on the Update Regenerator of the MRT Dump File Manipulation Toolkit (MDFMT) [2], and a modified Quagga implementation. This system gives researchers the ability to test additions to the BGP protocol implemented in Quagga faster and easier by allowing them to use two important features: replay identical BGP streams into Quagga and accelerate the evaluation of such BGP streams. The possibility of replaying the same stream of BGP updates into Quagga, allows researchers to detect misbehaviour of an implementation easier, as the input is known and the output can be predicted. Anyhow, many implementations need to be verified over a long period of BGP updates and a high amount of possibilities. The possibility to accelerate the processing of BGP updates, yet keeping the output consistent to real-time BGP behaviour, allows researchers to overcome

this hurdle. The QA reaches this two goals by using the URQA to regenerate BGP updates from an MRT dump file, and replaying them into a modified Quagga. The Quagga internal clock is accelerated by the URQA, which provides Quagga with time information read from the MRT dump file. BGP updates are sent to Quagga over a standard TCP connection, but synchronised with this "fake" time information. The output generated by Quagga is also synchronised with the faketime, therefore producing log files which reflect a real-time BGP behavior. The faketime is passed from the URQA to Quagga via shared memory using the mmap system library, which is available in C (programming language used for Quagga) and Python (programming language used for the URQA). The QA uses additional shared memory to keep data processing consistent to real-time behaviour. The following sections will explain the implementation in detail: Section II gives an overview of the two parts of the QA, in Section III we explain how the URQA differs from the UR of the MDFMT and in Section IV we explain the adaptations made to Quagga. We show some performance results in V, discuss problems encountered with the QA in VI and give an outlook of future work, like deploying a QA with multiple Quagga instances distributed on multiple hosts in Section VII. We conclude in Section VIII. Version 0.1 of the QA, as well as the MDFMT are available at [3]. Quagga 0.99.13 can be obtained at [4]. If you are not familiar with the MDFMT or the implementation of Quagga, we suggest the reading of [2] and [5].

II. THE QUAGGA ACCELERATOR IMPLEMENTATION

The Quagga Accelerator (QA) consists of two main parts:

<pre>router bgp <AS number> bgp router-id <router IP> neighbor <remote-ip> remote-as <AS number> neighbor <remote-ip> update-source <router IP> neighbor <remote-ip> filter-list 20 in neighbor <URQA IP address 1> remote-as <URQA AS number 1> neighbor <URQA IP address 1> update-source <router IP> neighbor <URQA IP address 1> filter-list 20 out neighbor <URQA IP address 2> remote-as <URQA AS number 2> neighbor <URQA IP address 2> update-source <router IP> neighbor <URQA IP address 2> filter-list 20 out ip as-path access-list 20 deny .*</pre>	<p>The AS number of the Quagga BGP speaker - not important for the URQA</p> <p>The IP address of the Quagga BGP, to be passed to the URQA as destination IP address</p> <p>The AS number of the remote peer</p> <p>The IP address of the Quagga BGP. This is necessary if multiple IP addresses are configured on a single interface</p> <p>Does not accept updates from the remote peer. This makes sense, as it acts only as intermediate for the URQA</p> <p>The AS number of the first URQA peer</p> <p>The IP address of the Quagga BGP. Same as above</p> <p>Does not send updates to the URQA peer. The UR doesn't accept them. This is important in order not to disrupt functionality</p> <p>The AS number of the second URQA peer</p> <p>The IP address of the Quagga BGP. Same as above</p> <p>Does not send updates to the URQA peer. Same as above</p> <p>Applies the filter-list 20 to all addresses</p>
--	--

TABLE I

AN EXAMPLE CONFIGURATION FOR A QUAGGA BGP RUNNING ON THE SAME MACHINE AS THE URQA AND USING THE LOOP-BACK INTERFACE FOR COMMUNICATION. THE URQA IN THIS CASE REGENERATES TWO BGP SESSIONS.

- An Update Regenerator (URQA) which parses the input file and transmits the information to Quagga together with the timing information.
- Multiple modified Quagga instances, which process the feed and perform all the logging using the timing information received by the URQA.

Based on this idea, it is necessary to refine some details: The Update Regenerator (URQA) described in this technical report is a modification of the Update Regenerator described in [2], experiencing the same restrictions and constraints. This applies also to the instances of Quagga that can peer with the URQA. The URQA can only peer to a single Quagga BGP speaker, but this Quagga instance can then peer to as many instances needed.

As shared memory is used for transmitting the time information between the URQA and Quagga, the additional constraint is, that all Quagga instances need to run on the same machine. In [2], the original UR sends updates depending on the real time of the system, while in this system the timing and thus the send rate are controlled based on load feedback by the Quagga instances participating in the system, communicated via a separate shared memory block (further called *syncmem*). The higher the load on the system, the slower the virtual

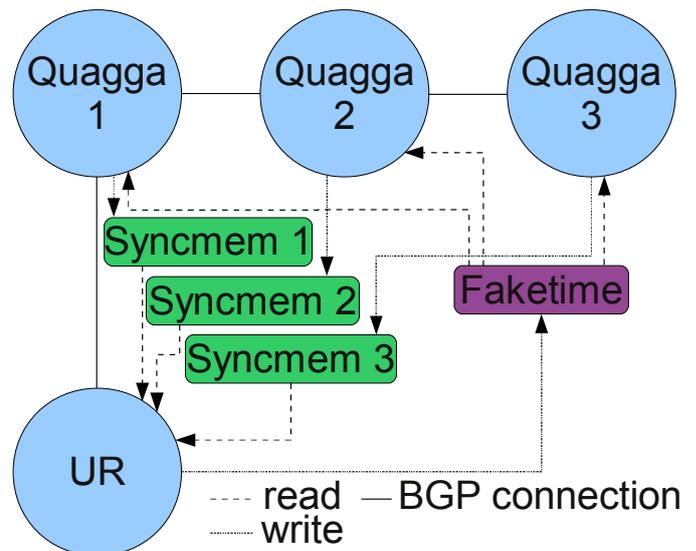


Fig. 1. Communication via shared memory between the URQA and multiple Quagga instances

time advances. An overview of the system is shown in Figure 1.

While the time information and the feedback are exchanged via shared memory, the BGP messages are still

bgpd	
-f <configfile> -l <ip address> -i <pidfile> -t <faketimefile> -s <syncfile>	
-f	The configuration file to use
-l	The IP address to listen on. Automatically disables installing routes into the kernel
-i	The pid file to use. This is necessary, if you want to run multiple Quagga instances on the same machine. One file per Quagga instance is required. The files must be writable by the user that runs Quagga (usually user quagga)
-t	The hook file for mmappping the shared memory to pass the virtual (fake) time information. Needs to be the same for all Quaggas. Is /tmp/faketime by default, which is also hard coded in the URQA. Can be omitted See section III-A
-s	The hook file for mmappping the shared memory to pass the synchronisation information (load feedback). One file per Quagga instance required. See section III-A

TABLE II
AN EXAMPLE COMMAND LINE FOR A SIMULATOR-ENABLED
QUAGGA BGP SPEAKER.

exchanged via real TCP connections. This implies, that both, the URQA and Quagga can bind to IP addresses and reach each other via TCP. This is best achieved setting up all necessary IP addresses on a loop-back interface. The configuration of the Quagga peers is the same as described for the UR in [2] (see Table I) but the Quagga BGP instances need additional options passed to the command line, as shown in Table II. This additional options are introduced in Quagga with the patches that allow the URQA to control the speed of the internal clock. The URQA and the changes to the Quagga source code are explained in detail in the following sections.

III. URQA: THE UPDATE REGENERATOR FOR THE QA

The URQA is basically the Update Regenerator from the MDFMT with added functions to control the internal clock of Quagga and to process MRT dump file entries in *log time*. The first changes regard the command line options. As the QA only works if Quagga and the URQA run on the same machine (compared to the UR which can connect also to remote BGP speakers), the Quagga bgpd is started from the URQA. The commands needed to run the BGP daemon as described in table II can be placed in a text file, separated by newlines. This file needs be passed as argument to the URQA via the new *-q* option flag.

After getting all necessary input (MRT input file, destination IP, AS-IP mapping file, Quagga commands file), the URQA starts to process the MRT file. First it

extracts the timestamp (expressed in seconds since the UNIX Epoch as defined in the POSIX standard [6]) from the first MRT header and sets it as the simulator start time. In order to start the Quagga bgpd instances, it is necessary to subtract some time from that timestamp, spawn the bgp daemons, give them enough time to connect to each other, make the URQA connect to one Quagga instance, and finally arrive to the start time again, which is the point at which updates are sent.

The detailed steps are:

- After extracting the timestamp, 90 seconds are subtracted which are the time within all BGP peering sessions in the system are established.
- The timestamp is written to the faketime hook-file as 4 byte word (system-wide representation of time). This ensures that a file of 4 bytes exists, in order to be able for the URQA and Quagga to map 4 bytes of shared memory. The shared memory is mapped from the file immediately after. See section III-A for the details.
- In the next step the Quagga command file is processed, and the Quagga BGP daemons are started. Each time a Quagga instances is started, the fake time is increased by 1, making Quagga believe one second has already passed by. In fact, to give Quagga the time to create TCP and BGP connections, the URQA sleeps for one real time second, equalling fake and real time. Each Quagga instance is linked to a *syncmem*, a second shared memory block, which uses the file passed to quagga via the *-s* option as hook. This hook file allows unrelated processes to share information about the address of the shared memory block (see section III-A). This shared memory (one for each Quagga instance) is used to communicate the “busyness” of Quagga to the URQA, and to allow it to adapt the speed for advancing the fake clock. The list of shared memory blocks is stored in an array called *synclist*.
- If the *synclist* is empty, the URQA quits, as no Quagga instances have been started.
- 5 seconds after starting the last Quagga instance, the URQA tries to connect to a Quagga instance, opening a TCP connection and sending an OPEN message like the original UR of the MDFMT. Each time one of the fake peers is connected, the fake time is advanced by one second. The URQA sleeps one real time second and then sends a KEEPALIVE, advancing the fake time again of one second, and sleeping one more real time second. This trick has

resulted in a successful connect on every run of the simulator.

- If the start time is still bigger than the current fake time, the fake time is advanced one real time second at a time, using the combination of increasing the fake time and sleeping.

Once the URQA and all Quaggas are connected to each other, the updates can be sent. The updates are read from the MRT dump file, causing a high amount of I/O operations, which limits the speed of the simulator mostly at the beginning of each simulation. Additionally, each Quagga instance needs to process the incoming Updates, and depending on the amount of updates arrived, that might take some time. Without using the *syncmem* for signalling when Quagga is under load, this could result in the fake time being advanced faster than Quagga processes updates which then would result in an inconsistent update behaviour. This load feedback allows the URQA to advance the fake clock slower in such a case, ensuring that there will not be any clock jumps while Quagga is processing updates. The *syncmem* is checked every time the fake clock is advanced. The faketime is increased without updates being sent until it reaches the same value of the timestamp of an update waiting to be sent, in which case, all updates with that timestamp are sent.

The detailed steps are:

- The updates which are recorded in the MRT file with the same timestamp as the faketime provides, are sent as quick as possible, without advancing the fake clock. Once the timestamp extracted from the MRT header of such an update is larger than the current faketime, the faketime is increased, leaving the updates to be sent only after the faketime reaches the value of the recorded timestamp again.
- The faketime is increased only when every connected Quagga instance has signalled that it is not under load anymore. This is achieved by Quagga writing a 0 into its *syncmem*. If the waiting time equals one second of real time, the faketime is advanced, and the reading of the remaining shared memory blocks of the *synclist* is skipped. This ensures that Quagga will not run slower than real time, avoiding unrealistic BGP behavior.
- As the timestamps in the MRT header of updates do not necessarily need to be sequential (there may be “gaps” of a few seconds), the faketime is increased second by second until the MRT timestamp value is reached. This prevents a clock jumping, which

might have unpredictable results in the operation in Quagga.

- If the keepalive timer expires during this process, a KEEPALIVE is sent.
- Once the faketime reaches the same value as the timestamp recorded in the MRT file, all updates with this timestamp are sent.

After all updates are sent, the URQA allows 30 real time seconds for cleanup, before closing the TCP connections and attempting to stop the Quagga BGP daemons. The URQA quits with a simple speedup statistic.

A. Python memory mapping

Mmap allows quick inter-process communication via shared memory, between processes which are not related. In the case of the URQA, mmap is used to write the time recorded in UPDATE messages in the MRT file into a shared memory area, which then is read from this memory block in Quagga on every call of a function which is originally supposed to return the wall clock time. Mmap uses a file on the file system as hook for the shared memory. In the case of the faketime, the URQA creates this file in the */tmp* folder of the root file system with the name *faketime*. This filename is currently hard coded in the URQA, and is the default in Quagga. After creating the file with write permissions — the URQA only writes into the shared memory, while Quagga only reads from it, making complicated semaphores unnecessary — the mmap shared memory is created using the mmap function of the standard Python library with the same name. It is required to pass mmap the size of the memory which must be smaller or equal than the file size, and the access rights. Python’s mmap module. The function allows various flags to define its memory access rights on UNIX platforms. The URQA uses *ACCESS_COPY* which gives the memory write or read access as specified on the underlying file. It also disables synchronisation of the memory with the file, disabling unnecessary I/O operations which would slow down inter-process communication; the reason for choosing mmap over pipes or named pipes. The timestamp is extracted as an integer number from the MRT header and is then written as a 4 byte word (host byte order) into the shared memory to ensure a constant size, and thus save processing time in the read and overwrite processes. This overwrite process is the continuous process in the script of writing into the mapped memory. Multiple Quagga instances need to access this memory at random time, always starting to read from the same starting address. To allow this behaviour without segmentation

faults, the mapped memory is overwritten each time the URQA updates the fake time. Python offers the *seek* function in combination with the *os* library flag *SEEK_SET*, to rewind the start address for writing into the same memory block each time. The *os* library is also a standard Python library.

Testing of the URQA with one instance of Quagga revealed that Quagga sometimes can't keep up with the speed the script processes advertisements, and processes updates in a wrong relative time frame, specially if there are a lot of advertisements within one second. This is most likely to happen at the start of a BGP communication, where the whole routing table is advertised. To avoid the problem a second mmapped shared memory is used (*syncmem*), which uses the file passed to Quagga at the command line through the *-s* option as hook. The URQA parses the command line and extracts the file name, creates the file, maps it to a shared memory and puts the pointer to the shared memory into a list, the *synclist*. Each Quagga instance has it's own *syncmem*, and writes either a 1 or a 0 into it, to signal whether it's ready to process data or not — with 0 meaning that there is no data to process. The URQA continuously reads from the *syncmem*, processing the *synclist* in a single loop. Once the *syncmem* of the last Quagga also signalled that it's ready, the URQA proceeds in increasing the *faketime*, and sending the relative updates. The *syncmem* shared memory is accessed with the *ACCESS_COPY* flag like the *faketime* memory, to allow fast communication.

Additionally to this adjustment process, the URQA keeps track of the real time, and continues to increase the simulated time each real time second, skipping the remaining entries of the *synclist*, in case it is blocked on a certain *syncmem* waiting for a ready signal from the relative Quagga *bgpd*. This ensures more realistic operation, as it can also happen in real BGP operation, that a thread in Quagga runs for more than a single second if under high load.

IV. CHANGES IN THE QUAGGA SOURCE CODE

The URQA is only one part of the QA as already explained. It is also necessary to have a patched Quagga to be able to create a whole system. Similar to the patch created in [5], the changes are made in files of the *lib* and *bgpd* folders. The first file that needs to be looked at is *bgp_main.c* in the *bgpd* folder. In this file the *mmap* shared memory is created and destroyed. The second file, *thread.c*, is located in the *lib* folder, and contains the clock functions and thread scheduler of Quagga. The third file, also in the *lib* folder is *log.c*

which contains the logging facilities. This last file needs to be adapted because of an unclear decision of the Quagga developers, to use the system wide *gettimeofday* function to retrieve wall clock time instead of calling the Quagga function. Changes have to be applied also to the *bgp_dump.c* file in the *bgpd* folder. This file is responsible for the timestamp in BGP MRT dumps, and gets its time information through the simple *time* system call, which bypasses the Quagga wide *quagga_gettime* function call that contains the simulated time. In order to have the timestamps in simulated time (necessary for the proper evaluation of BGP traffic), the *time* system call has to be replaced with the quagga simulated time call.

A. Memory mapping in C and the simulated clock in Quagga

Beginning with the changes made in the *bgp_main.c* file, it is also necessary to explain the C *mmap* function. Similar to the Python function, the memory area is mapped through a file. As this file needs to exist already in order to be able to read from it, the URQA creates the file and the shared memory for the URQA first, and executes the Quagga instances afterwards, which have access to the file of the correct size and properties. The *mmap* call in C is slightly simpler than in Python. The access information is retrieved from the file properties, so the call needs only the file as argument. The *mmap* function is called in *bgp_main.c*, but defined in *lib/thread.c*, storing the memory map itself is in a global variable within that thread file. The thread file contains a variety of functions to keep track of the time, and provide all Quagga processes with time information through the *quagga_gettime* function. These original functions rely on the *gettimeofday* system call to retrieve time information, which is replaced with a call to a function that reads the simulated time from the memory map.

The time returned by the memory map read process does not provide microsecond information although. Therefore it is necessary to rely on the microseconds provided by the *gettimeofday* call. Quagga also includes a lot of time adjustment mechanisms to ensure Quagga's functionality even if the system clock should suddenly provide wrong information and to avoid clock jumps. This results in Quagga providing two global variables which contain two different but related timestamps. The *recent_time* variable contains the actual wall clock time, while the *relative_time* contains the time since the start of Quagga. The relative time is computed from the recent or wall clock time, and is kept up to date by

keeping track of the microseconds passed by, in order to avoid the described wrong time information upon system clock changes. This is important as all timer threads in Quagga are based on the information provided by *relative_time*. The updating of the relative time using microseconds is a problem for the simulator on the other hand. This would let the time increase even if the URQA does not increase it. The behaviour has been changed to ignore the microseconds overflow, resulting in the possibility of having multiple microsecond overflows within a simulated second of time. As the timer threads do not rely on microseconds anyways, this solution does not affect the functionality of the simulator.

Quagga uses a variety of threads to process BGP updates. These threads are all placed into a main thread scheduler queue and executed in the *bgp_main.c* infinite loop, at the end of the main function. Thread types range from event threads (to process signals, BGP notifications and error events), read/write threads, timer threads (e.g. for KEEPALIVE messages and MRAI timers), to background threads including work queue threads (update the RIB, RIB consistency checks and other operations). On arrival of a high amount of UPDATE messages, the threads can become quite busy, and it might take some time for them to finish. It is necessary that during this busy operation, the simulated time does not increase, as it could result in threads taking several simulated seconds to finish, which is inconsistent to an original operation. In order to overcome this problem, Quagga uses the *syncmem*, the shared memory block mapped from the file passed through the *-s* option to the program, to signal the URQA that it is busy. The URQA stops increasing the simulated time, until every Quagga of the simulator signals that it is not busy anymore. Quagga checks the thread scheduler queue, in order to find out whether it is busy or not. This is done by looking for read, write or work queue threads, which are the threads which put a high load on the system.

The function relative to that operation is the function *thread_fetch*, which empties, but also fills the main thread queue. The thread queue is emptied, as long as there are threads in the queue, and is filled, if it is empty. The queuing process depends on the type of thread that is available. Each thread type has its own queue, event threads are in the event queue, timer threads in the timer queue, read threads in the read thread queue and so forth. Events are the first processed threads. The event queue is emptied and the event threads are placed into the main thread queue. The main thread queue is then processed without taking into consideration the remain-

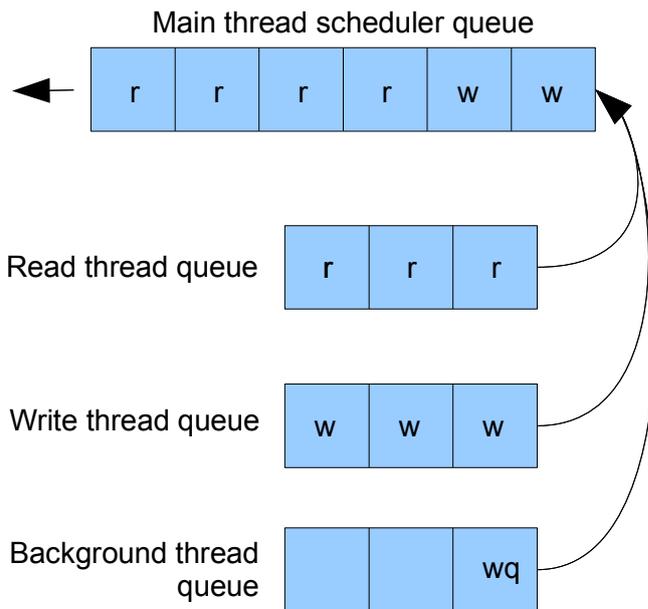


Fig. 2. Read thread, write thread and background thread queues are emptied and the content put into the main thread scheduler queue. The background thread queue contains also the work queue thread (wq).

ing threads. In case no event thread is scheduled, the remaining thread queues are emptied and the respective threads are placed in the main thread queue. Threads are processed in the following order: timer threads first, then read/write threads then background threads. The process is illustrated in Figure 2. Here the amounts of read and write threads that are passed from the respective queues to the main thread scheduler queue are noted in simple variables. The values of the variables are decreased each time such a thread is processed from the main thread queue.

To signal that it is busy, Quagga writes a 1 into the *syncmem* as soon as a read, write, or work queue thread are processed from the main thread queue. It has been detected that there is always only one relevant work queue thread that is executed for a whole read-write cycle of BGP updates, while there can be many read and write threads. To detect whether Quagga is not busy anymore, it is necessary that the current processed thread is not a work queue thread, and that the values of the variables keeping track of the remaining read or write threads to process, are below a certain threshold (5 for read threads, 0 for write threads). If everything matches, Quagga writes a 0 into the *syncmem* and the URQA will

either look for a ready signal at the next memory block shared with an other Quagga instance, or increase the faketime if this was the last Quagga instance to wait for. Selecting to use a different shared memory for each Quagga instance, avoids the necessity for system wide semaphores. The threshold of 5 for the read threads has been chosen to avoid Quagga processing the latest read thread which would eventually listen until new messages arrive, and Quagga would not be able to signal that it is ready. This situation would unnecessary slow down the QA, as it would wait until a real-time second passes by, before letting the URQA continue to increase the faketime. This situation can still happen if a work queue thread is processed without a read or write thread in between. As such an event is rather uncommon, the simulator slows done only slightly in overall time. The effects will be discussed in the following section.

V. EXPERIMENTAL TEST RESULTS

The QA needs to run multiple Quagga instances on a single system, therefore the resource usage is quite high. The QA has been tested on a dual core CPU with 2.4 GHz each core, 4 GB of RAM, and 140 GB of hard disk space. While the amount of memory available on the host is enough to accommodate many BGP instances using a full Internet-size BGP routing table, the CPU is only capable of handling a limited amount of BGP instances, even if they are equally spread over multiple cores. The amount of CPU time needed depends on the amount of peering connections that are created between the BGP instances. It also depends on the amount of BGP peering sessions recreated by the URQA and the amount of BGP update messages that travel along the system. The QA has been used reflecting a variety of routing topologies, with BGP speakers peering in many ways. The URQA has been used to regenerate up to 5 peering sessions (see Figure 3) and the input data has been collected by a Quagga BGP speaker connected to APNIC [7], and contains an Internet-size routing table with approximately 300K entries. This is translated into 300K updates at the beginning of each BGP session, causing each Quagga to generate really high CPU load at the beginning of a simulation. Additionally resource usage depends on the type of logging setup in Quagga. Dumping BGP updates in MRT binary format uses less CPU and disk space than logging updates in text format. The following subsections show the speedups gained with a test setup and possible failure modes of the simulator, related to CPU resource shortage.

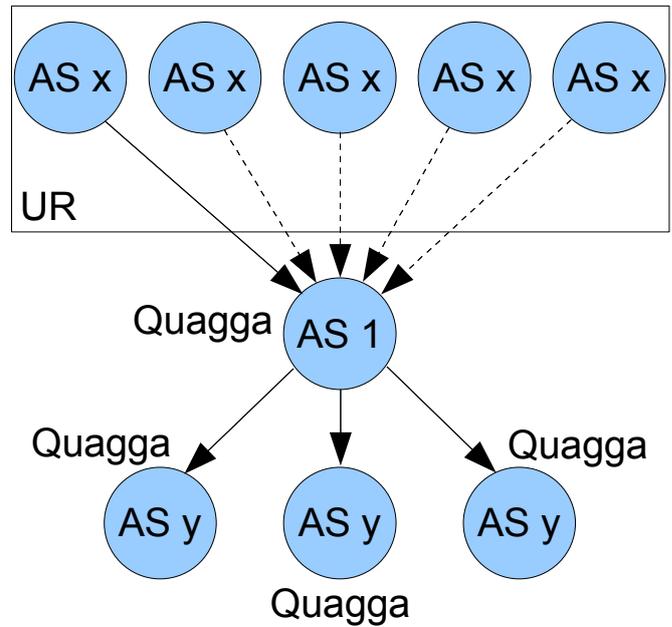


Fig. 3. The URQA has been used to regenerate up to 5 BGP peering sessions

A. Example Speedup and Resource Usage

Figure 4 shows the speedup of the QA over a simple BGP topology where the URQA regenerates 2 BGP peering sessions, and where up to 8 Quagga BGP speakers are connected with a single peering session to a single neighbor (see Figure 5). BGP messages are only propagated downstream and are logged at each Quagga instance, once in binary MRT format and once in text log format.

The speedup decreases quite rapidly and linear the more Quagga instances are running. The maximum speedup reached is of 18, using a single Quagga instance and MRT dumps. Running 2 Quagga instances it is still almost 17 times faster than real-time. The worst speedup reached is of 7.8 running 8 Quaggas and text logging. It can be seen, that using text log files slows down the QA compared to using MRT dumps, but between the two types of logging, the difference is more remarkable in the size of the generated log files. An input MRT file of 50 MiB generates an MRT file of the same size at the first Quagga instance – Quagga just re-dumps all the received updates, in practice generating the input file again — and about 12 MiB at the second instance. A text log file generated at the first Quagga instance with full debugging on instead can reach a size of approximately

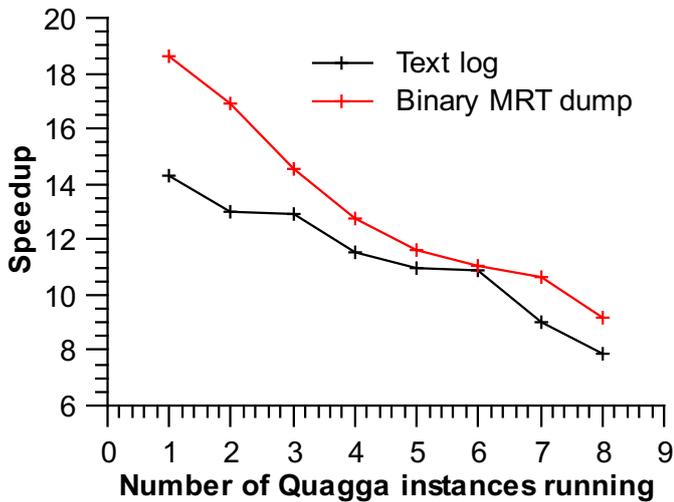


Fig. 4. Speedup achieved with up to 8 Quagga instances running. If used with MRT dump files, the speedup is higher than if used with text logging. The speedup decreases linearly with each additional Quagga instance.

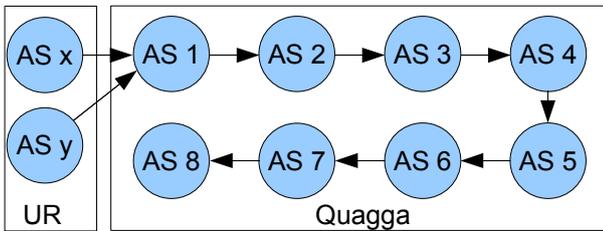


Fig. 5. A simple topology used to observe the speedup of the QA

2 GiB and 1 GiB at the second Quagga instance. With 8 Quagga instances the total disk usage using text logging reaches around 10 GiB for the text logs compared to approximately 150 MiB using MRT dumps.

VI. KNOWN ISSUES

The QA has been proved to work reliably with a small number of Quagga instances running on the same machine. A larger number of Quaggas, or for instance a larger number of inter-Quagga peering sessions, has been observed to put an excessive load on the CPU, causing certain Quaggas to drop a BGP peering session, and thus the whole simulator to fail. This happens mostly, because the Finite State Machine of a Quagga BGP instance can not advance to a certain state, as the thread responsible for the action could simply not be scheduled as needed due to CPU resource shortage, while in a neighbouring Quagga instance everything proceeds properly. In that case the Quagga with the FSM in a right state might send a message to the Quagga with the FSM in an im-

proper state causing the BGP connection to be dropped. Another possibility is also that Quaggas internal resource management is detecting a resource shortage and simply stops the execution of Quagga printing a resource usage report in the logfile. This failure modes can be observed also with a number of Quagga instances running on the same host at real-time. At the time of writing the Quagga thread system is being reworked, which might result in different behavior of the simulator with future versions of Quagga. The URQA is known to use quite a high amount of resources, specially if regenerating a high amount of peering sessions. Depending on the amount of peering sessions regenerated by the URQA, the amount of updates contained in the input file, the size of the routing table to be handled and the topology simulated, resource availability for the simulator may vary, and so will the number of Quagga instances possible to run without failure.

VII. FUTURE WORK

The QA is experimental work for a specific project. It uses a lot of resources, and can be optimised in many ways. The URQA is Python based and quite slow, switching to the C based libbgpdump [8] library could improve speed and resource usage, but makes the creation of the OPEN message more complex, as libbgpdump is not designed for creating MRT dumps, but for reading them. The simulator currently only works with IPv4; IPv6 needs to be enabled in the Python libraries used by the URQA. The URQA could also be extended in order to be able to peer with multiple Quagga instances and eventually use multiple MRT files as input. Efforts to create a distributed simulator which can make use of multiple hosts on a LAN have been made, with partially very good results. The preliminary results of this ideas are discussed in the following subsection and should be intended as suggestions for future work.

A. Simulation on Distributed Systems

It has been tested to share the fake time within the QA via file, without mapping it into a shared memory, and sharing the file through a NFS file system [1]. While the system basically proved to work, the speedup has been quite low, and the BGP communication logged was inconsistent with the original data, due to caching used by the network file system, refraining the internal clock of remote Quaggas to be advanced in time. The later shortcoming has been tried to avoid by using plain TCP communication between the main URQA and remote “UR-servers” on different machines which synchronise

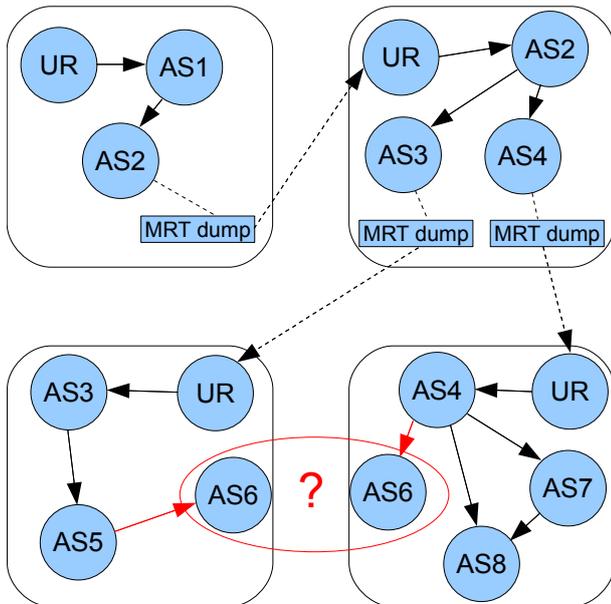


Fig. 6. Distributing the simulation of a topology over multiple hosts. Each host (squares) runs an instance of a URQA, which uses the piped MRT dump generated by a Quagga collector as input. As the red circle shows, it depends on the main topology, whether it's possible to distribute the simulation over multiple hosts.

the fake time and the load feedback of the local Quagga instances, avoiding the use of files and NFS. As for the NFS system, the Quagga instances were peering via real network to each other, exchanging BGP messages at log time. Again, the communication of the fake time worked well, and compared to the NFS system it was processed in time on the remote Quaggas, but the load feedback from Quagga to a “UR-server” and from there via network to the main URQA has proved very unstable and inconsistent making the whole QA system inconsistent. This idea could be improved with additional research and testing. An other idea was to run independent QA's on several machines, with each QA processing a part of the main BGP topology. Each distributed QA deploys one Quagga instance as collector and pipes the MRT dump of the collector to the URQA of a remote QA, which then regenerates the BGP session for the QA on the remote machine as shown in Figure 6. This setup does not need any additional changes to the URQA or Quagga, but can be achieved quite easily with common Unix programs and commands and some configuration and preparation¹.

Even though the mechanism to distribute the QA is quite simple, there are some constraints on how a

<code>mkfifo -m 0777</code>	<code><inputfile></code>	create a fifo to be read by the URQA with permissions 0777
<code>ssh <host> 'tail -f -c +0</code>	<code><dumpfile> tee -a' ></code>	Pipes the copy of a MRT dumpfile through an ssh pipe into the previously created fifo
<code>< start URQA ></code>		Start the QA system on the remote machine. The URQA will start processing updates as soon as data comes through the pipe. Depending on the configuration of ssh, it might be necessary to authenticate after issuing the ssh command and starting the URQA.

TABLE III

ON A REMOTE HOST, THE URQA READS FROM A FIFO, NOT A FILE. THE OUTPUT OF A QUAGGA BGP SPEAKER IS DUPLICATED AND REDIRECTED THROUGH AN SSH PIPE INTO THIS FIFO. THE URQA HANDLES THE INPUT LIKE USUAL.

topology has to be split. As the URQA currently is not able to use multiple MRT files as input, and it's not possible to run two instances of an URQA to feed the same Quagga (see AS6 in Figure 6), it is necessary to plan the split carefully, depending on the available hosts. Some topologies like a full mesh for example, are not possible to distribute at all. Graph theory has to be used to determine the possibility of splitting a topology into sub-topologies as needed. The simple commands to pass an MRT dump over the network to be used as input for the URQA is shown in Table III. These commands have to be executed on the remote host.

Using *SSH* the output of a Quagga MRT dump is continuously printed to stdout using the *tail -f* command, duplicated by the *tee* command² and piped over the network into a previously created named pipe (*fifo*). This *fifo* can be opened by the URQA like a file, and the MRT entries processed as usual. The content of the *fifo* is constantly updated as soon as the collecting Quagga dumps some MRT information, which makes the remote URQA not only adapt the speedup to the Quagga load feedback, but also to the speed messages arrive through the “MRT dump pipe”. The remote system needs to be started and stopped manually on each remote machine, as the input MRT dump (the *fifo*) only closes after tearing down the ssh session (stopping the *tail* and *tee* command), allowing the URQA to finish its job. As this is very early work, no speedup measurements for the distributed system simulators have been made yet. The

¹It has been successfully tested on a *FreeBSD 8 beta1* system

²It seems that Quagga stops running if the MRT dump is piped without duplicating

ideas of distributing the simulation over multiple hosts can be refined and expanded in many ways. Using the technique last presented for example, it might be possible to create a BGP simulator which could be distributed over a Grid. Again that depends on the topology and whether tasks of evaluating a part of the topology could be executed in parallel or not, and the speedup achieved could be questionable.

VIII. CONCLUSION

We presented a working implementation of the Quagga Accelerator which replays a historical data stream accelerated over a system of modified Quagga instances, still generating output correctly scaled in time. We have shown in detail, how the time information is extracted from MRT dump files by the Update Regenerator of the Quagga Accelerator (URQA) and passed through shared memory to a modified Quagga instances. We have also shown, how Quagga has been modified to think, the fake time information is the real time, to make it run faster, and how the load of Quagga is measured to keep the operation of the QA consistent. The performance analysis has shown that we can speedup the processing of Quagga updates using this implementation by 18 times. We also discuss that the QA uses a high amount of CPU which could infer the operation. To overcome this shortcoming, we also tested possible implementations of a distributed QA which deploys multiple Quaggas on multiple machines, which resulted in a partial success. Based on this facts we are able to confirm the benefits the QA can provide to BGP research and new Quagga implementations: it makes use of real historical BGP data, allowing to repeat tests with the same input keeping output coherent, and the system can process data faster than real-time, showing that its speedup mainly depends on the size of BGP topologies simulated. The QA is mostly useful to replicate small topologies, which need to evaluate a high amount of BGP data. The working implementation is available in version 0.1 for download at [3].

IX. ACKNOWLEDGEMENTS

The development of the Quagga Accelerator has been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley. The author would like to thank Geoff Huston for the initial idea for a QA and Grenville Armitage for the ideas on how to refine the implementation and stabilise the QA through a *syncmem*, as well as the idea to distribute the QA over multiple hosts.

REFERENCES

- [1] M. Rossi and G. Armitage, "Accelerated Processing of Historical BGP Events for Testing New BGP Heuristics," Centre for Advanced Internet Architectures (CAIA) - Swinburne University of Technology, Tech. Rep., March 2009. [Online]. Available: <http://caia.swin.edu.au/reports/090321A/CAIA-TR-090321A.pdf>
- [2] M. Rossi, "MRT dump file manipulation toolkit (MDFMT) - version 0.2," Centre for Advanced Internet Architectures (CAIA) - Swinburne University of Technology, Tech. Rep., July 2009. [Online]. Available: <http://caia.swin.edu.au/reports/090730B/CAIA-TR-090730B.pdf>
- [3] G. Armitage, G. Huston, and M. Rossi, "Reducing BGP Update Noise: Data gathering and analysis tools." [Online]. Available: <http://caia.swin.edu.au/urp/bgp/tools.html>
- [4] K. Ishiguro, "Quagga Software Routing Suite." [Online]. Available: <http://www.quagga.net>
- [5] M. Rossi, "Implementing path-exploration damping in the Quagga Software Routing Suite Version 0.99.13 - patch set version 0.3," Centre for Advanced Internet Architectures (CAIA) - Swinburne University of Technology, Tech. Rep., July 2009. [Online]. Available: <http://caia.swin.edu.au/reports/090730A/CAIA-TR-090730A.pdf>
- [6] IEEE, "Standard for information technology - portable operating system interface (posix). base definitions," *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Base Definitions*, pp. -, 2004.
- [7] "Asia pacific network information centre (apnic)." [Online]. Available: <http://www.apnic.net>
- [8] "Libbgpdump, C library for BGP MRT dump processing." [Online]. Available: <http://www.ris.ripe.net/source/libbgpdump-1.4.99.8.tar.gz>