Challenges in Testing: How OpenSourceRouting tests Quagga

Martin Winter
Network Device Education Foundation (NetDEF) / OpenSourceRouting
www.netdef.org / www.opensourcerouting.org
San Jose, CA, USA
mwinter@netdef.org

Abstract

This paper discusses the details how OpenSourceRouting tests Quagga and the challenges on testing a multi-platform application, supporting many different OS variations, CPU architectures and a community of various volunteers and commercial users. The goal of the talk is to give some inspiration to other projects on how to approach this and start a discussion.

Keywords

Testing, Routing Protocols, BGP, OSPF, ISIS, Quagga, Continuous Integration (CI), Open Source, Multi-Platform testing

Introduction

OpenSourceRouting is a project run by NetDEF (Network Device Education Foundation), a US based, non-profit 501c3 corporation. Our main activity under OpenSourceRouting is our work supporting Quagga. When we started this work, we realized that one of the missing tasks in the Quagga Community was thorough testing and decided to make this one of our key area in Quagga. It soon became clear, that there weren't good examples to copy from other communities - we had to mostly build from scratch based on our knowhow testing at router vendors and adapt to the Open Source world. Key focus was to be more open, simpler to understand and much more automated as we can't afford the headcount of a large router vendor.

In this paper, we would like to explain how OpenSourceRouting tests the Quagga project and discuss some of the challenges.

This should be mainly seen as inspiration for other projects and as a basis for discussion. While the paper talks about testing Quagga, the concepts, ideas and challenges are probably similar for many other projects.

The main challenges/requirements for testing Quagga are:

- *Community driven project.* There is no single authority to make decisions and there is no mandatory training for community members. The process and particularly the results need to be easy to understand.
- *Commercial tools for testing (mixed with Open Source).* Dedicated testing hardware can't be moved to the cloud and some commercial software may not allow it in its license.
- *Application which requires network topology for testing.* Many tests require multiple systems to be connected together to run the tests
- *Multi-OS / Multi-Platform Application.* The Application is designed to run on various Operating Systems and CPU architectures

The goal of this paper is to give some inspiration to other projects on how to approach this this complexity.

What is "Quagga"

The focus on this paper is not on Quagga, but rather on the testing. However, it helps to have a basic background on Quagga to understand the challenges and choices highlighted.

Quagga[7] is an Open Source (GPLv2 or later) implementation of a routing stack. It currently implements RIP, RIPng, OSPFv2, OSPFv3, ISIS and BGP routing protocols. It is not a fully functional router as it only handles the routes and not the forwarding. It is usually combined with a forwarding plane such as a Linux to make it into a simple router or it can be connected to external hardware to build a distributed router (i.e. a SDN deployment).

Quagga was originally forked from Zebra. It is a community project as there is no single company behind it. It is run and maintained by its community.

Current state of the Quagga community

The process is adapted to the way the Quagga Community currently works. Here is quick summary of the key elements. This isn't meant as critique of Quagga or as suggestion for any other project. It mainly states the facts and challenges. We didn't want to force the community to change the workflow - at least not at the beginning until we could show the value to the community. Similar limitations or choices might be found on many other (mostly long running) open source projects.

No Owner

There is no single owner or major sponsoring (and controlling) corporation behind Quagga. Decisions are either made by conscious on a mailing list or are frequently deferred until a conscious decision can be made later. There is a small group of maintainers, but beside the agreement that their job

is to push commits into the git (They are the ones with write access), their exact role is fluid.

Simple Git Model

Quagga is contained in a public git on Savannah[2], a smaller public Git hosting. The git model is mostly a "Centralized Workflow". There is a master branch and a frequent "proposed" testing branch of commits before they fold into master every few months. The master branch is expected to be stable and working code. Releases are set as tags directly on the master branch and not on release branches off master. Write access to any branches is limited to the same group of maintainers.

Email based patch submissions

Bug fixes and submissions of new code is done mainly by emailing the code to the Quagga-Dev[1] mailing list. Code review and discussion is done in simple email threads on the mailing list. Patches mailed to the list are automatically picked up by the patchwork tool [5] and published in the Quagga Patchwork Database [2].

Choosing a CI System

Continuous Integration (CI) is currently a hot topic and many choices exist. Some of the more popular ones are Jenkins[9], Travis CI[10] and Atlassian Bamboo[1]. Some CI Systems are hosted in the cloud only, some others allow running it on your own host. Another key difference is the supported Operating Systems and integration into other tools (ie Git host for triggering runs, publishing results, bug database connection etc) and supported languages.

It might be news to developers of CI systems, but the world had not (yet) evolved just to Linux with all applications written in Java and executed as web applications.

OpenSourceRouting decided on Atlassian Bamboo. It is less feature rich than some of the other CI systems, but it worked flawless and reliable in our evaluation and in production so far.

CI for projects not (just) using Linux

Most CI systems are designed for Linux and require some Java agent to be installed to function. Unfortunately, this makes it difficult to test against other systems, such as NetBSD which may not have any of the supported Java versions available (or where some functions are not implemented in Java). For our building stage with the required supported operating systems, we could not find a single CI system with a Java client which worked on all of them. Main issues were the older distributions which we support and the various *BSD based ones. In our system, we decided to natively build the code on the various supported Operating systems and specific distributions. This includes older distributions like CentOS 6 or FreeBSD 8. Some of our commercial testers are based on even older CentOS 5. None of the tested CI systems had Java agents which were supported or would reliable work on these older systems.

At the end, we decided to skip the Java agent for the multi-platform parts of our test and use a local Java agent on the CI system itself which then executed and controlled the target platforms with simple shell commands across SSH as shown in Figure 1

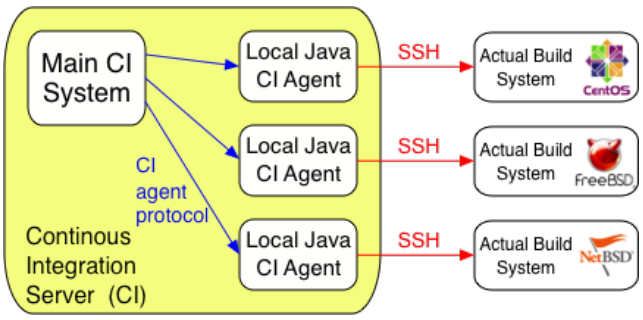

Figure 1: Using 2-stage Agent with execution using shell commands across SSH instead of local Java Agent

Hosted vs Local

Fully hosted would be impossible for us as we need to have at least some agents executed on dedicated network testing hardware. At the same time, most free online CI systems limit the number of parallel runs, the number of parallel agents (sub-tasks of the same test run) or number of executions per day. Our tests take a long time to run and we depend on a large number of parallel execution nodes. For this reason, we ended up using a CI on our own server hardware.

Other considerations

Stability ruled out Jenkins in our tests. While each bug got promptly fixed in the next version, at the same time new bugs (blockers for our application) got introduced as well. After a few months, we were forced to give up, as this made it too time-intensive for us to use.

Licensing cost was not an issue on any of the evaluated systems (at least not the self-hosted ones) as they were all free in general or free for Open Source projects.

Overview of a Quagga CI Run

Figure 2 shows an overview of the CI system. The major challenge is the time and the number of parallel execution nodes required for the various stages. We try to run everything in virtual machines as this allows a much higher scale.

As seen in Figure 2, a full run of the implemented CI system takes over 2 days and nearly 50 virtual machines running in Parallel. Even considering that many of these tests are only really executed on Ubuntu with Intel architecture, this makes it impossible to run on every commit. We are currently working on adding FreeBSD to the full protocol tests as well, but this will already push us towards 100 VMs in parallel to execute a single pass.

The main goal is to reduce the time to provide a limited "pass" reply within a reasonable time for the contributors.

[1] https://lists.quagga.net/mailman/listinfo/quagga-dev
[2] Quagga Patchwork DB: http://patchwork.quagga.net/

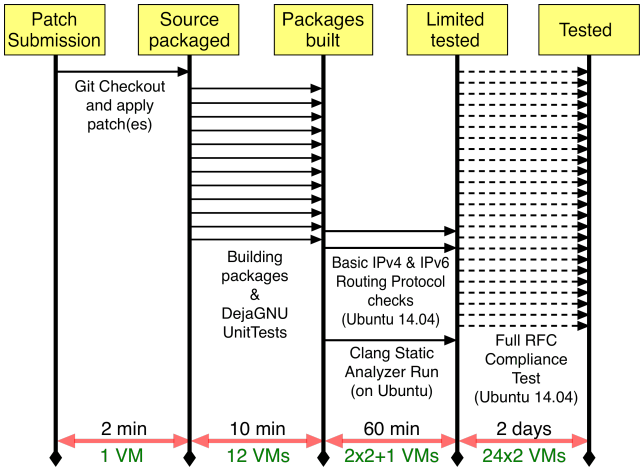

Figure 2: Overview of the automated CI Run

Our guess is that feedback should be less than 2 hrs after submission. Based on this, we run the fully automated pass only until the "Limited tested" status. The full RFC compliance check is only manually triggered on specific releases at this time to keep the test system load acceptable.

At the end of the "Limited Tested (or at any earlier stage when the tests are aborted based on a failure), the CI system will collect the details and send a simple email back to the submitter and the list as a response to the patch. We don't expect submitters to know or monitor the CI system, so an easy to understand email back is usually the best choice. In case of failing notification from the CI, the contributor would then send in a new, revised submission.

The most common failure is currently patches which don't apply (i.e. patches based on another git commit instead of the lastest) and failing to compile on some Operating Systems.

Git Checkout / apply patches

The process starts with a patch submission to the mailing list. Patchwork[5] will recognize the email as a patch, adds it to the patchwork database and assigns a unique patch number. Our CI system monitors Patchwork. Whenever a new patch is detected, the first process is to guess if it's a complete submission or a series of patches.

Recommended practice in the Quagga community is to use `git send-email` for the patch submission. This will make sure to avoid mailers reformatting (and corrupting) the patches and enforces a well understood format. In case of multiple commits in a single patch, `git send-email` will number in the subject of the emails as XX/YY (i.e. 03/20 for patch 3 in a series of 20). Our system can detect this and in case of a series of patches waits for the series to be complete before it's submitted to the CI system.

Once a single patch or a complete series of patches is detected, the patch (or the series of patches) is submitted. The base is always assumed to be the latest master from git. We want contributors to submit patches based on current code and not old revisions. It's also not possible to easy detect another base git commit as there is no indication in the patch email. (A fork model based on git pull requests would make this easier)

The first step in the CI system is to apply the patches in the correct order on top of the base commit and pack it up as a source tar file together with a file describing some extra information (Base Git commit, applied patch numbers, build time, reference back to this specific CI run etc).

If the patch(es) fail to apply, then an email is generated as seen in figure 3.

From: cisystem@netdef.org
To : contributor@example.com
Cc: mwinter@opensourcerouting.org, quagga-dev@lists.quagga.net
Subject: CI Testresult: FAILED (Re: [quagga-dev,14558] Problem with new link-params command in vtysh)
Date: January 22, 2016 at 12:20 AM

Continous Integration Result: FAILED
See below for issues.

This is an EXPERIMENTAL automated CI system.
For questions and feedback, feel free to email
Martin Winter <mwinter@opensourcerouting.org>.

Patches applied :
Patchwork 1789: http://patchwork.quagga.net/patch/1789
[quagga-dev,14558] Problem with new link-params command in vtysh
Tested on top of Git : eae18d1 (as of 20151209.135437 UTC)
CI System Testrun URL: https://ci1.netdef.org/browse/QUAGGA-QPWORK-214/

Get source and apply patch from patchwork: Failed

Applying Patchwork patch 1789
---
patching file vtysh/vtysh_config.c
Hunk #1 FAILED at 166.
1 out of 1 hunk FAILED -- saving rejects to file vtysh/vtysh_config.c.rej

Regards,
NetDEF/OpenSourceRouting Continous Integration (CI) System

Figure 3: Sample of a CI response email on a failed patch

Build Packages on each OS

After the Source is packaged, the CI system starts up a VM for several supported OS Distributions. The VM's are reset to a clean snapshot before the run.

At this time, we are building on:

- Ubuntu 12.04
- Ubuntu 14.04
- Debian 8
- CentOS 6
- CentOS 7
- FreeBSD 8
- FreeBSD 9
- FreeBSD 10
- NetBSD 6
- OpenBSD 5.8
- OmniOS (OpenSolaris)

On each system, the code is compiled, DejaGNU unit tests executed and finally a test package built.

The major challenge here is to pick some good `configure` options. Quagga can be built for many different configurations with additional options turned on or off. We pick 2 slightly different choices between the first pass to just build it and the 2nd pass in the package build, but in both cases, enabling all features.

With Quagga supporting a large set of Operating Systems, it is expected for this stage to frequently fail. Most contributors only test on their own OS. An example for a failed compile error is seen in Figure 4.

From: cisystem@netdef.org
To : contributor@example.com
Cc: mwinter@opensourcerouting.org, quagga-dev@lists.quagga.net
Subject: CI Testresult: FAILED (Re: [quagga-dev,14376,v3] lib, zebra: unify link layer type and hardware address handling)
Date: December 26, 2015 at 2:20 AM

Continous Integration Result: FAILED
See below for issues.

This is an EXPERIMENTAL automated CI system.
For questions and feedback, feel free to email
Martin Winter <mwinter@opensourcerouting.org>.

Patches applied :
Patchwork 1741: http://patchwork.quagga.net/patch/1741
[quagga-dev,14376,v3] lib, zebra: unify link layer type and hardware address handling
Tested on top of Git : eae18d1 (as of 20151209.135437 UTC)
CI System Testrun URL: https://ci1.netdef.org/browse/QUAGGA-QPWORK-204/

Get source and apply patch from patchwork: Successful

Building Stage: Failed

CentOS7 amd64 build: Successful
Debian8 amd64 build: Successful
Ubuntu1204 amd64 build: Successful
CentOS6 amd64 build: Successful
Ubuntu1404 amd64 build: Successful

Make failed for FreeBSD10 amd64 build: (see full log in attachment freebsd10_amd64_make.log)
```
  CC      rtread_sysctl.o
  CC      kernel_socket.o
kernel_socket.c:1127:41: error: no member named 'sdl' in 'struct interface'
      gate = (union sockunion *) & ifp->sdl;
                                   ~~~  ^
1 error generated.
*** Error code 1
Stop.
make[2]: stopped in /usr/home/ci/cibuild.204/quagga-source/zebra
```

Make failed for NetBSD6 amd64 build: (see full log in attachment netbsd6_amd64_make.log)
```
  CC      kernel_socket.o
kernel_socket.c: In function 'rtm_write':
kernel_socket.c:1127:39: error: 'struct interface' has no member named 'sdl'
*** Error code 1
Stop.
```
(continued...)

Figure 4: CI response example email on a failed compile

Basic Routing Protocol checks

The basic protocol checks is a simple selection of 2 to 4 tests out of the RFC Compliance suite for each protocol. This will verify that each protocol is able to start and form a neighbor. It also does some basic configuration commands as part of the protocol checks as it needs to run scripts to use the CLI for configuring the DUT.

Static Analyzer

In parallel to the Routing Protocol checks, we run a static analyzer. Currently this is the Clang Static Analyzer[6]. The result of the analyzer isn't evaluated, but just collected. We are still collecting ideas on how to translate the result of the static analysis into some simple pass/fail criteria. One possibility would be to just watch for changes in the output. The static analyzer doesn't require any dedicated or specialized hardware. As such, we run it on some cheap virtual web hosting VM (They are frequently sold around $40/year which is much cheaper than running it with the same number of CPU cores and memory on our own Hardware.)

Full RFC Compliance

For the Full RFC Compliance check, we are using a commercial Tool: Ixia IxANVL [3]. There are no open source tools available as far as we know. This tool runs on a Linux server, connecting to the DUT over multiple interfaces. It simulates the various topologies required to test each section of the RFCs. One of the main benefit of IxANVL is it's dynamic configuration (using Expect scripts) of the DUT. At the beginning of each Test, the tool logs into the DUT and changes the configuration. This gives us a very good test of the CLI as well. (Approx. 20% of the issues found are actually CLI issues because of the changes, i.e. crash when something gets unconfigured). All the tests are based on the previous test as well - there is no restart between the different tests on the same routing protocol. It makes the analysis a bit more difficult, but helps detecting bad state where a test may pass, but puts the DUT into a partial broken state.

For long time archiving and comparison, we collect all the logs (from the tester and from all the routing protocols and the PCAP files from each test and archive them into a SQL database. This allows an easy comparison against previous version and to pull reports as seen in Figure 5.

RFC Compliance Test Report
BGP4 Results

www.OpenSourceRouting.org
OpenSourceRouting
a project by the Network Device Education Foundation, Inc (www.NetDEF.org)

	Quagga 0.99.24	Git Master 2015-10-29	Git Proposed Round-5 2015-11-17	Git Proposed Round-5 2015-11-21	Git Proposed Round-5 2015-11-25
Type	QUAGGA	QUAGGA	QUAGGA	QUAGGA	QUAGGA
Commit ID	f191f1e	ffe7944	5ab56c7	440d4ce	496325d
Commit Date	2015-03-02	2015-10-29	2015-11-17	2015-11-21	2015-11-25
ANVL-BGP4-1.1 **MUST**	pass	pass	pass	pass	pass
	ANVL, setup verification				
	ANVL, Setup Verification DUT Listens on TCP port 179 for BGP4 Connection				
ANVL-BGP4-1.2 **MUST**	pass	pass	pass	pass	pass
	ANVL, setup verification				
	ANVL, Setup Verification Establish BGP4 connection to the DUT and transit to Established state				
ANVL-BGP4-1.3 **MUST**	pass	pass	pass	pass	pass
	ANVL, setup verification				
	ANVL, Setup Verification Router adds routes contained in the newly received Update Message to its routing table				
ANVL-BGP4-1.4 **MUST**	pass	pass	pass	pass	pass
	ANVL, setup verification				
	ANVL, Setup Verification Router forwards new Update routes				
ANVL-BGP4-2.1 **MUST**	pass	pass	pass	pass	pass
	RFC4271, Sect. 4, p 11, Message Formats				
	Message Formats The maximum message size is 4096 octets. All implementations are required to support this maximum message size.				
ANVL-BGP4-3.1 **MUST**	FAIL	FAIL	FAIL	FAIL	FAIL
	RFC4271, Sect.4.2, page 13, OPEN message format				
	OPEN Message Format After a TCP connection is established, the first message sent by each side is an OPEN message.				
ANVL-BGP4-3.2 **MUST**	FAIL	FAIL	FAIL	FAIL	FAIL
	RFC4271, Sect.4.2, page 13, OPEN message format				
	OPEN Message Format If the OPEN message is acceptable, a KEEPALIVE message confirming the OPEN is sent back.				

Test Report created at 2015-11-28 12:10:29 UTC Page 1 of 35

Figure 5: First page of example report for BGP4 RFC Compliance

Challenges of closed source testing tools In difference to using Open Source tools, we aren't allowed to share all the information. In the case of IxANVL, Ixia describes their proprietary knowhow in the way on how they run the tests and less on their actual code itself. This means that we are prohibited to share the exact test procedure with the community members. This causes extra work for us to either find the bug

ourself or document it well enough without revealing the test procedure.

The details as shown in Figure 5 are basically the parts we can publish to the public.

In our case, using the closed source tool still makes sense as we don't have the resources to build something similar from scratch and it helps us finding many errors. But we strongly suggest any Open Source community to make sure they are well aware of the limitations before picking closed source testing tools.

Protocol Fuzzer Tests

In addition to the Tests described above, we run regular tests with a Protocol Fuzzer (Spirent SPS-8000 - formerly part of MUdynamics). The protocol fuzzer mainly helps us for checking against potential DoS attacks by crashing the routing stack. Unfortunately, these tests take a very long time. A run against multiple protocols and a majority of the features can take 2 months of runtime. Part of the issue is that the protocol fuzzer only has 4 ports for 4 tests in parallel. The fuzzer we use knows the specific of the routing protocols and sends constant bad packets with different permutations (i.e. missing fields, fields too long, out of range, too short etc) and verifies between tests if the DUT still works (by bringing up a neighbor with the routing protocol. At this time, we don't publish the reports. There are basically 2 outcomes: No errors found (and the results are not interesting) or a problem found - which would be most likely a security issue which needs to be addressed before publishing the details.

Protocol Scalability and Performance

Quagga isn't a router - just the routing stack. Normal performance measurements for forwarding of data make no sense, but the performance of the route calculation, convergence and scalability still need to be tested. At this time, we don't have them integrated into the CI system. Nevertheless we use some Ixia Hardware Testers with IxNetwork [4] and a virtual Spirent TestCenter [8] for some tests. Plans for the future is to integrate these tests into the CI system.

Other testing considerations

We also try to avoid the issue where the same programmer who implements the protocol writes the tests. The RFC standards are sometimes not specific enough (or a newer standard or draft contradicts it) and it is important to get an independent view to agree on the implementation.

On top of these issues, it's a known issue that some large vendors contradict the standards by choice (or mistake) and it is sometimes more important to interoperate against them than following the standard. To this extend, we are running the same tests against Cisco and compare uncertainties against their implementation.

Testing at other similar Open Source Projects

Talking to other groups writing Open Source Routing code, we seem to be at a unique position to be the only organization to do any extensive testing. All of the other projects are only testing against some commercial vendors, against themselves (for scale) and sometimes against other open source tools.

The main issue is the lack of negative testing: There are very specific rules on how to react to bad packets from correcting, to ignoring packet and aborting the neighbor relationship. These tests can't be easy done against a "working" implementation. As an example, in BGP, update messages can have transitive flags and need to be forwarded unaltered even if not understood. This can (and frequently does) cause issues on the Internet where a bad update propagates and affects networks far away from the source.

Summary - Challenges

In conclusion, the major challenges in our testing are:

- *Make results trivial to understand.* In difference to commercial enterprises, there are always newcomers. Keeping the result as trivial to understand as possible is important. Even a failed result should be a good experience for a newcomer: He should see what's wrong, realize that he didn't waste the time from another community member and be encouraged to send in a fixed version.
- *Automate not just the tests, but the result parsing as well.* In our experience, automating the execution of a test is frequently the trivial part, while the parsing of the result is the difficult issue. Not all tools have a simple pass/fail result. I.e. Static Analyzer may show false warnings which are not easy to understand to someone seeing the output the first time. Same is valid for performance numbers.
- *Be careful with proprietary test tools*. Make sure to understand what part can be shared and reproduced by someone else. It might be ok to share the data, but useless if the community does not have the tools to reproduce the issues. As an example, our Protocol Fuzzer offers an excellent solution by having the capability to produce a Linux binary for a single failing test case with the permission to share this binary. This allows anyone else to reproduce the issue found.
- *Limited Open Source Tools*. The market for users requiring these tests are small. There are nearly no open source tools available. Some commercial companies developed their own tools in-house but generally refuse to share them. Some protocols (i.e. BGP) have a few more choices available, but nothing exists for others (i.e. IS-IS).
- *Cost of Test Equipment*. Without generous donations and loans, our job would be impossible. The cost of the test equipment in our lab as required is generally prohibitive.
- *Limit runtime to provide feedback*. Running tests for days or weeks before being able to give any feedback makes the interaction between developers and the testing difficult or impossible. We want to give feedback before the contributor moved on to the next project. At the time of the feedback, his memory should be still fresh and it should allow him a quick turnaround time to send a fixed version.
- *Parallel execution for features*. Many features can be tested in parallel. But running everything in parallel might hide some bugs like bad initialization or not freeing some

memory. Running multiple tests on the same setup in sequence (with reconfiguration, but no restarting between) helps finding these bugs.

- *Parallel execution for OS* While everyone talks about hosting VM's in the cloud, the cost of many of these clouds are prohibitive and the OS choices are too limited. Outside of Linux, the choices for OS on hosted VM's are limited. At the same time, running the tests for every OS on our own infrastructure is too expensive. We hope to expand the testing, but soon approach a situation of 100's of VM's testing in parallel for a single commit.

Author Biography

Martin Winter is a technical lead and cofounder of the Network Device Education Foundation. His research interests include routing platforms, networked systems, and software engineering as well as OpenSourceRouting, a non-profit project to improve Quagga. Winter received a BS in computer science from Brugg-Windisch HTL in Switzerland. He is co-chair of the Open Source Working Group at RIPE (European Internet Provider Forum).

Contact him at mwinter@netdef.org.

Questions?

For questions about testing contact the author at `mwinter@netdef.org`. For other questions about the Network Device Education Foundation (NetDEF) or the OpenSourceRouting project, please contact `info@netdef.org`

References

[1] Atlassian. Bamboo continuous integration, deployment and delivery system. https://www.atlassian.com/software/bamboo.

[2] Free Software Foundation, Inc. Savannah software forge for people committed to free software. http://savannah.gnu.org/.

[3] Ixia. IxANVL automated network validation library. http://www.ixiacom.com/products/ixanvl.

[4] Ixia. IxNetwork network topology testing and traffic analysis. http://www.ixiacom.com/products/ixnetwork.

[5] Kerr, J. 2010. Patchwork web-based patch tracking system. http://jk.ozlabs.org/projects/patchwork/.

[6] LLVM Project. Clang static analyzer. http://clang-analyzer.llvm.org/.

[7] Quagga Community. Quagga routing protocol suite. Homepage: http://quagga.net/ with official Git at http://savannah.nongnu.org/git/?group=quagga.

[8] Spirent Communications. Virtual TestCenter layer 2-7 test, measurement and protocol emulation. http://www.spirent.com/Products/TestCenter.

[9] The Jenkins Project. Jenkins open source automation server. https://jenkins-ci.org/.

[10] Travis CI, GmbH. Travis CI. https://travis-ci.com/.

License