Testing a Community Network Testbed Control System

Bart Braem, Jeroen Avonts, Chris Blondia, Steven Latré
Department of Mathematics and Computer Science
University of Antwerp - iMinds - MOSAIC Research Group
Middelheimlaan 1, B-2020, Antwerp, Belgium
Email: {firstname.lastname}@uantwerpen.be

***Abstract*—Development and continuous operation of network management systems is a major challenge to future networks, where a large number of semi-independent devices jointly try to realize a working network. This work considers network testbed management, and more specifically on testbed management software for community networks. Because of the inherently unstructured and chaotic nature of community networks, managing components inside a community network with a frequently varying and unpredictable performance is particularly challenging. This paper focuses on the verification of community network testbed control software which has to cope with these challenges. We show how the application of a container-based unit testing approach has a positive impact on development efforts and testbed stability.**

I. Introduction

Community networks can be described as "bottom-up broadband", or "networks by the people for the people". Evolving away from the centrally managed and commercially operated ISP networks, community networks have appeared around the world as an organic and highly distributed approach to networking [1]. Apart from the economic and social opportunities and possibilities, a number of challenges in terms of network operation and management quickly arise. Because community networks are managed by the members using them, they operate completely distributed, requiring robust self-management and where possible distributed quality of service within a highly heterogeneous network environment.

A number of research projects are currently studying community networks and their application in various scenarios. The GAIA IRTF working group is studying methods which enable global access to the internet for all, and considers community networks as a means to realize this ambitious goal [2]. Clommunity is a European FP7 project started in 2013 that researches the deployment of distributed, cloud-like infrastructures in the highly distributed community networks [3]. The European FP7 CONFINE project started in 2011 and studies the feasibility of community networks as a future internet, including related non-technical aspects such as social and economical opportunities and challenges [4].

To allow for remote experimentation on and with community networks by both community members and interested researchers, the CONFINE project has developed and currently operates the Community-Lab experimental infrastructure, a testbed for community networks experimentation. Community-Lab is based on the idea of embedding relatively modest research nodes in the partnering community networks, which can then be used to experiment with existing or new protocols, software or even hardware to investigate and improve performance in community networks.

Strongly modeled after PlanetLab [5], the project has developed a distributed testbed controller to manage the testbed research nodes [6]. After logging in to the Community-Lab controller website, researchers let the controller deploy their slices and slivers in research devices located in different community networks around Europe. When allocated, the slices and slivers are available over an IPv6 mesh tunnel with strong security properties (tinc), which allows providing uniform access over the independent and heterogenous networks [7]. Figure 1 gives an overview of the Community-Lab architecture where community network devices (CD) connect CONFINE nodes (CN) to the testbed server which runs the Community-Lab controller software. The slivers are depicted as diamond shapes, joined in slices.

Although Community-Lab tries to follow the PlanetLab pull-based management structure where possible, the development of the Community-Lab controller still proved to be very challenging and error prone. Given the nature of distributed systems, it is expected that the development of critical software such as a testbed controller is hard [8] [9]. Moreover, Community-Lab nodes are often installed in remote locations where manual intervention is cumbersome. An unstable Community-Lab controller would cause a large logistical overhead and could cause community network members currently hosting nodes voluntarily to stop their commitment. This paper

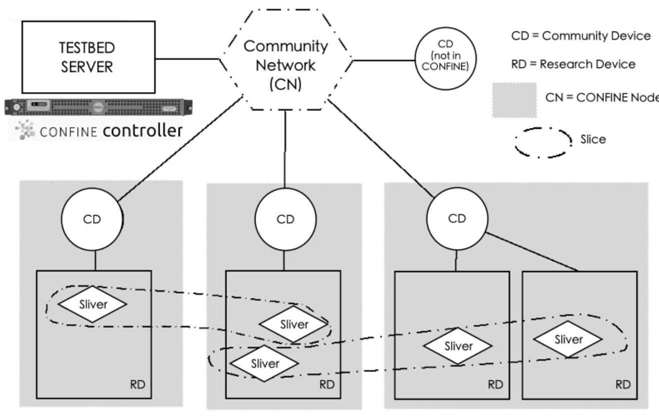

Fig. 1. Overview of the Community-Lab architecture

will explain how testing has helped overcome this stability challenge.

Testbed management software can take multiple approaches, we highlight the most popular ones. A number of systems follow the Slice-based Facility Architecture (SFA) from PlanetLab [10] such as GENI [11], while others follow a more operational approach such as the Orbit Management Framework (OMF) [12]. In both cases, development of cross-platform cross-testbed management software is a daunting task [13]–[15], although projects like Fed4FIRE try to tackle this by bringing together testbed operators and tool developers [16].

Unfortunately, to the best of our knowledge very little network research software is tested or verified more structurally. Often this is caused by time constraints, combined with a fear of the burden of testing and the uncertainty of the scale of testing benefits. In this respect, this work wants to motivate people to invest in testing. One interesting exception is the jFed tool developed in Fed4FIRE, used for daily verification of the testbeds federated in the project [17]. However, this verification only entails a high-level functional verification (e.g. logging in, reserving a slice). While better than no testing at all, this approach comes with a number of limitations. E.g., it will not detect crashing nodes and does not limit impact on production testbed resources. This work will outline a more elaborated testing strategy.

The contributions this work consist of the description of the application of unit testing on a community network testbed control system. The impact of deploying the testing system on development of a testbed control system is measured by considering the bug repository, which shows a significant difference.

The results of this work are more generally applicable, other related testbed control systems such as PlanetLab or OMF could also be tested using a very similar approach, resulting in more stable code and less uncertainty and doubt about the quality of these tools. Moreover, other network management systems could also be verified based on the proposed approach, because of related goals such as efficiency or isolation.

The remainder of this work is structured as follows. First, the goals for a testing system will be outlined. Then, the testing strategy to fulfill these objectives is detailed, including an overview of the tests building on such a strategy. In what follows, the impact of testing on the development of testbed control software is assessed with an evaluation of bug report data. Finally we present some conclusions and outline future work.

II. Testing Goals

Because of the high up-front effort when unit testing a software system, for Community-Lab a number of goals were first outlined before starting the testing system implementation. Based on this, a viable testing approach was chosen and implemented.

A. *Isolated*

An important testing strategy goal is to isolate system under test from the production systems. Otherwise, testing the real systems might cause high-impact failures when the testing system uncovers bugs (which is paradoxically a good result for the tests). A number of strategies could be followed to realize isolation, including testing on separate hardware or with different software installations.

More interesting from a development perspective is the isolation of the testing system from the system under test. Only standardized, publicly usable APIs should be used to test the system, rather than resorting to low-level quick-and-dirty hacks to test functionality. In this respect, testing a system will also make its APIs more feature complete, as all manual user interactions now have to be realized via the API. This implies that isolation should happen both at a resource level and a functional level.

B. *Reproducible*

One of the best results of a testing run is the uncovering of (previously unknown) bugs[1]. Preferably, these bugs are found before they arise in a production environment, although it is also beneficial to reproduce encountered erroneous situations in test code or even use tests to drive development [18].

To this end, similar to (network) experiments and benchmarking [19], the ability to accurately and deterministically reproduce test results with detailed background information is crucial in order to be able to properly debug and fix the issue triggered by the test.

C. *Realistic*

The testing system has to perform realistic tests, to be certain that the uncovered issues are practically viable on the one hand and to make sure that all possible issues in a production system can be reproduced by the testing system on the other hand. To realize this goal, the system under test clearly should closely match the real production systems, which entails testing with similar software, hardware and network settings.

D. *Automated*

Ideally, a test system should operate completely independently. It should not require any manual intervention from developers with tests triggered when applicable, at the most efficient moment (also see the next goal). Clearly, it is unfeasible to expect a developer or researcher to manually trigger a test run each time code is committed. This should be orchestrated intelligently.

E. *Efficient*

Although sometimes overlooked and a lower priority than the previous goals, it is important for a testing system to be efficient. Developers should not have to wait multiple days before their code is tested, as this will encourage ignoring the obsolete results from the testing system. Notice how this might contrast with the realism requirements outlined above.

[1]The best test result is an error-free test run with maximum coverage.

III. Testing Approach

To fulfill the goals outlined above, during the CONFINE project a testing system for Community-Lab was set up. Figure 2 gives an overview of the different components involved in the testing system.

A. *Test Control Software: Jenkins*

The top component is formed by the existing test control software Jenkins [20], which allows managing and reporting on tests via a user-friendly web interface. This directly helps to fulfill the automation goal, Jenkins is equipped extensive and flexible test management features. In the case of Community-Lab, all tests are triggered by monitoring the Community-Lab development git repositories. Tied to git, tests are started upon each change to the git repositories. The result is automated testing of every version, which allows to go back and verify the test results for previous code to trace back the cause of issues.

Adding to the efficiency goal, Jenkins allows the configuration of different testing schedules based on git repository polling schedules. In Community-Lab this is used to run the most important tests immediately while postponing more elaborate and longer tests, see the test descriptions in what follows for more detailed information.

Moreover, Jenkins allows running multiple jobs in parallel on a single machine or on multiple slave nodes. This parallelization offers additional efficiency improvements which are easily enabled by adding more hardware to the testing setup. In the case of the Community-Lab testing system older hardware has been used to speed up testing at limited cost.

Finally, for Community-Lab Jenkins is configured to run completely from shared scripts. This in contrast to the typical Jenkins setup, where a lot of code is hidden deep inside the configuration of Jenkins. In the case of Community-Lab, all Jenkins code is run from publicly available scripts, allowing developers to quickly and easily reproduce the testing setup locally. This strongly contributes to the reproducibility goal.

Notice that the specific software choice is arbitrary, any testing system or continuous integration system with similar features could have been chosen. Popular alternatives include hosted Jenkins setups like CloudBees [21], the Jenkins predecessor Hudson [22] and the hosted continuous integration system Travis CI [23]. In this case Jenkins was selected because of its open source code base and highly customizable plugin system.

B. *Emulated and Hardware Testbeds*

To perform realistic tests, the Community-Lab testing system is set up to test both an emulated testbed and real hardware. Both components are controlled by Jenkins over SSH, as shown in Figure 2.

The real hardware testbed consists of Community-Lab hardware dedicated to testing, connected over a network built after a community network. The emulated testbed on the other hand is built after the Community-Lab testbed, but with virtual nodes (run on virtual machines and containers) instead of real hardware.

The emulated testbed has a slightly lower performance because of the virtualization overhead. The only other major difference is the connection to nodes over plain SSH instead of tinc tunnels, which are unnecessary because all emulated nodes reside on the same physical system.

In essence, by design both the real hardware testbed and the emulated testbed are very similar to each other and to Community-Lab, to allow a high degree of realism.

C. *Containers*

Linux containers (LXC) are a lightweight virtualization solution which has recently become very popular in technologies such as Docker [24], [25]. Compared to full virtualization solutions containers can provide a more lightweight isolation by sharing one kernel with multiple isolated process groups [26].

Community-Lab relies on LXC to try and guarantee stable and (to a certain degree) non-interfering experiments on its embedded, relatively limited research devices.

For the testing with the emulated testbed, two containers are introduced. One researcher simulator, which implements the role of a researcher using the API to control Community-Lab resources, and one testbed container. The latter uses the Virtual CONFINE Testbed (VCT) container, which implements both a controller and mechanisms to spawn virtual nodes (with KVM virtualization). In the case of real hardware tests the VCT container is replaced by real nodes on separate hardware and controller software running on a separate machine.

For the testing system, the containers offer strong isolation, both from resources in case of crashed testbed controllers but also for the API. Completely separated systems allow properly testing the API, without resorting to hacks to test a given feature. Moreover, containers allow restarting from the same, clean environment for each test. This allows inter-test isolation and better reproducibility, while remaining efficient.

D. *Coverage Tracking*

Automated unit testing comes with an additional advantage: the test coverage can be measured. In Community-Lab, this allows us to measure test coverage to locate untested code. Often this obsolete code is caused by missing tests for certain functionality, but sometimes this also indicated recent code changes which were not tested yet. In both cases, test coverage tracking allowed focusing testing effort with minimal effort.

Another interesting advantage of test coverage tracking turned out to be dead code removal. While discussing code with low coverage, Community-Lab developers discovered various parts of code which could be removed without any lost functionality because the code was simply not used anymore. Because of the high complexity of testbed control software, any removed code is beneficial to lower the maintenance burden.

E. *Test Jobs*

A naive approach would consist of running all tests in one test job (running all tests consecutively), or running all tests as separate jobs. While the latter allows for highly parallel

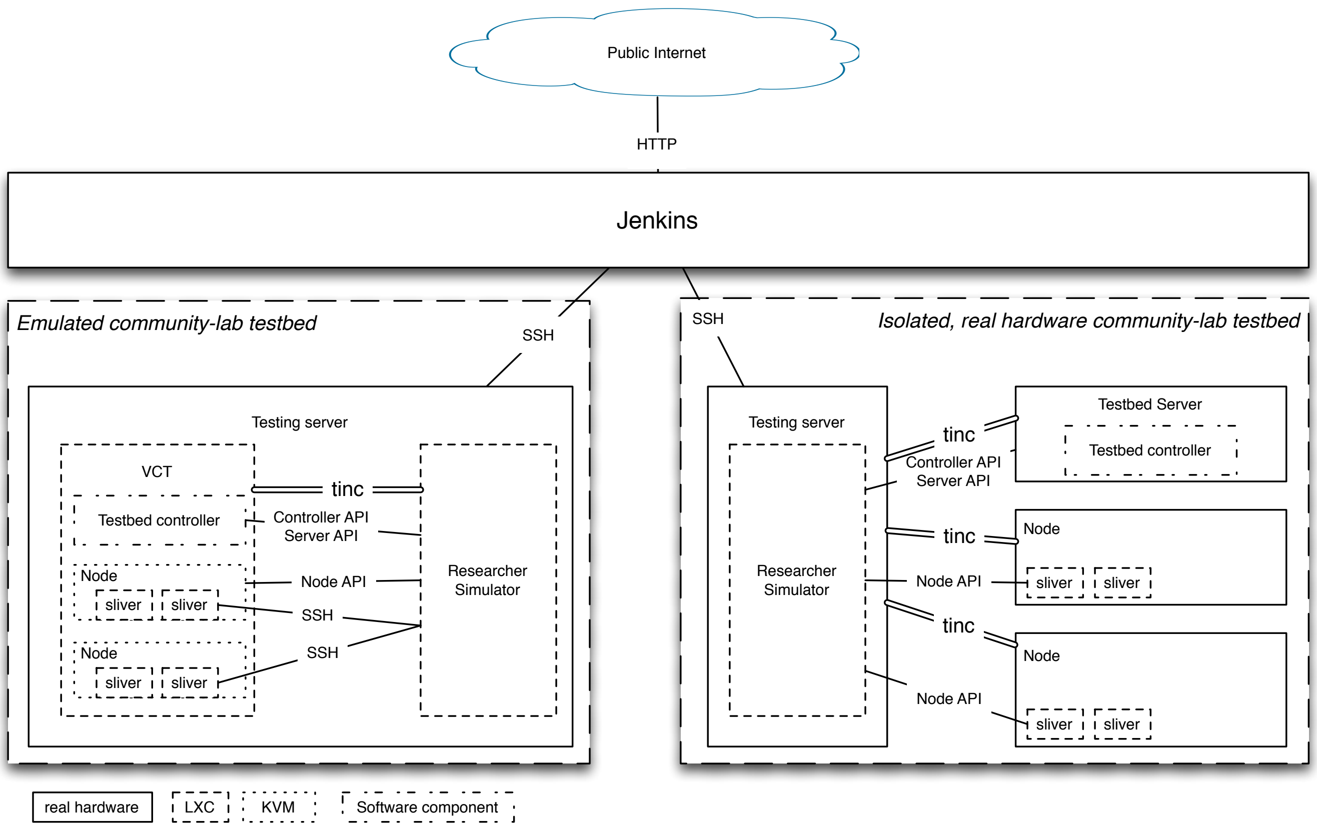

Fig. 2. Testing architecture overview

testing, the burden of adding all tests to Jenkins is high. For Community-Lab testing, the tests were organized in multiple logical jobs.

The first job consists of the simple, basic API tests. They involve logging in, creating slices and slivers and testing permissions related to those actions. This job triggers every 5 minutes, depending on git repository information.

The next job consists of the strict tests, which is a superset of the basic API tests. These tests trigger non-critical bugs which are strictly issues but e.g. depend on improvements to external software. This test only triggers daily because these issues are considered to be lower priority.

The installation tests start from a clean environment without any Community-Lab code and try to install a new testbed controller from scratch. Moreover, the tests also test upgrading from a previous version and from the stable to the development branches of the controller code. This job only triggers hourly.

The integration tests run more elaborated testbed scenarios, e.g. allocating multiple nodes and verifying network connectivity between the different nodes. These tests take considerably longer but are essential to guarantee basic testbed functionality. They are triggered hourly.

The real hardware job run the basic tests and the integration tests, on real research device hardware. These tests are essential to avoid hardware interoperability issues, which could otherwise possibly crash research devices in remote locations. This job triggers hourly.

The coverage tests run the same basic tests and integration tests, but considerably slower as test code coverage is measured. Because this takes longer and coverage is not essential, this job is only triggered daily.

Community-Lab also provides an Object-relational mapping (ORM) library to simplify API integration. While non-essential, this code is used by Fed4FIRE integration code. Therefore it is also included in the testing system but only triggered daily.

Finally, any new job will trigger the VCTBuilder job which will trigger the Firmware Builder job. The latter is responsible for checking for new CONFINE research device firmware, which will be built from scratch upon each commit. The generated firmware is uploaded to a shared server so any new firmware release only has to be built once. This firmware can then be included in a new build of the VCT container, possibly based on new testbed controller software. The result of these two special jobs which are not actual test jobs is the formation of a build pipeline, which allows efficiently building and verifying firmware and controller code. Figure 3 gives an overview of the job dependencies in this configuration.

IV. Evaluation

The testing system defined above has been implemented in the CONFINE project and is actively running at http://testing.confine-project.eu, with test reports and test statistics publicly available. A logical next step entails the evaluation of the impact and efficiency of this effort. When talking to the CONFINE developers they are very happy with the testing

Project	# Reports mentioning testing	Total # reports	Mention percentage
Community-Lab controller	54	193	28%
Community-Lab firmware	10	166	6%

TABLE I. THE RELATIONSHIP BETWEEN THE TESTS AND BUG REPORTS.

Project	# Pre testing	# Post testing	Reports rate (reports per month)
Community-Lab controller	41	152	3.7 to 15.2
Community-Lab firmware	102	64	6.8 to 6.4

TABLE II. THE RELATIONSHIP BETWEEN THE TESTS AND BUG REPORTS PRE AND POST THE INTRODUCTION OF TESTING.

Project	#Pre resolved rate	#Post resolved rate	Open bug rate
Community-Lab controller	0.82	16	2.88 to -0.8
Community-Lab firmware	2.9	7.9	3.9 to -1.5

TABLE III. THE RELATIONSHIP BETWEEN THE RESOLVED BUG RATES AND THE OPEN BUG RATE.

system, this section tries to show these benefits with hard numbers.

To prove the positive impact of this testing approach on the project and to motivate other researchers to invest in this effort and apply similar testing strategies, in what follows we present the results from an analysis of issued posted to the Community-Lab bug tracker at http://redmine.confine-project.eu almost one year after the start of the testing efforts.

When the testing system was introduced, the Community-Lab was already relatively mature after two years of development. Related work indicates that the number of bug reports in a project typically declines [27]. Given the project plan and the inspiration by existing systems, it was expected that the Community-Lab software would be relatively mature by the beginning or the testing efforts.

A. Bug reports

At the moment of writing, the test suite contains a total of 143 tests that cover 75% of the lines of the Community-Lab controller code, a very decent coverage considering the size of the project. As shown in Table I, after less than one year the number of bug reports mentioning the testing framework for the Community-Lab controller accounts already for 28% of the bug reports in only ten months. For the Community-Lab firmware, the bug reports created from the testing framework account for 6%. This illustrates a shifting approach and increasing adoption of testing by Community-Lab developers, which correlates with their positive feelings. The lower adoption by firmware developers correlates to a significantly lower testing of the CONFINE firmware code, which is not explicitly tested at this moment.

Next, consider the bug reports in the set of *pre test framework* reports, comparing to the *post test framework* reports, as shown in Table II. The rate of bug reports per month (brpm) for the Community-Lab controller increase from 3.7 brpm to 15.2 brpm, which is more than four times as many. Considering only the post test framework reports, the testing framework is mentioned in 36% of the reports since its introduction. The first three months, the contribution of bug reports mentioning the testing framework accounted for 25%. The last three months, this increased to 45% as the testing framework matured and the developers relied more on this system.

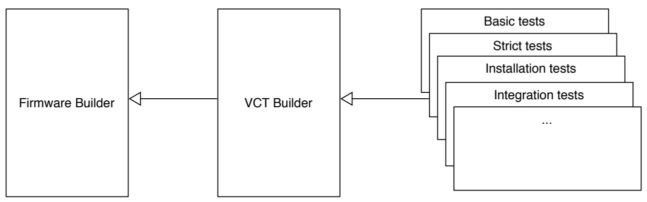

Fig. 3. Overview of the testing pipeline

For the Community-Lab firmware, the bug report rate did not change significantly since the introduction of the testing framework. However, out of the 64 bug reports since the introduction, 10 reports mention the testing framework, accounting for 16% of the bug reports. Clearly, firmware development in general was more active before the introduction of the testing system and developers are less actively using the testing framework. This can be explained by the testing system which until now focused on testing the controller API rather than directly testing the firmware API.

B. Resolved bugs

Looking at the number of bugs resolved since the introduction of testing, 47 out of the 54 Community-Lab controller issues mentioning the testing framework are resolved, this corresponds to 87%. Before the introduction of the testing framework the rate of resolved bugs for the Community-Lab controller was around 0.82 resolved bugs per month and thus lower than the rate of new filed bug reports shown in table II (3.7 brpm). Since the introduction of testing, this resolve rate increased to 16 resolved bugs a month. The testing framework is mentioned in 29% of those resolved bugs. This clearly indicates adoption for testing and more specifically verification of bug fixes, which is considered to be a very good practice.

For the Community-Lab firmware we find an increase in resolved bug rate from 2.9 resolved bugs a month (less than the 6.8 brpm) to 7.9 which is greater than the 6.4 brpm report rate (implying the number of open issues is declining). For the latter, 10% mentioned the testing framework. An overview of these results can be found in Table III.

As demonstrated, the resolved bug report rate exceeds the corresponding new bug report rate meaning that since the introduction of the testing framework, more bugs are resolved than filed in a month. In other words, the rate of open bugs is a negative rate since the introduction of the testing framework. We can conclude that the testing framework had and still has an impact, especially on the Community-Lab controller. This confirms the positive impact of a testing system, besides the positive feedback from the developers.

V. CONCLUSIONS AND FUTURE WORK

This work has described an automated, container based testing system with a demonstrated impact on the stability and efficiency of community-network testbed development. The up-front goals for a testing system were outlined and a design to fulfill these goals has been described, together with the positive impact shown by looking at bug report statistics.

The authors are aware that although the impact of testing on this system was shown to be positive and although the developers appreciate this system, more extensive verification of the claims in this work are necessary. An immediate candidate for application is the Community-Lab firmware API, which is not directly tested extensively at this moment. PlanetLab is another excellent candidate, because the SFA architecture adopted in Community-Lab is exactly this of PlanetLab. Moreover, although very powerful, the OMF framework is notoriously hard to install and operate for prolonged periods. We believe this popular testbed management system could also benefit from more elaborated testing.

The Community-Lab tests themselves could also be improved to cover more cases including user interfaces, because for an experimental facility properly working user interfaces are critical for user satisfaction and testbed adoption in general. To this end, currently a number of tests are being added which use the Django tests to verify functionality. Moreover, the project is actively considering using Selenium, which has a strong track record in user interface testing [28].

Finally, to have a stronger verification of robustness, a number of communication tests are also being designed. The goal is to deterministically interrupt network connectivity at various points during the interaction with the testbed, to verify resilience. Because of strong dependence on timings which could vary strongly with Jenkins slave performance, no such tests are present at this moment.

Acknowledgment

This work is supported by the CONFINE Integrated Project 288535. The authors would like to thank the Community-Lab developers for their feedback.

References

[1] J. Avonts, B. Braem, and C. Blondia, "A questionnaire based examination of community networks," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*. IEEE, 2013, pp. 8–15.

[2] B. Braem, L. Navarro, E. Pietrosemoli, E. L. de Redes, C. Rey-Moreno, A. Sathiaseelan, M. Zennaro, and A. S. ICTP, "Global access to the internet for all j. saldana, ed. internet-draft university of zaragoza intended status: Informational a. arcia-moret expires: December 20, 2014 universidad de los andes," *Internet-Draft*, 2014.

[3] U. C. Buyuksahin, A. M. Khan, and F. Freitag, "Support service for reciprocal computational resource sharing in wireless community networks," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a*. IEEE, 2013, pp. 1–6.

[4] B. Braem, C. Blondia, C. Barz, H. Rogge, F. Freitag, L. Navarro, J. Bonicioli, S. Papathanasiou, P. Escrich, R. Baig Viñas, A. L. Kaplan, A. Neumann, I. Vilata i Balaguer, B. Tatum, and M. Matson, "A case for research with and on community networks," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 3, pp. 68–73, Jul. 2013.

[5] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[6] A. Neumann, I. Vilata, X. León, P. E. Garcia, L. Navarro, and E. López, "Community-lab: Architecture of a community networking testbed for the future internet," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on*. IEEE, 2012, pp. 620–627.

[7] S. Khanvilkar and A. Khokhar, "Experimental evaluations of open-source linux-based vpn solutions," in *Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on*. IEEE, 2004, pp. 181–186.

[8] S. Muir, "The seven deadly sins of distributed systems." in *WORLDS*, 2004.

[9] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 249–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685068

[10] T. F. J. Augé and T. Friedman, "The open slice-based facility architecture (open sfa)," 2012.

[11] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "Geni: a federated testbed for innovative network experiments," *Computer Networks*, vol. 61, pp. 5–23, 2014.

[12] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "Omf: a control and management framework for networking testbeds," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 54–59, 2010.

[13] A. de Moor, "Improving the testbed development process in collaboratories," in *Conceptual Structures at Work*. Springer, 2004, pp. 261–274.

[14] J. Jia and Q. Zhang, "A testbed development framework for cognitive radio networks," in *Communications, 2009. ICC'09. IEEE International Conference on*. IEEE, 2009, pp. 1–5.

[15] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the internet," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 59–64, 2003.

[16] W. Vandenberghe, B. Vermeulen, P. Demeester, A. Willner, S. Papavassiliou, A. Gavras, M. Sioutis, A. Quereilhac, Y. Al-Hazmi, F. Lobillo *et al.*, "Architecture for the heterogeneous federation of future internet experimentation facilities," in *Future Network and Mobile Summit (FutureNetworkSummit), 2013*. IEEE, 2013, pp. 1–11.

[17] T. Walcarius, W. Vandenberghe, B. Vermeulen, P. Demeester, and D. Davies, "Health monitoring of federated future internet experimentation facilities," in *Networks and Communications (EuCNC), 2014 European Conference on*. IEEE, 2014, pp. 1–5.

[18] L. Madeyski, *Test-Driven Development*. Springer, 2010.

[19] S. Bouckaert, J. V.-V. Gerwen, I. Moerman, S. C. Phillips, J. Wilander, S. U. Rehman, W. Dabbous, and T. Turletti, "Benchmarking computers and computer networks."

[20] "Jenkins," http://jenkins-ci.org/.

[21] "Cloudbees," https://www.cloudbees.com/.

[22] "Hudson continuous integration," http://hudson-ci.org/.

[23] "Travis ci - free hosted continuous integration platform," https://travis-ci.org/.

[24] P. B. Menage, "Adding generic process containers to the linux kernel," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 45–57.

[25] "Docker," https://www.docker.com/.

[26] J. Ahrenholz, "Comparison of core network emulation platforms," in *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*. IEEE, 2010, pp. 166–171.

[27] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?" in *Usenix Security*, 2006.

[28] A. Holmes and M. Kellogg, "Automating functional tests using selenium," in *Agile Conference, 2006*. IEEE, 2006, pp. 6–pp.