Creating Repeatable Computer Science and Networking Experiments on Shared, Public Testbeds

Sarah Edwards
GENI Project Office
Raytheon BBN Technologies
10 Moulton St.
Cambridge, MA 02139
sedwards@bbn.com

Xuan Liu
University of Missouri,
Kansas City
5100 Rockhill Rd
Kansas City, MO 64110
xuan.liu@mail.umkc.edu

Niky Riga
GENI Project Office
Raytheon BBN Technologies
10 Moulton St.
Cambridge, MA 02139
nriga@bbn.com

ABSTRACT

There are many compelling reasons to use a shared, public testbed such as GENI, Emulab, or PlanetLab to conduct experiments in computer science and networking. These testbeds support creating experiments with a large and diverse set of resources. Moreover these testbeds are constructed to inherently support the repeatability of experiments as required for scientifically sound research. Finally, the artifacts needed for a researcher to repeat their own experiment can be shared so that others can readily repeat the experiment in the same environment.

However using a shared, public testbed is different from conducting experiments on resources either owned by the experimenter or someone the experimenter knows. Experiments on shared, public testbeds are more likely to use large topologies, use scarce resources, and need to be tolerant to outages and maintenances in the testbed. In addition, experimenters may not have access to low-level debugging information.

This paper describes a methodology for new experimenters to write and deploy repeatable and sharable experiments which deal with these challenges by: having a clear plan; automating the execution and analysis of an experiment by following best practices from software engineering and system administration; and building scalable experiments.

In addition, the paper describes a case study run on the GENI testbed which illustrates the methodology described.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; C.2 [**Computer Systems Organization**]: Computer-Communication Networks; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Experimentation, Management, Measurement, Performance

Keywords

networking testbeds, experiment methodology, repeatable experiments, system administration, GENI

1. INTRODUCTION

Public, shared computer science and networking testbeds such as GENI [9], Emulab [34], PlanetLab [28], ORBIT [31], DeterLab [24], Orca [10], FIRE [13], and others make a variety of resources available to experimenters conducting related research. The *wide variety and volume of compute and networking resources* available in these testbeds would be prohibitively expensive or impossible to have in a privately held lab. Some of these (e.g. PlanetLab, GENI) also provide *geographic diversity* of their resources.

By their very nature, community testbeds support *repeatability* [14] of experiments. In particular, these testbeds already provide standardized ways to request resources, operating system images, software, etc. These standard mechanisms are used to make repeated runs of the same experiment, varying parameters and configurations as necessary to make a scientifically sound argument.

Part of the promise of a shared, public testbed is that it makes it possible to not just repeat your own experiments but to *publish and share experiments* so that others can repeat them. While tools to make publishing experiments easy are certainly needed, it is possible to do so today since the artifacts needed to share an experiment are very similar to those needed by a researcher to repeat their own experiment. If a researcher automates, stores, and documents their experiment enough to reliably repeat it themselves, then they have most of the pieces needed to share it with others. Moreover, the testbeds represent a solution to one of the biggest challenges of reproducible experimentation; by providing widely available access to the laboratory and tools used to perform an experiment, another researcher can quickly and easily repeat an experiment using the same or equivalent equipment used in the original experiment.

Being able to repeat someone else's experiment however, does not guarantee the ability to reproduce the same results or the soundness of the original experiment. It is up to the experimenter to study the effects of the environment and produce repeatable results. These are issues beyond the scope of this article but are addressed by others [16].

This article introduces novice users of community testbeds to the process of creating and documenting an experiment so that it is repeatable by themselves and others.

Properties of Shared, Public Testbeds: Some of the very features of community testbeds which make them compelling present very different challenges from the situation where the experimenter either is, or personally knows, the operator of the testbed:

1. *Large, diverse topologies are possible.* Therefore experimenters are likely to use more resources than can be managed manually.

2. *Some (or all) resources are rare and have heavy demand.* Therefore the experimenter can not keep some resources for long periods.

3. *A shared testbed is not run by the experimenters.* Therefore, an experimenter may need to talk to the operators of the testbed for access to low-level debugging information.

4. *A geographically diverse, federated testbed is maintained by many different groups (GENI, PlanetLab) or at least reliant on others for power, network connectivity, etc.* This means outages and maintenances occur at a variety of times. Therefore all resources in the testbed may not be continuously available.

In summary, the very nature of a community testbed poses a unique set of challenges to the researcher. There is an inherent need for automation since researchers need to be able to bring their experiment up and down quickly, even possibly switching between equivalent resources. Moreover, experimenters need to be able to work with other members of their community to execute their experiment and resolve bugs.

To the best of our knowledge there is no work that provides guidelines for how to overcome the basic challenges in the use of public testbeds and that is the gap that we envision this paper to fill. This paper is intended to be a short and comprehensive guide for experimenters to deploy computing and networking experiments in community testbeds. The reader is encouraged to also turn to the multitude of resources that provide advice about how to systematically design their experiments and build a rigorous experimentation process.

The authors of this paper draw from their experience in supporting and deploying experiments in GENI. Although we are steeped in the GENI community we believe that the guidelines in this paper apply to researchers using other public infrastructure for their experimentation.

2. RECOMMENDED METHODOLOGY

Based on these properties, we believe that there are a few main pieces of advice for beginning experimenters:

1. Formulate a clear plan for your experiment; Section 3

2. Systematically design and execute your experiment:

   (a) Automate the execution and analysis of your experiment following best practices; Section 4

   (b) Build scalable experiments; Section 5

This paper covers each of these items in turn and explains them in the context of a case study run on the GENI testbed.

While much of the methodology in this article is generally applicable for running computer science and networking experiments, it becomes critical when using a community testbed.

2.1 Case Study: OSPF Convergence Time

Throughout this paper, we present a case study which illustrates the key points covered in this paper to show how to deploy an experiment systematically and repeatably.

This case study was done as a portion of a larger series of experiments to evaluate the algorithm described in [22] for the selection of standby virtual routers in virtual networks. The case study topology is a wide area, layer 3, virtual network formed from software routers running the OSPF (Open Shortest Path First) routing protocol. During the experiment, we monitor the OSPF updates at each router to determine how long changes take to propagate through the network after a network event (failure/recover).

In this paper we explain how to plan the experiment, choose a trivial initial setup, orchestrate and instrument the experiment, and then show how to scale up the experiment to a multi-node geographically distributed topology.

3. FORMULATING A CLEAR PLAN

3.1 What vs. How

Before reserving any resources, it is important to make a detailed plan for your experiment.

At a minimum there are two parts to this plan:

1. Experiment Design: What do you want to do?

2. Experiment Implementation and Deployment: How are you going to do it?

Determining the *what* and the *how* (and distinguishing between them) may be the most important part of your experiment and it is worth spending time to get it right.

3.1.1 What do you want to do?

First consider what question you are trying to answer. This requires identifying two parts: what is the system you want to test; and what question about that system are you trying to answer [16].

A picture explaining how the system under test works [16] in the real world (or how it would work if it were actually deployed) is often useful and can be used to describe the boundaries of the system. For even the simplest experiment, a picture and some notes describing metrics to collect, and the parameters to vary are invaluable [16].

3.1.2 How are you going to do it?

Second consider what pieces must be implemented faithfully in order to answer your question. The items within the system boundary should be faithfully represented, but items outside the system boundary may not require as much fidelity. A picture is also helpful to describe your experiment design and implementation.

Both your "what" and "how" pictures might be simple, hand-drawn pictures. With the prevalence of digital cameras, these are easily shared. Drawing tools and Powerpoint are other options for creating these simple but useful drawings.

Questions you might ask about "how" to build your experiment include:

- What resources are needed to run this experiment?

- How many resources (i.e., nodes, bandwidth, software, etc.) are needed for this experiment?

- What parameters should be varied? What metrics should be measured?

- How large should the experiment be? How many more resources are needed to scale the experiment?

Use the answer to these questions to select an appropriate testbed.

3.2 Select and Learn the Testbed

Currently, there are many testbeds available housed all over the world. Each of these testbeds has its own specialties to support particular categories of experiments, and they also have some features in common. For example, GENI provides nationwide layer 2 connectivity to support experiments which are not compatible with IP and today's Internet. Another example is DeterLab which is isolated from the public Internet which allows for security experiments that could not safely be run on a testbed connected to the Internet. In addition, commercially available cloud services such as Amazon Web Services (AWS) [2] are suitable for certain types of experiments.

You should select the testbed which will best support your experiment and satisfies the answers to the questions listed above.

Learning how to use the testbed helps reduce the time spent on debugging problems that do not belong to the experiment itself. Each testbed has a number of tutorials that teach how to use the tools provided by the testbed. Once you have selected a testbed for your experiment, it is worth spending a few hours trying some examples and getting familiar with the testbed.

3.3 Ready to Proceed?

At this point you should have at a minimum: a picture and accompanying text explaining "what" you want to do; a second picture and a procedure (including metrics and parameters to vary) explaining "how"; and have identified a testbed that is capable of supporting your experiment.

Ask a colleague to review these artifacts and incorporate their feedback. Armed with these artifacts, you are ready to proceed.

3.4 Plan for the Case Study

Fig. 1 presents how we plan and conduct the experiments for the case study. Since our goal is to measure OSPF convergence time during recovery from a link or node failure in a wide area virtual network, we need to create a virtual network that is composed of several virtual routers that are geographically distributed. There are at least two routing software suites (e.g., XORP and Quagga) which can be installed on generic Linux virtual machines, and we are able to configure the OSPF routing protocol with either of them.

Next we need to decide on a testbed. Because the software routers can be installed on generic virtual machines, a variety of geographically distributed testbeds could theoretically be used. Previously a similar experiment was deployed on the GpENI Virtual Network Infrastructure (GpENI-VINI) testbed [22], and the star-like GpENI substrate network topology impacted the round trip time in the virtual network. For the purposes of the case study, this feature does not matter. But the case study was done as part of a larger experiment to evaluate a selection algorithm to replace failed virtual routers in virtual networks. For the larger experiment the substrate topology has a big impact on the results.
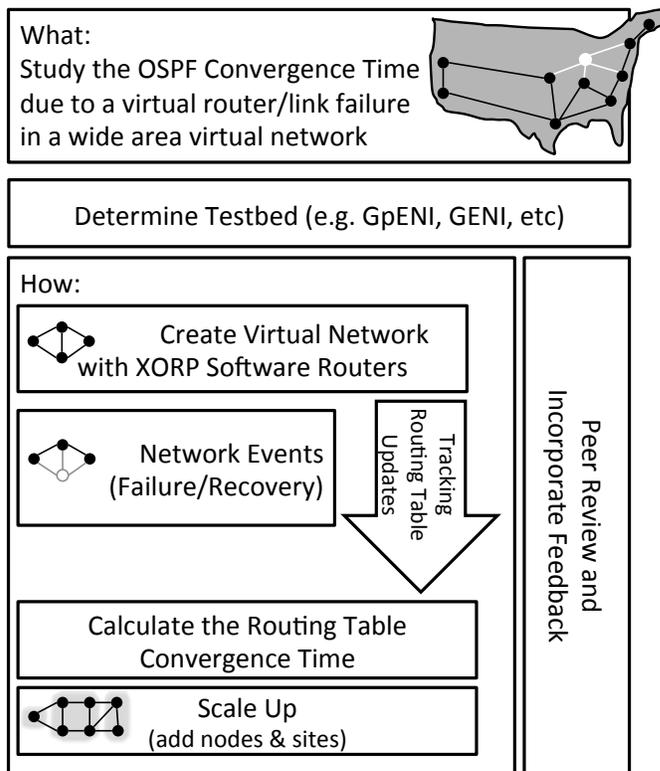

Figure 1: Plan for Case Study:
Capture OSPF convergence time during network events

As a result we chose GENI [9] which has a a realistic partial mesh substrate topology.

Next we need to plan the experiment deployment systematically. As shown in Fig. 1, we first create a trivial four-node virtual network with XORP software routers running the OSPF routing protocol. Second, we generate a failure by bringing down a virtual link or a virtual router and recovering the failure after some time. During this time, we periodically track the OSPF routing table updates on each XORP router recording the timestamp of when there are changes. When there are no more updates, we calculate the routing table convergence time. When we have tested the trivial setup without any problems, we can scale the experiment to a larger virtual network.

4. AUTOMATE USING BEST PRACTICES

Running a computer science or networking experiment is challenging because it requires three distinct skills: applying the *scientific method*, *software engineering*, and *system administration*.

Best practices for each of these skills should be applied when running an experiment. Luckily, these best practices have much in common and often focus on automation and testing. In the context of experimentation, this means automating the configuration, execution, and analysis of the experiment.

In particular, two good habits to keep in mind while executing and analyzing an experiment are: (1) always make and test one small change at a time; (2) save your work using version control. By doing these two things, you will always know your *last working configuration*.

4.1 What can go wrong?

To understand why these habits are so important, we first need to understand what can go wrong. The short answer is that human error makes it all but impossible to manually configure non-trivial topologies. Studies [25, 19] report that up to 50% of network outages are due to human configuration errors.

Imagine we know exactly what needs to be configured on a single node.[1] Let's assume that one configuration step (e.g. configuring IP addresses, routing tables, software setup) can be made without an error with a probability of 90% and that it is independent and identically distributed (i.i.d.)[2]. If there are five separate steps, then there is only a 59% chance that this single node will be configured correctly ($.9^5 = 0.59$) and only a 35% chance if there are 10 steps. The probability of success decreases exponentially as we increase the complexity of the setup. Manually configuring a trivial topology of even 3 or 5 nodes in this way can be very challenging.

If we automate all 5 or 10 steps with a single script, then we have a 90% chance of successfully configuring a single node as there is only one step a human has to do (i.e. execute the script). Likewise there is a 72% chance of bringing up a 3 node topology successfully ($.9^3 = 0.72$) on the first try.

However if we automate the execution of the script (in GENI this is called an *install script*) and we also systematically describe the configuration of the whole topology (in GENI these are *resource specifications* or *RSpecs*), we minimize our chance to make an error by having to execute only one manual step.

In summary, automation is key for reducing the chance of human error and makes it possible to bring up non-trivial topologies which would be all but impossible to bring up manually.

4.2 Good Habits

Fig. 2 shows how to make a series of changes to automate an experiment. Always start with a trivial setup. An important computer programming principle that is also very appropriate here is to *change one thing at a time* and then test it so you know if the change broke your setup (see Table 1 for examples of possible types of changes). Then, because of the high risk of human error, *automate* the change and test it. Finally always *save your work* using a version control system such as `cvs`, `svn`, or `git` so you can revert your changes if you make a mistake later. Every part of your experiment should be saved including software, configuration files, scripts, notes, collected data, analysis scripts and results, etc.

This *test-automate-test-save* cycle should be applied to both experiment deployment and data analysis. Working in this way allows you to always know the *last working configuration*. Experienced developers and system administrators work this way out of habit. As a result they are often able to debug their own problems and make clear and helpful bug reports when they can not resolve the issue themselves (see Section 4.3). This is a skill that is well worth developing.

Note that automation does not have to be limited to simple scripts. Configuration management tools such as `puppet`

and `chef` are widely used to automate the configuration and administration of large systems; they can also be used to create repeatable experiments as shown in [17] which uses `chef` to do so on Amazon's Web Services. Moreover, tools specific to testbeds have been developed; for example, OMF-F [29] is a tool for deploying networking experiments in federated domains, while LabWiki [18] and GUSH [6] are specifically designed to automate the configuration and orchestration of experiments in specific testbeds. The reader is encouraged to research and find the specific tools that would help in automating the deployed experiment on their chosen testbed.

4.2.1 Applying Good Habits to the Case Study

The case study (see Table 2 and Section 5.1.1) illustrates some important lessons:

1. *Image snapshots make life easier but like any artifact you must know the steps to reproduce it.* The router nodes run an OS image that is just a default Ubuntu image with a standard `XORP` package installed. The steps for creating it are documented in [4]. However, this installation takes approximately 40 minutes. So instead of installing `XORP` on each node at boot time, the `router` nodes use a custom image to substantially decrease the time to instantiate the experiment topology.

2. *Support scalability by writing scripts in a generic, auto-configuring way.* Manually configuring OSPF on a XORP software router involves entering at least 5 different pieces of information for each node. For a simple 4-node setup there are 20 entries that must be configured correctly which is time consuming and challenging to get correct. Instead, the OSPF configuration script determines IP addresses of interfaces and other needed parameters on the fly (by running `ifconfig`) so a single generic OSPF configuration script can be used for each router node without requiring any manual per node configuration.

4.3 Resolving Problems

Despite their best efforts and even if an experimenter follows the best practices described in this paper, something will inevitably go wrong due to bugs, accidental misconfiguration, or scheduled/unscheduled maintenances. There are generally two kinds of problems: those under your control and those not under your control. Make a good faith effort to address issues under your own control first which will resolve your problem quickest, but also do not be afraid to ask for help.

This is a time when knowing your last working configuration (see Section 4.2) will pay off. Your last working configuration gives you the information needed to both find your own solution to the problem and failing that to make a good bug report. Armed with the knowledge of what changed, you can search for a solution to your problem by consulting online documentation.

If you can not solve your own problem, an especially helpful form of bug report is to provide the last working configuration and explain the change that breaks it.[3] In this

---

[1]If the setup is still under development the following description becomes even more grim.

[2]For simplicity, we assume each step to be i.i.d. and approximate the probability of successful configuration with $Pr[\text{success of one step}]^{\text{number of steps}}$

---

[3]An even better form of bug report is to reduce the configuration to the smallest one that works and the smallest change that breaks it.
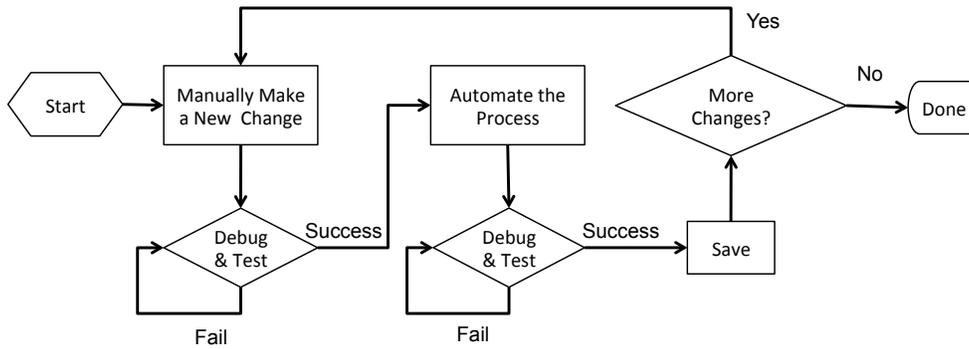
Figure 2: The *test-automate-test-save* cycle.

Table 1: Possible Changes to be Made When Building an Experiment

Type of Changes	Description
Software Installation	Install standard software and packages.
Develop Application	Develop an application running on a node for the experiment.
Configuration	Configure the software or nodes.
Number of Nodes and Links	Scale up the experiment.
Geolocation of Nodes	Build distributed experiments.
Orchestration	Orchestrate the experimental procedure. Vary parameter values.
Instrument	Measure what happens during the experiment.

case, the cause is often identified and resolved quickly. Bug reports that simply state "this complicated setup does not work" can take days of iterating back and forth to merely identify the issue. This is because as shown in Section 4.1, there can be dozens or hundreds of things that could be wrong. Debugging requires eliminating those, often with no context or access to the experiment.

Once you are ready to file a bug report and ask the community for help you should first identify the appropriate mailing list. For example, questions about using a specific software package should be directed to mailing lists for that package while problems with deployment on a specific testbed should be addressed to the operators of the testbed.

A good bug report should include the following information:

- *What you did.* This is a good place to describe the "last known working configuration" and the "change that breaks it".

- *What you expected to happen.* What does "working" look like?

- *What actually happened.* What does "broken" look like? (Include error messages and screenshots where applicable.)

- *Be sure to include the resources and tools you used.* For example, in GENI we always ask experimenters for: the name of their experiment, the geographical location of the resources, the file containing their experiment description, the way they access the testbed, and the tool they are using.

For all of the items be as specific and as thorough as possible. The authors of this paper have never seen a bug report with "too much" information.

5. BUILD SCALABLE EXPERIMENTS

Instead of running a complete experiment on a testbed in one shot, it is better to systematically design and build up your experiment to ensure that each step is correct using the same techniques you would use to write software or configure and manage a system. In this section, we illustrate a recommended generic procedure for systematic experiment design. Following these steps (shown in Fig. 3) will allow you to create repeatable experiments which will scale up to a meaningful size.

5.1 Start small (and scale up later)

A small experimental setup is helpful for testing applications, automation, and debugging errors. For any kind of experiment, always *start by building a trivial setup*. If everything works as desired with the small setup, then scale larger.

Fig. 4 shows examples of four different trivial topologies for different types of experiments. Note that in each of these cases you only need to automate at most two distinct types of nodes. That is, all of the nodes of the same type can be configured using a single image and the same scripts as long as those scripts are written in a generic way so that things like IP addresses and interface names are determined on the fly as described in Section 4.2.1.

In general, when selecting a trivial topology, select at least one node of each different type used in the experiment and create a simple topology among these nodes. Only repeat node types where necessary. If the experiment is eventually going to be deployed in a geographically distributed way, start with all the nodes in one location.

Using this trivial topology, test everything required for your original experimental design, including the four items shown in Figure 3:
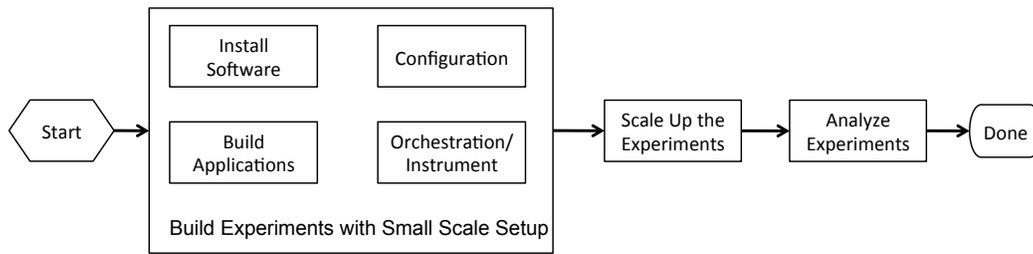
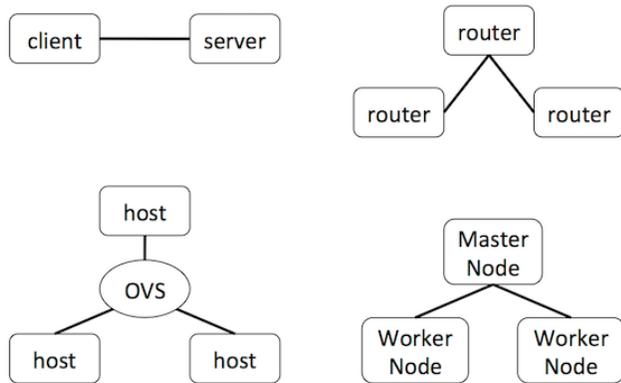Figure 3: Procedures for Building Experiments

Figure 4: Examples of Trivial Setups. A simple client-server architecture only requires two nodes. Studying routing invariably involves multiple nodes. The three router topology shown in the figure could be used as the starting point in an experiment meant to study IP routing since two of the nodes are not neighbors and will not be able to communicate unless IP routing is configured properly. The Open Virtual Switch (OVS) topology would be useful for doing some initial tests of an OpenFlow controller. Finally the master-worker topology represents a trivial Hadoop setup.

- *Install Software.* Different types of nodes, may require different operating systems and different software.

- *Build Applications.* Sometimes you will need to write your own applications for the experiment.

  For example, many GENI experimenters implement new OpenFlow controllers to support new proposed algorithms and functionality.

- *Configuration.* Whether using standard software or a newly designed application, you need to configure the nodes properly.

  For example, for a routing experiment, it is important to configure each software router correctly for a particular routing protocol (e.g. OSPF), and test whether the routing software is working properly on each node. For a Hadoop experiment, you need to configure the master node and worker node and test the communication between the master and workers.

- *Orchestrate/Instrument.* In order to tell what happened in your experiment, you need to *instrument* your experiment so that you can measure what happened. In addition, you need to *orchestrate* the experimental procedure.

The two habits from Section 4.2 (change one thing at a time; save your work) apply to each category above. Most of the heavy lifting of implementing your experiment is done building this small topology. If you automate the scripts in a scalable way, then adding more nodes is merely a matter of instantiating more instances of the same type of nodes.

When you test your trivial setup, be sure to validate the correctness of the behavior of your code and your experiment by doing some things where you know the outcome. For example, be sure to include both negative as well as positive tests. Most people will use ping to test for connectivity between their nodes (a positive test). However, a classic mistake to make on a testbed with a separate control plane is to accidentally use the control plane for connectivity. A reasonable test would be to break a link in the data plane and test that traffic actually stops flowing (a negative test). In general, the idea is to test and debug the behavior of your code and setup with 3 nodes before you try it with 10 or 100 nodes. Small topologies are tractable to debug and when you scale you will be more confident that the problem has to do with the number of nodes and not something fundamental to your code.

Once you have a working small topology, the next step is to *scale up* the experiment to include more nodes and links using the good habits described previously. Each new topology should be tested and saved before moving onto a new one.

Once your topology is of the necessary size, you can also start to vary parameters required for your experiment.

5.1.1 Systematically Creating the Case Study

Table 2 summarizes how the case study was developed on GENI following the best practices and good habits outlined in this paper.
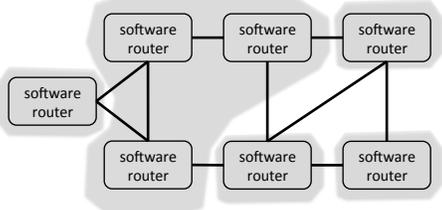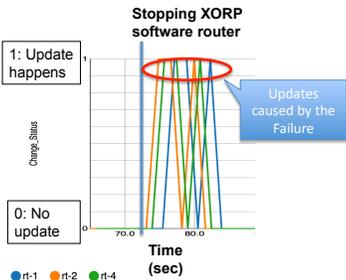
The case study starts small by defining and testing the two types of nodes needed to construct the minimum reasonable topology. This maximizes reuse of code and makes it easy to scale.

The two needed node types are: `router` nodes and `hosts` to act as end points.

Create and test `router` node type. The `router` nodes are all configured using a single OS image and the same set of scripts. In particular, `router` nodes are xen virtual machines running the following:

1. XORP OS image. Snapshot of a plain Ubuntu image

Table 2: Case Study: OSPF convergence time during network events.

Steps	Tools and Techniques Used

1. Create and test `router` node type

Convert a generic virtual machine into a software router by installing chosen routing software (XORP).
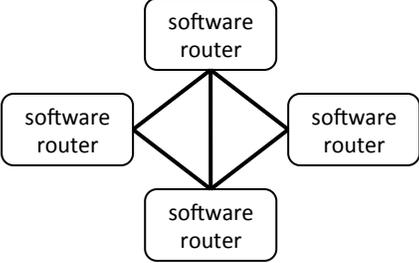
Automate experiment on a trivial scale. Reserve software router nodes in a simple topology, configure routers (manually at first and then automated), and instrument experiment (manually first then automated).

Install software router. Installation of XORP takes approximately 40 minutes. To prevent having to install XORP on each node, took an *image snapshot*. In subsequent steps the *software router* nodes are a xen VM with this XORP image installed.

Reserve smallest reasonable topology. In this case, two virtual routers which are not directly connected allows testing of OSPF functionality; two interfaces per node allows testing of OSPF configuration.

Configure OSPF. First, manually configure OSPF routing in this trivial setup to determine how it is done. Second, write scripts to automate OSPF configuration on routers in any topology and to start the routing software.

Track routing table updates. On each node, periodically capture the routing table and compare the current routing table with the previous snapshot. Output is a csv file for each node where each row is a UNIX timestamp followed by a 0 or 1 depending on whether there was a routing table change.

---

2. Create and test `host` node type
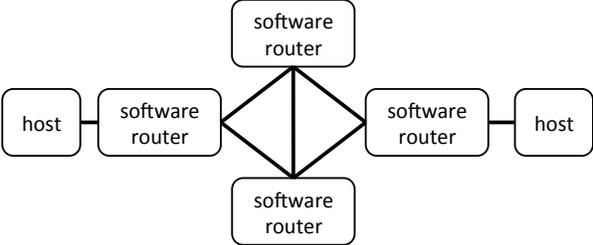
Add endpoints (hosts) to topology.

Run end-to-end traffic and validate experiment setup. Use iperf and ping to send traffic end-to-end. Fail individual software routers or links and measure how long it takes for the route updates to propagate throughout the network. Validate traffic flow by checking that pings and iperf traffic flow before and after node and link failures.

---

3. Scale number of nodes and geographic diversity

Each gray background denotes a different location.

First, scale up one dimension at a time.
- Scale up number of nodes in one geographic location.
- Scale up number of geographic locations.

Second, scale up both number of nodes and number of geographic locations.

---

4. Analyze results and perform repeated runs

Calculate convergence time for each run.

Repeat.

with `XORP` installed from a package.

2. OSPF auto-configuration scripts. Shell and `awk` scripts take the output of `ifconfig` and generate the `XORP OSPF` configuration.

3. Script to record changes to the node's routing table. A shell script polls the routing table and compares each version to the previous, recording a timestamp and whether there is a change in a `csv` file.

Create and test `host` node type. The `hosts` are simply VMs running a default image to allow the sending of end-to-end pings and iperf traffic. So `host` nodes are `xen` virtual machines running:

1. A standard Ubuntu image

2. `ping` comes by default on the image and `iperf` was installed using a package manager.

Scale up number of nodes and geographic diversity. An arbitrary number of these two types of nodes can be put together in arbitrary topologies and the experiment still works.

In order to scale up the topology, the second author of the paper created a custom script called `scaleup` which was written using (and is now distributed with) the `geni-lib` library [8]. `scaleup` parses a configuration file containing a description of the above node types and a terse description of a desired topology (e.g. ring, grid, etc) and outputs the appropriate GENI resource specification (RSpec). These RSpecs (i.e. topologies) can be reserved with any GENI resource reservation tool: omni, GENI Portal, etc.

Moreover `scaleup` supports scaling the size of a topology by allowing the experimenter to easily generate large topologies at a single location or generate topologies that are geographically diverse. The case study utilizes this capability to scale in three different ways: increasing the number of nodes at a single location, expanding to a geographically distributed setup, and a hybrid of the previous two options. **Analyze results and perform repeated runs.** The CSV output from the routing table comparison script on each `router` node can be graphed using LabWiki, a GENI instrumentation and measurement tool, or using the graphing program of your choice.

5.2 Sharing An Experiment

Once you are able to repeat your own experiment, you should have most of the artifacts needed to share your experiment with others. To enable others to repeat your experiment you need to provide access to at least:

- **All experiment artifacts** including: images, topology, installation, orchestration and analysis scripts, data (where possible), and analysis results. You should include documentation about how to access/use these artifacts. Whenever possible use standard file formats (e.g. share your data as a csv file versus the Excel proprietary format) and open source tools. Include the versions of the software used/needed to repeat your experiment.

- **Document how to repeat the experiment.** Two ways of doing that are: (i) a **README** or writeup explaining how to repeat the experiment, (ii) a **video** of running the experiment.

Have a friendly colleague try repeating your experiment using just the items listed above. Revise your documentation as necessary.

Sound scientific research requires that all the above artifacts are made publicly available to the community for verifying, reproducing, corroborating, and extending the obtained results. There are multiple public forums to do this; some popular ways are through publicly available webservers and wikis, or through publicly available repositories like github [5] and bitbucket [3]. Testbeds also have forums for publishing experiments like apt [1], a new tool that enables publishing of experiments based on Emulab or GENI.

5.2.1 Sharing the Case Study

The case study is documented in [4] which includes a `git` repository containing the individual steps to create each experiment artifact shown in Table 2 as well as the actual artifacts and the final multi-site topology.

6. BIBLIOGRAPHY AND RELATED WORK

The problem of repeatability and reproducibility in experimentation has always concerned the scientific community. This problem is even more challenging in computation and networking research [20, 23, 32, 33]. Community testbeds [9, 28, 7, 34, 31] alleviate some of the repeatability problems by providing researchers with a common testing ground. Although these testbeds provide a collaborative environment to advance computer science research, they themselves pose new challenges in their use. Researchers have looked into how to enhance testbeds [26, 11, 12] with tools in order to make them easier to use and provide a more complete infrastructure for experimentation. Some tools, like GUSH [6] and OMF [29, 30] are focused on enabling the automation of testbed experiments, while others facilitate the transition from simulation to experimentation [15, 7] as a means to enable more researchers to use experimentation as a validation method.

To the best of our knowledge there is no work that provides advice on how to overcome basic challenges in using public testbeds and this is the gap that we envision this paper to fill. The reader is encouraged to also turn to the multitude of resources that provide advice about how to systematically design their experiments and build a rigorous experimentation process. Lilja's book [21] gives an introduction to performance analysis, while Jain's book [16] (especially Section 2.2) is a very good resource that provides detailed methodologies. Vern Paxson [27] also provides strategies for collecting sound Internet measurements.

7. CONCLUSION

Community testbeds have been widely deployed over the past decade and their usage for experimental validation in computing and networking research has increased dramatically. Although they solve many problems and facilitate repeatability, they themselves present unique challenges to an uninitiated user. By helping numerous experimenters to deploy their experiment in GENI, and by having deployed several setups ourselves we have made notice of common pitfalls and developed a methodology about designing and building repeatable experiments that scale and can be shared with others.

In this paper we present this methodology and describe best practices about deploying an experiment in a community testbed, along with a simple case study that showcases

how following these guidelines can help in reducing deployment effort and time.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] Adaptable profile-driven testbed. https://aptlab.net/.

[2] Amazon web services website. http://aws.amazon.com.

[3] Bitbucket. https://bitbucket.org/.

[4] Case study archive: Measuring ospf updates. http://groups.geni.net/geni/wiki/PaperOSRMethodology.

[5] Github. https://github.com/.

[6] J. Albrecht and D. Y. Huang. Managing distributed applications using gush. In *Testbeds and Research Infrastructures. Development of Networks and Communities*, pages 401–411. Springer, 2011.

[7] M. P. Barcellos, R. S. Antunes, H. H. Muhammad, and R. S. Munaretti. Beyond network simulators: Fostering novel distributed applications and protocols through extendible design. *J. Netw. Comput. Appl.*, 35(1):328–339, Jan. 2012.

[8] N. Bastin. geni-lib. http://geni-lib.readthedocs.org, 2013-2014.

[9] 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*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds Part I.

[10] J. Chase, L. Grit, D. Irwin, V. Marupadi, P. Shivam, and A. Yumerefendi. Beyond virtual data centers: Toward an open resource control architecture. In *in Selected Papers from the International Conference on the Virtual Computing Initiative (ACM Digital Library), ACM*, 2007.

[11] F. Desprez, G. Fox, E. Jeannot, K. Keahey, M. Kozuch, D. Margery, P. Neyron, L. Nussbaum, C. Perez, O. Richard, W. Smith, G. von Laszewski, and J. Voeckler. Supporting experimental computer science. 03/2012 2012.

[12] J. Duerig and et al. Automatic ip address assignment on network topologies, 2006.

[13] S. Fdida, J. Wilander, T. Friedman, A. Gavras, L. Navarro, M. Boniface, S. MacKeith, S. Avéssta, and M. Potts. FIRE Roadmap Report 1 Part II, Future Internet Research and Experimentation (FIRE), 2011. http://www.ict-fire.eu/home.html.

[14] D. G. Feitelson. From repeatability to reproducibility and corroboration. *Operating Systems Review*, January 2015. Special Issue on Repeatability and Sharing of Experimental Artifacts.

[15] M. Fernandez, S. Wahle, and T. Magedanz. A new approach to ngn evaluation integrating simulation and testbed methodology. In *Proceedings of the The Eleventh International Conference on Networks (ICN12)*, 2012.

[16] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, Inc., 1991.

[17] J. Q. Jonathan Klinginsmith. Using cloud computing for scalable, reproducible experimentation. Technical report, School of Informatics and Computing Indiana University, Indiana University, Bloomington, Indiana, August 2012.

[18] G. Jourjon, T. Rakotoarivelo, C. Dwertmann, and M. Ott. Labwiki: An executable paper platform for experiment-based research. *Procedia Computer Science*, 4(0):697 – 706, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[19] L. Keller, P. Upadhyaya, and G. Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 157–166, June 2008.

[20] R. LeVeqije, I. Mitchell, and V. Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science Engineering*, 14(4):13–17, July 2012.

[21] D. Lilja. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press, 2000.

[22] X. Liu, P. Juluri, and D. Medhi. An experimental study on dynamic network reconfiguration in a virtualized network environment using autonomic management. In F. D. Turck, Y. Diao, C. S. Hong, D. Medhi, and R. Sadre, editors, *IM*, pages 616–622. IEEE, 2013.

[23] J. P. Mesirov. Computer science. accessible reproducible research. *Science (New York, N.Y.)*, 2010.

[24] J. Mirkovic, T. Benzel, T. Faber, R. Braden, J. Wroclawski, and S. Schwab. The deter project: Advancing the science of cyber security experimentation and test. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 1–7, Nov 2010.

[25] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[26] K. Park and V. S. Pai. Comon: A mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1):65–74, Jan. 2006.

[27] V. Paxson. Strategies for sound internet measurement. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 263–271, New York, NY, USA, 2004. ACM.

[28] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, Jan. 2003.

[29] T. Rakotoarivelo, G. Jourjon, and M. Ott. Designing and orchestrating reproducible experiments on federated networking testbeds. *Computer Networks*, 63(0):173 – 187, 2014. Special issue on Future Internet Testbeds Part II.

[30] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar. Omf: A control and management framework for networking testbeds. *SIGOPS Oper. Syst. Rev.*, 43(4):54–59, Jan. 2010.

[31] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1664–1669 Vol. 3, March 2005.

[32] V. Stodden, F. Leisch, and R. D. Peng, editors. *Implementing Reproducible Research*. The R Series. Chapman and Hall/CRC, Apr. 2014.

[33] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9 – 18, 1995.

[34] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, Dec. 2002.