Automatic Online Validation of Network Configuration in the Emulab Network Testbed

David S. Anderson[†] Mike Hibler Leigh Stoller Tim Stack Jay Lepreau
University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, UT 84112–9205

{davidand, mike, stoller, stack, lepreau}@cs.utah.edu www.emulab.net

***Abstract*— Emulab is a large-scale, remotely-accessible network and distributed systems testbed used by over a thousand researchers around the world. In Emulab, users create "experiments" composed of arbitrarily interconnected groups of dedicated machines that are automatically configured according to user specifications. In the last year alone, users have run over 18,000 such experiments, expecting consistent and correct behavior in the face of the ever-evolving 500,000 line code base and 3,000 discrete hardware components that comprise Emulab. We have found normal testing to be insufficient to meet these expectations, and have therefore provided continuous, automatic validation. This paper describes *Linktest*, an integral part of our validation framework that is responsible for end-to-end validation during the configuration of every experiment. Developing and deploying such a validation approach faces numerous challenges, including the need for a code path entirely independent of the rest of the Emulab software. We describe our system's motivation, its design and implementation, and our experience.**

I. Introduction

This paper describes an end-to-end validation system for experiment configuration in a public scientific testbed, Emulab [1], [2]. Emulab provides an experimentation facility that allows researchers to configure and access networks composed of a dozen classes of networked devices, including emulated, simulated, wireless, and wide-area nodes and links. Researchers access these resources by specifying a virtual topology via a script in an extended syntax of the popular network simulator NS [3]. Emulab uses this specification to automatically configure a corresponding physical topology using, for example, cluster PCs and switched Ethernet segments. An Emulab *experiment* consists of a set of nodes, the operating system and software running on those nodes, links and LANs connecting the nodes, traffic shaping attributes of those links and LANs, network routing information, and runtime dynamics such as traffic generation.

Emulab's mission of providing a public evaluation platform for arbitrary workloads places a premium on accuracy, precision, and generality. Further, Emulab is itself an experimental facility that must support arbitrary workloads even as it undergoes radical changes. We of course perform regression testing, but our experience shows that even a large suite of regression tests can miss crucial errors. The feature interaction of such a large system as ours can, and has, led to bugs that can only be found by checking *every* experiment at run time. Our automated tests after every experiment "swapin"—the instantiation of a virtual topology onto physical hardware—help Emulab achieve the goal of valid emulation by ensuring the physical topology correctly instantiates the virtual topology submitted by the user. They allow Emulab staff to identify and troubleshoot problems that would otherwise be too time-consuming to manually identify in a large, highly connected experiment. By using a completely separate code path for validation testing, sharing only the experiment specification, we ensure that bugs in the testbed experiment setup infrastructure do not cause matching bugs in the test suite. Validation tests use end-to-end tests to verify connectivity and traffic shaping from actual link behavior.

We have the following requirements for a suite of validation tests for Emulab experiment configuration:

No Experiment Reconfiguration. Validation tests should not change the configuration of the experiment to facilitate testing, since their purpose is to verify everything works as the user specified. Otherwise, the process of undoing any changes introduced by the tests themselves, to return to "production mode," could introduce *new* errors into the configuration. This implies, for example, that since traffic shaping parameters include loss, tests to determine node connectivity, bandwidth, and latency must be able to tolerate a reasonable number of lost packets.

Alternate Code Path. Validation tests should use the virtual topology as specified in the primary source—the NS script—rather than the representations created in the Emulab database. In this way they avoid relying on the testbed parser and intermediate representation.

End-to-End Validation. Validation tests should run only on experiment nodes that the NS script specifies in the virtual topology. The tests should not need to take advantage of, or even be aware of, underlying Emulab mechanisms such as interposed traffic shaping nodes or the traffic monitoring capabilities of the switch infrastructure.

Support Arbitrary Configurations. Emulab experiment configurations are arbitrary and unpredictable. Validation tests should not be limited in their ability to verify extreme network

† David S. Anderson now at UCSD; work done at University of Utah. Largely sponsored by NSF grants CNS–0335296, CNS–0205702, and EIA–0321350.

configurations that may be required by researchers.

Portability. Since the operating system of a node is part of the experiment specification, and because we cannot reconfigure the experiment, validation tests should be portable to a variety of operating systems. These particularly include FreeBSD, Linux, and Windows XP, which are commonly used in the Emulab environment.

Speed. Emulab experiments take only a few minutes to swap in, regardless of the number of nodes or links in an experiment. Validation tests add overhead to the swapin time, which can adversely impact the ease of use and overall throughput of Emulab, if the startup time becomes unreasonably long.

Scalability. Since Emulab currently has over 350 physical PC-class nodes, each of which may contain tens of virtual nodes, scalability is a crucial factor for validation tests. Besides running quickly, individual tests must run in parallel to support scaling for all but the most trivial experiments.

Non-Intrusiveness. Validation tests should avoid overuse of shared resources that might affect other experiments. Practically speaking, this means that the nodes within an experiment should set up, coordinate, and perform as much of the work as possible instead of relying on a central shared Emulab server.

In the rest of this paper we explore our technical approach to these goals and to what degree we met them. Section II briefly describes the specification and instantiation of Emulab experiments. Section III describes the Linktest design and implementation. Section IV presents measurements of the speed, scaling, and accuracy of Linktest. Section V discusses some of the current limitations and what we might do in the future. Section VI describes related work on testbed validation, while Section VII concludes.

II. Emulab Experiments

As mentioned in the previous section, an Emulab experiment encapsulates a virtual topology and its associated characteristics including the set of nodes, the software running on the nodes, the network links that connect nodes, and the traffic shaping attributes of those links. Figure 1 shows a simple experiment with six nodes, four end nodes connected via two routers, all running different operating systems. The interconnecting links have differing characteristics, as can be seen in Figure 2, which shows the NS specification for the experiment. Note that though the logical topology has two types of nodes, "machines" and "routers," to Emulab they are both just nodes and are both mapped to physical PCs. Emulab will enable IP forwarding and set up appropriate IP routing if desired, but it is up to the experimenter to further differentiate the "roles" of the nodes.

In the remainder of this section we briefly describe experiment setup, from the configuration file through the instantiation on physical hardware, to illustrate the complexity of the process and the many points at which errors affecting network configuration could occur.

Parsing. The first stage of experiment setup is to parse the configuration file and build a representation of the experiment in the Emulab database. Emulab's parser is a version

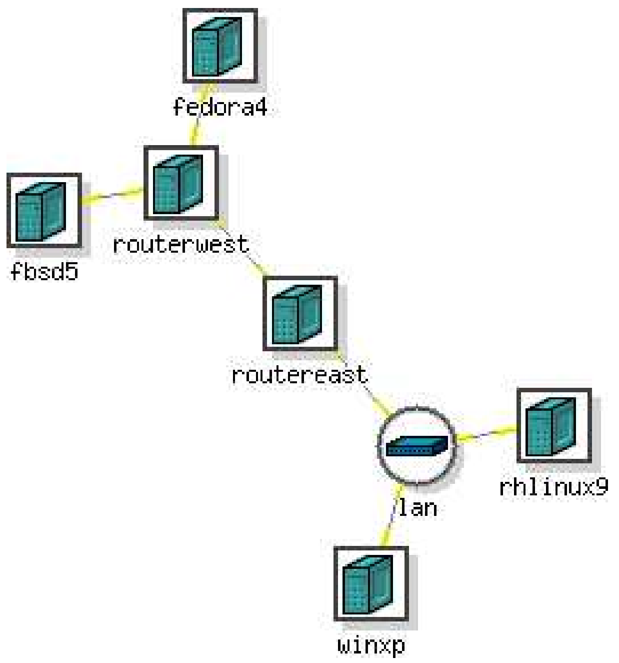

Fig. 1. An example experiment topology with four "machines" and two "routers" connected with links and a LAN.

```
set ns [new Simulator]
source tb_compat.tcl

# Nodes and their OSes: FreeBSD, Linux and Windows XP
set fbsd5 [$ns node]
tb-set-node-os $fbsd5 FBSD54-STD
set fedora4 [$ns node]
tb-set-node-os $fedora4 FC4-STD
set winxp [$ns node]
tb-set-node-os $winxp WINXP-UPDATE
set rhlinux9 [$ns node]
tb-set-node-os $rhlinux9 RHL90-STD

# Routers
set routerwest [$ns node]
set routereast [$ns node]

# Links and Lans
set link1 [$ns duplex-link $fbsd5 $routerwest 10Mb 5ms DropTail]
tb-set-link-loss $link1 0.05
set link2 \
    [$ns duplex-link $fedora4 $routerwest 10Mb 5ms DropTail]
tb-set-link-loss $link2 0.05
set lan [$ns make-lan "$winxp $rhlinux9 $routereast" 100Mb 0ms]
set trunk \
    [$ns duplex-link $routereast $routerwest 50Mb 50ms DropTail]

# Pre-compute routes
$ns rtproto Static

$ns run
```

Fig. 2. Emulab-extended NS script used to produce Figure 1.

of the standard NS parser, extended to recognize Emulab-specific constructs. Where possible, Emulab uses existing NS syntax, enabling some topologies to be emulated on Emulab or simulated with NS using the same configuration. New syntax has been introduced to enable Emulab-only features such as specifying what software (OS and applications) runs on a node. As the parser executes the the NS script, it populates the database with information about the virtual topology.
Possible errors at this level include problems in the Emulab parser that result in mis-parsing link specifications or storing topology information incorrectly in the database. One bug was latent for a year and did not become manifest unless users named their nodes with a particular string. We had not fully cleaned NS's namespace, leading to silent collision. By implementing an independent parser, Linktest detects these errors.

Physical Resource Assignment. The global resource allocation phase binds elements of the virtual topology described in the database with physical hardware when the experiment is swapped in. The Emulab resource mapper [4] considers the desired characteristics of both nodes and links and strives to make the most efficient (in terms of global resource management) mapping possible given the available physical resources. As part of the mapping process, additional traffic shaping nodes may be integrated into the physical topology to handle link shaping. Once the mapping is made, the assigned physical topology information is associated with the experiment in the database. Emulab then uses this database information to set up switch VLANs for connected interfaces on the assigned PCs and to load the desired OS disk images on those PCs [5].
Errors in the database description of physical resources are the primary cause of problems at this level. Such errors could cause the wrong switch ports to be configured. Again, since Linktest does not rely on database state, but rather makes its own independent observations, it detects these problems.

Node Self-Configuration. Once an experiment has been assigned physical resources in the database, the physical PCs associated with the experiment, both user-specified and the additional traffic shaping nodes, are configured. In Emulab, all machines use a *self-configuration* process, where a small set of scripts on each node run at boot time, obtaining information from the central database and using that information to customize the node. This dynamic, boot-time configuration is an alternative to static pre-configuration of the OS disk image for each node, and allows "generic" OS disk images to be used.
There are many potential sources of error at this stage. Subtle or unexpected problems with the hardware could result in uncaught errors allowing experiment setup to succeed, but without being properly configured. For example, an interface might come up in half-duplex mode where the switch believes it to be full-duplex, resulting in dropped packets in one direction. Likewise, unanticipated failures in the Emulab setup scripts could wreak havoc. A real example of this occurred recently in the Emulab Linux image, where a script was changed, seemingly harmlessly, to fully specify the path of the command that initialized on-node traffic shaping. However, this caused the script to run the wrong version of the command, which did not support the necessary options and exited without a proper error status. The result was that the node setup succeeded, but no traffic shaping was being performed. Finally, different versions of the same OS can exhibit subtle differences. For example, at some point, the Linux program which dynamically loads and initializes device drivers was rewritten and began using a differently named configuration file. The result was that interface-specific options we had been setting in the old configuration file were silently ignored, causing changes in the latency of packets traversing the interface.

To summarize, link errors in Emulab are caused by both hardware failures and misconfiguration due to software errors. In the case of hardware failures, errors that have been caught include malfunctioning network adapters and broken patch panel ports. These errors tend to be all-or-nothing and fail in predictable patterns. In the case of software misconfiguration, errors are somewhat more subtle to detect due to the coarse timing granularity of traffic shaping. Emulab delay nodes induce a processing delay of approximately 1 ms while processing packets, even though the delay processing is handled in-kernel. Due to this granularity, Linktest focuses on detecting unambiguous link errors.

III. Design and Implementation

This section describes the Linktest test suite for validating experiment topology. Linktest currently includes tests to verify LAN and link connectivity, latency, loss, and bandwidth as well as Emulab-configured IP routing. Linktest is implemented in approximately 3,500 lines of C and Perl code, well under 1% of the 500,000 lines of code in Emulab.

Before describing the implementation and operation of Linktest, we first cover some basic characteristics of the tests and how they fulfill the requirements set forth in Section I.

Speed and Scalability. All individual tests run as quickly as possible while still producing a statistically significant result. This involves running tests for different lengths of time or with a different number of packets, depending on the bandwidth or loss rate of a link. Where possible multiple characteristics are inferred from the same test. To meet our scalability requirements for large, heavily interconnected topologies, tests are performed in parallel to the extent possible without affecting the results due to resource over-consumption. This process is described in detail later.

Portability. To meet our portability requirement, we choose IPv4 as the protocol for all tests, as it has the most widely implemented API (sockets) and a large number of existing measurement tools are built on top of it. All Linktest tests use existing, well-known measurement tools.

No Experiment Reconfiguration. The Linktest tests are designed to produce valid results for links shaped in multiple dimensions; for example, accurately measuring bandwidth on a lossy link. Rather than changing traffic shaping (removing loss in this example), Linktest bases all tests on UDP datagrams rather than TCP streams. This approach comes at a cost of accuracy corresponding to the loss, but eliminates the risk of introducing new errors.

Non-Intrusiveness. The Emulab requirement of experiment isolation provides much of the non-intrusiveness desired. Allocated nodes are dedicated to the experiment, and thus the full resources of the node are available for Linktest. This permits us to run the tests at elevated priority to avoid interference from other OS activity. Emulab conservatively allocates bandwidth on the experimental network switching infrastructure and isolates experiments from each other with switch VLANs, enabling Linktest tests to run "flat out" without affecting other experiments. Linktest avoids use of shared servers whenever possible. In addition to the tests themselves, the bulk of test setup and synchronization is done on the nodes. This includes computing test schedules, determining reachability for routing tests, and barrier synchronization.

Alternate Code Path. The configuration of the various Linktest tests is derived directly from the NS description of an experiment as described later. This includes both the per-link characteristics used in the individual link tests and the node reachability data used to test IP routing.

End-to-End Validation. Linktest tests run only on the nodes specified in the NS specification and, as mentioned, are variants of common UDP/IP applications running on FreeBSD, Linux, or Windows. Thus they have no knowledge of how Emulab implements shaped links. In fact, Emulab support two methods for shaping links, and the same tests are used for both.

A. The Alternate Parser Path

The first stage of a Linktest run happens whenever an experiment specification is parsed; i.e., during experiment creation or modification. The NS topology description is first run through the normal Emulab NS parser, which populates the database with virtual topology information. This topology information is later extracted from the database and passed to nodes during the self-configuration step, to configure node interfaces and traffic shaping and to calculate IP routing information. The database information is also used to construct a *topology map*, a file containing a one-line description for every node and link in the experiment.

Linktest's alternate parser is also run at parse time, taking the original NS specification and producing a topology map directly. The two maps are then compared, and if they are not identical, an error is generated. It is this topology map that is used on each node to drive the Linktest validation tests described below.

B. Linktest Tests

The Linktest suite of tests is triggered automatically at the end of experiment setup, after nodes have finished their self-configuration but before they are made available to the user or begin executing user-supplied scripts. Each node uses the topology map generated by the Emulab parser to drive a set of tests on all links connected to the node, or in the case of routing, to determine which nodes are reachable from the node.

Linktest mostly uses bidirectional tests rather than one-way tests to reduce the complexity of validation. Using one-way tests requires individual node pairs to use barrier synchronization to properly coordinate source and sink processes. In the case of delay, one-way testing is inherently problematic because of clock skew between nodes. A bidirectional testing approach does have tradeoffs, the most prominent being the use of round-trip time to measure delay on links that have asymmetric delay. This tradeoff may be eliminated by future work.

Following is a description of the individual tests, given in the order in which they run.

LAN and Link Connectivity. Linktest uses `ping` to verify direct link connectivity for both point-to-point links and LANs. For each link or LAN associated with a node, the node attempts to "ping" the other member(s). The test passes if the node receives at least one reply from each. To ensure the test can run on very slow or lossy links, Linktest sends 10 packets at a 200 ms interval. Sending 10 packets accommodates a high degree of loss. For example, with a relatively high loss rate of 10%, loss may be modeled as a binomial random variable with n=10, p=0.1. The probability of receiving at least one reply is greater than 0.999. To verify that the other end is directly connected, rather than being reached through multiple hops, the time-to-live (TTL) of the `ping` packet is set to one.

Latency. Linktest uses the round-trip time (RTT) reported by `ping` operations in the connectivity test to verify link latency. The advantages of using RTT are that it is straightforward, fast, and detects common link errors. The disadvantage is that RTT cannot accurately detect problems with links that have asymmetric latencies. Additionally, as a consequence of "piggy-backing" on the connectivity test, the latency may be determined by only a single packet on links with extremely high-loss rates (e.g. 50%), since a single packet is all that is required for a successful connectivity test.
The original Linktest latency test was designed with asymmetric links in mind. The test measured one-way delay using `rude` and `crude` [6], a UDP traffic generator and collector, and corrected timestamps using the Network Time Protocol (NTP) as described in [7]. However, the one-way delay measurements proved to be imprecise due to excessive clock skew on nodes. We intend to revisit using one-way delay for latency measurements in the future.

IP Routing. Linktest uses a separate `ping` pass to verify the setup of IP routing, if Emulab-provided routing is indicated by the NS script. A reachability graph is computed from the topology map on each node, and all reachable nodes that are not directly connected are sent `ping` packets as in the connectivity test. As the number of reachable nodes is generally much higher than the number of directly connected nodes, and thus the number of `ping` operations required is much higher, multiple `ping` operations are issued simultaneously from each node. This is in contrast to the inter-node parallelism of the other tests described in Section III-C.
One minor complication with route testing is that Emulab offers an option for "Session" routing, which provides dynamic

route calculation using a routing daemon on each node. Even with a fixed topology and unchanging link characteristics, it will take time for the routes to converge with this option. We currently handle this in an ad-hoc fashion by waiting extra time before beginning the routing test and by doing a single retry of the test if any route `ping` operations fail.

Loss. Linktest uses `rude` and `crude` to send a one-way UDP packet stream over each link and count the number of received packets. For non-zero loss rates, the number of packets sent is inversely proportional to the loss rate of the link; the higher the loss rate, the fewer packets are sent. To keep test runs short (currently fixed at four seconds), accommodate slow links, and still produce a link loss estimate with a high degree of confidence, we only run the loss test on links that fall within acceptable bandwidth and loss-rate combinations. As an example, for bandwidths less than 1 Mb, the loss rate must be sufficiently high that we would see errors by sending no more than 100 packets per second; otherwise the test is not run. For links with a specified loss rate of zero, 100 packets per second are sent.

Bandwidth. Linktest uses `iperf` [8] to measure "bandwidth." More precisely, we are measuring *link capacity*, but we will continue to refer to it as bandwidth for this discussion. `iperf` is run for a fixed time at a send rate 10% above the expected bandwidth of the link. `iperf` is run in "tradeoff" mode, where the test first sends packets in one direction, then the two sides switch roles and the test sends packets in the other direction. The send rate is based on the maximum expected bandwidth of the two directions. Since TCP will "adapt" its bandwidth to link characteristics, we use UDP packets instead to assure that we can calculate an accurate bandwidth value in the face of link loss and high latency. Ethernet MTU size packets are used to ensure the lowest packet rate needed to achieve the target bandwidth. The `iperf` test reports received bandwidth for both directions. The reported values are adjusted to compensate for packet header overhead not considered in `iperf`, further adjusted for the loss rate, and then compared with the expected values for both directions of the link.

Link attributes have the potential to affect bandwidth validation, as is further detailed in Section IV. Because Linktest uses a UDP capacity test to measure bandwidth, increased loss results in a roughly linear decrease in measured bandwidth. High bandwidth exposes the presence of delay nodes through a 1% increase in measurement error.

The final result of the bandwidth test may vary within acceptance tolerances of 1.5% greater than or 3% less than expected bandwidth. The tolerances are based on the 99% confidence interval of prior bandwidth tests on known-good links, plus a 1% margin to compensate for processing delay. The tolerances are not strictly end-to-end in that they are wider than would be needed in the absence of traffic shaping overhead. However, the tolerances are necessary in practice to accommodate the overhead and avoid false positives from small variations in measurement results.

C. Parallelism and synchronization.

Linktest takes advantage of Emulab's conservative resource allocation to extract a high degree of parallelism from the test suite. At a minimum, for all pair-wise link tests, we can simultaneously run tests on all non-overlapping node pairs, as we know that nodes are dedicated to running the tests and that the provisioned switch infrastructure can handle a full traffic load on all links. We further increase the potential parallelism by running some tests in both directions on a link at once.

The primary restriction is that only one type of test is performed at a time on a link. This allows for early termination of the test suite in many cases. For example, if the connectivity test fails, there is no point in performing the other tests.

The algorithm for computing test schedules is handled completely by the nodes in the experiment, and is fully distributed with the exception of a simple barrier synchronization protocol which uses one of the nodes as a server. Barrier synchronization occurs between each type of test (latency, loss, etc.) and within each test type at "run" boundaries. During a run, nodes simultaneously run tests over the maximal set of links that may be tested at once subject to the restriction of one test type per link.

The workload containing links to be tested is computed simultaneously on each node. An entry in the workload contains a link's attributes, including source, destination and traffic shaping properties. Since each node starts with the same topology map and uses the same algorithm, they all produce the same workload. This computation could be done once on one machine and the result distributed to all others, but the calculation is simple and CPU cycles are not a concern.

Once the workload has been calculated and all nodes have resynchronized at the barrier, each node looks at all workload entries to see whether it has anything to do. If the node matches source or destination to its own identifier, it starts the appropriate test application and begins testing the link in concert with the node at the other side of the link. Otherwise, it simply waits out the run. After all nodes complete their tests, the run has ended. When all runs for all test types are completed, Linktest is complete.

To make this process more concrete, consider an example run that consists of the link description:

```
routereast routerwest 50000000 0.0500 0.000000 ...
routerwest routereast 50000000 0.0500 0.000000 ...
```

from the topology in Figure 1. Since links can be asymmetric with respect to their characteristics, there is a line describing each direction. This example describes a symmetric link between the two router nodes with 50 Mb bandwidth, 50 ms latency (0.0500 seconds), and no loss.

For the combined connectivity and latency test, *routereast* would see that it is the source (listed first) in the first line and initiate a `ping` to *routerwest*. As `ping` is a round-trip test, handled by the receiver in the OS kernel, there is no need to either start an explicit receiver or initiate an explicit return `ping` on *routerwest*. Thus the test harness running simultaneously on *routerwest* will read the lines in the same order, see that it is *not* the source on the first line, and deduce that it has nothing to do but wait for this run to finish.

For the loss test, which uses `rude` and `crude`, the behavior of the two nodes is more symmetric. Since loss is a one-way test, each node initiates a `rude` process for the line on which it is the source and a `crude` process for the line on which it is the destination. Since `crude` will accept and process packets from a `rude` process on any interface, the test actions can be simplified. Instead of starting a `crude` process every time it is needed for a link, a single `crude` process is started on each node at the beginning of the loss test. It runs for the duration of the test, handling the destination actions for all links.

Bandwidth testing with `iperf` is similar, requiring explicit sender and receiver processes. At the beginning of the test, an `iperf` server is started on each node to handle cases where the node is the destination. A minor difference on the sender side (when a node is the source of a link) is that `iperf` is run in tradeoff mode, so that the single invocation of `iperf` will first send traffic one direction to the server, then the server will automatically send a stream back in the other direction. The consequence of this is that, just like `ping` tests, we only start the `iperf` sender on *routereast* in the above example.

As noted earlier, parallelism in the routing test is handled differently. For each node, the topology map is used to construct a "reachable node" list, which is potentially much larger than the list of directly connected nodes used for the other tests. As a concession to the increased number of `ping` operations required to test the route to each node on this list, and due to the low resource requirements of `ping`, we originate up to 10 simultaneous `ping` operations from each node to the nodes on its list. When those 10 have have completed or timed out, 10 more are started. This process continues until all nodes on the reachable node list have been tried.

IV. Measurements

Two lines of measurement are of interest to Linktest performance. With microbenchmarks we show, for individual per-link tests, the accuracy of each test. With macrobenchmarks we document, for the overall execution, how well Linktest scales as the number of nodes and their degree of connectivity increase.

A. Microbenchmarks

In the case of reachability (connectivity and routing), loss, and latency tests, the operation of Linktest is simple. Summarizing from previous sections:

Reachability. The test succeeds if a single packet returns from the target host. This test is straightforward whether the reachability under test concerns a directly connected host or a host several hops away. It is also insensitive to varying combinations of latency and bandwidth. Sample sizes are high enough that the probability of all packets being lost is negligible.

Loss. The test succeeds if the number of lost packets is within the 99% confidence interval for the sample size and expected loss rate. Random number generators in the traffic shaping subsystem reliably produce a uniform distribution of lost packets.

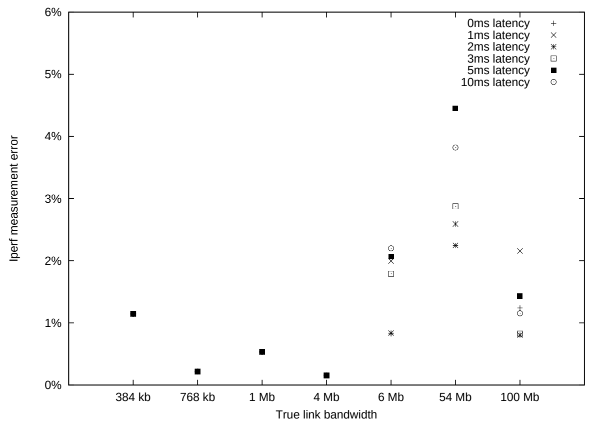

Fig. 3. Linktest measurement error for selected bandwidth and latency combinations.

Latency. The test succeeds if the latency of the round trip time is within 0.5 ms on the low end, or 3.5 ms on the high end of the expected value from the sum of link latencies. A somewhat coarse upper bound is used due to the 1 ms baseline granularity of traffic shaping nodes.

Bandwidth. `iperf` streams are used to infer bandwidth. This is the most complex test in Linktest because as a UDP-based timed test, it is sensitive both to loss and latency traffic shaping. The remainder of this section examines the accuracy of `iperf` for bandwidth validation on Emulab.

For our measurements, a driver script repeatedly executed a one-way `iperf` test over a variety of common bandwidth, latency, and loss settings within an experiment. The driver script used an Emulab command to dynamically set the bandwidth, latency, and loss for a link prior to the beginning of each measurement. Measurement error was calculated as the ratio of the difference between the user-requested bandwidth and the bandwidth reported by `iperf` over the user-requested bandwidth.

Figure 3 shows the results of the `iperf` microbenchmark with loss held at zero, for all tested values of latency. Measurement error stayed under 5% in all cases, and under the 3% bandwidth tolerance except in the case of 54 Mb bandwidth and 5 ms and 10 ms latency. Running `iperf` with these settings consistently returned measurements with a relatively larger error than other datapoints. The 99% confidence interval for bandwidth measured at 54 Mb and 5 ms latency is from 50.79 Mb to 52.39 Mb—at best a 2.9% deviation from the user-requested bandwidth. This particular setting will likely result in frequent false-positives and needs to investigated further.

Figure 4 shows the results of the `iperf` microbenchmark with latency held at zero, for all tested values of loss. Measurement error again stayed under 5% in all cases, and under the 3% bandwidth tolerance except in the case of 100 Mb bandwidth and 5% and 10% loss (not shown at

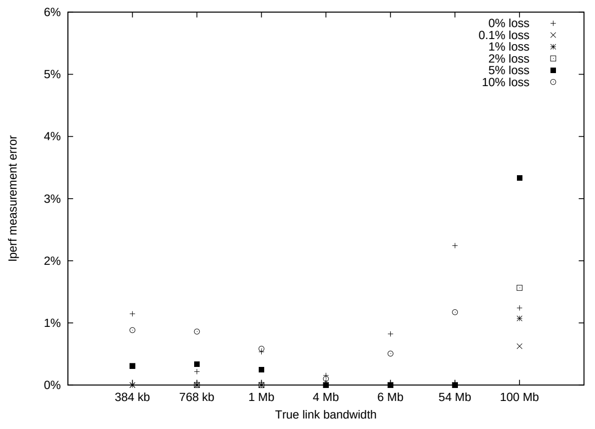

Fig. 4. Linktest measurement error for selected bandwidth and loss combinations.

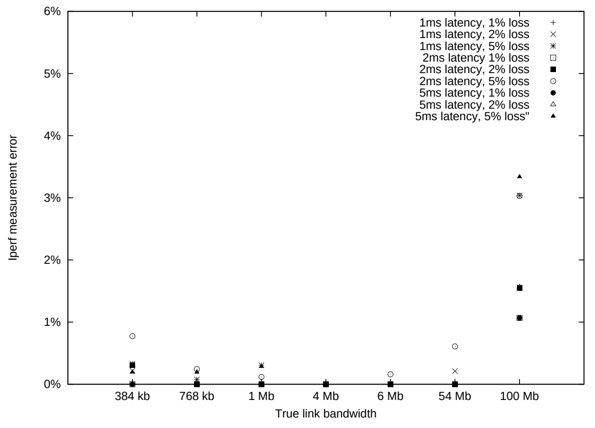

Fig. 5. Linktest measurement error for selected bandwidth, loss and latency combinations.

6.09%). The 100 Mb bandwidth datapoints show somewhat greater error than the slower settings, but are within tolerance for lesser values of loss. In this case, Linktest highlights a known limitation of Emulab, that delay nodes lose bandwidth precision above approximately 95 Mb.

Figure 5 shows combined results of various moderate latency and delay settings. `iperf` measurements used by Linktest continue to stay well within the 3% bandwidth tolerance except at the 100 Mb case. In general, `iperf` has not shown high sensitivity to combining measurement attributes.

B. Macrobenchmarks

We performed scaling measurements to determine how much overhead an automatic invocation of Linktest adds to experiment swapin for experiments of varying sizes. For our measurements, we chose two representative topologies that scale in obvious ways but have distinctly different degrees of connectivity. One topology was a simple LAN, where we varied the number of nodes in the LAN from 5 to 50. In the LAN topology, every node has one link. For the second topology, we constructed a "mesh," where nodes are arranged in a grid with each node directly connected to the nodes before and after it in both the X and Y directions. With nodes along the edges wrapping around, every node in the topology is connected to four others. We scaled the mesh topology from 9 (3x3) to 49 (7x7) nodes.

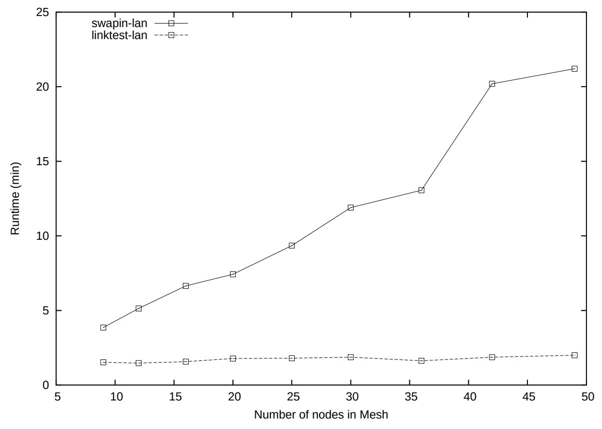

Fig. 6. Linktest and total swapin time for a "mesh" topology with increasing numbers of nodes.

For each topology, we measured the total time taken to swap in the experiment and run Linktest and also isolated the time taken for just the Linktest run. The shaping attributes for nodes in the topologies were fixed at 54 Mb bandwidth, 20 ms latency, and no link loss. The exact values here are not important, as all individual tests run for a fixed time. The contributing factors are the number of individual tests that have to be run and what level of parallelism we can extract from the topology. Note that the routing tests were not run in the LAN topology as all nodes are directly connected.

Figure 6 demonstrates the scaling of the mesh topology. Here we see that as the number of nodes in the topology increases, the total swapin time increases dramatically. This is due to the high degree of connectivity in the topology, which requires creation of large numbers of switch VLANs. VLAN creation is an expensive operation and represents 49-79% of the total swapin time. In contrast, the time required for running Linktest increases modestly, representing only 12-40% of the total swapin time. This confirms the viability of automatically running Linktest on every experiment swapin.

Figure 7 demonstrates the scaling of the LAN topology. In the general case (top two lines), Linktest on a LAN topology scales poorly relative to the mesh topology, representing 54-74% of the total swapin time. Since a LAN (one link per node) is conceptually much simpler than the mesh (four links per node), this may seem counter-intuitive. The reason that Linktest is so expensive in this topology is that every node in an N-node LAN is effectively directly connected to every other node in the LAN, requiring $N(N-1)/2$ tests of each type to be performed. Moreover, because of the way in which Linktest runs (Section III-C) the $(N-1)/2$ tests of any type from any one node are performed serially, so there is a much lower degree of parallelism during the run.

As LANs are a common topology in Emulab, this poor

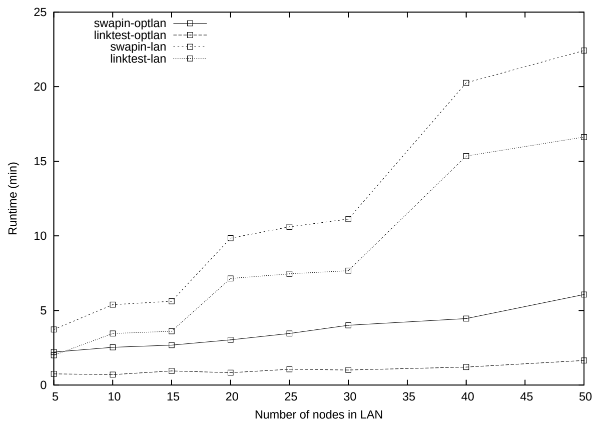

Fig. 7. Linktest and total swapin time for a LAN topology with increasing numbers of nodes. The upper lines show the general, all-pairs testing case. The bottom lines show an optimized case for symmetric LANs.

scaling motivated us to implement an optimization to reduce the number of tests that are needed for validation of common LANs. This optimization is based on the observation that for LANs in which all connections to the LAN have the same shaping characteristics, we need not perform "all pairs" testing. For these "symmetric" LANs, after verifying full connectivity to ensure the switch VLAN has been set up properly, we need only test the link of each node once, not once with every other node. We apply this optimization to the loss and bandwidth tests, resulting in Linktest run times that are an order of magnitude better for moderately large LANs, as shown by the lower two lines in Figure 7. In these cases swapin time is no longer dominated by Linktest runtime.

V. Limitation and Future Work

In this section we briefly discuss some of the limitations of the current implementation of Linktest, and how we might address them in the future.

Emulab currently only supports constant values for shaping characteristics, considerably simplifying the job of Linktest. However, we are in the process of introducing jitter models to the traffic shaping facility. This will require more sophisticated testing on the part of Linktest. A related limitation is Linktest's inability to accurately validate links with non-trivial queue packet drop models such as RED and GRED, both of which are supported in Emulab's shaping mechanism. Testing these models would be challenging. In practice users rarely use RED or GRED, so we don't currently plan to tackle this difficult issue.

Linktest currently only verifies the links specified by the user; it does not test for the existence of extra links that were not requested. These could be detected using "negative" ping tests to check for excess connectivity. It is an open question whether we could make this practical by sufficiently constraining the number of such tests required.

Since Linktest is only run at experiment swapin time, our current use of Linktest effectively assumes static shaping characteristics across the swapped-in life of the experiment. However, the Emulab event system provides a convenient facility for dynamically changing these characteristics during a run, either according to a pre-scheduled event sequence, or by interactively injecting events. Validating link characteristics in these situations, without interfering with the running experiment, is not possible. For static event sequences, it would be possible to pre-test the combination of characteristics that will occur, but this adds an entirely new dimension to the scaling problem.

Emulab's support of asymmetric links requires use of "one way" tests to ensure truly accurate results. This is not difficult except for measuring latency, which requires all machines to have synchronized clocks or the ability to determine the skew between machine clocks. As mentioned in Section III, we previously implemented one-way delay tests, but with unacceptable results. We intend to revisit this methodology, perhaps bounding clock skew to the microsecond-range using Veitch's techniques [9].

A current practical issue is that Linktest has exposed highly transient problems in our testbed which we have not yet tracked down, after significant effort. For example, a random 2 out of 900 node pairs on a 30-node LAN will report high loss rates, but the problem cannot be replicated, and users report no problems. We speculate the switches are the cause, but it could be the node operating systems or the NICs. It could even be false positives from Linktest, but that seems unlikely.

VI. Related Work

Configuration testing is part of standard industry practice in the deployment of new networks [10]. However, there is little published research about performing such tests effectively and quickly. Network construction is traditionally a rare and heavy-weight process as compared to post-deployment monitoring and maintenance. The advent of configurable network testbeds has greatly increased the frequency of network "construction," however, and therefore has led to the need for fast and automated validation systems such as Linktest.

Our work adapts existing measurement tools—which are generally designed for measuring dynamic properties—to checking the static properties of a network. We repurpose these tools further: although they were designed to discover the behavior of a physical network, we use them to validate the intended properties of a synthetic (emulated) network. In addition to the tools that we incorporate, including `iperf` [8] and `rude`/`crude` [6], there are many other tools for estimating available bandwidth, end-to-end latency, end-to-end loss, and so on. Rather than cite individual tools here, we refer readers to Cottrell's large, online catalog [11].

As for general-purpose networks, most existing tools for network testbeds and grids focus on measuring dynamic properties and detecting dynamic faults, in contrast to checking initial configuration. For example, the PlanetLab Slice Anomaly Detector [12] uses machine-learning techniques to detect problems with PlanetLab [13] nodes. After initial training, the system scans presently running processes and uses the classifier to detect anomalous instances. Other systems are based on benchmark suites. For instance, MicroGrid [14] uses both microbenchmarks and macrobenchmarks to validate the Grid and TCP simulation accuracy. Similarly, Teragrid uses the Inca test harness [15] to perform both grid benchmarks and microbenchmarks to verify link bandwidth. In contrast to all these systems, Linktest takes a simpler approach. Acceptable bounds for network attributes are given by the user's network model and predetermined measurement tolerances (Section IV), and these properties are verified before a user is granted access to the configured resources. This is in contrast to dynamic systems, in which resource problems may be detected after a user has started working.

Linktest checks a network's measured behavior against a model of the network's desired properties, which are given in an NS script. NS [3] itself is a simulator that interprets scripts and simulates networks in software, at a fine level of detail. Linktest is therefore akin to the validation test suite [16] that comes with NS: the NS suite checks the correctness of the NS simulator, whereas Linktest checks the correctness of (parts of) Emulab's NS interpreter. These test suites operate at different levels of abstraction, however, corresponding to the different ways in which scripts are interpreted by NS and Emulab.

Linktest is essentially a built-in self-test for (part of) Emulab's testbed management software. Testing is routine during software development, and many applications are distributed with test suites that can be run by users. It is much less common, however, for a software system to automatically test itself each time it is run. Linktest implements this every-time testing strategy for Emulab due to the need for accuracy and the complexity of the network configuration task which involves many independent hardware and software components. Although some researchers have considered software systems that probe their components for failures [17], built-in self-tests are much more common today in the realms of hardware and embedded systems [18]. Our experience with Emulab shows that Linktest and similar built-in self-test mechanisms [19] are needed by modern software systems, too, to ensure correct and autonomous operation, both now and in the future.

VII. Conclusion

Automated validation on experiment swapin helps ensure experiments are configured with the correct experiment topology despite ongoing changes to Emulab. Requirements for such a validation system include speed, scalability, no reconfiguration, non-intrusiveness, operating system portability, an alternate code path, end-to-end validation, and support for arbitrary experiment configurations. In this paper we presented Linktest, a validation system that meets these requirements. Linktest uses an alternate code path to extract an experiment's topology from the NS specification file. The extracted information is used to drive a series of tests validating the link emulation set up by Emulab. The Linktest suite of tests is parallelized to reduce runtime and make it practical to run on every swapin. Microbenchmarks show the tests to be accurate to within 3% in most situations.

Acknowledgments

We thank the anonymous reviewers, Amin Vahdat, and Eric Eide for their helpful feedback on this paper, and thank Eric for editing and formating assistance.

References

[1] 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," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002, pp. 255–270.

[2] University of Utah Flux Research Group, "Emulab: The Utah Network Testbed," http://www.emulab.net/.

[3] The Vint Project, "The ns Manual," http://www.isi.edu/nsnam/ns/doc/-ns_doc.pdf.

[4] R. Ricci, C. Alfeld, and J. Lepreau, "A solver for the network testbed mapping problem," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 32, no. 2, pp. 65–81, Apr. 2003.

[5] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, scalable disk imaging with Frisbee," in *Proc. of the 2003 USENIX Annual Technical Conf.*, San Antonio, TX, June 2003, pp. 283–296.

[6] J. Laine, S. Saaristo, and R. Prior, "Rude & Crude," http://rude.sourceforge.net/.

[7] V. Smotlacha, "One-way delay measurement using NTP," in *Terena Networking Conference 2003*, Zagreb, Croatia, May 2003.

[8] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf homepage," http://dast.nlanr.net/Projects/Iperf/.

[9] D. Veitch, S. Babu, and A. Pasztor, "Robust synchronization of software clocks across the Internet," in *Internet Measurement Conference (IMC)*. ACM, Oct. 2004.

[10] Cisco Systems, Inc., "New solution deployment: Best practices white paper," Cisco Systems, Document 15113, Oct. 2005, http://www.cisco.-com/warp/public/126/newsoln.pdf.

[11] L. Cottrell, "Network monitoring tools (Web page)," Jan. 2006, http://-www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html.

[12] H. Larsen, "PSAD: PlanetLab slice anomaly detection," Jan. 2005, http://www.cs.princeton.edu/~hlarsen/work/psad.pdf.

[13] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the internet," in *Proc. of HotNets-I*, Princeton, NJ, Oct. 2002.

[14] X. Liu, H. Xia, and A. A. Chien, "Validating and scaling the MicroGrid: A scientifi c instrument for Grid dynamics," *Journal of Grid Computing*, vol. 2, no. 2, pp. 141–161, June 2004.

[15] S. Smallen, C. Olschanowsky, K. Ericson, P. Beckman, and J. M. Schopf, "The Inca test harness and reporting framework," in *Proc. of the 2004 ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, Nov. 2004.

[16] "The network simulator ns-2: Validation tests," Jan. 2006, http://-www.isi.edu/nsnam/ns/ns-tests.html.

[17] R. Chillarege, "Self-testing software probe system for failure detection and diagnosis," in *Proc. of the 1994 Conf. of the Centre for Advanced Studies on Collaborative Research*, Toronto, ON, Oct. 1994.

[18] C. E. Stroud, *A Designer's Guide to Built-in Self-Test*. Springer, May 2002.

[19] M. G. Newbold, "Reliability and state machines in an advanced network testbed," Master's thesis, University of Utah, May 2005, http://-www.cs.utah.edu/flux/papers/newbold-thesis-base.html.