DART: Distributed Automated Regression Testing for Large-Scale Network Applications

Brent N. Chun

Intel Research Berkeley
2150 Shattuck Ave. Suite 1300
Berkeley, CA 94704
Tel: +1 510-495-3075
Fax: +1 510-495-3049
bnc@intel-research.net
Keywords: Distributed systems, Performance Analysis, Testing, Validation

Abstract

This paper presents DART, a framework for distributed automated regression testing of large-scale network applications. DART provides programmers writing distributed applications with a set of primitives for writing distributed tests and a runtime that executes distributed tests in a fast and efficient manner over a network of nodes. It provides a programming environment, scripted execution of multi-node commands, fault injection, and performance anomaly injection. We have implemented a prototype implementation of DART that implements a useful subset of the DART architecture and is targeted at the Emulab network emulation environment. Our prototype is functional, fast, and is currently being used to test the correctness, robustness, and performance of PIER, a distributed relational query processor.

1 Introduction

Recently, we have seen the emergence of a number of novel wide-area applications and network services. Examples include distributed hash tables (DHTs) [24, 19, 21, 18, 31], wide-area storage and archive systems [11, 12, 5], distributed query processors [10, 29], content distribution networks [14, 8], robust name services [17], and routing overlays [1, 25]. These distributed applications provide diverse functionality to end users, but nevertheless have one common goal: to deliver correct behavior and high performance in the presence of high concurrency, node and network failures, and transient and persistent performance anomalies. Designing and implementing applications with these characteristics presents significant technical challenges.

With sequential (i.e., single-node) applications, unit testing [4] is an effective and widely used mechanism for building correct, robust, and maintainable software. In unit testing, users write tests that exercise and verify the functionality of specific parts of an application. Over time, users build up a collection of such tests, each covering an increasing fraction of the application's overall functionality. A testing framework automates the execution of unit tests and is applied whenever the application is modified. The end result is that code changes can be automatically verified to have not broken existing functionality (as covered by the unit tests), thereby leading to increased confidence when performing significant modifications to existing code. Building on these ideas, the motivation of this work is to develop an analogous set of automated testing mechanisms with associated benefits for large-scale network applications.

Designing appropriate mechanisms for automated testing of distributed applications presents several challenges. First, such mechanisms need to be fast and scalable to enable large-scale testing and performance analysis. This, in turn, will enable programmers developing distributed applications to obtain rapid feedback on the implications of incremental design and implementation choices. Second, such mechanisms should be flexible to allow applications to be tested along multiple dimensions including correctness, robustness (e.g., in the presence of faults), and performance. Finally, these mechanisms

should enable testing under a wide range of operating conditions in terms of network delays, bandwidth, and packet loss in addition to node and network faults and performance anomalies.

To address these challenges, we have designed DART, a framework for distributed automated regression testing. DART provides users with a programming environment and a set of primitives which can be used to construct a wide variety of distributed tests. Building on a set of scalable cluster tools, DART also provides a runtime that enables efficient execution of such distributed tests at scale. DART targets cluster-based network emulation environments such as Emulab [30] and ModelNet [26] to enable testing under a wide range of network operating conditions. Such environments typically provide two networks: an emulated network to emulate wide-area network delays, bandwidth, and packet loss and a separate, non-emulated control network (e.g., 100 Mbps or Gigabit Ethernet). It is the latter network that DART uses to efficiently and reliably control the execution of distributed tests.

We have implemented a prototype of DART that is targeted to the Emulab [30] network emulation environment. The system implements a core subset of our design which provides enough functionality that we have found it to be useful in practice. In particular, we have and continue to use DART to test and benchmark PIER [10], a distributed relational query processor that runs over a DHT. This paper describes the motivation, design, implementation, and performance analysis of DART and is organized as follows. In Section 2, we motivate the need for automated large-scale testing for distributed applications. In Section 3, we present DART's system architecture. In Section 4, we describe a prototype implementation of DART targeted for Emulab. In Section 5, we measure the performance of our DART implementation for core primitives, a baseline distributed application, and PIER. In Section 6, we present related work and in Section 7, we conclude the paper.

2 Large-scale Distributed Testing

With single-node applications, unit testing frameworks provide two key components to the programmer: a set of commonly used mechanisms for writing tests and a runtime that automates test execution. Common mechanisms in unit testing frameworks include templates for setting up and tearing down unit tests, functions for verifying that actual outputs match expected outputs, and functions for communicating test outcomes back to the user. Using these mechanisms, programmers write tests that verify the functionality of specific parts of their application. Depending on the test, verification might include verifying that actual outputs match expected outputs, that bad / corner case inputs are handled correctly, that an application meets expected target performance metrics, and so forth.

A key benefit of these unit testing frameworks is that they *lower the barrier* to verifying correctness, robustness, and performance in an application's implementation. By providing a common set of mechanisms to write tests and a runtime to execute tests, unit testing frameworks make developing, maintaining, and applying unit tests less cumbersome and less error prone by factoring out a common set of machinery and by automating the test execution process. When the barrier to running tests is low, programmers employ them more often and subsequently reap the benefits of verifying that what worked before continues to work even after significant code changes.

While unit testing is pervasive in the world of single-node applications, there has been little work on providing an analogous set of mechanisms for large-scale distributed applications. We believe that providing such mechanisms will be a key enabler towards rapidly building distributed applications that are correct, robust, and deliver high performance under a wide range of operational environments. Providing such mechanisms requires factoring out and implementing commonly used mechanisms for distributed testing and implementing a runtime layer that executes these mechanisms in a fast and efficient manner. Ensuring that the testing infrastructure is itself fast and robust is key since rapid, correct feedback to the programmer usually implies that the programmer will use the system more often when developing.

3 Architecture

This section describes the DART system architecture. As mentioned, the goal of a DART system is to support automated testing of large-scale distributed applications. For a given distributed application, a user may wish to perform a variety of tests that test the application's correctness, robustness, and performance under a range of operating environments. DART supports automated execu-

tion of a suite of such distributed tests, where each test involves: (i) setting up (or reusing) a network of nodes to test the application on, (ii) setting up the test by distributing code and data to all nodes, (iii) executing and controlling the distributed test, and finally (iv) collecting the results of the test from all nodes and evaluating them. To support this automation, DART relies on a number of components (Figure 1) which are described further in this section.

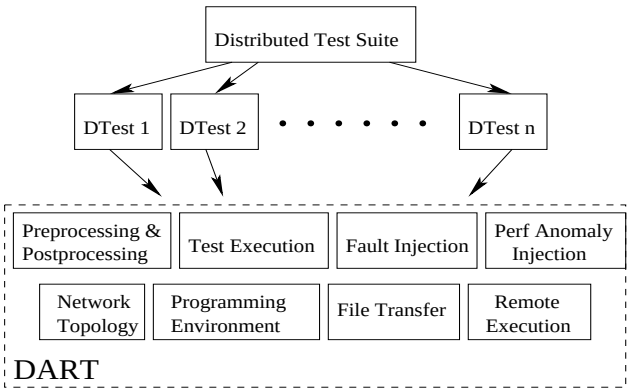

Figure 1: DART architecture. Each distributed application has a suite of distributed tests. Each test is instantiated and executed using DART.

3.1 Network Topology

The first step in executing a DART test is setting up a network of nodes to test the application on. In emulated network environments, such networks are constructed using a set of cluster machines with emulated inter-node network delays, bandwidth, and packet loss. In Emulab [30], for example, users set up experiments consisting of network topologies which specify end hosts, routers, and network links with varying delay, bandwidth, and loss characteristics. Each experiment is then physically instantiated using a set of cluster nodes, a per-experiment VLAN, and wide-area network emulation using DummyNet [20]. ModelNet [26], another emulation environment, provides similar functionality. In addition, its adds per-hop delay, bandwidth, and loss emulation as well as distillation of large network topologies which enables trade-offs between scalability and emulation accuracy to be made (e.g., when using large network topologies [7]).

Given a target environment, a DART implementation provides two ways for a user to specify network topologies. First, DART provides a set of parameterizable network topologies (routers and end hosts), each of which maps down to a description in an underlying network topology language (e.g., Emulab ns-2 files). Second, DART supports raw network topologies as expressed in the target platform's network topology language. In DART, parameterizable topologies are provided mainly as a convenience. Such topologies might include topologies representative of real networks, topologies which might be easy or hard for different classes of applications, and/or topologies that reflect realistic end host heterogeneity in terms of last-hop bandwidth, latency, and host availability [22]. In many cases, we anticipate parameterizable topologies will provide a sufficiently broad range of environments to test and characterize the behavior of a distributed application before moving towards real wide-area network environments (e.g., PlanetLab [15], RON [2], etc.) where additional noise can make it difficult to ascertain whether observed problems are due to the application or due to the infrastructure and the real world.

3.2 Remote Execution and File Transfer

The second step in executing a DART test is setting up the test by distributing code and data to all nodes. Efficiently setting up and subsequently (Section 3.4) executing distributed tests in DART relies heavily on two key components of the DART runtime: multi-node remote execution and multi-node file transfer. In DART, there are a number of cases where multi-node remote execution is needed. For example, in testing a peer-to-peer application, multi-node remote execution might be used to start the application up on all nodes in the system and, some time later, to start a set of clients who issue requests. Before such a test can even run, code and data will also need to be distributed to all nodes, and this further requires having the ability to perform multi-node file transfers. Remote execution needs to be efficient because nodes might be controlled in various ways throughout a test (e.g., starting up servers, starting up clients, creating and controlling adversaries, etc.). File transfer needs to be efficient because code and data may be large and distributing such data to multiple nodes in a large scale test will be costly if it is read from, say, a centralized NFS file server. Consequently, a DART implementation needs to provide fast remote execution and file transfer primitives if the system aims to scale up to large system sizes.

3.3 Scripting and Programming Environment

To facilitate writing distributed tests, DART provides scripting to specify high-level details of test execution and a minimal programming environment which provides low-level details for writing actual distributed test code that runs on the system. Each test in DART has both an XML test script and test code and data. The test script specifies a unique test name, a unique topology name (to enable topology reuse), a network topology (e.g., an Emulab ns-2 file), test code and data, a test duration, a preprocessing script, a set of scripted commands, a set of scripted faults, a set of scripted performance anomalies, and a postprocessing script. Test scripts are interpreted by DART and associated actions are executed using the DART runtime. For example, a script for a distributed storage system might specify code and data for the storage system, start a set of storage servers on all nodes, start a client that writes and reads specific data, and verify consistency of the results in a postprocessing script.

DART provides a minimal programming environment to facilitate the writing of distributed test code. When executing DART tests, one node is designated as the master while all remaining nodes are designated as slaves. The DART runtime uses the master as the point of control for executing and coordinating the entire test. Similar to GLUnix [16], any scripted command executed on any node through DART is provided with the following environment variables:

- `DART_TEST`: unique test name.
- `DART_NODES`: space-delimited list of node IP addresses on the emulated network.
- `DART_NUM_NODES`: number of nodes in the DART test.
- `DART_MY_VNN`: node number from 0 to `DART_NUM_NODES` - 1.
- `DART_MASTER`: master's emulated IP address.
- `DART_GEXEC_MASTER`: master's control IP address.
- `DART_MY_IP`: this node's emulated IP address.
- `DART_GPID`: globally unique identifier for this particular test instance.
- `DART_COMMON_DIR`: directory for code and data common to all nodes.
- `DART_MY_INPUT_DIR`: input directory for per-node code and data.
- `DART_MY_OUTPUT_DIR`: output directory for per-node code and data (e.g., for writing test output, logfiles, etc).
- `DART_ALL_OUTPUT_DIR`: aggregated output directory of all `DART_MY_OUTPUT_DIR` directories. This directory is populated during a collect phase at the end of a test.

Using these environment variables facilitates writing distributed tests using DART. For example, consider testing the correctness of query evaluation in PIER. Such a test needs to instantiate a PIER process on every node and it needs to instantiate clients on a subset of nodes, each of which will issue queries to the system and save the results for verification. Starting PIER up on a node minimally requires at least one piece of information: the IP address of a landmark node to bootstrap all nodes into the DHT. Using the above environment, one obvious possibility for this is to simply use the DART master (`DART_MASTER`). Each PIER process will also want to save relevant output for potential debugging (e.g., `stderr` in case an exception occurs) and PIER clients will need to save query results for postprocessing to verify query evaluation correctness. Using the above environment, capturing program output would be done by simply writing files to `DART_MY_OUTPUT_DIR`. When the test completes, DART collects output from all `DART_MY_OUTPUT_DIR` directories on all nodes and places them in `DART_ALL_OUTPUT_DIR` on the master where the results of the test are then computed (e.g., checking actual output against known, correct output).

3.4 Preprocessing, Execution, and Postprocessing

The third and fourth steps of executing a DART test are executing and controlling the distributed test and, lastly, collecting the results of the test from all nodes and evaluating them. Each distributed test in DART goes through preprocessing, execution, and postprocessing phases to compute the results of the test. Each of these phases is scripted by the user using the primitives provided by DART. Given a network of nodes (e.g., an experiment on

Emulab) and code and data that has been distributed to those nodes, preprocessing is the first stage and entails executing whatever commands that are necessary before actually running the test. For example, if software packages (e.g., RPMs or tarfiles) were distributed as part of the code and data distribution phase, then preprocessing would be the place where one-time installations of this software would take place. We separate preprocessing from the actual execution of the test since, for a given application, we expect it will be frequently be the case that an application performs the same preprocessing in each of a series of tests (e.g., installing the same set of RPMs, such as the Java JDK in PIER's case).

Once preprocessing is complete, DART then proceeds to the execution phase where execution and control of the distributed test is performed to completion. This phase primarily entails scheduling and executing user-specified, scripted commands on specific subsets of nodes at specific points in time (e.g., starting a set of servers up, starting a set of clients, etc.). Further, depending on the test, it might also involve injecting faults and performance anomalies in certain parts of the system at certain points in time. A churn test for a peer-to-peer application, for example, might involve first starting the application on all nodes in the system, letting the system stabilize for several minutes, then injecting a sequence of node join (scheduled command) and leave (scheduled process or node fault) events into the system and measuring the system's behavior over time (e.g., the success or failure of routing requests in the case of structured peer-to-peer overlays).

Finally, once the distributed test has finished executing, a postprocessing stage is performed to collect all the output from all the nodes and to apply a user-specified postprocessing test to process the test's output and verify its goodness. The definition of goodness will be specific to the application and the type of test being performed. For example, a correctness test might verify that actual replies to client requests match the correct, expected values (which would be computed offline a priori). A robustness test might verify that after killing some subset of nodes that the system continues to function as expected (e.g., suppose it was designed to be k-fault tolerant). Finally, a performance test might compute the overall performance numbers from all nodes and verify that these performance numbers lie within some expected bounds. Each test produces output, which may optionally be sent back to the user's machine (e.g., performance numbers) and returns a 1 or a 0 depending on whether the test succeeded or failed (as defined by the user).

3.5 Fault Injection

To understand how a distributed application behaves in the presence of node and network faults, DART also provides fault injection primitives which may be specified by the programmer when scripting a distributed test. Which primitives are supported in a particular implementation will depend on the capabilities of the underlying platform. In the best case, node, process, and network failures are all supported and can be scripted to execute at specific times on specific parts of the system (e.g., a specific subset of nodes):

- **Node failures**: specifies hard failures of specific subsets of nodes over specific periods in time. In Emulab, such failures can be scripted using underlying support from Emulab's event system.
- **Process failures**: specifies the hard failure of specific processes (e.g., by name, by `uid`, etc.) on a given node. In contrast to node failures, the node continues to operate properly.
- **Network failures**: specifies the failure of specific parts of the network at specific points in time. As with node failures, network failures can also be scripted through support from Emulab's event system (e.g., to turn a network link off at a specific time).

3.6 Performance Anomaly Injection

In addition to hard node and network faults, another important class of failures of interest are performance failures [3]. For example, consider the case where a 1.5 Mbps network link does not fail completely but its effective bandwidth drops to 0.001 Mbps. While technically the link has not failed in the sense that it fails to route packets, the performance impact of such a performance degradation is likely to have significant implications for application performance. Understanding how applications behave in the presence of such performance faults is an important step towards building robust distributed applications. Towards this end, DART provides a set of primitives to introduce performance anomalies into the system. Similar to hard failures, the types of

scripted performance anomalies supported by DART include:

- **Node and process performance anomalies**: decreased or varying CPU, memory, network, and I/O performance. Such anomalies might be introduced by using sufficient powerful schedulers [28, 9, 6, 23] in combination with support from the underlying emulation environment.

- **Link performance anomalies**: increased delay, decreased bandwidth, and increased packet loss in specific parts in the network. Such anomalies might be introduced using support provided by the underlying target platform (e.g., using Emulab's event system to dynamically change link delays, bandwidth, and packet loss).

4 Implementation

We have implemented a DART prototype targeted to the Emulab network emulation environment. Our prototype is implemented using a combination of C and Python and supports a subset of the architecture described in Section 3. Parameterizable network topologies, efficient multi-node remote execution and file transfer, a scripting and programming environment, and preprocessing, execution, and postprocessing of arbitrary scripted commands at specific times on subsets of nodes are all supported. Our prototype is functional, efficient, and is currently being used on a routine basis for testing, debugging, and benchmarking PIER.

4.1 GEXEC and PCP

As mentioned, multi-node remote execution and file transfer are key primitives that are used heavily throughout DART and hence need to be fast and efficient. To address this need, we have designed and implemented GEXEC, a fast multi-node remote execution system, and PCP, a fast, multi-node file transfer utility. Both systems rely on a hierarchical design based on a k-ary tree of TCP sockets over a specific set of nodes (e.g, nodes specified using the `GEXEC_SVRS` environment variable for GEXEC). Such trees are built on every invocation of either the `gexec` or `pcp` command using a $O(log_k(n))$ tree building step which involves routing tree create messages down to leaf nodes and routing tree create acknowledgments back to the root. We use a tree-based approach primarily for parallelism and to utilize aggregate resources across all nodes.

GEXEC provides multi-node remote execution of arbitrary commands by routing commands down the tree to all nodes. For all commands, GEXEC supports transparent forwarding of Unix signals, `stdin`, `stdout`, and `stderr` to allow control of remote processes and also obtain remote output. Control and data are all transferred over the tree, down in the case of signals and `stdin` and up in case of `stdout` and `stderr`. Two remote execution models are supported: default and detached. In default mode, the failure model is that if any node fails during the execution, GEXEC aborts on all nodes. In contrast, in detached mode, GEXEC simply builds the tree, starts the command on all nodes, and exits. Both modes are used in DART (e.g., default mode for executing bootstrapping commands, detached mode for running the application being tested, which might crash).

PCP provides fast multi-node file transfer by routing files down the tree in an incremental fashion in 32 KB chunks. Starting with the root, chunks are sent to each node's children. As each chunk is received, each node writes the chunk to local disk, then forwards the chunk off to each of its children. Because files are transferred using a k-ary tree and transferred in chunks (which incur small store-and-forward delays as compared to sending the entire file at once), PCP provides both parallelism and pipelined execution that leads to very high aggregate bandwidth usage. Generally, the optimal choices for tree fanout and message size will depend on node network bandwidth, the network's configuration, and disk write bandwidth. As we show in the next section, using a fanout of 1 and 32 KB messages delivers high performance on Emulab and thereby makes multi-node file transfer a highly efficient primitive in our DART prototype.

4.2 Master and Slaves

Our DART prototype targets the Emulab network emulation environment and uses GEXEC and PCP as the basis for fast distributed test execution (Figure 2). In our implementation, tests are remotely instantiated and controlled using two machines: `users.emulab.net` and a master node arbitrarily chosen from the set of nodes in the test's network topology. We use `users.emulab.net` to manage network topologies

for DART (e.g., creating and destroying experiments). Each node in an Emulab experiment is assigned one or more emulated IP addresses and one control IP address. We use `users.emulab.net` to obtain information about the network configuration of each Emulab experiment. This information is subsequently used to control distributed test execution by running GEXEC and PCP over the fast, control network.

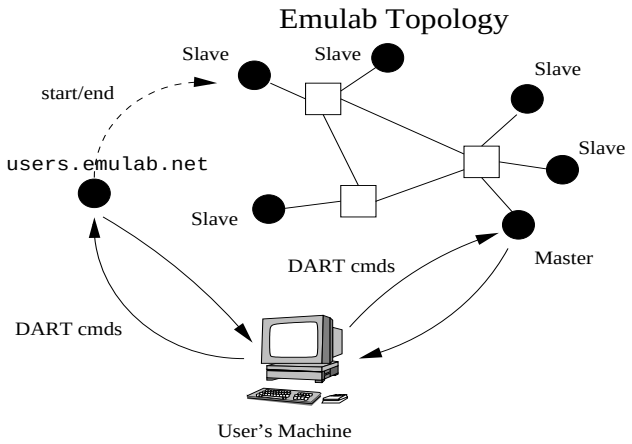

Figure 2: DART implementation on Emulab.

Each Emulab experiment created using DART is bootstrapped with a few common features that are required for DART to operate properly. First, each node is bootstrapped with a small set of core software including GEXEC, PCP, and `authd`, an authentication service used by both GEXEC and PCP. Second, each node is configured to boot the RedHat 7.3 Linux distribution which uses the Linux 2.4.18 kernel. The common software set is required since this software forms of the basis of the DART runtime. The use of Linux on the nodes is needed primarily because the versions of GEXEC and PCP currently used in DART do not run on FreeBSD, the other node operating system available on Emulab.

Once an Emulab topology is instantiated, all subsequent control is done through the master which essentially serves as a proxy for executing distributed tests in DART. Among the master's tasks are: distributing code and data to all nodes, providing the programming environment for distributed tests, and performing preprocessing, execution, and postprocessing of tests across all nodes. In our current implementation, we use `ssh` to securely execute commands on the master and use GEXEC to execute commands and PCP to transfer code and data to other nodes in the system. For example, to reset an experiment such that it can be reused, we use `ssh` to send a reset command to the master and use GEXEC, invoked from the master, to quickly reset all nodes in the network by remotely removing old files and killing old processes from the previous test.

5 Evaluation

In this section, we analyze the performance of our DART implementation. We begin by measuring the performance of two key primitives: multi-node remote execution and multi-node file transfer. As described in Section 4, these primitives are implemented by GEXEC and PCP, respectively, and are used extensively in our DART prototype. Next, we analyze the overall performance of performing DART tests for both a baseline distributed application and PIER, a distributed relational query processor. All experiments were performed on Emulab. The first set of experiments were performed on 64 Pentium III nodes: 18 of which were 600 MHz nodes with 256 MB of memory, 46 of which were 850 MHz nodes with 512 MB of memory. The second set of experiments were performed on 32 Pentium III nodes: 10 of which were 600 MHz nodes and 22 of which were of the 850 MHz variety. All nodes in both cases ran the Linux 2.4.18 kernel and were connected via 100 Mbps Ethernet.

5.1 Performance of DART Primitives

The first set of measurements characterizes the performance of multi-node remote execution and multi-node file transfer using GEXEC and PCP. Figure 3 depicts remote execution performance on multiple nodes using GEXEC. Each curve corresponds to GEXEC's performance using a different tree fanout. Recall that GEXEC performs multi-node remote execution by first building a k-ary tree where k is the fanout at each non-leaf node and using this tree to control remote execution. Each point on each curve represents the remote execution time (milliseconds) to execute a simple command (`/bin/date`) on n nodes ($n = 1, 2, 4, \ldots, 64$). Each point on each curve is the average of 30 different runs on a subset of Emulab nodes. Overall, we observe that remote execution using GEXEC is fast (typically about 100 ms) and that remote execution times do not appreciate much as we scale the system size up. This, in turn, implies fast and efficient control of distributed tests in DART using GEXEC.

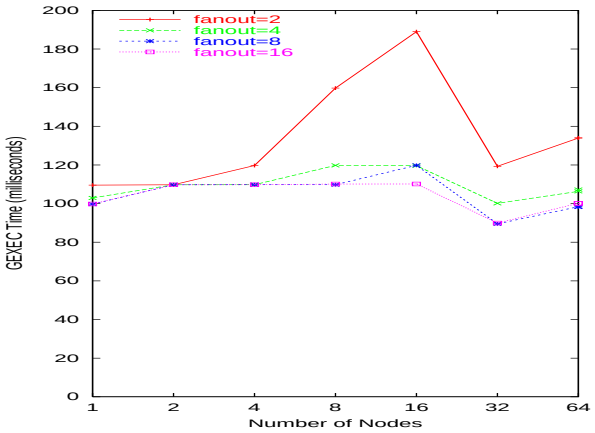

Figure 3: GEXEC performance on Emulab. Each curve corresponds to a different tree fanout, while each point represents the remote execution time (milliseconds) to execute a simple command (`/bin/date`) on n nodes ($n = 1, 2, 4, \ldots, 64$).

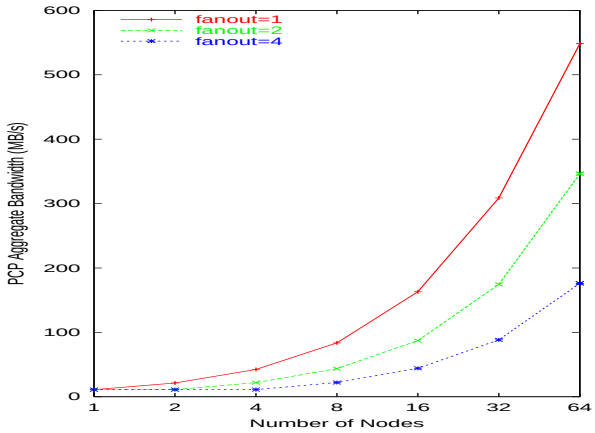

Figure 4: PCP performance on Emulab. Each curve corresponds to a different tree fanout, while each point represents the aggregate bandwidth delivered when distributing a 34.7 MB file (the Java 1.4.2_03 JDK RPM) to n nodes ($n = 1, 2, 4, \ldots, 64$) using PCP.

Next, we perform a similar experiment to measure the performance of multi-node file transfer using PCP. Similar to GEXEC, PCP also builds a k-ary tree and uses this tree to perform parallelized, pipelined file transfer. Figure 4 shows the aggregate bandwidth delivered when distributing a 34.7 MB file (the Java 1.4.2_03 JDK RPM) to n nodes ($n = 1, 2, 4, \ldots, 64$) using PCP and using 32 KB messages. Each curve corresponds to a different tree fanout and each point on each curve is the average of 20 different runs. Using a tree fanout of 1 (i.e., a chain), we observe that PCP is able to deliver an average of 548 MB/s of aggregate bandwidth when distributing a 34.7 MB file to 64 nodes. Larger tree fanouts do not help in the case of Emulab since each node is connected by 100 Mbps Ethernet (i.e., 12.5 MB/s of peak bandwidth) and each node can write to disk at least that fast. Hence, our DART prototype uses PCP's default fanout of 1 which, as shown, delivers high performance and enables data to be moved around efficiently when conducting large-scale DART tests.

5.2 Overall DART Performance

The second set of measurements quantify the overall performance of performing DART tests for both a baseline distributed application and a real distributed application (PIER). The baseline distributed application is the null distributed application. It's an application that runs on 32 nodes but does not perform any computation. The test returns immediately and thus the times associated with this test are, in the current implementation, a lower bound on the total time to execute a distributed test in DART. PIER, as mentioned, is a distributed relational query processor that runs over a DHT. We use DART to routinely perform a number of tests on PIER. In this instance, we present performance results when testing the correctness of a distributed selection query on 32 nodes using different query plans (e.g., different packet sizes). (The test queries static per-node data and hence we know what the correct query result ought to be.) The time to perform this particular test once the test has been set up on all nodes is 700 seconds. The goal of these measurements is to show that the overhead of performing DART tests is small relative to distributed test times, which we anticipate will involve running a test for at least several minutes (e.g., as in the PIER selection query test) in most cases.

Each test involves four potential components. First, there is the time to set up the network topology for the test (*esetup*). This involves the time to securely transfer an Emulab network topology file to `users.emulab.net` and to instantiate the Emulab experiment. Second, there is the time to set up a particular distributed test (*dsetup*). The main cost here is transferring code and data to the master node in the Emulab experiment and distributing code and data to the slaves. Third, there is the time to perform preprocessing, exe-

cute and control the distributed test, collect the results on the master, and perform postprocessing (*drun*). Fourth, there is the cost of reseting the test environment on all nodes (*dreset*)). This involves clearing out results from the previous test and killing all processes associated with the previous test. Note that a test may reuse a network topology from a previous experiment if that test uses the same topology (e.g., the same 32-node topology in our measurements). When running a test for the first time on a network topology, no *dreset* cost is incurred since the system is clean, whereas when reusing a topology for a different test, the *dreset* cost must be paid.

	Base	Base reuse	PIER	PIER reuse
esetup	202.3	–	206.3	–
dsetup	16.2	16.0	52.6	46.2
drun	28.8	29.2	758.7	735.7
dreset	–	4.2	–	4.0
Total	247.3	49.4	1017.6	785.9

Table 1: Breakdown of overall times (seconds) to run distributed tests on 32 Emulab nodes using DART for a baseline null application and a 700 second correctness test in PIER for a distributed selection query.

Table 1 shows the overall times (seconds) to run distributed tests on 32 Emulab nodes using DART for a baseline null application and a 700 second correctness test in PIER for a distributed selection query. For both the baseline and for PIER, we present results when a new Emulab experiment is instantiated and when an existing Emulab experiment is reused (the reuse columns), the latter case requiring an additional reset component to prepare for a new test.

We observe the largest baseline cost to be *esetup*, the time to instantiate a new 32-node Emulab experiment. Measurements on Emulab revealed this time to be, on average, 204.3 seconds which is consistent with previous measurements [30]. The relatively high cost of creating a new Emulab experiment suggests reusing existing Emulab experiments when conducting tests on the same network topology. As mentioned, reusing a topology requires an additional reset phase to clear old files and kill old processes. Our measurements indicate that these costs are, on average, 4.1 seconds which is relatively low. Still, this number is relatively high compared to GEXEC remote execution times. (We use GEXEC to clear old files and kill old processes from the master.) This is largely due to our use of a new `ssh` connection each time we communicate with either `users.emulab.net` or the master. This overhead is also a significant component in the other baseline costs as well, namely *dsetup* and `drun` which on average were 16.1 seconds and 29.0 seconds respectively. When reusing the network topology, the total baseline cost to execute a null distributed test on 32 nodes was 49.4 seconds.

Turning to PIER, the key numbers of interest are the *dsetup* and *drun* times. We measured the average *dsetup* time for PIER to be 49.4 seconds, while for the baseline, the average *dsetup* cost was 16.1 seconds. The main difference between the two is the additional cost associated with transferring code and data to the master and from the master to all slaves. In the PIER case, code and data transferred from the user's desktop to the master was 3.32 MB in size (four different directories), while code and data transferred from Emulab's NFS fileserver to the master totaled 37.0 MB, the size of the Java 1.4.2_03 JDK and the static data being queried. As shown in Figure 4, transferring data from the master to all slaves using PCP is efficient. However, as with the baseline, liberal use of new `ssh` connections again incur significant overhead. In the current implementation, each directory being transferred causes a new `ssh` connection to be created to the master, each of which usually takes approximately 2-3 seconds. We intend to optimize this by establishing a single secure connection with the master and reusing it in the future. This should reduce the gap between the baseline and PIER by approximately 12-18 seconds.

Despite the overhead of multiple `ssh` connections to the master, we see that the overhead of using DART to perform distributed tests of PIER is still quite reasonable relative to the typical time to perform a meaningful test. In this case, the selection query correctness test needs to run for 700 seconds. This includes a 120 second delay to allow the DHT to stabilize and for PIER to build up a multicast tree to perform query dissemination to all nodes. It also includes the time to perform a selection query in four different ways, in each case allowing the query to run for 120 seconds and leaving 10 seconds in between each query to avoid query interference. Finally, a minute is alloted before finally shutting down the test, which leads to a test time of 700 seconds. Relative to the total time, the DART overhead in this case is 11.3% (i.e., 85.9 seconds out of 785.9 seconds) which we believe is quite reasonable given the `ssh` performance improvements we intend to make and the fact that distributed

testing using DART is entirely automated and does not require any human intervention.

6 Related Work

There have been relatively few efforts aimed at building frameworks for large-scale testing of distributed applications. In this relatively small space, the closest related project is TestZilla [27]. Like DART, TestZilla provides a framework for testing distributed applications and leverages a set of scalable cluster-based tools in its implementation. In TestZilla, distributed tests are executed through a centralized coordinator and the system provides mechanisms for network topology specification (in a non-emulated cluster setting), file system and process operations, barrier synchronization, and logging and collection of output files. Architecturally, DART and TestZilla share many of the same characteristics although both aim to provide slightly differing feature sets. Unlike DART, which focuses on wide-area distributed applications in an emulated network environment, TestZilla is focused primarily on cluster-based applications in a Windows environment. As a consequence of this, TestZilla relies heavily on Windows-specific features in its implementation. In terms of scalability, both systems rely on scalable cluster-based tools for test control. Unfortunately, given that no published numbers on TestZilla's performance were available, a direct performance comparison could not be made.

ACME [13] provides a framework for automatically applying workloads, injecting perturbations, and measuring the performance and robustness of distributed services based on user specifications written in XML. It targets both emulated network environments such as Emulab and ModelNet as well as real wide-area testbeds such as PlanetLab. In ACME, control, measurement, and injection of perturbations is done through per-node sensors and actuators which, in turn, are controlled through a distributed query processor. Like DART and TestZilla, control in an ACME experiment is done using a centralized experiment control node. Using the query processor, measurements are taken by issuing queries which read desired sensors on multiple nodes in the system. Similarly, actions (e.g., rebooting a node, modifying a link's bandwidth) are invoked by issuing queries that invoke appropriate actuators. Early experience using ACME to evaluate the robustness of three key-based routing routing layers (Chord, Tapestry, and FreePastry) showed that ACME was able to uncover a number of interesting properties and bugs under various workloads and perturbations. Compared to DART, ACME shares many of the same goals. Architecturally, however, ACME differs quite a bit owing to its use of a distributed query processor and the sensor/actuator abstraction as the basis of its implementation.

7 Conclusion

We have developed DART, a framework for distributed automated regression testing of large-scale network applications. We presented the DART system architecture and described the mechanisms DART provides, including scripted execution of multi-node commands, fault and performance anomaly injection, and the runtime layer that supports these mechanisms. We have implemented a DART prototype that implements a useful subset of the architecture and are using this prototype in ongoing testing and benchmarking of PIER, a distributed relational query processor. Our prototype is built on fast and efficient multi-node remote execution and file transfer primitives and incurs reasonable overheads (e.g., 11.3% overhead for a PIER selection query correctness test) for typical distributed tests of interest. Future work on DART includes implementation of additional test mechanisms (e.g., fault injection using Emulab's event system), additional performance optimizations, and further work on gaining experience using DART to test PIER and other wide-area distributed applications. We believe that distributed testing frameworks will be a key enabler towards rapidly building distributed applications that are fast, robust, and deliver high performance across the wide-area.

Acknowledgements

We would like to thank the Emulab team for providing access to the Utah Emulab cluster and for being highly responsive to numerous questions and various feature requests.

References

[1] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the 18th*

ACM Symposium on Operating Systems Principles (October 2001).

[2] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. Experience with an evolving overlay network testbed. *ACM Computer Communications Review 33*, 3 (2003), 13–19.

[3] ARPACI-DUSSEAU, R. H. *Performance Availability for Networks of Workstations*. PhD thesis, University of California, Berkeley, 1999.

[4] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[5] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (October 2001).

[6] DEMERS, A., KESHAV, S., AND SHENKER, S. Anaylsis and simulation of a fair queueing algorithm. In *Proceedings of the 35th IEEE Computer Society International Conference (COMPCON)* (March 1990), pp. 380–386.

[7] ELLEN W. ZEGURA, K. C., AND BHATTACHARJEE, S. How to model an internetwork. In *Proceedings of IEEE Infocom '96* (March 1996).

[8] FREEDMAN, M., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (March 2004).

[9] HAND, S. Self-paging in the nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (February 1999).

[10] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the internet with pier. In *Proceedings of the 29th International Conference on Very Large Data Bases* (September 2003).

[11] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems* (November 2002).

[12] MUTHITACHAROEN, A., MORRIS, R., GIL, T., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation* (December 2002).

[13] OPPENHEIMER, D., VATKOVSKIY, V., AND PATTERSON, D. A. Towards a framework for automated robustness evaluation of distributed services. In *Proceedings of the 2nd Bertinoro Workshop on Future Directions in Distributed Computing (FuDiCo II): Survivability: Obstacles and Solutions* (June 2004).

[14] PAI, V. S., WANG, L., PARK, K., PANG, R., AND PETERSON, L. The dark side of the web: An open proxy's view. In *Proceedings of the 2nd Workshop on Hot Topics in Networks* (November 2003).

[15] PETERSON, L., CULLER, D., ANDERSON, T., AND ROSCOE, T. A blueprint for introducing disruptive technology into the internet. In *Proceedings of HotNets-I* (October 2002).

[16] PETROU, D., RODRIGUES, S. H., VAHDAT, A., AND ANDERSON, T. E. Glunix: A global layer unix for a network of workstations. *Software - Practice and Experience 28* (1998), 929–961.

[17] RAMASUBRAMANIAN, V., AND SIRER, E. G. The design and implementation of a next generation name service for the internet. In *Proceedings of the ACM SIGCOMM '04 Conference on Communications Architectures and Protocols* (August 2004).

[18] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference on Communications Architectures and Protocols* (August 2001).

[19] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a dht. In *Proceedings of the USENIX 2004 Annual Technical Conference* (June 2004).

[20] RIZZO, L. Dummynet and forward error correction. In *Proceedings of the USENIX 1998 Annual Technical Conference (FREENIX Track)* (June 1998).

[21] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms* (November 2001).

[22] SAROIU, S., GUMMADI, K. P., AND GRIBBLE, S. D. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems 9* (2003), 170–184.

[23] SHENOY, P., AND VIN, H. M. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 ACM SIGMETRICS Conference* (June 1998), pp. 44–55.

[24] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference on Communications Architectures and Protocols* (September 2001).

[25] SUBRAMANIAN, L., STOICA, I., BALAKRISHNAN, H., AND KATZ, R. Overqos: An overlay based architecture for enhancing internet qos. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (March 2004).

[26] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation* (December 2002).

[27] VOGELS, W. Testzilla: a framework for the testing of large-scale distributed systems. Available from: `http://www.cs.cornell.edu/vogels/TestZilla/default.htm`.

[28] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation* (1994), pp. 1–11.

[29] WAWRZONIAK, M., PETERSON, L., AND ROSCOE, T. Sophia: An information plane for networked systems. In *Proceedings of the 2nd Workshop on Hot Topics in Networks* (November 2003).

[30] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation* (December 2002).

[31] ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. CSD-01-1141, University of California, Berkeley, Computer Science Division, 2001.