Apt: A Platform for Repeatable Research in Computer Science

Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb,
Jonathon Duerig, Keith Downie, and Mike Hibler

University of Utah
Salt Lake City, Utah

{ricci, gtw, stoller, kwebb, duerig, kdownie, mike}@cs.utah.edu

ABSTRACT

Repeating research in computer science requires more than just code and data: it requires an appropriate *environment* in which to run experiments. In some cases, this environment appears fairly straightforward: it consists of a particular operating system and set of required libraries. In many cases, however, it is considerably more complex: the execution environment may be an entire network, may involve complex and fragile configuration of the dependencies, or may require large amounts of resources in terms of computation cycles, network bandwidth, or storage. Even the "straightforward" case turns out to be surprisingly intricate: there may be explicit or hidden dependencies on compilers, kernel quirks, details of the ISA, etc. The result is that when one tries to repeat published results, creating an environment sufficiently similar to one in which the experiment was originally run can be troublesome; this problem only gets worse as time passes. What the computer science community needs, then, are environments that have the explicit goal of enabling repeatable research. This paper outlines the problem of repeatable research environments, presents a set of requirements for such environments, and describes one facility that attempts to address them.

1. INTRODUCTION

Many parts of computer science have a strong empirical component: for example, research in fields as diverse as operating systems, networking, storage systems, data mining, formal verification, databases, cloud computing, machine learning, image analysis, etc. all require experiments to establish credibility. Conducting and publishing research in these areas necessarily involves setting up an appropriate experimental environment, running experiments, and analyzing data. Demonstrating the value of new research usually involves comparing a new system to an existing one to show better performance, scalability, or functionality under some set of assumptions.

Basic repeatability (that is, obtaining the same results under the same conditions) is only a stepping stone to more complex experimental goals: demonstrating improvement over previously published results, revisiting results under changed assumptions, or (less commonly in computer science) validating earlier results by *reproducing* them under independent implementation, etc. Repeatability is, however, too often overlooked; many others have noted this problem as well [29, 4]. The effect is that when a new system is compared to an older one, or new results are compared to old, it can be difficult to tell how much of the difference between the systems are attributable to the systems themselves, and how much are attributable to a difference in environment. If new experiments are compared directly against older numbers, are the environments similar enough in the factors that matter for the numbers to be directly comparable? What are "the factors that matter" for a particular set of experiments? If the older system is re-run by the authors of the newer experiment, are there any differences in the way it was run, compiled, or configured that might cause the results to be significantly different than originally reported?[1] Without the basic standard of repeatability, it can be be difficult or impossible to answer these questions.

On the surface, it would appear that repeating experiments in computer science should be simple: the artifacts that are evaluated are usually software, often with some associated dataset. To repeat published results, "all" one must do is run the software and analyze its performance or the results that it returns. Embedded in this seemingly simple sequence of events, however, is a large obstacle: one must have an *appropriate environment* in which to run the software. Again, on the surface, this does not sound like a particularly hard problem: software by its very nature runs in an artificially-constructed world, so re-creating a particular environment should be a simple matter. The problem, of course, is that the behavior of a piece of software is the result not only of the software itself, but of a large number of variables that are part of its execution environment: these include the processor it runs on, its attached memory and storage systems, the network, operating system, compiler, libraries, etc. This is only the tip of the iceberg however, as the version, configura-

[1]Many times, of course, we vary these factors intentionally, but when we do so, it should be deliberate and with the purpose of testing a hypothesis, and not happen by accident (and perhaps go unnoticed) simply because we had a different environment easily at hand.

tion options, and even compile-time options for any of these components can affect an experiment. As an example of the sensitivity of experimental results to their environment, Mytkowicz et al. [18] showed that a commonplace compiler experiment (evaluating speedup due to optimization passes) can be affected by such seemly innocuous factors as the order in which linking is performed and the size of the UNIX environment (total size of environment variables.)

Variations of these problems are encountered throughout computing, and various package management systems [27, 6, 3] have been created to handle software and configuration dependencies. There are two problems, however, that make package management, in itself, an insufficient solution to the problem of repeatable experimentation. First, the standard of scientific repeatability sets a higher bar than the problem addressed by package managers: the question we need answered is not "will it run?" but "will it run, and produce the same results?" Answering the second question in the affirmative often means having a detailed record of the exact hardware on which the experiment was originally performed. Second, most research software is written to validate a particular scientific hypothesis, after which the author often moves on to other pursuits. Porting the software to environments other than the exact one used by the original experimenters (operating system, compiler version, etc.) is extraneous to proving the hypothesis, so it is neglected; this neglect creates an undesirable barrier to use of that software by others. Because research software often depends on other research software, frequently configured in specific and non-standard ways, the proper environment can be particularly fragile and hard to reconstruct.

A recent study by Collberg et al. [4] attempted to find, download, compile, and run source code for over 600 papers published in top-tier conferences and journals. Their finding was that, when software was available at all (which was a disappointingly low 44% of the time), it was difficult to get it to run; the authors were only able to run 44% of the research code that they could obtain. An (in-progress) follow-up by another group [9] suggests that, with more time and domain knowledge, many of the problems encountered in the first study are surmountable; however, the point remains that it is often difficult to run research software. Of the 126 compilation problems reported in [4], approximately half (60) are directly attributable to a difficulty in producing an appropriate environment (missing or failed dependencies, unavailability of the appropriate hardware or operating system, etc.), and another quarter (35) may be due to this cause.

We argue that, in most cases, the initial author of a piece of research software should not be burdened with porting it to other platforms or future releases of operating systems, nor should this burden be placed (where it falls by default today) on subsequent researchers wanting to use that software. Instead, the position we take in this paper is that *it should be possible to re-create the original conditions under which research artifacts were built, executed, and evaluated.* To be clear, we do not claim that research software should *never* be well-packaged, made portable, or maintained over time; on the contrary, much of the software we value most has these properties. Our claim is instead that these properties are not necessarily required for *all* research software, but that it should nonetheless be possible to repeat published experiments. In the remainder of this paper, we lay out a set of properties for repeatable experimentation environments, and describe the design of a facility, Apt, that is a particular embodiment of them.

2. REPEATABLE ENVIRONMENTS

We begin by defining the terms we use to describe repeatable research.

2.1 Definitions

For the purposes of this paper, we use a simple definition of a *repeatable experiment*: an experiment consists of executing a *system under test* (SUT) in some *execution environment* with some set of *input data*. The experiment produces some some set of *output data* that is to be used to evaluate the system; this data may be the output of the SUT itself and/or performance data gathered about the execution. The execution environment is the closure of all artifacts, both equipment and software, needed to execute the experiment, as well as the configuration of those artifacts: this includes processors, RAM, disks, networks, operating systems, libraries, etc. For an experiment to be considered *repeatable*, a re-execution of the SUT must produce the same output data; when the output data are performance data or some other data with a nondeterministic component, they should fall within some acceptable ϵ of the original values.

In some experiments, the *output* of the SUT is the primary result of interest: for example, how well does a new algorithm learn features in a dataset, or what are the sizes of the routing tables produced by a particular routing algorithm? For these types of experiments, getting the same output *may* not require an identical execution environment. It can still be be difficult, however, to re-create a *suitable* environment for repeat experimentation: the SUT may require specific versions of operating systems or libraries, may require large amounts of computational or storage resources, may require a network of connected systems for execution, or may depend on fragile software stacks that are themselves research artifacts. These all represent barriers to repeating the experiment. Worse, there can be *hidden* dependencies that affect the output of the SUT which are discovered only when the output cannot be repeated in other environments.

In other experiments, the *performance* of the SUT is the primary result of interest: for example, how fast does a routing protocol converge under various churn rates, or how does the scaling of a program analysis change as the size of the program grows? These experiments necessarily depend even more heavily on the specifics of the environment. Hidden effects of the environment are even more insidious for performance experiments, because they can occur anywhere in the hardware or software stack—detailed knowledge of processors, memory, storage, networking, compilers, operating systems and any software libraries used may be necessary to understand, or even discover, them. A change to any of these variables can result in significantly different performance. While it is often desirable to understand how these sorts of factors affect a system, we must be able to establish baselines, and we have nowhere to start unless their values can be known and controlled.

Many experiments are a mix of these two styles. Intuitively, performance evaluations [15] seem more common in computer science than in the domain sciences that use computation, since the former is concerned with computer systems themselves, while the latter is focused on their application.

We aim to address both *output* and *performance* results, but in all cases concern ourselves only with *closed world* experiments: those in which all factors that affect the output of the experiment are internal to the execution environment, and the environment can be defined closely enough that all factors can be controlled or observed by the experimenter.

2.2 Properties

Building on our description of the problem, we arrive at a set of properties that are desirable in a platform for repeatable research:

Consistency. The overriding property of a platform for repeatable research must be the ability to re-create the environment that was used for an earlier experiment. At a minimum, this consistency must cover the aspects of the environment that affect the output of the SUT (for all experiments) and the performance of the SUT (for performance experiments.) In practice, it can very difficult to discover precisely which aspects of the environment may have these effects, so repeatable environments must make tradeoffs with respect to which aspects of the environment they reproduce, and whether those aspects are determined statically or dynamically.

One corollary to consistency is that other activity on the system executing the SUT must not affect the results of the experiment. For example, if running on an experimenter's desktop computer, other processes must not affect the results, and if running on a testbed or other shared infrastructure, simultaneous experiments must not interfere with each other.

Diversity. While consistency is valuable, it is also important to recognize that it has hazards associated with it. Namely, if a homogeneous platform is used for a large body of research, artifacts of the experimentation platform can show up in results and can easily be mistaken for more fundamental "truths." For example, if performance of a set of algorithms is compared on a system with a particular processor, memory system, and set of I/O devices, we learn valuable things about the performance of these algorithms on this particular system, but we do not necessarily learn how these results would generalize to other systems. Running the SUT on a different system may yield different conclusions. It is thus important that, in addition to providing a consistent environment for repeating existing results, the experimentation platform provide access to a diverse set of resources (both hardware and software).

Transparency. A key factor that separates a scientific instrument from a general-purpose computing facility is the ability to understand and control the environment. Any platform will necessarily have properties that affect the experiments run on it due to its choices of hardware and software. It is important in scientific experimentation to understand which variables can be controlled and which cannot, and for those that cannot, to know what their fixed values are or to observe these values over the course of an experiment. For example, an experimenter may not be able to control the network topology of the facility they are using, but should know what that topology is, and the factors (such as switching or routing technologies) that may affect their experiments.

Encapsulation. As discussed earlier, a truly repeatable execution environment must capture the closure of both the hardware and software environment that an experiment was run in, as well as the configuration of both. This must be done in sufficient detail that it is possible for later experimenters to re-create the original environment precisely. We refer to the process of capturing this environment as "encapsulation," and refer to the object thus formed (whatever its specific implementation) as a "capsule."

Publishing and Archiving. The goal of repeatable experimentation is, of course, not for the individual who initially ran an experiment to repeat it, but for others to be able to. It should therefore be possible for experimenters to publish the capsules they create, to refer to specific capsules in publications, and for those references to persist over a long period of time.

Adaptability. Learning a new infrastructure can be a time-consuming activity; researchers should be free to focus on the experiments they are interested in, rather than learning to use the platform. This points to a specific role for domain experts: those who must take the time to learn to use the "raw" infrastructure and adapt it (through configuration and software) to specific research domains.

Longevity. Research results often remain relevant for years or longer, and authors reporting on new work should be able to compare to results that may have appeared in the literature years before. While an *indefinite* lifespan may not be feasible for capsules, a *long* one should be an explicit design goal of the environment. In particular, it should be possible to use a capsule even if the original creator is unable or unwilling to assist; this is a common source of problems today as students graduate, faculty move on to other research pursuits, etc.

3. APT

Apt (the (A)daptable (P)rofile-driven (T)estbed) is our attempt to build a facility with the properties outlined in the previous section. Apt takes a *hosted* approach to building a repeatable environment: it consists of a cluster and a control system that instantiates encapsulated experimentation environments on that cluster. The fundamental philosophy behind Apt is that the surest way to build the foundation of a repeatable environment is to have a set of hardware resources that have known properties and are available, long-term, to all researchers. On top of this, Apt provides the ability to create an on-disk storage environment that can be recreated by later experimenters in a bitwise identical fashion. We believe that this represents a *minimum* requirement for fully repeatable research; this is especially true for research for which performance is a metric of interest, or that requires a full network to execute.

Tools for scripting and orchestrating experiments, analyzing results, etc. are also key for repeatable research. Apt makes the choice not to directly integrate these tools, allowing experimenters to use whichever they wish. This has the advantage of being flexible with respect to different research domains (which have different needs in terms of experiment automation [28, 21]), working styles, and research workflows—we believe this maximizes utility. This choice does have the disadvantage, however, that it does not force users to adequately automate experiments to make repeatability easy.

Apt is built around *profiles*, which are a specific realization of the *capsule* abstraction. Profiles describe *experimental environments*; when a profile is instantiated, its specification is realized on one of Apt's clusters using virtual or physical machines. The creator of a profile may put code, data, and other resources into it, and the profile may consist of a single machine or may describe an entire network. Typically, everyone who instantiates a profile is given their own dedicated resources with the associated code and data loaded; thus, all experiments are conducted in isolation from each other. The infrastructure to support the manipulation of these profiles is based on mature systems, primarily Emulab [30] and GENI [2].

Many different types of users play different roles in their interactions with Apt. New profiles are introduced by *profile creators* who are domain experts; they describe and perhaps perform the original experiment (potentially evolving it through multiple iterations). Subsequent experimenters intending to reproduce or tailor the experiment will at first base their research on that same profile (we label such users as *profile instantiators*). Lastly, some experiments will establish services that might be intended for access primarily by users of the service itself who never interact with the Apt infrastructure directly; we call the latter class *end users*.[2]

Profile creators must register for an Apt account for permission to publish their profiles. Profile instantiators are encouraged but not required to register (guest instantiators are more limited in the types and quantity of resources they may request). There is no need for end users to register with or even be aware of Apt.

User interaction with Apt is primarily through a web interface[3]; there is typically no need for any user to install software on their own host. For many profiles, the Apt web pages suffice for all interaction (even `ssh` login is incorporated, via Web-based terminal emulation). The typical workflow in Apt is:

1. An experimenter starts with a profile provided by Apt, typically one with a basic installation of a standard operating system. The experimenter instantiates this profile, and is given a set of hardware resources in the Apt cluster to which he or she has (temporary) exclusive access.
2. The experimenter sets up their experimental environment by logging into the instantiated resources and installing software, copying in input data, etc.
3. The experimenter takes a "snapshot" of the state of the environment that they have created. This creates a new profile, which belongs to the experimenter.
4. The experimenter runs their SUT through a set of experiments and collects results; when finished, he or she *terminates* the experiment, releasing the hardware resources.
5. The experimenter publishes their results, along with a link to the profile (described in more detail below.)
6. Others may follow this link and instantiate the profile for themselves. When they do, they are given their own temporary allocation of resources. These resources are loaded with the software environment set up in Step 2, and the second experimenter may now run the experiments in Step 4 to reproduce the original results.

[2]For example, a user of Apt may set up a service such as a cloud, a compute cluster, or a service for executing certain types of experiments; users of this service interact directly with it, rather than with Apt, and are considered end users.

[3]`http://www.aptlab.net`

3.1 Profiles

Apt's profiles capture an experimentation environment by describing both the software needed to run an experiment and the hardware (physical or virtual) that the software needs to run. By providing a hardware platform for running these profiles, Apt essentially enables researchers to build their own testbed environments and share them with others, without having to buy, build, or maintain the underlying infrastructure.

Each profile clearly needs to record enough information to recreate the experimental environment on demand. The central reference for a profile's state is its *RSpec* (resource specification), a concept borrowed from GENI [26]. An RSpec is an XML document describing the experiment; each node and network link has a corresponding element in the RSpec and detailed information about node or link configuration is inserted into child elements or attributes. While the RSpec describes the experiment, it does not and cannot encapsulate the entire state of the SUT; some RSpec elements are merely references to data recorded elsewhere (e.g., disk images). The RSpec format is specified using XML schemas, and these schemas are versioned; this will allow Apt to keep backwards compatibility as they evolve. Apt provides a GUI for manipulating RSpecs, and users may also inspect or edit them manually. Work is also underway to provide a scripting language—the output of an execution of one of these scripts is an RSpec, which can be used to describe a profile. In the general case, the script may be re-executed on each instantiation, but for the purposes of repeatability, published profiles will refer to a specific RSpec produced by a prior execution of the script.

An RSpec is not always a *complete* description of an experiment instance (indeed, such a description would frequently be either unwieldy or overly specific); many elements and attributes permitted in RSpecs are optional. When they are omitted, Apt either assumes default values for the missing parameters, or in certain cases is free to substitute any value (so that the omitted parameter behaves like a wildcard). For instance, if a node is requested but no disk image is specified, a default one will be loaded; if a node is requested and no hardware type is specified, Apt will choose *any* of the free nodes that can otherwise satisfy the request. Before a profile is shared with others for the purposes of repetition, these variables should be bound so that the other user gets the same values. Since Apt is a hosted environment with a closed set of hardware types, strings identifying hardware types are used as shorthand for a more exhaustive hardware specification. An instantiated experiment also produces a *manifest*, which records the specific nodes and switches used for a particular instantiation.

Apt also maintains a repository of disk images to be loaded on demand [14]. There are several facility-maintained images globally available, and users have the option of specifying their own images too (which is typically done by requesting one of the system images, modifying it appropriately, and then taking a snapshot of the result to be saved as a user image). Images in Apt currently cover the contents of disks, but not other firmware such as BIOS or NIC firmware; this is a potential area of future work.

Multiple revisions of a profile can be saved, which avoids problems associated with irrevocable changes to experiments or disk images. Therefore, Apt offers repeatability not only of the most recent version of an experiment, but to any designated previous snapshot of its prior state as well.

Apt is capable of provisioning either physical or virtual machines (or even a mixture of both within the same experiment). Unregistered guest users may request virtual machines only. For experiments studying only the *output* of the SUT, VMs might be perfectly adequate; however, when the results of interest include the *performance* of the SUT, running directly on physical hardware might be essential to guarantee repeatability. Even in the latter case, though, we encourage users to do development and debugging on VMs (where we can multiplex nodes to conserve resources), and only once test runs prove satisfactory is it necessary to choose to instantiate on hardware to gather publishable results.

3.2 Repeatability

Apt is designed to meet the repeatability properties presented in Section 2:

Consistency. In most cases, Apt is able to provide identical hardware across multiple instantiations of the same profile, faithfully replicating most or all of the relevant hardware environment. The profile's RSpec records the requested hardware, and can provision equivalent resources across multiple instantiations. By default, each user enjoys exclusive and unlimited (root) access to each node in the experiment[4]; the fact that no other users can interfere with nodes that are temporarily theirs and only theirs can contribute a great deal of consistency, difficult or impossible to replicate on clusters employing fine-grained time-sharing.

[4]The alternative is to request VMs hosted on physical machines potentially shared with other users; we encourage use of this facility when appropriate to conserve scarce resources.

Obtaining this ideal of identical hardware across instantiations does require care on the part of the profile creator. Given precise specifications in the profile, the Apt infrastructure will attempt to provide such an environment each time. In the absence of such constraints, the system will choose to provision each experiment instance based on whichever testbed resources are available at the time of the request—something clearly beyond the control of the experimenter, and not consistent in the general case. Therefore, it is important to remember that Apt provides merely a mechanism experimenters may exploit to enhance consistency, and not a blanket guarantee of automatic consistency.

Apt is also capable of providing bitwise identical storage state (which is adequate in most cases to yield an identical software environment). It is a simple matter for an Apt user to request a snapshot of a node's filesystem(s), or an entire disk. Therefore, once the SUT and its input data have been prepared, such images can be included in the profile and Apt will take care of software consistency across instantiations. Apt uses the Frisbee [14] disk loader to distribute and install images. Frisbee is fast (generally proceeding at the full write speed of the target disk) and scalable (fully parallelized using IP multicast), ensuring quick instantiation.

Another important element of consistency is the degree to which the experiment environment is isolated from other experiments (or other unrelated activity elsewhere on the network). Clearly, when a SUT includes software running within a VM, there is potential for its performance to be affected by operations outside that VM but residing on the same physical host, and so certain types of consistency are severely limited in those cases.

All Apt nodes offer multiple network interfaces, and the Apt infrastructure allows experimenters to specify topologies of private virtual networks connecting the interfaces of their nodes. The switches and links are conservatively provisioned so that the requested bandwidth is guaranteed, and external influence from any concurrent experiments is minimised. However, all nodes in Apt share a public control network, and the traffic on this network is beyond any individual experimenter's control. While most experiments will be able to carry their critical network traffic on dedicated (private) experimental networks, they should still expect residual effects from the shared control net (for instance, all experimental nodes will generally see broadcast ARP traffic belonging to nodes outside their experiment). Effects like these do place upper bounds on the quality of consistency Apt can promise.

Diversity. The hardware currently available in Apt is quite homogeneous, which is a significant barrier to hardware diversity. (For instance, there are two classes of x86 nodes offered: one type with 8 Xeon E5-2450 cores and 16 GB RAM, and the other with 16 Xeon E5-2650 cores and 64 GB RAM.) This reflects a tradeoff in Apt's design, namely the choice between larger quantities of each of a small number of distinct hardware types or small quantities of each of a large number of types. The former is preferable for availability, and the latter preferable for diversity. During hardware evaluation, diversity support was one factor under consideration, and so support for a variety of experiment types has been added where possible. (For instance, all 192 Apt nodes are connected via a "flexible fabric" where one network interface per node, and its corresponding switch port, can be dynamically configured as either FDR Infiniband, 40 Gbps Ethernet, or non-standard 56 Gbps Ethernet.)

Apt can provide greater software diversity, since there are few *a priori* restrictions compared to those inevitable with hardware. For both physical and VM nodes, profile creators have complete control over software configuration (to the point of providing custom filesystems and kernels if necessary). To some extent, Apt also supports artificial generation of diversity (for instance, hooks are provided where users can specify actions to be performed during resource provisioning, and these could be used to automatically vary parameters of the SUT).

Transparency. Apt makes some attempt to allow the user to discover properties of the SUT beyond their control which might nonetheless affect the outcome of their experiment. The properties the user *can* control are expressed in the RSpec associated with the profile (designated as the *request* RSpec); at profile instantiation time, this request is annotated with extra information associated with the current instance and becomes the *manifest* RSpec. The annotations will include dynamically assigned properties such as IP addresses on public interfaces, and also specify the values chosen when omitted parameters reverted to defaults or were otherwise bound. They also include details of the network, such as the set of switches and links used to connect hosts.

Encapsulation. Apt attempts to support the encapsulation principle; its profiles are intended to map one-to-one with capsules. A profile's RSpec primarily addresses the hardware aspect of a capsule (and to a lesser extent, the software side too), while any referenced disk images complete the specification of the software environment.

Emulab's existing whole-filesystem (and optionally whole-disk) imaging as used by Apt goes a long way toward ensuring that the entire software environment is contained in the capsule. However, there are some instances where per-user software environments (shell, "dotfiles", environment variables, etc.) might vary, and there are documented cases in which certain seemingly innocuous user environment changes (such as the number or size of UNIX environment variables) can have surprising effects on experimental results [18]. In addition, while system-provided disk images are not routinely updated (in general, old versions are left alone and newer versions are added as alternative options), sometimes images might be modified to address critical security vulnerabilities. If a consistent software environment is so critical to an experiment that even security patches should not be applied, then the profile should specify a particular image version to ensure that encapsulation is not violated.

Publishing and Archiving. All profiles created in Apt are given a URL: this URL can be shared via email or the web, published in papers, etc. Apt profiles are *versioned*: the creator of a profile may update it, creating a new version. *Ephemeral* versions (the default) are used for development or debugging. They are typically used only by the original experimenter, and are not guaranteed to be kept for long by the system. When the profile creator is ready to share a profile, he or she *publishes* it, creating a new published version which gets its own unique URL and can thus be shared.[5] Apt keeps all published versions available. A URL may refer to either the profile in general (in which case it resolves to the most recent published version) or to a specific version. When viewing a profile, a user may select any one of the versions for instantiation. Apt recommends that, when publishing results, authors publish the versioned link to the specific version used to gather those results. They may, however, update the profile after publication to fix bugs, add new features or additional analysis, etc. This update does not overwrite the published version, and visitors to the profile have the option of using either the one used for the paper or the updated one.

In order to save space for storing the disk images associated with versioned profiles, Apt has enhanced the Frisbee disk loader [14] to store "base" images and "deltas" containing only changed blocks. We expect to explore the use of deduplicated storage [24] to further reduce the space required to store versioned images.

Adaptability. Apt seeks to make the infrastructure as unobtrusive as possible, and in doing so, to make it relatively easy to use for experimenters from a wide variety of domains. It does take some experience with Apt to *create* a profile, but our goal is to require a minimum amount of experience to *use* one. When following a URL to a profile, a user is asked for only three things: a username, an email address and (optionally) an SSH public key. The process of instantiating a new experiment from a profile typically takes only a few minutes. For the (common) case of profiles containing only a single node, a web-based terminal is automatically opened with a shell on the node. (If the user provides an SSH key, he or she may also log in using a standard SSH client.) Apt profiles are also permitted to run services like webservers to provide an alternate interface to interact with the experiment.

Apt profiles contain instructions (written by the creator) explaining how to run the experiment: these instructions typically point the user to a set of scripts to run and the location of any input and output data. For more complicated multi-node profiles, the instructions may include a "tour": a tour contains a sequence of steps, each of which refers to a node or link in the topology. The typical use of a tour is to explain the different roles of the nodes in the experiment: "This is the traffic source", "this is the traffic sink", etc.

Longevity. While the lifespan of a system in an uncertain environment can be accurately measured only in hindsight, there are some reasons to expect that Apt might offer reasonable longevity for the foreseeable future. Firstly, unlike most clusters dedicated to a particular research project (and destined to be removed or at least deprecated at the conclusion of that project), Apt is developed, maintained and operated by a team of professional staff. Secondly, predeces-

[5] An example of a published profile URL is `https://www.aptlab.net/p/tbres/nsdi14`

sor systems upon which Apt is based have and continue to demonstrate significant longevity in practice. For instance, the original Emulab installation continues to operate, and despite a great deal of evolution in both its hardware and software, is still capable of booting Red Hat Linux 7 on the original 600 MHz Pentium III nodes with which it originated fifteen years ago.

4. RELATED WORK

While Apt represents one way of addressing the needs of repeatability, it is by no means the only possible design for repeatable experimentation. For example, DataMill [22] automatically incorporates diversity into its testing process, and OCCAM [21] supports a workflow customized to computer architecture research. Entrants in the Executable Paper Grand Challenge [8] take other approaches, such as focusing on high-level tools for computational science [19] or using VMs run on user machines to target output, rather than performance, experiments [10].

Even stronger consistency can be obtained by simulating the entire environment [20, 31, 17] rather than executing directly on hardware. Simulation offers alternative models for encapsulation (no hosted facility is required) and longevity (old simulations will continue to work as long as the simulator is maintained). On the other hand, simulators tend to offer lower diversity and be designed only for certain classes of experiments.

Apt's use of disk images for encapsulating software, input data, and configuration maximizes repeatability and consistency, but it does not help experimenters understand how the environment was created. Declarative systems for describing a software environment [5, 23] could provide interesting alternatives or complements to image-based encapsulation. Approaches that involve distributing software environments as virtual machine images or filesystem-level snapshots [12] have the advantage that they do not require expensive hosted infrastructure, but give up performance consistency. Systems designed to capture the process of experimentation itself [7, 16, 11, 25, 28, 13, 1] (running the SUT, collecting, and analyzing results) could also provide a nice complement to Apt; currently, Apt leaves these tasks in the hands of the experimenter.

While Apt can potentially support a great deal of software diversity, its hardware diversity is bound by the limited heterogeneity of the phyiscal installation. Larger scale testbeds, especially federated systems where multiple sites contribute hardware resources independently, are likely to boast greater hardware diversity than any single facility. GENI [2] is an example of a system where experimenters might be able to make use of very diverse hardware, including not only computational and local network resources as in Apt but also wide-area links, mobile nodes, wireless hardware, and other resources beyond the scope of Apt. Many of the techniques Apt exploits to enhance repeatability (such as disk image snapshots, "bare metal" provisioning, and profile versioning) could also be applicable to GENI and similar federated systems.

5. CONCLUSION

In this paper, we make the claim that it is desirable for computer scientists to have repeatable environments in which to conduct their experiments. We have described a set of desirable properties for such an environment, and showed that they are feasible by presenting the design of one system that addresses them.

Regardless of the mechanism used, it seems clear that more resources for repeatable experimentation are required for computer science. We hope that, if they are made simple enough to use, this will encourage more researchers to publish their artifacts along with papers, and will generally increase the quality of published evaluations.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1338155. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

6. REFERENCES

[1] J. Albrecht and D. Y. Huang. Managing distributed applications using Gush. In *Proceedings of the Sixth International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, May 2010.

[2] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5–23, Mar. 2014.

[3] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. Hartman. Stork: Package management for distributed VM environments. In *The 21st Large Installation System Administration Conference (LISA)*, Nov. 2007.

[4] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren. Measuring reproducibility in computer systems research. Technical report, University of Arizona, Mar. 2013.

[5] Docker. `https://www.docker.com/`.

[6] The dpkg package manager. `https://wiki.debian.org/Teams/Dpkg`.

[7] E. Eide, L. Stoller, and J. Lepreau. An experimentation workbench for replayable networking research. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, Oct. 2007.

[8] Elsevier Ltd. The executable paper grand challenge. `http://www.executablepapers.com/`, 2011.

[9] Examining "Reproducibility in Computer Science". `http://cs.brown.edu/~sk/Memos/Examining-Reproducibility/`.

[10] P. V. Gorp and S. Mazanek. SHARE: A web portal for creating and sharing executable research papers. *Procedia Computer Science*, 4(0):589 – 597, 2011.

[11] J. Griffioen, Z. Fei, H. Nasir, X. Wu, J. Reed, and C. Carpenter. Measuring experiments in GENI. *Computer Networks*, 63(0):17 – 32, 2014.

[12] P. J. Guo. CDE: A tool for creating portable experimental software packages. *Computing in Science and Engineering*, 14(4):32–35, 2012.

[13] P. J. Guo and M. Seltzer. Burrito: Wrapping your lab notebook in computational infrastructure. In *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance*. USENIX Association, 2012.

[14] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, June 2003.

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

[16] Labwiki. `http://labwiki.mytestbed.net/`.

[17] J. Liu, S. Mann, N. V. Vorst, and K. Hellman. An open and scalable emulation infrastructure for large-scale real-time network simulations. In *INFOCOM 2007 MiniSymposium*, May 2007.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

[19] P. Nowakowski, E. Ciepiela, D. Haręźlak, J. Kocot, M. Kasztelnik, T. Bartyński, J. Meizner, G. Dyk, and M. Malawski. The collage authoring environment. *Procedia Computer Science*, 4(0):608–617, 2011.

[20] ns-3. `http://www.nsnam.org/`.

[21] OCCAM: Open curation for computer architecture modeling. `http://www.occamportal.org/`.

[22] A. Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 137–149, Prague, Czech Republic, April 2013.

[23] Puppet. `http://puppetlabs.com/`.

[24] S. Qunlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, Jan. 2002.

[25] T. Rakotoarivelo, M. Ott, I. Seskar, and G. Jourjon. OMF: a control and management framework for networking testbeds. In *SOSP Workshop on Real Overlays and Distributed Systems (ROADS)*, Oct. 2009.

[26] Resource specification (RSpec) documents in GENI. `http://groups.geni.net/geni/wiki/GENIExperimenter/RSpecs`.

[27] The RPM package manager. `http://www.rpm.org`.

[28] S. Schwab, B. Wilson, C. Ko, and A. Hussain. SEER: A security experimentation environment for DETER. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, Aug. 2007.

[29] J. Vitek and T. Kalibera. Repeatability, reproducibility and rigor in systems research. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Oct. 2011.

[30] 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 *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*. USENIX, Dec. 2002.

[31] Wind River Systems. Simics full system simulator. `http://www.windriver.com/products/simics/`.