Appeared in *Proceedings of the 11th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, Baltimore, MD, August 2018.

Supporting Docker in Emulab-Based Network Testbeds

David Johnson Elijah Grubb Eric Eide

University of Utah
{johnsond, grubb, eeide}@cs.utah.edu

Abstract

Researchers conduct experiments in a variety of computing environments, including dedicated testbeds and commercial clouds, and they need convenient mechanisms for deploying their software within these disparate platforms. To address this need, we have extended Emulab so that it can instantiate and configure container-based virtual devices using Docker images. Docker is a de facto standard for packaging and deploying software in cloud environments; now, researchers can use Docker to package and deploy software within Emulab-based testbeds as well. We describe how Emulab incorporates Docker and how it extends Docker images to support the interactivity that researchers expect within a testbed. We show that Emulab can use many popular Docker images to create testbed experiments. We expect that Emulab's support for Docker will make it easier for researchers to move their activities freely, both into the testbed and out into production.

1 Introduction

Reliably initializing and configuring devices is essential for conducting repeatable experiments in a testbed. Since its inception, the Emulab testbed-management system [17] has used a custom disk-imaging subsystem for installing operating systems and other software on devices. That subsystem, called Frisbee [11], was built and optimized for distributing disk images within a controlled cluster of machines—and *only* within a cluster. Today, however, many Emulab users rely on multiple platforms to conduct experiments over the course of a full study. In addition to using multiple clusters (e.g., Utah Emulab, CloudLab, and Chameleon)[1] to run tests atop myriad hardware, a cybersecurity researcher might debug an experiment by scaling it down to run in VMs on a laptop, and perform "production testing" by scaling it up for a commercial cloud. An experimenter who moves among multiple platforms needs to deploy software and configure devices in each of these environments—a task that Frisbee was never designed to tackle.

Happily for testbed users, the DevOps community has already met the challenge. Docker [6] is a popular tool that allows a person to package a collection of software into an image and then instantiate virtual devices based on that image. Docker images are portable across multiple infrastructures because they are based on *containers*, which run atop an infrastructure-managed OS kernel and hardware device. The Docker ecosystem includes public repositories of popular "virtual appliances" and tools that allow people to create and publish their own images.

We have extended the Emulab testbed-management software to support the allocation and configuration of container-based devices using Docker images. The main benefit is to make it easier for experimenters to move activities into Emulab-based testbeds—including the Utah Emulab site, CloudLab, PhantomNet, and POWDER[2]—and out to production or other environments. An additional and important benefit is scale. Because containers are a lightweight virtualization technology, testbed users can run experiments that involve large numbers of virtual nodes while using only modest physical resources. Obtaining these benefits was not straightforward. Our goal was to add Docker support to Emulab in a way that preserves both the Emulab features that testbed users expect and the Docker features that users of that ecosystem expect. Smooth integration requires that existing Docker images "just work" as the bases of devices in Emulab. This paper describes how we met this challenge.

We make two contributions. First, we present the design and implementation of Docker support within Emulab. Emulab automatically adapts Docker images for use in a testbed: this allows container-based virtual devices to (1) be "first-class citizens" within experiments and (2) support features that researchers expect of devices within Emulab, such as interactive login. Second, we show that Emulab's Docker support works with existing, third-party Docker images. Emulab can automatically adapt 52 of the 60 most popular images from Docker Hub into the Emulab environment and instantiate containers from them (§5.1). The process also works for images from research projects that we selected as case studies (§5.2). The time to adapt a Docker image for Emulab is suitable for on-demand conversion, and Emulab can quickly create large experiments using Docker images (§5.3).

[1] https://www.emulab.net/, https://cloudlab.us/, https://www.chameleoncloud.org/

[2] https://phantomnet.us/, https://powderwireless.net/

2 Background: Emulab

Emulab is testbed-management software that supports controlled and repeatable experiments in systems, networking, cybersecurity, and other areas of computing. The Utah Emulab testbed manages hundreds of bare-metal servers that have multiple network interfaces to a variety of layer 1 and 2 switches. It provisions subsets of those resources into isolated, emulated networks of physical and virtual nodes for users on demand. Emulab resembles the control software of today's clouds, which provision virtual machines and networks atop physical substrate, although Emulab in general offers more kinds of resources, lower-level resource control, and stronger isolation.

To allocate resources in Emulab, a user writes a *profile*, which is a description of everything needed to build an emulated, networked research environment: physical and/or virtual hardware (servers, storage, switches), network configuration (links and LANs, IP addresses, traffic shaping), and software (operating systems, packages, scripts). The user then instantiates the profile to create an *experiment*. Emulab provisions the experiment with the hardware and software described in the profile. The profile may request specific hardware and configuration, such as requesting specific physical machines, or asserting that some virtual machines should run on specific physical host nodes. Alternatively, the user can leave some or all of the node and network configuration details to Emulab, which will find a best-fit mapping onto its available physical resources. Once an experiment is allocated, its creator has full control over its resources. Users interact with nodes over a remotely accessible *control network*. In addition to simply logging into nodes and running installed software, a user can change software and configuration, capture disk images, change traffic-shaping parameters, take down links, reboot nodes, and so on.

3 Leveraging Docker in a Network Testbed

The core abstraction that a network testbed provides to its users is a collection of physical or virtual *nodes*, connected into networks of *links* and *LANs*. A user provides the testbed with a description of the environment he or she requires: the hardware, network, software, and automation needed to run the user's experiment(s). The testbed maps this description to a subset of its resources, configures them, and makes them available to the user.

Docker enables its users to create, deploy, and manage *containers*, which are process-level "virtual machines" facilitated by the underlying operating system abstractions. A container is an execution environment that sits atop a host's operating system kernel and hardware resources; a collection of processes runs *within* the container. The container determines the filesystems, device interfaces, network stack, users, privileges, and other resources that are visible to the processes running inside it. These may be very different from the resources that are available to processes that run on the host outside of the container; indeed, through the careful selection of resources, containers are generally configured to *isolate* the processes that run within them. Docker deploys containers along with filesystem images: the user-space software running within the container is therefore independent of the software provided by the container's host.

Prior to Docker, a variety of other systems (e.g., `lxc`, OpenVZ, Linux vservers, and FreeBSD jails) provided similar container experiences. The overwhelming success of Docker can be attributed to its overall accessibility to users, especially its image-management toolchain and central, public image repository. Any system that seeks to build on top of Docker, therefore, must be careful to preserve the user experience that makes Docker attractive.

The primary use case envisioned by Docker's developers was not that a Docker container would serve as a kind of node in a testbed. Still, we assert that Docker can be successfully integrated into testbeds, and that it can provide useful provenance and reproducibility features to experimenters. Below, we examine the possibility of utilizing Docker as a type of network testbed node in Emulab and discuss the challenges (C1–C4) we encountered while designing our integration. Our goals were not only to provide a testbed-oriented container orchestration service, but also to allow users to seamlessly use the Docker toolchain in concert with Emulab's services and features.

3.1 Broadening the Process Model (C1)

The primary use case that Docker supports involves a large collection of containers, potentially spanning many machines, in which each container runs a single application (one or more processes launched by a single parent) that provides a specific service (i.e., a microservice architecture). For instance, there are popular Docker images that run `memcached` and `redis`[3] as services. In this model, container images are tailored to support only their single service, unencumbered by unnecessary software or configuration. Many Docker images do not include an `init` daemon to launch and monitor processes, or even a basic set of user-space tools and libraries.

The Docker community discourages whole-OS-like containers that run an `init` and the traditional litany of supporting services, e.g., `syslogd`, `sshd`, and `crond` [9]. The rationale is that these services are not needed in every container: in a large-scale deployment, if a single service container crashes, orchestration software can simply spawn another container to fill the void. Docker 1.13 (API version 1.25) added the ability to run a containerized service automatically from a simple init daemon called

[3] `https://memcached.org/`, `https://redis.io/`

`tini` [13]. However, `tini` only runs a single service process, reaps zombies,[4] and forwards signals received from outside the container to the service process. This is sufficient to protect a poorly written service, but does not support multiple services.

Testbed-based experimentation requires more than the ability to run a single service per node. Within an experiment, it is beneficial for each node to run basic OS services (e.g., `syslogd` and `sshd`) to support activities such as debugging and exploration. It is also beneficial for each node to run a suite of testbed-specific services to configure the node, conduct in-node monitoring, and automate experiment deployment and control (e.g., launch programs or dynamically emulate link failures). This environment requires a full-featured `init` daemon and a common, basic set of user-space software and libraries.

Thus, the first challenge is to support a full-featured container environment with multiple processes and a real `init`, transparent to the user, despite the fact that many existing Docker images do not include such software.

3.2 Onboarding Docker Images (C2)

For a Docker container to be a first-class node within an experiment, Emulab must install a suite of "client-side" tools into the container. These talk to a central configuration server that returns per-node metadata; they create user accounts, install public keys, configure the network, and install and run software. The most straightforward way to install the client-side tools is to add them to the Docker image that is used to create the container.

Docker images are structured in *layers*. This allows many images to share one copy of common content (by sharing layers), and it simplifies image change management (because changes can be isolated to individual layers). Installing Emulab's client-side tools into a Docker image can be accomplished by creating a new image that is like the original but adds a new layer with the tools. This is tricky, however, because the tools must be compiled against the software already installed in the image. The Docker community recommends many best practices for image builders [8], but the most important are (1) minimizing the number of layers and (2) minimizing the size of each layer. To follow these principles, one must avoid introducing unneeded software and layers (e.g., build-time-only dependencies) into the images that are augmented with Emulab's client-side tools. Once a Docker image has been augmented with Emulab's client-side software, the testbed can cache the augmented image.

A final decision in integrating Docker images with an existing testbed lies in simply deciding how users can refer to them. Emulab's image model has long supported versioning and provenance, but images are referenced by unique identifiers, in contrast to the Docker image registry's content-addressed approach. Another difficulty lies in managing permissions, e.g., making sure that a user's Docker images remain private.

In summary, the second challenge lies in managing a testbed-local catalog of Docker images: augmenting images as they enter the catalog, doing that flexibly and at scale, and controlling access to the catalog.

3.3 Addressing the Network Architecture (C3)

Network testbeds and clouds offer similar core orchestration services to allocate and provision nodes and networks for users. Both handle the tasks of mapping a logical network description to physical and virtual resources; allocating node and network resources; creating isolated virtual networks; performing physical- and virtual-node software configuration; and image storage and capture. Both provide virtual network features at layer 2 (e.g., support for bridges, bridge-based firewalls, and layer 2 isolation) and layer 3 (e.g., IP endpoint configuration, DNS, tunnels, and traffic shaping).

Docker's Container Network Model (CNM) [5] describes simple network abstractions that support common, container-based, virtual-network use cases. It describes a model consisting of *sandboxes* (containers and their private network stacks), *endpoints* (the connections between sandboxes and networks), and *networks* (collections of directly communicating endpoints). Although this model is abstract and flexible, it does not specifically model network configuration at different layers of the stack.[5] Instead, the Docker `libnetwork` controller provides de facto configuration patterns: for instance, it provides all networks with a gateway address and assumes that all networks may want interconnectivity via routing at layer 3. Thus, in the Docker distribution, there are no private, layer 2-only subnets whose address ranges could safely be reused across different isolated networks. Because of this, two Emulab experiments cannot both have isolated virtual networks that happen to have the same subnet address on the same machine. This situation might arise if Emulab places the Docker-based nodes of the two experiments on a single *shared* physical host.

The third challenge to using Docker in a network testbed is to merge Docker's simple model of the network with the more complex model provided by the testbed.

3.4 Coordination (C4)

Docker offers limited extensibility to support uniquely featured orchestration engines like testbeds. It provides a limited plugin system (developers can write network-driver, volume-driver, and IPAM plugins), but no support

[4] A *zombie* is an exited child process that a (poorly designed) parent process does not wait for.

[5] Contrast CNM with OpenStack's networking service, which explicitly models well-known objects and their configuration—e.g., networks, subnets, gateways, ports, routers—at different layers of the stack.

for synchronous, per-container, run-time hooks at key life-cycle container events, e.g., startup, shutdown, network interface attach and detach, and snapshot. The ability to hook these events would support extensibility by an orchestration engine like a testbed, whose feature set is more broad than what the single-host Docker daemon provides. For instance, a hook that executes after virtual network device creation, but prior to container boot, could ensure that custom traffic-shaping rules are in place prior to software execution inside the container.

Adding hook points to Docker *for the testbed* would compromise compatibility with Docker *outside the testbed*. The fourth challenge, therefore, is to coordinate the actions of the testbed with the life cycles of Docker containers, without modifying Docker.

4 Docker-Emulab Integration

Our goal is to let Emulab users employ Docker containers as networked, virtual nodes in their experiments, while preserving as much of the Docker user experience as possible. Below, we describe how we met our goal and addressed the challenges discussed in §3.

We integrated Docker into Emulab's virtual-node abstractions: Docker containers are now a *type* of node that a user can request. There were three main integration points: (1) Emulab's profile language, allowing users to request and configure Docker containers as virtual nodes; (2) Emulab's server-side logic, which handles virtual node and network allocation, provisioning, and image hosting; and (3) Emulab's client-side software, which handles container creation, virtual-network configuration, and image retrieval and augmentation. The first two are extensions or implementations of existing abstractions, with the exception of Docker image support within Emulab's server-side software, which we discuss in §4.2. We encountered the greatest difficulties while extending Emulab's client-side abstractions to use Docker to launch containers.

Once Emulab's server-side processes have allocated resources to an experiment, it reboots and reloads physical machines with a disk image that contains Docker. Once each machine boots, Emulab's client-side software configures its user accounts, network interfaces, and software, and finally begins container deployment. When Emulab's client-side software builds the first container on each host during experiment setup, it performs several one-time configuration actions (e.g., configuring Docker's network and storage drivers; allocating all unused disk space to LVM volumes for use by our software and by Docker; configuring the Emulab control network in Docker; etc.). These actions require a global lock to be held; once complete, further container deployment is parallelized.

In our integrated system, Docker containers can be created in *dedicated* or *shared* mode. In dedicated mode, containers run on physical nodes that are reserved to the user's experiment, and the user has root-level access to the underlying physical machine. In shared mode, containers run on physical machines that host containers from potentially many experiments, and users do not have access to the underlying physical machine.

Docker allows containers to be *privileged* or *unprivileged*: a privileged container has administrative access to the underlying host. In our integrated system, only dedicated-mode containers may be privileged. Shared-mode containers must be unprivileged due to the Linux and Docker security models.

4.1 Supporting Multi-process Containers (C1)

To support experimentation, a testbed node must run many processes (§3.1). To achieve this for Docker-based nodes, Emulab modifies Docker images to run an init system rather than a single application. (It installs the init system during *augmentation*, described in §4.2.)

Emulab builds and packages (in temporary containers with a build toolchain) and installs (in the final augmented image) `runit`, a simple, minimal init daemon, and sets it as the augmented image's `ENTRYPOINT`. The `ENTRYPOINT` is the command run by Docker as PID 1 inside the container. Emulab configures `runit` to spawn `sshd`, `syslogd`, and the Emulab client-side services.[6]

Changing the `ENTRYPOINT` means that the new image will not run the command that was specified for the original image. Moreover, `runit` (or the user-specified alternate init) must be run as `root`, but the image creator may have specified a different `USER` for processes that execute in the container. To fix these problems, we emulate the original `ENTRYPOINT` as an `runit` service and handle several related `Dockerfile` settings as well: `COMMAND`, `WORKDIR`, `ENV`, and `USER`. The emulation preserves the semantics of these settings, with the exception that the user-specified `ENTRYPOINT` or `COMMAND` is not executed as PID 1. Only the `ENTRYPOINT` and `COMMAND` processes run as `USER`; processes started from outside the container via "`docker exec`" run as root.

4.2 Onboarding Docker Images (C2)

Naming and storing images. When creating a profile, Emulab users can specify the disk images to be run on each physical or virtual node. To allow Docker containers to be configured in profiles, we extended Emulab's image toolchain to support the Docker image format and deployment mechanisms.

Emulab users refer to images via *imageids*: an imageid can be mapped to a path to the image content and a version. This is a good match with Docker, because one can identify a Docker image with a repository name (e.g.,

[6] If the original Docker image already has an init system, a user can specify that it should be used instead of `runit`. Emulab supports `upstart` and `systemd`.

wordpress) and a tag that is attached to a specific version of the repository content (e.g., 1.0). We adopt a *repository*:*tag* convention for naming Docker images in profiles, and we bind these names when users take Emulab snapshots of Docker containers (i.e., when Emulab effectively runs "docker commit" to capture a new version of the image running in a container).

Traditional Emulab imageids point to the on-disk file(s) that constitute an image, but Emulab Docker imageids point instead to a Docker image inside a private, secure *registry* [10]. The registry supports token-based authentication and authorization. We implemented a secure token endpoint service that complies with Emulab's user authentication and authorization model, and integrates with Emulab's image-deployment mechanism.

Augmenting Docker images. To support standard Emulab features for Docker-based nodes, Emulab adds its client-side software to the Docker image specified by the user. We call this process *augmentation*, and it is entirely automated. If the Docker image is based on Ubuntu, CentOS, Debian, or Alpine, Emulab will automatically build and install the client-side software and its dependencies. This process employs Dockerfile best practices to preserve image provenance and avoid adding unnecessary layers to the image [8]. For instance, the augmentation process builds the Emulab client-side software and runit in temporary containers to avoid polluting the final image with unnecessary build dependencies; it copies only the built artifacts into the final image.

The augmentation process supports four user-selectable levels, each representing a trade-off between increased image size and feature availability:

- *basic:* install sshd, syslogd, runit, and initscripts
- *core:* also install the Emulab client-side software and its run-time dependencies
- *buildenv:* also install the build-time dependencies for the client side
- *full:* also install some network-oriented utilities, to be similar to a typical Emulab VM disk image

These options allow an Emulab user to select the feature set he or she needs.

Emulab also supports the use of unmodified, external Docker images. There are several drawbacks to unaugmented images (§3), but even in this situation, Emulab can offer assistance to experimenters. If a container is running an unaugmented image in dedicated mode, Emulab runs an sshd on the container host that listens on high-numbered ports exposed to the user. That sshd is configured to "docker exec" an interactive shell or command of the user's choosing in the unaugmented container.

4.3 Virtual Networking (C3)

Control network. The Emulab control network allows users to access the nodes in their experiments from the Internet. Physical machines have routable IP addresses, but virtual machines and containers are assigned private, unroutable addresses to support experiments that scale to thousands of virtual nodes. Still, users need to be able to reach their containers via the control network.

Emulab provides this accessibility in three steps. First, when a container host boots for the first time, the Emulab client-side software creates a "dockercnet" network that is bridged to the Emulab control network. We support both macvlan and bridging for this virtual control network. Second, Emulab configures per-node NAT to expose sshd over the container host's public IP address; Emulab's control software also utilizes NAT. Third, Emulab augments Docker's built-in firewalling with its own service.

Isolated layer 2 virtual networks. When a user requests a network (link or LAN) in a profile, Emulab models it as an isolated layer 2 virtual network. Experiment networks may connect to one another via experiment nodes acting as routers, but experiment-network traffic has no default route out of the network.

Because Docker's libnetwork does not provide isolated, unrouted layer 2-only networks, we extended libnetwork and its core bridge and macvlan drivers to support a pure layer 2 mode, where layer 3 features such as a default route via a gateway address are not configured unless requested. We also extended libnetwork's IPAM driver to allow overlapping subnet address ranges to be used in networks that are in layer 2 mode. Because there is no connectivity between these layer 2, gateway-less networks, overlapping subnets on the same physical machine are safe. Our extensions allow Emulab to create virtual networks for separate experiments that involve containers on a shared physical host (i.e., shared-mode containers). (Our modified version of Docker includes Docker's master branch commits up to d4e48af4 on April 30, 2018, and thus supports API v1.37.)

Docker does not support per-container traffic shaping. To configure Emulab's traffic-shaping features if the experiment requires them, Emulab's per-container event listener (§4.4) watches for container creation and installs the necessary tc rules on the container-host side of the virtual interface being shaped. Because shaping is handled outside the container, the container can remain unprivileged (allowing shaping for shared-mode containers) and a user cannot change shaping from inside the container.

Name resolution. Docker assumes that it controls name resolution (/etc/hosts and /etc/resolv.conf) on container hosts, but in Emulab it cannot do so, because Emulab provides its own DNS (which names more than just containers). In Emulab, a node's FQDN maps to its control-network IP address, and *nodename*-*linkname* short names map to the IP addresses of experiment-network (i.e., experiment-internal) endpoints.

In our Emulab-Docker integration, we use Docker's `/etc/hosts` generation facility to populate the short-names map that Emulab would normally configure within the container. Emulab mounts `/etc/resolv.conf` in read-only mode from the host into the container.

4.4 Extending Docker for Emulab (C4)

Our integration extends Docker containers with features not found in Docker itself, such as traffic shaping, without modifying Docker (§3.4). Although Docker does not provide a hook mechanism, it does announce container life-cycle events to listeners. After deploying each container, Emulab subscribes to all container events, such as *start*, *restart*, and *commit*. When a container has been created and is starting or restarting, Emulab applies firewall rules and traffic-shaping rules in the host context, once the container's network devices exist. When a container is shutting down, Emulab removes these rules.

5 Evaluation

To validate our Emulab-Docker integration, we found existing Docker images that are relevant to Emulab users and tested whether Emulab can automatically instantiate nodes using those images (§5.1, §5.2). For each image, we used Emulab's portal (web) interface to instantiate an experiment with a node based on that image, and we tested whether the default process defined by the image's `Dockerfile` was running on the allocated testbed node. To better understand our Emulab-Docker integration, we performed experiments to characterize the augmentation process and overall run-time performance (§5.3).

5.1 Images from Docker Hub

We tested the 60 images registered at Docker Hub that were most popular as of July 4, 2018. The current Emulab Docker system supports images based on Debian 8, 9, and sid; Alpine 3.6, 3.7, and 3.8; Ubuntu 14.04, 16.04, and 18.04; and CentOS 7. This meant that 52 of the 60 images were potentially runnable on our testbed infrastructure. For each remaining image, we created an experiment using the Emulab portal interface and determined if Emulab could automatically run the image.

Our results are summarized in Table 1 and categorized into three possibilities: full support, *partial support*, and ~~not supported~~. We judged that an image was fully supported if, without any additional setup, the image booted using the default `COMMAND` and `ENTRYPOINT` defined in its `Dockerfile`; behaved as it would have, had it been started using typical Docker tooling; and was fully accessible via Emulab's network and monitoring tools. We judged that an image was partially supported if it only booted within Emulab and was accessible via Emulab's online and network interfaces. The only images that we judged not supported were those built from scratch or based on unsupported base-image types.

Linux Distro:	alpine, centos, debian, ubuntu, ~~amazonlinux~~, ~~busybox~~, ~~fedora~~
Debian:	buildpack-deps, cassandra, chronograf, drupal, elasticsearch, ghost, golang, gradle, groovy, haproxy, httpd, influxdb, java, jenkins, jruby, kibana, logstash, mariadb, maven, memcached, mongo, mysql, nextcloud, nginx, node, openjdk, owncloud, percona, perl, php, postgres, python, rabbitmq, redis, rethinkdb, rocket.chat, ruby, sentry, solr, sonarqube, tomcat, wordpress, *telegraf*
Alpine:	consul, docker, kong, neo4j, vault, ~~registry~~
Scratch:	~~hello-world~~, ~~nats~~, ~~swarm~~, ~~traefik~~

Table 1: Images from Docker Hub tested on Emulab. Images are categorized first by image base, then by whether the image is fully supported, *partially supported*, or ~~not supported~~.

Conf.	Software	Base
SNAPL	Everest: verified HTTPS [1]	ubuntu
OSDI	KLEE: symbolic execution engine [3]	ubuntu
PLDI	FunTAL: FL with assembly [14]	debian
S&P	Angr: a binary analysis toolkit [16]	ubuntu

Table 2: Research projects' Docker images tested on Emulab

In some cases, judging the level support was tricky. For example, to be useful, the `wordpress` image needs an external `mysql` instance. While our Emulab Docker integration does not support Docker's `--link` flag for connecting containers, it does support the specification of environment variables, which `wordpress` can use to find its database. Environment variables were also relevant to our judgment that `mariadb` is fully supported. That image requires the user to indicate a default password to successfully run its `ENTRYPOINT`.

The default `COMMAND` and `ENTRYPOINT` of the `haproxy` image fail, but we consider this image to be fully, not partially, supported. This is because starting the image using a typical "`docker run`" command leads to the same result. The documentation for `haproxy` recommends using the image as a base and adding a new layer that injects a configuration file. Running an Emulab experiment with an image built in that way leads to success.

The `telegraf` image has unique semantics and is the only image that we classify as partially supported. We omit details due to space, but the cause of the problem is that Emulab does not support the `--net` option to Docker.

5.2 Images from Research Projects

We also searched for available artifacts from research projects that utilize Docker containers. For all the working artifacts we found, built on base images supported by

Emulab, we were able to use the Docker images to create experiments in the testbed. In addition, for each image, we were able to repeat the tests and results described either in corresponding papers or project websites. Table 2 describes the images we ran using our Emulab-Docker integration. The research artifacts shown represent a broad array of use cases and demonstrate our system's potential value to researchers.

5.3 Benchmarks

We evaluated Emulab's Docker support in two areas: the image augmentation process and the creation of large Docker experiments. Our experiments were performed on CloudLab's xl170 machines (10-core Intel 2.4 GHz E5-2640v4 CPU, 64 GB RAM, one Intel DC S3520 480 GB SSD, two dual-port Mellanox ConnectX-4 25 Gb NIC).

Augmentation. We analyzed the time needed to create Emulab experiments using initially unaugmented Docker images. In these cases, Emulab performs augmentation on demand. We measured the time to augment each of the ten supported base OS images into a new image that supports Emulab's features. We augmented each image at the *core* level (§4.2) and performed our experiments on a *shared* container host running Ubuntu 16.04. Previously, we had deployed the first container on this machine, so its one-time setup tasks had already been performed. We ran each experiment creation three times and computed the averages over the trials.

The results are shown in Table 3. The *Total* column shows the total experiment creation time, from instantiation to successful boot of the augmented image. *Client* shows just the time to configure and deploy a container running the augmented image, and *Aug.* shows just the time to download the base image, analyze it, and build and install its artifacts (e.g. `runit` and the Emulab client-side software) and new packages. The *Size* column shows the augmented image's size, and the *Incr.* column shows the relative size increase over the base image. From our data, we conclude that the time to create experiments from unaugmented Docker images is acceptable for on-demand experiments. (Experiment creation is faster in practice, because Emulab caches augmented Docker images.)

Large-scale experiments. We created large-scale Docker experiments to assess the scalability of our software and illustrate the experience an Emulab user might have. For each experiment, we colocate 200 containers onto each physical host, and vary the number of physical hosts, yielding total container counts between 200 and 5,000. All containers are attached to a layer 2 LAN, and each runs an augmented `ubuntu:14.04` image from Emulab's private Docker registry. The physical host machines run Ubuntu 16.04 with Docker Community Edition v17 (API v1.27). For each physical host count, we conduct three trials and compute average times.

	Time (s)			Image Size	
Image	**Total**	**Client**	**Aug.**	**MB**	**Incr.**
alpine:3.6	144	101	97	274	72.3×
alpine:3.7	144	102	98	276	69.8×
alpine:3.8	144	104	100	279	66.3×
centos:7	205	161	157	303	1.6×
debian:8	244	204	200	254	2.1×
debian:9	163	125	121	225	2.3×
debian:sid	169	127	123	238	2.3×
ubuntu:14.04	332	292	288	272	1.3×
ubuntu:16.04	181	139	135	219	2.0×
ubuntu:18.04	183	142	138	193	2.5×

Table 3: Augmentation results for supported base images

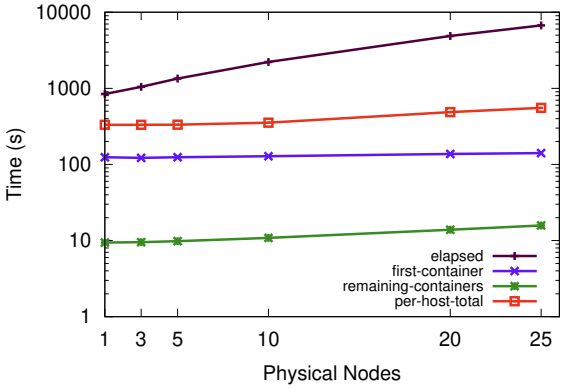

Figure 1: Times related to instantiating Emulab experiments that include many (200–5,000) Docker-based nodes. Each physical machine hosts 200 Docker containers. Note the log-scale y-axis.

Figure 1 shows our key results. The *elapsed* points (top) show the average time from experiment start to successful deployment of all containers. The *per-host-total* data show the average total time each physical host spent deploying its containers. The *first-container* points show the average time required to deploy the first container (which incurs significant one-time physical host configuration). The *remaining-containers* points show the average time spent deploying each of the other 199 containers on each physical host; these deployments are parallelized.

These results show an acceptable level of performance: 14 minutes to deploy a 200-container experiment (including physical-host reboot, reload, and configuration), and 1.87 hours to deploy a 5,000-container experiment. They show reasonable scalability on the container hosts and that more optimization is required on the server side.

6 Related Work

In adding Docker support to Emulab, our goal was to help users move their experiment-related activities among multiple platforms. The Kameleon system by Ruiz et al. [15] has the same goal but takes an approach different from ours. We have extended Emulab to work with Docker im-

ages directly. In contrast, Kameleon is a "software appliance builder" than inputs a specification and produces images for multiple environments including Docker, QEMU, and the custom format used by the Grid'5000 testbed. Kameleon offers the advantage of more explicit specifications, which may aid reproducibility; our approach has the advantage of stronger integration with existing artifacts and ecosystems.

A secondary but important benefit of using Docker containers in Emulab is scale, i.e., the ability to run experiments that include greater numbers of virtual nodes. As described by Wroclawski et al. [18], the DETERLab testbed supports flexible scaling through its container system: a DETERLab user can instantiate a single experiment specification in multiple ways—where nodes are realized via physical devices, VMs, containers, processes, or even threads—and thus trade-off fidelity and scale. This is clearly a more dynamic range than that of the work presented in this paper, but the primary focus of Emulab's Docker support is not on scale, but on artifact portability.

Others have noted the usefulness of Docker for supporting repeatable experiments in computer science [2, 4]. By adopting Docker as a node-imaging alternative for Emulab, we aim to benefit from the Docker ecosystem and support the reuse of Docker-based research artifacts within Emulab-based testbeds.

One can think of Emulab as a kind of orchestrator. Container orchestrators for cloud environments, such as Kubernetes [12] and Docker Swarm [7], offer features that are useful for maintaining applications: e.g., dynamic load balancing, scaling, and monitoring. In contrast, Emulab focuses on orchestration at experiment creation (e.g., node placement and configuration) and then gets out of the way so that researchers can conduct their experiments.

7 Conclusion

Our position is that experimenters need ways to transport their research artifacts between the many environments they may use over the lifetime of a study: from inception on a laptop, to evaluation on a testbed, to initial deployment in a cloud. To streamline this migration, we have extended the Emulab testbed software to support container-based devices using Docker images. We have shown that Emulab can automatically adapt many popular and research-derived Docker images to the testbed environment, enabling migration while preserving testbed features such as interactive experimentation.

Artifact sharing statement. Our support for Docker containers in Emulab-based testbeds is part of the Emulab open-source software, available at `https://gitlab.flux.utah.edu/emulab/emulab-devel`.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant Number 1513121.

References

[1] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hriţcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. In *Proc. SNAPL*, pages 1:1–1:12, May 2017. doi: 10.4230/LIPIcs.SNAPL.2017.1.

[2] C. Boettiger. An introduction to Docker for reproducible research. *OSR*, 49(1):71–79, Jan. 2015. doi: 10.1145/2723872.2723882.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, Dec. 2008. URL `https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

[4] J. Cito and H. C. Gall. Using Docker containers to improve reproducibility in software engineering research. In *Proc. ICSE Companion*, pages 906–907, May 2016. doi: 10.1145/2889160.2891057.

[5] Docker Inc. Container Network Model, Oct. 2016. URL `https://github.com/docker/libnetwork/blob/master/docs/design.md`.

[6] Docker Inc. Docker Community Edition (CE) web site, 2018. URL `https://www.docker.com/community-edition`.

[7] Docker Inc. Swarm mode overview, 2018. URL `https://docs.docker.com/engine/swarm/`.

[8] Docker Inc. Best practices for writing Dockerfiles — Docker Documentation, 2018. URL `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.

[9] Docker Inc. Run multiple services in a container — Docker Documentation, 2018. URL `https://docs.docker.com/config/containers/multi-service_container/`.

[10] Docker Inc. Docker Registry — Docker Documentation, 2018. URL `https://docs.docker.com/registry/`.

[11] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proc. USENIX ATC*, pages 283–296, June 2003. URL `https://www.usenix.org/legacy/events/usenix2003/tech/hibler.html`.

[12] The Kubernetes Authors. Kubernetes web site, 2018. URL `https://kubernetes.io/`.

[13] T. Orozco. Tini — a tiny but valid `init` for containers, Apr. 2018. URL `https://github.com/krallin/tini`.

[14] D. Patterson, J. Perconti, C. Dimoulas, and A. Ahmed. FunTAL: Reasonably mixing a functional language with assembly. In *Proc. PLDI*, pages 495–509, June 2017. doi: 10.1145/3062341.3062347.

[15] C. Ruiz, S. Harrache, M. Mercier, and O. Richard. Reconstructable software appliances with Kameleon. *OSR*, 49(1):80–89, Jan. 2015. doi: 10.1145/2723872.2723883.

[16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (State of) The art of war: Offensive techniques in binary analysis. In *Proc. IEEE S&P*, pages 138–157, May 2016. doi: 10.1109/SP.2016.17.

[17] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002. URL `https://www.usenix.org/legacy/event/osdi02/tech/white.html`.

[18] J. Wroclawski, T. Benzel, J. Blythe, T. Faber, A. Hussain, J. Mirkovic, and S. Schwab. DETERLab and the DETER project. In R. McGeer, M. Berman, C. Elloitt, and R. Ricci, editors, *The GENI Book*, pages 35–62. Springer, 2016. doi: 10.1007/978-3-319-33769-2_3.