Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC

Reid Priedhorsky and Tim Randles
{reidpr,trandles}@lanl.gov
Los Alamos National Laboratory
High Performance Computing Division
Los Alamos, NM, USA

ABSTRACT

Supercomputing centers are seeing increasing demand for *user-defined software stacks* (UDSS), instead of or in addition to the stack provided by the center. These UDSS support user needs such as complex dependencies or build requirements, externally required configurations, portability, and consistency. The challenge for centers is to provide these services in a usable manner while minimizing the risks: security, support burden, missing functionality, and performance. We present Charliecloud, which uses the Linux user and mount namespaces to run industry-standard Docker containers with no privileged operations or daemons on center resources. Our simple approach avoids most security risks while maintaining access to the performance and functionality already on offer, doing so in just 800 lines of code. Charliecloud promises to bring an industry-standard UDSS user workflow to existing, minimally altered HPC resources.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Security and privacy → Operating systems security**; • **Software and its engineering → Process management**;

KEYWORDS

containers, user environments, least privilege

ACM Reference format:
Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 10 pages.
https://doi.org/10.1145/3126908.3126925

1 INTRODUCTION

HPC users have always been asking for more, better, and different software environments to support their scientific codes. "Bring your own software stack" functionality, which we call *user-defined software stacks* (UDSS),[1] is motivated by user needs such as:

(N1) *Software dependencies* that are numerous, complex, unusual, differently configured, or simply newer or older than what is already provided.
(N2) *Build-time requirements* unavailable within the center, such as relatively unfettered internet access.
(N3) *Validated software stacks* and configuration to meet the standards of a particular field of inquiry.
(N4) *Portability* of environments between resources, including workstations and other test/development systems not managed by the center.
(N5) *Consistent environments* that can be easily, reliably, and verifiably reproduced in the future.
(N6) *Usability* and comprehensibility.

A further motivation is that HPC users see the exciting things happening in the outside world and want similar functionality. In our case, this is public compute clouds such as Amazon's AWS,[2] Google's Compute Engine,[3] and Microsoft's Azure,[4] as well as open-source image and container management tools such as Docker.[5]

Complicating the matter are several potential drawbacks and difficulties of UDSS:

(D1) *Security risks.* By introducing very flexible new features, UDSS can expand a center's attack surface.
(D2) *Support burden.* Again due to sharply increased flexibility, UDSS may increase the cost of system configuration and/or user support.
(D3) *Missing functionality.* Because a UDSS is by definition separated from the native software stack, implementations can limit features such as accelerator hardware, filesystems, and high-speed interconnects. This is important because it is these high-end features that make supercomputing centers, as opposed to business data centers, interesting and special.
(D4) *Performance penalty.* UDSS implementations can introduce various type of overhead that negatively impact performance.

These motivations and potential pitfalls suggest three design goals for an HPC-focused UDSS implementation:

(G1) Provide a standard, inter-operable, and reproducible workflow. (all N, D2)
(G2) Run on existing, minimally modified HPC hardware and software resources. (all D)
(G3) Be very simple. (N6, D1, D2)

[1] No consensus vocabulary for this or related concepts exists. Alternate terms are *flexible stacks*, *flexible environments*, *user-defined environments*, and *user-defined images*, and others.

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11.
https://doi.org/10.1145/3126908.3126925

[2] https://aws.amazon.com
[3] https://cloud.google.com/compute/
[4] https://azure.microsoft.com
[5] https://www.docker.com

In this paper, we introduce Charliecloud, a UDSS implementation targeted at these design goals. Charliecloud provides an industry-standard, reproducible workflow based on Docker, and its user namespace-based implementation eliminates the need for privileged or trusted operations on any center-managed resources.[6] It is a simple, open-source[7] implementation of roughly 500 lines of C and 300 lines of shell code.

The remainder of this paper is organized as follows. First, we outline how Linux containers work and their relationship with the UDSS concept. Next, we describe Charliecloud and how it meets the design goals, followed by a comparison to other products. We close with the implications of Charliecloud and possible future work.

2 LINUX CONTAINERS OVERVIEW

In this section, we briefly review Linux containers and key related concepts upon which Charliecloud is built. We do so in part for clarity: *container* is a widely used term with varying definitions, so this section outlines precisely the assumptions our argument depends on.

2.1 Privileged Linux namespaces

Linux has six *namespaces* that isolate different classes of kernel resources; a process and its children see a set of these resources independent from other processes [16, 21]. Five are what we call *privileged namespaces*, requiring root privileges to create; the sixth, unprivileged namespace is covered in the next section. The privileged namespaces are:

(1) *mount*: Filesystem tree and mounts.
(2) *PID*: Process IDs. A process inside a PID namespace has different PIDs depending on whether it is being viewed from inside or outside the namespace.
(3) *UTS*: Host name and domain name. (The name derives from "UNIX Time-sharing System".)
(4) *network*: All other network-related resources, including network devices, ports, routing tables, and firewall rules.
(5) *IPC*: Inter-process communication resources, both System V and POSIX.

The six namespaces can be mixed and matched, but there are quirks.[8] For example, a mount namespace cannot create a new `/sys` unless it is also a network namespace, because `/sys` includes files that can be used to manipulate the network configuration.

Namespaces are always active, i.e., all Linux processes have namespace IDs for all six namespaces,[9] and they can be nested. Everything is owned by a namespace. For example, though it cannot create its own, a mount namespace can bind-mount its parent's `/sys`, to which the parent namespace controls access.

They are manipulated by three system calls. `unshare(2)` creates and joins new namespaces [23], `clone(2)` creates a child process and can put it in new namespaces (the caller's namespaces are unchanged) [20], and `setns(2)` joins an existing namespace [22].

[6] A setuid mode that does not require user namespaces is also provided, to allow testing on systems that do not support them.
[7] `https://github.com/hpc/charliecloud`
[8] Many of these quirks are not documented and must be explained elsewhere, often the Linux source code itself. All quirks we encountered are documented in the Charliecloud source code
[9] Try `ls -l /proc/self/ns`.

These features are useful for UDSS because they allow a guest filesystem tree installed in a directory of the host to become the root of a mount namespace, and the other namespaces can be used for additional isolation as needed.

2.2 Unprivileged namespaces

The sixth namespace, *user*, was added in Linux 3.8 and the following few releases. Its goal is to give unprivileged processes access to traditionally privileged functionality in specific contexts when doing so is safe [17, 24].

The first process in a new user namespace has all capabilities in the new namespace, but none in the parent user namespace, even if created by root. Thus, processes and kernel resources inside the user namespace can be manipulated arbitrarily, but only in ways that do not affect the parent namespace (i.e, the host) — privilege is an illusion.

Another key component of user namespaces is the UID and GID mappings. Part of creating a user namespace is to define a one-to-one mapping between UIDs and GIDs in the parent namespace and the new child user namespace. (The procedure is the same for both, so we omit discussion of GID mapping for brevity.) A common use is to map one's normal, unprivileged UID to 0 inside the namespace, thus appearing to be root.

For unprivileged users, these maps are not arbitrary: unprivileged processes may map only their EUID. This limits access to things already accessible, because while any UID can be selected in the user namespace, it must map to the user's existing host UID. Also, all access using unmapped UIDs will be rejected. For example, `setuid(2)` cannot be used to access another user's files, because the other user's UID grants no access if unmapped and cannot be set on the host side of the map.

This mapping is used to translate UIDs in both directions. When a UID-based access decision is initiated inside the namespace, the map translates the child namespace UID to its corresponding parent namespace UID, and the latter is used for access control. For example, bind-mounting any directory into the container is safe, because it is the user's real, unprivileged IDs on the host, not the fictional ones in the user namespace, that control access. In the opposite direction, for example, files owned by the user will be translated from the user's real UID to the in-container UID. Thus, with the mapping to UID 0 described above, all of a user's files will appear to be owned by root when listed inside the namespace.

Figure 1 illustrates a hello-world user namespace implementation. This is an unprivileged, untrusted, non-setuid program; given kernel support, any user can run it, or the more complete implementation in Charliecloud, with no sysadmin assistance.

User namespaces are a powerful tool for implementing container-based UDSS tools because they let a normal, unprivileged user create an independent filesystem tree and safely access host resources, even if they hold "privileges" inside the container.

2.3 cgroups

Linux cgroups are a mechanism for limiting resource consumption of processes [33]. This is most useful in a multi-tenant setting, i.e., multiple users running jobs on the same node. In a single-tenant

```
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
   uid_t euid = geteuid();
   int fd;

   printf("outside userns, uid=%d\n", euid);

   unshare(CLONE_NEWUSER);
   fd = open("/proc/self/uid_map", O_WRONLY);
   dprintf(fd, "0 %d 1\n", euid);
   close(fd);
   printf("in userns, uid=%d\n", geteuid());

   execlp("/bin/bash", "bash", NULL);
}
```

Figure 1: Hello world implementation of a user namespace, available as `examples/syscalls/userns.c` in the Charliecloud source code. This program creates the namespace with `unshare(2)`, maps within-namespace UID 0 to the invoking user's EUID by writing `uid_map`, and then starts the world's most useless root shell.

setting, excess resource consumption will adversely affect only the original user.

There are further reasons why cgroups might be desirable in HPC. For example, CPU pinning can help performance and make NUMA behavior deterministic. However, cgroups can be applied to any process; we consider them orthogonal to containers.

2.4 Vocabulary

With the key concepts explained, we now list the vocabulary used in this paper.

A *user-defined software stack* (UDSS) is a software environment provided by a user to support their HPC code(s). Typically, this environment is reasonably complex, i.e., more than a library or three. The acronym *UDSS* is both singular and plural.

An *image* embodies a UDSS. It can be a tarball, a directory tree, a filesystem image, or something else. Images can be *activated* or *run*, but UDSS cannot.

Host refers either to a physical node or the bare-metal operating system running on it.

A *guest* is a subordinate OS running on a host with some mechanism for isolation from that host. This term is most commonly used in the context of virtual machines, but we use it in a technology-agnostic way. *Activating* an image doesn't necessarily require a guest, e.g. with `LD_LIBRARY_PATH`, but *running* an image does. Note that *guest* does not refer to the virtual machine emulator or hypervisor itself.

A *container* is a guest-host embodiment that uses kernel isolation mechanisms, e.g. as opposed to a virtual machine, which uses emulated hardware. That is, guests share the host's kernel but have an independent filesystem root; other kernel resources may or may not be shared. The isolation is irreversible, even by root, in contrast with `chroot(2)` [36].

In the case of Linux, containerization requires the mount namespace. The other five are optional, as are cgroups.

2.5 Charliecloud containers

A key design goal of Charliecloud is simplicity (G3). This invokes the principle of least privilege [35] and the UNIX philosophy to "make each program do one thing well" [28].

Under this guidance, Charliecloud keeps all processes on center-owned resources unprivileged, leveraging user namespaces to access a few important system calls. Thus, isolation becomes a tool for functionality, to let users run their codes, rather than a security boundary. Adding the mount namespace lets users provide their UDSS, completing the picture.

This approach contrasts with other container implementations, which run UDSS with a great number of features, options, indirections, security boundaries, and lines of code. We argue that this is unnecessary and needlessly complicates the issue. Running a UDSS is straightforward, not heroic.

With respect to the other options outlined above:

- cgroups are not needed because Charliecloud is agnostic to single- vs. multi-tenant policy. These needs are no different from non-container jobs and are managed orthogonally with separate tools.
- The PID namespace is not needed because we do not need to hide host processes from the UDSS. It also has a key quirk: unlike the other namespaces, `unshare(2)` does not enter the new PID namespace; instead, a subsequent `fork(2)` creates PID 1 in the new namespace, and further `fork(2)`s fail with `EPERM` [18]. Thus, the `fork(2)` + `exec(2)` model must be used, instead of the simpler `exec(2)` model, and a supervisor process is needed to proxy signals.
- Instead of the UTS and network namespaces, Charliecloud relies directly on the host network stack. In addition to avoiding indirection that increases complexity and can reduce performance, this has had specific security impact: Charliecloud was not vulnerable to a recent privilege escalation involving combined network and user namespaces [9].
- The IPC namespace is also not used, which lets different containers and the host communicate using shared memory. This simplifies many workflows.

In the rest of the paper, we argue that this simple approach is the best way to meet the proposed design goals.

3 CHARLIECLOUD AND THE DESIGN GOALS

Charliecloud is a lightweight, open-source UDSS implementation based on the Linux user namespace and targeted to the design goals outlined above. It uses Docker to build a UDSS image, shell scripts to unpack the image to an appropriate location, and a C program to activate the image and run user code within. Selected host directories and files are bind-mounted into the running image to provide access to existing functionality.

This section describes how Charliecloud meets the design goals introduced above.

3.1 Standard, reproducible workflow (G1)

Here, we describe the Charliecloud workflow, using an MPI[10] "hello world" program.[11] It is presented in full in order to demonstrate the simplicity of the Charliecloud model.

We selected Docker for image building because it is an industry standard and because it provides a well-defined, reproducible procedure for defining, composing, and sharing[12] images. However, we found Docker's security posture and complexity undesirable for running images, using instead a simple C wrapper we wrote ourselves.

While running the Docker daemon and executing Docker commands require privileged access, this happens on user-managed resources; nothing in the Charliecloud workflow requires privileged or trusted processes or daemons on center-managed resources. All privileged steps take place on user systems, and the scripts escalate with `sudo` as needed.

Leveraging Docker in this way enables participation in its larger image ecosystem. While the example below builds a custom image, one could instead use a pre-built one obtained with `docker pull`. Similarly, images once built can be shared with `docker push`.

We tested this workflow with OpenMPI 1.10.5 on Debian Stretch with vendor kernel 4.9, RHEL 7 with upstream kernel 4.4, and other distributions and versions.

3.1.1 Define UDSS. (Unprivileged; user-managed system.) The first step in the workflow is to write a Dockerfile that defines the necessary environment; this is illustrated in Figure 2. Any guest Linux distribution compatible with the host kernel can be used.

Because Charliecloud can mount host directories in the container, users can choose whether to install their application inside the container or read it from the host at runtime; the former is more portable and the latter more flexible.

3.1.2 Build Docker image. (Privileged; user-managed.) Next, we build the Docker image with `ch-build`, a convenience wrapper around the privileged `docker build` command that manages HTTP proxy variables and helps find the proper Dockerfile. The two arguments are a tag for the image and the Docker *context directory*, in this case the Charliecloud source code.

```
$ cd examples/mpihello
$ ch-build -t hello ../..
[...]
Successfully built 0f90d6ba020b
```

3.1.3 Examine image. (Privileged; user-managed; optional.) The Docker image can be examined with standard Docker commands. By default, the environment is different than what is provided by Charliecloud, because no host directories are passed through and the commands run as host root.

```
$ sudo docker run -it hello /bin/bash
# cd /hello
# ls hello*
hello  hello.c
# exit
```

3.1.4 Flatten image. (Privileged; user-managed.) In this step, we convert the Docker image to a tarball; it is then a regular file and thus easy to manage. `ch-docker2tar` wraps the privileged `docker export` command.

```
$ ch-docker2tar hello /tmp
[...]
131M /tmp/hello.tar.gz
```

3.1.5 Distribute tarball. (Unprivileged; both user- and center-managed.) Thus far, the workflow has taken place on user-owned systems. Now, we must copy the tarball to the center-owned system where it will be deployed, for example with `scp` or similar to a staging area on the cluster front-end.

3.1.6 Unpack tarball. (Unprivileged; center-managed.) Charliecloud provides a wrapper to `tar xf` that includes a few sanity and convenience checks.

```
$ ch-tar2dir /tmp/hello.tar.gz /img/hello
/img/hello unpacked ok
```

Note that because `tar` is running unprivileged, potentially malicious files such as devices cannot be created, even if they are in the tarball.

Typical unpacking destinations include `tmpfs`, compute node local storage, or a shared scratch filesystem, perhaps leveraging the trend toward close-to-node storage such as burst buffers [26]. Unpacking can take place during job allocation time or as the first step in the user's job script.

3.1.7 Run user program. (Unprivileged; center-managed.) We now run the actual user code. This is done with the `ch-run` C program, which sets up the namespaces, bind-mounts host directories, changes the container root directory to the user image using `pivot_root(2)` — a privileged system call made available to an unprivileged process by the user namespace — and replaces itself with the user command using `execvp(2)`.

```
$ stat -L --format='%i' /proc/self/ns/user
4026531837
$ mpirun -n 4 ch-run /img/hello /hello/hello
0: init ok, 4 ranks, userns 4026532257
1: init ok, 4 ranks, userns 4026532268
2: init ok, 4 ranks, userns 4026532270
3: init ok, 4 ranks, userns 4026532272
0: send/receive ok
0: finalize ok
```

The `stat` command identifies the parent (host) user namespace by reading the inode number of `/proc/self/ns/user`. The arguments to `ch-run` are the directory containing the UDSS image and the command to run within the container. In this example, `mpirun` is running on the host, and the worker processes are in separate containers.

[10] We chose MPI for this exposition due to its familiarity among the HPC target audience. Because it is already dominant in HPC, however, we expect that actual demand for MPI UDSS will modest compared to more esoteric frameworks.
[11] Available at `examples/mpihello` in the Charliecloud source code.
[12] For example, via Docker Hub: `https://hub.docker.com`

```
FROM debian:jessie

# OS packages needed to build OpenMPI.
RUN    apt-get update && apt-get install -y g++ gcc make wget \
    && rm -rf /var/lib/apt/lists/*

# Compile OpenMPI.
ENV VERSION 1.10.5
RUN wget -nv https://www.open-mpi.org/software/ompi/v1.10/downloads/openmpi-${VERSION}.tar.gz
RUN tar xf openmpi-${VERSION}.tar.gz
RUN    cd openmpi-${VERSION} \
    && CFLAGS=-O3 CXXFLAGS=-O3 \
       ./configure --prefix=/usr --sysconfdir=/mnt/0 \
                   --disable-pty-support --disable-mpi-cxx --disable-mpi-fortran \
    && make -j$(getconf _NPROCESSORS_ONLN) install
RUN rm -Rf openmpi-${VERSION}*

# This example
COPY examples/mpihello /hello
WORKDIR /hello
RUN make clean && make
```

Figure 2: Dockerfile for the MPI hello world program. Starting with the latest version of Debian Jessie, this installs the necessary compilers and support packages from the distribution repository and then compiles OpenMPI from source. The demo program is then copied into the image from the Charliecloud source code and compiled.

Because this approach uses the host's MPI infrastructure, it leverages existing configuration work to scale job launch appropriately for the resource.

3.2 Use existing HPC resources well (G2)

In this section, we outline potential drawbacks of introducing Charliecloud and how they are mitigated.

3.2.1 Security risks (D1). Charliecloud relies on two things to maintain security. First, the Linux kernel to enforce access control and other aspects of security. This is a well-accepted approach in HPC; the extension — that user namespaces will ensure that guest UIDs are an illusion — is relatively minor and exercises some new paths through kernel code. User namespaces have been available since kernel version 3.8, released February 18, 2013 [3], and thus have had four years to mature as of this writing.

For example, one cannot use Charliecloud to obtain a copy of memory, despite being guest root:

```
$ ch-run /img/hello --uid 0 -- id -u
0
$ ch-run /img/hello --uid 0 -- whoami
root
$ ch-run /img/hello --uid 0 \
  -- dd if=/dev/mem of=/tmp/pwned
[...] '/dev/mem': Permission denied
```

Second, because all Charliecloud operations on center-owned resources are done as the invoking, unprivileged user, there are no opportunities for shenanigans such as creating device files or setuid binaries. This is true despite the user-selected UID and GID within the container; recall that the kernel's user namespace code maps these container IDs back to the user's real host IDs before performing access checks.

We have validated the security of Charliecloud, on the same systems as described in §3.1 above, by attempting to:

- Use the standard `chroot(2)` escape [36] to change the guest filesystem root.
- Read inaccessible files in `/dev`, `/proc`, and `/sys`.
- Bypass file and directory permissions (we enumerated 2,881 modes).
- Create device files on all mounted filesystems.
- Bind to privileged ports on all host IP addresses.
- Re-mount the host's root filesystem.
- Change supplemental groups with `setgroups(2)`.
- `seteuid(2)` to an unmapped UID.
- Send a signal to a process owned by a different user.

This test suite, which is included in the source code, identified no privileged functionality that could be accessed.

3.2.2 Support burden (D2). Possibly-increased support requirements come from two sources and are modest. First, Charliecloud itself must be supported. Second, users must avoid doing things that are sub-optimal, counterproductive, or troublesome for others, e.g., configuring MPI to use the management network instead of the HSN. Such issues of course happen today; however, because installing software becomes easier, they may be more common.

Aside from this, support for Charliecloud containers is scalable. A center can choose to provide as little or as much support for Charliecloud UDSS content as it likes. We expect a learning curve here, because the UDSS is somewhat opaque, e.g., will staff need to execute an image to diagnose its problems? In this regard, however, Charliecloud is more accessible than other UDSS solutions because its images are plain directories rather than filesystem image files.

```
$ ch-run -d /data /img/hello \
  -- findmnt -R -o fstype,target
FSTYPE      TARGET
tmpfs       /
tmpfs       |-/run
tmpfs       |-/mnt
btrfs       | `-/mnt/0
tmpfs       |-/home
ext4        | `-/home/aturing
devtmpfs    |-/dev
ext4        |-/etc/passwd
ext4        |-/etc/group
ext4        |-/etc/hosts
proc        |-/proc
sysfs       |-/sys
tmpfs       `-/tmp
```

Figure 3: Simplified mount tree for a Charliecloud container. This illustrates the container's root filesystem (`/`), various `tmpfs` local to the container (`/run`, `/mnt`, `/home`), default bind mounts from the host (`/home/aturing`, `/dev`, `/etc/*`, `/proc`, `/sys`, `/tmp`), and a user-specified custom mount (host `/data` is mounted at `/mnt/0`).

On the other hand, Charliecloud's customization tools may reduce support burdens as well. For example, center staff could provision a lighter-weight underlying cluster image if users bring their own environments.

3.2.3 Missing functionality (D3). We identify three reasons a UDSS might lose functionality:

(1) *Devices and filesystems*, such as GPUs and parallel scratch. Charliecloud solves this by bind-mounting key host directories and files into the container, including `/dev`, along with additional user-specified directories, thus inheriting the host's access credentials. This is illustrated in Figure 3.
(2) *User-space drivers* needed for hardware such as GPUs and high-speed networks. The solution is to make the library files available inside this guest. This can be done in multiple ways, including installing distribution-provided drivers, center-provided base images, and bind mounts. Frequently, driver versions need not match what is installed on the host; for example, OpenMPI works this way, because only the worker processes inside the container communicate over the high-speed network.
(3) *Configuration* specific to the host. One way to address this is to bind-mount configuration directories into the guest, which is why Charliecloud bind-mounts the user's home directory by default. Another is by using frameworks that read the host configuration outside the container and pass it to the guest workers, as `mpirun` does above. (Recall that because the UTS and network namespaces are not used, network-related settings such as host names and IP addresses are valid for the guest as well.)

In short, Charliecloud's simple approach and minimal isolation mean that missing-functionality risks are low, and issues that do arise can be straightforwardly solved.

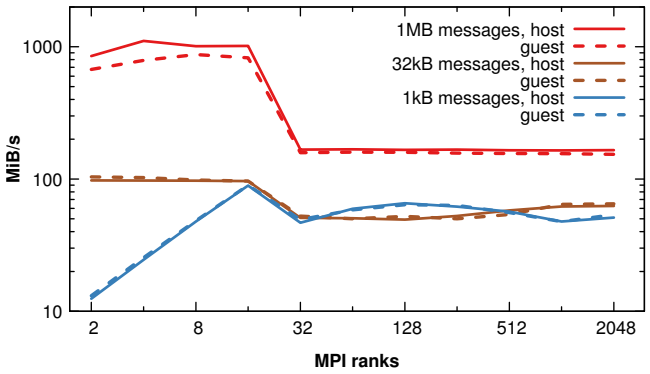

Figure 4: Parallel transfer benchmark *sendrecv* shows no systematic performance penalty at the 1 kB and 32 kB message sizes. At the 1 MB message size, the cost is roughly 5% once the problem grows to multiple nodes; we suspect this is due to tuning differences rather than containers.

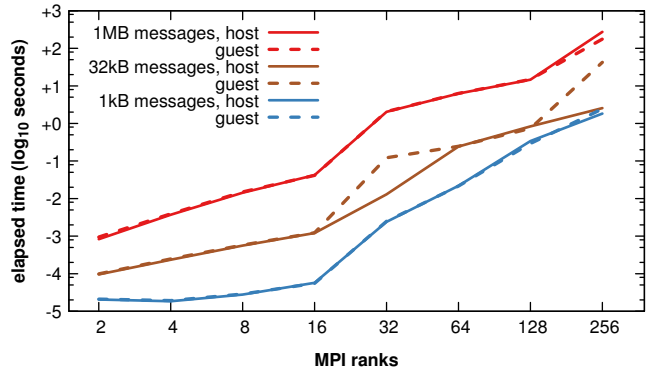

Figure 5: Collective benchmark *alltoall* shows no systematic performance penalty. (We suspect the glitches at 32 kB for 32 and 256 ranks, as well as 1 MB/256, are due to network, not container, issues.)

3.2.4 Performance (D4). We expected Charliecloud containers to impose minimal performance penalty, because the guest is using the same kernel and devices as the host. In particular, we expect minimal network overhead because we share the network namespace with the host, in contrast to prior work that showed substantial costs for indirect network access [34]. These expectations are consistent with prior work that showed minimal overhead in a lightweight container setting with direct network access [40].

To evaluate this expectation, we ran two Intel® MPI Benchmarks v2017 [11] under OpenMPI 1.10.5 on one of our Ethernet clusters with 16 cores per node. The host OpenMPI build was configured by one of our professional sysadmins, while the guest OpenMPI was built naïvely as shown in Figure 2. The guest run used `mpirun` and `orted` running on the host and one benchmark worker process per container (as in the `mpihello` example above).

The results, as illustrated in Figures 4 and 5, supported our hypothesis. In all but one test, there was no observable container penalty. The last stabilized at ~5% cost; we suspect this modest

penalty is due to configuration or compiler and was not inherent to the containerization.

3.3 Be very simple (G3)

Charliecloud is 500 lines of C code and 300 lines of shell script — much less than the next-smallest alternative. It has minimal dependencies: recent Linux kernel with headers, the C standard library, and a POSIX shell and utilities. There are no daemons, whether privileged, trusted, or otherwise; in particular, there is no container supervisor process, and there is no cache, persistence, or configuration files. Charliecloud requires no network infrastructure such as bridges or routing table modifications. Images can be manipulated with standard commands or any general data staging service. This simplicity makes Charliecloud easier to understand, debug, deploy, and support.

4 RELATED WORK

This section outlines the context that Charliecloud arises from and how it advances the state of the art. Table 1 summarizes this comparison against key alternatives.

4.1 Admin-managed customization

Customized software, i.e., not provided by the base OS, is a long-standing need in HPC. Traditionally, this has been fulfilled by system administrators installing various software upon user request; users then select at run-time which options they prefer. Here, we briefly review two of the many options for accomplishing this.

Environment modules [8, e.g.] are a widely installed class of tools. Administrators install applications and libraries into globally accessible directories and write *modulefiles* to manage the environment, e.g., by setting variables such as `PATH` and `LD_LIBRARY_PATH`. Users then activate and deactivate specific environments using the command line tool `module`, either interactively or in job scripts.

With CHOS[13] [30], created at NERSC and in production there since 2004, administrators install a complete Linux distribution into a directory. The user-facing tools then activate this OS using `chroot(2)` and a kernel module to manage "magic symlinks" into CHOS directories that change based on a process' PID.

These approaches have advantages; they do bring users critical flexibility, and software is installed by experts. On the other hand, because sysadmins do the installation and maintenance work, only software that has high demand justifies this effort; unusual software needs, whether innovative or crackpot, are unmet. This can create a chicken-and-egg problem: a package has low demand because it's unavailable and it's unavailable because there is low demand.

4.2 Self-compile

The traditional method for users to take care of themselves is what we call *self-compile*: download, compile, and install software into a home directory or other user area.

This is available almost everywhere already; it consumes minimal sysadmin effort (help users do it, and provide infrastructure such as compilers and base libraries); by definition, it does not employ any privileged operations anywhere; and the resulting software has direct access to all center resources.

However, it has disadvantages: it is tedious, error-prone, and hard to update; the software can conflict with the OS, e.g., if a different version of a library is installed; dependencies are hard to manage; and builds generally happen in the same environment as deployment, which is troublesome if dependencies like internet access are not available.

In principle, users can self-compile arbitrary software. In practice, this is not feasible due to the level of effort.

4.3 Virtual machines

A *virtual machine* (VM) is a program that emulates a physical computer [39]. Thus, one can install an operating system and applications into this emulator, rather than on a physical computer. VMs can be implemented completely in software, at high cost in performance, or a kernel-level hypervisor can use CPU features to execute most instructions in hardware. A small subset, such as those that access devices, are trapped for software emulation.

This flexibility has attracted interest from the HPC community [12], as well as vendors such as Amazon who want to host HPC workloads in their virtualized data centers [1].

Applied to the UDSS problem, a user installs whatever guest OS they wish inside the virtual machines. Because VMs provide strong isolation, it is generally safe to let users have unlimited control of the guest, including root access.

This flexibility and isolation yields a strong advantage: the user can install any kernel, or even a different operating system, and adjust most OS and kernel settings. Also, because the VM itself runs as an normal, unprivileged user (ideally the invoking user), no root or trusted operations are needed.

On the other hand, virtualization brings disadvantages as well. First, VMs must be provisioned with a complete OS, just like a physical host; this duplicates host functionality such as the kernel and system daemons. Standard, reproducible provisioning tools do exist but are targeted to expert sysadmins and designed to be featureful, not easy to use.

Second, performance and complexity can be an issue. While compute-heavy code generally performs well due to hardware virtualization, many resources require indirection to be available to the guest. For example, a guest has its own Ethernet devices and IP addresses, so network infrastructure such as bridges and routing tables is needed. Filesystems can be made available with tools such as VirtFS [15], and SR-IOV lets PCIe devices split off non-configurable "virtual functions" that can be safely offered to guests [27]. In general, these indirections are mature and performant when used by industry, but HPC-specific functionality is less so.

Another option is *PCI passthrough*, where devices are removed from the host and given exclusively to the guest; this solves indirection problems but brings new ones, since the guest is no longer isolated. This trades performance against flexibility and security. For example, unsanitized GPU memory has led to information leakage [2], opportunities to load malicious firmware may exist, and allowing arbitrary user-specified packets on the network is unsafe.

In short, virtualization is a heavy-weight solution. If the UDSS is incompatible with the host kernel or its settings, then it should be

[13] `https://github.com/NERSC/chos`

Attribute	self-compile	virtual machine hypervisor	Shifter chroot	Singularity priv. ns.	Docker userns*	rkt userns*	NsJail userns	Charliecloud userns
Workflow (G1)								
User-defined kernel and settings	·	✓	·	·	·	·	·	·
Use package managers, e.g. `apt-get`, `yum`	·	✓	✓	✓	✓	✓	✓	✓
No conflicts with host software	·	✓	✓	✓	✓	✓	✓	✓
Industry-standard image build	·	·	✓	·	✓	✓	·	✓
Reproducible image build	·	·	✓	✓	✓	✓	·	✓
Resources (G2)								
No privileged or trusted daemons	✓	✓	·	✓	·	✓	✓	✓
No additional network infrastructure	✓	·	✓	✓	·	✓	✓	✓
Network filesystems see no UDSS metadata	·	✓	✓	✓	✓	✓	✓	✓
Direct device access	✓	·	✓	✓	✓	✓	✓	✓
Direct filesystem access	✓	·	✓	✓	✓	✓	✓	✓
Direct high-speed network access	✓	·	✓	✓	✓	✓	✓	✓
Simplicity (G3)								
Implementation language	n/a	varies	C, Python, C++, sh	C, sh, Python	Go	Go	C	C, sh
Lines of code	n/a	varies	19,000	11,000	133,000	52,000	4,000	800
No resource manager-specific code	✓	✓	·	✓	✓	✓	✓	✓
No communication framework-specific code	✓	✓	✓	·	✓	·	✓	✓
No root operations on center resources	✓	·	·	·	·	✓	✓	✓
No guest supervisor process	✓	·	·	·	·	✓	·	✓
No cache, configuration, or other state	✓	·	·	·	·	·	✓	✓

Table 1: Summary comparison of selected UDSS implementations on attributes driven by our design goals; the sense of boolean attributes is that a check mark reflects the more desirable value. An asterisk indicates that the implementation can use unprivileged namespaces, but this is not the standard mode. Charliecloud compares favorably on all attributes except the ability to select a UDSS-specific kernel and kernel settings, for which a virtual machine is needed. Information in this table was gathered from publications [14], documentation [5, 7, 25, 37], and source code (Shifter commit `9d296ce`, Singularity 2.2.1, Docker 17.03.1-ce, rkt 1.25.0, NsJail commit `730991b`, Charliecloud commit `9c03f44`). We counted lines with `wc`, briefly reviewing each implementation's code to count only feature modules, not support code such as tests or the build system.

considered; if not, then simpler solutions are more desirable. The latter is the space targeted by Charliecloud.

4.4 `chroot(2)`

`chroot(2)` is a traditional UNIX system call that changes a process' filesystem root [19]. It can be used for UDSS by installing a Linux distribution into a directory and then `chroot(2)`ing into that directory. If the host's kernel is compatible with the guest distribution, then this is a simple base upon which to implement an UDSS tool.

This approach has the advantage that host resources are straightforward to provide directly, because the guest is using the same kernel and, with appropriate bind-mounts, has access to the same devices and filesystems.

On the other hand, `chroot(2)` is a privileged system call, so escalation, privilege dropping [4], and the attack surface must be managed carefully. Also, `chroot(2)` has a history of ill-advised attempts to use it as a security boundary — a role it was never intended for, as by design it is trivial to escape a chroot [36] — which can make its politics troublesome.

Shifter[14] [14] is a `chroot(2)`-based effort from NERSC to make UDSS available to HPC users. While still new, Shifter has successfully run production codes [13].

Shifter uses Docker to offer a standard, repeatable image building workflow. Once an image is built, the user submits it to an unprivileged but trusted *image gateway*, which injects configuration and binaries, flattens it to an ext4 filesystem image, and copies this image to a parallel filesystem visible from compute nodes. This is a key innovation, as it insulates the network filesystem from image metadata traffic.

[14] `https://github.com/NERSC/shifter`

The UDSS image is mounted and activated by a setuid-root executable. In its recommended and most scalable configuration, Shifter uses resource manager plugins set up the environment and invoke the activation executable. The UDSS is then available transparently as soon as the user gains control of the job. This configuration also adds a setuid-root SSH daemon into the image.

To our knowledge, Shifter is the first HPC-targeted UDSS solution with a good workflow and direct resource access. However, it relies on trusted and privileged operations, its resource manager integration increases complexity, and it requires servers and daemons for the image gateway.

4.5 Linux containers

Containers are a kernel feature. However, user-space wrappers provide necessary convenience and usability. In this section, we outline the major existing implementations.

4.5.1 Basic tools. Simple command-line container tools that wrap the namespace system calls, `unshare(1)` and `nsenter(1)`, are included in the util-linux package,[15] which is installed by default on most Linux distributions. A similar wrapper is available in Google's NsJail,[16] which also includes network forwarding, `inetd`-style process spawning, and `seccomp` syscall filtering. Finally, `systemd` includes a namespaces wrapper `systemd-nspawn` [38].

In principle, one of these tools could provide the basis for Charliecloud. However, this would add dependencies and complexity for little gain, because it would require coordinating the system call dance via indirection through another tool rather than simply making the system calls directly.

4.5.2 Tools used in industry. Docker is perhaps the most well-known container implementation. It is an open source product[17] supported by Docker, Inc., with several advantages. Most importantly, the Dockerfile image specification language [6] makes it straightforward to build reproducible images, and the Docker Hub website enables sharing and composing images. Accordingly, this workflow is becoming an industry standard. Further, Docker can be configured for direct access to host resources via namespace selection and bind mounts. It supports user namespaces, but this is not the standard mode of operation [7].

Docker runs containers with runC,[18] which can also be used independently. The motivation for spinning off an additional open-source product is to create libraries and tools that support an open "universal container runtime" [10] to standardize high-quality container workflows and image formats. This has developed into a multi-vendor Open Container Initiative (OCI) [29].

The key disadvantages of Docker/runC are that it is large and complex, it depends on privileged daemons on both admin- and user-facing nodes (even with user namespaces), and it is written in Go, a language that HPC centers often lack expertise in. Critically, however, Docker's security culture in a poor fit for HPC; for example, access to any `docker` subcommand is equivalent to full root access by design [32].

A second project targeting OCI compliance is rkt,[19] an open-source Go project backed by CoreOS, Inc. [5]. rkt avoids the need for trusted daemons and optionally uses the user namespace, but it is still a large project with much functionality not needed for HPC. It can run Docker images and also provides a competing image specification language.

Finally, LXC is an older, though still active container project.[20] It is a library (`liblxc`) and tools (LXD) to manage containers and images. It fills a similar role but lacks a Dockerfile-like reproducible image building workflow.

We decided to use the Dockerfile/Docker Hub image workflow, for the reasons above, but create our own container runtime for Charliecloud. Existing runtimes do not meet our design goals: they are too complex, pose unacceptable security risks, and are targeted for web applications, not HPC.

4.5.3 HPC-focused tools. To our knowledge, only one other HPC-focused container implementation exists.

Singularity[21] is a Lawrence Berkeley National Laboratory project targeting single-file container images that can be run on any variety of Linux [25]. Images are specified with a custom Dockerfile-like language, using the host operating system to bootstrap an ext4 filesystem image[22] and then programs within the image to finish populating it. Direct host resources are available because Singularity uses only the mount and PID namespaces.

Because Singularity does not use the user namespace, it can run on older Linux distributions but requires privileged system calls. Also, Singularity contains code to integrate with OpenMPI 2.1. These differences highlight the dissimilar design goals of Singularity and Charliecloud.

5 IMPLICATIONS AND FUTURE WORK

This paper makes two key contributions. First, we propose clear user needs, risks, and design goals for UDSS in HPC. Second, to our knowledge, no HPC container solution that meets these goals has been previously proposed. We offer Charliecloud, an open-source implementation that does. Charliecloud provides a standard, reproducible, and unprivileged container workflow (G1), runs on existing HPC hardware and software (G2), and is extremely simple at 800 lines of code (G3). We argue that useful UDSS containers for HPC are easy, not hard, and do not require extensive infrastructure, lots of isolation, or a large software development effort.

We have recently deployed Charliecloud in production and look forward to reporting on its real-world use for real science. Additional future work includes exploring OCI compliance and benchmarking real application performance.

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. Department of Energy National Nuclear Security Administration under contract DE-AC52-06NA25396, via the LANL Institutional Computing Program and the

[15] `https://github.com/karelzak/util-linux`
[16] `https://github.com/google/nsjail`
[17] `https://github.com/docker/docker`
[18] `https://runc.io`
[19] `https://coreos.com/rkt`
[20] `https://linuxcontainers.org`
[21] `http://singularity.lbl.gov`
[22] Note that use of filesystem images expands the attack surface; for example, a recent kernel vulnerability involved crafted ext4 images [31].

Advanced Simulation and Computing Program. One of Doug Jacobsen's Shifter presentations [13] jump-started our reasoning about the motivations for UDSS. Jacobsen improved our understanding of Shifter by answering our questions, as did Gregory Kurtzer for Singularity; in both cases, any remaining errors are ours alone.

REFERENCES

[1] Amazon Web Services, Inc. 2015. *An introduction to high performance computing on AWS.* White paper. Amazon Web Services, Inc. https://d0.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf

[2] Evan Andersen. 2016. How Nvidia breaks Chrome incognito. (Jan. 2016). https://charliehorse55.wordpress.com/2016/01/09/how-nvidia-breaks-chrome-incognito/

[3] Diego Calleja. 2013. Linux 3.8. (April 2013). http://kernelnewbies.org/Linux_3.8

[4] Hao Chen, David Wagner, and Drew Dean. 2002. Setuid Demystified. In *USENIX Security Symposium*. http://crypto.stanford.edu/cs155/papers/setuid-usenix02.pdf

[5] CoreOS Inc. 2017. rkt 1.25.0 documentation. (2017). https://coreos.com/rkt/docs/1.25.0/

[6] Docker, Inc. 2016. *Dockerfile reference*. Documentation. Docker, Inc. https://docs.docker.com/engine/reference/builder/

[7] Docker Inc. 2017. *Docker Docs*. Documentation. Docker, Inc. https://docs.docker.com

[8] John L. Furlani and Peter W. Osel. 1996. Abstract yourself with modules. In *USENIX System Administration Conference*. http://modules.sourceforge.net/docs/absmod.pdf

[9] Tyler Hicks. 2017. CVE-2017-7184: kernel: Local privilege escalation in XFRM framework. (March 2017). http://seclists.org/oss-sec/2017/q1/689

[10] Solomon Hykes. 2015. Introducing runC: A lightweight universal container runtime. (June 2015). https://blog.docker.com/2015/06/runc

[11] Intel Corporation 2016. *Intel® MPI benchmarks: User guide and methodology description*. Documentation. Intel Corporation. https://software.intel.com/sites/default/files/managed/66/e8/IMB_Users_Guide.pdf

[12] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. 2010. Performance analysis of high performance computing applications on the Amazon Web Services cloud. In *IEEE CloudCom*. https://doi.org/10.1109/CloudCom.2010.69

[13] Doug Jacobsen and Shane Canon. 2015. Contain this, unleashing Docker for HPC. (May 2015). http://www.nersc.gov/assets/Uploads/nersc-brownbag-docker-jacobsen-canon.pdf

[14] Douglas M. Jacobsen and Richard Shane Canon. 2015. Contain this, unleashing Docker for HPC. In *Cray User Group*. http://www.nersc.gov/assets/Uploads/cug2015udi.pdf

[15] Venkateswararao Jujjuri, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty. 2010. VirtFS—a virtualization aware file system pass-through. In *Ottawa Linux Symposium (OLS)*. https://www.kernel.org/doc/ols/2010/ols2010-pages-109-120.pdf

[16] Michael Kerrisk. 2013. Namespaces in operation, part 1: Namespaces overview. *Linux Weekly News* (Jan. 2013). https://lwn.net/Articles/531114/

[17] Michael Kerrisk. 2013. Namespaces in operation, part 5: User namespaces. *Linux Weekly News* (Feb. 2013). https://lwn.net/Articles/532593/

[18] Michael Kerrisk et al. 2015. *pid_namespaces(7)*. Man page. http://man7.org/linux/man-pages/man7/pid_namespaces.7.html

[19] Michael Kerrisk et al. 2016. *chroot(2)*. Man page. http://man7.org/linux/man-pages/man2/chroot.2.html

[20] Michael Kerrisk et al. 2016. *clone(2)*. Man page. http://man7.org/linux/man-pages/man2/clone.2.html

[21] Michael Kerrisk et al. 2016. *namespaces(7)*. Man page. http://man7.org/linux/man-pages/man7/namespaces.7.html

[22] Michael Kerrisk et al. 2016. *setns(2)*. Man page. http://man7.org/linux/man-pages/man2/setns.2.html

[23] Michael Kerrisk et al. 2016. *unshare(2)*. Man page. http://man7.org/linux/man-pages/man2/unshare.2.html

[24] Michael Kerrisk et al. 2016. *user_namespaces(7)*. Man page. http://man7.org/linux/man-pages/man7/user_namespaces.7.html

[25] Gregory M. Kurtzer. 2016. Singularity. (July 2016). http://singularity.lbl.gov/

[26] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST)*. https://doi.org/10.1109/MSST.2012.6232369

[27] Scott Lowe. 2009. What is SR-IOV? (Dec. 2009). http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/

[28] Doug McIlroy, E. N. Pinson, and B. A. Tague. 1978. UNIX time-sharing system: Foreword. *Bell System Technical Journal* 67, 6 (1978).

[29] Open Container Initiative 2016. *About*. Mission statement. Open Container Initiative. https://www.opencontainers.org/about

[30] Larry Pezzaglia. 2012. CHOS in production. (April 2012). https://www.nersc.gov/assets/pubs_presos/chos.pdf

[31] Red Hat Inc. 2016. CVE-2016-10208. (Nov. 2016). https://access.redhat.com/security/cve/cve-2016-10208

[32] Reventlov. 2015. Using the docker command to root the host (totally not a security issue). (April 2015). http://reventlov.com/advisories/using-the-docker-command-to-root-the-host

[33] Rami Rosen. 2016. Namespaces and cgroups, the basis of Linux containers. (Feb. 2016). http://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf

[34] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. 2015. Performance evaluation of containers for HPC. In *Euro-Par 2015: Parallel Processing Workshops*. https://doi.org/10.1007/978-3-319-27308-2_65

[35] Jerome H. Saltzer. 1974. Protection and the control of information sharing in Multics. *CACM* 17, 7 (July 1974). https://doi.org/10.1145/361011.361067

[36] Simes. 2002. How to break out of a chroot() jail. (May 2002). https://web.archive.org/web/20160209154009/http://www.bpfh.net/simes/computing/chroot-break.html

[37] Robert Swiecki. 2016. NsJail. (Dec. 2016). https://google.github.io/nsjail/

[38] systemd contributors. 2017. *systemd-nspawn*. Man page. https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html

[39] Wikipedia editors. 2016. Virtualization. (Feb. 2016). https://en.wikipedia.org/w/index.php?title=Virtualization&oldid=704408822

[40] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. 2013. Performance evaluation of container-based virtualization for high performance computing environments. In *Euromicro Parallel, Distributed, and Network-Based Processing*. https://doi.org/10.1109/PDP.2013.41