Linux Containers for Fun and Profit in HPC

REID PRIEDHORSKY AND TIM RANDLES

Reid Priedhorsky is a Staff Scientist at Los Alamos National Laboratory. Prior to Los Alamos, he was a Research Staff member at IBM Research. He holds a PhD in computer science from the University of Minnesota and a BA, also in computer science, from Macalester College. His work focuses on large-scale data analysis from both systems and applications perspectives. Recent lines of research include using social media and Web traffic to monitor and forecast the spread of disease as well as developing technology to bring data-intensive computing and user-defined software stacks to existing high-performance computing systems. In his spare time, he enjoys reading, bicycling, hiking (especially in the mountains and deserts of the American West), tinkering with things, photography, and hanging out with his wife and son. reidpr@lanl.gov

Tim Randles has been working in scientific, research, and high-performance computing for many years, first in the Department of Physics at the Ohio State University, then at the Maui High Performance Computing Center, and most recently as a member of the HPC Division at Los Alamos National Laboratory. His current work is focused on the convergence of the high performance and cloud computing worlds. When Tim isn't working, he enjoys brewing beer, cheesemaking, taking hikes, and working on computer games. He lives in Santa Fe with his wife and three cats. In an ideal world they'd also have a few goats and some chickens. trandles@lanl.gov

This article outlines options for user-defined software stacks from an HPC perspective. We argue that a lightweight approach based on Linux containers is most suitable for HPC centers because it provides the best balance between maximizing service of user needs and minimizing risks. We discuss how containers work and several implementations, including Charliecloud, our own open-source solution developed at Los Alamos.

Innovating Faster in HPC

Users of high performance computing resources have always been asking for more, better, and different software environments to support their scientific codes. We've identified four reasons why:

- **Software dependencies** not provided by the center. Examples include libraries that are numerous, unusual, or simply newer or older; configuration incompatibilities; and build-time resources such as Internet access.
- **Portability** of environments between resources. For example, it is helpful to have the same environment across development and testing workstations, local compute servers for small production runs, and HPC resources for large runs.
- **Consistency** of environments to promote reproducibility. Examples include validated software stacks standardized by a field of inquiry and archival environments that remain consistent into the future.
- **Usability** and comprehensibility for meeting the above.

These needs for flexibility have been traditionally addressed by sysadmins installing various software upon user request; users can then choose what they want with commands such as `module load`. However, only software with high demand justifies the sysadmin effort for installation and maintenance. Thus, more unusual needs go unmet, whether innovative or crackpot—and it's hard to tell which is which beforehand. This can create a chicken-and-egg problem: a package has low demand because it's unavailable, and it's unavailable because it has low demand.

This motivates empowerment of users with "bring your own software stack" functionality, which we call *user-defined software stacks* (UDSS). The basic notion is to let users install software of their choice, up to and including a complete Linux distribution, and run it on HPC resources.

Of course, this approach has drawbacks as well. We've identified three potential pitfalls:

- **Security:** By introducing very flexible new features, UDSS can expand a center's attack surface, especially if they depend on privileged or trusted functionality.
- **Missing functionality:** Separation from the native software stack can interfere with features such as file systems, accelerator hardware, and high-speed interconnects that make HPC centers interesting and special.
- **Performance:** Implementations must take care to avoid introducing overhead that meaningfully impacts performance.

Options for User-Defined Software Stacks

We believe the needs and pitfalls above lead to three design goals for an HPC-focused UDSS implementation.

First, it should provide a standard and reproducible workflow. A standard workflow reduces training and development costs while enhancing the portability of staff skill sets; a reproducible workflow, in contrast with a "tinker 'til it's ready, then freeze," makes the creation of UDSS images simpler and more robust.

Second, it should run on existing, minimally modified HPC hardware and software resources. This is for two reasons. First, the pitfalls above are already well-controlled in HPC centers; smaller modifications add fewer risks than larger ones. Second, the challenges of orchestrating large parallel applications are well-addressed by HPC centers. We have good resource managers (Slurm, Moab, Torque, PBS, etc.), good high-performance parallel file systems (Lustre, Panasas), good high-speed networks (InfiniBand, OPA), and more. These solutions need not be reimplemented and reoptimized using novel technology.

Finally, it should be as simple as is practical while still delivering the necessary features. This is in keeping with the UNIX philosophy to "make each program do one thing well" [2].

We see three basic options for implementing UDSS: self-compile, virtual machines, and Linux containers.

Compile It Yourself

The traditional method for users to take care of themselves is to simply compile what they need in a home directory or other user area. This is available almost everywhere already, employs only unprivileged functionality, and yields direct access to all center resources. However, it is also tedious and error-prone, hard to update, and does not provide portability or consistency of environments. In principle, users can self-compile arbitrary software; in practice, its difficulty is very limiting.

Virtual Machines and Public/Private Cloud

A *virtual machine* (VM) is a program that emulates a physical computer. One then installs an operating system and applications into this emulator. This is appealing because it gives users ultimate flexibility and strong isolation; it is reasonable to let them install even non-UNIX operating systems and have full administrative privileges. Modern virtual machines perform excellently for things needed by industry, such as CPU-bound tasks and Ethernet networking.

However, the approach has challenges. Performance is often an issue for things uncommon in industry, such as HPC high-speed networks; this can sometimes be mitigated by compromising on isolation. Virtual machines must be provisioned with a complete OS, including kernel and system daemons, and the support infrastructure such as virtual networking is complex.

There is a view that HPC should become more like cloud computing, which offers on-demand, loosely coupled virtual machines. However, this approach requires that either users or sysadmins reimplement and reoptimize much of the functionality that HPC centers already offer.

Our belief is that HPC centers should offer virtual machines only if credible UDSS require not only a custom user space but a custom kernel as well. Otherwise, its disadvantages dominate.

Linux Containers

A middle approach is containers, which share "the only" kernel with the native software stack, accomplishing isolation with Linux *namespaces* and related features. (For further reading, we recommend Michael Kerrisk's series in *Linux Weekly News* [1]as well as `namespaces(7)` and related man pages.)

Note that *container* is a widely used term with varying definitions. The view outlined here is the one we find most sensible.

Privileged Linux Namespaces

Linux has six namespaces that isolate different classes of kernel resources; processes in one namespace see a different view of system state than processes in another. Five namespaces are what we call *privileged*, needing root to create; the sixth, unprivileged one, is covered in the next section. The privileged namespaces are:

1. **Mount:** File-system tree and mounts
2. **PID:** Process IDs—a process in a PID namespace has a different PID inside and outside the namespace
3. **UTS:** Host name and domain name (the name deriving 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, both System V and POSIX

The six namespaces can be mixed and matched, but there are quirks. 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 (try `ls -l /proc/self/ns`). Namespaces form a tree, with parent/child relationships, and everything is owned by a namespace. For example, though it cannot create its own, a mount namespace can bind-mount its parent's, to which the parent namespace controls access.

Namespaces are manipulated by three system calls: `unshare(2)` puts an existing process into new namespaces, `clone(2)` can put a new child process into new namespaces, and `setns(2)` joins an existing namespace.

These features are useful for UDSS because they allow any directory to become the file-system root of a mount-namespaced process, and the other namespaces can be added for additional isolation as needed.

The Unprivileged User Namespace

The sixth namespace, *user*, was added starting in Linux 3.8. Its goal is to give unprivileged processes access to traditionally privileged functionality in specific contexts when doing so is safe. This is accomplished with namespace-specific capabilities and user/group IDs.

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.

The relationship between child and parent namespace UIDs is controlled by a one-to-one mapping defined during namespace setup. The situation with GIDs is analogous. A common use is to map one's normal, unprivileged UID to 0 inside the namespace, thus appearing to be root inside the namespace.

If the namespace is created by an unprivileged user, the parent side of this map may only be the existing EUID. This limits access to things already accessible, because while any UID can be selected in the child namespace, it must map to the user's existing, real 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 parent side of the map.

This one-to-one mapping is used to translate UIDs in both directions. When a UID-based access decision is initiated inside the namespace, the map translates the in-container UID up through the namespace tree to its corresponding base 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.

Thus, processes and kernel resources inside the user namespace can be manipulated arbitrarily, but only in ways that do not affect the parent namespace—privilege is an illusion.

```
#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);
}
```

Listing 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.

Listing 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 implementations 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 file-system tree and safely access host resources, even if he or she holds "privileges" inside the container, without depending on the container implementation for security.

Additional Components

Other Linux features commonly used in container implementations include:

- `cgroups(7)`, which track and limit resource consumption of processes. This can be useful in multi-tenant settings to keep users from stomping on each other.
- `prctl(2)` with `PR_SET_NO_NEW_PRIVS`, which prevents `execve(2)` from increasing privileges. This can protect against some privilege escalation bugs, e.g., in setuid binaries.
- `seccomp(2)` filters system calls, thus mitigating security issues in the excluded calls.
- SELinux and AppArmor have various features that can change what the processes may do.

These features can be applied to processes in general, not just containers. For example, if a `seccomp(2)` filter increases the security of container jobs, why not apply it to all jobs? That said, it may be reasonable for container implementations to use these tools under a "belt and suspenders" philosophy, if the benefit outweighs the complexity gain.

Container Implementations

There are many container implementations. We divide them generally into two categories, full-featured and lightweight, which serve different use cases.

Full-Featured

Full-featured container implementations have (shockingly!) lots of features, for example some subset of:

- Image building
- Image management (e.g., storage, caching, tagging, signing)
- Images stored in custom formats
- Image sharing (repository/registry, search, Web site)
- Orchestration
- Storage management (overlay management, back-end drivers)
- Runtime setup (default command, start-up script, `inetd`-type functionality)
- Stateful containers that can be started and stopped
- Supervisor daemons, e.g., to proxy signals as required by PID namespace

Typically, these implementations comprise a security boundary.

Examples from industry include Docker/runC, rkt, and LXC, along with perhaps `systemd-nspawn(1)` and NsJail; examples from HPC include NERSC's Shifter and LBNL's Singularity.

These many features are implemented because they are useful, but there are drawbacks. For example, access to the `docker` command is equivalent to root by design [4]. One could write a wrapper, but input sanitization is a perilously difficult problem.

All these features must be supported for configuration, security, and user support. For example, Docker comprises 133,000 lines of code, some of which are privileged, and Docker is written in Go, a language HPC centers tend to lack expertise in.

It can be done, of course, but it's a major step for an HPC center and must be done with great care. We believe that deploying a lightweight solution is an easier path.

Lightweight

In contrast, lightweight implementations have few features. Most basically, given an image, they run a containerized process within that image. Typically, image building is delegated to other tools, whether designed for containers or not (e.g., `debootstrap(8)`).

Lightweight implementations minimize security responsibility, and they have fewer lines of code to evaluate, support, and secure. This makes deployment lower cost and easier for HPC centers to justify.

Examples from industry include `unshare(1)` from `util-linux`, along with perhaps `systemd-nspawn(1)` and NsJail. In HPC, we are aware of only our own Charliecloud, discussed below.

We believe that lightweight implementations are best for HPC centers. They bring the most important dimensions of cloud-like flexibility without compromising the existing tools and strengths of HPC centers or demanding their reimplementation and reoptimization.

Charliecloud

Our basic design is motivated by two observations. First, full-featured implementations are not a good fit for HPC centers. However, some of their features are really important: most importantly, image building and image sharing.

```
$ cd charliecloud/examples/hello
$ ch-build -t hello ../..
Sending build context to Docker daemon 12.24 MB
[...]
Successfully built 2972e7281f75
$ ch-docker2tar hello /var/tmp
57M /var/tmp/hello.tar.gz
$ ch-tar2dir /var/tmp/hello.tar.gz /var/tmp/hello
/var/tmp/hello unpacked ok
$ ch-run /var/tmp/hello -- echo "I'm in a container"
I'm in a container
```

Listing 2: Building and running "hello world" in Charliecloud requires only a few simple commands. The tarball image created in Step 3 can be run on any host where the Charliecloud runtime is installed; Docker is no longer needed once the image is built.

Thus, our open-source, lightweight container implementation takes a dual approach. We put building and sharing in a sandbox that is separate from HPC center resources. This could be a user workstation or a virtual machine: somewhere safe to give the user root. In this sandbox, Charliecloud wraps Docker for image building, and the other Docker tools are also available, including sharing via pull/push to any Docker Hub repository.

Running images uses our own runtime that is unprivileged and independent of Docker. This can be on center resources or anywhere else with the Charliecloud runtime installed, such as the same sandbox for development and testing. Listing 2 is an example of this workflow.

This brings us back to our three design goals:

1. A standard, reproducible workflow is accomplished by using Docker for image building. This enables use of Dockerfiles, an industry standard for reproducible builds. Working atop Docker for image management also integrates our solution into the robust Docker image ecosystem.
2. Running on existing HPC resources is accomplished with our `ch-run` runtime, which provides just enough isolation using the mount and user namespaces to run a container image. Similarly to `time(1)`, which provides an environment that records resource usage, `ch-run` provides a container environment.

   `ch-run` requires no privilege and depends on the Linux kernel for security, just like any other user process. Performance is the same as native in our tests, modulo noise, because minimal isolation yields direct access to all resources: compute, network, file systems, accelerators, and the rest. `ch-run` scales using standard HPC tools. For example, a large application can be started simply with `mpirun -np $BIGNUM ch-run bigprog`.
3. Simplicity: Charliecloud is a collection of five shell scripts and two C programs totaling roughly 900 lines of code. For comparison, NsJail is 4,000 lines, Singularity 11,000, Shifter 19,000, and Docker 133,000.

We have recently deployed Charliecloud in production and are working with Los Alamos scientists on its use and performance for real-world science code. We look forward to sharing these results.

If you'd like to learn more, Charliecloud's source code is available from GitHub (https://github.com/hpc/charliecloud), and its documentation is on the Web (https://hpc.github.io/charliecloud). Further technical detail is available in our forthcoming Supercomputing paper [3].

References

[1] Michael Kerrisk, "Namespaces in Operation, Part 1: Namespaces Overview," *Linux Weekly News*, January 4, 2013: https://lwn.net/Articles/531114/.

[2] Doug McIlroy, E. N. Pinson, and B. A. Tague, "UNIX Time-Sharing System," Foreword, *Bell System Technical Journal*, vol. 67, no. 6, 1978.

[3] Reid Priedhorsky and Tim Randles, "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC," in *Supercomputing*, 2017 (forthcoming).

[4] Reventlov's Silly Hacks, "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.