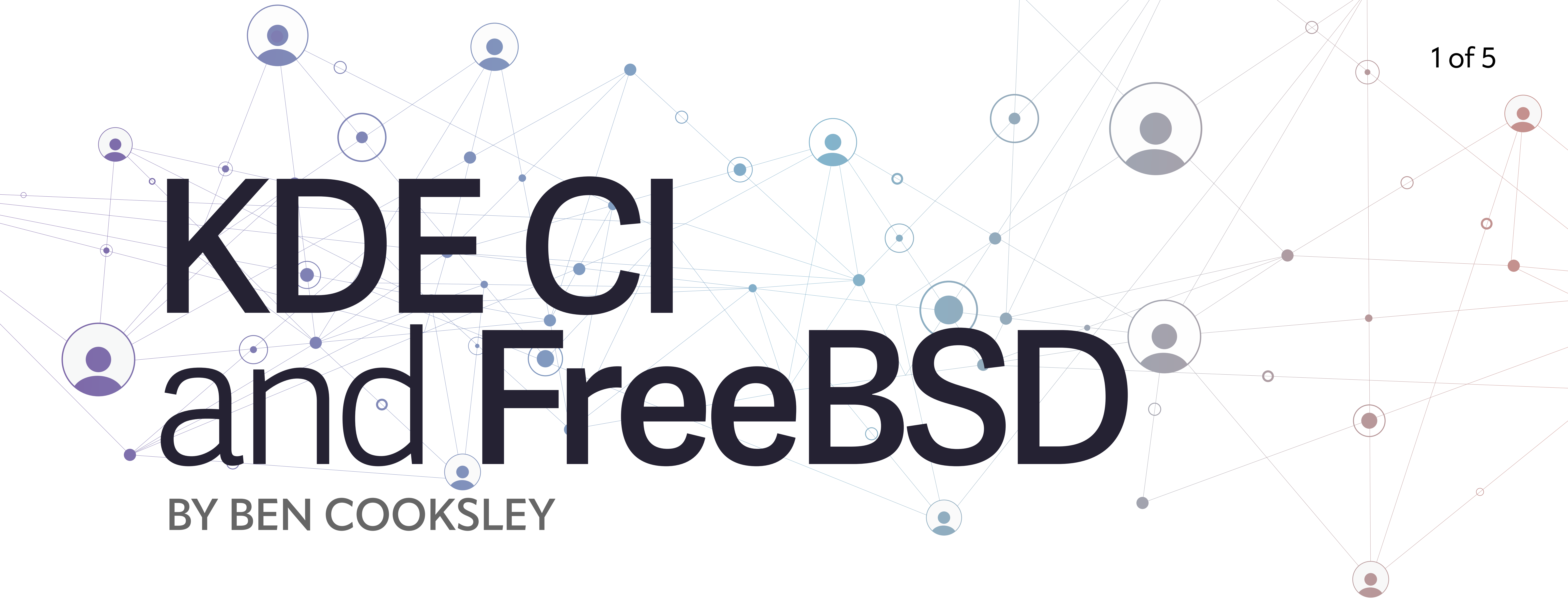

KDE CI and FreeBSD

BY BEN COOKSLEY

Continuous Integration (CI) is something KDE has worked on improving now for some years, with the first implementation of CI for KDE software starting back in August 2011. Since then, the system has evolved substantially, picking up support not only for multiple versions of Qt (the toolkit used to write most KDE software) but also support for multiple platforms.

Running all these builds reliably and consistently, across the multiple operating systems involved, is something that is only possible thanks to containers, which are increasingly ubiquitous across all platforms. To understand the challenges containers solve, as well as other challenges in building scalable CI systems though, we must go back to the beginning of KDE CI.

When the system first started life, it was a relatively simple Jenkins setup—with builds performed on the same server that hosted Jenkins. This made life quite simple, however it also had limitations. As demand for builds increased with more projects onboarding to the system, it soon became clear that more machines would be needed.

This presented a bit of a conundrum though, as KDE software tends to require other KDE libraries in order to be built, and not just any version either—usually the most recent version. This meant that it wouldn't just be a case of increasing the number of builders, we would also need to ensure that the latest version of dependencies were still available.

The system has evolved substantially, picking up support not only for multiple versions of Qt but also support for multiple platforms.

Due to the amount of time needed to build the full chain of dependencies for our applications, the concept of just building everything each time was quickly ruled out—meaning it would be necessary to share the binaries resulting from those builds. After a quick review of our options, rsync was quickly selected as our preferred choice, and once again all was well.

Enter FreeBSD

By 2017, the system encountered it's next set of growing pains, as it became desirable to start adding support for new platforms, which is where FreeBSD enters the picture for the first time.

This initial implementation of FreeBSD support within our CI system was relatively simple, and made use of virtual machines running on our Linux CI workers. These machines were individually set up with help from the KDE on FreeBSD team, and much like our Linux builds at the time, included everything needed to build KDE software.

This approach however did have its downsides. While we did ensure that all of the builders made use of the same custom FreeBSD repository that had all the necessary dependencies in it to build our software, each of the machines was still built individually. This made scaling the system non-trivial, as any changes had to be applied to each builder one by one.

It did succeed, however, in ensuring that KDE software could be reliably built on FreeBSD, and ensured dependencies were packaged in advance of KDE software starting to make use of them—improving the experience for the KDE on FreeBSD team substantially.

At the same time as we added support for FreeBSD, we also adopted something that was still fairly new at the time for our Linux builds—Docker. For the first time, we were able to produce a single master setup that could be distributed across all our builders, making it easy to roll out changes across the CI system without having to manually apply them to each machine. The golden age of container-based builds had begun its arrival. The only downside was that it was Linux only, so the question remained of how to replicate this on other platforms.

We also adopted something that was still fairly new at the time for our Linux builds—Dockers.

Before we could tackle that though, all of this growth in build capability had resulted in some new, and slightly unexpected problems starting to show up. From time to time builds would randomly fail, with logs indicating that files were missing or symlinks broken. Later checks would show that the files were there, and subsequent runs completed successfully. The problem? Atomicity.

Up to now, we only had a small handful of build nodes, which had certain limitations on their performance. The new setup however had much more capable hardware, and as such was completing builds faster—meaning it was increasingly likely that rsync would be mid-upload when another build went to download build artifacts. This is why we were seeing missing files and broken symlinks in some builds as the unfortunate build happened to start at the same time one of its dependencies happened to be syncing build results.

Thankfully, the answer once again was fairly simple—moving to using tarballs of build artifacts. This let us publish the full set of files from a build in one fluid atomic operation, and in addition to a change of protocol over to SFTP (to accommodate platforms without rsync) meant that the CI system was once again operating smoothly—with quite a bit more resource power, and supporting quite a few more platforms.

The issue of having to maintain machines individually by hand, however, did not go away. The migration to Gitlab and Gitlab CI made this issue more apparent than ever, as build nodes began to run out of disk space due to accumulated code checkouts and other build artifacts would quickly fill disk space. We also battled problems with leftover processes from tests chewing up CPU time (sometimes entire cores even)—all things that did not exist at all on our Docker based Linux builds.

Many options and solutions were discussed on this, including improvements to Gitlab Runner and how it handles the "shell" executor, cronjobs to perform cleanup of build artifacts and code checkouts, as well as building something on top of FreeBSD Jails. None of these, however, would replicate the same experience we had on Linux with Docker.

Finding Podman

So it was one morning while looking at FreeBSD containerization options that we stumbled across Podman and its companion ocijail support. This promised us all the things we were used to enjoying with our Docker-based Linux setups, but on FreeBSD.

Significantly, it would mean that the issues we had been experiencing with stray processes and leftover build artifacts that we had to clean up manually would be solved. And it would also let us make use of a standard Open Container Initiative registry (such as Gitlab's built-in Container Registry) to distribute images of FreeBSD to all our builders—solving our issue of having to maintain the machines individually.

Getting a working image built would be our first challenge. For Linux systems, Docker and Podman are quite well established with detailed documentation on the available base images and what those base images contain. With the appropriate FreeBSD base image found though, we thought it would be a simple case of adding our normal FreeBSD package repository and installing everything we would normally need.

Getting a working image built would be our first challenge.

We would soon find our first bump on the road, as, unexpectedly, CMake indicated on our first build run in a container that it was unable to find a compiler. We thought this quite odd, as usually FreeBSD systems ship with a compiler installed. After a bit of digging, we found the first major difference between FreeBSD containers and a normal FreeBSD system—namely, that they are significantly stripped down, and therefore don't include a compiler.

After a couple of iterations, we ended up adding in a compiler and C library development headers—which allowed our very first piece of KDE software to be built in a FreeBSD container. Thinking we now had everything sorted, we pressed ahead—only for subsequent bits of KDE software to fail as additional development packages were needed. Many iterations later (including installing a bunch more development and non-development FreeBSD-* packages) we were finally presented with a completed build for a number of key KDE packages.

With this sorted, attention now turned to what Gitlab Runner calls a "helper image", which is something it uses to perform Git operations and upload artifacts from builds to Gitlab itself among other things. While we could have made use of FreeBSD's support to run Linux binaries, that would have been an imperfect solution. So we naturally set out to build this natively for FreeBSD as well. After replicating what had been done by Gitlab themselves to build the images, but in FreeBSD, we soon had what we thought would be the final piece ready.

It was now that the fun part of the adventure began, and a deep dive into the inner workings of Gitlab Runner and Podman began. The first hurdle thrown was moments after we connected Gitlab Runner to Podman (using it's Docker compatibility option), when our first build was met with the message of "unsupported os type: freebsd".

A quick search of the Gitlab Runner codebase revealed that for Docker it checked the operating system of the remote Docker (or in our case: Podman) host. A quick patch and rebuild of Gitlab Runner later, and we had a very similar, but not quite identical error: "unsupported OSType: freebsd". More patching to Gitlab Runner followed only for a third much more ominous error to be returned, especially given Gitlab Runner is written in Go:

```
ERROR: Job failed (system failure): prepare environment:
Error response from daemon: runtime error: invalid memory address or nil pointer
dereference(docker.go:624:0s.
Check https://docs.gitlab.com/runner/shells/index.html#shell-profile-loading for more
information.
```

With this it became apparent that much more work would be required to get this working, however, given the promise of what containerized builds could deliver, we persisted and started looking into where this was failing. After some research through the Gitlab Runner codebase, we found code that didn't seem to be doing anything too special:

```
inspect, err := e.client.ContainerInspect(e.Context, resp.ID)
```

And so began many hours of debugging, searching for why this one line of code failed on FreeBSD yet worked absolutely fine on Linux (regardless of whether it was Podman or Docker). Eventually, however, we stumbled on the cause: the Podman daemon itself was falling over and abandoning the request. With this information in hand, the issue was soon easily reproduced by trying to run "podman inspect" against a running container, resulting in the expected crash we wanted to see. Success!

It became apparent that much more work would be required to get this working.

Having searched through Gitlab Runner, focus now turned to Podman itself. Before long the cause had been narrowed down to code that was specifically called for "inspect" operations, and soon after that a specific line was identified that tried to interact with Linux specific constructs no matter the platform. Yet, another patch later, we had a "podman inspect" that did not crash, and then shortly after that, our first FreeBSD build starting successfully.

Running Builds on FreeBSD

That first build may have failed (due to known issues with Git and the way Gitlab Runner interacts with containers that run as builds as users other than root), but the important thing was we had running builds on FreeBSD.

At this point, you may have thought we were home free and could begin to roll out FreeBSD based containerized builds to all KDE projects. Final testing, however, revealed one final issue: that network speeds in our FreeBSD containers appeared to be quite a bit slower than we had expected, being significantly slower than what the FreeBSD hosts were capable of.

Thankfully, this was not a new issue, was something that others had run into before, and was something we had anticipated we would encounter. This particular issue has been well written up by Tara Stella in the past, in their experiences diving into the world of Podman and FreeBSD containers, and is caused by Large Receive Offload or LRO. One quick configuration change later and we had the performance we expected—and were finally ready to go live.

Today, KDE runs FreeBSD CI builds using Podman and ocijail-based containers exclusively, with 5 FreeBSD host systems handling the build requests. These builds are performed using two different CI images—one for each of the two supported versions of Qt (being Qt 5 and Qt 6) ensuring that KDE software can be cleanly built from scratch and, optionally, has fully passing unit test results.

Since migrating over from FreeBSD dedicated virtual machines to FreeBSD containerized builds, we have gone from receiving complaints from developers due to broken builders and having to undertake maintenance on our builders several times a week (and sometimes even daily), to receiving no complaints in several weeks and only needing to undertake periodic maintenance.

The patches that we wrote (just a couple of lines for both Podman and Gitlab Runner) have been successfully upstreamed and should now be available for all to use and enjoy in building out their own CI setups.

The benefits of switching to containers—especially for Continuous Integration systems – cannot be understated and are something any team that maintains a system should consider investigating, as the returns are well worth the initial cost of migration.

BEN COOKSLEY is an accountant and also a computer scientist known for his contributions to the KDE community, particularly in system administration and infrastructure. His interest in sysadmin work stems from a curiosity about how systems operate and integrate.

Write For Us!

Contact Jim Maurer with your article ideas.

(**maurer.jim@gmail.com**)