ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-M04BC005

APPLES TO ORANGES: A COMPARISON OF EPICS BUILD AND DEPLOYMENT SYSTEMS

S. Rose*, D. Araujo†, A. Lindh Olsson‡, L. Magalhães§
European Spallation Source ERIC, Lund, Sweden

Abstract

ESS currently uses two different systems for managing the build and deployment of EPICS modules. Both of these use modules that are packaged and prepared to be dynamically loaded into soft IOCs, based on the `require` module developed at PSI. The difference is the deployment: For the accelerator, we use a custom python script to define and build an EPICS environment which is then distributed on a global NFS share to the production and lab networks. For the neutron instrumentation side, in contrast, we use Conda to build individual EPICS environments, where the individual packages are stored on a shared artifactory server.

In this paper we will provide an overview of some of the challenges, contrasts, and lessons learned from these two different but related approaches to EPICS module deployment.

E3

History

The ESS EPICS [1] Environment (e3) [2] is based on the approach developed at PSI which uses the standard EPICS base executable `softIocPVA` to run IOCs. Instead of statically or dynamically linking support modules and compiling them into a custom binary executable, we instead configure and dynamically load the provided shared libraries. This is all done using the module `require` [3] which acts as both a parallel build system for EPICS modules, as well as an EPICS module in its own right that dynamically loads other EPICS modules.

e3 was developed initially by Jeong Han Lee, who introduced the notion of a "wrapper". After he left, the development and maintenance of e3 was taken over by a small team (the e3 team). This team initially consisted of representatives from all of the main groups that comprise ICS (Integrated Control Systems): Software, Hardware and Integration, and Infrastructure. While the team has evolved over time, it maintains its interdisciplinary nature and tight connections with all of its stakeholders.

Structure

One challenge when with working with community EPICS code is that one will often need to provide site-specific customisation to each support module; this presents, for example, a challenge when needing to update a module: how does one keep local changes in sync? How does one track that which comes from the community, and that which is specific to your site?

e3 resolves this by using a "wrapper"[1]: a separate repository which contains a reference to the community code together with any site-specific modifications that are necessary. This can include patches, site-specific database or template files, as well as any other site-specific build or run-time configuration. It also includes metadata such as a description of the dependencies required to build and load the module into an IOC; see Fig. 1.

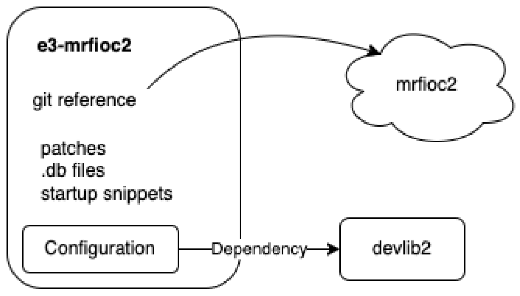

Figure 1: Example e3 wrapper

This approach allows for development in line with community collaboration and release best-practices: we can update community releases if critical errors are found (such as a recent `caPutLog` memory leak), as well as develop and test patches before submitting them to the community for review.

In order to do this, e3 has opted to use a fork of PSI's `require` module. In contrast to traditional EPICS IOCs, this module provides dynamic run-time loading and registering of support modules. It requires additional build configuration, but handles some rudimentary run-time dependency resolution.

One of our main divergences from the PSI `require` module is our use of wrappers. However, there are other significant changes such as our use of "virtual environments", as well as a standard versioning schema which distinguishes site-specific changes from community releases. This is necessary to avoid (among other things) so-called 'dependency hell' (see Fig. 2).

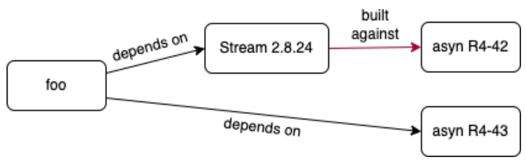

Figure 2: Dependency hell

This is handled by appending a *revision number* to the version. Two builds that use the same source version but dif-

* simon.rose@ess.eu
† douglas.araujo@ess.eu
‡ Anders.LindhOlsson@ess.eu
§ lucas.magalhaes@ess.eu

[1] Note that in Conda this is called a 'recipe'; however, that name distinction is not particularly relevant.

fer either in dependency or wrapper configuration will have different revision numbers. This allows us, for example, to handle the scenario in Fig. 2 by re-building Streamdevice, but built against a different version of asyn (see Fig. 3).

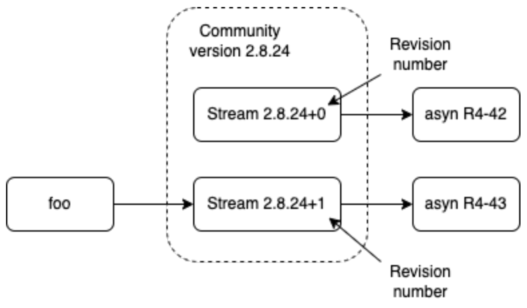

Figure 3: Using revision numbers to fix dependency issues

We should further note that there are several ways of handling this type of issue; Conda itself uses both a build number and a hash of the dependencies to differentiate between builds that use the same source.

Conda and ESS

B. Bertrand and A. Harrisson presented [4] an alternative approach to managing a shared EPICS environment which uses the open-source package manager Conda [5] to build and manage the pool of support modules as well as the individual IOCs. Instead of using a shared NFS server, Conda uses an ESS-hosted artifactory server; the individual IOCs reside in their own distinct Conda "environments", which contain the necessary dependencies in order to run.

A proof-of-concept for using this approach at scale was designed and presented. However, some potential issues that might affect the then upcoming commisioning stages were identified. As a result, the e3 team switched to developing a more standard NFS-based approach for EPICS distribution. In the meantime, due to a different commisioning schedule for the Neutron instrumentation team, we were able to continue developing the conda-based solution for their use.

CURRENT APPROACH

NFS e3

In order to satisfy our requirements, we further developed the build backend (require and its custom build rules), as well as developed a completely new python-based build frontend (e3-build) which acts as a build and environment manager. The input to this build manager is a *specification*, as shown in Fig. 4, and is broadly a list of tagged commits to the wrappers. e3-build then takes these commits, reads in the dependency information specified therein, determines an appropriate and consistent build order, and then builds any necessary modules that are missing from the current environment.

ESS then uses GitLab-CI [6] to automatically build and deploy new modules, (see Fig. 5). The workflow is as follows:

1. Module request comes in from integrator

```
config:
  base: 7.0.7-NA/7.0.7-37d472c
  require: 7.0.7-5.0.0/5.0.0-6a40805
metadata:
  type: specification
  version: 1
modules:
  adandor:
    versions:
    - 7.0.7-5.0.0/2.8.0-6f3a1f4+1
    - 7.0.7-5.0.0/2.8.0-6f3a1f4+2
  adcore:
    versions:
    - 7.0.7-5.0.0/3.12.1+2-50b90f0
  adsupport:
    versions:
    - 7.0.7-5.0.0/1.10.0+2-205ab18
  asyn:
    versions:
    - 7.0.7-5.0.0/4.43.0+3-871c171
  autosave:
    versions:
    - 7.0.7-5.0.0/5.10.2+2-45dc1de
  busy:
    versions:
    - 7.0.7-5.0.0/1.7.3+2-1ea0f0a
```

Figure 4: Example specification file

2. e3 team ensures that the module is ready and appropriate for release
3. Automatically build and run any available tests
4. If it passes, tag the wrapper and add it to the specification
5. Automatically run the package manager to build and deploy to the test environment
6. If that succeeds, deploy to production environment

An important part of the above is that the e3 team acts as gatekeepers for ESS' EPICS environment. While many modules can be released with little oversight (such as modules developed by integrators to manage site-specific equipment), for certain modules (in particular, asyn—see Fig. 6) we are much more careful about release schedules in order to avoid long-term maintenance issues. In general, any module that is a dependency of many other modules is something that we take much more care about its release schedule into our environment.

Conda e3

During this time, we continued using the existing Conda proof-of-concept for managing the control system for the Neutron Instrumentation team. This allowed us to focus on some of the necessary automation workflows and tools that had not been ready when the proof-of-concept was originally presented.

For example, in the original design a build of a support module would trigger a rebuild of every single downstream dependency. While this allows for a quick and simplified workflow for module developers, it encourages and abets the exact combinatorial explosion of dependencies that we had

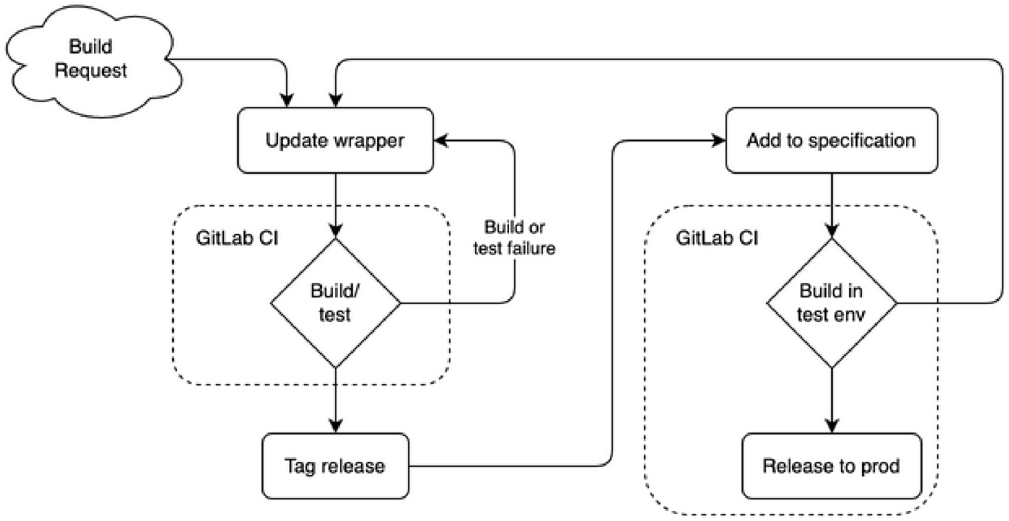

Figure 5: Module deployment workflow

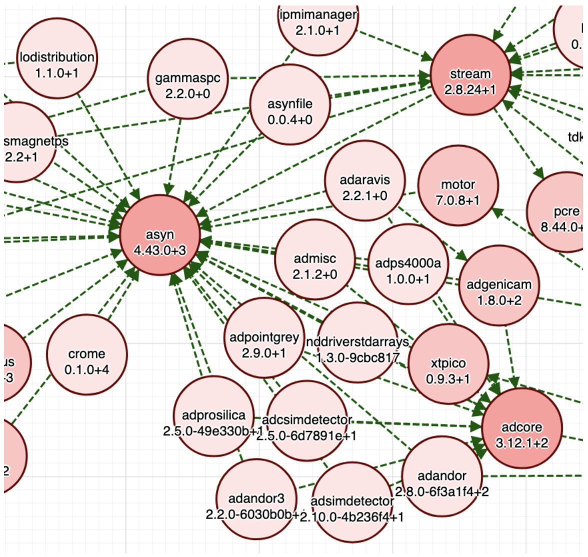

Figure 6: Module dependency graph

been trying to avoid.

A lot of work has been put into improving procedures and automation tools to allow for automation where appropriate, but also for control at gatekeeping by a small dedicated team, exactly as the original requirements stated.

Finally, the use of Conda as an EPICS build system over the past several years has revealed that it is both capable and mature enough to handle the commissioning needs of our Neutron Instrumentation network. Our next aspiration is to re-eveluate it as an EPICS build and deployment system over the entirety of ESS' control system.

CHALLENGES AND LESSONS LEARNED

Challenges

Overall challenges One challenge that facilities of any moderate size face is keeping a good overview and control of the list of support modules and their versions. This is in part to avoid the issue of systems that become so old that no one knows how to update them, or wants to change them for fear of breaking a system that 'works'.

One partial solution to this problem is one of standardisation—if you enforce that integrators are only able to pull from a relatively limited collection of support modules, then you can better ensure that there will be fewer conflicts as well as better ease of managing and updating systems.

Upon starting to integrate the two systems (NFS and Conda e3) into a common deployment system, we began to address an issue related to standardisation that we knew would arise: namely that the build and dynamic loading systems had diverged significantly, as well as the specific pool of module versions. This was anticipated, but nonetheless provided a challenge when we started to synchronise across e3 versions.

Challenges specific to NFS e3 While developing the NFS-based solution the e3 team took a lot of inspiration from the capabilities of Conda. Despite some identified potential issues, we still saw a lot of value with that approach and used that as inspiration for our development. Nevertheless, there were quite a few specific issues that arose during this time.

```
require llrfsystem, 3.18.0+0

epicsEnvSet("LLRF_P", "RFQ-010")
epicsEnvSet("IDX", "101")
epicsEnvSet("LLRF_R", "RFS-LLRF-$(IDX)")
# etc., etc.
epicsEnvSet("RKLY", "RFS-Kly-110:")
epicsEnvSet("SEC", "rfq")
epicsEnvSet("RFSTID", "0")

iocshLoad("$(llrfsystem_DIR)/llrfsystem.iocsh")
```

Figure 7: Snippet of IOC startup script

- Non-EPICS related dependencies: At its heart, e3 and

its related tools are a system for managing EPICS modules. If an IOC depends on a system package (either at build-time or run-time) then there is no natural way to track this within our deployment system. This leads to dependencies that are tracked in multiple different locations.
- Moreover, these build-time dependencies are globally managed in the Docker image [7] that we use to perform the CI builds. The build dependencies themselves are simply a list of every single dependency we have at some point needed, with no particular structure or ordering. While this has not yet provided any issues, it is not hard to foresee that this is not a lasting solution.
- Separation of run-time and build-time dependencies: Related to the above, we have no natural way of tracking and managing which dependencies are build-time or run-time ones. This applies to both EPICS and non-epics dependencies.
- IOC dependencies are mixed with run-time configuration: A typical startup script for an IOC contains (see Fig. 7) both dependency descriptions as well as configuration. These are separate types of information that ideally should be managed separately.
- Dependency resolution is extremely simplistic: Unlike most package managers which allow a developer to specify a range of valid dependencies (e.g. `1.0.0 < v < 2`), our dependencies must be specified (nearly) exactly. This adds a lot of maintenance work for module developers.
- NFS availability: Since IOCs need access to the shared build server during startup, NFS availability can be (and has been observed to be) a potential issue. By contrast, since Conda environments are deployed locally to the IOC host, they do not require an external NFS connection in order to run (although they might require other services such as a nonvolatile share for autosave data).

Challenges specific to Conda e3 As had been identified during our early prototyping, Conda faces its own set of technical challenges.
- Combinatorial explosion: If one naïvely builds all downstream dependencies each time a module is built, then one ends up with a situation that is maximally flexible for integrators and developers, but which is far worse from a maintainer's perspective. Each IOC update is potentially its own special case with its own special list of issues to address instead of being a relatively predictable update.
- Managing development vs. production environments: The above can be handled by having a clear separation between development and production environments. Development environments and module pools can be maximally flexible, while production ones can be more restricted. Nevertheless, what is the best way to handle this while using Conda?

Lessons Learned

Upsides Despite the challenges, there turned out to be several advantages to having two parallel build and deployment systems.

One advantage was the cross-polination of ideas from one solution to the other. While we initially abandoned the Conda proof-of-concept to further develop our NFS-based solution, we realised that there was a lot of good functionality that Conda provided that we could adapt into our production system, even if it was architecturally quite different.

Another advantage was that we were able to continue at ESS to explore and develop to proof-of-concept using Conda as a mechanism for EPICS build and deployment. While there were concerns raised initially, None of them turned out to be true showstoppers. Moreover, while developing the NFS-based solution we ran into many technical issues that were difficult to resolve, but that Conda provided a natural solution to.

Finally, on a personal level—which probably does not scale much or translate to other facilities—we can say that our development team learned *a lot*. The need to design our own custom package manager and build and deployment toolchains in a small team is not a small task, and one that improved our understanding of EPICS, GNU Make, and package management in general.

Downsides Not surprisingly, a major downside and challenge that we faced was keeping the two build and deployment systems in sync. There were several aspects to this:
- Coördination of module versions for production systems. One purpose of having a team responsible for the EPICS distribution is to act as curators and gatekeepers. Unfortunately, having two distinct teams and module pools naturally led to divergent module versions which required some coördination to address.
- Divergent build systems. The Conda build system was able to mostly directly use the original PSI `require` without much change, which of course required a lot less maintenance. In contrast, the NFS e3 build system had been modified quite a lot. Synchronising these proved to be quite a challenge.

Furthermore, there is a difficulty when pursuing two opposing goals at the same time: a lot of energy can be wasted on indecision, with many discussions being revisited again and again. Sometimes, like a sailor boarding a boat, you simply need to confidently make the leap and stick with it.

Finally, while as stated above it was beneficial to explore both solutions and to allow for cross-polination, actually maintaining two build and deployment systems meant that regardless of what the final production system looks like we will have produced a lot of redundant code that will ultimately be left unused.

CONCLUSION

Maintaining two separate build and deployment systems is challenging and takes a lot of resources. In particular, coördination between the two will become more work the more established they become.

However, allowing space and resources to develop proof-of-concepts can also be quite valuable. The danger can otherwise be that one gets stuck in a 'local minima' of sorts, an established system that has issues that become more difficult to resolve over time due to how entrenched they become.

When comparing the two specific solutions in question, NFS-based and Conda based, they each have their own issues.

For Conda, there remains an issue with managing the combinatorial explosion that can come from allowing arbitrary package versions. We are currently working on developing the correct workflow to act as gatekeepers for releasing to a production channel that should help address this issue.

For the NFS-based solution using our custom environment manager (`e3-build`), there still remains several difficult issues—in particular, many of those that related to dependency mangement beyond simple EPICS-based ones—that are resolved relatively easily by Conda.

In some sense, the realisation that managing an EPICS environment is a subproblem of managing package environments has led us to re-evaluate Conda as an EPICS package manager.

While at the moment we still exist in a liminal state between EPICS environment solutions, we would like to move towards a more standard and community-accepted environment and package manager. Sadly, there does not at the moment exist such a tool used within the EPICS community. Our hope is that further development of our Conda e3 solution might be able to provide a solution that can be adopted and developed more widely [8].

ACKNOWLEDGEMENTS

This work would not be possible without the work of some giants before us, as well as all of the help that we have received along the way. In no particular order we would like to thank Benjamin Bertrand, Anders Harrison, Dirk Zimoch, Jeong Han Lee, Timo Korhonen, and Wayne Lewis

REFERENCES

[1] EPICS, `https://epics-controls.org/`

[2] e3 documentation, `https://e3.pages.esss.lu.se`

[3] Require, `https://github.com/paulscherrerinstitute/require/`

[4] B. Bertrand and A. Harrisson, "Building and Packaging EPICS Modules With Conda", in *Proc. ICALEPCS'19*, New York, NY, USA, 2020, pp. 223–227. `doi:10.18429/JACoW-ICALEPCS2019-MOPHA014`

[5] Conda, `https://conda.io/`

[6] GitLab CI, `https://docs.gitlab.com/ee/ci/`

[7] Docker, `https://www.docker.com/`

[8] XKCD: Standards, `https://xkcd.com/927/`