ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUDPP03

IMPROVEMENT OF EPICS SOFTWARE DEPLOYMENT AT NSLS-II

A. A. Derbenev[†], Brookhaven National Laboratory, Upton, USA

Abstract

The NSLS-II Control System has workstations and servers standardized to the usage of Debian OS. With exceptions like RTEMS and Windows systems where software is built and delivered by hand, all hosts have EPICS software installed from an internally-hosted and externally-mirrored Debian package repository. Configured by Puppet, machines have a similar environment with EPICS base, modules, libraries, and binaries. The repository is populated from epicsdeb, a community organization on GitHub. Currently, packages are available for Debian 8 and 9 with legacy support being provided for Debian 6 and 7. Since packaging creates overhead on how quickly software updates can be available, keeping production systems on track with development is a challenging task. Software is often customized and built manually to get recent features, e.g. for AreaDetector. Another challenge is services like GPFS which underperform or do not work on Debian. Proposed improvements target keeping the production environment up to date. A detachment from the host OS is achieved by using containers, such a Docker, to provide software images. A CI/CD pipeline is created to build and distribute software updates.

SYSTEM OVERVIEW

The NSLS-II control system is built on Experimental Physics and Industrial Control System (EPICS) infrastructure with a typical controls application being created as an Input-Output Controller (IOC) [1]. In addition to IOCs which are meant to communicate with hardware and other IOCs to implement control logic and functions, the software suite includes a diverse set of higher-level tools, services, libraries, command line and graphical interface applications. Examples are Channel Access command line tools and Python interface, Archiver Appliance, Control System Studio and Phoebus, Olog, Alarm Server, MASAR. While there can be several dozen IOCs per beamline to serve hardware integration and automation needs, tools usually come one installation per workstation, and services come one per beamline.

Controls applications are typically built and run in a specific development and runtime environment. Servers which run IOCs are standardized to use Debian operating system (OS). EPICS base, modules, and tools necessary for IOC development and operation are delivered as Debian packages available from the NSLS-II repository maintained by NSLS-II Controls [2]. EPICS source code is not "debianized" by default and is converted to package format on GitHub thanks to collaboration efforts [3]. When an IOC system is configured, apt package sources are specified appropriately, and a set of default packages is installed via usage of Puppet.

With the development environment made available and any special dependencies manually installed (e.g. vendor-supplied libraries for hardware), IOC systems become ready for building and running EPICS applications. A typical IOC is manually checked out from the internal GitLab or Mercurial repository, built in-place, and registered to run in the system via the sysv-rc-softioc utility. The manage-iocs toolkit provided by the utility serves as a uniform and standard way of running production IOC instances, and provides essential features like detached console access, logging, run/stop/restart control, and status reporting. The approach to application delivery is hence manual, limited to application level, and is version control system (VCS) based for deployment and change management.

KEY CONSIDERATIONS

A multitude of software is involved in running the machine, and development of controls applications is constantly ongoing as updates become available and new controls integration and automation needs emerge. It is prudent to make sure that approaches to software delivery for NSLS-II Controls stay current with evolving technology and requirements. A well-understood and standardized solution brings many benefits from reduced costs of systems scaling and replication to ease of continuous maintainability to long-term sustainability of Controls software infrastructure.

Whatever the approach proposed, it should aim to respect the multitude of solutions, practices, and mechanisms currently employed for NSLS-II Controls applications delivery. When considering any kind of standardization, it is important to recognize that Controls environment is often shared by different developer groups, and many stakeholder parties have their interest in the approach which is to be set as standard. Service developers and maintainers, IOC developers, tool developers, beamline staff, IT, etc. contribute to the evaluation of existing and proposed solutions and make sure that critical needs are met. Several considerations were identified.

Scalability

With NSLS-II Controls spanning over accelerator systems and over two dozen beamlines, it is important for the software delivery approach to be flexible and applicable for all environments which need to be supported. The solution should resolve facility-scale software delivery needs by design, as made possible by an existing set of control system standards which make controls environment mostly uniform across beamlines. Practical example of this consideration is implication that hundreds of IOCs and other apps will need to be managed eventually.

† aderbenev@bnl.gov

Maintainability

Controls software is subject to ongoing updates and changes, e.g. OS versions, EPICS base and modules updates, Python version changes, replacement of existing and integration of new hardware, digital certificates renewal, etc. The approach should be flexible enough to either accommodate such changes easily without incurring high development costs, or be decoupled from these changes by design. The goal is to reduce maintenance requirements in terms of engineer time spent on the support of the delivery system itself. An example of this consideration is that preference is given to mature products with rich support available (e.g. Git for version control, Ansible for automation).

Accessibility

One of main goals for the solution design should be its accessibility for software developers. Preferably capitalizing on industry-grade technology and practices, it should not incur an excessive burden on the process of implementing and introducing changes to controls software. The system should not be targeted only towards software development experts and should provide most of its functionality with minimum entrance threshold (e.g. via defaults, templates, helper scripts etc.). The solution should also be covered with enough documentation. An example of this consideration is that convenience user-facing interface should be provided.

Support of Persistence

One of major needs tied to control system software is preserving applications "persistence", which in this case is defined as any changes generated at and associated with the software runtime. With a delivery mechanism in place, source code and application configurations can be recreated at any time by re-deploying the software. In contrast e.g. for IOCs, persistence usually comes in a form of machine values which are saved in files on the disk and which cannot be recovered by the deployment mechanism if lost. An example design consideration is that the delivery mechanism cannot be "imperative", i.e. it cannot completely overwrite the software instance when deployment is performed without somehow preserving its persistence.

Release and Staging Function

Handling software releases is an important part of the update mechanism. To reduce the cost of unforeseen update issues, a "staging" mechanism is beneficial to provide an easy switch to a stable software version in case of update failure. In its design, the delivery solution should incorporate these considerations by providing a simple and easily reversible release mechanics. An example of this consideration is automatic backup on deployment.

Versioning

Not all issues associated with software updates are immediately, or even just in short-term, discernible. Some issues tend to emerge only on specific conditions or in specific combinations of application parts, and it may take time to discover instability in a deployed update. Versioning allows to mitigate these risks by providing a history of software states which can be at any time recovered to a last known stable condition. The solution employed should allow that to happen both for the software in question, and for its dependencies. One practical example of this consideration is that dependency version information should be available for all software managed.

In-place Production Modifications

NSLS-II is a scientific facility, which means that stable and uninterrupted scientific output of any instrument is the main goal of controls software. Practically it means that controls systems should be mutable enough to incorporate research demands, e.g. support of new hardware, episodic modifications due to experiment needs, prompt production modifications to counter a discovered system issue etc. When applied to software, it means a need to modify production instances of software. The solution should either allow post-factum integration of changes introduced, or provide speedy mechanisms of introducing changes so that the overhead was small enough to not prevent its usage. An example of this consideration is a possibility to run integrity verification checks to make sure that deployed instances are in sync with application configurations stored.

Preliminary Testing

While being a common requirement, in-place modifications which target runtime needs are not the only changes introduced to the system. Just as often, system updates can be handled in a more planned fashion, and since those are to be applied on an already working system, preliminary testing becomes an important part of the process. If not sufficiently covered, this will introduce associated risks which will eventually stagger the update process and prevent rolling updates. A proper solution should allow software testing which does not require any system downtime, either through running the updated software in an isolated test environment, or providing a powerful utility for running test suites. Practical consequence here is that the solution should incorporate testing as a regular part of its workflow, with clear difference between test and production delivery.

Replication of App Instances

Due to standardization efforts, it is common for NSLS-II controls to reuse hardware and software base on different beamlines. The same EPICS driver can be used to support a family of devices, so many IOCs can be compiled from the same source code and deployed in a similar fashion, save for the unique configuration of every separate instance. Instead of treating every application as a unique case with source being separately versioned, it is feasible to use same code reference so that all dependent systems could be managed and updated in a uniform fashion without introducing desynchronization of individual instance versions. The solution employed should allow easy "replication" of existing application instances to create new, similar deployments. A practical example of this is to allow different software instances to be built from the same source code repository, with application configuration being separately recognized.

Ease of Instance Recovery

Deployment of any existing application instance should be straightforward and fast to address the priority purpose of controls to provide uninterrupted scientific research. For cases when recovery is necessary, it should be possible to restore the application function using standard delivery practices without resorting to manual bypasses. Otherwise, non-standard recovery methods would need to be resolved later, which would introduce risks of deployment scheme deterioration. E.g. manually checking out and building an IOC code would result in a production instance being not in sync with the delivery pipeline. An example can be allowing deployment in a single command, or providing tools for easy assimilation of local changes.

Accommodation for Unique Cases

While there is a set of standards available for NSLS-II Controls solutions, they mostly cover implementations which are replicated across several, or all, beamlines. Examples are motion control and detector solutions. However, it is also common for the accelerator and beamlines to have one-of-a-kind hardware requirements which are not seen anywhere else in the system. Standardizing such cases is not feasible, but as they are a part of Controls domain, the solution employed should be able to incorporate these unique software cases. In terms of solution design, an example of that consideration is application instances being treated with a sufficient abstraction level to not rely excessively on standardization implications.

Support of Multiple Platforms/OSs

NSLS-II Controls is mostly standardized to the usage of Debian OS as its IOC runtime. With that, however, there is not a 100% uniformity in terms of OS versions used. Further, some beamline hardware, most often detectors, can some in a “turn-key” format with servers which are essentially frozen at certain Linux distribution for support reason, or even come with Windows support. Some solutions utilize embedded platforms like VME and cPCI. When designing a software delivery solution, it is important to consider and make a decision about the extent of support provided in terms of platform coverage, and whether or not certain standards should be supported or even enforced in that regard. Practically, this consideration can result in exclusion of embedded systems from the list of managed, e.g. because they are not compatible with Ansible automation.

SUPPORT CONSIDERATIONS

As it was discovered from previous initiatives directed towards software delivery (SNACK [4]), two most important factors tied to their long-term success and adoption are investing in training of developers, and having a solid capacity of solution support. The latter includes incorporating new functions requested, addressing discovered issues, dealing with dependencies updates and infrastructure changes, and maintaining the environment in which the tool runs, be it services (LDAP, Ansible, GitLab), hardware (build, testing, orchestration hosts, network), or system configuration (certificates, keys, system users). More specifically, a facility-scale, complex software delivery system requires hardware, software, and configuration upkeep and maintenance:

- CI/CD tools, pipeline, and workflow design will require ongoing investments in services configuration, update, and usage training.
- Depending on how testing is performed and whether a separate testing environment is present, any gateway (i.e. testing-to-production) servers and the network infrastructure will require configuration and further updates.
- If virtualization technology is used, container hosts will require configuration, extension on demand, updates, and general system support, including hardware.
- Based on experiences from the system support and users’ feedback, testing and delivery mechanisms will have to be continuously revised and refined, and updates to the tool chain addressed (e.g. a new GitLab version).

EXPLORED APPROACHES

When designing an application deployment system, one of earliest considerations which appear is defining the scope of the solution. Based on the amount of investment planned, the architecture of the approach can span several levels, from application to server to beamline to facility. Likewise, a set of required features should be defined, which may or may not incorporate testing, staging, backups, redundancy, and other items listed as key considerations above.

At this time and for purposes of improving controls software delivery at NSLS-II, following approaches and their expansiions are being used or explored:

Manual In-place Delivery with Version Control

For purposes of maintaining machine and instrument operations, the existing scheme of manual, application-level software delivery sufficed for years. Familiar to developers, it provides unmatched flexibility and is very friendly to quick fixes and in-place modifications. That convenience, however, comes at a cost of diminishing system knowledge, unaccounted changes, lack of consistent testing and update capacity, effort duplication, and quality creep. Still, the approach can be formalized and leveraged in such a way that most its deficiencies are mitigated, e.g. through more standardization of deployment practices, centralized knowledge bases, and reuse of existing application instances. Some of these deficiencies can be mitigated, e.g. as done in the IOC support module manager SUMO [5].

This approach is a de-facto NLSL-II standard for IOC deployments.

Manual Usage of Pre-built Binaries

An improvement of the existing approach for cases when different applications are using the same source code, introduction of a central binary storage promotes better standardization of environments, e.g. OS versions, while also providing a convenient route to handling software updates on a scale beyond individual application level. Instead of building every application with its own set of dependencies, pre-built binaries are provided, associated with a known assembly of software versions.

This approach is currently being investigated for deployment of NSLS-II AreaDetector software, with one of main challenges being catering to cases which require special dependencies (e.g. ZeroMQ), or platforms (e.g. Windows), to run.

Orchestrated On-demand Delivery

Instead of utilizing manual approach to setting up production software instances, some automation solution can be used so that deployed applications were a product of a well-defined delivery process. A straightforward option would be automation of otherwise manual deployment actions by using a tool like Ansible or Puppet. Another example is rsync-dist tool which delivers built binaries to remote servers while also being decoupled from SVC [6].

This approach is used to set up EPICS developer environments on IOC servers with Puppet installing necessary packages. SNACK utility uses Ansible to perform IOC builds and deployments.

Pipelined Build and Delivery with CI/CD

A solution which is more in line with industry-grade DevOps, a Continuous Integration and Continuous Delivery (CI/CD) pipeline would allow to create a more wholesome and solid workflow for accommodating changes from development to test to production. There are many open source and proprietary solutions available: Jenkins, Hudson, GitLab, Travis, AppVeyor, etc. Each of these options has various advantages and disadvantages. For example, Travis and AppVeyor are hosted service which reduces IT expertise but limits the kinds of testing that can be performed. Jenkins and Hudson are web containers that need to be managed on site but allow for more complex tests. An example of this approach is IOCs delivery system at FRIB [7].

At a glance, to meet the CI/CD requirements of NSLS-II controls, the CI engine would need to support:

- Multiple concurrent & ordered builds
- Multiple builders (maven, make, etc.)
- Support for multiple languages
- Multiple VCS systems and VCS hosting services
- Support for creating release jobs
- Support for containers and images

Current plans are to use this approach to set up a delivery pipeline for EPICS tools and services, namely Control System Studio, Phoebus, Olog, Alarm Server, and others.

Virtualization

Controls applications typically run on physical or VM hosts which come with a full developer environment and appear as a complete Debian OS system for as much as any application is concerned. These systems have a well-defined beamline affiliation and appropriate network configuration. A significant shift from this approach would be switching to a container-based delivery, or to usage of lightweight virtual systems, where separate applications come as images from which containers are being created and run. That approach would decouple the delivery scheme from considerations associated with underlying hardware or host OS, allow easier isolation, and compatibility with existing CI/CD approaches. However, this approach may not be appropriate for user interface applications and software which deals with heavy data transfer or processing.

In exploration of this approach, a sample solution was created based on the usage of Docker. An image is provided for EPICS base and modules, and IOC images can be made to run containers on any system which provides Docker support. With a special toolkit, any IOC can quickly be converted to the image format with a potential to be run in a container on most system servers. Further solution refinement is required to address various key considerations mentioned.

CONCLUSION

Since the work has begun to improve EPICS software deployment at NSLS-II, many realizations have been made about requirements posed by parties involved in controls software development, system and infrastructure management and support. Lots of insights were derived from SNACK experience, and several emergent needs were identified which can potentially be resolved by introducing a more unified controls software delivery mechanism. Examples are new EPICS services such as Phoebus, new software versions such as EPICS 7, and widely used software such as AreaDetector. Currently, investments are made to identify and design a wholesome approach which could accommodate all these and other needs, with specific focus being put into utilizing modern CI/CD tools and approaches.

REFERENCES

[1] EPICS Home, https://epics.anl.gov

[2] NSLS-II EPICS Repository, https://epicsdeb.bnl.gov/debian

[3] GitHub epicsdeb Packaging, https://github.com/ep-icsdeb

[4] SNACK toolkit, https://epics.anl.gov/meetings/2018-06/talks/06-15/AM/9.2-SNACK.pdf

[5] Building EPICS Support Modules with SUMO, in *Proc. EPICS Collaboration Meeting 2019*, Aix-en-Provence, France, Jun. 2019, p. 1, https://indico.cern.ch/event/766611/book-of-abstracts.pdf

[6] Deploying EPICS Applications with rsync-dist, in *Proc. EPICS Collaboration Meeting 2019*, Aix-en-Provence, France, Jun. 2019, p. 1, `https://indico.cern.ch/event/766611/book-of-abstracts.pdf`

[7] Continuous Delivery and Deployment of EPICS IOCs at FRIB, in *Proc. EPICS Collaboration Meeting 2019*, Aix-en-Provence, France, Jun. 2019, pp. 1-2, `https://indico.cern.ch/event/766611/book-of-abstracts.pdf`