17th Int. Conf. on Acc. and Large Exp. Physics Control Systems ICALEPCS2019, New York, NY, USA JACoW Publishing
ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA014

BUILDING AND PACKAGING EPICS MODULES WITH CONDA

B. Bertrand[*], A. Harrisson, ESS, Lund, Sweden

Abstract

Conda is an open source package, dependency and environment management system. It runs on Windows, macOS and Linux and can package and distribute software for any language (Python, R, Ruby, C/C++…). It allows one to build a software in a clean and repeatable way. EPICS is made of many different modules that need to be compiled together. Conda makes it easy to define and track dependencies between EPICS base and the different modules (and their versions). Anaconda's new compilers allow conda to build binaries that can run on any modern linux distribution (x86_64). Not relying on any specific OS packages removes issues that can arise when upgrading the OS. At ESS, conda packages are built using gitlab-ci and pushed to a local channel on our Artifactory server. Using conda makes it easy for the users to install the EPICS modules they want, where they want (locally on a machine, in a docker container for testing…). All dependencies and requirements are handled by conda. Conda environments make it possible to work on different versions on the same machine without any conflict.

INTRODUCTION

Distributing binaries of C/C++ programs is still not an easy task today. Compiling locally and making modules available via an NFS share is a common solution in the EPICS community. Operating system package managers like yum or apt are not new of course but they require one to build different packages for each Linux distribution. In recent years, language specific package managers became more and more popular. There is no new programming language without its own package manager. All modern languages have their own: Python (pip), JavaScript (npm), Rust (cargo)... It's even coming to C/C++ with Conan [1] for example. Conda [2] is another popular solution that is not linked to a specific OS or language. Conda makes it possible to build, distribute and install binary packages with all their dependencies allowing users to concentrate on their task and be more productive.

EPICS AT ESS

e3

The EPICS Environment at ESS named e3 [3] is based on the concept developed by Dirk Zimoch at PSI that allows dynamically loading of EPICS module resources. All the IOCs use the same executable from EPICS base (softIoc) that is not linked with any libraries. Shared libraries and resources are all loaded at runtime by the require [4] module that also performs dependency resolution. An IOC is started by running the *iocsh.bash* script and a startup command script that describes the modules to load. The *iocsh.bash* script is a wrapper that starts a softIoc. For *require* to load the required modules, they have to be in a defined file structure as depicted in Fig. 1.

```
/epics/base-3.15.5/require/3.0.0/siteMods/
├── asyn
│   └── 4.33.0
│       ├── db
│       ├── dbd
│       ├── include
│       └── lib
├── autosave
│   └── 5.9.0
│       ├── bin
│       ├── db
│       ├── dbd
│       ├── include
│       └── lib
```

Figure 1: e3 tree structure.

The current e3 implementation includes some scripts making it easy for developers to build locally the modules they want and has proven to work very well for development.

Development vs Production

With an NFS share for production use, some extra complexity appear compared to local development:

- keeping multiple versions of modules
- OS compatibility

For production use, the tree structure must allow adding new versions of modules while keeping the existing ones. This is easy for modules that nothing depend on, but becomes quickly very difficult to manage when rebuilding dependencies is required. When a new module version is added, all modules that depend on it have to be recompiled to use this new release (due to ABI compatibility). But their own version might not have changed. The notion of build number is thus needed to have different builds of a same version. This is exactly one of the problem that package managers are designed to handle.

Another issue is that the modules can only be used on the OS they were built on. Having modules built for different Linux distributions, in the same tree structure, requires one to modify the EPICS_HOST_ARCH, changing it from linux-x86_64 to something specific to the distribution, and compiling each module on different OS. This is not only true between Linux distribution like Debian and CentOS but even between major versions of a distribution like CentOS 7 and CentOS 8. As we'll see, this is also something conda can solve.

CONDA

Conda is an open-source, cross-platform, binary package manager first released in 2012. It comes from the Python

[*] benjamin.bertrand@esss.se

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA014

community and was created to help data scientist to install libraries with C/C++ dependencies. As such it is general and language-agnostic. Conda is already used in the EPICS community for pvaPy [5]. PSI [6] also has some conda recipes for epics-base.

Conda Concepts

A **conda package** is a compressed tarball file (.tar.bz2) or .conda file that contains a collection of files to be installed and some metadata (under the info directory). The files to install can be system-level libraries, text files or binaries. The package filename includes the package name, version, and build string joined together by hyphen as shown in Fig. 2.

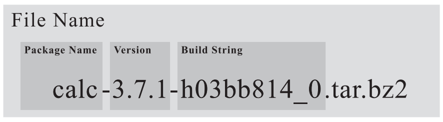

Figure 2: Package naming convention.

The build string *h03bb814_0* includes a hash based on the package dependencies (*h03bb814*) and the build number defined in the recipe (*0*).

Not all packages include a hash in the build string. A hash is only added if the recipe has an explicit dependency in the build, host, or run sections that has a matching entry in *conda_build_config.yaml* which is a pin to a specific version. For the calc package mentioned above, the hash is based on the dependencies described in Fig. 3.

```
{
    "epics_base": "7.0.3",
    "c_compiler": "gcc",
    "require": "3.1.0",
    "seq": "2.2.6",
    "sscan": "2.11.2",
    "cxx_compiler_version": "7",
    "cxx_compiler": "gxx",
    "c_compiler_version": "7"
}
```

Figure 3: calc package hash input.

The build string is an important information as it makes it possible to rebuild the same version of a module with different dependencies.

A **conda channel** is a repository of conda packages. Conda packages are downloaded from remote channels, which are URLs to directories containing conda packages. By default conda will download packages from https://repo.anaconda.com/pkgs/. Many other channels are available. conda-forge [7] is a popular community-led channel with more than 7000 packages. Hosting a conda channel only requires a webserver to serve the packages (following a defined directory structure) and index file (repodata.json). Private channels can be created on anaconda.org or on a private server. JFrog Artifactory [8] is a universal repository manager that supports conda channels out of the box.

A **conda environment** is a directory that contains a specific collection of conda packages. A conda environment is a virtual environment that is completely isolated from other environments. Each environment includes the subdirectories /bin, /etc, /include, /lib, /share, mimicking the Linux Filesystem Hierarchy as shown in Fig. 4.

```
/opt/conda/envs/epics
├── base
├── bin
├── compiler_compat
├── conda-meta
├── etc
├── include
├── lib
├── man
├── share
├── ssl
└── x86_64-conda_cos6-linux-gnu
```

Figure 4: Conda environment structure.

Conda won't modify system dependencies. An environment can be installed anywhere and doesn't require superuser privileges. As conda packages can be installed anywhere, binaries and libraries shall be relocatable. This is done using *RPATH*, the run-time search path hard-coded in an executable file or library. The RPATH is set to a long placeholder prefix during the package creation. At install time this prefix is replaced with the path of the conda environment.

Conda tracks and manages the dependencies between every installed package in an environment. An environment can only include one version of a package. Installing a new version will remove the previous one and update any required dependencies. To work on different versions of a package, you have to create different environments.

To use an environment, it should be activated by running the *conda activate <env>* command. Activating an environment puts the environment bin directory in the PATH so that installed binaries can be found. The activation can perform other actions, like exporting variables, depending on the packages installed. The *conda deactivate* command puts back the shell in its previous state.

CONDA-BUILD

Building a conda package requires a recipe. A **conda recipe** is a directory that contains at least a *meta.yaml* file describing the package. Only the name and version are mandatory fields. But a useful recipe usually includes the source (this can be an url, a git repository), build number, build and run requirements and some commands to test the built package. The command to build the package can be included in the meta.yaml format (using the script field). But by default conda expects a build.sh (on Linux and OSX) or build.bat (on Windows) script.

To create a package, the conda-build command:

1. Reads the recipe metadata
2. Downloads the source into a cache.
3. Extracts the source into the source directory.
4. Applies any patches.
5. Re-evaluates the metadata, if source is necessary to fill any metadata values.
6. Creates a build environment and then installs the build dependencies there.
7. Runs the build script. The current working directory is the source directory with environment variables set. The build script installs into the build environment.
8. Performs some necessary post-processing steps, such as shebang and rpath.
9. Creates a conda package containing all the files in the build environment that are new from step 5, along with the necessary conda package metadata.
10. Tests the new conda package if the recipe includes tests by creating a test environment with the newly created package and its dependencies and running the test scripts.

Build Variants

The nature of binary compatibility (and incompatibility) means that we sometimes need to build binary packages (and any package containing binaries) with several variants to support different usage environments. This is the case for EPICS modules. The version of a module used at runtime must be identical to the version that was used at build time. Strictly speaking, it should be a version that is ABI compatible. In EPICS world, very few modules respect the semantic versioning, meaning that using the same version (major.minor.patch) is often needed.

The runtime requirement to use can be specified with the *run_exports* key as shown in Fig. 5.

```
build:
  number: 0
  run_exports:
    - {{ pin_subpackage('calc', max_pin='x.x.x') }}
```

Figure 5: Recipe run_exports.

With the *pin_subpackage* function, pinning values can be decoupled from recipes. The above specifies that if a package lists calc has a host dependency, the version used at build time will automatically be added as a dependency at runtime. If the calc version 3.7.1 is used at build time, the pining expression at runtime would evaluate to $>= 3.7.1, < 3.7.2a0$. While if using max_pin='x.x', it would evaluate to $>= 3.7.1, < 3.8.0a0$.

The version of the dependencies doesn't have to be included in the recipe. Those can be defined in a separate *conda_build_config.yaml* file, which can be global or local. Using a global file has two benefits. It enables you to separate the version of the requirements from the recipe (you don't have to update the recipe itself to create a new build) and it keeps the versions of all dependencies centralized ensuring all packages use compatible versions.

Portable Binaries

Making C/C++ binary packages that can run on many Linux distribution is tricky [9]. The binary depends on the libstdc++.so and libgcc_s.so libraries. If you compile on a recent Linux distribution, the libstdc++.so library your binary was linked against might contain symbols that are not present on an older distribution. To ensure compatibility, one approach is to compile on the oldest Linux distribution you want to support. Anaconda used to compile with CentOS 5 and conda-forge with CentOS 6. One disadvantage is that you are usually forced to use the old compiler that comes with the distribution.

At the end of 2017, Anaconda switched to their own compilers. This comes with a lot of benefits: new and thus improved compiler capabilities, including better security and performance. It also means that you don't have to rely on an old Linux distribution to build compatible binaries. The compilers come with their own libstdcxx-ng and libgcc-ng libraries. conda-forge started the migration to the new compilers at the end of 2018 and successfully completed it in January 2019.

Using conda and anaconda compilers, binaries that run on any Linux distribution can be built without much effort. This requires of course that all the dependencies are built with conda, but we can rely on that for the large number of packages available on conda-forge (readline, ncurses, perl, boost...). This makes the goal to compile once and run anywhere a reality.

EPICS AND CONDA

Conda has many features that fit very well with the EPICS use at ESS. By using conda, dependency management can be left to the package manager and performed at install time instead of runtime (by require). Dependency management is a hard problem. It can in fact be expressed as a SAT problem [10] (Boolean satisfiability problem) which is NP-complete. In a conda environment, only one version of a package can exist. Require doesn't have to select the proper versions of the modules compatible.

This is really similar to what is done in dynamic languages like Python. Python modules are installed by a package manager (like pip or conda) in a virtual environment and the Python program loads the required modules at runtime but it doesn't handle the version to load. Only one version is expected to be present in the environment. The exact version of the Python modules needed are defined in a separate file (usually *requirements.txt* when using pip).

A conda environment can be described in an *environment.yml* file, which is the equivalent of pip *requirements.txt*. The versions of the modules to use don't have to be in the

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA014

startup command script anymore. By doing so we separate the requirements from the program, following configuration management best practices.

Putting everything we saw together, we can create recipes for EPICS modules. They all depend on epics-base and require and are quite similar. A new recipe can be created using a cookiecutter [11] template. The cookiecutter command in Fig. 6 will create the files shown in Fig. 7.

```
(base) CI0011906:~ $ cookiecutter git+https://gitlab.esss.lu.se
/ics-infrastructure/cookiecutter-e3-recipe.git
company [European Spallation Source ERIC]:
module_name [mymodule]:
summary [EPICS mymodule module]:
Select module_home:
1 - gitlab.esss.lu.se
2 - github.com
Choose from 1, 2 [1]:
module_version [1.0.0]:
```

Figure 6: Recipe creation.

```
mymodule-recipe/
├── .gitignore
├── .gitlab-ci.yml
├── LICENSE
├── README.md
├── recipe
│   ├── build.sh
│   └── meta.yaml
└── src
    └── Makefile
```

Figure 7: Recipe template.

The *build.sh* script, shown in Fig. 8, is identical for all EPICS recipes.

```
#!/bin/bash

# LIBVERSION shall only include MAJOR.MINOR.PATCH for require
LIBVERSION=$(echo ${PKG_VERSION}| cut -d'.' -f1-3)

# Clean between variants builds
make clean

make MODULE=${PKG_NAME} LIBVERSION=${LIBVERSION}
make MODULE=${PKG_NAME} LIBVERSION=${LIBVERSION} db
make MODULE=${PKG_NAME} LIBVERSION=${LIBVERSION} install
```

Figure 8: common build.sh script.

Figure 9 shows a typical example of an EPICS module meta.yaml file (calc).

The source section refers to an archive of the module. The local src directory can be used to add extra files, like a specific *Makefile* needed to use require. In the build section, *run_exports* is used to impose the runtime dependency of packages using this module.

In the requirements, the build tools (compilers) and host requirements are listed.

The test section uses the run-iocsh [12] tool to test the compiled module. *run-iocsh* is a small Python script that was developed for testing purpose. The *iocsh.bash* script to

```
{% set version = "3.7.1" %}

package:
  name: calc
  version: "{{ version }}"

source:
  - url: https://github.com/epics-modules/calc/archive/R3-7-1.tar.gz
    sha256: cb302bc87b8c3364ac332cdd70e8bbc9ac9b5cb7b31a18f815893aa2ba0c684e
  - path: ../src

build:
  number: 0
  run_exports:
    - {{ pin_subpackage('calc', max_pin='x.x.x') }}

requirements:
  build:
    - {{ compiler('c') }}
    - {{ compiler('cxx') }}
  host:
    - epics-base
    - require
    - sscan
    - seq

test:
  requires:
    - run-iocsh
  commands:
    - run-iocsh -r calc

about:
  home: https://github.com/epics-modules/calc
  license: EPICS Open License
  summary: "APS BCDA synApps module: calc"
```

Figure 9: calc meta.yaml recipe.

run an IOC starts an interactive IOC shell meant to run in the foreground. *run-iocsh* runs *iocsh.bash* using subprocess and sends the exit command after a delay. It raises an exception if an error occurred. This command makes it easy to perform a basic test on any new compiled modules. It ensures that the module and its dependencies can be loaded by require.

CONDA-BOT

At ESS, we have an internal GitLab [13] server to store our code. GitLab includes built-in CI/CD tools to easily apply continuous software development. Each recipe includes a *.gitlab-ci.yml* file that points to the same template shown in Fig. 10.

This *.gitlab-ci.yml* file builds and uploads the recipe to Artifactory on every push to the master or "release-xxx" branches.

Using GitLab webhooks events, we can automate even more the workflow to re-build recipes. Indeed when a new version of a recipe is built, the recipes depending on that package should also be rebuilt to ensure binary compatibility between the different packages.

conda-bot [14] is a GitLab bot that was designed for this purpose. It listens to webhooks events and triggers several actions:

- create a merge request to update the recipe when a tag is pushed to a repository from the epics-modules group.
- update the global conda_build_config.yaml file to pin packages to the latest version published to Artifactory.
- trigger the build of dependent recipes

```yaml
build:
  tags:
    - docker
  stage: build
  image: ${CONDA_BUILD_IMAGE}
  script:
    - mkdir pkg
    - >
      conda-build --error-overlinking
      --override-channels -c ${ARTIFACTORY_CONDA_CHANNEL}
      --skip-existing
      --output-folder pkg recipe
  artifacts:
    expire_in: 24h
    paths:
      - pkg/

release:
  tags:
    - docker
  stage: release
  image: registry.esss.lu.se/ics-docker/jfrog:latest
  script:
    - 'echo "conda channel: ${ARTIFACTORY_CONDA_CHANNEL}"'
    - jfrog rt config artifactory --url=${ARTIFACTORY_URL} --apikey=${ARTIFACTORY_API_KEY}
    - jfrog rt upload "pkg/linux-64/*.tar.bz2" ${ARTIFACTORY_CONDA_CHANNEL}/linux-64/
    - jfrog rt upload "pkg/noarch/*.tar.bz2" ${ARTIFACTORY_CONDA_CHANNEL}/noarch/
  only:
    - master
    - /^release-.*$/
```

Figure 10: Template .gitlab-ci.yml

conda-bot is based on gidgetlab [15], a Python framework to interact with GitLab API and to write GitLab bots: applications that run automation on GitLab, using GitLab Web-Hooks and API. Using webhooks, it can monitor changes on a group of repositories. When pushing a tag to one EPICS repository, the bot can automatically create a merge request on the linked recipe to bump the version number. If the build of the package is successful (this includes the tests part of the recipe), the merge request is automatically accepted. The package is built again (on the master branch) and uploaded to the conda repository. In case of failure, the recipe needs some manual modifications and an e-mail is sent to the user who pushed the tag on the original repository.

Once package "A" has been released, its version in the global *conda_build_config.yaml* is updated. This is also done automatically by the conda-bot. Once done, the build of the reverse direct dependencies is triggered. Those reverse dependencies are computed by parsing the repodata.json file, which is the index of the conda channel. The recipe of those packages don't have to be updated. The version of the requirements is not pinned inside the recipe but taken from the *conda_build_config.yaml*. Triggering a build will create a new package (with a new hash in the build string) based on the new released version.

CONCLUSION

Using conda for building and deploying EPICS modules is an alternative solution to using an NFS share. It brings a lot of benefits that come with a package manager. Conda makes is easy to manage dependencies and build different variants of a package. It separates the dependencies from the runtime (require). The anaconda's compilers make it possible to benefit from a recent version of gcc and to build portable (cross-distribution) Linux binaries. Being able to install binaries locally is a great gain for the developers. It makes all the phases of a project more effective: development, testing and deployment. This is still a proof of concept and hasn't been used to deploy IOCs in production. But conda is already used to deploy epics-base as well as applications like the Channel Access and pvAccess gateways.

ACKNOWLEDGEMENTS

This work wouldn't be possible without all the tools and infrastructure provided by the conda and conda-forge community. I'd also like to thank Dirk Zimoch for his work on require and Jeong Han Lee for his work on e3.

REFERENCES

[1] Conan, https://conan.io

[2] Conda, https://conda.io

[3] ESS e3, https://github.com/icshwi

[4] PSI require, https://github.com/paulscherrerinstitute/require

[5] pvaPy, https://github.com/epics-base/pvaPy

[6] PSI conda recipes, https://github.com/paulscherrerinstitute/conda-recipes

[7] conda-forge, https://conda-forge.org

[8] JFrog Artifactory, https://jfrog.com/artifactory

[9] Break the Chains of Version Dependency, http://www.crankuptheamps.com/blog/posts/2014/03/04/Break-The-Chains-of-Version-Dependency

[10] SAT problem, https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[11] cookiecutter, https://cookiecutter.readthedocs.io

[12] run-iocsh, https://gitlab.esss.lu.se/ics-infrastructure/run-iocsh

[13] GitLab, https://about.gitlab.com

[14] conda-bot, https://gitlab.esss.lu.se/ics-infrastructure/conda-bot

[15] gidgetlab, https://gidgetlab.readthedocs.io