ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC007

CONTINUOUS INTEGRATION AND DEBIAN PACKAGING FOR RAPIDLY EVOLVING SOFTWARE*

A. Barker†, J. Georg, M. Hierholzer, M. Killenberg, T. Kozak, D. Rothe, N. Shehzad, C. Willner
Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, Hamburg, Germany

Abstract

We describe our Jenkins-based continuous integration system and Debian packaging methods, and their application to the rapid development of the ChimeraTK framework. ChimeraTK is a C++ framework for control system applications and hardware access with a high level of abstraction and consists of more than 30 constantly changing interdependent libraries. Each component has its own release cycle for rapid development, yet API and ABI changes must be propagated to prevent problems in dependent libraries and over 60 applications. We present how we configured a Jenkins-based continuous integration system to detect problems quickly and systematically for the rapid development of ChimeraTK. The Debian packaging system is designed to ensure the compatibility of binary interfaces (ABI) and of development files (API). We present our approach using build scripts that allow the deployment of rapidly changing libraries and their dependent applications as Debian packages.These even permit applications to load runtime plugins that draw from the same core library, yet are compiled independently.

INTRODUCTION

Minor release upgrades to software libraries typically exhibit binary compatibility. This ensures that projects reliant on these libraries can depend on a stable application binary interface (ABI), in addition to having compatible source code, known as the application programming interface (API).

Maintaining ABI compatibility requires significant effort and hampers the flexibility needed for a rapidly evolving software framework. On the ChimeraTK project [1], the needs of control system applications drive new feature development, and the release timelines should be short. Hence we have decided against preserving binary compatibility for minor software releases. However, this decision comes with the important drawback of always having to re-compile all dependent projects. This warrants special attention when creating software packages and setting up a continuous integration (CI) system.

DEBIAN PACKAGING

Library packages are binary compatible for the lifetime of a distribution. A binary incompatible package requires a different package name. This is solved by incorporating major and minor version numbers into the package name, so that different versions are formally completely different packages. Binary compatible patches remain viable since the patch level is not part of the package name.

A library depending on changing binary sources is not itself binary stable, even without source code changes. To enable new releases based on changes to the dependencies, we have extended the minor version to include a build version which changes as dependencies change. This build version also contains the distribution code name. Since this is part of the minor version, it also becomes part of the so-version, such that the C++ linking layer is taking it into account. For example: for major version 3, minor version 11, with Ubuntu20.04 code-named "focal", and build number 1, the Debian package is named `libchimeratk-deviceaccess03-11-focal1` and the .so-file is `libChimeraTK-DeviceAccess.so.03.11focal1`.

Our packaging script [2] has a dependency database that stores all dependency versions for each build. Then the build version is increased if the dependency versions change. In addition, the script has an inverse dependency lookup mechanism to identify all libraries that depend on the library being rebuilt. Then their build versions are also increased and they are recompiled, so resulting in a new, binary consistent ensemble of libraries.

Debian packages for applications are usually not rebuilt, so untouched applications remain intact and unaffected by changes in the framework. Packages for applications also have the major, minor, and build version in their package names. Thus, there is a dedicated package for each release. In contrast to libraries, packages from the same application exclude each other's installation. This is done by introducing a virtual package via the the package metadata, noting "provides" and "conflicts" with the package base name without version number. However, the use of package names with version number allows for easy roll-back to a previous version if needed.

Binary Compatible Plugins

ChimeraTK DeviceAccess is a C++ library that provides access to hardware or device servers. Different communication protocols are accommodated by so-called device backends. These include PCI express and Linux UIO [3]. Other backends accommodate various control system middleware like DOOCS [4], EPICS [5] and OPC UA [6]. Generic applications, like the graphical user interface QtHardMon or the DeviceAccess Python bindings, are not linked against all the backends. This prevents the user form having to install all of those control system software stacks, even if they will not be used. DeviceAccess has a runtime loading mechanism to make these backends work with the generic applications, and this runtime plugin loading requires binary compati-

* We acknowledge support from DESY (Hamburg, Germany), a member of the Helmholtz Association HGF.
† anthony.barker@desy.de

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC007

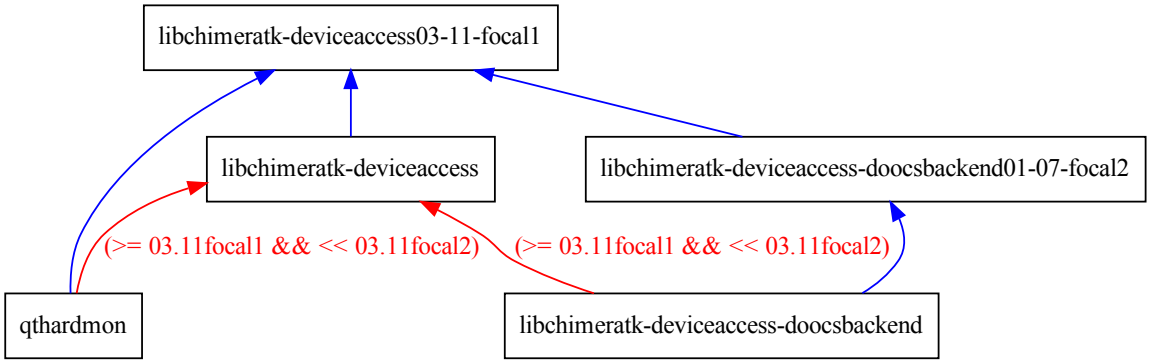

Figure 1: Diagram of Debian package dependencies of a generic DeviceAccess applications, `qthardmon`, and backend `libchimeratk-deviceaccess-doocsbackend`. The dependencies to the `libchimeratk-deviceaccess` meta-package (red) have an explicit version number. These dependencies are a ranges to match the major and the extended minor version since the actual version number also contain the patch level and the Debian build version. The blue dependencies do not have an explicit version in the metadata because they refer to a package with the version number in the package name.

bility. Consequently, the application and the DeviceAccess backend have to be compiled and linked against the same version of DeviceAccess. The installation of a compatible version of a backend must be ensured, especially when the application is being updated.

A simple approach would use the development package, which omits the version number in its package name (e.g., `libchimeratk-deviceaccess-dev`) and always contains the latest version. If both the application and the backend package were to depend on the DeviceAccess development package, it would ensure consistent updates. However, the development packages also depend on other development packages. Consequently, this would require the installation of the entire development stack for both the application and the backend. The development stack does not only include all the header files but also the compilers, which is not wanted on productive systems. To circumvent this issue, supplementary meta-packages for DeviceAccess and the backends have been introduced.

DeviceAccess features a meta-package[1], called `libchimeratk-deviceaccess`, which depends on the correct version of the DeviceAccess library. It serves as a dependency anchor for generic applications and loadable backends. The generic application depends on the *exact* version of this meta-package. These relationships are diagrammed in Fig. 1.

In addition to the package with the version number in the package name, each backend also has a package without the version number. It contains the shared object symbolic link without version number (e.g. `libchimeratk-deviceaccess-doocsbackend.so`) which is usually part of the development package. Like the generic applications, this backend package also depends on the *exact* version of `libchimeratk-deviceaccess`. The development package depends on the meta-package to provide the symbolic link, and brings the headers and build chain files so a C++ application can link against the versioned .so-file at compile time.

In contrast to regular applications, which are uniquely associated with their backends, generic applications must be inter-operable with every backend and so must recompile every time a new DeviceAccess package is built, so as to have an installable version that matches `libchimeratk-deviceaccess`. This is accomplished through their exact version dependency. As regular C++ libraries, the backends must also be recompiled, but this is already accomplished by the inverse dependency mechanism.

Now if a newer version of the backend is being installed, this will update `libchimeratk-deviceaccess` to the latest version, which in turn will update all the applications that depend on it (and all other backends), because of the exact version dependency. If a new version of a generic application is installed, this will update all backends and the other generic applications. Thus, a binary compatible backend is always available for the applications to load, even though the DeviceAccess library does not provide binary compatibility between different releases.

A regular application usually knows which backends it needs. It does not use the plugin mechanism and links against the versioned .so-file at compile time. So, different applications can use different versions of the backend without interfering with each other, and the library is automatically installed via the Debian packaging mechanism. As described above, they do not have to be re-packaged with each DeviceAccess release.

[1] an empty package which provides additional dependencies

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC007

CONTINUOUS INTEGRATION WITH JENKINS

Continuous integration is an invaluable tool to improve the code quality and detect code issues early. In a modular framework with multiple interdependent libraries the two most important pieces of information are: "Is the project OK?" and "Do changes in a library break any downstream projects?".

For ChimeraTK and its dependent applications, we have set up a Jenkins server [7] which performs the following tests for each project:

- Check that the code compiles
- Check for compiler warnings
- Run unit tests, including code coverage report
- Check for memory leaks using an address sanitiser build
- Check for timing races using a thread sanitiser build
- Build for multiple target platforms
- Make debug and release builds
- Use multiple compilers

We use a dedicated high performance build server with 256 cores and 1 TB of RAM. The ChimeraTK libraries make extensive use of C++ templates, which causes long compile times of several minutes for a single project, even when doing parallel builds on the high performance build server. Build times are particularly long when network-based communication protocols with long timeouts are involved, or when algorithmic behaviour is tested against complex simulations. Running the extensive unit and integration test suites can take longer than half an hour, and this is for one project among many.

ChimeraTK is a modular framework which comes with more than 20 interdependent libraries and tools, and our software stack contains more than 30 client applications. In combination with the high demands of the individual projects, this presents a big challenge to the CI system.

To check that downstream projects are not broken by changes in an upstream project, Jenkins' built-in dependency resolver triggers all of the dependent projects. Re-compiling all downstream projects is especially essential in our situation where binary compatibility is not ensured. With many projects each taking several minutes to compile and longer still to run the tests, the total execution time for the CI chain quickly adds up and can exceed our computing resources. The problem is exacerbated by the fact that Jenkins does not resolve diamond dependencies (see Fig. 2). Even when using the option to suppress rebuilds if an upstream project is building, it triggers downstream libraries and nearly all application projects multiple times. This consumes significant computing resources. It can also cause many "false positive" failed jobs. This can happen in the case of a binary incompatible change, where the downstream project is already triggered before all its dependencies have been recompiled against the new code base, causing the job to fail.

The typical time for the whole CI chain to execute was more than 5.5 hours if the most upstream projects were triggered. This is far too resource intensive to be done for each commit. To be useful for the developer, the CI signal that something has been broken should be made available quickly, in a few minutes at most. Otherwise it cannot be integrated into the development workflow and provides no significant benefit over a nightly build.

In addition, the heavy load sometimes overloads the Jenkins server, making it unresponsive and resulting in even more failed jobs due to Jenkins running out of memory (although this occurs while running on a separate machine, not on the build host).

Fast Track and Nightly Builds

To get faster response times, we implemented a so called "fast track", which is a stripped down version of the CI chain with only the essential tests. Each project gets split up into two Jenkins jobs: the "fasttrack" build job, which just compiles the library or application without its tests, and the "fasttrack-testing" job, which compiles and executes the tests. The idea here is to allow that downstream projects can start to build once the parent build is completed, without waiting for the extensive tests to compile, which can take much longer than compiling the libraries themselves. This speeds up the entire build chain. In addition, the fast track is only executed for one target platform and one build type: a debug build on Ubuntu20.04. We also use Groovy scripts to steer Jenkins into avoiding diamond dependencies by implementing a dependency database. These measures result in reducing the build time for the ChimeraTK libraries to around 20 minutes, hasten applications compilation, and deliver preliminary results typically within half an hour.

However, the turn-around time could not be reduced as much as desired. Even a half hour of waiting is too long to be included in our work flow. In addition, the wait time is inconsistent. For some downstream projects with only one or two dependencies, the build time is usually well under 5 minutes, but can take up to 3 hours if other chains are triggered in parallel.

Triggering all dependencies from within a job has other drawbacks: mostly in the form of excessive false-positive build job failures. Build jobs fail if Jenkins cannot start or finish them. Also, either the homemade dependency database or a job may crash due to Jenkins running out of memory. This problem compounds because the calling job is counted as failed as well. Then jobs often fail for unknown reasons unrelated to the quality of the software under test, and there seem to be a wide variety of root causes. For instance, if the default branch is named "main" rather than "master". These problems compound since if a triggered build crashes, the calling job is then also labelled as failing. This cascades

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC007

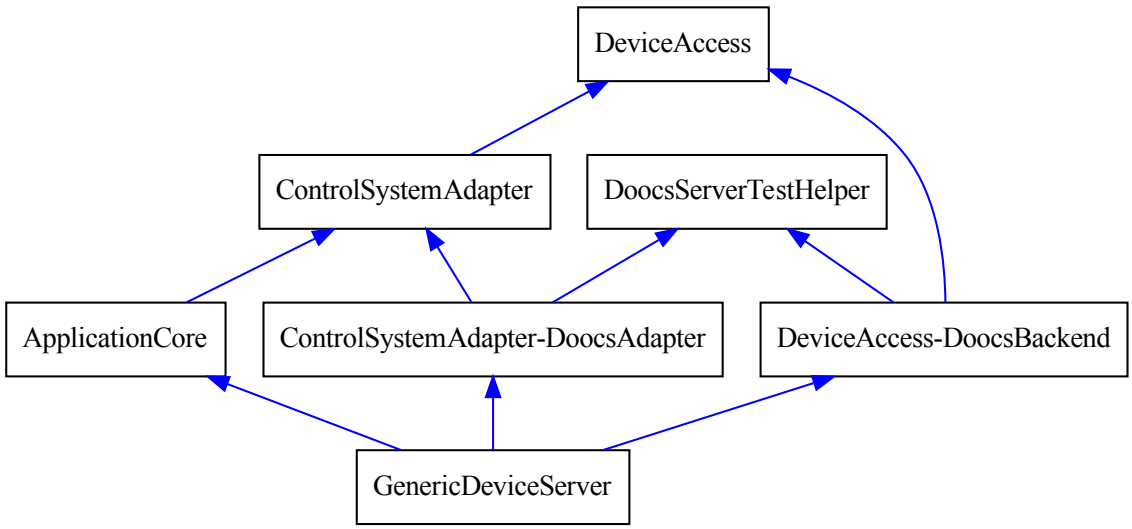

Figure 2: Diamond dependencies of the ChimeraTK GenericDeviceServer. The server depends on ApplicationCore and the ControlSystemAdapter-DoocsAdapter, which both depend on the ControlSystemAdapter. A second diamond is formed by the ControlSystemAdapter-DoocsAdapter and DeviceAccess-DoocsBackend, which both depend on the DoocsServerTestHelper. And a third, larger diamond is formed with DeviceAccess at the top, because both the ControlSystemAdapter and the DeviceAccess-DoocsBackend depend on it.

through the dependency tree, so a single failing application project, that may actually work perfectly, and which depends on many ChimeraTK libraries can make all its library jobs throughout the dependency tree get marked as "failed". This leads to a dramatic over-reporting of job failures with the Jenkins dashboard filled with reports of failed jobs, even though more than 90% of them have no legitimate issues. Consequently, usefulness of Jenkins is degraded for alerting the development team to real problems.

These problems all have to be ironed out to make the CI system a reliable and effective tool. Debugging the CI system is extremely tedious because the Groovy scripts can only be tested on the on-line Jenkins system. A modified Groovy script has to be checked into the git repository with the Jenkins scripts, and then the build chain in Jenkins has to be triggered. It is very frustrating to wait half an hour or more to run the modification, only to crash with a syntax error or get a report that it still cannot find the dependent project that you want to trigger. Hence, development on the CI system is slow and the overall stability has a lot of potential to be improved. Also introducing the dependency database to avoid the diamond dependency re-triggering has not solved the issues of Jenkins becoming unresponsive and needing to be restarted, or of Jenkins sometimes running out of memory, which seems to be a Java-related problem.

Building Branches Including Downstream Projects

All the tests described so far are all executed on the master branches of all projects. During development in feature or bug-fix branches, it is desirable to test whether the downstream projects are broken by the changes before merging them into the master. Often, changes in a core library are initiated by new features in an applications or another library. A branch in a downstream project might only compile on the corresponding branch of the upstream project. Ideally, Jenkins would check if the downstream project has a branch with the same name when triggering dependencies. Triggering branches is implemented, but not yet fully working as desired. And even a perfectly working branch build system would have the issue that it takes too long to process the whole chain, and that it has the potential to overload the Jenkins server. Branches under development are pushed frequently, and if each build triggers the CI chain, this quickly piles up and it takes hours for Jenkins to work this off, and finally give the answer whether all downstream projects are still working.

CONCLUSION

Not having binary compatibility for minor software releases enables the flexibility required to quickly and efficiently implement features needed by the applications, as well as a short software development cycle. It comes at the

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC007

price of always having to re-compile all dependencies every time a library changes.

For Debian packages, this can be solved by a naming convention that incorporates the version and build numbers in the package and library names, and packaging scripts which have a dependency database and are automatically incrementing the build number as needed.

Continuous integration with Jenkins is challenged by the large code base and diamond dependencies, such that the build-in dependency resolution cannot be used. Implementing a dependency database using custom Groovy scripts is computationally heavy and makes Jenkins unreliable. The attempt to introduce a stripped-down "fast track" process did not reach the desired speedup; it still takes as much as three hours to compile all the projects and to run their tests, yielding little benefit over a nightly build. Technically failing Jenkins jobs cause many false positive failures, which in turn cause most projects to register a "failed" status, even if more than 90% of them have no issue.

REFERENCES

[1] ChimeraTK: Control system and Hardware Interface with Mapped and Extensible Register-based device Abstraction Tool Kit, https://github.com/ChimeraTK/

[2] ChimeraTK Debian packaging scripts, https://github.com/ChimeraTK/DebianPackagingScripts

[3] The Linux Userspace I/O HOWTO, https://www.kernel.org/doc/html/v6.5/driver-api/uio-howto.html

[4] The Distributed Object Oriented Control System (DOOCS), http://doocs.desy.de/

[5] Experimental Physics and Industrial Control System (EPICS), http://www.aps.anl.gov/epics/index.php

[6] OPC Unified Architecture Specifications - Part 1: Overview and Concepts, https://reference.opcfoundation.org/v104/Core/docs/Part1/

[7] Jenkins automation server, https://www.jenkins.io/