An Open Continuous Deployment Infrastructure for a Self-driving Vehicle Ecosystem

Christian Berger(✉)

Department of Computer Science and Engineering,
University of Gothenburg, Gothenburg, Sweden
christian.berger@gu.se

Abstract. Self-driving vehicles are an ongoing research and engineering topic even though first automotive OEMs started to deploy such features to their premium vehicles. Chalmers University of Technology and University of Gothenburg are operating and maintaining a vehicle laboratory comprising 1/10 scaled cars, a Volvo XC90, and a Volvo FH truck to conduct studies with automated driving. This laboratory is used both from researchers from different disciplines and in education. The experimental software for all these platforms is powered by the same software environment for different hardware architectures. Therefore, maintaining and deploying new features and bugfixes to the users of this laboratory in a fast way needs to be organized in a reproducible yet easily maintainable manner. This paper outlines our open approach to encapsulate our build, test, and deployment process using VirtualBox, Docker, and Jenkins.

1 Introduction

Today's engineers are challenged by the development and maintenance of increasingly complex autonomous driving systems. To conduct research within this domain, our laboratory setup comprises a virtual test environment to experiment with algorithms for such vehicles, a fleet of standardized 1/10 scale miniature cars, and a workshop housing a Volvo XC90 and a Volvo FH truck. Our algorithms on these different platforms are powered by our open-source middleware OpenDaVINCI[1] and the vehicle software environment OpenDLV[2].

Having a unified and open-source software environment for all these different platforms, where we have full flexibility and design freedom, allows us to conduct our research and education in the way that is supporting the goals of the laboratory with the different vehicles in the best way. However, preserving the full flexibility in design and implementation requires also to maintain a certain level of deployment infrastructure to support the different research and educational projects with "ready-to-use" packages that work on the different development platforms.

[1] http://code.opendavinci.org.

[2] https://github.com/chalmers-revere/opendlv.

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
K. Crowston et al. (Eds.): OSS 2016, IFIP AICT 472, pp. 177–183, 2016.
DOI: 10.1007/978-3-319-39225-7_14

To underline the challenge, an embedded systems course taken by around 75–80 students every year is using our scaled cars with ARM-based hardware environments for educational purposes. Therefore, maintaining pre-compiled packages while having a low response time to react on feature and change requests as well as on unveiled issues while testing the resulting packages on the different architectures is resource-intense.

In this paper, we outline the technical setup that realizes our goal to reduce the manual maintenance and deployment work by highly automating software building, testing, packaging, and deployment using VirtualBox, Docker, and Jenkins; as such, the software architecture and algorithms for the self-driving vehicles are not in the paper's focus (cf. [Ber14] for further background). Having our infrastructure in place, we achieve (a) reproducible build and packaging environments, (b) can directly test pre-compiled binary packages by setting up Docker containers imitating a client environment using "throw-away" containers [Ber15], and (c) have the possibility to rollback and restore a previous release if something went unexpectedly wrong.

The rest of the paper is structured as follows: Sect. 2 summarizes related work and Sect. 3 describes the technical setup for the encapsulated build environment. Section 4 concludes the paper and discusses future topics to be addressed.

2 Related Work

Build farms for open-source software to support a professional deployment of software packages are available with different foci: The GCC compile farm [WWWb] provides different hardware and software platforms (Linux and BSD); however, the offered operating systems are not up to date as required for our laboratory. The Debian Linux distribution is providing a build service [WWWa] targeting package maintainers and release managers. Such deb-packages can also be automatically built by using Launchpad [WWWc] aiming at Ubuntu-based distributions; however, rpm-packages would not be supported (even though the tool `alien` could transform a deb-package into an rpm one and vice versa). In contrast, OpenSuSE's build farm [WWWd] is offering a build environment for different packages, even though it is primarily used for rpm-based platforms.

As our goal is to achieve a high-degree of automation and documentation of the entire deployment process without administering different build farms, we decided to setup and maintain our own environment. The closest concept to our demands would be Docker's own build procedure [Fra15] that we partially explored in our previous work [Ber15]. As we have to maintain the technical infrastructure for a Docker-based continuous deployment anyways, this paper is complementing our previous work addressing the packaging for different Linux distributions using automated and encapsulated builders and distribution testers.

We have released all essential scripts as open-source: https://goo.gl/XhEq15.

3 Encapsulated Build Environment

The open-source software environment OpenDaVINCI used for the vehicle laboratory is the result from the research and experience of developing several self-driving vehicular systems during the last years: Started at the 2007 DARPA Urban Challenge [RBL+09], over a self-driving experimental SUV car [Ber10], up to a research and educational 1/10 scale vehicle platform [Ber14] (Fig. 1).

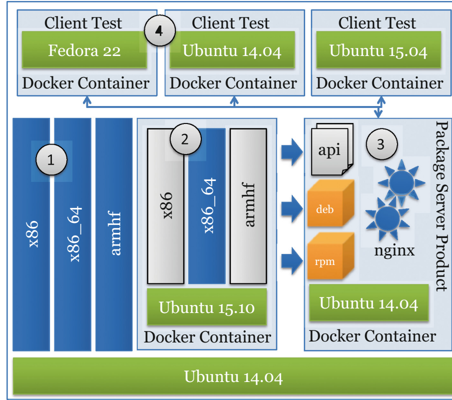

Fig. 1. Encapsulated build environment: (1) building binaries on native platform, (2) building binaries within Docker, (3) package server as Docker image, and (4) regression tests with typical client setups.

3.1 OpenDaVINCI Software Environment

The open-source software environment OpenDaVINCI is a lean and portable C++ middleware to realize distributed software components exchanging messages. The core functional properties comprise encapsulation of typical programming idioms used with distributed, data-exchanging software components like concurrency, UDP-, TCP-, and serial-communication, abstraction of shared memory and time, and publish/subscribe and round-robin coordinated data exchange.

These low-level functional features are extended by a domain-specific library providing additional functions typically required by automotive software systems to realize self-driving functionality: Methods to describe a logical road network, a visualization environment (bird's eye perspective as well as 3D rendering), and components to embody simulations (vehicle kinematics, sensor simulations for a virtual camera, infrared, and ultrasonic sensors).[3]

The different components are compiled into individual static and dynamic libraries as well as stand-alone applications. The libraries enable the transparent reuse in headless simulations as part of unit-tests[4], while the stand-alone

[3] https://goo.gl/7SGR7G.
[4] https://goo.gl/tKCEp1.

applications can be distributed to different machines and supervised by a central component `odsupercomponent` to monitor their life-cycle, to provide either uncoordinated publish/subscribe communication, or to enforce deterministic communication and scheduling following the round-robin pattern.

Our Jenkins build system is using CMake and GCC 4.8 or higher on the Linux and BSD platforms, Clang on Mac OS X, and Visual Studio 2013 on Windows.

3.2 Regression Testing

The aforementioned features are tested with a growing set of more than 570 unit tests realized with CxxTest and executed by Jenkins for the following 64 bit platform configurations chosen from the rankings from DistroWatch.com: ArchLinux, CentOS7, Debian 8.2, DragonFlyBSD 4.2 and 4.4, ElementaryFreya, Fedora 21-23, FreeBSD 10.2, MacOS X, Mageia 5, Mint 17.3, NetBSD 7.0, OpenBSD 5.8, openSuSE 13.1 and 13.2, Scientific Linux 7, Ubuntu 14.04.3 LTS, 15.04, and 15.10, Windows 8.1 and 10, and Zorin 9.1 and 10. The following 32 bit environments are tested as well: FreeBSD 10.2 (32 bit), Mint 17.1 (32 bit), Ubuntu 14.04.3 LTS (32 bit), and Windows 7 (32 bit).

Jenkins is using the aforementioned platforms on a Mac OS environment with VirtualBox while running itself in a VirtualBox virtual machine itself for simplified maintainability allowing a regular backup to rollback in the case of issues when updating the individual platforms. All platforms are accessed via SSH to unify the scripting of the build process and to report back the results from the build and the CxxTest test suites.

The regression testing is following a 12 h schedule per day; as the typical development environment is Ubuntu 14.04.3 LTS, this platform is additionally triggered by any new commit to the master branch on GitHub. Thus, developers have to wait a maximum of around twelve hours to know whether their changes run safely on all supported platforms.

3.3 Encapsulated Continuous Deployment

The project's central GitHub page provides access to the latest features and reports also the results from the Jenkins regression tests. Thus, members of the research laboratory as well as students using the source distribution can simply pull therefrom – either the current development head or a stable release.

As the complete compilation on a single core machine takes around 15 min, we also offer pre-compiled binary packages[5] for Ubuntu 14.04.4 LTS and 15.04 (i386, armhf, and amd64), Ubuntu 15.10 (amd64), and Debian 8.2 (i386, armhf, and amd64) as deb, and for CentOS 7, Fedora 21 and 22, and openSuSE 13.1 and 13.2 (i686, amd64, and armhf) as rpm.

Our deployment process is incorporating regression testing as well: All created binary packages are tested on a fresh system environment before the new release is being publicly accessible to the world.

[5] http://goo.gl/BTEHEs.

Regression Deployment. Any new release is deliberately initiated but Jenkins is triggering a "dry-run deployment" on any newly pushed change to our master branch to report whether a release would succeed or fail. This "dry-run deployment" is using the same build and test environment as the script for the real deployment process executed on Ubuntu 14.04 LTS (64 bit).

As a first step, the local working copy is updated to the respective revision. Next, the source tree is built for x86-64 systems using GCC 4.8.4. Afterwards, the 32 bit variant is compiled using the same compiler adjusted therefor. Finally, the source tree is built for ARM systems using the arm-linux-gnueabihf tool chain 4.8.2 for hard-float environments as our miniature vehicle fleet is using the Odroid XU3 platform. Once a respective build has completed, the resulting binaries and libraries are bundled into deb and rpm packages.

As the resulting binaries would not be directly usable on Ubuntu 15.10, the actual build process would need to be executed on a different software setup. For this purpose, we have encapsulated the actual build into a Docker image.[6]

The Dockerfile for this image bases on the Ubuntu 15.10 distribution and contains the required build environment and the required library dependencies for OpenDaVINCI. The final step for the Docker container is the execution of the actual build. The build itself is simply running the same steps as described for manually compiling OpenDaVINCI from sources on Ubuntu 15.10.[7] As the source folder is that part of the Docker image, which is changing the most, it is simply mapped into the running Docker container. The resulting build is executed as follows: `docker run --rm=true -v $HOME/OpenDaVINCI:/opt/OpenDaVINCI seresearch/wily:latest`.

The advantage of this approach is the textual description and full automization of the actual build process. Thereby, further build environments can be added and maintained easily. The regression testing builds as described in Sect. 3.2 including running all test suites in the VirtualBox environments take approximately 15–20 min per platform; the encapsulated Docker build takes around 17 min to complete.

Testing Pre-compiled Packages. After all packages have been successfully built, the webserver environment for delivering these packages is encapsulated in a Docker image as well. Thereby, not only the complete runtime configuration for the production server providing the packages is described and documented but it also allows for a safe rollback to previous versions if an issue occurs. Furthermore, migrating the production server to a different hardware server or even a Cloud-infrastructure is possible.

Before the newly produced production server will be enabled for world-wide access, it will be started as "production-server-under-test" for internal access on a specific port only. Thus, the produced binary packages can be tested in a user-like configuration environment to ensure that the user-workflow is running as expected.

[6] https://goo.gl/ueqTpH.
[7] http://goo.gl/yR1JDe.

Therefore, a fresh client setup for the specific Linux distribution will be created with Docker adding the "production-server-under-test" as package source. Then, the pre-compiled binaries will be installed and a small test program will be compiled and executed[8]; afterwards, the images therefor will be discarded afterwards [Ber15].

The freshly produced pre-compiled binaries are tested on the following client configurations: CentOS7 (rpm), Debian 8.2 (deb), Fedora 21 & 22 (rpm), openSuSE 13.2 (rpm), and Ubuntu 14.04 & 15.04 (deb). Thereby, the typical Linux distributions that are used by the users of the vehicle laboratory are covered. After all these tests have been successfully passed, the currently running production server is deactivated and the freshly created Docker image containing the new pre-compiled binaries will be activated.

3.4 Completing Continuous Deployment

Besides the actual pre-compiled binaries that are served by the production's server web environment, the project's website is added as well as the software's API documentation.[9] Finally, the software's tutorials are automatically generated on any new commit to our GitHub repository from the ReadTheDocs.org service.[10] In total, a complete deployment running all the different Docker-encapsulated builders and tests takes around 90 min.

4 Conclusion and Future Work

Our university's laboratory equipment comprises a fleet on miniature vehicles, a Volvo XC90 SUV, and a Volvo FH truck. To facilitate algorithm reuse and exchange as well as preserving full flexibility and design freedom in our source code, we are providing and maintaining the open-source software environment OpenDaVINCI powering these vehicles. To simplify the use of this software in research projects but especially for large-scale student project courses, we also provide pre-compiled binaries and tutorials.

In this paper, we have outlined our technical infrastructure enabling continuous deployment while preserving even regression testing of the deployment itself by encapsulating the build, test, and deployment processes into different Docker images. While initial effort needed to be spent to setup such an environment, the monthly maintenance effort like regular system updates is very low by having a high degree of automation using a precise documentation of the realized processes in the supporting scripts and build files. As part of this paper, we release the scripts for our continuous deployment infrastructure as open-source.

Future work will address the parallelization of the different builders to further reduce the deployment time. Additionally, further platforms and builders are planned to be added to extend the repository of offered pre-compiled binaries.

[8] https://goo.gl/1Wq9hf.

[9] http://api.opendavinci.org.

[10] http://docs.opendavinci.org.

Acknowledgments. The Chalmers REVERE laboratory is supported by Chalmers University of Technology, AB Volvo, Volvo Cars, and Västra Götalandsregionen.

References

[Ber10] Berger, C.: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles. Shaker Verlag, Aachener Informatik-Berichte, Software Engineering Band 6, Aachen, Germany (2010)

[Ber14] Berger, C.: From a competition for self-driving miniature cars to a standardized experimental platform: concept, models, architecture, and evaluation. J. Softw. Eng. Robot. **5**(1), 63–79 (2014)

[Ber15] Berger, C.: Testing continuous deployment with lightweight multi-platform throw-away containers. In: Großpietsch, K.-E., Kloeckner, K. (eds.) Proceedings of the 41th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Funchal, Madeira, Portugal, August 2015

[Fra15] Frazelle, J.: New Apt and Yum Repositories, July 2015. https://blog.docker.com/2015/07/new-apt-and-yum-repos/. Accessed 15 Jan 2016

[RBL+09] Rauskolb, F.W., et al.: Caroline: an autonomously driving vehicle for urban environments. In: Buehler, M., Iagnemma, K., Singh, S. (eds.) The DARPA Urban Challenge. STAR, vol. 56, pp. 441–508. Springer, Heidelberg (2009)

[WWWa] Debian Build Service. https://buildd.debian.org. Accessed 15 Jan 2016

[WWWb] GCC Compile Farm. https://gcc.gnu.org/wiki/CompileFarm. Accessed 15 Jan 2016

[WWWc] Launchpad. https://launchpad.net. Accessed 15 Jan 2016

[WWWd] OpenSuSE Build Service. http://build.opensuse.org. Accessed 15 Jan 2016