ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA154

DATA ACQUISITION SYSTEM DEPLOYMENT USING DOCKER CONTAINERS FOR THE SMuRF PROJECT

J. Vasquez, SLAC National Accelerator Laboratory, Menlo Park, USA

Abstract

The SLAC Microresonator Radio Frequency (SMuRF) system is being developed as a readout system for next generation Cosmic Microwave Background (CMB) cameras. It is based on a FPGA board where the real-time digital processing algorithms are implemented, and high-level applications running in an industrial PC. The software for this project is based on C++ and Python and it is in active development. The software follows the client-server model where the server implements the low-level communication with the FGPA while high-level applications and data processing algorithms run on the client. SMuRF systems are being deployed in several institutions and in order to facilitate the management of the software application releases, dockers containers are being used. Docker images, for both servers and clients, contain all the software packages and configurations needed for their use. The images are tested, tagged, and published in one place. They can then be deployed in all other institutions in minutes with no extra dependencies. This paper describes how the docker images are designed and build, and how continuous integration tools are used in their release cycle for this project.

THE SMURF PROJECT

The next generation of cryogenic CMB (Cosmic Micro-wave Background) cameras [1] require densely instru-mented sensor arrays. These arrays have large number of sensors, in the order of 10,000 to 100,000 per camera. The readout of this large number of sensors is a big challenge that requires substantial improvements in highly-multi-plexed readout techniques.

The SMuRF system is being developed as a readout sys-tem for this next generation CMB cameras. It aims to read 4000 microwave channels between 4 and 8 GHz, in a com-pact form factor. The system reads out changes in flux in resonators by monitoring the change in transmitted ampli-tude and frequency of RF tones produced at each resona-tor's fundamental frequency.

The SMuRF system is unique in its ability to track each tone while minimizing the total RF power required to read out each resonator, thereby significantly reducing the line-arity requirements of the system.

SLAC COMMON PLATFORM

The SMuRF system is based on the SLAC Common Platform Hardware, Firmware, and Software.

SLAC Common Platform Hardware

The SLAC Common Platform hardware is based on the ATCA (Advanced Telecommunication Computing Archi-tecture) standard.

The carrier card contains the FPGA (Xilinx KU15P Ul-trascale+) as well as all the digital, management, and power distribution devices. The analog RF devices are located on two double-wide AMC (Advanced Mezzanine Cards) daughter cards. An RTM (Read Transition Module) card contains slow speed analog devices.

The FPGA has 8x 12.5Gbps uplink and downlink JESD204b interfaces to each AMC card, SPI buses to the RTM, as well as a 10Gbps Ethernet link to the ATCA crate's backplane.

All these three boards (carrier, AMCs, and RTM) are in-stalled in one slot of an ATCA crate. The crate provides a dual-star Ethernet backplane, cooling, power distribution, and a management network based on IPMI (Intelligent Platform Management Interface).

Figure 1 shows all the component of the SLAC common platform hardware.

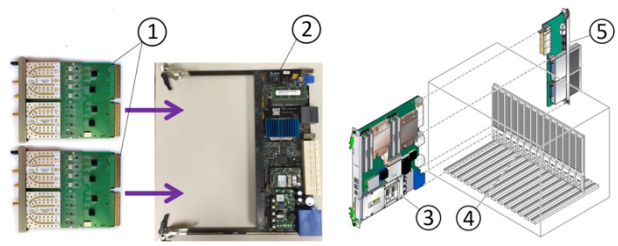

Figure 1: SLAC common platform hardware. 1) AMC daughter cards, 2) carrier card, 3) assembly of a carrier card with AMC daughter cards, 4) ATCA crate, 5) RTM card.

SLAC Common Platform Firmware

The SLAC Common Platform Firmware is a set of VHDL libraries which contain protocols, device access and commonly used modules for all applications that use the SLAC Common Platform hardware.

The SMuRF firmware application uses these set of li-braries, as well as an application specific digital signal pro-cessing module. The final application digitizes and pro-cesses up to 400 channels in a 4 GHz bandwidth.

SLAC Common Platform Software

Rogue [2], a C++ library with Python bindings, is used as a framework to write the low-level software application that communicates directly with the FPGA.

SMURF SYSTEM ARCHITECTURE

A SMuRF system is formed by an ATCA crate and an external industrial PC. The ATCA crate has one or more carrier cards in it. Each carrier card has his own FPGA, which run the SMuRF firmware application, a set of AMC daughter cards and an RTM card. The slot number 1 of the ATCA crate is reserved for an Ethernet switch card.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA154

The software applications are executed in the external industrial PC, which runs Ubuntu 18.04. The PC has a dedicated 10G Ethernet connection to the ATCA Ethernet switch. The ATCA Ethernet switch has a point-to-point connection to each carrier card in the crate though its backplane network. The PC can then access any card on any slot at the same time.

Figure 2 shows a block diagram of a SMuRF system.

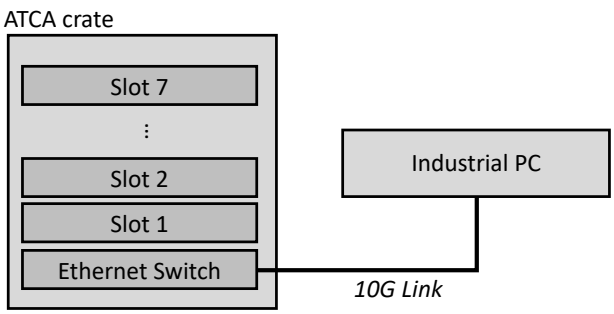

Figure 2: SMuRF system block diagram.

SOFTWARE ARCHITECTURE

The software application is based on the client-server model. The server application is based on Rogue, and it is used for handling the low-level access to the FGPA, and a first stage processing of the data. The client application is based on python, and it is used as a high-level interface to control the SMuRF system as well as to process and analyzed the data.

The server application is in charge of providing access to the firmware registers for configuration as well as monitoring of the system status. On the other hand, it also handler the reception of asynchronous streams of data originated at the FPGA application, which consist on 2240-byte packets at a 10kHz rate. When the data is received, a first processing stage happens, which includes down-sampling and filtering of the raw data.

Rogue provides an implementation of a portable EPICS [3] CA (Channel Access) [4] server. The SMuRF server application uses this EPICS server as a main communication interface with clients.

The client application, called pysmurf [5], it is a pure python application. It uses PyEpics [6] to access the server application's EPICS server. Pysmurf provides a set of high-level routines that allow uses to configure and use a SMuRF system, including data taking and analysis.

Beside the server and client, there are other applications that are used in an SMuRF system:

ATCA Monitor

This application is based on Rogue, and it is used to monitor the status of the ATCA crate, and the cards installed in it via IPMI. This application provides hardware identification information (like FPGA firmware version, card serial numbers, crate model, etc.) as well as runtime debug information (like fan speed, temperature, voltages, and current levels, etc.).

Timing Master Controller

In some deployments, a timing system is required. For these cases, an SLAC timing generator is used, which was previously developed for other applications at SLAC. It is based on the SLAC Common Platform, and consist of an ATCA carrier card, with special AMC daughter cards, and a special firmware application. The card is installed in slot 2 of the ATCA crate and distribute a timing data stream though the backplane to all the other carrier cards in the crate.

The software application used to control this system is an EPICS IOC (Input Output Controller) application developed at SLAC. This same IOC application is used to control the timing master in the SMuRF system.

Debug Tools

These tools, developed at SLAC, are used for debugging purposes. Some tools are C++ applications which communicate with the SLAC ATCA carrier's IPMI controllers (IPMC) and print out low level diagnostic information about both the hardware and firmware status.

Another example is a C++ application and a bash script wrapper used to load new firmware into the SLAC ATCA carrier's FPGA.

Finally, we also have EPICS command line client tools (caget, caput, camonitor).

DOCKER CONTAINERS

A docker container [7] is a unit of software that packages code and all its dependencies. Therefore, application running inside a container can be easily and reliably moved to different computing environments.

A docker container is a running instance of a docker image. A docker image, on the other hand, is a static file which contains all the needed components to run the application.

A docker image can be built and published in a central location. Then, the image can be pull and run into different servers, regardless of the underlying infrastructure (operating system, library versions, tools, etc.). The only requirement for the target server is to run the docker engine [8], which is available for most of the modern operative systems.

DOCKER CONTAINERS FOR THE SMURF SYSTEM

The SMuRF project is being developed by several institutions in the US. Each institution needs to run the SMuRF software applications. Moreover, the software (as well as firmware) application themselves are being actively, and rapidly, developed, so new releases are frequent. In general, there is a strong compatibility dependency between the firmware and software application versions.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA154

All these factors where the main motivation for deciding to use docker containers as a deployment method for the SMuRF project.

Each application used in the SMuRF project is deploy as a container image. The docker image contains the application code, binaries, dependencies, as well of any configuration file need by the application.

Containers running in a typical SMuRF system are showed in Fig. 3 and described below.

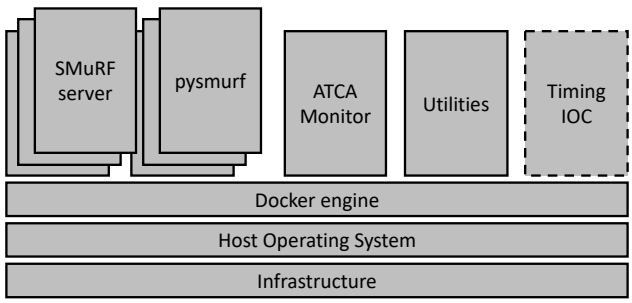

Figure 3: SMuRF application running in docker containers. The "Timing IOC" container is present only in some particular deployments.

Server Applications

The server docker image contains the server application and all its dependencies, for example, all the Rogue framework, EPICS libraries, among others.

For the case of the SMuRF server application, which is tightly dependent of the firmware application version, the docker image contains the firmware image file as well. The image contains a startup script which is run by default when the docker container is runt. This script will automatically check is the firmware version running in the target FPGA matches the image version it is expected, and if not, it will load the correct version before starting. This feature makes very easy for a user to change the application version they are running.

For each ATCA carrier in the system, there will be a server container instance.

Client Application

The client docker image contains pysmurf as well as some python tools and modules needed for it, for example, ipython to run the client in an interactive python session, matplotlib for generating data plots, among others.

For each server container instance, there will be a corresponding pysmurf client container. As the communication between the client and server is based on a network protocol (the EPICS CA protocol), the client can be run in a different PC respect to the server, as long as the network between the PCs is setup correctly.

ATCA Monitor

The ATCA monitor application is contained in a docker image, which includes the Rogue framework and its dependencies.

As there is only one ATCA crate in a SMuRF system, there is only one instance of ATCA monitor.

Debug Tools

The debugging tools are available as a separate docker image called "utilities". This container can be manually started by the user when needed, and there can be multiple instances running at the same time.

Timing Master Controller

Finally, the timing IOC runs in yet another container. In this case, as the IOC application was taken as-is from the SLAC environment (which is based on RHEL6). The docker image used for it is based on CentOS 6, with the EPICS base and modules version used at SLAC to replicate the same environment. This allowed us to use the same application without making any modification to the source code.

ADDITIONAL DOCKER ADVANTAGES

An additional advantage of using docker containers is that the initial configuration for the server PC is minimal. It only requires installing the host operating system (which for the SMuRF system is Ubuntu 18.04, although is really doesn't matter), install the docker engine, and configure the network interfaces.

On the other hand, additional features of the docker engine are used to improve the usability of the SMuRF system. For example, the docker compose tool [9] is used to start several dependent containers with a single command. For example, when a SMuRF server is started for a particular ATCA carrier card, a corresponding pysmurf container is started as well, already configured to point to the correct server application. Also, dependencies are used to make sure that the ATCA monitor as well as the Timing IOC (when used) constrainers are started as soon as the first server container is started.

Likewise, features like the overlay network model [10], which allows to create a distribute network among multiple docker engines running in different PCs, could be used to run the pysmurf client containers in a different PC respect to the server application, in a transparent way for the user.

DOCKER CONTAINERS FOR DEVELOPMENT

The concept of using docker container is very convenient for the release of stable application versions. However, for the SMuRF project it was required to have a development environment as well. Therefore, we decided to use the same docker container as development environment.

Using docker containers provides a uniform development environment, which is also consistent with the final release environment. Using the same environment for development and release, guaranty that all the testing and validation done during the development process are still valid in the final release version.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA154

The use of container for development purposes it is based on the concept of volumes. Using volumes, it is possible to mount a directory located in the host file system inside the docker container. This allows the developer to make persistent changes to the code, while still compiling it and running it inside the docker container environment.

There are mainly two types of developments done in the SMuRF project: firmware and software development.

For the case of firmware development, although the firmware image itself cannot be modified, configuration files which are required by the firmware application can. Also, it is possible to test new firmware versions with the current stable software versions. This is usually the first step when releasing a new server docker image. During firmware development, a server container which points to firmware files located in the host file system it is used.

On the other hand, for the case of software development, the server software application code lives in the host file system, and it is mounted inside the container. Then, the code can be edited, compiled and execute, from within the docker container itself. Both firmware and software development can be done at the same time as well in the same docker container.

At the end of the development process, all the changes are pushed to their respective repositories, new tagged version are released, and new docker images are generated and published. Figure 4 shows a block diagram of this development cycle.

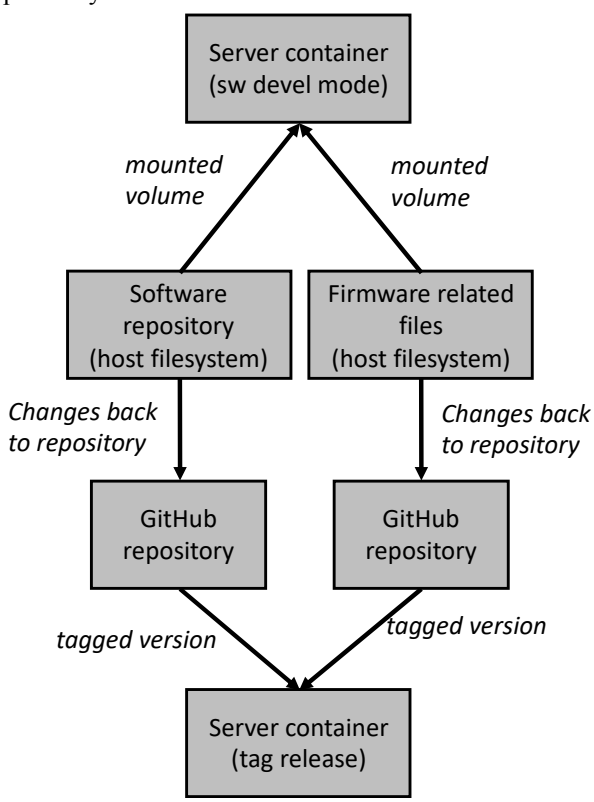

Figure 4: Software and firmware development cycle using docker containers.

DOCKER IMAGE LAYERED STRUCTURE

As dependencies are common among most of the docker images used in the SMuRF project, we are using a layered structure for building then, as described in Fig. 5.

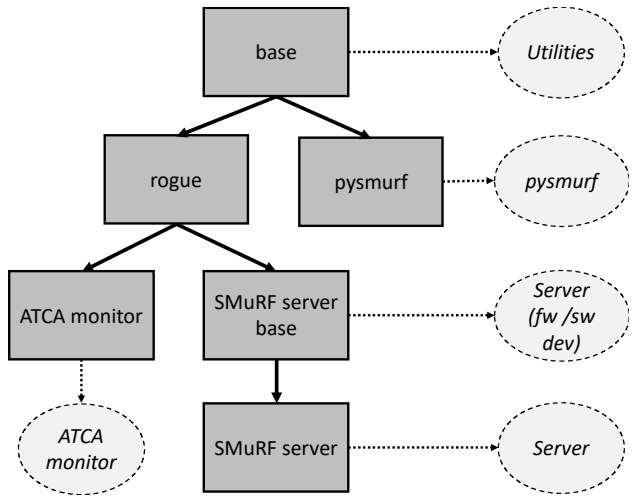

Figure 5: Docker image layered structure. Square boxes represent docker images, and the arrows show their dependencies. Circular boxes represent docker containers; each one is an instance of the image indicated by the dotted arrow.

First, a "*base"* image is built. It uses ubuntu:18.04 as its base image, and add common system packages and tools, for example, ipmi tools, python3, EPICS base, pyepics, among others. The utility container is a running instance of this image.

Secondly, a "*rogue"* image is built, using the *"base"* image as its base, and adding the Rogue framework and its dependencies. Likewise, the *"pysmurf"* image is built from the same *"base"* image adding pysmurf and its dependencies. The pysmurf client container is a running instance of this *"pysmurf"* image.

From the *"rogue"* image, docker images for the applications that use Rogue are created using it as its base image: the *"ATCA monitor"* image which includes the ATCA monitor application and its dependencies, and the *"SMuRF sever base"* image which includes the SMuRF server application and its dependencies, but excluding the firmware related files. The ATCA monitor container is a running instance of the "*ATCA monitor"* image. On the other hand, as this *"SMuRF sever base"* image does not contain firmware related files, its instance container is used for firmware and software developments.

Finally, the image *"SMuRF server"* is built using the *"SMuRF server base"* image as a base and adding the firmware related files. The server containers are running instances of this image.

For the case of the timing master application, it is based on a completely different operating system and packets. So, it is a stand-alone image which uses centos:6.10 as a

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA154

base image, adding SLAC specific packages and EPICS modules.

DOCKER IMAGE RELEASE CYCLE

The code for each application is hosted in GitHub repositories [11]. Travis [12], which is a Continuous Integration (CI) tool, it is used to automatically build docker image every time a tagged version is released.

Every time a tag is push to one of the repositories, a Travis build is triggered. When this happens, a user-defined process runs in the Travis building servers; this process is setup to build the docker image and push it to the Dockerhub public repository [13]. Once there, they can be pull to any of the SMuRF servers.

Figure 6 shows a diagram of this release cycle process. The development workstation can be the SMuRF server PC itself when access to the hardware is required for the development process. The development process follows the diagram showed in Fig 4.

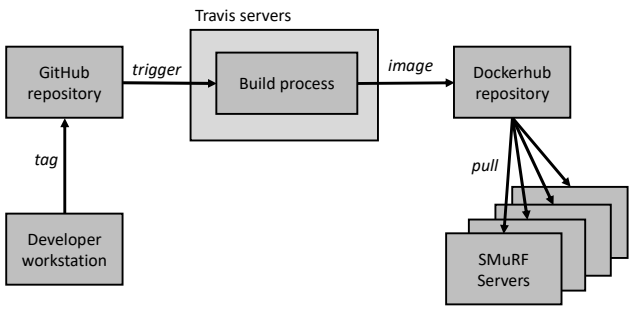

Figure 6: Docker image release cycle.

CONCLUSIONS

The SMuRF system aims to be the readout system for the next generation CMB cameras. It is a project currently being developed in several institutions across the US. Software and firmware applications are in rapid development and evolution, however they are also tools needed for other type of development, like hardware, sensors, high level applications, among others.

Docker containers are used as a deployment method for these applications. The use of container facilitates both the release of new version from part of the developers as well as the deployment of new versions from part of the user. Likewise, docker containers are also used for development purposes. They provide a uniform development environment, which matches exactly with the final release environment.

In a SMuRF system, each application runs in independent docker container. The docker engine provides tools, like the docker compose, which are used to facilitates the startup process of a complete system with dependencies between applications.

The use of container has showed a lot of benefits in a complex, dynamic, and largely distributed project as the case of the SMuRF project. Integration with modern CI tools has made the release process automatic for the developer. Also, once a docker image is available it can be easily pull to any server and run in a deterministic and relievable way by any user.

REFERENCES

[1] arXiv:1809.03689 [astro-ph.IM]

[2] Rogue, https://github.com/slaclab/rogue

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

[4] EPICS Channel Access, https://epics-controls.org/resources-and-support/documents/ca

[5] pysmuf, https://github.com/slaclab/pysmurf

[6] PyEpics, https://cars9.uchicago.edu/software/python/pyepics3

[7] Docker containers, https://www.docker.com/resources/what-container

[8] Docker engine, https://docs.docker.com/engine

[9] Docker compose, https://docs.docker.com/compose

[10] Docker overlay networks, https://docs.docker.com/network/overlay

[11] GitHub, https://github.com

[12] Travis, https://travis-ci.com

[13] Dockerhub, https://hub.docker.com