A TESTBED FOR VALIDATING THE LHC CONTROLS SYSTEM CORE BEFORE DEPLOYMENT

J. Nguyen Xuan, V. Baggiolini, CERN, Geneva, Switzerland

Abstract

Since the start-up of the LHC, it is crucial to carefully test core controls components before deploying them operationally. The Testbed of the CERN accelerator controls group was developed for this purpose. It contains different hardware (PPC, i386) running various operating systems (Linux and LynxOS) and core software components running on front-ends, communication middleware and client libraries. The Testbed first executes integration tests to verify that the components delivered by individual teams interoperate, and then system tests, which verify high-level, end-user functionality. It also verifies that different versions of components are compatible, which is vital, because not all parts of the operational LHC control system can be upgraded simultaneously. In addition, the Testbed can be used for performance and stress tests. Internally, the Testbed is driven by Atlassian Bamboo, a Continuous Integration server, which builds and deploys automatically new software versions into the Testbed environment and executes the tests continuously to prevent from software regression. Whenever a test fails, an e-mail is sent to the appropriate persons. The Testbed is part of the official Controls System development process wherein new releases of the controls system have to be validated before being deployed operationally. Integration and system tests are an important complement to the unit tests previously executed in the teams. The Testbed has already caught serious bugs that were not discovered by the unit tests of the individual components.

MOTIVATION

As described in the previous publication [1], the accelerator controls system can roughly be described as 3 tier architecture, mainly written in Java and C/C++. It is composed of many layers developed by separate teams. Along with software, hardware has also been evolving and nowadays the operational environment consists of different hardware architecture running different OS at the same time.

Upgrades are very challenging since many components and teams are involved. There is no place for failure, since a beam downtime of the LHC itself only costs about 50’000CHF/h.

The different components are well tested individually with unit tests, but without any systematic function and integration tests. This is the reason why we started the Testbed project.

THE TESTBED

Overview

The main goal of the Testbed is to test components together before they are deployed into operations and validate a working set of versions. This practice is part of an overall development process to which also the SIP initiative belongs [2].

The scope of the Testbed is to test general purpose controls components. It does not include GUIs, or devices with a specific function, such as a power converter or a beam loss monitor. The tests focus on functional aspects, mainly integration and system testing, but also include verification of backward compatibility and regression testing. Some tests validate the reaction to failures. Failure can be provoked artificially by shutting down services in the Testbed, and that without disturbing the operational environment. The Testbed is composed of several different machines, representing the variety of hardware and operating systems used in the CERN accelerator complex. The core software components of the controls system are deployed on these machines and clients are emulated, a detailed description follows in the next section.

Diving into Details

The Testbed tries to mimic the operational environment of the core components (Fig. 1).

Starting from the bottom, FECs (Front-End Computer) are needed to send simulated data and are synchronized by the timing system. Both hardware types from operations are integrated: PPC with LynxOS and i386 with Linux. On these FECs, two implementations from different generation are used: the PS’s GM (General Module) and the LHC’s FESA (Front End System Access).

The middle-tier is composed of common middleware services which are:

- the CCDB (Controls Configuration Database) containing data essential for most of the components
- a JMS (Java Messaging System) broker to pass messages
- RBAC (Role Based Access Control) security to restrict some actions to certain users
- CMW (Controls Middleware) services as the directory server which provides the different server addresses and the proxy which protects FECs from too many connections

Finally at the top, communication libraries with APIs like JAPC (Java API for Parameter Control) or RDA (Remote Device Access) commonly used by clients are tested.

This setup allows calls from client APIs with various paths and properties, as fetching a value from a device directly or through the proxy, enabling RBAC authentication, and so on.

THE TESTS

Type of Tests

As already mentioned, the scope of the Testbed is functional and system testing. Tests validate (1) the typical primary functions and interactions the controls system provides, (2) the correct reaction of the controls system to typical errors (e.g. device failures) and (3) the backward compatibility of new components.

An example of primary functionality is device access. All controls devices implement a device/property model. The most basic interactions with such a device are *set*, *get* and *subscription* on the properties. They can be triggered at various API levels, at the lower level RDA API and at the higher level JAPC API. To validate this functionality a test first reads the property value, then sets the property to a new value, and finally checks that the new value is published through the subscription mechanism.

Correct reaction to failures can be tested with a special device class that simulates typical device errors, e.g. sending wrong data or not responding at all. Tests check amongst other things that correct exceptions with the right error messages are thrown.

Backward compatibility is checked by deploying the new and the old version (e.g. of the communication middleware) into the Testbed, and checking that they interoperate correctly. In general, different version combinations that may occur are tested, e.g. different versions of FESA with different versions of the middleware components.

Test Organization

The order in which tests are executed is important to make sure we get accurate test results.

Before running the actual function tests, the Testbed runs a series of self-tests, called preconditions tests. They verify that all Testbed components are active and configured as expected. For example, they check if a device is online, if the timing works, if the directory service and the database are accessible. If any of the preconditions fails, the Testbed stops, waiting for the defective components to be fixed. Otherwise the process continues and the actual function tests are executed. Preconditions tests are important to keep the test results clean and correct; they make sure that functional tests only fail on real malfunctions in the controls system, not because of misconfigured Testbed components.

The set of tests is also run in a well-defined, bottom-up sequence. Because a higher-level test will involve all those previous components, we first make sure that the lowest components are fully operational before running tests on higher components. First the tests targeting low level components are run, such as tests on the timing system, the directory service or the proxy. If a test on the proxy fails, then tests from JAPC are likely to also fail. By testing the proxy before JAPC, we ensure that a fault in the proxy is recognized as such, and not as a JAPC fault.

Writing and Maintaining Tests

Tests should be written by the teams who provide the library or the component or by a team dedicated to testing in order to be able to keep up with the project changes or to simply maintain the tests. But in practice the teams focus more on development of new features, so the tests are often written by third persons following the specifications.

The Testbed administrator is in charge of verifying and integrating new tests into the test suite. It happens that bad written tests give wrong errors result, but they are fairly easy to spot and quickly fixable. Those errors are

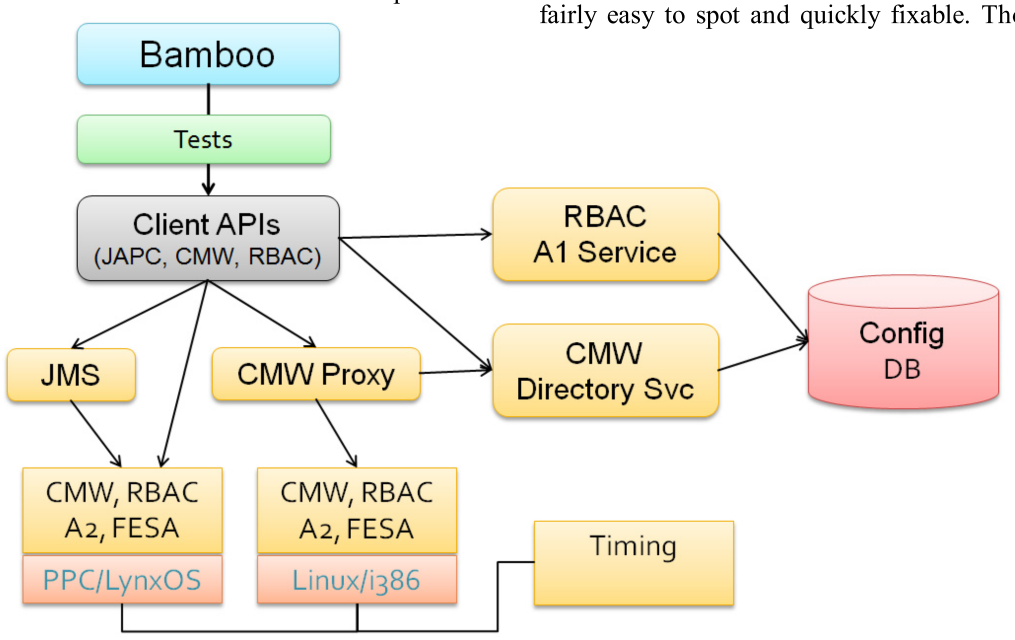

Figure 1: The Testbed structure.

habitually detected at development time.

Tests are updated when a new component with non backward compatible changes are deployed. Usually the new deployed components make the tests fail, so unless the tests are changed, they will keep failing.

Running Tests

Tests are run by our continuous integration server Atlassian Bamboo [3]. Normally, Bamboo is used as follows. A so-called Bamboo test plan monitors a source repository and triggers a build process when source code changes are committed by some developer. A build process checks out the source code, compiles the sources and runs the unit tests. If all this succeeds, a cascade of other dependent test plans can be triggered. If it fails, Bamboo sends typically an e-mail to the owner of the test plan and to the committer. Bamboo displays the results of unit tests very nicely with graphs, relevant statistics and metrics, and the detailed logs of the whole execution are available. A history of all test plan executions is kept as well. Within a test plan, several stages can be defined to divide a build into several steps. These steps are run sequentially and we use them to run the tests in a defined bottom-up sequence (Fig. 2).

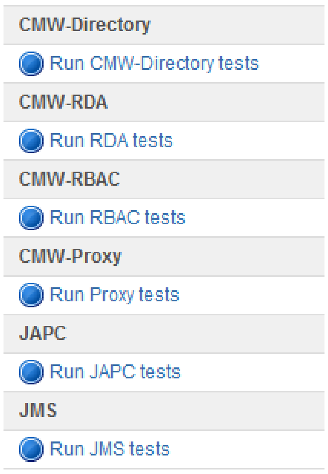
CMW-Directory
Run CMW-Directory tests
CMW-RDA
Run RDA tests
CMW-RBAC
Run RBAC tests
CMW-Proxy
Run Proxy tests
JAPC
Run JAPC tests
JMS
Run JMS tests

Figure 2: Bamboo's stages.

We use Bamboo to run the test suite of the Testbed every two hours. Since we use the JUnit framework to write our tests, full reports are also shown on Bamboo. Keeping the logs from each build is very handy because it happens that an error appears only once in a while, not at each run.

Using Bamboo to drive the Testbed has its limitations, the automatic deployment mechanism has to be scripted by hand (more about it in the next section), and setting up the environment such as JDK versions before running the tests is not obvious.

Automatic Deployment

As the Testbed is used for beta-testing, it is important that bugfixes can be deployed easily and quickly. We therefore invested in automatic deployment directly from the sources. Whenever a new component is ready, it is automatically built and deployed into the Testbed.

We use SVN branches to identify the source code that should be deployed into the Testbed. If Bamboo detects a commit to a specific branch, it automatically builds that branch and all the dependent projects as well. The resulting build artifacts are stored in a special binary repository, which eventually contains a set of components that have been built together. We added a post build mechanism to Bamboo, based on shell scripts, which deploys the executable artifacts into the Testbed. We use Apache Maven [4] for building, which works out-of-the box for Java. For C/C++ products, we have developed a Maven-compatible build system based on the Maven NAR plugin [5] that provides similar functionality as available in Java.

THE CASE OF CMW PROXY

This section explains in more details one particular component and the corresponding tests executed by the Testbed. The CMW proxy is a separate process used to shield devices on a front-end computer (FEC) from too many client requests. When several clients subscribe to a device through the proxy, it will manage all the subscriptions, but only do one subscription to the device. At first thought, the proxy seems to be a pretty simple piece of software, but in reality its functionality is rather tricky to implement, because it needs to be transparent to the clients and has many constraints due to RBAC security or FECs' implementation.

The middleware team, which is responsible for the proxy, updated a few of their C++ products in the release candidate repository, including the RDA communication library. They needed to deploy a new version of the proxy, which depends on RDA. Using the automatic deployment mechanism, the new proxy was updated and the tests worked fine during the first runs. But the next day, the tests were failing, because some proxied devices did not properly respond to requests. It turned out that the devices all worked fine, but the proxy was in a faulty state. We finally saw that this problem was caused by a newly introduced bug in RDA which did not properly close the connections. The Testbed had to run for 5 hours in order to reproduce this bug.

BENEFITS

The Testbed has been running for one year now and has already shown the following benefits.

More Confidence

Developers using the Testbed feel more confident in their product. The Testbed is an important complement to the unit tests, not a replacement. By running a series of tests after new software version, the Testbed ensures that a change does not break the core functionality of the controls system. The Testbed already caught several bugs and revealed few inconsistencies. The above example of the CMW proxy product took several days to fix this bug and therefore saved some hours of beam downtime.

The Testbed is vital in our environment where many developers belonging to several teams contribute to the controls system. If one team provides a new version of their component, everyone can see immediately and well before operational deployment whether the controls system still works.

Better Understanding

The accelerator controls system is complex and hard to understand as a whole entity, hence the Testbed helps in that direction by simulating requests done operationally from the CCC. One can think of the RBAC implementation in combination with the proxy, in specific cases the proxy is overriding the client's permission. First we thought that it was a bug, but in fact it was done on purpose to force users to use the proxy.

Saving Money

The Testbed allows to validate the controls system core before it is deployed in the real accelerator complex. The overall cost of the Testbed (hardware and man power) is small compared to the cost of LHC downtime. Without having done a scientific analysis, we estimate that cost of the downtime avoided by the Testbed outgrows the cost of the Testbed itself.

Small Laboratory

The Testbed is a down-scaled replication of the accelerator controls system, which can serve as an experimental laboratory for many purposes. It was already used for early validation of new systems in an early stage of development. For example the new logging system which involves front-ends, middleware, the proxy and the JMS broker.

Another example is the new build and release tool we are working on, based on Apache Maven. It was first used in the Testbed, before even giving it to any of our Java and C/C++ developers.

FORESEEN IMPROVEMENTS

The most important improvement is to write more and better tests. Our developers should write functional tests in the same natural manner as they already write unit tests. We also have to extend test coverage to validate not only main functionality but also more advanced and less frequently used tests. It still happens (and we cannot avoid) that some bugs are discovered during operations. In these cases we need to enforce that a test is written to expose this bug.

As a second priority, we intend to extend the scope of the Testbed in several ways.

We will add new systems to be validated. At the moment, only the lower layers of the controls system are deployed into the Testbed. We plan to add higher-level core components, such as the Software Interlock System (SIS) [6], the LSA/InCA [7] system and the high-level settings management and controls system for our accelerators.

We will provide several different Testbed configurations with different versions of hardware, operating systems, Java virtual machines, and controls components. Functionality is needed to automatically re-configure the Testbed and re-deploy the controls system on it.

Finally, we might open up the Testbed to other types of tests than mere functional tests, and include performance and scalability tests.

REFERENCES

[1] N. Stapley et al., an integration testing facility for the CERN accelerator controls system, Proceedings of ICALEPCS'09 , Kobe, Japan.

[2] K. Sigerud et al., The Software Improvement Process – Tools and Rules to Encourage Quality, Proceedings of ICALEPCS'11, Grenoble, France.

[3] Atlassian Bamboo, http://www.atlassian.com/software/bamboo

[4] Apache Maven, http://maven.apache.org

[5] J. Nguyen Xuan et al., A C/C++ build system based on maven for the LHC controls system, Proceedings of ICALEPCS'11, Grenoble, France.

[6] J. Wozniak et al., Software Interlock System, Proceedings of ICALEPCS'07, Knoxville, USA.

[7] S. Deghaye et al., Cern Proton Synchrotron Complex High-Level Controls Renovation, Proceedings of ICALEPCS'09, Kobe, Japan.