ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP02

BLISS – EXPERIMENTS CONTROL FOR ESRF EBS BEAMLINES

M. Guijarro*, A. Beteva, T. Coutinho, M. C. Dominguez, C. Guilloud, A. Homs,
J. Meyer, V. Michel, E. Papillon, M. Perez, S. Petitdemange
ESRF The European Synchrotron, 71 Avenue des Martyrs, 38000 Grenoble France

Abstract

BLISS is the new ESRF control system for running experiments, with full deployment aimed for the end of the EBS upgrade program in 2020. BLISS provides a global approach to run synchrotron experiments, thanks to hardware integration, Python sequences and an advanced scanning engine. As a Python package, BLISS can be easily embedded into any Python application and data management features enable online data analysis. In addition, BLISS ships with tools to enhance scientists user experience and can easily be integrated into TANGO based environments, with generic TANGO servers on top of BLISS controllers. BLISS configuration facility can be used as an alternative TANGO database. Delineating all aspects of the BLISS project from beamline device configuration up to the integrated user interface, this paper will present the technical choices that drove BLISS design and will describe the BLISS software architecture and technology stack in depth.

BLISS PROJECT SCOPE

The BLISS project brings a holistic approach to synchrotron beamline control. The scope of the BLISS project goes from hardware control up to the end-user interface. BLISS does not include data analysis, which is devoted to another software package at ESRF called *silx* [1].

CONFIGURATION

The BLISS configuration entity, a.k.a **Beacon**, aims to provide a complete and centralized description of the entire beamline. BLISS distinguishes between 2 kinds of configuration information: either configuration is **static**, as a stepper motor axis *steps per unit*, ie. the configuration information will not change over time once the object is configured ; or the configuration is subject to change, like a motor velocity for example. In this case, this is called a **setting** and settings are all backed up within the redis database [2].

Static Configuration

The **static** configuration consists of a centralized directory structure of text based files, which provides a simple, yet flexible mechanism to describe BLISS software initialization. The *YAML* [3] format has been chosen because of its human readability (cf. Figure 1).

BLISS is an object oriented library and its configuration follows the same model. Objects are identified in the system by a unique **name**. BLISS reserves the YAML key *name* as the entry point for an object configuration.

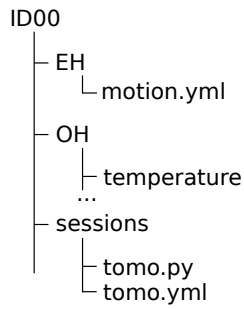

Figure 1: YAML tree example.

Each particular BLISS class may choose to profit from the BLISS configuration system. The BLISS configuration is powerful enough to describe not only control objects like motors, counter cards or detectors but also user interface objects like sessions or procedures.

The following YAML lines exemplify motor and session configurations:

```yaml
# motion.yml
class: IcePAP
host: iceid311
plugin: emotion
axes:
- name: rotY
  address: 3
  steps_per_unit: 100
  acceleration: 16.0
  velocity: 2.0

# tomo.yml
class: Session
name: tomo
config-objects: [rotY, pilatus, I, I0]
setup-file: ./tomo.py
measurement-groups:
- name: sensors
  counters: [I, I0]
```

Settings

Beacon relies on *Redis* to store **settings**, ie. configuration values that change over time, and that needs to be applied to hardware equipments at initialization time. This allows to be persistent across executions. Taking again the motor example, if a motor velocity is set to a certain amount from a BLISS session, when it is restarted the last known velocity is applied to the axis. Settings values use *Redis* structures: settings can be hashes (mapped to a Python dictionary), lists, and scalar values. BLISS offers a

* guijarro@esrf.fr

ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP02

`bliss.config.settings` helper submodule to deal with Beacon settings directly from the host Python program.

Beacon Server

A client can access the remote configuration through a service provided by the *beacon-server* which, on request, provides a complete or partial YAML configuration. The BLISS library provides a simple API for clients to retrieve the configuration from the server as a singleton **Config** object:

```
>>> from bliss.config.static import get_config
>>> config = get_config()
>>> rotY = config.get('rotY')
>>> rotY.position()
23.45
```

The *beacon-server* is also responsible of managing a *Redis* server instance and optionally a **configuration web application** (cf. Figure 2).

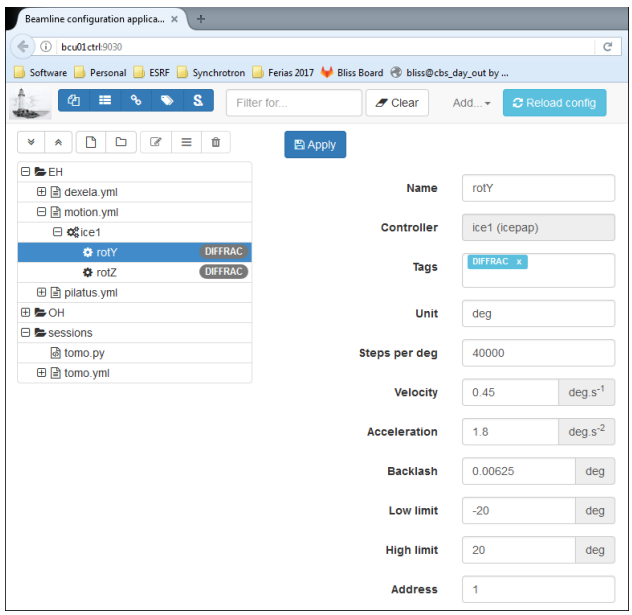

Figure 2: Beacon configuration tool.

TANGO Database

Additionally, Beacon can also provide an alternative implementation of the TANGO Database [4] service based on the same YAML configuration structure.

HARDWARE CONTROL

To the hardware control point of view, challenging points are to support an increasing number of devices to fit the experimental needs of scientists and to be able to deal with increasing complexity of devices (synchronization or communication protocols for example)

Generic Controllers

To achieve these goals, BLISS provides **generic controllers** which implement the complex, logical part of the control for each main class of devices encountered and leave to the developers the task to implement only the specific part of the control.

This approach is very efficient for instruments with a great variety of models. The price to pay is an increase of the complexity of the generic controllers. But this strategy is not exclusive: some controllers are, at least for now, not generic; either because we have no common behavior between different models or because it is much simpler to have a dedicated control. We can mention: Keithley electrometers or some ESRF cards like *OPIOM* or *MUSST*.

The first generic controllers we provide are dealing with:

- Motors Controllers
- Multichannel Analyzers (for fluorescence detectors)
- Temperature Controllers
- 2D detectors (via *Lima*)

Motors

Motor controllers are based on five fundamental classes (`Controller`, `Axis`, `Group`, `Encoder` and `Shutter`). The generic motor controller objects, and derivative devices, provide management of:

- typical basic parameters: velocity, acceleration, limits, steps per unit
- state, motion hooks, encoders reading, backlash, limits, offsets
- typical actions: homing, jog, synchronized movements of groups of motors

The minimal coding part to support a new controller consist, for the developer, in providing implementation of elementary functions like: `read_position()`, `read_veolcity()`, `set_velocity()`, `state()`, `start_one()` and `stop()`.

A **Calculation Controller** is also proposed to build virtual axes on top of real ones.

The list of motor controllers already implemented in BLISS, in use at ESRF, includes (but is not limited to) controllers like Aerotech, FlexDC, Galil, IcePap, Newfocus, PI piezo or Piezomotor PMD206.

Multichannel Analyzer Controllers

The principle is very similar for MCA electronics. An interesting detail of the implementation is the usage or zeroRPC [5], to deport control from a windows computer to the workstation where BLISS is running. This behavior allows to cohere with the **direct hardware control** principle.

First targeted MCA are XIA devices: Xmap, Mercury and FalconX. They will be followed by Maya2000 from OceanOptics and Hamamatsu.

ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP02

Simulators

For each type of generic controller, we have built "simulation devices" to test our own code and to provide test devices to help users with the creation of their control sequences.

A simulator like the **mockup motor controller** is used to test the logical part of the motor controller within the frame of a collection of unit tests executed in a continuous-integration process.

SCANNING

BLISS implements a general scanning engine to run all kinds of scans, that emancipates from the dichotomy of step-by-step or continuous scans. Indeed, BLISS introduces the concepts of **acquisition chain**, **acquisition master** and **acquisition device** to be able to perform any kind of scan.

Acquisition Chain

The representation of the acquisition chain is a **tree**. The hierarchical nature of the acquisition chain allows to formalize the dependencies between nodes. There are 2 kinds of nodes:

- master nodes, that trigger data acquisition
- device nodes (leaves), that acquire data

Acquisition chain objects expose 3 methods corresponding to the 3 phases of a scan:

- `prepare()`
- `start()`
- `stop()`

During the preparation phase, the acquisition chain tree is traversed in **reversed level order** (reversed Breadth-first search [6]) in order to prepare the device nodes first, then masters and so on until the tree root ; on each element, `.prepare()` is called. Preparation is decoupled from start in order to make sure minimum latency will happen when starting the scan. Indeed, during preparation each equipment is programmed or configured for the scan. By default, preparation of all equipments is done in parallel.

At start, the same tree traversal procedure is applied as within the preparation phase ; on each chain element, `.start()` is called ; device nodes will begin to wait for a trigger whereas master nodes will start to produce trigger events. It is important to note that devices are **always** started before masters, and that trigger events can be hardware or software. A scan can be seen as an iterative sequence ; after `.start()` method is executed, the acquisition chain enters the first iteration.

It continues until the first master signals acquisition is finished, or in case of error, or if the scan is interrupted. Then, `.stop()` methods are called on each tree element.

Monitoring scan example The following chain describes a scan with one timer master, triggering 3 diode counters (cf. Figure 3).

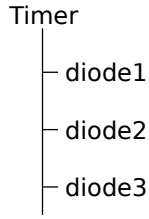

Figure 3: Monitoring scan chain.

Basic scan example This chain describes a scan with one `motor` master, triggering a `timer` master, that triggers in turn 3 diode counters (cf. Figure 4). This is typically the kind of scan *Spec* does with the `ascan` or `dscan` macros, except that in the case of BLISS the step-by-step or continuous nature of the scan does not depend on the acquisition chain, but on the type of master and device nodes.

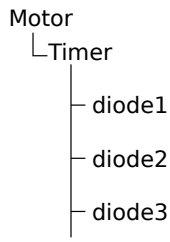

Figure 4: Basic scan example.

Associating basic scan and monitoring This chain describes a scan with 2 top masters: one `motor` and one `timer` (cf. Figure 5). The branch with the motor is like the basic scan above, except that a 2D detector is taking images at predefined motor positions, and for each image it acquires X and Y beam position. On the second branch, there is a simple temperature monitoring. The two top masters run in parallel.

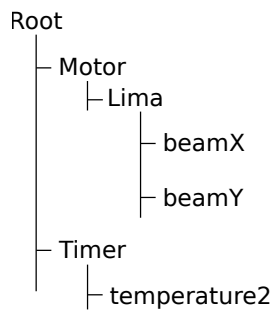

Figure 5: Associating basic scan and monitoring.

DATA MANAGEMENT

BLISS has built-in data management facilities as a first-class citizen. Each node object in the chain has a name, which clearly identifies data sources. Associated with the tree view of the acquisition chain, BLISS creates a data model from the bottom-up that closely follows experiments.

ISBN: 978-3-95450-200-4 doi:10.18429/JACoW-PCaPAC2018-WEP02

Acquisition Channels

Acquisition chain objects, being masters or devices, define zero or more `AcquisitionChannel` objects which have:

- a name
- a type
- a shape

Acquisition channels describe the kind of data produced by the underlying BLISS control objects.

Data Writing and Publishing

During scans, data is placed in the appropriate acquisition channels; then, the scanning engine temporarily publishes the channels to the *Redis* database, either as plain values for scalars or as references to data files for bigger data. Concurrently, channel data is written by the active data writer object. By default, BLISS saves data in HDF5 format.

Any external process can monitor *Redis* to get notified of on-going acquisitions.and to explore acquired data. This facilitates online data analysis. Scan data is kept in *redis* for a configurable amount of time (set to one day by default).

BLISS provides a Python helper module to iterate over produced data.

BLISS USER INTERFACES

On top of the BLISS library, two user interfaces have been developed in order to provide an entry point for users on beamlines to get access to BLISS functionalities.

Bliss Command Line Interface (CLI)

The `bliss` command line interface is based on *ptpython* [7]. It provides a Python interpreter enhanced with BLISS-specific features. Most notably, the interpreter input loop is replaced to include `gevent` [8] events processing.

`bliss` can load BLISS sessions, via a `-s` command line switch. The command line interface automatically loads session objects, and executes an optional setup script. All globals are exported to the `bliss.setup_globals` namespace, in order to allow users to import session objects in their own scripts.

BLISS Shell Web Application

BLISS ships with an 'experimental' version of a web-based command line interface similar to `bliss` (see above), offering more graphical display possibilities thanks to the web platform (cf. Figure 6).

CONCLUSION

This document presented the context for the launch of the BLISS project, and went through a technical review of all aspects of the development currently conducted at ESRF to renew the beamline experiments control system in the perspective of the EBS [9].

At the moment, BLISS is in an active development phase. Middle term goals include the development of new hardware

Figure 6: BLISS web shell.

controllers, the port of *Spec*-based experiment protocols to BLISS with the collaboration of ESRF scientists, and the improvement of BLISS user interfaces to provide data visualization capabilities using the ESRF *silx* toolkit.

BLISS has already been deployed on Macromolecular Crystallography beamlines, and more ESRF beamlines will benefit from BLISS before the end of the year: Materials Chemistry and Engineering (ID15A), High-Energy Materials Processing (ID31), Materials Science (ID11). BLISS takes up the challenge of deploying a complete new system while the former one is still in production and while beamlines stay in user operation. The BLISS project main objective is to have all ESRF beamlines equiped with BLISS in 2020.

BLISS opens new perspectives in term of beamline experiments control, to bring advanced scanning techniques and enhanced data management to all ESRF beamlines.

REFERENCES

[1] silx, `http://silx.org`

[2] redis, `http://redis.io`

[3] YAML, `http://yaml.org`

[4] TANGO, `http://tango-controls.org`

[5] ZeroRPC, `http://zerorpc.io`

[6] Edward F. Moore,"The shortest path through a maze", in *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press. pp. 285–292. As cited by Cormen, Leiserson, Rivest, and Stein, (1959).

[7] ptpython, `https://github.com/jonathanslenders/ptpython`

[8] gevent, `http://gevent.org`

[9] J. M. Chaize *et al.*, "The ESRF Extremely Brilliant Source - A 4th Generation light source", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017. `doi:10.18429/JACoW-ICALEPCS2017-FRAPL07`