ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-WEBPL05

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, Grenoble, France 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.

RATIONALE

Over the last 26 years, *Spec* [1] has been the main experiments control system at ESRF.

Spec is a software package for instrument control and data acquisition featuring a command line interface (CLI) with a read-eval-print-loop (REPL). Users can immediately call commands and more complicated sequences, written in the *Spec* macro language inspired by *awk* [2]. *Spec* has built-in step-by-step scans support, and features a long list of natively supported devices, from motor controllers to detectors. *Spec* can also communicate with beamline control systems like EPICS or TANGO, to extend the range of supported hardware. Last but not least, *Spec* features a client/server mode, to be able to control a *Spec* session from a remote process.

Spec success within the Beamline Control Unit (BCU) at ESRF, and among the synchrotron users community in general, is a vibrant example of a well-crafted piece of software, which has a limited, yet sufficient, set of features that satisfy users in their day-to-day activity, while offering enough flexibility for more advanced use cases. There is certainly a lesson to learn for any new software project.

However, at some point a limit was reached and tons of workarounds to circumvent *Spec* limitations have been implemented to be able to do continuous scans and to support multiple, fast detectors data acquisition, or to deal with *Spec* single-task execution model. The lack of extensibility of the *Spec* macros language combined with other limitations like the absence of debugging tools lead to the implementation of time-consuming, hard to maintain solutions.

Nowadays, with the perspective of the Extremely Brilliant Source (EBS) [3] program, new beamlines and new kinds of experiments require cutting-edge tools to support the more complex data acquisition protocols. This is the ambition for the BLISS project, that was started in December, 2015. Other drivers for the BLISS project include the *PaNdata* [4] initiative, to add metadata about all data produced at ESRF, that would benefit from more advanced data management from the experiment control software. Finally, beamline control can greatly benefit from latest advances in IT industry and one of the main goals of BLISS is to make the newest technology available for synchrotron experiments.

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.* [5]

TECHNICAL CHOICES

BLISS is a software package composed of a Python library and a set of tools.

Python Library

Python is a de facto standard in the scientific community, and is very popular with a huge ecosystem. Python is a multi-paradigm, dynamic language, with a clear syntax. Python ships with an extensive standard library, and features advanced debugging and profiling capabilities. The interpreted nature of Python makes it very well suited as a programming language for running beamline experiments scripts in comparison with compiled languages. Last but not least, another essential asset of Python in the context of BLISS is that it can easily interface C or C++ libraries, which makes it really unavoidable as a glue language between low-level hardware control and BLISS library.

The idea of having BLISS as a Python extension package at the first place allows to embed BLISS into existing Python applications, and is also a very flexible way of writing tools around core functionalities for the BLISS project. A similar

[*] guijarro@esrf.fr

approach was taken by the ESRF Lima project [6] for area detectors control, and proved to be very successful.

BLISS Architecture

BLISS library has a modular architecture, composed of 5 distinct entities (cf. figure 1):

- configuration
- communication helpers
- hardware control
- scanning
- data management

The Configuration entity, called **Beacon** (Beamline Configuration), traverses the entire project. It constitutes the corner stone of BLISS, as it provides entry points for all others BLISS components. In particular, Beacon is also responsible for settings management and data channels (cf. Configuration).

Communication helpers provide an uniform way to access equipments interfaced via serial line, gpib, modbus or tcp/udp. The communication entity can also give access to *TANGO* [7] device proxies thanks to the *PyTango* [8] Python module.

Hardware controllers are effectively interacting with the experiment devices (cf. Hardware Control).

The **scanning** entity is responsible for executing scans, using hardware control layer devices (cf. Scanning).

The **data management** entity publishes and stores scan data (cf. Data Management).

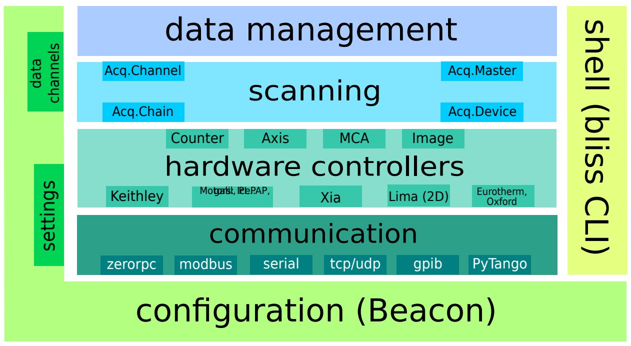


Figure 1: Bliss architecture.

Cooperative Multitasking

Multitasking is the ability of executing multiple tasks concurrently. In the case of one CPU, the way to run multiple tasks is to rapidly switch between them. In the case of multiple CPUs, parallel processing can happen ; however, if the number of tasks exceeds the number of CPU, then each CPU also multitasks. All tasks execute by alternating between CPU processing and I/O handling. For I/O, **tasks might wait** - behind the scenes, the underlying system will carry out the I/O operation and wake the task when it is finished (cf. figure 2).

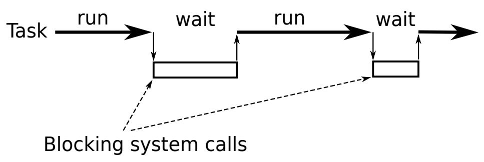


Figure 2: Task execution.

There are different models for concurrency:

- multithreading (MT), OS threads, common memory space
- actor model [9], multiple processes, messaging
- asynchronous I/O, event loops and callback chains

Data acquisition is an I/O bound activity: in many cases, a data acquisition process is waiting for an I/O operation to complete. Multiple CPUs does not really help with I/O, since the resources are exclusive: parallel processing would not happen in anyways. Moreover, it comes with a whole set of issues that makes writing MT programs utterly difficult. As stated by P. Hintjens in the *ZeroMQ guide* [10]:

> [...] one lesson we've learned from 30+ years of concurrent programming, it is: just don't share state. It's like two drunkards trying to share a beer. It doesn't matter if they're good buddies. Sooner or later, they're going to get into a fight. And the more drunkards you add to the table, the more they fight each other over the beer. The tragic majority of MT applications look like drunken bar fights.

The second model can be implemented in Python thanks to the *multiprocessing* module. Independent copies of the Python interpreter can get to communicate through message passing (IPC) using pipes, FIFOs, memory mapped regions or sockets. Messaging implies serialization: Python objects need to be converted to a byte stream, this can be done through the standard *pickle* module. However, it is quite heavy in term of memory consumption does not scale well.

Therefore, BLISS implements the third concurrency model. Asynchronous I/O is one of the main function of operating systems: it permits processing to continue before an I/O operation has completed. Basically this is the *select* system call (and more efficient variants) in Unix or Windows systems. An event loop runs within the main process, that get notified of the state of I/O operations (can do a non-blocking read, can do a non-blocking write). This usually gives great response times, low latency and CPU usage, and brings all the advantages of one single main thread.

Gevent BLISS is built on top of *gevent* [11], a coroutine-based Python networking library. It features a fast event loop based on *libev,* [12] and introduces lightweight execution units (tasks) based on *greenlet.* [13] *greenlet* encapsulates the concept of micro-thread with no implicit scheduling, which is the definition of a coroutine [14]. A coroutine is like a normal subroutine, except that it has **yielding points** instead of a single return exit. Scheduling comes from the event loop of *gevent*: while I/O operations occur in a task, *gevent* yields automatically to execute another task.

Direct Hardware Control

BLISS is based on the idea of direct hardware control, inspired by *Spec*. Whenever it is possible, controlling hardware directly is beneficial:

- easier debugging
- efficient data transfers
- no external control process

Several BLISS instances will establish their own connections in order to control beamline equipments. Of course, most of the time concurrent access is not permitted: only one task at a time can access a beamline hardware resource. As a consequence, BLISS implements a protocol in order to deal with objects resources sharing (cf. Distributed control objects).

When direct hardware control is not possible, because the equipment does not support multiple connections or if the equipment has to be controlled from another computer, BLISS uses a **proxy** to the control logic deported on the equipment side, for example with a *TANGO* [7] device server.

Distributed Control Objects

Control objects defined in several BLISS instances are supposed to be identical objects (from the same class). Two similar objects controlling the same hardware are called *peers*. BLISS introduces the concept of distributed peer-to-peer control:

- each peer has direct hardware control
- **lazy initialization**: initialization of a control object does not imply communication with the hardware
- when accessing hardware, a peer **asks for permission**
- hardware initialization is done only once, by the first peer
- when a peer state changes, for example the position of an *Axis* object, other peers are informed and **updated** accordingly via *Data Channels*

The transfer of control between peers is based on a **protocol**: peers are all connected to the **Distributed Lock Manager** service of Beacon (cf. figure 3).

Redis

BLISS uses *redis* [15] to implement essential features of the project:

- settings cache (cf. Configuration)
- data channels communication broker
- ephemeral data store (cf. Data management)

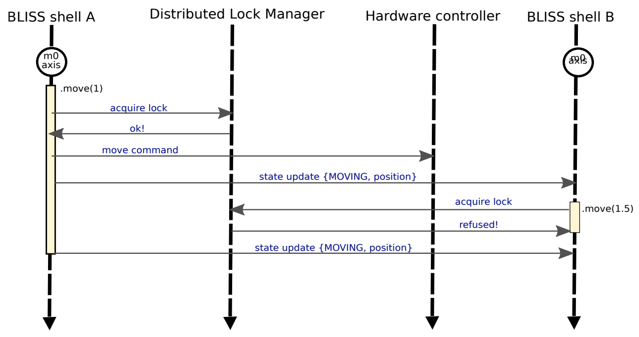


Figure 3: Distributed control objects example.

BLISS data channels relies on the **PUB/SUB** facility of redis to support message passing between BLISS sessions.

The redis server is started, and monitored, by the configuration layer of BLISS.

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.

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* [16] format has been chosen because of its human readability (cf. figure 4).

```
ID00
 ├ EH
 │  └motion.yml
 ├ OH
 │  ├temperature
 │  ...
 ├ sessions
 │  ├tomo.py
 │  └tomo.yml
```

Figure 4: YAML tree example.

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.

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:

```
# 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

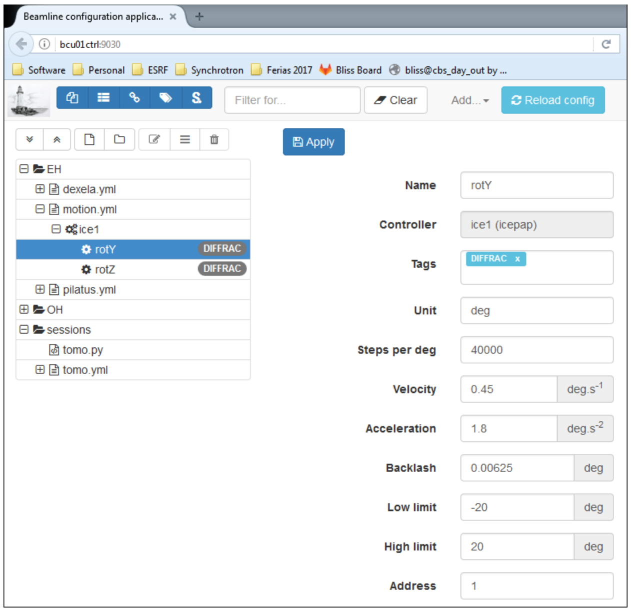
Beacon relies on *Redis* to store **settings**, ie. configuration values that change over time, and that needs to be applied to hardware equipments at initialization time. This allows to be persistent across executions. Taking again the motor example, if a motor velocity is set to a certain amount from 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 `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 5).



Name	rotY	
Controller	ice1 (icepap)	
Tags	DIFFRAC x	
Unit	deg	
Steps per deg	40000	
Velocity	0.45	deg.s^{-1}
Acceleration	1.8	deg.s^{-2}
Backlash	0.00625	deg
Low limit	-20	deg
High limit	20	deg
Address	1	

Figure 5: Beacon configuration tool.

TANGO Database

Additionally, Beacon can also provide an alternative implementation of the TANGO Database 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 [17], 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.

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 [18]) 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 (cf. figure 6) describes a scan with one timer master, triggering 3 diode counters.

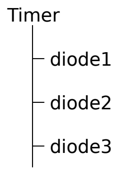


Figure 6: Monitoring scan chain.

Basic Scan Example This chain (cf. figure 7) describes a scan with one `motor` master, triggering a `timer` master, that triggers in turn 3 diode counters. 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.

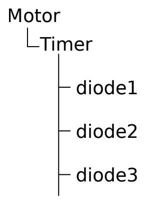


Figure 7: Basic scan example.

Associating Basic Scan and Monitoring This chain (cf. figure 8) describes a scan with 2 top masters: one `motor` and one `timer`. 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.

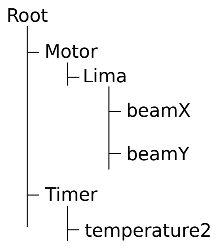


Figure 8: Associating basic scan and monitoring.

Base Acquisition Chain Classes

`AcquisitionMaster` is the base class for master nodes in an acquisition chain. `AcquisitionDevice` is the base class for device nodes in an acquisition chain. Both objects wrap existing BLISS control objects, in order to be included in a scan. There are as many `AcquisitionMaster` and `AcquisitionDevice` classes as BLISS control objects and ways to do the scan sequence. For example, `LinearStepTriggerMaster` can take 2 BLISS `Axis` objects, to produce triggers at linear motor positions ; `MeshStepTriggerMaster` can take **the same** 2 objects to produce triggers to form a grid.

Similarly to `AcquisitionChain` objects, `AcquisitionMaster` and `AcquisitionDevice` also have `.prepare()`, `.start()` and `.stop()` methods. In addition, the `.trigger()` method is called from master nodes to children whenever a trigger event is generated from hardware or software. The trigger event starts the reading on device nodes. As soon as data is read, it is pushed to the data writing module.

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.

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 temporarly 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.

TRANSITION FROM SPEC TO BLISS

The migration to full BLISS-controlled experiments, will face the refactoring of a huge quantity of existing procedures developed on ESRF beamlines in *Spec*.

In order to ensure a smooth transition from *Spec* to BLISS, some tools are provided with BLISS to be able to interact more easily with our actual control system:

- A generic counter to use any *TANGO* attribute as a BLISS counter
- **BlissAxisManager** *TANGO* server, that makes BLISS `Axis` objects available as *TANGO* devices
- A *Spec* ‘macro motor’ on top of BlissAxisManager to be able to configure *Spec* motors with BLISS (via *TANGO*)
- A generic *TANGO* server that gives access to any BLISS object

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.

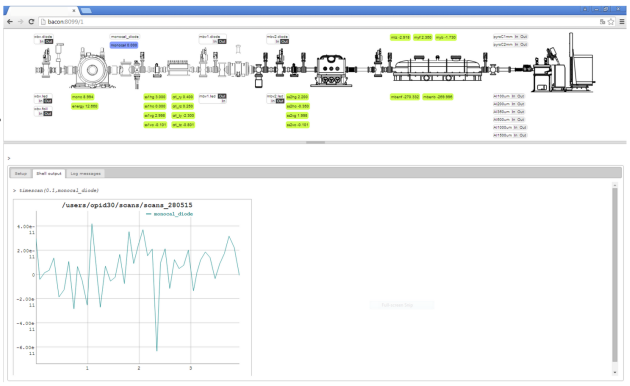
BLISS Command Line Interface (CLI)

The `bliss` command line interface is based on *ptpython* [19]. It provides a Python interpreter enhanced with BLISS-specific features. Most notably, the interpreter input loop is replaced to include `gevent` 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` (cf. figure 9), offering more graphical display possibilities thanks to the web platform.



```
> timescan(0.1,monocal_diode)
```

Figure 9: BLISS web shell.

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.

At the moment, BLISS is in an active development phase. Middle term goals include the development of new hardware 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] Certified Scientific Software, http://certif.com

[2] awk, http://www.linuxcertif.com/man/1/awk/

[3] J.M. Chaize *et al.*, "The ESRF's Extremely Brilliant Source - a 4th Generation Light Source", ICALEPCS 2017, Barcelona, Spain, 2017, FRAPL07, this conference.

[4] PANdata, http://pan-data.eu/

[5] silx-kit, http://www.silx.org

[6] A. Homs *et al.*, "LIMA: A Generic Library for High Throughput Image Acquisition", in *Proc. of ICALEPS 2011*, Grenoble, France, WEMAU011, 676-679, (2011)

[7] TANGO, http://www.tango-controls.org

[8] pytango, https://github.com/tango-controls/pytango

[9] dtic, https://www.dtic.mil/dtic/tr/fulltext/u2/a260196.pdf

[10] zguide, http://zguide.zeromq.org/page:all

[11] gevent, http://www.gevent.org

[12] libev, http://libev.schmorp.de

[13] greenlet, https://greenlet.readthedocs.io

[14] ACM Digital Library, https://dl.acm.org/citation.cfm?id=366704

[15] redis, http://redis.io

[16] yaml, http://yaml.org

[17] zerorpc, http://www.zerorpc.io/

[18] Moore, Edward F., "The shortest path through a maze", *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)

[19] ptpython, https://github.com/jonathanslenders/ptpython