BEAMLINE EXPERIMENTS AT ESRF WITH BLISS

M. Guijarro*, G. Berruyer, A. Beteva, L. Claustre, T. Coutinho, S. Debionne, M.C. Dominguez, P. Guillou, C. Guilloud, A. Homs, R. Homs, M.C. Lagier, A. Mauro, J. Meyer, V. Michel, C. Muzelle, S. Olhsson, P. Pancino, E. Papillon, M. Perez, S. Petitdemange, L. Pithan, F. Sever, V. Valls, H. Witsch
ESRF, The European Synchrotron, Grenoble, France

Abstract

BLISS is the new ESRF beamline experiments sequencer. BLISS is a Python library and a set of tools to empower scientists with the ability to write and to execute complex data acquisition sequences. Complementary with *TANGO*, the ESRF control system, and *silx*, the ESRF data visualization toolkit, BLISS ensure a smooth user experience from beamline configuration to online visualization. After a 4-year development period, the initial deployment phase is taking place today on half of ESRF beamlines, concomitantly with the ESRF Extremely Brilliant Source upgrade program. This document presents the BLISS project in large, focusing on feature highlights and technical information as well as more general software development considerations.

THE BLISS PROJECT

BLISS stands for BeamLine Instrumentation Support Software. The BLISS project started in December, 2015 inside the Beamline Control Unit (BCU, Software Group), and comes within the scope of the ESRF Extremely Brilliant Source upgrade program (ESRF-EBS) [1].

The ESRF-EBS is a global project, to put ESRF and all partners countries at the forefront of X-ray science and instrumentation. A major milestone will be reached in 2020 with the end of the construction of a new, revolutionary storage ring and the restart of ESRF user program. ESRF will then become the world's first high-energy, fourth-generation synchrotron light source. This exciting feat will offer unprecedented tools for the exploration of matter and for the understanding of life at the macromolecular level.

In particular, 4 new beamlines are being built and new instrumentation is currently under development ; the main objective of BLISS is to provide scientists with the more advanced experiments control and data acqusition software in order to take advantage of the new ESRF-EBS tools and equipments. BLISS has the ambition to fulfill the needs of the more demanding experiments.

Project Goals

- to empower scientists with the ability to write and to execute complex data acquisition sequences
- to offer an easy to use Command Line Interface (CLI) and an online data visualization application
- to provide generic building blocks to implement any kind of scan
- to get the most out of the capabilities of beamline hardware
- to provide frameworks to quicken integration of new hardware
- to facilitate online data analysis
- to enable data management

BLISS SOFTWARE

BLISS is primarily a Python 3.7 library. It is furnished with a set of tools to manage experimental setups using configured beamline devices, to write experiments control sequences and to execute them in an adapted environment, and to do online data visualization.

BLISS is an open-source software, licensed under *LGPL* [2]. BLISS is free to use by anyone ; however ESRF does not officially provide support to external users without a formal collaboration agreement.

BLISS Package

BLISS is packaged with [3]. Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Although it was initially targetting Python software, it can be used to package any kind of software and offers:

- separated environments, different execution contexts
- OS-independent, community-packaged software (from the libc level)
- the ability to create custom packages easily

BLISS Releases

BLISS follows a one-month release cycle. The first BLISS stable version 1.0.0 is to be released at the beginning of next year (January, 2020). Starting with the first stable release, BLISS version numbers will follow *semantic versioning* [4]. Basically the first number (1) represents a major version, that can only increment in case of incompatible API changes. The second number (0) increases when backward-compatible changes are added. The last number (0) is a patch-level: this is mainly to indicate bug fix releases.

Source Code Management

As of today, all BLISS code is contained within a single *git repository* [5]. This simple approach helps to ensure coherency over the code base, in particular in case of refactoring or modifications with non-trivial consequences on existing parts of the project compared to a solution based on multiple git repositories linked with sub-modules for example.

* guijarro@esrf.fr

Figure 1: From left to right: B.Formet, M.Guijarro, P.Pancino, L.Pithan, P.Guillou, S.Petitdemange, C.Guilloud.

The BLISS project development website is hosted in the *ESRF gitlab server* [6]. List of issues, ongoing discussions, link to the documentation pages and more are all publicly available.

Contributing to BLISS

Contributions to BLISS are welcome. The project follows the *github flow* [7] workflow for contributions. It is a set of best practices that combines feature-driven development and feature branches with issue tracking.

BLISS Development Team

Within the ESRF Beamline Control Unit (BCU), a team of 7 engineers (Fig. 1) is more specifically dedicated to BLISS development.

Mission The mission of the BLISS team members is to write BLISS core features, to write documentation, to be responsible for BLISS quality assurance, and to share the knowledge across Beamline Control Unit members.

Development Methodology The BLISS project development methodology is inspired by *Kanban* [8].

Communication In order to improve the communication between team members, a daily stand-up meeting is organised to tell about ongoing tasks, to deliver information about the project and to exchange about problems.

Pair-programming is encouraged whenever it is possible, in order to produce better quality code and to ensure at least 2 team members share the knowledge on some particular parts of the project.

Merge requests cannot be merged by the same person that is producing the code: a systematic code review pass has been introduced in order to produce better quality code and to enhance communication.

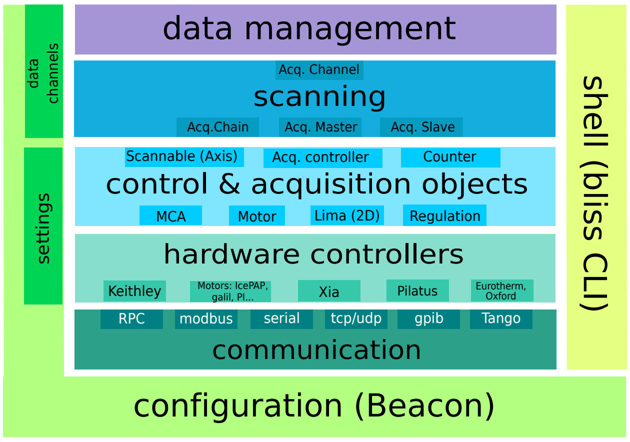


Figure 2: Bliss architecture.

BLISS ARCHITECTURE

The BLISS package has a modular architecture (Fig. 2) composed of 7 distinct families of components:

- configuration and settings (*Beacon*)
- communication helpers
- hardware controllers
- control and acquisition objects
- scanning
- data management
- command line interface (BLISS shell)

The Configuration entity, called **Beacon** (Beamline Configuration) is the corner stone of BLISS, as it provides entry points for all others BLISS components (cf. Configuration and Beacon server).

Communication helpers provide a uniform way to access equipments interfaced via serial line, gpib, modbus or tcp/udp for direct communication. BLISS also fully supports *TANGO* [9] devices (cf. BLISS Asynchronous I/O model).

Hardware controllers are effectively interacting with the beamline hardware devices. For example, this is the place to implement Python objects that would implement commands to "talk" to a device following the protocol defined by the manufacturer, using the communication objects provided by the communication helpers layer, or through TANGO calls.

BLISS provides generic **control & acquisition objects** (cf. Control and Acquisition objects), as an adapter layer to integrate arbitrary hardware and to foster code reuse thanks to the factorization of common code.

The **scanning** entity is responsible for executing scans, using acquisition controllers, counters and scannable objects from the previous layer (cf. Scanning).

Finally, the **data management** entity publishes and stores scan data (cf. Data Management).

BLISS CONFIGURATION AND BEACON SERVER

Beacon (Beamline Configuration) is a server process that is essential to BLISS, since it provides services for many BLISS features (see Fig. 3):

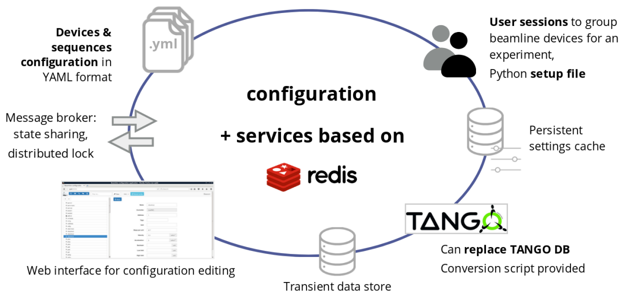

Devices & sequences configuration in YAML format
.yml
User sessions to group beamline devices for an experiment, Python setup file
configuration
+ services based on
redis
Message broker: state sharing, distributed lock
Persistent settings cache
TANGO
Can replace TANGO DB
Conversion script provided
Web interface for configuration editing
Transient data store

Figure 3: Beacon services.

- static configuration database
- configuration editor web application
- settings, i.e. runtime configuration properties
- channels, to exchange data between different BLISS objects instantiated in different processes
- publishing of data produced during scans, for online data analysis or visualisation
- distributed lock management, to be able to lock and release resources for a particular Beacon client

Beacon relies on *Redis* [10] for most of those services. Redis is an open source, in-memory data structure store, used as a database, cache and message broker.

Last but not least, Beacon embeds an optional *TANGO* [9] database-compatible server. It allows to store TANGO configuration information in Beacon YAML files, thus keeping the configuration for a whole beamline within the same files, with the same format, at the same location.

Static Configuration Database

The **static** configuration is a centralized directory structure of text files (see Fig. 4) in the *YAML* [11] format, which provides a simple, yet flexible mechanism to describe devices within a BLISS system.

YAML has been chosen because the standard native types (list, dictionary, numbers, unicode strings…) can be easily mapped to Python equivalents, it handles comments (contrary to JSON), and also because it is human-readable (contrary to XML).

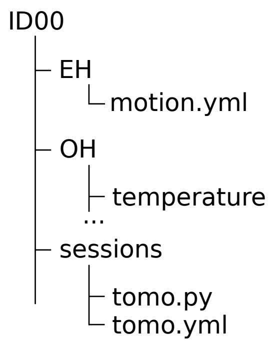


Figure 4: YAML tree example.

Objects are identified in the system by an unique **name**. BLISS reserves the YAML key *name* as the entry point for an object configuration. When loading static configuration data, Beacon goes all over the configuration database directories, and builds an internal representation of the objects that are defined in YAML mapping nodes from the files. Ultimately this structure is flattened and exposed as a Python dictionary with key-value pairs, keys being object names and values being the corresponding configuration information.

The following YAML lines show an example of the configuration of a movable axis (motor) called `rotY`:

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

The information contained in the YAML files is interpreted when the corresponding object is loaded. This interpretation depends on the type of object, in order to create the expected Python object. BLISS has **configuration plugins**, which can be used to extend supported object types. In the example above, the `emotion` (ESRF Motion) plugin is used to interpret the contents of the YAML file, in order to instantiate an `Axis` object.

Configuring Sessions Sessions are defined in the static configuration database, using the `session` plugin. A BLISS session represents an experimental setup associated with experiment control sequences. Indeed, two keys have to be defined in a session YAML file:

- the `config-objects` key is a list of BLISS objects from the configuration,
- the `setup-file` key specifies a Python file to be executed after session configuration objects have been initialized

The setup file is executed in the `setup_globals` namespace of BLISS, so all objects that are defined during setup are exposed to this namespace for further use in user scripts.

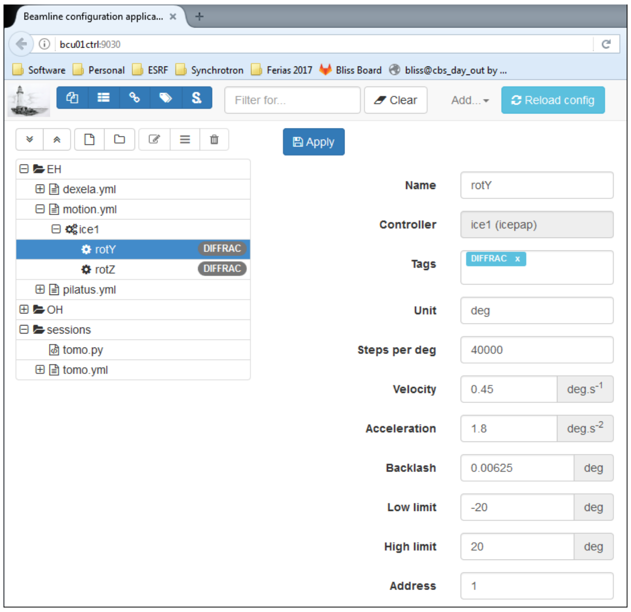
Only one session can be active at a time, however sessions can be nested using the `include-session` key.

Sessions are loaded by the BLISS shell (cf. BLISS shell), giving access to an experimental setup and associated procedures within an integrated command line interface.

Configuration Editor BLISS YAML files can be edited using the Beacon configuration editor (see Fig. 5). The editor runs as a web application, hosted by the Beacon server.

Beacon Settings

Beacon **settings** are configuration values stored in Redis, attached to BLISS objects, that change over time and that need to be persisted across executions.



EH
- dexela.yml
- motion.yml
  - ice1
    - rotY DIFFRAC
    - rotZ DIFFRAC
- pilatus.yml

OH

sessions
- tomo.py
- tomo.yml

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.

Example of settings are motor velocity, acceleration and limits ; default values are stored in the static configuration YAML files, but they are superseded by settings stored in Redis, that correspond to the last values set by the user. Those are applied whenever the BLISS motor object is recreated.

Settings can also be used to persistently keep (or share) information across executions of BLISS, or across BLISS processes like: selected counter for plot, enabled loggers, log level, scan saving path, file template, etc.

Channels

Beacon **channels** leverage the built-in publish/subscribe features of Redis in order to provide a simple way to **exchange data between different BLISS instances**.

Contrary to Beacon settings, channels data is not persisted. When the last client holding the data disconnects, the channel data is cleared.

Use cases for channels are:

- to update state between processes, for example: for Axis objects, position and state (MOVING, READY…) are shared between listeners
- to provide caching, e.g. to skip costly parameters reloading if last set values kept in a channel are the same between calls
- to prevent unwanted hardware initialization, in case the same BLISS object is used by multiple processes, initialization can be skipped if it has already been done.

ASYNCHRONOUS I/O MODEL

BLISS is based on the asynchronous I/O model. Indeed, data acquisition being mainly an I/O bound task, the choice has been made to achieve concurrency using asynchronous I/O coupled with an event loop and callbacks instead of relying on an implementation via OS threads, for example. As a result, BLISS implements cooperative multitasking, which greatly reduces race conditions, locking issues and general problems linked with preemptive context switching.

gevent

BLISS event loop is built on top of *gevent* [12], a coroutine-based Python networking library.

gevent provides lightweight execution units (tasks) based on *gevent* [13].

Green Threads

While I/O operations occur in a task, gevent yields automatically to execute another task. For this reason, gevent greenlets can be seen as "green threads", i.e lightweight, cooperative threads compared to heavier, pre-emptive OS threads. However, communication primitives are blocking by default. For example, a `recv()` call waits for incoming data. BLISS communication helpers ensure I/O operations are gevent-friendly.

Direct Communication with Hardware Communication helpers to connect to serial lines, serial line, gpib, modbus or tcp/udp devices ensure all I/O operations would not block. This is mainly useful when implementing low-level protocols from device manufacturers.

Interfacing TANGO Devices TANGO devices offer a higher-level alternative to communicate with hardware. Thanks to the **green mode** feature of *PyTango* [14], TANGO device proxies returned by the BLISS communication helper are compliant with gevent, thus allowing seamless integration of TANGO-controlled hardware in BLISS.

CONTROL AND ACQUISITION OBJECTS

The integration of hardware devices in BLISS is a three-step process. For any instrument or piece of equipement, first the appropriate configuration plugin has to be determined according to the device type. Then, the hardware controller class has to be written, to be abl$e to communicate with the equipment. This involves the communication helper classes provided by BLISS, or ta TANGO device proxy. Finally, the appropriate control objects have to be written, on top of the hardware controller class. BLISS provides 4 mini-frameworks in order to facilitate hardware integration.

Control Frameworks

Motor Control 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
- trajectories (if hardware supports it)

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

Lima (2D Detectors) All area detectors at ESRF are controlled with Lima [15].

The BLISS Lima base class allows to seamlessly integrate any compatible 2D detector. Due to the standardization provided by Lima, there is no special method to implement since it is based on the Lima Tango server attributes and commands, which are generic.

Multi-Channel Analyzers BLISS has support for Multi-Channel Analyzers, thanks to a base class that encapsulates the underlying low-level hardware controller.

Regulation The Regulation framework helps to integrate equipments like temperature or pressure controllers and provides `Input`, `Output` and `RegulationLoop` classes, to be used on top of a low-level hardware controller.

A **Software Regulation Loop** class emulates a PID correction loop.

Acquisition Objects

Acquisition objects derive from 3 main classes:

- `Scannable`
- `Counter`
- `AcquisitionController`

`Scannable` objects can be identified as actuators, that are initiating a scan. `Counter` objects represent a quantity that is measured during a scan. Counters need to be associated to an `AcquisitionController`. The acquisition controllers are in charge of counter values acquisition, and to insert counters in the acquisition chain.

Control objects can be extended with those base classes, in order to be used within a scan.

SCANNING

BLISS embeds an innovative scanning engine, designed to support a whole range of data acquisition procedures. The majority of scans performed at ESRF can be written in a few lines of Python code, thanks to well-defined concepts and a clean API.

Acquisition Chain

BLISS introduces the concept of **Acquisition Chain** to describe any kind of scan. An Acquisition chain is a tree structure to represent devices involved in a scan and how they are related.

Acquisition Master

Master devices encapsulate triggering controllers like motor controllers (position trigger), 2D detectors (readout trigger), or even software controllers to define a sequence of triggers for the slave devices.

Slave Acquisition Devices

Slave devices encapsulate data acquisition controllers, and define data channels ; data is acquired following the sequence of triggers coming from the master.

Acquisition Chain Building

Acquisition chains do not come from the configuration: they are built in Python code for each scan instead.

Building a chain consists of associating acquisition masters and acquisition slaves, using the `.add()` method of the `AcquisitionChain` object. A scan toolbox provides a set of functions to easily build acquisition chains, using sensible defaults.

An acquisition chain can have multiple top masters, and within a branch, masters and slaves can be nested to any level.

Scan Object

The BLISS Scan object runs the acquisiton chain.

Preparation is decoupled from start to ensure a minimal latency when starting the scan. Indeed, during preparation each equipment is programmed or configured for the scan. During the preparation phase, the acquisition chain tree is traversed in order to prepare the device nodes first, then masters and so on until the tree root ; `.prepare()` is called on each element.

By default, preparation of all equipments is done in parallel.

At start, the same tree traversal procedure is applied as with 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.

Timing statistics are recorded during the scan. This helps to diagnose problems with hardware devices and to debug new procedures.

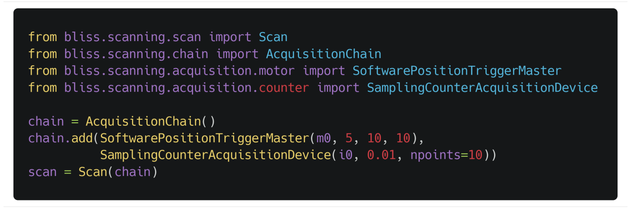
Preset Objects

Preset objects can be added to an acquisition chain, to execute some code before, after, or inbetween iterations.

Simple Continuous Scan Example

In the case of a continuous single-motor *m0* scan acquiring data from an *i0* diode, the acquisition chain can be written as shown in Fig. 6.

It corresponds to the following scan: *m0* will move at constant speed from 5 to 10 (thus, *m0* will actually start before 5 and finish movement after 10), defining 10 points equally



```
from bliss.scanning.scan import Scan
from bliss.scanning.chain import AcquisitionChain
from bliss.scanning.acquisition.motor import SoftwarePositionTriggerMaster
from bliss.scanning.acquisition.counter import SamplingCounterAcquisitionDevice

chain = AcquisitionChain()
chain.add(SoftwarePositionTriggerMaster(m0, 5, 10, 10),
          SamplingCounterAcquisitionDevice(i0, 0.01, npoints=10))
scan = Scan(chain)
```

Figure 6: Code example for a simple continuous scan.

distributed. For each position defining a point, *i0* diode reading will be triggered for 10 milliseconds ; the average value is stored for the scan. In this case, no synchronization hardware is involved, it is a *software continuous scan*, see Fig. 7.

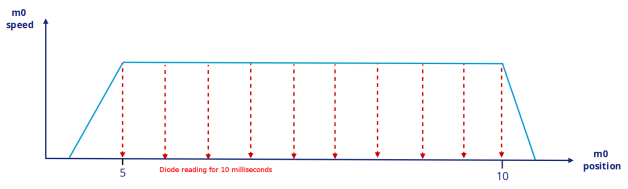


Figure 7: Software continuous scan example.

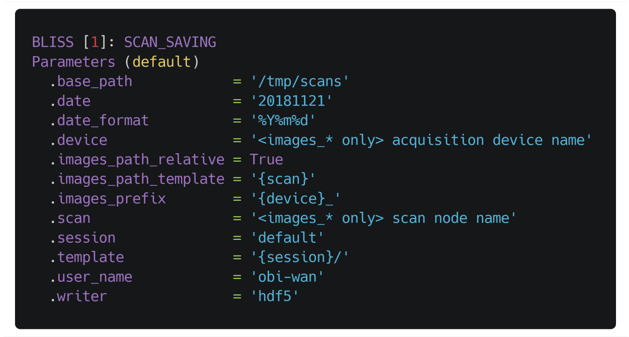
DATA MANAGEMENT

`AcquisitionMaster` and `AcquisitionSlave` objects of the acquisition chain contain **`AcquisitionChannel` objects**. Each acquisition channel has a name, a type and a shape (0D, 1D, 2D…). The role of acquisition channels is to emit data while acquisition is running. Then, data is saved to disk in a scan file, and published to the Redis instance managed by the BLISS Beacon server, enabling online data analysis and other processing on the data stream.

HDF5 Scan Data Files

Scan files are saved in the HDF5 [16] file format. HDF5 files allow organizing data in a tree view structure. BLISS follows the Nexus convention [17],which defines a format within a HDF5 file that can serve as a container for all relevant data associated with a scientific instrument or beamline.

BLISS `SCAN_SAVING` global object helps user to tell where to save the main HDF5 file for scans, and images data (see Fig. 8)



```
BLISS [1]: SCAN_SAVING
Parameters (default)
  .base_path            = '/tmp/scans'
  .date                 = '20181121'
  .date_format          = '%Y%m%d'
  .device               = '<images_* only> acquisition device name'
  .images_path_relative = True
  .images_path_template = '{scan}'
  .images_prefix        = '{device}_'
  .scan                 = '<images_* only> scan node name'
  .session              = 'default'
  .template             = '{session}/'
  .user_name            = 'obi-wan'
  .writer               = 'hdf5'
```

Figure 8: SCAN_SAVING

`.base_path` corresponds to the top-level directory where scans are stored. Then, `.template` completes the path. It uses Python's string interpolation syntax to specify how to build the file path from key values. Keys can be freely added. Key values can be numbers or strings, or functions. In case of function key values, the function return value is used.

`SCAN_SAVING.get()` performs template string interpolation and returns a dictionary, whose key `root_path` is the final path to scan files.

Metadata

Scannable, Counters and Acquisition Controller objects can hold metadata to be stored in Nexus containers along with the scan data.

For all data items, it is possible to store additional metadata that is published with the data stream.

Data Streaming

As soon as the reading task of the scan is running, acquisition channels publish data to the Redis database that is managed by the Beacon server.

Scalar values are stored as plain values, whereas by default 1D arrays are serialized. A reference is stored for bigger data, like 2D images. In Redis, data is stored for a limited period of time (1 day by default) and for a limited amount (1GB by default).

Any program can connect to Redis to access the data stream on the fly to perform data analysis. Clients can listen to the Redis process to be notified of scan progress and to perform data analysis at the same time data is acquired. The scan data flow is represented in Fig. 9.

Online data analysis can be implemented using the BLISS API to connect to the redis stream and to get data while the scan is running. The BLISS API gives access to the data through the more efficient path. For example, in the case of a 2D image from a Lima Tango server, the reference in Redis is resolved on demand and the image data can be directly returned from a Tango server memory, if it is still there, in order to save disk I/O.

BLISS supports complicated data collection protocols with feedback, where the analysed data can be fed into the scanning procedure, in order to better adapt the scan to the sample. Indeed, scans are always executed iteratively, which gives some possibility to change parameters on the fly.

External Data Writer

As of today, data is both written to a HDF5 file and published to the BLISS Redis database. A new development is ongoing, to delegate data saving to an external listener process. Decoupling acquisition from saving and archiving brings the following benefits:

- better performance for scans, with asynchronous scan file writing
- buffering alleviates the real time constraint on the saving in respect to acquisition, and permits intermediate processing
- makes it easier to have both ESRF and user archiving happening at the same time

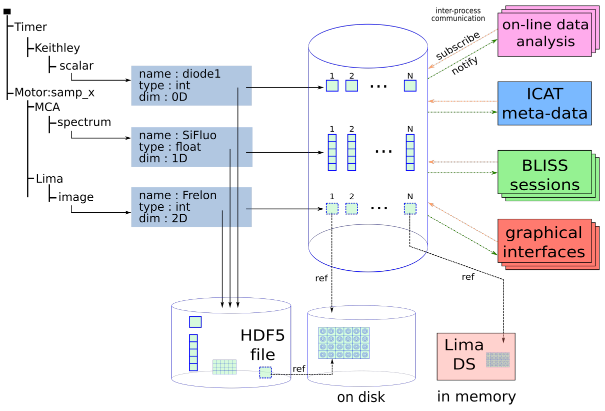


Figure 9: BLISS scan data flow.

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 Shell

The `bliss` command line interface (CLI) is based on *ptpython* [18] and *tmux* [19]. It provides a Python interpreter enhanced with BLISS-specific features.

`bliss` can load BLISS sessions, via a `-s` command line switch. A **typing helper** feature allows user to type commands without taking care of parenthesis and commas inherent to the Python language, in order to have a better experience in interactive use.

Flint

Flint is a *Qt* [20] desktop application made with the ESRF *silx* toolkit [21].

Flint (Fig. 10) is started automatically when a scan starts in the BLISS shell, and listens to Redis in order to display incoming live scan data from the data stream.

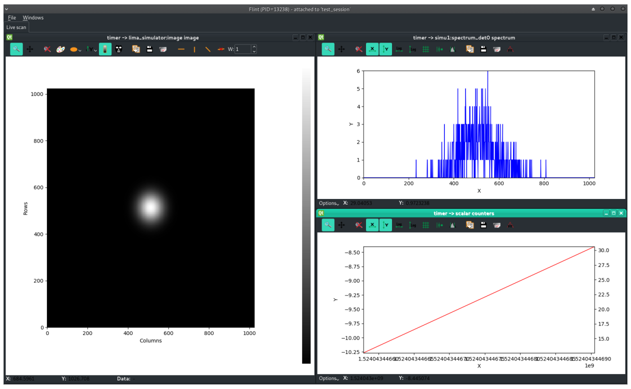

timer -> lima_simulator:image image
timer -> simu1:spectrum_det0 spectrum
timer -> scalar counters
Columns
Rows

Figure 10: Flint: visualization application.

Flint leverages the silx library to provide advanced visualizations like scatter plots or maps, and to perform operations on data like peak search or statistics calculation.

Flint can also plot numpy arrays directly from the BLISS shell. Last but not least, it has some interaction features to define and select areas in a 2D image, or to point and click on curves to move scan motors to specific positions.

BLISS DEPLOYMENT

In 2018, BLISS has been tested on different experimental setups, on various ESRF beamlines. The idea was to assess BLISS technical choices and design, while starting to collect users feedback. During 2019, the development of the project continued in order to integrate the hardware for the first target beamlines. Indeed, today BLISS is in the initial deployment phase at ESRF. In the next years (2020, 2021, 2022) all ESRF beamlines will be fully equipped with the new software.

The migration to full BLISS-controlled experiments will face the refactoring of a huge quantity of existing procedures developed in the legacy control system.

In order to ensure a smooth transition to BLISS, some tools are provided with BLISS to be able to do partial updates and to ease with the transition:

- 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 generic TANGO server that gives access to any BLISS object or procedure
- A **Wago** TANGO server, based on the BLISS Wago controller to benefit of new features in the old system

REFERENCES

[1] J.M. Chaize *et al.*, "The ESRF's Extremely Brilliant Source - a 4th Generation Light Source", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 2010-2015, 2018. `doi:10.18429/JACoW-ICALEPCS2017-FRAPL07`.

[2] LGPL, `https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License`

[3] Conda, `https://docs.conda.io/en/latest/`

[4] Semantic Versioning, `https://semver.org/`

[5] Git Repository, `https://git-scm.com/`

[6] ESRF gitlab Server, `https://gitlab.esrf.fr/bliss/bliss`

[7] github Flow, `https://guides.github.com/introduction/flow/`

[8] Kanban, `https://en.wikipedia.org/wiki/Kanban`

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

[10] Redis, `https://redis.io`

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

[12] gevent, `http://www.gevent.org`

[13] greenlet: Lightweight concurrent programming Motivation, `https://greenlet.readthedocs.io`

[14] PyTango, `https://github.com/tango-controls/pytango`

[15] A. Homs *et al.*, "LIMA: A Generic Library for High Throughput Image Acquisition", in *Proc. ICALEPS'11*, Grenoble, France, paper WEMAU011, pp. 676-679, 2011.

[16] The HDF Group, Hierarchical Data Format, version 5, 1997-2019. http://www.hdfgroup.org/HDF5/

[17] M. Könnecke *et al.*, "The NeXus data format", *J. Appl. Cryst.*, vol. 48, pp. 301-305, 2015. doi:10.1107/S1600576714027575

[18] ptpython — A better Python REPL, https://github.com/jonathanslenders/ptpython

[19] tmux – terminal multiplexer, https://github.com/tmux/tmux

[20] Qt — Widget toolkit and Application framework, https://www.qt.io/

[21] SILX — ScIentific Library for eXperimentalists, https://github.com/silx-kit/silx