ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THSDSC04

CamServer: STREAM PROCESSING AT SwissFEL AND SLS 2.0

A. Gobbo†, A. Babic
Paul Scherrer Institut, Villigen PSI, Switzerland

Abstract

CamServer is a Python package for data stream processing developed at Paul Scherrer Institute (PSI). It is a key component of SwissFEL's data acquisition, where it is deployed on a cluster of servers and used for displaying and processing images from all cameras. It scales linearly with the number of servers and is capable of handling multiple high-resolution cameras at 100 Hz, as well as a variety of data types and sources. The processing unit, called a pipeline, runs in a private process that can be either permanent or spawned on demand. Pipelines consume and produce ZMQ streams, but input data can be arbitrary using an adapter layer (e.g., EPICS). A proxy server handles requests and creates pipelines on the cluster's worker nodes according to rules. Some processing scripts are available out of the box (e.g., calculation of standard beam metrics) but users can upload custom ones. The system is managed via its REST API, using a client library or a GUI application. CamServer's output data streams are consumed by a variety of client types such as data storage, image visualization, monitoring and DAQ applications. This work describes the use of CamServer, the status of the SwissFEL's cluster and the development roadmap with plans for SLS 2.0.

INTRODUCTION

SwissFEL is an X-ray free-electron laser facility at Paul Scherrer Institute (PSI) [1]. It delivers pulses of X-ray radiation with a duration of a few femtoseconds and operating at a repetition rate of 100 Hz [2]. The SwissFEL control system is based on EPICS [3]. Even though all data sources can be accessed via EPICS Channel Access, most of data acquired in the machine and beamlines is said to be *beam-synchronous* [4], meaning each sampled data record is tagged with a pulse identifier. Beam-synchronous data is streamed by the EPICS's IOCs using the in-house developed *BSREAD*, a library that provides data serialization and stream control over ZMQ [5].

Timing and synchronization are critical at SwissFEL. All beam synchronous sources are connected to the SwissFEL timing system [6] that distributes a unique pulse identifier for each FEL pulse. Sources send out data through a stream stamped with the pulse identifier.

Beam-synchronous data sources can be used together effectively when synchronized, meaning that records from distinct sources belong to the same pulse. The SwissFEL data-acquisition system provides tools for synchronization and thus enables online processing of beam-synchronous sources. Data channels from different sources can be efficiently aggregated and combined into new data streams.

Streams from various sources are directed to the *DataBuffer*, which performs functions of data synchronization, dispatching and temporary storage (buffering). The DataBuffer is an in-house development based on Java Spring [7], running in a cluster of 15 servers. Clients can request from it synchronized streams containing data from multiple sources, where all source values are aligned to the same pulse identifier. Data acquisition can be implemented either by receiving such streams or by post-retrieving data using the DataBuffer REST API, requesting a pulse range for a list of channels. Beam-synchronous scalars, waveforms and images are streamed at 100 Hz. The DataBuffer receives typically 8 GB/s of images, and 450 MB/s of scalars and waveforms, retaining data for some days.

Over the past decades stream processing [8] has played an increasingly critical role on the web and within organizations. It offers low-latency results, enabling immediate insights and faster, more informed decision-making. In the context of SwissFEL, stream processing provides effective online feedback, data reduction, filtering, and therefore resource efficiency - saving on processing, storage, and network resources. For example, the machine protection system can disable pulses, and stream processing allows for the discard of data relating to unfit pulses before processing and storage. Additionally, stream processing moves the resource intensive processing from clients to servers, while standardizing the processing algorithms.

CamServer was initially created with the goals of providing unified access to SwissFEL cameras, processing camera data, and streaming images and data forward. With time, it evolved into a generic stream processing system for SwissFEL. Development started in 2017 and features were gradually included over the years. The project is hosted on GitHub [9]. Python [10] was the language of choice for the project, allowing users to benefit from the well-known Python scientific stack to implement custom processing algorithms.

Initially CamServer executed only one standard image processing algorithm for the characterization of the beam. Later, support to custom processing scripts was added and new use-cases were then supported. More than simply sending data to visualization clients, CamServer also had the capability to send and receive processed data to and from the DataBuffer, making it a key component of the SwissFEL data acquisition system. CamServer is capable of processing generic streams (other from camera sources, such as DataBuffer streams), and can merge and synchronize streams before processing.

Initially deployed on a single server, currently CamServer runs on a Linux cluster of 13 servers and can scale further.

† alexandre.gobbo@psi.ch

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THSDSC04

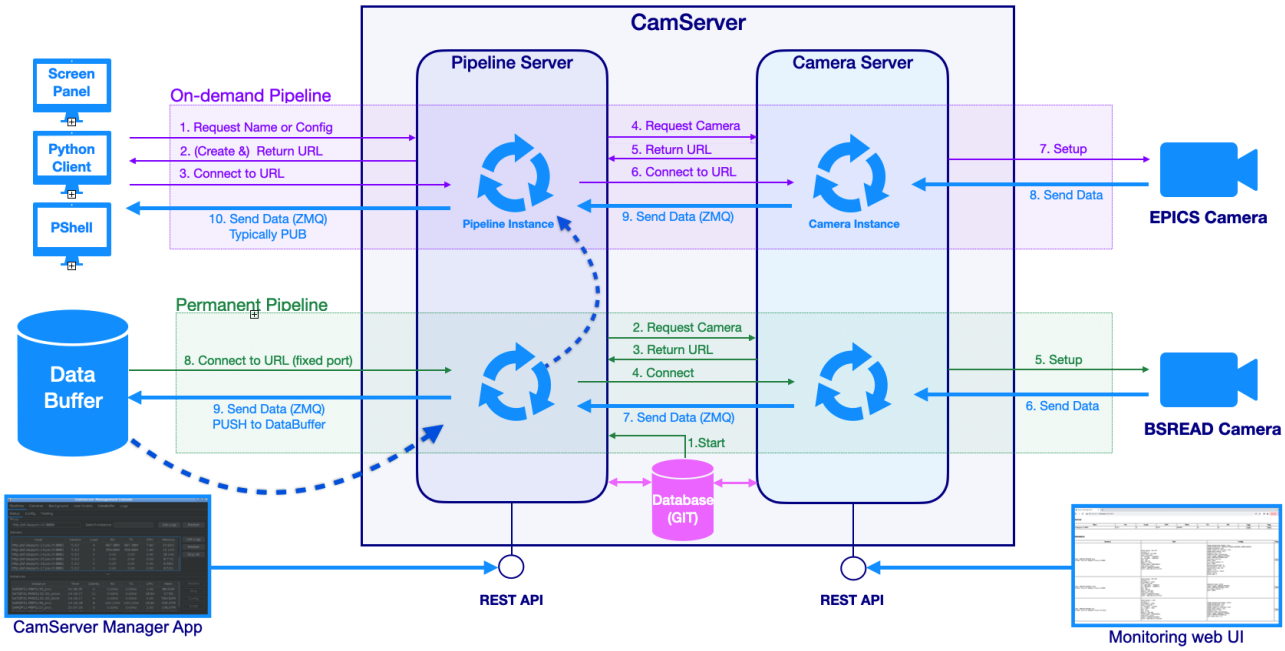

Figure 1: Architecture.

ARCHITECTURE

The data flow and overall architecture of CamServer is presented in Fig. 1. In its simplest form it consists of two services, *CameraServer* and *PipelineServer:*

1. CameraServer ensures there is one single connection to each camera to prevent overloading camera IOCs. It also converts different camera protocols, performs pre-processing, and makes camera data available.
2. PipelineServer performs data processing, optionally aligning streams, and outputs the result to a stream.

Both services share the same architecture, featuring a management process, a database and a REST API. The management process is responsible for initiating processing units, referred to as *instances*, within separated processes for scalability and isolation. Instances are started either automatically, based on pre-defined configurations, or in response to client requests. The database contains camera and pipeline configurations (as JSON files), custom Python scripts, user libraries (packaged as .egg files), and data files (e.g., image background files). The API enables clients to control, monitor, and configure the system. Client libraries written in Python and Java encapsulate the access to the API.

The services offer a web-based user interface for monitoring performance and visualizing running instances. Additionally, there is a desktop management application called *CamServer Manager* [11], designed to streamline system configuration and monitoring. It simplifies the process of uploading scripts, data files, and libraries.

In the case of CameraServer, a camera instance encapsulates the unique connection to a camera. It receives images from a specific source, converts them to a standard representation, performs image correction operations such as rotating, mirroring or cropping, generate the calibrated X and Y axis, and forwards the image and its metadata in a stream. The output is a ZMQ pub-sub stream which can feed multiple processing units and can be also directly accessed by client software or propagated to the DataBuffer.

In PipelineServer, the processing unit is called a *pipeline*. A pipeline instance receives a stream from CameraServer or another source (e.g., the DataBuffer), executes a processing function on the input data and forwards the result in a stream. Optionally a pipeline can receive two streams, aligning them before processing. Pipelines can be cascaded, which means the output of one can be configured as the input of another.

Clients can request the creation of new pipeline instances or access existing ones. They can dynamically modify pipeline parameters, including the processing function, background image, thresholding, and the region of interest.

A pipeline instance is started by providing a unique name and either a configuration structure or the name of a saved configuration. Multiple clients can share the same pipeline instance, meaning they receive data from the same stream. CameraServer allows only one running instance of each image source, and hence only one connection to each camera. In contrast, PipelineServer permits the creation of multiple pipeline instances based on the same configuration, each with a distinct name.

Pipelines can be *permanent* or *on-demand*. Permanent pipelines run continuously and typically serve as sources for the DataBuffer or other pipelines. On-demand pipelines are started by a client and terminate as soon as there are no more connected clients, and after a configurable timeout. When a client requests a pipeline, it receives back the URL of the instance's stream. Camera instances are created as needed when used by a pipeline.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THSDSC04

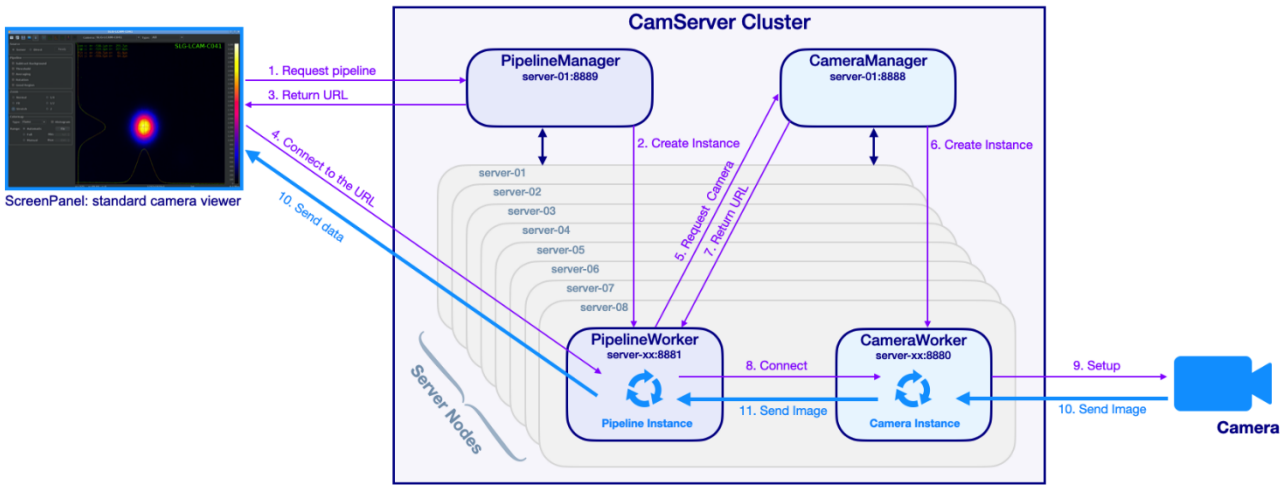

Figure 2: Deployment.

A pipeline executes a processing function on incoming data. This function receives, as parameters, the data dictionary, the timestamp, the pulse identifier, and the configuration dictionary. It must return a dictionary containing the output data, which is then encoded into the output stream. A pipeline can also dump the output to hdf5 [12] files, but in SwissFEL there is no storage mounted in the processing servers, which operate exclusively on streams.

CamServer supports different types of parallelization, which can be configured within the pipeline without requiring changes to the processing function:

- Multi-threading: beneficial for I/O bound pipelines, or else for CPU bound pipelines using *Numba* [13], which can benefit from execution in multiple threads.
- Multi-processing: useful for CPU bound pipelines, but most performant when handling small data, such as low-resolution cameras, due to the necessity of serializing input data to the processes through pipes.
- Fan-out/fan-in: used for CPU bound pipelines processing big data. A fan-out pipeline dispatches data over ZMQ to multiple worker pipelines, and a fan-in pipeline gathers, orders, and forwards the outputs.

As an alternative for performance improvement, CamServer also allows users to define a processing function as a Python C extension. This can be achieved by either uploading the compiled extension library or providing a C file that will be dynamically compiled into an extension.

DEPLOYMENT

Both PipelineServer and CameraServer can scale by dividing each service into a *manager* component, which serves as a proxy, and multiple *worker* components distributed across different server nodes. A *cluster* consists therefore of one server running the manager service and multiple servers (nodes) running the worker service.

In a clustered deployment, the API and client libraries remain identical as when running in single-server configuration. The calls to the manager are propagated to the nodes and URL of instances in the nodes are returned to clients. The data flow of a clustered deployment is presented in Fig. 2.

Each node runs both the *PipelineWorker* and the *ClientWorker* services. In fact, two clusters are formed, one for CameraServer, and one for PipelineServer, using the same nodes. Setting up a cluster requires additional configuration, including defining rules for assigning pipelines to specific nodes and ports. By assigning pipelines to nodes, it is possible to isolate traffic and processing. For example, servers allocated for a particular purpose or beamline will not affect other nodes. Typically, instances are allocated with random ports within a range, and clients access the REST API to obtain the stream URL based on the instance's name. Alternatively, users can configure a specific port for an instance, allowing clients to access the stream directly via a fixed URL.

When creating a new instance, the manager selects a worker node based on the configuration. If no rule exists for that pipeline, a node is selected between the *generic servers* - the ones not assigned to a specific set of pipelines and cameras. In principle, the server with the lower load is selected, but it is preferred to run pipelines and their corresponding camera instances on the same node to minimize unnecessary traffic between nodes.

For many years CamServer services have run within a *Docker* [14] container. A single Docker image was used for all the services. More recently this has changed. Deployment now is done using an *Ansible* [15] script that creates an individual Python environment in each node. This change was done to allow beamlines to install specific packages in their allocated servers if needed, and to simplify and accelerate the development cycle.

FIGURES AT SWISSFEL

Currently the SwissFEL CamServer cluster is composed by 13 servers, each equipped with Intel(R) Xeon(R) Gold 6342 @ 2.80 GHz, featuring 48 cores, 96 threads, and 25 Gb network adapters. Of the 13 servers, 4 are generic, 5 are beamline specific, 3 are used for mission critical pipelines and one for testing.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THSDSC04

At 100 Hz each server is limited by network bandwidth to 2 high-resolution cameras (2560x2160), but it can support numerous low-resolution cameras.

The cluster continuously receives and transmits data at a rate of approximately 5 GB/s. It is currently configured with nearly 500 processing pipelines and 200 cameras.

More than 50 permanent pipelines run continuously, sending data to the DataBuffer. On average more than 10 on-demand pipelines run at a time and are mostly created via *ScreenPanel*, the standard camera viewer application for SwissFEL. ScreenPanel is a CamServer client and is based on *PShell* [16], a data acquisition and display software equipped with tools to facilitate access to CamServer and streams in general.

FUTURE PLANS

There are plans to utilize CamServer in SLS 2.0 [17] for beam monitoring, emittance measurement and bunch length measurement. Currently, a simple single-server setup has been deployed to initiate these developments.

CamServer is also being considered for handling detector data, potentially to provide online feedback for experiments. A first prototype has been implemented in SwissFEL. This requires a working node inside the beamline network, for the processing to be close to the detector. It can be done with a dedicated CamServer deployment, or else controlled by the centralized manager.

One upcoming planned feature may be particularly advantageous for detector data processing and broader applications within SLS and SwissFEL beamlines: CamServer's capacity to initiate and oversee execution of external processes. A specific type of pipeline would be introduced to configure this scenario.

External processes could be developed in any language and would be loosely coupled to CamServer, optionally exchanging data over ZMQ. When starting such processes CamServer would provide connecting points by command line arguments or environment variables. They would be URLs for the process to receive data, send logs, receive configuration changes, and send the output stream. The aim is to give users the freedom to implement the processing pipelines in the way and language they prefer while leveraging CamServer's tools for efficient management, configuration, and real-time monitoring of pipeline execution.

LESSONS LEARNED

The following observations can be useful for similar projects:

- The Python platform inherently presents difficulties for packaging and deployment. Projects must define a strategy for handling them in the long time.
- A dockerized deployment is elegant and consistent but, for the scale of our cluster, it was preferable to abandon it to simplify our development cycle and benefit from the flexibility of customizable environments.
- Python threading is inefficient for CPU bound tasks due to the Global Interpreter Lock (GIL). Strategies to overcome this problem is a necessity in systems that perform heavy processing, for example by implementing true parallelism in C or by multiprocessing.
- NumPy can benefit from external linear algebra libraries. Counterintuitively *OpenBLAS* [18] outperformed *MKL* [19] in our Intel servers, when deployment was dockerized, as MKL had been compiled on a different CPU. Sometimes performance was degraded following an image upgrade, as either library could be used depending on the *Miniconda* [20] base image. A base docker image fixing basic libraries was then used to prevent it.
- Initially all cores of our servers would run at 100% when handling compressed data. That's because the *Bitshuffle* [21] library was not being bound to the proper number of cores. Bitshuffle had to be recompiled, setting the *OpenMP* [22] number of threads to one.
- Up to Python 3.6 there was an issue in NumPy, or one of its libraries, when copying images bigger than the huge page size, rocketing CPU usage to 100%. Like the issue with Bitshuffle, this was only seen in the servers and not on regular computers. The problem happened internally in NumPy when copying read-only images received by other processes before in-place operations. Big images had to be copied by chunks before being sent to NumPy to avoid that. This was solved later in Python 3.8 library stack.
- It is not possible to configure ZMQ, in push-pull pattern, to discard old messages instead of the newest ones when the queue is full. When ZMQ transmission breaks, old messages remain in ZMQ buffer. When communication resumes old messages are send before fresh new ones are. If this behavior is not acceptable the application must set ZMQ buffer size to one and handle the buffering itself.
- CamServer always used the latest versions of *PyZMQ*. Bugs very difficult to track emerged when having massive communication with Java clients using older version of *JeroMQ* - such as in the DataBuffer. It is advised to use latest ZMQ libraries on all languages.
- Finally, projects with uncertain scope must prepare for changes and scaling from beginning in order to succeed.

CONCLUSION

Over the past years CamServer established as one of the core components for controls and data-acquisition at SwissFEL. The system can scale as new cameras and pipelines are continuously being added and new use cases are being implemented. It is likely to become the tool for beam monitoring and measurement at SLS 2.0. It is also an option to provide, in beamlines, online feedback for detector data during experiments.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THSDSC04

REFERENCES

[1] Paul Scherrer Institute, https://www.psi.ch/

[2] Milne C *et al.*, "SwissFEL: The Swiss X-ray Free Electron Laser", Applied Sciences (Switzerland), vol. 7, no. 7, p. 720. doi:10.3390/app7070720

[3] EPICS, Experimental Physics and Industrial Control System, http://www.aps.anl.gov/epics/

[4] S.G. Ebner *et al.*, "SwissFEL - Beam Synchronous Data Acquisition - The First Year", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, paper TUCPA06, pp. 276-279. doi:10.18429/JACoW-ICALEPCS2017-TUCPA06

[5] ZMQ, http://zeromq.org/

[6] B. Kalantari, R. Biffiger, "SwissFEL Timing System: First Operational Experience", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, paper TUCPL04, pp. 232-237. doi:10.18429/JACoW-ICALEPCS2017-TUCPL04

[7] Spring, https://spring.io/

[8] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: the what, where, when, and how of large-scale data processing*, O'Reilly Media, Inc, 2018.

[9] A. Gobbo, A. Babic, CamServer, GitHub, 2017, https://github.com/paulscherrerinstitute/cam_server/

[10] Python, https://www.python.org/

[11] A. Gobbo, *CamServer Manager*, GitHub, 2021, https://github.com/paulscherrerinstitute/cam_server_manager

[12] The HDF Group. Hierarchical Data Format, version 5, https://www.hdfgroup.org/hdf5/

[13] Numba, https://numba.pydata.org /

[14] Docker, https://www.docker.com/

[15] Ansible, https://www.ansible.com/

[16] A. Gobbo, S. G. Ebner, "PShell: from SLS beamlines to the SwissFEL control room", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, paper TUSH102, pp. 979-983. doi:10.18429/JACoW-ICALEPCS2017-TUSH102

[17] H. Braun *et al.*, "SLS 2.0 storage ring. Technical design report", Paul Scherrer Institut, Villigen PSI, Switzerland, Report No.: 21-02, 2021.

[18] OpenBLAS, https://www.openblas.net/

[19] MKL, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html/

[20] Miniconda, https://docs.conda.io/projects/miniconda/

[21] Bitshuffle, https://github.com/kiyo-masui/bitshuffle/

[22] OpenMP, https://www.openmp.org/

.