ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUCG002

DATA ACQUISITION AND ON-THE-FLY PROCESSING FROM HIGH RATE DETECTORS AT MAX IV

A. Lilius, P. Bell, M. Cascella, Z. Matej, A. Salnikov, MAX IV Laboratory, Lund, Sweden

Abstract

At MAX IV, we have developed a high-performance data acquisition (DAQ) system to handle the high rate detectors exploiting the brightness of the fourth generation source. This system integrates multiple detector types, including photon counting and charge integrating detectors as well as sCMOS cameras, into a unified DAQ framework. Data are streamed to a central Kubernetes cluster which mounts an IBM Storage Scale (GPFS) storage, with control provided via Tango. The system provides live feedback from the detectors/cameras and is furthermore extended to provide on-the-fly data reduction via the "Dranspose" framework, a horizontally scalable, distributed data analysis pipeline. We describe the components of our DAQ and processing framework, highlighting its performance for live data streaming and on-the-fly reduction.

INTRODUCTION

The MAX IV synchrotron laboratory in Lund, Sweden, currently operates 16 beamlines for users, providing X-ray techniques in diffraction and scattering, imaging, and spectroscopy covering a wide scientific program. The 3 GeV ring [1] was the first fourth generation light source when it began operation in 2016, offering low emittance, high brilliance and a high degree of coherence. The ability to fully exploit these beam properties is dependent on the performance of the detectors at the beamline experimental end-stations. Although each technique has its own specific requirements, there is a general need for large-area pixelated detectors to run at high frame rates. Such devices generate large volumes of data at high rate which must be captured and stored without losses, in experiments where the detectors are operated continuously for extended periods. Moreover, it is not sufficient to simply record this data to disk for later analysis; to guide the experiments, on-the-fly analysis and visualisation must be provided. These requirements motivate a data acquisition (DAQ) and online analysis framework based on *data streaming* as opposed to file-based operations. In accordance with this, a unified streaming format has been implemented to ensure more consistent and reliable streaming across the facility network. The system standardisation is necessary to cover all detector types for ease of support and operation, but also enough flexibility for different applications.

MAX IV DETECTOR-DAQ SYSTEM

An overview of the detector-DAQ scheme including a simplified view of the network connectivity is shown in Fig. 1. It can be divided into a control and data streamer layer, which is detector-specific and runs close to the detector at the beamline, and a data stream receiver and processing layer, which is detector-agnostic and takes place in a central Kubernetes-managed DAQ cluster. The DAQ cluster mounts the IBM spectrum scale GPFS storage where data are saved in hdf5 files, directly as raw detector images and optionally after some processing step(s). The two layers are described in detail in the subsequent sections, followed by a description of the operation of the combined system.

Data Streaming From the Beamline

Hybrid photon counting from commercial suppliers are usually delivered with a custom server referred to as the Detector Control Unit (DCU). The communication protocol between the detector head and the DCU is typically a closed system that we do not have access to, using for example UDP over twisted pair or optical fibre. In case of CMOS cameras, a server fulfilling the role of the DCU is usually provided by MAX IV. This may hold a "frame grabber" card supplied with the camera, for example a CoaXpress PCI Express card or a Camera Link PCI Express card. Additionally, some CMOS cameras use a USB3 connection. The distance between the DCU located in the beamline electronics rack and the camera in the experimental hutch means that for these types of connections some method of range extension is often required, for example by conversion of the Camera Link to optical fibre and back.

For genuinely on-the-fly analyses and visualisation it is necessary to operate on a data stream rather than on files. This data stream utilises a dedicated "purple" network separate from the "blue" network of the control system (which at MAX IV is Tango [2]). The Eiger detectors from Dectris [3] are unique in natively providing a stream of data from the DCU over ZeroMQ, typically over a single 40 Gbit fibre network interface. For all other detectors and cameras, we have developed a *streamer* application that runs on the DCU, providing a data stream from a ZeroMQ push socket similar to that of the Eiger. The streaming format most commonly used and the one that has been chosen as the standard uses multipart JSON and binary encoding for the message headers and frames, respectively. On the DCUs the ZeroMQ stream is made available on a 40 Gbit fibre network interface (or 10 Gbit if sufficient for the data rate of the detector). Data are then streamed to the DAQ cluster over dedicated 100 Gbit capable Ethernet (purple line in Fig. 1).

The implementation of the streamer application is specific to the particular detector or camera. In some cases the streamer application can be developed and deployed on the DCU thanks to a high level API or SDK from the detector or camera supplier, which provides methods to get the image frames into memory. Implementation in Python is preferred, though sometimes with bindings to code supplied in another language, such as C via CFFI. The Dectris Pila-

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUCG002

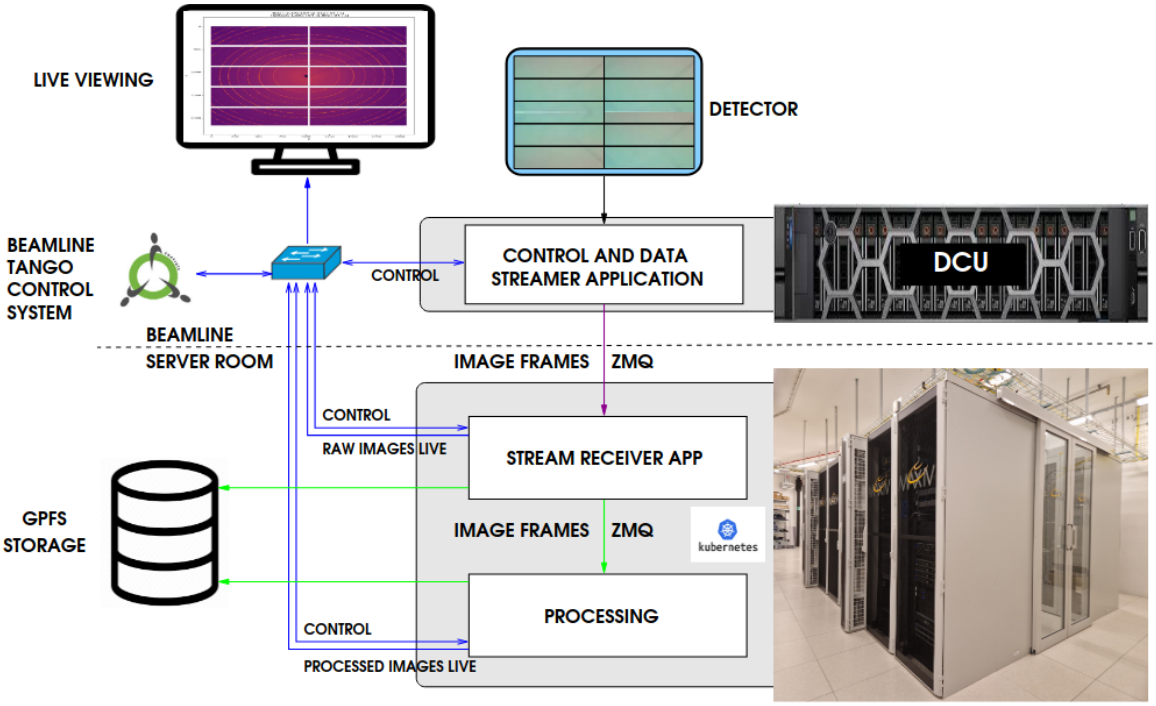

Figure 1: Overview of the MAX IV detector-DAQ scheme. A detector-specific data streamer application runs on the Detector Control Unit which has a 40 Gbit data connection to the Kubernetes DAQ cluster (purple) and a 10 Gbit connection to the beamline control network (blue). A general purpose stream receiver application runs on the DAQ cluster for each detector, with optional additional processing steps. Data are streamed between each layer over ZeroMQ. A slow connection between the DAQ cluster and the beamline control network allows control and monitoring of the status of the stream receiver and for live-viewing of raw or processed data at low frame rates.

tus3 [4] and XSpectrum Lambda [5] detectors are special cases and merit further description. For the Pilatus, acquired images are written as files to a RAM disk on the DCU by the Dectris-installed "camserver" application. Our streamer application is notified as these files are created, reads them into memory and then push them out over ZeroMQ. For the XSpectrum Lambda, this is only system currently in operation that is served by more than one DCU; in this case one server per two of the four modules (each with at 40 Gbit connection to the DAQ cluster). The delivery of image frames into memory synchronised between the DCUs is handled by a provided system library against which the streamer applications are built. Four streams are generated, one per module, which must be recombined in the receiver.

For the photon counting detectors lossless compression is always applied on the DCU. In the case of the Eigers, Bitshuffle-LZ4 is applied by the Dectris software before the frames are pushed. The factor of compression vary between few and few hundred depending on application [6]. The streamer application deployed on the Lambda DCUs uses the same compression algorithm. For the Pilatus, the files are created on the DCU in compressed Crystallographic Binary File (CBF) format and the streamed image data retains this compression. For the CMOS cameras which are not noiseless however, i.e. do not contain many pixels with zero values in any given frame, the lossless compression does not work well and is not applied.

Except for the Eiger detectors, which have their own HTTP-based REST-like control interface, the streamer application also gives the control over the hardware, e.g. to set exposure times and and start and stop acquisitions. Where we have full access to the DCU, as in the case of the CMOS cameras, we install Tango there and the streamer application is realised directly as a Tango device. Where this is not convenient, for example because the supplied DCU has a non-standard operating system[1], Tango is not installed and the streamer application is developed with the fewest possible dependencies. The application would then provide its own remote interface to allow a Tango device (written in Python) running on one of the standard control system machines to interact with it. Considering again the case of the XSpectrum Lambda, the DCUs are preinstalled with Debian and the streamer application written in C++ implements a ZeroMQ request-reply interface to communicate with a remote Tango device. For the Eiger it is straightforward to create a Tango device running on any beamline control system machine to communicate over the REST API provided. In all cases, the DCUs have a 10 Gbit copper connection to the beamline control system (blue connection in Fig. 1.)

[1] The MAX IV control system uses Rocky Linux with some legacy CentOS machines

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUCG002

For all detectors and cameras, the Tango device presents a uniform interface, a "MAX IV standard detector device". An essential, minimal set of commands and attributes are exposed, notably an Arm command and self explanatory attributes TriggerMode, ExposureTime and nTriggers, abstracting however these settings may be referred to in the underlying interface (SDK etc) and thereby hiding the complexity of the specific hardware.

Data Reception in the DAQ Cluster

As described above, data (image frames) are streamed over dedicated 100 Gbit capable Ethernet to the DAQ cluster, though all systems discussed here use a maximum 40 Gbit connection at the DCU. The DAQ cluster itself is divided across two server rooms with an internal architecture shown in Fig. 2.

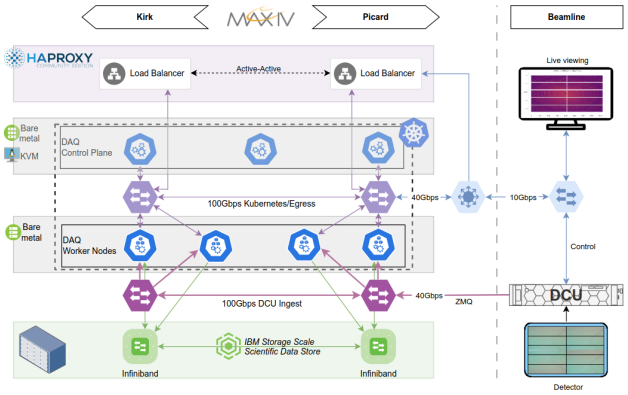

Figure 2: Overview of the MAX IV IT infrastructure for DAQ with a focus on the internals of the DAQ cluster.

Since the streamer application running on the DCU generates a data stream loosely following the standard of the Dectris Eiger, a common *stream receiver* application has been developed to handle all detectors and cameras. A schematic of the application is shown in Fig. 3. Written in Python, it consists of horizontally scalable number of *workers* (one or many ZeroMQ pull clients) to receive the frames, a *collector* for ordering and distribution of the data, an hdf5 *file writer* and a *forwarder* (ZeroMQ push socket) for additional online processing of the original data in the DAQ cluster. The two queues, *worker queue* and *writer queue*, are custom Python deque objects designed to provide additional thread safety during put and get operations, as well as support asynchronous get calls. The worker queue is used by the workers to place received ZeroMQ messages, which the collector retrieves and orders. The messages consists of a header with meta data including the frame number, and the collector goes through the messages from the worker queue asynchronously and orders them based on this number; it is essential that the position of the frame in the eventual data file matches the trigger number that generated it. Once a message is ordered, it is placed in the writer queue, where the file writer retrieves it and writes it to disk. The file writer process uses the h5py library and implements a minimal version of the NeXuS NXDetector base class. The header message includes various meta data fields of this NeXuS class including detector settings such as whether various corrections are applied, the exposure time, etc.

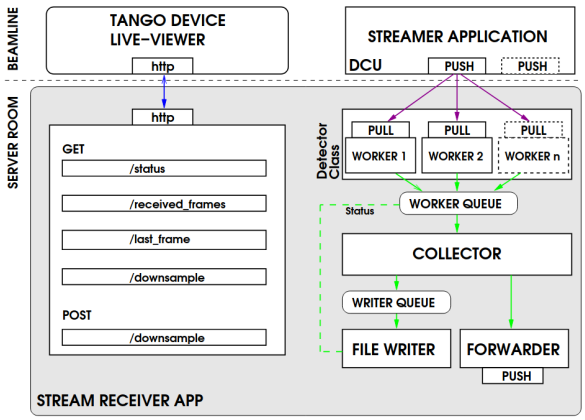

Figure 3: The stream receiver zoom in of box in Fig. 1. Tango device may be running on DCU or within distributed control system hence streamer may be part of same application or can involve some communication between the two, e.g. HTTP.

The stream receiver code has a few adaptations to accommodate the specific peculiarities of certain detectors, limited to a Detector Class for which custom workers can be implemented if needed. For the Pilatus, the stream receiver workers must uncompress the images in CBF format and recompress them with Bitshuffle-LZ4 to be in common with other files from the photon counting detectors; with eight workers, this uncompressing and decompressing keeps up with the 250 Hz frame rate of the Pilatus3 2 Megapixel detectors. For the Lambda, four pull sockets, each with one worker, are used to receive the data from the four modules. Saving the four modules as one image would equate to a 9 Megapixel image, so instead the file writer saves the data from each module into a single file but as a stack of four images per frame with coordinates and rotation information as meta data to allow the final shape to be recreated in subsequent data analysis.

The stream receiver application provides an API through an HTTP interface for checking, for example, the state and status and number of received frames, and to obtain the most recent frame for live-viewing. Once deployed in the Kubernetes DAQ cluster, the stream receiver's HTTP address is exposed via an ingress. The state and status follow and simple state machine in which the allowed states are Idle, when waiting for a "series start" header message, and Running, after receiving and header message but before an "end of series" message. A simple live-viewer based on SILX [7, 8] to be made available for all detectors and cameras, which works by continuously making HTTP requests to the stream receiver for the latest frame and displaying it in a SILX GUI widget. There is an option to downsample the images for the

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUCG002

live-view which is essential for some larger area cameras for which the data rate, even at 1 Hz, would fill the bandwidth of a client computer in the control room.

Instances of this stream receiver application are deployed as needed on the Kubernetes DAQ cluster using a Helm chart. The Helm chart provides templated manifests, and each deployment is customised through a YAML file specifying the configuration. Helm renders the chart into Kubernetes manifests, which create Pods running lightweight Docker containers that host the stream receiver instances. The Pods run within the DAQ cluster, scheduled across available nodes in the Kubernetes environment. These processes run under a beamline service account, which has write permissions for the corresponding beamline folder in the GPFS file system. This setup ensures that only authorised machines at MAX IV can read and write to the central storage, preventing unrestricted access. Additionally, the YAML configuration file defines the number of workers for each receiver, the host and port to connect to, and the port used to republish or forward frames. The data forwarded by the stream receiver at full rate serves as the entry point for downstream analysis pipelines and, for live analysis, is primarily consumed by Dranspose.

LIVE ANALYSIS AT MAX IV — DRANSPOSE

During beamtimes, critical decisions on how to proceed with an experiment must be made constantly. Therefore, providing feedback with the best possible data analysis with the lowest possible latency, mostly in the form of visualisations, is important. For low data rates, writing and monitoring a file works well. However, processing tens of gigabytes per second is difficult with a filesystem, and streaming solutions are therefore preferable. Apart from the high throughput, it is important that the users consuming the feedback are able to quickly tweak the analysis to adapt to changing conditions. While there exist generic stream processing frameworks like Apache Spark, a preferable solution is to combine all major relevant features from such tools into a more applicable application for our purposes.

At MAX IV Dranspose, a data analysis pipeline for high-throughput data acquisitions, has therefore been developed. It is a horizontally scalable, distributed system deployable to a Kubernetes cluster, relying on Redis for coordination and ZeroMQ for data streaming. A novel map-reduce was chosen as programming paradigm for this project. In a classical map-reduce, every event is processed by an independent worker which forwards their result to a reducer. Often, experimental data has temporal dependencies such as the evolution over time, which in standard map-reduce is only analyzable in the reducer. The constrained map-reduce implemented in Dranspose enables stateful workers combined with load balancing for fast temporal difference calculations by sending consecutive frames to the same worker. Additionally, it enables event formation from multiple sources to get a complete data view for a specific trigger. This is essential when data from one detector needs to be combined with that from another, including for example in an I_0 normalisation, or when detector data must be interpreted with some other information such as the position of a moveable in scan. The reducer writes the analyzed data to an HSDS service, which provides standardised, simultaneous access for visualisation tools like SILX, h5pyd, or h5web. Dranspose takes care of the orchestration and distribution of data but allows users to easily adapt and update the map and reduce Python functions. For ease of development, record and replay actions are provided to develop the analysis on a local machine before the experiment.

Several applications for Dranspose include merging rotation stage encoder positions to a CMOS camera at 3GB/s for live tomographic reconstruction, azimuthal integration with I_0 normalisation to, live XRF concentration mapping, or crystallography spot finding. With gained confidence from the scientific users, Dranspose is useful for data reduction, and the feedback may be directly consumed by the control system to decide on the progression of a scan in a closed loop.

SYSTEM PERFORMANCE

The ultimate performance of the DAQ system can be limited by both hardware and software components. For example, the streamer and receiver software is built on ZeroMQ is therefore subject to certain limitations imposed by the underlying parts of that library. These limitations may in turn depend on the hardware (CPU, memory, network) on which the ZeroMQ-based applications run. Or additionally, having the stream receiver application write hdf5 files to disk makes the overall data throughput from the detector to data storage dependent on the hardware of the cluster nodes. Since the system consists of many interdependent components, the entire chain must be tested as a whole to find the overall effective limit and then the bottlenecks identified.

To evaluate the operational capacity of the system, tests have been conducted in an isolated environment made to replicate the system in place. A data generator in place of a real detector was paired with an instance of the usual stream receiver (with of a single worker). They were deployed on separate nodes within the DAQ cluster, simulating a real-world scenario similar to the current setup at MAX IV but avoiding the 40 Gbit connection from the DCU to the Kubernetes ingest. The data generator mimics a detector by generating dummy frames and streaming them to the stream receiver, which receives them and writes them to disk in the usual way. The generator's frame rate and frame size can be regulated to evaluate the point at which the stream receiver is no longer able to keep up with the resulting incoming data rate. This approach allows us to study potential future scenarios involving hypothetical detectors with significantly higher data rates than we have today, allowing us to identify the limits of the current system and thereby plan for the next generation of detectors.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUCG002

Two aspects were evaluated during the testing: the frame rate, in fps, from the generator to the stream receiver and written to disk; and the network throughput in GB/s, measured using the psutil [9] Python package during the tests. To cover a range of possible detector sizes (in megapixels) and compression rates, frame sizes of 20 kB, 200 kB, 2000 kB, and 20000 kB were used in the measurements (For reference, an Eiger 1M detector with a factor 10 compression and 16 bit depth would correspond to a frame size of 200 kB). To establish reliable statistical error bars for each data point, every measurement was repeated ten times under identical conditions. The final data points represent the average of these repeated measurements and the error bars are the standard error of this mean.

Figure 4 presents the results of the frame rate evaluation, illustrating how well the stream receiver keeps up with the generator's frame rate for different frame sizes. The linear dotted blue line shows the hypothetical ideal performance where the stream receiver copes with any generated frame rate. It can be seen then for example that a detector generating frame sizes of 200 kB, after any compression, could be handled by the DAQ system up to around 2000 fps. The Dectris Pilatus (2M) and XSpectrum Lambda (3M) detectors, in use at MAX IV, have been marked on this line at their respective operating frame rates to provide perspective on the current uses of the MAX IV system.

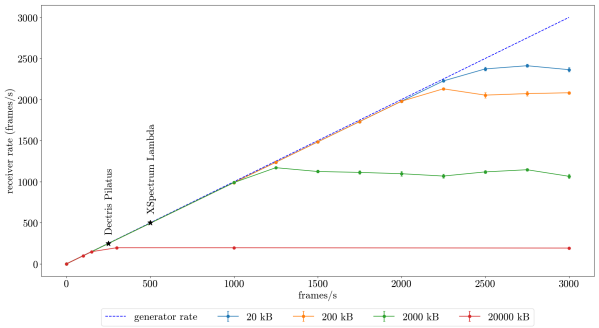

Figure 4: Measured operating frame rate of a stream receiver instance deployed in the MAX IV DAQ cluster, including writing of data to disk, as a function of generated input frame rate, for different frame sizes. The dotted blue line indicates hypothetical perfect performance of the system. For reference the operating frame rate of a selection of detectors are shown, which may not be the maximum capability of the detector.

Figure 5 shows the stream receiver's throughput in GB/s for the same selection of frame sizes. It can be observed that while the highest achievable frame rate, approximately 2 000 frames/s, is attained with the smallest of these frame sizes (20 kB), the highest data throughput, around 4 GB/s, is achieved with the largest frame size (20 000 kB). This indicates a limit somewhere in the DAQ chain in operations per second rather than a network bandwidth limit. Small frames are not processed proportionally faster than large ones, i.e. 20 000 kb frames can be processed at 200 fps (Fig. 4) but 200 kb frames cannot be processed at 20 000 fps and instead use a much smaller fraction of what is shown to be the available data throughput of at least 4 GB/s. Studying the system as a whole, the origin of the frame rate limit is not known. However, one explanation can be the writing of files to disk with h5py. If the file writing is disabled in the stream receiver, still with one worker it *is* now observed to run at over 20 000 fps for frames of 200 kb and thereby make full use of the 4 GB/s bandwidth.

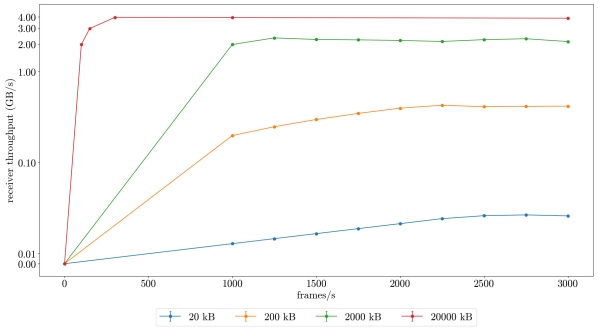

Figure 5: Measured operating data rate of a stream receiver instance deployed in the MAX IV DAQ cluster, including writing of data to disk, as a function of generated input frame rate, for different frame sizes.

CONCLUSION

The MAX IV DAQ system described in this paper serves over 30 detector systems, operated on a daily basis. This results in large amounts of data from many different sources at potentially the same time. With a DAQ kubernetes cluster in place and a highly operational stream receiver application, developed to handle a unified ZeroMQ data stream, this is managed fully. Live analysis in the form of Dranspose has been developed to help guide users to alter their experiment on-the-fly and save potential time and increase its quality. By applying all efforts toward a better operational facility, MAX IV handles data rates of 2 kHz, including file writing, and throughput rates of at least 4 GB/s over the network. These rates are fully capable of handling the detectors currently in place. However, work to maintain and develop an even more capable DAQ system is under way, for any future higher rate detector that might be used at MAX IV.

REFERENCES

[1] P. F. Tavares, S. C. Leemann, M. Sjöström, and Å. Andersson, "The MAX-IV storage ring project", *J. Synchrotron Radiat.*, vol. 21, no. 5, pp. 862–877, Aug. 2014. doi:10.1107/s1600577514011503

[2] J.-M. Chaize *et al.*, "TANGO - An Object Oriented Control System Based on CORBA", in *Proc. ICALEPCS'99*, Trieste, Italy, Oct. 1999, paper WA2I01, pp. 475–479.

[3] A. Casanas *et al.*, "EIGER detector: application in macromolecular crystallography", *Acta Crystallogr., Sect. D: Biol.*

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUCG002

Crystallogr., vol. 72, no. 9, pp. 1036–1048, Aug. 2016. doi:10.1107/s2059798316012304

[4] T. Loeliger, C. Brönnimann, T. Donath, M. Schneebeli, R. Schnyder, and P. Trüb, "The new PILATUS3 ASIC with instant retrigger capability", in *Proc. 2012 NSS/MIC*, Anaheim, CA, USA, Oct. 2012, pp. 610–615. doi:10.1109/nssmic.2012.6551180

[5] D. Pennicard *et al.*, "The LAMBDA photon-counting pixel detector", *J. Phys.: Conf. Ser.*, vol. 425, no. 6, p. 062010, Mar. 2013. doi:10.1088/1742-6596/425/6/062010

[6] T. Donath *et al.*, "EIGER2 hybrid-photon-counting X-ray detectors for advanced synchrotron diffraction experiments", *J. Synchrotron Radiat.*, vol. 30, no. 4, pp. 723–738, Jun. 2023. doi:10.1107/s160057752300454x

[7] silx 2.2.2, http://www.silx.org/doc/silx/latest/index.html

[8] silx-kit/silx: 2.1.0, https://zenodo.org/records/10996641

[9] psutil documentation, https://psutil.readthedocs.io/.