ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA06

SwissFEL - BEAM SYNCHRONOUS DATA ACQUISITION – THE FIRST YEAR

S. G. Ebner, H. Brands, B. Kalantari, R. Kapeller, F. Märki, L. Sala, C. Zellweger,
Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

The SwissFEL beam-synchronous data-acquisition system is based on several novel concepts and technologies. It is targeted on immediate data availability and online processing and is capable of assembling an overall data view of the whole machine, thanks to its distributed and scalable back-end. Load on data sources is reduced by immediately streaming data as soon as it becomes available. The streaming technology used provides load balancing and fail-over by design. Data channels from various sources can be efficiently aggregated and combined into new data streams for immediate online monitoring, data analysis and processing. The system is dynamically configurable, various acquisition frequencies can be enabled, and data can be kept for a defined time window. All data is available and accessible enabling advanced pattern detection and correlation during acquisition time. Accessing the data in a code-agnostic way is also possible through the same REST API that is used by the web-frontend. We will give an overview of the design and specialities of the system as well as talk about the findings and problems we faced during machine commissioning.

OVERVIEW

As described in the general SwissFEL paper [1], there are two categories of data to be dealt with at SwissFEL, namely synchronous and asynchronous data. This paper will only focus on the synchronous part of the system although both are using same/similar infrastructure and software.

Figure 1: Basic building blocks.

As shown in Figure 1 the beam synchronous data acquisition system consists of simple basic building blocks. Each block will be described in detail in the following sections.

The beam synchronous data acquisition system consists of two independent subsystems dealing with **small** (scalars and waveforms) and **large** data (cameras, detectors). Both subsystems consist of the same building blocks although different software components are used. The differences we will be outlined in the respective sections below.

Sources

All beam synchronous sources are connected to the central SwissFEL timing system [2]. The realtime timing system distributes the readout triggers as well as the unique pulse-id for each FEL pulse. After reading out the data upon receiving a readout trigger, the sources immediately attach the corresponding pulse-id to the data and send out an atomic message including all readout data and pulse-id.

Each readout value of a source is called channel and a source can have various channels.

Each source can be dynamically configured via a REST API, i.e. what channels to read out and in what frequency, without the need of reboot or restart.

A source can be implemented in various ways. At the moment, there are sources implemented as Epics IOCs and real-time applications running on a real-time Linux at SwissFEL.

Synchronization / Dispatching

Data send out by the sources is received by the Dispatching layer. For small data, this layer consists of currently twelve machines that form a cluster with the ability to synchronize data on the fly.

Clients can transparently request synchronized streams of channels coming from different sources via a REST API from this layer. Upon a client request, the Dispatching layer creates a customized data stream for the client as if it would originate from a single source. This technique frees the client from synchronizing data on its own.

Once the client disconnects from the custom stream, the Dispatching layer takes care that the stream gets closed and all required resources are cleaned up.

The Synchronization and Dispatching layer decouple the sources from the clients. Therefore, sources are protected from being overwhelmed by client requests.

Beside the ability to synchronize data and provide custom streams this layer also forwards all data to the buffering layer that is hosted on the same machines.

For large data this layer currently consists of currently one machine taking care of the receiving of camera data, compressing the images, doing (optional) standard analysis on the data and passing the data on to the large data buffering system.

Buffering

The Buffering layer temporarily stores all beam synchronous data for later retrieval. At the time of writing, the retention period of data inside the buffer is two days for scalar values, two hours for waveforms and two hours for images.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA06

After this time, data is being reduced to 1 Hz and is kept for one additional month until it is discarded.

Next to the default retention policies, users can register custom retention policies, i.e., the retention time and reduction algorithm can be specified for individual or groups of channels.

Data can also be tagged manually to be kept for longer. This is done for example for RF data once a breakdown is detected for a cavity.

For buffering small data, we currently use a cluster of twelve machines with local SSD storage. For buffering large data, we use one machine connected via Infiniband to a GPFS filesystem.

We use a proprietary file based persistence approach for the data to be buffered. Each channel is stored in a single file which is rotated after some time.

Both buffering systems can be queried and data can be retrieved via a REST API. The exact same API can also be used by users to query non beam synchronous data

Online Analysis / Feedback

Online analysis and feedback enables users to see, analyse and react to the incoming data. This kind of applications request and use a customized stream of data with all required channels from the synchronization and dispatching layer. By doing so the amount of coding needed boils down to simply implement the analysis and/or visualization logic.

Right now we have various applications in this layer ranging from Python, Java and Matlab. Some of the applications used are presented in [3].

Communication Protocol

The communication protocol used for transferring data between the components is called BSREAD. It is based on ZMQ [4] and follows a design used and established for large scale detectors at the Paul Scherrer Institute. Data transmitted with this protocol is self-describing and makes use of the multipart message feature of ZMQ.

For each machine pulse there is one message send from a sender (see Figure 2).

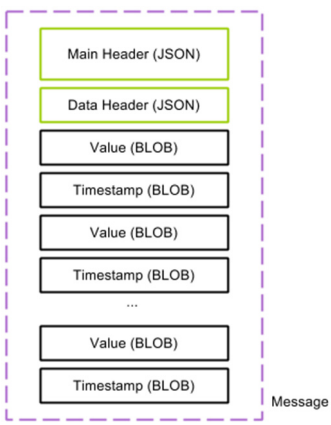

Figure 2: Message.

The first part (sub-message) of the message consists of a JSON string holding the protocol version, the pulse-id, global timestamp and a hash and compression of the next sub-message of the message.

The second part of the message consists of a JSON string holding an ordered dictionary for each channel transmitted for this pulse with the name of the channel, type, number of elements, compression and other metadata.

The remaining sub-messages of the message consist of the individual channel values and timestamp (each a sub-message) in the sequence the channels are mentioned in the data header.

Right now we are supporting bitshuffle-lz4 [5] compression for data and data header. We apply compression to images and waveform data of a specific size. We can reach compression factors up to 5.

While using the ZMQ protocol, we make use of the built-in delivery schemes push/pull and pub/sub to achieve failover and load-balancing.

Web Frontend

Data within the buffering system can be browsed using a web UI (see Figures 3, 4, 5).

The web UI sits on top of the exact same REST API exposed by the buffering systems to extract data from it.

To allow fast and responsive browsing of data, we apply simple but powerful data reduction on the server side.

For example, if a query for a channel returns more than 512 data points, we use data binning and calculate the min, max and mean value for each bin and show this reduced data instead.

Using this approach, we avoid transmitting large amounts of data points that cannot be displayed accurately on the client's machine as the screen resolution is not high enough. Also, this positively impacts user experience as the UI remains responsive, even for large amounts of underlying data.

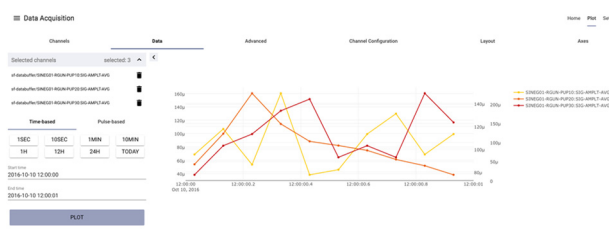

Figure 3: Web Frontend showing scalar data.

The same reduction is applied for waveform data. Instead of transmitting the whole waveform for each data point, we only transmit the min, max and mean value of the waveform. And again, if more than 512 data points are queried/visualized, we additionally start binning and calculate the bin's min, max and mean.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA06

Figure 4: Web Frontend browsing waveform data.

In case the user needs to see the raw value, he can interactively zoom into the data by holding a specified keyboard key. This will reload the data only for the zoom range and apply the same reduction procedure described as before until the raw data is reached.

Figure 5: Web Frontend browsing image data.

For efficiently visualizing images from the ImageBuffer in the web UI similar binning and reduction approaches are applied. We generate and transmit a PNG image on the server side unless the raw data of a single image is requested. This has been shown to be much quicker than transmitting raw data and displaying this data in a 2D fashion using a plot library. To further improve responsiveness and loading time, lazily loading of images, based on the user's scroll position, has been implemented.

The Web UI can also be used to query non beam synchronous data from the Epics Channel Access Archiver. This way users are able to (roughly) correlate non synchronous and synchronous data within the same tool.

Implementation

The implementation of the various systems is done in different programming languages.

Current production sources are implemented in C and C++. For testing and verification, we have Python and Java implementations.

The Synchronization / Dispatching as well as the Buffering layer and all REST APIs are implemented in Java. We are extensively using Java 8 streams and asynchronous processing. Here, the most important libraries we use: SpringBoot [6], Netty [7], Hazelcast [8], Jeromq [9], and Bittshuffle-lz4.

For the online analysis we are providing a Python library that can be used by scientists. This library is also used from within Matlab through the Matlab/Python interface.

The web frontend is implemented in Polymer [10]. For plotting, we are relying on the Javascript-based plotting library plot.ly [11].

LESSONS LEARNED

There are a lot of lessons learned for the first year of operation. Generally, we can distinguish between technical and human/social lessons learned.

Technical

The most important technical findings are:

- **Keep things clean, slim and simple**. This includes the reduction of system interfaces as well as dependencies to libraries and frameworks. Over the course of a year we replaced our Cassandra based storage with a custom file-based storage. In order to achieve, e.g. data locality, we had to circumvent built-in features of Cassandra. Although we lost some (nice to have) features which Cassandra provided out of the box, we immediately saw a ten- to hundred-times performance gain and ended up with less code.
- **Have well defined REST APIs** rather than libraries for special programming languages.
- **Avoid data copies**. In Java use direct byte buffers and operate on bytes for performance critical code. An efficient way to avoid data copy from the network stack is the use of the Netty.io library.
- **Compress data**. Compressing data while transmitting and storing greatly improves performance. Bitshuffle-lz4 is a very good compression for most the cases.
- **Avoid premature optimizations**. A comment we hear often is that transmitting information via JSON is inefficient. In none of the components is this an issue or the limiting factor. Using a different serialization would make things more complicated. Also optimizations we introduced at the beginning based on the frequency of incoming data proved to be counterproductive and caused a lot of issues as it turned out that certain sources do not send data regularly.
- **Use asynchronous processing and concepts**. The use of (Java) asynchronous data processing and concepts speed things up significantly and lets the system behave more responsively.
- **Use Solid State Disks (SSD)**. Data access pattern of buffering (and archiving) systems are random I/O

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA06

especially for data recorded recently. SSDs are ideal for these non-regular access patterns.

- **Spend (lots) of time on the (web) UI**. Having to deal with tons of data the (web) UI has to be carefully designed and implemented. Without clever data reduction and interaction, a responsive UI and satisfactory user-experience is not possible.
- **Testing**. Although testing is very time consuming, it is key to the success of the system. Every time an error occurs, add unit tests to reproduce, fix and test for the error. Lots of errors cannot be tested on test systems via unit and integration tests upfront, some only occur on the live system. Therefore, set aside time for such production-level issues.
- **Collect statistics**. Statistics are first class data to collect and store. Information on whether/when channels are connected/disconnected, how many times a connection was dropped, and data is accessed, are all important for operating such a system. Also the detailed knowledge on up and downtime of the system(s) are essential to draw the right information out of data.

Human / Social

The most important human/social findings are:

- **Mindset.** People are not (yet) accustomed to the large amount of data collected and provided by the beam synchronous data acquisition system. In general, the issues that arise with "big-data" are massively underestimated. Often people complain that data access is very slow, not realizing that they requested multiple Gigabytes or even Terabytes of data.
- **Data retention policy required.** Dealing with vast amounts of data, what is needed is a dedicated data owner who defines a retention and reduction policy. Without this, large amounts of money and resources will be wasted.
- **Be prepared.** Problems and issues of other systems will first be visible in the DAQ and buffering system. Therefore, always collect and keep statistics and metrics to help in pin-pointing the issue.
- **Documentation.** Keep documentation accessible, simple and in a copy/paste manner. This helps people especially when dealing with the large amount of data.
- **The design/implementation/support of the software ecosystem around the DAQ system is as important as the DAQ system itself.** Without the proper tools, libraries, infrastructure, support, etc., people will be lost and won't be able to work efficiently. A decent amount of time and resources are required to teach and support users to deal with the system.

CONCLUSION

The SwissFEL beam synchronous data acquisition system has been in production use for about one year.

While keeping the system up and running, we implemented various major changes to accomplish the tasks at hand and improve the performance and stability of the system.

Although not all data sources are in place we are confident that the current system will be stable, resilient and able to cope with the final load once all sources are up and scientific experiments are running.

ACKNOWLEDGEMENT

The SwissFEL beam synchronous data acquisition system is the result of an effort of many people, in addition to the present authors. We hereby gratefully acknowledge their valuable contributions to this project!

REFERENCES

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

[2] B. Kalantari and R.Biffiger, "SwissFEL timing system: first operational experience", presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper TUCPL04.

[3] A. Gobbo and S. Ebner, "PShell: from SLS beamlines to the SwissFEL control room", presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper TUSH102.

[4] ZeroMQ, http://zeromq.org

[5] Bitshuffle-lz4, https://github.com/kiyo-masui/bitshuffle

[6] Spring Boot, http://projects.spring.io/spring-boot/

[7] Netty, http://netty.io

[8] Hazelcast, https://hazelcast.org

[9] Jeromq, https://github.com/zeromq/jeromq

[10] Polymer, https://www.polymer-project.org

[11] Plot.ly, https://plot.ly