ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA167

EPICS DATA STREAMING AND HDF FILE WRITING FOR ESS BENCHMARKED USING THE VIRTUAL AMOR INSTRUMENT *

D. Werder†, M. Brambilla, M. Könnecke, Paul Scherrer Institut, Villigen, Switzerland
M. Jones, Tessella, Abingdon, United Kingdom
T. Richter, ESS ERIC, Lund, Sweden
A. Mukai, J. Nilsson, ESS ERIC, Copenhagen, Denmark
M. J. Clarke, F. A. Akeroyd, Science and Technology Facilities Council, Didcot, United Kingdom

Abstract

As a contribution to the European Spallation Source as part of BrightnESS, the Paul Scherrer Institut is involved in the streaming of EPICS data and the writing of NeXus compliant HDF5 files. We combine this development with the transition of the AMOR instrument at the Paul Scherrer Institut to EPICS and a streaming based data architecture. To guide our development before ESS has operational equipment, we use a detailed simulation of the instrument AMOR at SINQ to test and integrate our data streaming components. We convert EPICS data sources to Google FlatBuffers as our message format and distribute them using Apache Kafka. On the file writing side, we combine the messages from EPICS data sources as well as from neutron events to write HDF5 files at rates up to 4.8 GiB/s using Parallel HDF. This platform will also be used for testing the experiment control software on top of EPICS.

INTRODUCTION

The European Spallation Source (ESS) [1] will offer higher brightness and higher data rates than previous neutron sources. This demands a capable infrastructure to handle the produced data. As part of the BrightnESS [2] project, the Paul Scherrer Institut (PSI) [3] contributes to the development of the data streaming layer and to the writing of NeXus-compliant [4] HDF [5] files.

Instruments at ESS use among others the Experimental Physics and Industrial Control System (EPICS) [6] for control, to expose status information and to publish measured values. We incorporate EPICS data sources into the data streaming layer by converting EPICS data to a common serialization format and streaming them to a unified messaging layer.

Experimental data is also made available to the users in the form of NeXus-compliant HDF files. We present the ongoing development of the HDF File Writer which reads the selected data streams from the common messaging layer and creates the HDF file as configured by the user. The architecture is modular and extensible. General purpose writer modules are available which can handle the common types of data streams, while specialized writer modules can be easily added. This allows handling of new types of data streams or exploiting possible invariants of a data stream for more aggressive optimizations.

During development, we also use a simulation of the AMOR [7–9] instrument at PSI which contains a range of virtual devices modelled after the real instrument, and in addition some extensions specific to ESS.

In this conference note, we present the current status of the streaming of EPICS data to the messaging layer and the writing of HDF files.

STREAMING EPICS DATA SOURCES

Data Sources

EPICS is used at many scientific facilities around the world as a control system, including at particle accelerators and telescopes. At ESS, data coming from EPICS sources typically includes the sample environment, choppers and motion control.

Motion control and choppers contribute to the overall data acquisition with a rather low data rate, even though at ESS, the chopper top-dead-centre (TDC) events will be recorded and made available via the EPICS interface. Even though TDC events contribute a slightly higher rate, the events will be gathered in batches already at the EPICS interface so that the EPICS update rate is much lower than the actual TDC event rate. Furthermore, the EPICS update rate can be kept independent of the TDC event rate.

The sample environment provides information such as temperature, positioning and electromagnetic fields. While data rates from temperature sensors are very low, fast changing electromagnetic fields can produce a moderate data rate. Similar to the chopper TDC events, measured field values can be batched at the EPICS interface.

Messaging Layer

All generated data at ESS is represented uniformly as messages which get pushed into a queue of the intermediate messaging layer. The usage of a separate messaging layer improves decoupling between the individual components of the system by enforcing well defined common interfaces. It simplifies the overall design because components do not need separate interfaces to each other (n:m), but only to the common messaging layer (n:1). This allows for easier addition and replacement of individual components of the ESS data streaming architecture in the future. This also facilitates the scaling of the data distribution layer.

* This work is funded by the European Union Framework Programme for Research and Innovation Horizon 2020, under grant agreement 676548.
† dominik.werder@psi.ch

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA167

Apache Kafka [10] is used as the message broker for data streaming at ESS. It offers a configurable number of persistent commit logs, which represent the message queues. Producers can publish messages to their chosen queue, while consumers can monitor the queue of interest. Kafka offers tuneable scalability and redundancy.

An individual message queue (called "topic") can be split among several commit logs (called "partitions") which can run on the same or different physical machines. This allows distributing the load on a topic over multiple brokers at the cost of losing the relative ordering between messages in different partitions.

Redundancy is provided by the possibility to replicate each partition a configurable number of times. Given a replication factor n, a producer can demand $0..n$ acknowledgements depending on its safety requirements.

One distinguishing feature of Kafka compared to other messaging solutions is the persistence of the commit log. The messages are persisted to the file system for a configurable maximum time or until a maximum commit log size has been reached. This still allows for very high throughput because the file system access is mostly sequential. The persisted commit log allows consumers to optionally start reading from a point in the past, which allows executing a task again in case a component failed, or in the case of the HDF File Writer, to (re-)generate the HDF file for a previous experiment.

The official Kafka client is written in Java, but a set of third party open source libraries exist as well. One of those is `librdkafka` [11] which is a widely used implementation of the Kafka protocol in C/C++. It allows very high throughput, both in terms of number of messages and number of bytes, facilitated by a memory-efficient, zero-copy design.

Serialization Format

Interoperability demands that the type of each message on the messaging layer can be unambiguously identified by every component. This requires the usage of a common message format at least at the top level of all messages. The common message format should be light-weight, memory efficient, flexible, open and must allow for lossless representation of numeric data. It should not demand expensive conversions.

Based on these criteria, we use FlatBuffers [12] for the data streaming layer because it offers very low processing overhead, has good support for a variety of languages, is under active development and proven in existing projects. It is a statically typed format and all definitions of the common FlatBuffer message types ("schemas") are kept in a common repository [13].

FORWARD EPICS TO KAFKA

To stream EPICS data sources to the messaging layer, we develop `forward-epics-to-kafka` [14]. The application monitors a given list of EPICS variables for updates, converts the data received from EPICS to a FlatBuffer according to a configured conversion module, and publishes the FlatBuffer as a message on the Kafka messaging layer.

The monitoring of the EPICS variables is implemented on top of the EPICS v4 C++ API which allows us to monitor EPICS process variables via the more recent EPICS PV-access protocol. Internally, the EPICS v4 library also uses the older EPICS Base stack which allows us to also monitor EPICS variables via the older Channel Access Protocol using the same API.

The user configures the list of EPICS variables to be monitored via a configuration file (JSON) at program start, or via command messages at runtime. Command messages can also be used to stop the monitoring of EPICS variables as well as to change other options.

Status and log messages are published by the EPICS Forwarder in configurable intervals to a given Kafka topic. This allows consumers of the status topic, for example the Experiment Control Program, to present this information to the user, or to allow monitoring software to detect the proper operation of the application.

The EPICS Forwarder is designed with multi-core hardware in mind and can utilize a configurable number of cores on the machine.

Updates are taken from EPICS and enqueued in the internal conversion queue together with the necessary information on how the EPICS data should be converted to a FlatBuffer. A pool of worker threads processes these items in the queue and performs the conversion from the EPICS `PVStucture` object to a FlatBuffer according to the configured conversion module. After the conversion, the worker thread enqueues the FlatBuffer for transmission to the selected Kafka topic.

EPICS Forwarder Performance

Table 1: Values obtained on two machines connected by 10 Gb/s Ethernet.

Throughput	260 MB/s
Frequency	60 kHz

We benchmark the EPICS Forwarder on two machines connected with 10 Gb/s Ethernet and a single Kafka broker results in the values of Table 1. The obtained throughput coincides with the maximum write speed of the Kafka broker on that setup. We can forward the EPICS updates up to the maximum frequency to which the EPICS library itself can deliver them. The EPICS Forwarder is not a limiting factor in this setup. Further benchmarking is required on different hardware.

HDF FILE WRITER

Data streams from the Kafka messaging layer must be written to NeXus [4] compliant HDF [5] files. To that end, we develop `kafka-to-nexus` [15]. While the HDF library is very performant, the throughput on a single CPU core

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA167

will at some point be limited. There are different ways to improve the throughput further, of which we will mention "Direct Chunk Write" and "Parallel HDF".

Direct Chunk Write

Datasets in a HDF file can be chunked, which means that the data does not occupy a continuous range of the file, but instead is divided into a regular n-dimensional hypergrid. Each hypercube in this space is called a chunk. Chunked datasets are also the only way to work with extensible datasets in a HDF file.

Direct Chunk Write [16, 17] is an optimized way to write data to a chunk. This is based on the idea that a chunk does occupy a continuous range in the HDF file, and is therefore much simpler to access directly. Specifically, the HDF library does not have to find the set of chunks and their locations in the HDF file that an arbitrary read or write request would touch. Also, Direct Chunk Write allows to bypass optional format conversions and data compression which is useful if such conversions are already applied to the data before a write request.

There is though one major drawback of Direct Chunk Write, which is that it has to start on an edge of the chunk hypergrid. This is no strong constraint for applications where the size of each written message is fixed and known in advance, but it makes arbitrary write requests harder to implement especially if one strives for a minimal- or zero-copy architecture.

Parallel HDF

The parallel version of the HDF library is built on top of the Message Passing Interface (MPI) [18]. This allows an application to utilize multiple CPU cores while still being able to use the full HDF API and especially to issue arbitrary writes and reads, in contrast to Direct Chunk Write. Parallel HDF is therefore a very appealing option for the HDF File Writer.

We will show that the HDF File Writer can provide excellent throughput by using Parallel HDF, while still maintaining the full flexibility of the single-threaded version.

HDF File Writer Performance

The performance of the HDF File Writer is essentially limited by the available I/O. There are two main aspects to consider, which are the input from the Kafka messaging layer and the output to the file system where the HDF file should be written to.

As discussed previously for the EPICS Forwarder, the Kafka messaging layer allows us to scale the input bandwidth by routing data streams through different physical brokers.

On the other side, to be able to write HDF files at the expected high data rates, the file system layer has to be backed by an appropriately fast I/O layer. For our tests, we use GPFS (now IBM Spectrum Scale [19]) connected via Infiniband FDR as shown in Table 2.

We first look at the HDF write performance separately where we simulate a fast Kafka connection by pregenerating

Table 2: Specifications of the compute node that is used to benchmark the file output performance of the HDF File Writer.

Property	Value
CPU	`Intel Xeon E5-2690V4`
Cores	`14`
Threads	`28`
Clock	`2.60 GHz`
RAM	`256 GB`
Cache	`35 MB`
File system	`GPFS via 4x Infiniband FDR`

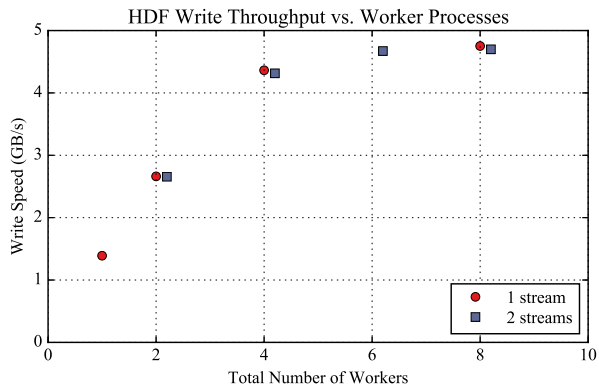

Figure 1: Throughput vs. Number of Worker Processes. The points for the 2-stream case are slightly displaced for readability.

the incoming Kafka messages in RAM and using these to feed the writer. Otherwise, the messages follow the normal code path. They are enqueued for the set of MPI worker processes which handle the data stream, and written to file by one of the worker processes.

Figure 1 shows that we can reach the maximum write performance of about 4.8 GiB/s on our test system with about 6 worker processes. The throughput was measured for the cases of 1 and 2 data streams being written to HDF. We see that the performance is comparable for these two cases. Further testing with an increased number of streams is necessary.

For chunked datasets, the size of the chunks is an essential tuning parameter. Too small chunk sizes require an unproportionally large overhead, while too large chunk sizes can thrash the chunk cache and cause excessive paging especially in applications where random reads are common.

We investigate how variations of the chunk size affect the overall throughput while we keep the number of worker processes w fixed at $w = 8$. The result is shown in Fig. 2 and demonstrates that the optimal throughput is achieved over a fairly wide range of chunk sizes. Still, the chunk size must not be too small. The throughput is shown for two different sizes of the individual messages and we see that the message size has a rather small effect on the overall throughput.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA167

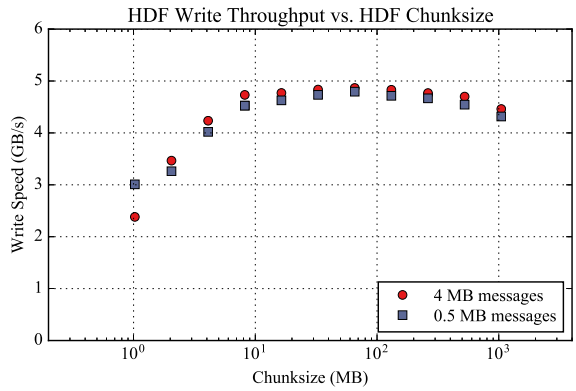

Figure 2: Throughput vs. Chunk Size. An almost optimal throughput can be achieved over a quite large range of chunk sizes.

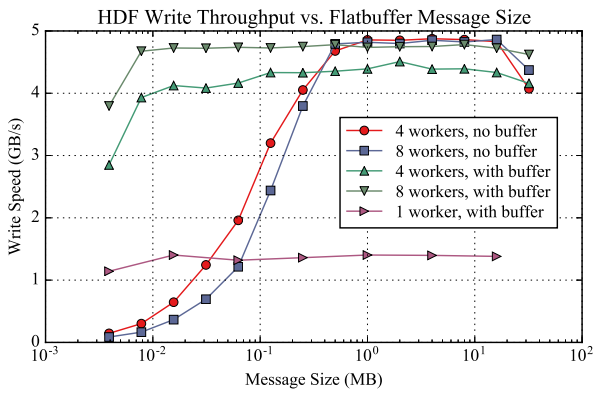

Figure 3: Throughput vs. Message Size. The connecting lines only serve to guide the eye.

Finally, in Fig. 3 we investigate how the throughput depends on the size of the individual messages being written. The first two sets of points write the messages directly to HDF. We observe how the throughput decreases for small messages. There, it pays off to make an additional copy and to buffer small messages so that we can write them in larger batches. We note though that we need a larger number of worker processes in the case of buffered writes to reach the maximum throughput. This is summarized in Table 3.

Another important aspect of the scalability of the system is the distribution of CPU load of the individual threads. Ideally, we would like the load to be distributed equally among the threads. We especially do not want a single thread, or a small group of threads, to consume a large fraction of CPU resources because these threads would likely cause a bottleneck first.

To investigate this behaviour in the HDF File Writer, we record the usage of each CPU core in intervals of 1 s. We then histogram all those usage values over the runtime of the program. If one or more threads are limited by their CPU usage, this would show in the histogram as a spike at 1.0.

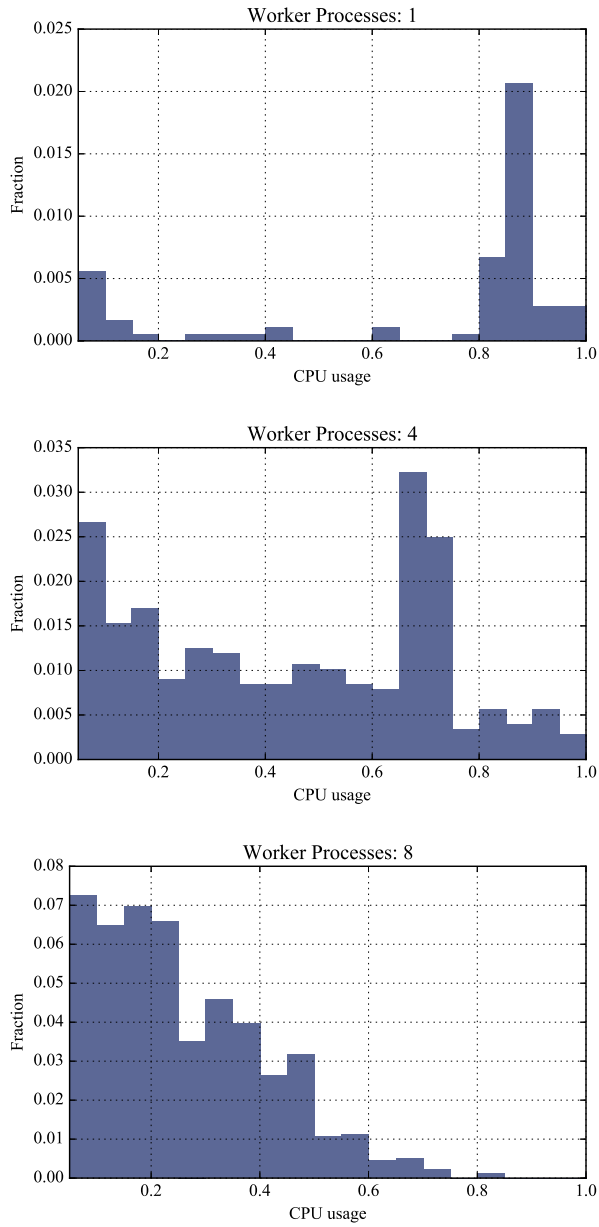

Figure 4: Histogrammed CPU usage over the runtime of the program and all CPU cores. We see how the increase in the number of workers reduces the load on individual cores. Most importantly, we demonstrate that no single thread is a bottleneck for the performance and that all threads put a comparable load on the CPU.

In Fig. 4 we show how the histogrammed CPU usage changes with the number of worker processes. We can see that given enough processes, the performance is indeed not limited by a single thread which gives further confidence in the scalability of the application.

SINQ AMOR SIMULATION

To guide our development before ESS receives operational equipment we use a simulation [20] of the instrument

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA167

Table 3: The effect of buffered I/O and number of worker threads on the total average CPU usage.

Buffered	Workers	Avg CPU (%)	Write (GiB/s)
no	4	8.6	4.86
no	8	9.3	4.82
yes	4	11.0	4.39
yes	8	13.1	4.74

AMOR [7–9] located at PSI. The simulation contains a neutron event generator which translates a histogram measured at AMOR into artificial neutron events, a simulation of a Dornier chopper, an EL734 [21] motor controller, a Dimetix laser distance sensor and several magnets.

Simulated devices which deliver a low data rate are implemented in Python based on the Twisted library, while the neutron event generator is written in C++ to be able to simulate high throughput.

The simulation of the Dornier chopper has in addition an EPICS interface which also simulates the TDC events which are not available at AMOR.

SOURCE CODE

All software is developed as open source and available on Github [22].

CONCLUSIONS AND OUTLOOK

We present prototypes of the software which is under development for the streaming of data and writing of HDF files at ESS. The conversion and streaming of EPICS variables is handled by the EPICS Forwarder [14] which converts EPICS data to FlatBuffers and sends them to the Kafka messaging layer in a highly configurable and extensible way.

The HDF File Writer [15] reads data streams from the Kafka messaging layer and writes NeXus compliant HDF files using Parallel HDF. We demonstrate a throughput of 4.8 GB/s which is close to the bandwidth of the file system on our test machine.

Further development of the HDF File Writer includes optional Direct Chunk Write in the case of buffered I/O to increase the efficiency even further and more features to improve usability.

ACKNOWLEDGEMENTS

We would like to thank Heiner Billich for the possibility to benchmark the HDF File Writer on one of the compute nodes of the Swiss Light Source with access to their GPFS.

REFERENCES

[1] European Spallation Source
https://europeanspallationsource.se

[2] BrightnESS
https://brightness.esss.se

[3] Paul Scherrer Institut
https://psi.ch

[4] NeXus file format
http://www.nexusformat.org

[5] HDF Group
https://www.hdfgroup.org

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

[7] D. Clemens *et al.*, "AMOR – the versatile reflectometer at SINQ", Physica B: Condensed Matter, 276, p.140-141 (2000).

[8] M. Gupta, T. Gutberlet, J. Stahn, P. Keller, D. Clemens, "AMOR - the time-of-flight neutron reflectometer at SINQ/PSI", Pramana, vol.63, p.57-63 (2004).

[9] J. Stahn and A. Glavic, "Focusing neutron reflectometry: Implementation and experience on the TOF-reflectometer Amor", Nucl. Inst. a. Meth. A, vol.821, p.44-54 (2016).

[10] Apache Kafka
https://kafka.apache.org

[11] librdkafka
https://github.com/edenhill/librdkafka

[12] Flatbuffers
https://google.github.io/flatbuffers

[13] Streaming Data Types Repository
https://github.com/ess-dmsc/streaming-data-types

[14] forward-epics-to-kafka
https://github.com/ess-dmsc/forward-epics-to-kafka

[15] kafka-to-nexus
https://github.com/ess-dmsc/kafka-to-nexus

[16] Direct Chunk Write Manual
https://support.hdfgroup.org/HDF5/doc/Advanced/DirectChunkWrite/UsingDirectChunkWrite.pdf

[17] Direct Chunk Write API
https://support.hdfgroup.org/HDF5/doc/HL/RM_HDF5Optimized.html

[18] Message Passing Interface
http://mpi-forum.org

[19] IBM Spectrum Scale (formerly GPFS)
https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage

[20] sinq-amorsim
https://github.com/ess-dmsc/sinq-amorsim

[21] Motorcontroller EL734
https://github.com/ess-dmsc/sinq-amorsim/blob/master/docs/motorcontroller.pdf

[22] Repositories ESS DMSC
https://github.com/ess-dmsc