ISBN: 978-3-95450-241-7 ISSN: 2673-5350 doi:10.18429/JACoW-IBIC2022-MOP40

SYNCHRONOUS DATA SERVICE AT THE EUROPEAN SPALLATION SOURCE

R. Titmarsh*, ISIS, STFC, Rutherford Appleton Laboratory, Oxfordshire, UK
J. F. Esteban Müller, J. P. S. Martins, European Spallation Source ERIC, Lund, Sweden

Abstract

The Synchronous Data Service (SDS) is a tool to monitor and capture events in the European Spallation Source, building on top of the EPICS control system. Large amounts of data from different input-output controllers are acquired and synchronised at the level of beam pulses. The acquisition can be triggered by beam events through the timing system or manually by a user. Captured data is stored in standardised NeXus files and indexed in a database for easy searching and retrieval.

INTRODUCTION

At ESS, the operation of the proton linac, target, and neutron experiments are controlled using the EPICS [1] control system. EPICS is a distributed control system, and that means that the data produced in different Input/Output Controllers (IOC) is received by clients in a non-deterministic order.

At our facility, EPICS' data is archived by the Archiver Appliance [2]. This tool stores each signal as a time series, so the effort of correlating data from different sources lies with the client side.

In some applications, it is important to obtain a snapshot of the machine status containing several signals from various sources that belong to the same beam pulse. Typical examples are machine tuning and optimization, or troubleshooting of failures (post-mortem).

In order to ensure that the data is synchronized in all relevant devices, a precise timing system capable to generate triggering signals is required. In this paper we describe both the hardware and software implementation under development at ESS that will allow pulse-synchronous data acquisitions and archival, which we refer to as the Synchronous Data Service (SDS).

SOFTWARE ARCHITECTURE

A diagram of the SDS architecture is shown in Fig. 1. The main design goals are scalability and flexibility, and for that reason we followed a microservices pattern.

The service consists of 3 different types of components or microservices: one or more collector services that collect data from EPICS IOCs and produce NeXus [3] files, an indexer service that aggregates metadata from the collectors, and one or more data retriever services that allow users to query for the data in different ways. These components and their prototype implementation are described in more detail below.

These microservices rely on a storage backend consisting of an Elasticsearch [4] database that stores the metadata and a CEPH-based [5] distributed file system that contains the NeXus files.

Collector Services

We developed a prototype version of a generic collector service. It is implemented in Python 3 using the asyncio framework. PV Access is supported through the p4p [6] package. Channel Access support was considered not necessary since our control system is fully EPICS7 compliant.

This service creates a set of "collectors" from definitions in a JSON file, each collector listens for a single type of event on a set of EPICS PVs. Each instance of this service has a "collector manager" that listens to the EPICS PVs of the collectors within it, parses the received messages and forwards them to the relevant collector. Multiple instances of the service can be deployed to distribute the load. The collector collects the events into "datasets" by the pulse ID of the triggering pulse. The information about the triggering event must be included by the IOC and it is discussed later.

Using asyncio allows the many collectors to run concurrently, allowing multiple datasets to be constructed at the same time. This means it can handle events arriving out of order or overlapping each other.

The dataset is a virtual HDF5 file that is expanded as events are received. When the dataset is complete (defined as a configurable timeout, default is 2 seconds), it is written to a remote file system for storage and its metadata is sent to the indexer service. Datasets can be submitted with an "expire by" timestamp after which the data can be removed from storage.

Custom collector services can be developed to support other protocols or for different use cases. For instance, a post-mortem event triggered after the machine trips can generate a huge amount of data. In that case some systems may be configured to act as collector services by storing data locally for later transferring it to the central storage and to the indexer service.

Indexer Service

This service receives the collector definitions and the datasets' metadata and stores them in a database to enable fast search and retrieval.

The prototype was developed in Python 3 using the FastAPI [7] framework. It provides a REST interface that can be consumed by the collector services.

The Elasticsearch database was chosen for its capability of handling large volumes of data and for enabling complex

* ross.titmarsh@stfc.ac.uk

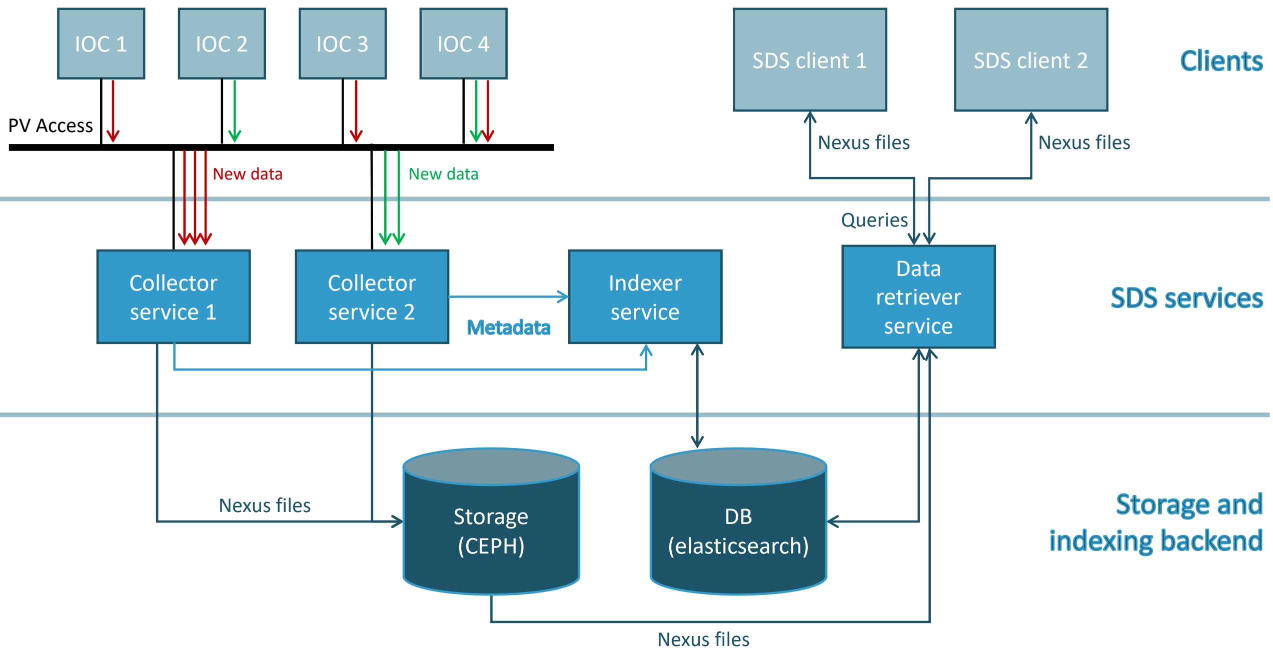

Figure 1: SDS architecture. The clients are shown on the top layer and can be either EPICS IOCs generating new data or SDS clients querying datasets and receiving NeXus files. In the middle layer it is shown the SDS services, including the collector services that collect data from the IOCs and interface with the indexer service that aggregates the metadata, and also the data retriever service that provides a REST interface for users to retrieve data. Finally, the bottom layer shows the storage and indexing backends.

queries with high performance. It is also simple to scale its performance by setting up a cluster.

Data Retriever Service

The data retriever service is the main entry point for users who want to access the data.

The technologies chosen for this prototype are the same as for the indexer service: Python 3 and the FastAPI framework. The retriever provides a REST interface for querying datasets and files. Datasets can be queried by collector, a date range, or a triggering pulse id range. Files can be retrieved using the same queries as for datasets, or directly by specifying a filename or dataset ID.

When a query result spans over several files or a subset of a file, the retriever service generates a new NeXus file that contains only the requested datasets.

The retriever service can also be scaled by deploying several instances and using an HTTP load balancer, e.g., Nginx or Apache.

DEPLOYMENT

A mirror of the code in our internal repository service, Gitlab, runs a continuous integration pipeline that generates a Docker image containing the SDS software. This simplifies the deployment to just pulling the Docker image and setting up a few environment variables.

TIMING SYSTEM

The timing system of the ESS linac is based on the MRF timing system [8]. The MRF system works with deterministic transmission of events from the Event Master (EVM) to the Event Receivers (EVR) via dedicated fibre-optics network. On the EVR side, the arrival of events is used to generate configurable electrical signals that can be used as acquisition or synchronization triggers on various devices. SDS will use dedicated events to trigger acquisitions, either periodically or on-demand. SDS events can be manually triggered by a user or generated after another event (e.g., machine trip).

The MRF timing network can also distribute data packets to all nodes, which are decoded at the EVRs and exported as EPICS PVs. At ESS, this data package is transmitted just before the next machine cycle and contains information about the beam that will be generated in the next round, such as beam mode, beam destination, beam length and the unique ID of the machine cycle. EPICS IOCs that are driven by events and triggers from the timing system can obtain precise timestamps and the pulse ID from the EVRs and add this information to the data structure of the PVs that contain measurement data. This ID is used by the SDS to correlate the data.

SDS can in principle be used together with any other timing system as long as it defines events and pulse IDs in the same way. In practice, one could also implement a custom collector that generates these metadata from other sources.

INPUT/OUTPUT CONTROLLERS

Some design considerations are required for IOCs to be compatible with SDS. The PVs that will be monitored by SDS need to be triggered by an event receiver (EVR) from the timing system. For systems that don't have a physical interface to EVRs, software triggers derived from the specific event counter can be used to synchronize acquisition (with reduced accuracy). For PVs with array data, the preferred structure is the `NTNDArray` format, in which the metadata containing the information about the timing event (event code and name) that triggered the SDS event can be appended to the structure as `NTAttribute` fields. For scalar PVs, a custom Normative Types structure will be used to encapsulate data and metadata. In both cases the pulse ID from the timing event that triggers the data acquisition needs to be copied into the `userTag` field of the PV and the timestamp corresponding to the SDS event should be copied to the `seconds` and `nanosseconds` fields.

In some use cases, the user might be interested in data from several pulses before and/or after the SDS triggering event. For that cases, we implemented a configurable circular buffer that continuously runs in the IOC. When an SDS event is received, it sends all the pulses stored in the buffer and then keeps sending the next few pulses as configured. Each PV update sent by the IOC will be tagged with the corresponding pulse ID, and all updates will share the same metadata.

NEXT STEPS

Now that we have a running prototype of the SDS software, we are planning to deploy it to a test environment for integrated testing with real hardware.

After that, a graphical user interface would be needed. At the moment we are exploring the use of PSI's Databuffer UI [9] as a simple user interface that aggregates data both from SDS and also from the Archiver Appliance instances running at ESS. It displays data from PVs as time series.

Later we would need to develop a dedicated UI that enables users to browse the collectors and datasets, and that more easily displays correlated data for a single pulse.

SUMMARY

The software architecture of the SDS system has been presented and described in detail. A first prototype of the system is ready and tested with emulated IOCs. The system shows high flexibility and the possibility of scaling up for better performance.

The code can be found at `https://github.com/EPICS-SDS/sds`.

ACKNOWLEDGEMENTS

The authors would like to thank Ricardo Fernandes for previous work on collecting system requirements, and together with Lars Johansson for information about databuffer-ui. We are also grateful to Timo Korhonen for his useful input on the system design. Thanks also to Bryan Jones for his support in making possible this collaboration between ISIS and ESS.

REFERENCES

[1] EPICS collaboration, `https://epics-controls.org/`

[2] M. Shankar, The EPICS Archiver Appliance documentation, `https://slacmshankar.github.io/epicsarchiver_docs`

[3] NeXus data format website, `http://www.nexusformat.org/`

[4] Elasticsearch website, `https://www.elastic.co/`

[5] CEPH website, `https://ceph.io/en/`

[6] M. Davidsaver, PVAccess for Python documentation, `https://mdavidsaver.github.io/p4p/`

[7] S. Ramírez, FastAPI documentation, `https://fastapi.tiangolo.com/`

[8] MRF website, `http://www.mrf.fi/`

[9] Databuffer-ui repository, `https://github.com/paulscherrerinstitute/databuffer-ui`