ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC002

CONCEPT AND DESIGN OF AN EXTENSIBLE MIDDLE-LAYER APPLICATION FRAMEWORK FOR ACCELERATOR OPERATIONS AND DEVELOPMENT*

M. Schütte[1†], A. Grünhagen[2], J. Georg, H. Schlarb,
Deutsches Elektronen-Synchrotron DESY, Germany
[1]also at Hamburg University of Technology, Hamburg, Germany
[2]also at HAW Hamburg, Hamburg, Germany

Abstract

Data collection and analysis are becoming increasingly vital not only for the experiments conducted with particle accelerators but also for their operation, maintenance, and development. Due to lack of feasible alternatives, experts regularly resort to writing task-specific scripts to perform actions such as (event triggered or temporary) data collection, system failure detection and recovery, and even simple high-level feedbacks. Often, these scripts are not shared and are deemed to have little reuse value, giving them a short lifetime and causing redundant work. We report on a modular Python framework for constructing middle-layer applications from a library of modules (parameterized functionality blocks) by writing a simple configuration file in a human-oriented format. This encourages the creation of maintainable and reusable modules while offering an increasingly powerful and flexible platform that has few requirements to the user. A core engine instantiates the modules according to the configuration file, collects the required data from the control system and distributes it to the individual module instances for processing. Additionally, a publisher-subscriber messaging system is provided for inter-module communication. We discuss architecture & design choices, current state and future goals of the framework as well as real use-case examples from the European XFEL.

INTRODUCTION

Control systems for particle accelerators and other large experimental physics facilities provide a common infrastructure for a vast and heterogeneous collection of subsystems which expose a considerable amount of diagnostic, monitoring and configuration data. For example, over nine million data channels are exposed in the Distributed Object-Oriented Control System (DOOCS) control system at European XFEL [1]. This data is becoming increasingly vital for ensuring reliable operation, cutting-edge system performance, and failure detection and prevention. This is shown by many recent projects working on data-driven methods for accelerator setup [2, 3], operation [4], anomaly detection [5] and predictive maintenance [6]. Such project rely on a solid data foundation which is as heterogeneous and complete as possible, as explained in [7], where a long term archive for the full state of an entire sub-system of the European XFEL is set up. Logging such vast amounts of data continuously over long time periods is however technically and financially challenging, and few subsystems today have such a data acquisition (DAQ) system readily available.

Problem Description

To circumvent this limitation, we frequently observe operators and researchers utilizing control system library bindings in high-level scripting languages such as MATLAB or Python for loop-based polling of interesting data channels. For shot-based facilities, this is usually extended with data synchronization logic. Past the collection stage, the obtained data is (pre)processed with reoccurring operations, such as bit field extraction, sorting, outlier removal, averaging, Fourier transform computation etc. and eventually plotted. In a final stage, this offline analysis may result in the development of an online algorithm to automate routine tasks, generate reports, recover from system failure states or implement basic feedbacks e.g. for performance optimization.

This ad hoc scripting, in contrast to dedicated control system application development, has the advantages of rapid prototyping workflows and can be done directly by the scientists and operators without having to rely on scarce software developer resources. However, the do-it-yourself approach has severe downsides as well. Information about such development efforts is usually just spread by word-of-mouth, not everyone profits immediately. Scripts are seldom committed to a central code repository, making them hard to find and maintain. Multiple versions circulate and when in doubt, new users often rather start from scratch than understand and fix an older code. This is also because ad hoc scripts rarely follow good coding practices and speed is favoured over readability. This is fatal not only because the same task is solved redundantly over and over again, but also each time bugs may be introduced that cost additional time to fix or, even worse, stay unnoticed and produce incorrect results [8].

Solution Proposal / Requirements

In order to keep and even extend on the rapid prototyping advantages of script based middle layer software, a guard rail system is required, providing immediate benefits to the operators and scientists using it and providing incentives

* The authors acknowledge support from DESY Hamburg, Germany, a member of the Helmholtz Association HGF.
† maximilian.schuette@desy.de

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC002

to share code, stick to good coding practices and maintain written code. We believe that such a guard rail system is best implemented as an easily usable and extensible software framework where new functionality can be added in terms of modular code components, maintained as a community project where power users maintain the core components and help improve contributions from the community.

In the following section, we present such a framework, based on the Python programming language, as it was conceived at the Machine Beam Controls (MSK) group at Deutsches Elektronen-Synchrotron (DESY) and implemented for (but not limited to) the European XFEL and FLASH machines running on DOOCS [9]. We detail core functionality, application configuration and the framework design. We then discuss the current status of the project and first experiences and use cases of the framework. Finally, we summarize our contribution and outline next steps for the proposed software framework. Throughout, we will refer to the general framework concept as Extensible Middle-Layer Application Framework (ExMAF) and our own implementation for DOOCS as DOOCS eXtensible Middle-Layer Application Framework (DxMAF) [10].

AN EXTENSIBLE MIDDLE-LAYER APPLICATION FRAMEWORK

Overview

The core design principles of the proposed ExMAF are maximization of reusability and flexibility. The former is achieved through generalization by clean separation of configuration from functionality, while the latter is achieved through modularization of functionality. Any application running on the framework is thus defined through a configuration file, specifying which modules to run and with which parameters. The framework's core engine is responsible for parsing the configuration file, instantiating the modules, collecting data and distributing it to the module instances and passing on signals between modules. To illustrate with an example, a simple trip event logger may be configured by combining a module that continuously compares an indicator data channel from the control system against a threshold with a ring file writer module, collecting additional monitoring channels and saving the current ring buffer to the disk upon receiving a signal from the threshold checking module. In the following, the module concept, the core engine functionality and the configuration syntax will be described in detail.

Modules & Messaging

In the context of ExMAF, a module is a marked Python class that is instantiated with a custom set of parameters. To do useful work after instantiation, the module class implements interfaces in the form of abstract base classes provided by the framework. Common examples of interfaces are `DataSubscriber` for automatic delivery of data from the control system as it arrives and `EventSubscriber` and `EventPublisher` for communication with other modules based on named event pipes supplied by the `PyPubSub` package [11]. Modules provide great flexibility by enabling users to replace general modules with highly specialized ones as needed, without introducing backwards-incompatible changes that would impact already existing applications.

Some care needs to be taken however when writing modules. As modules are sequenced linearly, long computations that are not off-loaded properly (e.g. using the `multiprocessing` package) can cause high latency and application stalling. Due to blocking processing of signals, deadlocks may occur. Finally, race conditions may occur due to unknown order of execution. In the author's opinion, the high responsibility of module programmers is outweighed by the low weight and flexibility of the framework.

For inspiration, we list a few module ideas, many of which have already been implemented or partially in DxMAF.

- `NpyFileWriter`: Efficiently dumps received data to NPY files on the disk along with a metadata file in JSON format. Handles out-of-order data and automatically splits files when reaching a maximum file size. Signals other modules once files have been closed or receives signals to close the current set of files. Useful for basic data collection.

- `NpyRingFileWriter`: Similar to `NpyFileWriter` but overwrites old data once a fixed size buffer has been filled. Buffer can be either on-disk or in-memory. Buffer seam position is stored in the adjacent metadata file. Useful for condition based data acquisition, e.g. trips.

- `FileMover`: Upon receiving a signal, transfers the files specified by the signal or matching a wildcard parameter to a destination storage. Useful for transferring data from control system nodes to a central storage such as dCache [12].

- `ThresholdChecker`: Compares a set of data channels against lower and upper threshold values. If any one channel breaks either threshold, a signal is emitted and latched until all channels are again within bounds. Useful for time-domain trip detection.

- `SpectrumChecker`: A specialized version of the `ThresholdChecker`, performing threshold checking on a power spectral density estimate of the data. Useful for frequency-domain event detection.

- `CommandRunner`: Executes a shell command upon receiving a signal. Command parameters may be taken from the signal context. Useful for utilizing external tools within the application.

- `EmailDispatcher`: Sends an email to specified recipients upon receiving a signal. Useful for notifying operators of events.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC002

Configuration

As mentioned in the beginning of this section, configuration in the context of this framework does not only mean providing parameters to the core and modules, but in its structure, the configuration file directly defines the application running on the framework. As such, it can vaguely be considered a domain-specific language (DSL), as it allows users without prior programming skills to construct new applications easily using a human-readable structured format.

Listing 1: Sample configuration file

```
1  # DxMAF configuration file
2  # Defines a simple trip event logger for
       ↪ high jitter events
3
4  extensions: ./dxmaf/extensions
5
6  duration: 14d
7  # stop_time: 2042-01-01T00:00:00
8
9  application:
10 - type: ThresholdChecker
11   channels:
12   - XFEL.SYNC/LASER.LOCK.XLO/XTIN.MLO1/
         ↪ CURRENT_INPUT_JITTER.RD
13   args:
14     lower_limit: -inf
15     upper_limit: 20
16     topics: high_jitter
17 - type: NpyRingFileWriter
18   channels:
19   - XFEL.SYNC/LASER.LOCK.XLO/XTIN.MLO1/
         ↪ CURRENT_INPUT_JITTER.RD
20   - XFEL.SYNC/LASER.LOCK.XLO/XTIN.MLO1/
         ↪ OXC_IN.SPEC
21   - XFEL.SYNC/LASER.LOCK.XLO/XTIN.MLO1/
         ↪ LOCK_STATUS.VALUE.RD
22   args:
23     output_dir: high_jitter_trip_%Y-%m-%
         ↪ d_%H%M%S
24     ring_file_size: 1024
25     memory_buffering: true
26   topics:
27   - high_jitter
```

Listing 1 shows an actual configuration file for the trip event logger example used before, written in YAML, the configuration language chosen in DxMAF. Walking through from the top, the configuration starts with a docstring comment, followed by a core setting, `extensions` in line 4, pointing the framework's engine to the module repository. Lines 6 and 7 show two ways to automatically terminate the application after or at a certain time. The `application` key in line 9 initiates the definition of the actual middle-layer service. Each mapping in the enumeration instantiates a module of the class identified by the `type` field. The `args` field accepts a mapping of argument-value pairs to be passed to the constructor of the module. Remaining siblings, in this example `channels` and `topics` are special arguments inherited from implemented interfaces, which require introspection from the engine. An argument `topics` on the top-level module definition indicates to the engine, that the module wishes to recieve messages published to the listed named channels. As an argument, 'topics' indicates named channels, to which the module will publish its own messages.

DxMAF uses strict schema validation provided by the `strictyaml` package [13] to ensure that typos in the configuration do not cause (potentially critical) unwanted behaviour. Subschema for the modules are automatically generated from the module class constructor signatures and the implemented interfaces. To perform proper type validation, the signatures must be annotated using Python type hints [14]. The `typing.Optional` qualifier and default arguments are also recognized, allowing the user to omit these keys in the configuration.

Core

The purpose of the framework's engine is two-fold. In a first stage, the engine bootstraps the application by parsing command line arguments, reading the configuration file, loading the module repository, generating a schema for the configuration file as described above and validating it, and then instantiating the requested modules with the user parameters. The second stage, the main loop, consists of polling data or processing pushed data from the control system as requested by modules and passing it on to each module, channel by channel. Delegating this task to the core minimizes control system calls and data transfer volume, as modules are agnostic and would have to query the same channel redundantly. In the example of listing 1, the channel `XFEL.SYNC/LASER.LOCK.XLO/XTIN.MLO1/` ↪ `CURRENT_INPUT_JITTER.RD` is requested by both modules, but only read once by the engine, then distributed to both modules.

FIRST EXPERIENCES

Since the first alpha release around spring 2020, DxMAF has been used for numerous studies (short- & long-term data acquisition) and is actively used to monitor accelerator systems for trips and to provide data snapshots for post-trip root-cause analysis. In [6], DxMAF was used to collect over 100 data channels from 25 low-level radio frequency (LLRF) stations in the European XFEL for fault analysis. In [15], DxMAF was used to periodically collect snapshots of high-resolution (several MB/s) ADC streams across multiple devices for predictive maintenance algorithm development. An ongoing study by the machine control systems group collects long-term phase and amplitude data from klystrons, approx. 0.25 MB/s for 12 channels, for anomaly detection. Following the earthquakes on February 6th and September 8th 2023 in Turkey and Morocco, respectively, DxMAF application instances have been deployed to collect

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC002

long-term data streams to search for seismic activity affecting the European XFEL. Likewise, DxMAF has been used several times to monitor the effect of regional events such as concerts or storms on the highly-sensitive optical synchronization system at European XFEL and FLASH. As a next step, a module for detecting certain signatures of seismic activity is under consideration.

Beyond these noteworthy application examples, DxMAF has been used numerous times for ad hoc measurements and studies, and we repeatedly received positive feedback from the users, specifically regarding the deployment speed and the resulting data quality and ease of access, compared to other available data collection methods. Even though applications occasionally refused to start up due to configuration errors (as intended) and sometimes due to bugs resulting from previously unconsidered use cases, we cannot recall a situation in which the application seemingly ran fine but produced unusable or erroneous data files.

We attribute the success of DxMAF to the reliability of the tried & tested code and the technical support provided by project developers to the users in setting up DxMAF applications. With each new use case, developers were able to spend time improving on and adding new modules and core functionality rather than writing the same base code over and over again, thus growing the feature set and by extension the user community.

CONCLUSION & OUTLOOK

In this paper, we have motivated and presented a general concept for an extensible middle-layer application framework and detailed central design choices for DOOCS specific implementation DxMAF. We showed that the modular design helped to reduce redundant work for users and increased the quality of the code and the resulting data. Clean separation of functionality from use case specific configuration in combination with a human-friendly configuration format ensures wide applicability and ease of use. We listed several projects where the framework has helped produce scientific results more quickly and reliably and reported on favourable user feedback.

For future work, several lines of development exist. The perhaps most ambitious goal is to transfer the implementation of DxMAF to other control systems, either by branching the project and maintaining separate cores for each target control system, or by making the control system interface modular as well, similar to [16]. Another high-value target is to conceptualize and implement data pipelining functionality for modules, likely by proving an appropriate interface for modules to implement. Low-hanging fruits are increasing user autonomy by improving on the project documentation and adding support for additional output data formats by writing extra modules or generalizing the existing `NpyFileWriter` module.

REFERENCES

[1] T. Wilksen *et al.*, "The Control System for the Linear Accelerator at the European XFEL: Status and First Experiences", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 1–5. doi:10.18429/JACoW-ICALEPCS2017-MOAPL01

[2] J. Kaiser, O. Stein, and A. Eichler, "Learning-based optimisation of particle accelerators under partial observability without real-world training", in *Proc. 39th Intl. Conf. Machine Learning (ICML'2022)*, 2022.

[3] J. Kaiser *et al.*, "Learning to do or learning while doing: Reinforcement learning and Bayesian optimisation for online continuous tuning", 2023. doi:10.48550/arXiv.2306.03739

[4] J. St. John *et al.*, "Real-time artificial intelligence for accelerator control: A study at the Fermilab booster", *Phys. Rev. Accel. Beams*, vol. 24, p. 104 601, 10 2021. doi:10.1103/PhysRevAccelBeams.24.104601

[5] A. Eichler, J. Branlard, and J. H. K. Timm, "Anomaly detection at the European X-ray Free Electron Laser using a parity-space-based method", *Phys. Rev. Accel. Beams*, vol. 26, no. 1, p. 012 801, 2023. doi:10.1103/PhysRevAccelBeams.26.012801

[6] A. Grünhagen, J. Branlard, A. Eichler, G. Martino, G. Fey, and M. Tropmann-Frick, "Fault analysis of the beam acceleration control system at the European XFEL using data mining", in *Proc. 30th IEEE Asian Test Symp.*, 2021, pp. 61–66.

[7] M. Schütte, A. Eichler, T. Lamb, V. Rybnikov, H. Schlarb, and T. Wilksen, "Subsystem Level Data Acquisition for the Optical Synchronization System at European XFEL", in *Proc. IPAC'21*, Campinas, Brazil, May 2021, pp. 2167–2169. doi:10.18429/JACoW-IPAC2021-TUPAB291

[8] Things you should never do, part I, `https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/`

[9] Distributed Object-Oriented Control System, DOOCS, `https://doocs.desy.de/`

[10] DxMAF, `https://gitlab.desy.de/xfel-facility/dxmaf`

[11] PyPubSub, `https://pypubsub.readthedocs.io`

[12] T. Mkrtchyan *et al.*, "dCache: Inter-disciplinary storage system", in *EPJ Web Conf.*, vol. 251, 2021.

[13] StrictYAML, `https://hitchdev.com/strictyaml/`

[14] PEP 484 – type hints, `https://peps.python.org/pep-0484/`

[15] A. Grünhagen, A. Eichler, M. Tropmann-Frick, and G. Fey, "Condition monitoring and fault detection of a laser oscillator feedback system", in *Proc. 33rd Intl. Conf. Info. Model. Know. Bases EJC*, Maribor, SI, 2023, pp. 61–66.

[16] G. Varghese *et al.*, "ChimeraTK - a software tool kit for control applications", in *Proc. 8th Int. Part. Accel. Conf.*, 2017.