ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOMS016

A PIPELINE FOR ORCHESTRATING MACHINE LEARNING AND CONTROLS APPLICATIONS

I. Agapov*, M. Böse, L. Malina
Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany

Abstract

Machine learning and artificial intelligence are becoming widespread paradigms in control of complex processes. Operation of accelerator facilities is not an exception, with a number of advances having happened over the last years. In the domain of intelligent control of accelerator facilities, the research has mostly been focused on feasibility demonstration of ML-based agents, or application of ML-based agents to a well-defined problem such as parameter tuning. The main challenge on the way to a more holistic AI-based operation, in our opinion, is of engineering nature and is related to the need for significant reduction of the amount of human intervention. The areas where such intervention is still significant are: training and tuning of ML models; scheduling and orchestrating of multiple intelligent agents; data stream handling; configuration management; and software testing and verification requiring advanced simulation environment. We have developed a software framework which attempts to address all these issues. The design and implementation of this system will be presented, together with application examples for the PETRA III storage ring.

RATIONALE

One of the promises of AI technology for research facilities is the increased automation of operation. A significant progress has been achieved in recent years in understanding the potential of AI and ML for accelerator operation. A usual common feature of the developed ML solutions is the increased complexity of the software and the algorithms: so, a Neural Network (NN)-based controls approach usually requires a more complex software stack, model storage, model tuning and re-training. In our previous work [1] we realized that being a more powerful representation in control problems, NNs can be prohibitively complex compared to more simple linear control approaches, as the results have to be constantly interpreted and evaluated, and the software stack common in data science becomes unsatisfactory for robust on-line applications. Significant human intervention was always required, making the operation not more but less autonomous. Based on this experience we started developing a software framework that would help address these issues. The framework allows execution and communication of services, each responsible for a certain subset of tasks necessary for intelligent control and operation.

Services with well-defined functionality and interfaces. All operations are performed by *services*. A *service* performs well-defined action such as orbit correction, retraining a model etc. Each *service* has a well-defined API, with standard operations being starting, stopping, and re-configuring. An arbitrary number of *services* can be running, any functionality can be implemented as soon as the API is adhered to, and automatic *service* discovery is allowed.

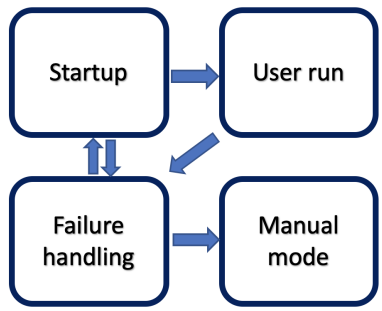

Figure 1: Basic states for autonomous operation.

Distributed deployment and communication bus. All *services* have access to a common communication bus, and are able to send and receive messages. A message header contains information such as the addressed service (or broadcast) and the body contains an arbitrary set of instructions and data in the form of a dictionary. *Services* can run anywhere on the network, and the system can be transparently scaled up by deploying certain *services* on an HPC cluster.

Decentralized architecture While the software does not impose any constraints on the kind of *services* that are being run, the design is geared towards the needs of autonomous operation, based on the following paradigm (see Fig. 1): there are a number of *services* related to machine startup, that can include: health checks of various subsystems, first-turn steering and trajectory correction, orbit correction, optic correction, accumulation (top-up). A supervision *service* monitors the machine startup and decides if the startup has been accomplished successfully, in which case transition into the "use run" mode can be performed, otherwise failure handling is activated, which can either attempt to resolve the issue autonomously or transition into manual mode. In the user mode a number of monitoring and slow correction *services* are active. A user run can either end normally or in a failure. A failure resolution can be attempted in an autonomous mode, or manual control can be initiated. Failure handling is one of the most important aspects of autonomous operation, with research activities ongoing.

Digital twin: simulation mode. The software follows the methodology of OCELOT [2,3] where the so-called *MachineAdaptors* are used to encapsulate the control systems

* ilya.agapov@desy.de

ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOMS016

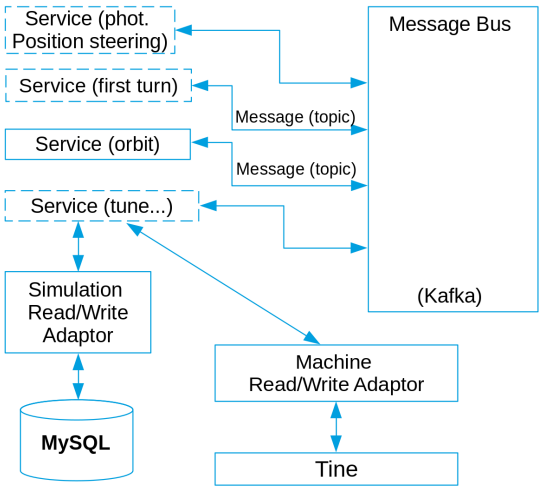

Figure 2: Schematic of the proposed software framework.

specifics, and keep most of the code control system agnostic, which ensures easy portability between facilities. Here we make one step further and introduce a *digital twin*. It is implementing as a database of machine parameters (such as magnet settings, the BPM readings etc.) and set of services that can either update this database on a regular basis or be triggered by a certain parameter change (e.g. closed orbit is recalculated when a corrector setting changes). Complex dynamics models (such as the ground motion) can be implemented.

The concept is schematically represented in Figure 2.

IMPLEMENTATION DETAILS

acss is implemented in a microservice architecture in which each service is running in an isolated process in a Docker container [4]. The Docker containers are orchestrated with Docker-Compose [5]. Services in *acss* are communicating with each other by publishing and subscribing to events over Kafka [6].

acss has core services which are needed for the basic functionality, shown in Fig. 3. The fist one is the register service which is responsible for health checks and to ensure that the names of every service is unique by holding a list of all registered services.

The second core service is the observer service which is responsible for storing processed messages in a key-value store and provides a Rest API for checking if a message is processed. This is needed to interact with *acss* by interactive computing platforms like Jupyter because *acss* internal communication is asynchronous.

The last core service is the machine service which is the switch between simulation or real machine. This switching is realized by changing the read/write method of the machine service at the initialization phase. In simulation mode the machine service is reading an writing from a database which is reflecting the simulation. In production mode the machine service is reading directly form the accelerator specific control system. The machine service is also responsible

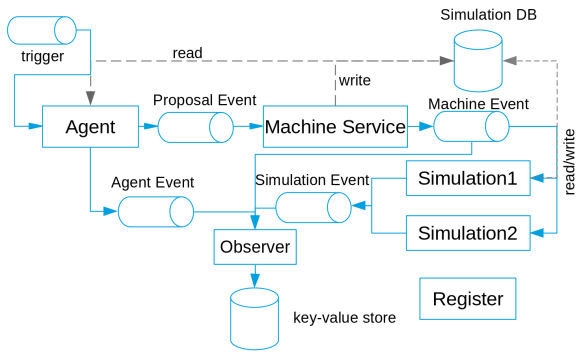

Figure 3: Workflow framework to implement services.

for processing write commands which are published by the proposal event. After each writing the machine service publishes what has changed on the machine event topic.

Beside core services there are user services which are programmed by the user. The user have to inherit from a service base class and overload the defined abstract member functions. In the current implementation there are two service base classes. The first one is simulation service which is for writing a simulation for a specific physical effect. Each simulation service is subscribed to the machine event and will be triggered when a new machine event is published. The second is the agent service which is foreseen for implementing algorithms which are making proposals based on machine observations. This proposals will be published on the proposal event topic.

USER STORIES / USE-CASES

In the following we highlight some user stories/use-cases.

- Starting and stopping services via web interface:
  - Starting agents/services: only agents that are in the repository can be started
  - All running agents/services and simulations should be seen on the web interface
- Writing own agents/services, simulations and adapter by users:
  - *For Simulations:*
    - Multiple simulations can run at the time
  - *For agents/services:*
    - Multiple agents/services can run at the time
    - The user can reconfigure parameters while the agents/service is running
    - An agent/servics can be triggered by other agents/services
    - An agent/service can set and get parameters for the simulation and for the real machine
    - The user can write machine parameter to the real machine/simulation and wait until the parameter are written or fire and forget

ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOMS016

    - The user can write multiple write commands at once to the real machine/simulation
    - An agent/service can wait for one or multiple simulations to be finished
    - The agents/services should communicate with each other over a standardised protocol
- The user can switch easily between simulation and real machine
- The user can start, stop, list, and trigger agents via *Jupyter* notebook
    - The user can wait for agents to be finished
- A user should have his own session with his own resources (e.g. agents, simulations, etc.) which is decoupled from other user sessions in order to not interfere other users

BEAM TESTS

The pipeline was tested extensively at the PETRA III storage ring at DESY using the orbit correction agent as an example. The orbit correction using the SVD approach is standard, and is easy to implement and cross-check with existing software. The goal of the tests was to understand the modalities and interfaces when the orbit correction is triggered in a way that does not allow for direct interaction with a user. A pipeline featuring an orbit correction service was running in the controls network, and the service was triggered through a *Jupyter notebook* script. The service was tested in advance in simulation model on the *digital twin*. Main lessons lessomns learned were as follows:

i. Re-configuring the algorithm (e.g. excluding malfunctioning BPMs or orbit correctors, adjusting SVD cutoffs etc.) is often required and implementing appropriate APIs for on the flight service reconfiguration was necessary.

ii. Judging the success, number of iterations, and stopping criteria is easy for an expert by often difficult to implement algorithmically, and it is very likely that in the initial phases of an autonomous system deployment the exit to the manual mode is to be expected often.

iii. Generally, paradigm for testing is of paramount importance. While the "physics" algorithms can be tested on the *digital twin*, a lot of errors are happening in the hardware communication layer, and introducing another communication layer makes testing the system non-trivial. Based on this experience, logging capabilities were improved substantially.

SUMMARY AND OUTLOOK

The current implementation of the *acss* pipeline core is available under [7]. Some service examples for PETRA III can be found under [8]. A number of algorithms is available as *python* code but not yet as an *acss* service for PETRA III. Table 1 lists the status of service implementation. Note that failure handling capabilities are not yet worked out in detail and require special attention beyond pure software engineering. Beyond storage ring functionality, there are plans to implement sets of *acss* services for linear accelerator operation.

Table 1: Status of Service Implementation for PETRA III

Service	Status
Orbit/optics sim., static	✓
Orbit/optics sim., with ground motion	✗
First-turn threading	✗
Orbit correction	✓
Orbit measurement AC dipole	✓
Optics measurement ORM	✗
Undulator gap compensation calculation	✗
TM-PNN [1] retraining	✗
Failure handling	Needs R&D

ACKNOWLEDGEMENTS

This work received funding from the Helmholtz International Labs project HIR^{3}X.

REFERENCES

[1] A. Ivanov and I. Agapov, *Phys. Rev. Accel. Beams*, vol. 23, no.7, p. 074601, 2020. doi:10.1103/PhysRevAccelBeams.23.074601

[2] I. Agapov, G. Geloni, S. Tomin, and I. Zagorodnov, "OCELOT: A software framework for synchrotron light source and FEL studies," *Nucl. Instrum. Meth. A*, vol. 768, pp. 151-156, 2014. doi:10.1016/j.nima.2014.09.057

[3] S. Tomin, I. Agapov, W. Decking, G. Geloni, M. Scholz, and I. Zagorodnov, "On-line optimization of European XFEL with OCELOT," *Proc. of ICALEPCS'17*, Barcelona, Spain, 2017. doi:10.18429/JACoW-ICALEPCS2017-WEAPL07

[4] docker: https://docs.docker.com/

[5] docker-compose: https://docs.docker.com/compose/

[6] kafka: https://kafka.apache.org/

[7] Core framework: https://github.com/desy-ml/acss-core.git

[8] Services: https://github.com/desy-ml/acss-services.git