



Full Length Article

Constellation: The autonomous control and data acquisition system for dynamic experimental setups

Simon Spannagel ^a^{*}, Stephan Lachnit ^a, Hanno Perrey ^b, Justus Braach ^c,
 Lene Kristian Bryngemark ^d, Erika Garutti ^e, Adrian Herkert ^a, Finn King ^a,
 Christoph Krieger ^e, David Leppla-Weber ^a, Linus Ros ^b, Sara Ruiz Daza ^a,
 Murtaza Safdari ^f, Luis G. Sarmiento ^d, Annika Vauth ^e, Håkan Wennlöf ^g

^a Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany

^b Prevas Test & Measurement AB, Fabriksgatan 2C, 222 35 Lund, Sweden

^c European Organisation for Nuclear Research (CERN), Geneva, Switzerland

^d Lund University, Department of Physics, Box 118, 22100 Lund, Sweden

^e Universität Hamburg, Institut für Experimentalphysik, Luruper Chaussee 149, 22761 Hamburg, Germany

^f Fermi National Accelerator Laboratory, Wilson St & Kirk Rd, Batavia, 60510, USA

^g Nikhef, Science Park 105, Amsterdam, 1098XG, The Netherlands

ARTICLE INFO

Keywords:

Data acquisition
 Experiment control
 Finite State Machine
 Decentralized
 Network
 Autonomy

ABSTRACT

The operation of instruments and detectors in laboratory or beamline environments presents a complex challenge, requiring stable operation of multiple concurrent devices, often controlled by separate hardware and software solutions. These environments frequently undergo modifications, such as the inclusion of different auxiliary devices depending on the experiment or facility, adding further complexity. The successful management of such dynamic configurations demands a flexible and robust system capable of controlling data acquisition, monitoring experimental setups, enabling seamless reconfiguration, and integrating new devices with limited effort.

This paper presents Constellation, a flexible and network-distributed control and data acquisition software framework tailored to laboratory and beamline environments, that addresses the limitations of existing solutions. The framework is designed with a focus on extensibility, providing a streamlined interface for instrument integration. It supports efficient system setup via network discovery mechanisms, promotes stability through autonomous operational features, and provides comprehensive documentation and supporting tools for operators and application developers such as controllers and logging interfaces.

At the core of the architectural design is the autonomy of the individual components, called *satellites*, which can make independent decisions about their operation and communicate these decisions to other components. This paper introduces the design principles and framework architecture of Constellation, presents the available graphical user interfaces, shares insights from initial successful deployments, and provides an outlook on future developments and applications.

1. Introduction

The coordinated operation of instruments such as detectors, motorized stages, or power supplies is crucial to modern scientific experiments in order to produce reliable and reproducible results. The individual components require configuration, synchronization and monitoring throughout the experiment, the recorded data needs to be transmitted to central storage nodes, and dependencies between instruments have to be resolved. These tasks are realized by control and data acquisition software frameworks.

Large research facilities and permanent experimental installations frequently develop specialized solutions based on established large-scale frameworks [1–4]. These systems are designed to scale efficiently and to provide excellent long-term stability. However, their complexity often necessitates significant engineering effort for the integration of new instruments and for the operation of the system.

This approach is commonly unattainable for smaller experiments, such as laboratory or beamline experimental setups, test beam campaigns, or detector prototype studies. These projects require more

* Corresponding author.

E-mail address: simon.spannagel@desy.de (S. Spannagel).

flexible and lightweight solutions that can be adapted to changing experimental conditions quickly and by the instrument experts themselves. Over the years, several control and data acquisition frameworks focused on small-scale experiments have been developed [5–9] and remain in use today. They are, however, often tailored to the specific needs of individual projects or require further application-specific adaptations and frequently rely on custom communication layers. The lack of comprehensive documentation and standardized interfaces further complicates reuse beyond the original scope of the framework.

Constellation is a flexible, network-distributed control and data acquisition framework that provides a reliable, maintainable, and easy-to-use system that eases the implementation of new instruments. It is built atop established open-source libraries for network communication (ZeroMQ [10]) and data serialization (MessagePack [11]), which provide a robust foundation and significantly reduce the required development effort. Unlike similar frameworks, Constellation does not rely on a central control server but is designed as a decentralized network whose components operate autonomously. The well-documented and clear interface for instrument control enables rapid integration of new devices and allows scientists to connect new instruments with minimal added effort and with the choice between an implementation in C++ or Python. The framework is designed with flexibility in mind and targets applications ranging from laboratory test stands up to small and mid-sized experiments with several dozens of connected instruments and large data volumes.

This paper provides an overview of the Constellation framework and its components and is structured as follows: Section 2 introduces the objectives and methodology of the framework development, while Section 3 provides an overview of the framework architecture. The functionality of its components are detailed in Sections 4, 5 and 6, while Section 7 showcases selected first applications of the framework. A brief overview of the extensive documentation is given in Section 8 before the concluding remarks and outlook in Section 9.

2. Objectives & Design Process

The development of control and data acquisition systems is often initiated in response to specific and immediate needs related to instrument control and monitoring, data storage, and network communication. Consequently, either existing solutions are adapted to these requirements or new ones are developed, often without the opportunity to establish a clear system architecture. This approach often limits the reusability of the solution in other settings or environments.

The work leading to the development of Constellation was centered on the idea of turning this paradigm on its head. Rather than being driven by an urgent and narrowly defined use case, a community of interested developers from diverse backgrounds collaborated within the *Exchange on & Development of Data Acquisitions (EDDA)* initiative [12] to design a robust and versatile framework applicable to a wide range of future use cases.

Several concepts have contributed to the formation of the framework structure and vision, which are outlined below.

2.1. Starting From User Needs

A preparatory phase was aimed at identifying and engaging a potential user community and forming a common vision of the project objectives and ambitions, primarily targeting colleagues at various research institutes. Feature requests, experiences and use cases were systematically collected in the form of *user stories*, which comprised brief descriptions of typical data acquisition scenarios in laboratory-scale experiments. An illustrative example for such a user story is the following:

I have a temperature-controlled detector with high-voltage biasing applied [...]. If something goes wrong with the temperature control, I want the high-voltage power supply to ramp down automatically and in a controlled manner.

These accounts were subsequently distilled into a set of common requirements, forming a *wishlist* of desired system functionalities. The following requirements could be extracted from the example:

- An alarm system to notify of anomalies.
- Capability of constituents to automatically react to notifications.
- Possibility for constituents to transition into a safe operation mode.

Similarly, other requirements were extracted from all collected user stories and compiled as an input for the discussions during collaboration meetings.

2.2. The Hackathon Format

Several EDDA collaboration meetings have been held in the form of so-called *hackathons*. Over dedicated periods of time, several developers met to collaborate in a shared workspace. Here, sufficient preparation was key, both in terms of shaping of a common vision for the framework, and concerning the technical groundwork such as review of available libraries for network communication. The discussed concepts were successfully tested in-situ using toy implementations.

The initial day of each hackathon was dedicated to discussing the objectives and their implications for the system architecture. Throughout, discussions on the principles underlying the framework reemerged, for which sufficient time was always made available.

It is envisioned to perpetuate an adapted version of the hackathon principle as EDDA collaboration meetings to ensure a continuous development of Constellation and to serve as a forum for its user community.

2.3. Working with RFCs

Communication protocols are at the heart of every network-distributed software framework, and interoperability relies on precise descriptions thereof. Therefore, a design-driven approach was adopted for the software development through the use of Request for Comments (RFC)-style protocol documents [13]. For every communication protocol proposed for the framework, an RFC document was drafted, detailing the exact behavior of the sending and receiving nodes. This methodology facilitated parallel and independent implementation efforts across the two supported programming languages C++ and Python, and ensured consistency and interoperability across the system.

The RFC documents are part of the framework documentation which will be discussed in Section 8.

2.4. Continuous Integration & Deployment

Constellation follows a sustainable development model which includes rigorous testing and an extensive Continuous Integration (CI) pipeline. The release pipeline, shown in Fig. 1, commences with the *build* stage which includes builds for all supported operating systems and Python versions as well as additional jobs testing different compiler and linker combinations and sanitizer libraries such as *Thread Sanitizer* [14].

In the *testing* stage, all components of the Constellation core libraries in both implementations are tested through unit tests with *Catch2* [15] for C++ and *pytest* [16] for Python. A combined test coverage of over 85% for both core libraries is achieved. Static code analysis as well as formatting checks are applied in the *lint* and *format* stages, and the documentation is checked for spelling mistakes and built as HTML and PDF documents in the *docs* stage.

The Continuous Deployment (CD) comprises an automatic deployment of releases to *Python Package Index PyPi* [17], on *Flatpak* [18] and as release tarball on the GitLab repository [19]. Furthermore, the website [20] is updated automatically with the latest documentation upon every release. A distribution through Homebrew for macOS installations is currently in preparation.

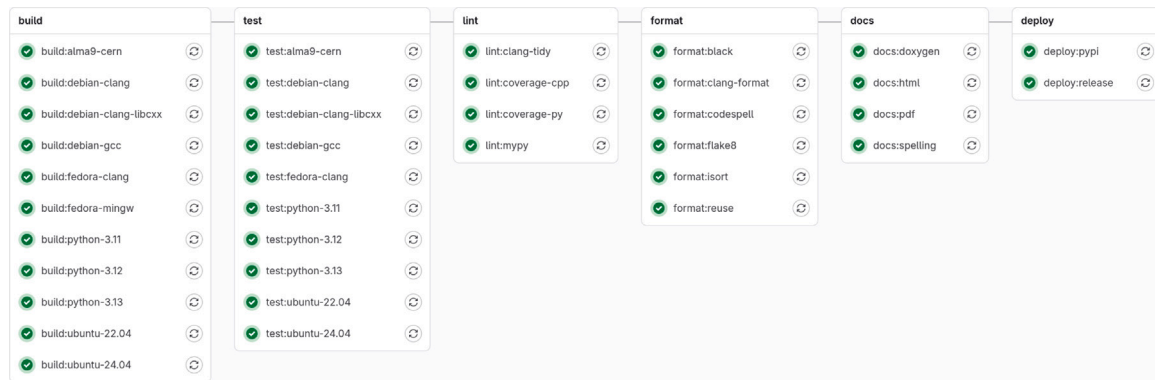


Fig. 1. Screenshot of the Constellation GitLab Continuous Integration (CI)/CD pipeline for a release version. Building, formatting and documentation generation run in parallel, while testing, linting and deployment depend on a successful build.

3. Framework Architecture

The name *Constellation* is borrowed from space flight, where satellite constellations operate with relatively large autonomy, transmitting data between them, and communicating with a ground control center which issues commands and controls the entire system. Consequently, the individual components of the Constellation control and data acquisition framework are dubbed *satellites* and are complemented by *controllers* and *listeners* which represent user interfaces to the system.

Multiple Constellation setups can operate in parallel on the same network without interference. A *group* name is provided to each component upon startup, and communication is only possible between components within the same group. In this paper, the term *Constellation* either refers to the name of the software framework, or to a set of satellites operating within the same network and group.

The development repository Constellation is hosted at the DESY GitLab instance [19] and a mirror is offered on GitHub [21]. Constellation is published under the European Union Public Licence, version 1.2 (EURL-1.2) [22]. Releases are named after International Astronomical Union (IAU) designated star constellations ranked by the solid angle that they occupy in the sky. Currently, two independent implementations of the communication protocols and the satellite code exist, written in C++ 20 and Python 3.11+.

Releases are automatically published to Zenodo [23] as software references. This paper describes the current version *Constellation 1.0 (Corona Australis)*.

3.1. Components of a Constellation

The Constellation framework knows three different types of components, namely *satellites*, *controllers* and *listeners*. Each of them has a different purpose and partakes in different communications.

Satellites. These are the main constituents of a Constellation. They implement the code controlling an attached instrument, realizing data receivers, and any other component that should follow the Constellation operation synchronously. At the core of a satellite is its finite state machine (FSM) which governs the state of the satellite and the attached instrument. A detailed description of the satellite is provided in Section 4.

Controllers. This component represents the main user interface to a Constellation. Controllers can send commands to satellites via the control protocol, parse and interpret configuration files, and display the current state of the entire system. Graphical or command-line user interfaces typically are implemented as controller-type components, providing the possibility for both direct human interaction and scripted procedures.

Controllers are stateless, i.e. they are not a satellite of the Constellation. The main advantage of this approach is that multiple control interfaces can be active simultaneously, and that they can be closed and reopened by the operator, or even crash, without affecting the operation of the Constellation.

The currently provided controllers are described in Section 6.1 along with their features.

Listeners. As the name suggests, components of this type only listen to communications of other components, typically via the monitoring protocol, and are entirely passive otherwise. Consequently, listeners are stateless and the Constellation is not affected by them appearing or disappearing during operations.

A typical example for a listener component is a log message interface which subscribes to logging information from satellites in the Constellation and displays them to the operator. Currently available listener components are presented in Sections 6.2 and 6.3.

3.2. System Architecture & Protocols

Constellation is built around a set of communication protocols defining the exchange of messages among its nodes. These protocols have been defined early on as described in Section 2.3 and serve as platform- and implementation-independent architecture of the framework. This means that new implementations of e.g. a satellite can be written in any language. The communication channels are independent of each other and follow clear communication patterns such as *publish/subscribe* for one-to-many distribution of information or *request/reply* for a client-server-based communication. Most of the protocols are TCP/IP communication based on the ZeroMQ messaging library [10] and build upon the ZeroMQ Message Transport Protocol (ZMTP). The messages of these protocols are encoded using MsgPack [11].

All Constellation communication is handled exclusively via ephemeral ports as defined in RFC 6335 [24], and consequently no privileged user account is required for running a node, unless the controlled hardware requires so. Limitations arising from this architecture will be discussed in Section 3.3.

The five Constellation communication protocols are described in the remainder of this section.

3.2.1. Network Discovery

A common nuisance in volatile networking environments with devices appearing and disappearing is the discovery of available devices and services. While some established implementations based on multicast DNS [25] exist for the purpose of finding services on a local network, such as zeroconf [26] or avahi [27], these come with significant downsides such as missing standard implementations, being limited to individual platforms, or a large and complex set of features not required for the purpose of Constellation.

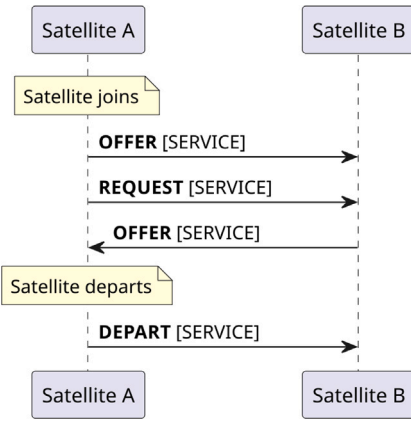


Fig. 2. Sequence diagram for Constellation Host Identification & Reconnaissance Protocol (CHIRP) showing the message flow between an already running satellite (B) and a newly started satellite (A). Satellite A offers a service and requests a service from others. Satellite B answers the request, and finally Satellite A departs again.

Other solutions, such as DIM [28], require central name servers to collate service information and distribute it to subscribers, a concept that conflicts with the notion of Constellation as a decentralized network.

Hence, the *Constellation Host Identification & Reconnaissance Protocol (CHIRP)* has been devised [29]. It is an IPv4 protocol intended to be used on local networks only which uses a set of defined beacons sent as multicast messages over UDP/IP to announce or request services. These services represent the communication endpoints of the other Constellation communication protocols. The beacon message contains a unique identifier for the node and its Constellation group, the relevant service as well as IP address and port of the service. Three such beacons exist:

- OFFER: A beacon of this type indicates that the sending node is offering the service at the provided endpoint.
- REQUEST: This beacon solicits offers for the respective service from other nodes.
- DEPART: A departing beacon is sent when a node ceases to offer the respective service.

Each service offered by the participating Constellation node is registered with its CHIRP service. Upon startup of the program, an OFFER beacon is sent for each of the registered services as indicated in Fig. 2. The REQUEST beacon allows nodes to join late, i.e. after the initial OFFER beacons have been distributed. This means that at any time of the framework operation, new nodes can join and request information on a particular service from the already running Constellation participants. When services are shut down, the DEPART beacon will prompt other nodes to disconnect.

All subsequently described protocols use CHIRP in order to find peers for their respective services.

3.2.2. Heartbeat Exchange

Autonomous operation of the Constellation requires the constant exchange of state information between all participants. This is implemented in the form of heartbeat messages sent via the *Constellation Heartbeat Protocol (CHP)*. Heartbeat messages contain the current state of the sender FSM and the time interval after which the next heartbeat is to be expected, alongside a set of flags defining how other satellites shall react e.g. to changes of the FSM state.

With this information, heartbeat receivers can independently deduce the state of other nodes, or identify non-responsive nodes in case

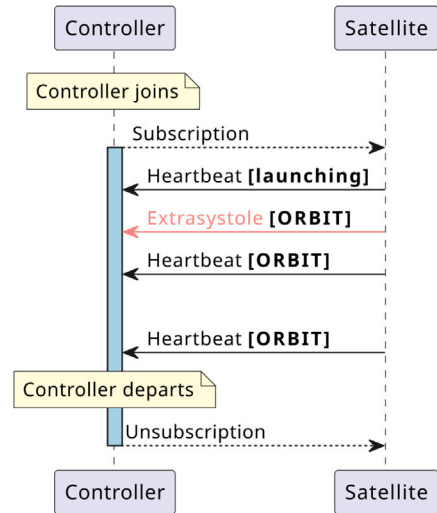


Fig. 3. Constellation Heartbeat Protocol (CHP) sequence diagram between a controller and a satellite indicating regular heartbeats as well as extrasystole messages for state changes. Here, the satellite is performing a transition and issues an extrasystole once the transition is completed. The vertical colored activation bar indicates the time period of heartbeat monitoring, the bold text in square brackets denotes the transmitted state.

of network or machine failure. The variable sender-defined intervals between messages enable packet congestion control by implementing a flexible message frequency adapted to the load. For this purpose, the sending satellite tracks the number of receivers and adjusts the interval between messages Δt accordingly. The interval is chosen as the minimum of $\Delta t_{\min} = 500$ ms and a configured maximum interval Δt_{\max} with the square root of the number of receivers \sqrt{N} :

$$\Delta t = \min \left(\Delta t_{\max}, \max \left(\Delta t_{\min}, \Delta t_{\min} \cdot \sqrt{N} \cdot \ell \right) \right), \quad (1)$$

where ℓ is a load factor that allows the rate to be adjusted in the future to take additional factors into account. It is currently set to a fixed value of 3.

The receiving node expects a new heartbeat message from each of the senders within their announced heartbeat interval. If it fails to receive such a message in time, an internal *lives* counter is decremented. After losing three lives by not receiving a heartbeat message, the corresponding sender is considered non-responsive and appropriate action is taken. The successful reception of a heartbeat resets this counter to its initial value.

In addition to regular heartbeat patterns, so-called *extrasystoles* are sent whenever the state of the sender changes. This enables an immediate reaction to remote state changes without having to wait for the next regular heartbeat update interval. In case of long heartbeat intervals and a large node count, a delayed propagation of information is avoided.

Fig. 3 demonstrates the heartbeat exchange between a satellite and a controller instance. The controller connects to the satellite and receives regular heartbeat messages as well as intermittent extrasystole messages for state changes. Similarly, Fig. 4 shows the heartbeat exchange between two satellites with reciprocal subscription and heartbeat reception.

A detailed description of the features implemented in Constellation based on the information transmitted by means of heartbeat messages can be found in Section 5.

3.2.3. Control Commands

Commands from controller instances to satellites are transmitted via the *Constellation Satellite Control Protocol (CSCP)*. It resembles a client-server architecture with the typical request-reply pattern as indicated

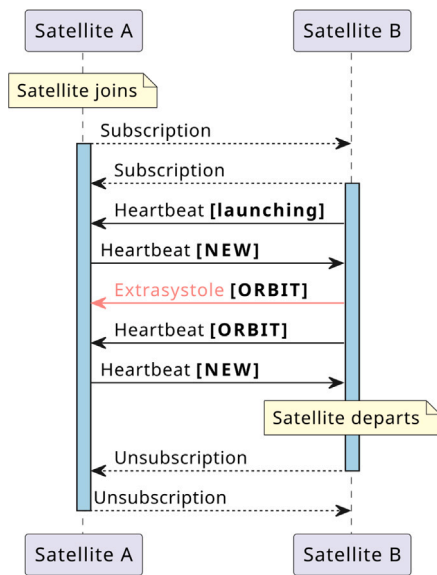


Fig. 4. Constellation Heartbeat Protocol (CHP) sequence diagram for the communication between two satellites exchanging heartbeats and extrasystoles. Here, satellite B is performing a transition and emits an extrasystole once the transition is completed. The vertical colored activation bars indicate time periods of heartbeat monitoring.

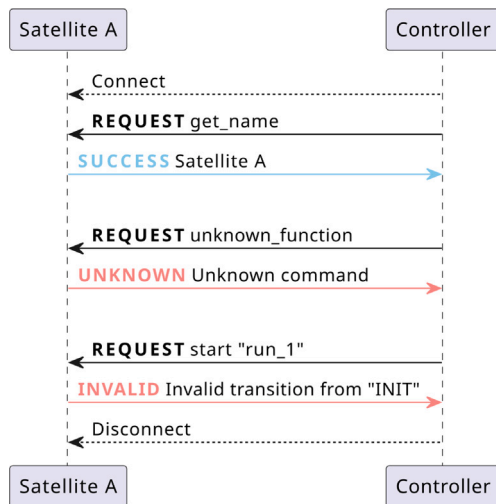


Fig. 5. Sequence diagram for Constellation Satellite Control Protocol (CSCP). A controller connects to a satellite and queries its name. Following are attempts to call an unknown function and to initiate an invalid state transition.

by the sequence diagram shown in Fig. 5. Here, the satellite acts as the server while the controller assumes the role of the client. Often, controllers will also subscribe to the CHP information of the satellites in order to directly receive real-time state updates as described above.

The command message consists of a message verb with a type and a command, and an optional payload to transmit data. The controller exclusively sends messages with type REQUEST, while the satellite answers with a message type matching the situation, such as SUCCESS when the command was executed successfully, INVALID if for example a transition was requested that is not possible from the current state, or UNKNOWN in case the command is not known to the satellite.

The satellite command palette contains a set of standard commands to query properties and initiate state transitions, but can be extended by a specific implementation as will be described in Section 4.2.

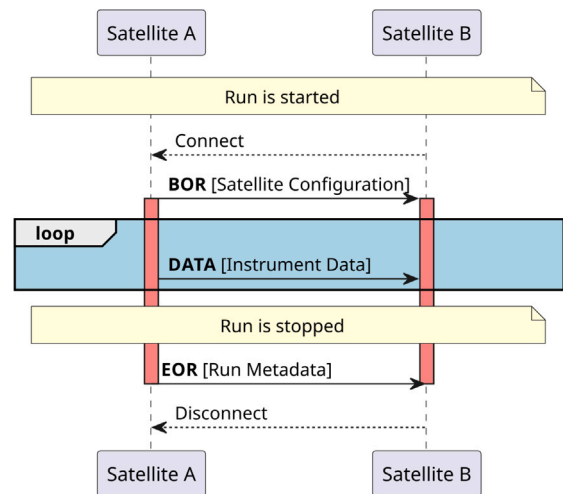


Fig. 6. Sequence diagram for data transmission via Constellation Data Transmission Protocol (CDTP). After a run is started, the receiver (Satellite B) connects to the transmitter (Satellite A). The transmitter first sends a BOR message, followed by DATA messages. Upon stopping of the run, the EOR message is sent. The colored activation bar indicates the period of active data taking.

3.2.4. Data Transmission

Data are transferred within a Constellation network using the Constellation Data Transmission Protocol (CDTP). It uses point-to-point connections via TCP/IP, as indicated in Fig. 6, which allow the bandwidth of the network connection to be utilized as efficiently as possible. The message format transmitted via CDTP is a lightweight combination of a header frame with sender information and a payload that differs depending on the message type. CDTP knows three different message types, indicated by a flag in the header frame:

- BOR - Begin of Run: This message is sent automatically at the start of a new measurement, i.e. upon entering the RUN state of the FSM as described in Section 4.1. It marks the start of a measurement in time and its payload frame contains the currently active satellite configuration for later reference, as well as other metadata such as the current run ID.
- DATA: This is the standard message type of CDTP which consists of the header frame and any number of so-called data records containing the data of the respective instrument. Each data record may contain multiple data blocks and is marked with an incrementing data sequence counter, providing the possibility for additional offline data integrity checks.
- EOR - End of Run: This message is sent automatically at the end of a measurement, i.e. when the sending satellite leaves the RUN state as described in Section 4.1. It contains metadata collected by the satellite over the course of the run.

The interface for satellites has been designed with maximal ease-of-use in mind, and does not require additional effort in satellite code to optimize data for network throughput. Data records are passed from the satellite code to the framework library and cached in a lock-free queue [30]. Whenever enough data have accumulated or a configurable timeout is reached, these data records are grouped into a CDTP DATA message and sent to their respective receivers.

Fig. 7 shows a benchmark scenario run over a 10G fiber-optic network using randomly generated data. At very small data record sizes, the data transfer is limited by memory allocation operations on the sending node as indicated by the performance difference between using the system memory allocator and an optimized allocator implementation such as *mimalloc* [31]. For record sizes above a few kilobyte, the network link bandwidth can be fully utilized.

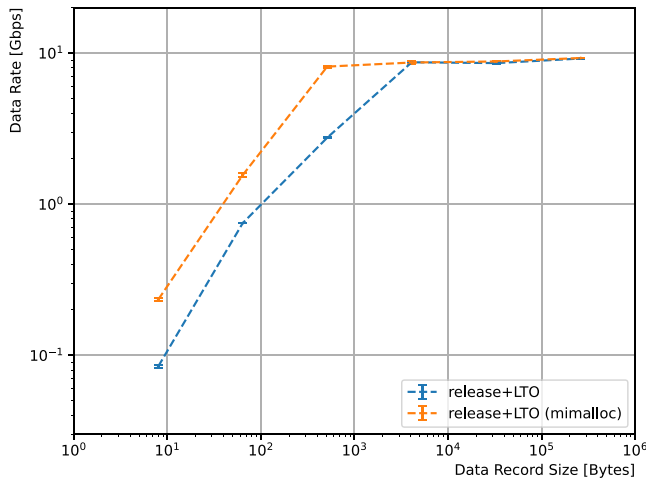


Fig. 7. Constellation Data Transmission Protocol (CDTP) network throughput benchmark over a 10G fiber-optic link. Shown is the achieved data rate as a function of the data record size for two different build configurations in *release* mode with link time optimization (LTO) activated, one using the system memory allocator and one *mimalloc* [31].

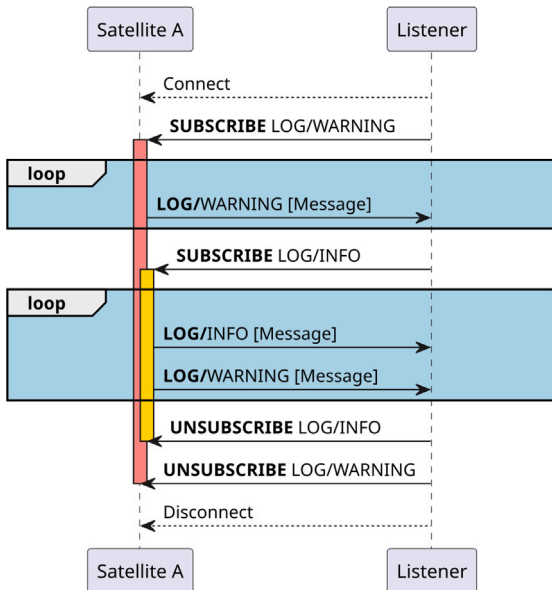


Fig. 8. Sequence diagram for Constellation Monitoring Distribution Protocol (CMDP). A listener subscribes first to log messages of severity level WARNING and then INFO and receives messages accordingly. The colored activation bars indicate the subscription status to a certain log topic from Satellite A.

3.2.5. Logging & Telemetry

The distribution of log messages and telemetry data within Constellation is handled by the *Constellation Monitoring Distribution Protocol (CMDP)*. The protocol is built around publisher and subscriber sockets which allow one-to-many distribution of messages. The CMDP protocol supports topics, which allows to select only the relevant slice of information from an otherwise very verbose communication, and therefore reduces the required network bandwidth.

The sequence diagram displayed in Fig. 8 displays the transmitted information depending on the subscription. First, a listener subscribes only to log levels with the severity WARNING and consequently only these messages are transmitted over the network. Subsequently, the listener also subscribes to INFO messages, which increases the number of messages sent.

Subscriptions to logging levels and telemetry are completely independent of other protocols and can be performed at any time. This means that listeners receiving log messages can be ended and restarted at will, and the subscriptions can be changed while the Constellation is running undisturbed, enabling flexible and adaptable monitoring configurations.

The C++ implementation of Constellation uses *spdlog* [32] to efficiently queue log messages while the Python implementation relies on the native logging mechanisms.

3.3. Limitations

Constellation targets laboratory test stands up to small and mid-sized experiments. The focus on flexibility and ease of integrating new instruments comes with some limitations, which will be discussed in the next paragraphs.

In its current version, the framework is intended to run in closed internal networks only, in the following referred to as *subnets*. It is assumed that the subnet and all connected nodes can be trusted, that there are no malicious actors active, and that the transmitted information is non-confidential to any actor on the subnet. It is possible to configure nodes to bind to specific network interfaces only, but the default configuration—chosen for user convenience—is to bind to all available network interfaces of the node. In general, users are required to satisfy their personal threat model by external means such as firewalls, physical isolation and virtualization on the edges of the subnet Constellation runs on.

In order to maintain a simple interface for control as well as instrument integration, the available states and transitions of the satellite FSM described in Section 4.1 are fixed and cannot be extended. Furthermore, in its current state, Constellation does not support a control hierarchy, and all satellites are directly managed by controllers. However, future versions of the framework could implement a multi-tier control hierarchy e.g. by using group names to separate control domains.

Finally, the number of heartbeat message exchanges over the CHP protocol grows quadratically with the number of nodes in the Constellation. While the congestion control described in Section 3.2.2 can offset performance impacts for several dozen or even hundred nodes, very large node counts may impair the response times to failure.

4. The Satellite

The central components of a Constellation group are satellites. A satellite is a program responsible for controlling an instrument or providing functionality such as data storage and is built around an FSM. It is the only component in a Constellation which partakes in all protocols discussed in the previous section. In this section, the main features of the Constellation Satellite are described.

The *state* of a satellite is governed by its FSM described in the subsequent section. By referring to the current state of the satellite, it can be unequivocally deduced what actions the satellite is currently performing and what transitions are possible. Sometimes, however, additional context can aid in interpreting this machine-readable state information. This information is provided by the satellite *status*, a human-readable string describing e.g. the summary of actions performed during a state transition. In case of a failure, the status string can provide information on the cause of the error.

4.1. Finite State Machine

The FSM controls the behavior of the satellite and guarantees that the system is always in a defined state. The FSM has been designed carefully to strike the balance between robustness, features, and simplicity. State transitions are either initiated by a controller sending transition commands through CSCP for regular operation as described in Section 3.2, or by autonomous actions that will be discussed in Section 5. In the subsequent paragraphs, the different states and their transitions are introduced.

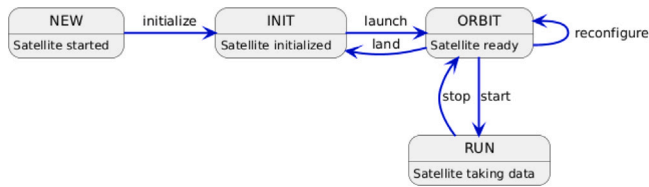


Fig. 9. State diagram of the satellite FSM with the four steady states NEW, INIT, ORBIT and RUN, linked by the transition commands.

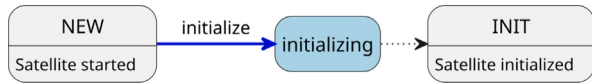


Fig. 10. Example for a transitional state. From the INIT state, the initialize command moves the FSM into the initializing transitional state which automatically transitions to the INIT state upon completion.

Steady states. In regular operations, i.e. without unexpected incidents in the Constellation, the satellite FSM will transition between four steady states. Here, steady indicates that the satellite will remain in this state until either a transition is initiated by a controller, or a failure mode is activated. The simplified state diagram for the normal operation mode is shown in Fig. 9.

The individual states of the satellite correspond to well-defined states of the attached instrument hardware or satellite functionality:

- The NEW state is the initial state of any satellite. It indicates that the satellite has just been started and that no configuration has been received from a controller.
- The INIT state indicates that the satellite has been initialized by having received configuration data from a controller. A first connection to the instrument may have been made at this point.
- The ORBIT state signals that the satellite is ready for data taking. When the satellite enters this state, the instrument has been configured successfully, is powered up and is ready for entering a measurement run.
- The RUN state represents the data acquisition mode of the satellite. In this state, the instrument is active and queried for data, the satellite handles measurement data, i.e. obtains data from the instrument and sends it across the Constellation, or receives data and stores it to file.

The RUN state implements the data taking operation of the satellite, and a `running` method is provided for satellite implementations to fill it with life. The method is called upon entering the RUN state and should exit as soon as requested by the satellite FSM. Each run is assigned a run identifier when started, consisting of alphanumeric characters, underscores or dashes.

The anomalous steady states ERROR and SAFE will be discussed in Section 5.

Transitions. The code which sets up functionality or configures attached instruments is mostly executed in so-called transitional states. Unlike steady states, they are entered by a state transition initiated through CSCP or a failure mode, but exited automatically upon completion of the action.

Such a transition diagram is shown in Fig. 10. In this example, the transition is triggered by the `initialize` command sent by the controller. The satellite enters the `initializing` transitional state and works through the instrument initialization code. Upon success, the satellite automatically transitions into the INIT steady state and remains there, awaiting further transition commands.

In this scheme, actions controlling and setting up the instrument hardware directly correspond to transitional states, and each of the

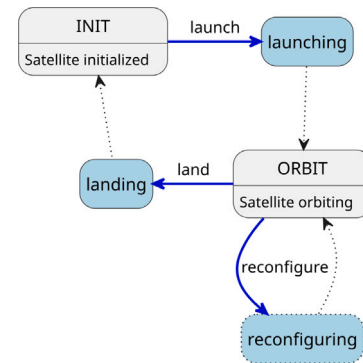


Fig. 11. State diagram detailing the possibilities of reconfiguring a satellite in ORBIT. Either the satellite has to go through the landing, initializing and launching states, or directly through reconfiguring.

transition commands corresponds to entering the corresponding transitional state. Transitional states therefore provide a direct feedback to the operator during potentially long-running actions such as the repositioning of a moveable stage or the ramping of a high-voltage power supply output.

The `initializing` transitional states connects the steady states NEW and INIT, the states `launching` and `landing` allow switching between INIT and ORBIT, and runs are started and stopped through the `starting` and `stopping` transitional states, respectively.

The optional `reconfiguring` transitional state enables quick configuration updates of satellites in ORBIT state without having to pass through the INIT state as indicated in Fig. 11. A typical example for reconfiguration is a high-voltage power supply unit, which is slowly ramped up to its target output voltage in the `launching` state. Between runs, the applied voltage is supposed to be changed by a few volts — and instead of the time-consuming operation of ramping down via the `landing` transition and ramping up again, the voltage is ramped directly from its current value to the target value through the `reconfigure` transitional state. This transition needs to be specifically implemented and activated in each satellite implementation in order to make it available in the FSM.

4.2. Controlling the Satellite

Satellites expose a set of remote procedure calls, called *commands*, through the CSCP protocol described in Section 3.2. Controllers can query for commands using the `get_commands` request which returns a list of available commands and FSM transitions along with their descriptions. They can be divided into two types:

Standard commands. These commands are provided by all satellite implementations. They comprise commands to initiate state transitions of the FSM, e.g. `launch`, as well as methods to query additional information from the satellite such as its name via `get_name` or the last known run identifier via `get_run_id`.

Custom commands. Individual satellite implementations may extend this command interface by implementing custom commands specific to this satellite type and the attached instrument hardware. Custom commands can receive input parameters through the command message payload, and can be limited to only be available in certain states of the FSM. A typical example for a state-limitation for a custom command is a compliance check for a high-voltage power supply. Reading this status when the satellite is in its ORBIT or RUN state and the output voltage is turned on will produce the expected result, while reading the value in the NEW or INIT states where the power supply is not fully configured and the output voltage is off does not provide meaningful information.

4.3. Publishing Monitoring Data

Constellation comes with an asynchronous monitoring mechanism which can transmit log messages of different verbosity levels as well as telemetry data over network to other nodes via CMDP as described in Section 3.2. In this context, asynchronous means that the code which emits a message continues immediately, while the processing of the message and its storage or transmission via network happens independently in the background. This way, even many messages of very high verbosity level will not affect the performance of the node. Furthermore, the subscription mechanism filters messages on the sending side, and only monitoring data with an active subscription will be transmitted.

Logging messages are distributed along with their verbosity level and a topic, in the form LOG/<LEVEL>/<TOPIC>. The verbosity levels range from TRACE and DEBUG for detailed information on library functions to WARNING for unexpected but non-fatal problems and STATUS for high-level status information such as state changes.

Some verbosity levels contain a lot of information. In order to allow further filtering of messages on the sending side, the topics divide the messages of each verbosity level into sections to which nodes can subscribe individually. An example subscription to LOG/TRACE/FSM would log messages on the very verbose TRACE level, but further constrained to the FSM, discarding all other messages.

Similarly, the telemetry is distributed with topic information following the pattern STAT/<TOPIC>. This allows nodes to subscribe to individual metrics, e.g. STAT/TEMPERATURE for a temperature monitoring endpoint.

The satellite interfaces provide convenience macros which allow sending metrics in regular time intervals or only whenever the value has changed.

Notification messages are published under the subscription topics LOG? and STAT?. They are automatically distributed whenever the available topics of the sending node have changed. This allows receivers such as user interfaces to conveniently display all available topics to the user.

5. Autonomous & Collaborative Operation

Constellations operate in a decentralized manner, each satellite runs autonomously and no central server which manages a common state is required. Instead, the communication of state information between the individual components of a Constellation is performed directly through the heartbeat messages introduced in Section 3.2. The information available from these heartbeats can be used for a series of features introduced in the following.

5.1. Failure Modes & Safe State

The autonomous satellite operation mandates a mechanism to deal with unexpected events occurring in the operation of a single satellite, but also within the entire Constellation. For this purpose, the satellite FSM knows two additional steady states, also referred to as *anomaly states*, reserved for anomalous situations:

The ERROR state is entered whenever an unexpected event occurs within the satellite code, the instrument control or the data transfer. This state can only be left by a manual intervention via a controller by resetting the satellite back into its INIT state.

The SAFE state on the other hand, is entered by the satellite when detecting an issue with another satellite in the Constellation. This state resembles the *safe mode* of an uncrewed spacecraft, where all non-essential systems are shut down and only essential functions such as communication remain active. For a satellite this could encompass powering down instruments or switching off power supply outputs. When no specific interrupt transition is implemented, this defaults to passing through the `stopping` and `landing` transitions described in Section 4.1. Also this state can only be left by a manual intervention via a controller, and the satellite will transfer back to its INIT state.

The main difference between the two failure states is the possible statement about the condition of the respective satellite. The SAFE state is achieved via a controlled shutdown of components and is a well-defined procedure, while the ERROR state is entered, for example, through a lack of control or communication with the instrument and therefore does not allow any statement to be made about the condition of attached hardware.

5.2. Satellite Autonomy & Roles

A satellite can convey information to peers by three means: by sending heartbeat messages or by ceasing to send them, by altering the state information transmitted via the heartbeat messages, and by departing from the Constellation using a CHIRP DEPART message. This information allows other satellites to react to a communicated anomaly state, the absence of heartbeats for a defined period of time or a departure of the satellite as detailed in Section 3.2.2.

In order to control the reaction of other satellites in case of irregularities, each heartbeat message contains a set of flags detailing the desired treatment of the sending satellite. To ease the configuration of a Constellation for operators, these flags are grouped into so-called *roles* which can be assigned to each satellite individually through the configuration provided with the `initialize` command. The following roles are defined:

ESSENTIAL. Satellites with this role are treated most strictly by their peers. Every reported issue will lead to an interruption of operation, including reported anomaly states, missing heartbeats or an orderly departure via CHIRP. If the incident appears during a run, this run will be marked as *degraded* in the EOR metadata described in Section 3.2. A typical example would be a satellite controlling a power supply.

DYNAMIC. This role is less strict with respect to orderly departure. An incident reported via their state or missing heartbeats will still cause peer satellites to interrupt operation, an orderly departure using a CHIRP message will be logged but does not stop operation. If this occurs during data taking, the run will be marked as *degraded*. This is the default role the satellite assumes if not configured differently.

TRANSIENT. Satellites in the transient role are considered dispensable to the core data taking operation of the Constellation. Consequently, neither their departure nor a reported anomalous state or missing heartbeats cause an interrupt of the ongoing operation. Incidents are logged and ongoing measurement runs are marked as *degraded*. A typical example would be e.g. an auxiliary environmental temperature sensor.

NONE. This role indicates entirely optional satellites. The remaining Constellation remains unaffected by incidents or departures of these satellites, and even ongoing runs will not reflect possible incidents in their metadata.

5.3. Conditional Transitions

Sometimes satellites require to be initialized, started or stopped in a specific order — for example, the instrument they control might depend on receiving a hardware clock from another component that is only available after initializing the corresponding satellite. This constitutes a directed acyclic graph of dependencies between satellites for each of the transitions.

For this purpose, Constellation provides the

```
_require_<transitional state>_after
```

configuration keywords, available for each of the transitions described in Section 4.1. The value is a list of other satellites that need to have successfully performed this transition before a given dependent satellite will perform its own transition action. The respective satellite

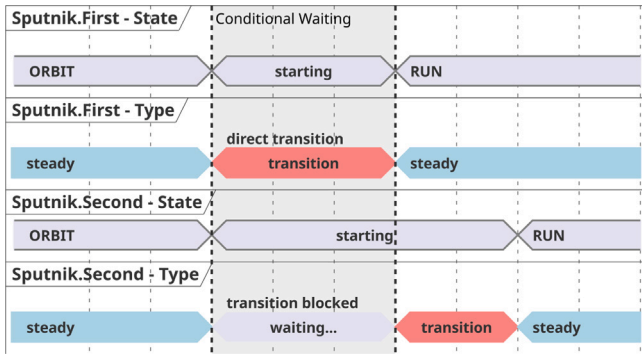


Fig. 12. Timing diagram of a conditional transition of satellite Sputnik.Second which waits for a successful transition of Sputnik.First.

will receive these conditions from the controller via the configuration passed in the initialize command and evaluate them upon entering transitional states.

If, for example, the satellite Sputnik.Second receives the condition

```
_require_starting_after = ["Sputnik.First"]
```

it will enter the starting transitional state but wait before progressing through its own starting action. The latter will commence as soon as satellite Sputnik.First has successfully completed the transition and emitted the state RUN via heartbeats as visualized in Fig. 12.

This method allows satellites to asynchronously and autonomously progress from steady state to steady state without the necessity of a controller supervising the order of action. It also allows multiple satellites waiting for the same remote condition to progress simultaneously once the condition is fulfilled. In contrast, a transition order orchestrated by a controller necessarily would progress sequentially.

Depending on the Constellation, not necessarily all combinations are valid and will function. For example, one satellite awaiting a hardware clock signal from another satellite, which depends on the former to finalize its initialization transition, will eventually time out. To avoid circular dependencies and the resulting deadlocks, controllers implement a directed graph validation logic using a depth-first approach to finding cyclic dependencies.

Waiting for remote conditions to be satisfied will be interrupted by the following events:

- The remote satellite that the condition depends on is not present or disappeared.
- The remote satellite that the condition depends on returns an ERROR state.
- The waiting satellite runs into the timeout for conditional transitions.

In all cases the waiting satellite aborts the pending action and transitions into its ERROR state.

6. User Interfaces

Constellation comes with a set of graphical and command-line user interfaces which will be described in this section.

6.1. Controller Interfaces

Controllers are the main interfaces through which operators can set up and alter the state of a Constellation. Currently, the framework provides two different controllers, a graphical user interface and a scriptable interactive command-line interface. Further interfaces, such as a stand-alone controller for a single satellite intended for laboratory setups, are under development.

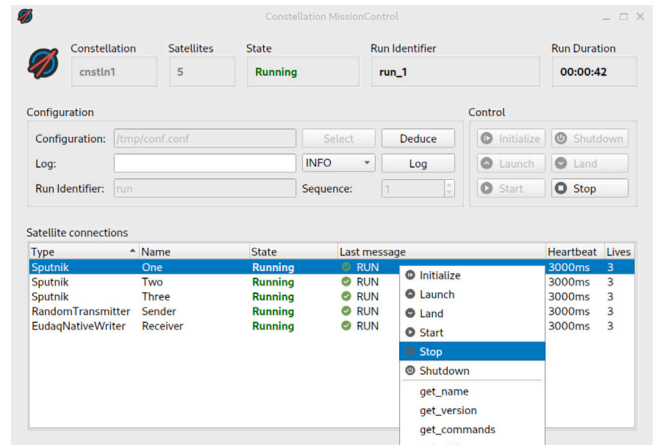


Fig. 13. Main window of the MissionControl controller interface, divided into header, control area and satellite list. The header indicates that the Constellation is in running state, and five connected nodes are shown in the satellite list, with an active context menu for the first satellite.

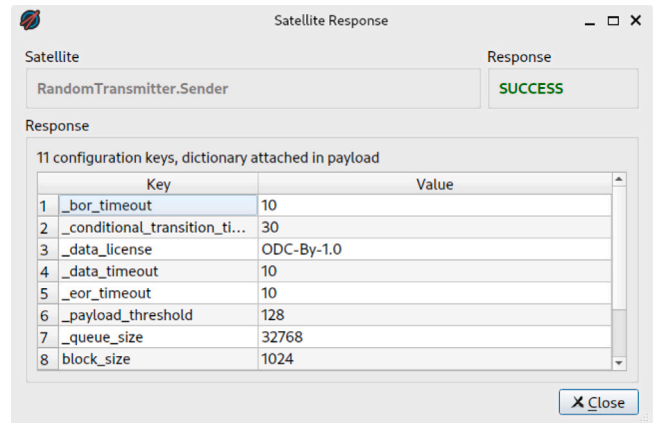


Fig. 14. Satellite command response dialog of the MissionControl controller interface, displaying the command payload to the get_config command.

Graphical controller: MissionControl. This graphical user interface is a general purpose controller for Constellation based on Qt6 [33]. The main window of MissionControl shown in Fig. 13 can be divided into three parts:

The header portion displays information on the Constellation which the controller is connected to, such as its name, the number of satellites and the global state. Here, the global state is a summarized state of all satellites. It represents the lowest state of any individual satellite, and, if not all satellites are in the same state, is amended by the symbol \approx to indicate a mixed global state.

Furthermore, the current or last run identifier along with its run duration are displayed. None of this information is stored locally; instead, it is all fetched from the running Constellation upon startup of the controller.

The section below the header is the control area steering the entire Constellation at once. The input fields allow selection of the configuration file, dispatch of log messages by the operator, and setup of the run identifier. The buttons on the right serve as control for the satellite FSM (cf. Section 4.1). Buttons for transition commands unavailable in the current global state are deactivated and grayed out.

Finally, the lower part of the main window is occupied by the list of all connected satellites of the Constellation. The type and name of each satellite are displayed alongside its current state, the last command response and the current CHP heartbeat interval and remaining lives. Double-clicking a satellite item from the list opens a dialog with

```

import time
from constellation.core.controller import
    ScriptableController

# Create controller
ctrl = ScriptableController(group_name)
constellation = ctrl.constellation

for ivl in [8, 64, 128]:
    # Reconfigure to new parameter value
    cfg = {"interval": ivl}
    constellation.Sputnik.reconfigure(cfg)
    ctrl.await_state(SatelliteState.ORBIT)
    # Repeat measurement four times:
    for run in range(1, 4):
        constellation.start(f"i{ivl}_r{run}")
        ctrl.await_state(SatelliteState.RUN)
    # Run for 15 seconds
    time.sleep(15)
    constellation.stop()
    ctrl.await_state(SatelliteState.ORBIT)

```

Listing 1: Example for a Python controller script which scans over three values of a satellite parameter, taking four fifteen-second runs for each of them. For simplicity, this example assumes the Constellation to be in the ORBIT state already.

additional connection details. A context menu, also shown in Fig. 13, provides direct access to the commands of the individual satellite, including transitions and possibly defined custom commands (cf. Section 4.2). All information is obtained directly from the running satellite and does not depend on any local cache or pre-loaded information in the controller. In case a command returns payload information, such as for example the `get_config` command, a dialog window displays this to the operator as shown in Fig. 14.

While the run identifier is a free-form string, this controller defines it as a name followed by a sequence number. The sequence number is automatically incremented when stopping and starting a run, such that a unique run identifier is always generated and sent to the Constellation.

Since satellites operate independently of the controller, and controllers can be started and closed at any time as described in Section 3.1, they might not necessarily all have access to the same configuration file. To alleviate this, the configuration of all satellites can be deduced from the Constellation in operation directly via the “Deduce” button of the controller. This will collect the current configuration from all connected satellites and open a dialog window to select a storage location for the configuration file.

Scriptable controller. In some scenarios, a scriptable command-line interface might be preferable to a graphical user interface. For this purpose, a Python controller class is provided, both as standalone script and as interactive shell based on IPython [34]. Individual satellites as well as the entire Constellation can be queried and controlled directly through the `ScriptableController` class. Complex routines such as automated parameter scans can be implemented in a few lines of Python code as demonstrated in Listing 1.

6.2. Receiving Log Messages

Log listeners subscribe to log topics through CMDP, to record log messages, and to present them to operators. Constellation offers different options to access log messages.

Graphical logging interface: Observatory. This interface is a flexible logger for Constellation based on Qt6 [33]. Its main window shown in Fig. 15 is structured in three parts:

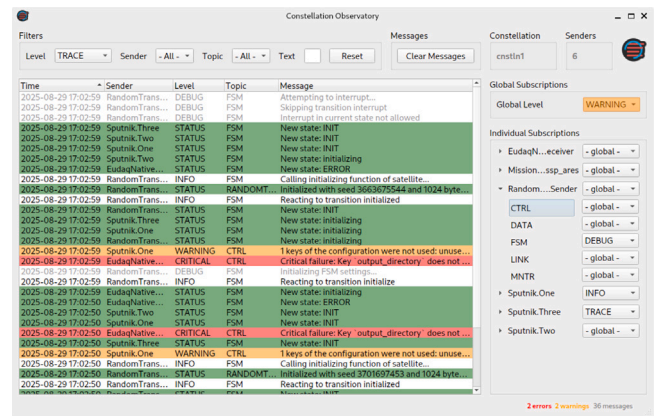


Fig. 15. Main window of the Observatory logging interface showing the log messages from an attempt to initialize the Constellation.

The central area of the window is occupied by the log message display. Each received message is displayed as one line in the display, with its timestamp, sender name, log level, topic and message text, colored according to the log level severity. By default, log messages appear time-ordered with the most recent message at the top. Double-clicking a log message opens a dialog with additional information such as message tags or source code location information.

The right sidebar displays information on the Constellation along with the number of connected log senders at the top, followed by a drop-down to select the global log subscription level. Below, a list of individual log senders, such as satellites or controllers, can be found, each with their own drop-down selection for log levels. When expanding the individual senders by clicking on their name, the sender-specific list of provided CMDP log topics is shown. These settings provide fine-grained control over the subscribed log topics and determine which log messages are transmitted by the respective senders and received by the listener in addition to those subscribed via the global log level. For user convenience, the log levels function as thresholds, i.e. any subscription automatically includes all higher-severity levels.

The filter settings for received messages are located at the top of the window. With their help, the displayed list of log messages can be filtered by any combination of log level, sender name, log topic, or text matching the log message. It should be noted that filtering messages does not prevent them from being transmitted over the network. For this, the corresponding subscription has to be adjusted.

The status bar of the application lists the total number of received messages and indicates the number of messages received with the levels WARNING or CRITICAL.

Storing log messages: FlightRecorder. In many application scenarios, keeping log messages for later inspection is an important feature. Constellation provides the *FlightRecorder* satellite that implements this functionality and provides the possibility to store log messages either to a single log file or to a set of rotating log files based on their size. Alternatively, the satellite can start a new log file every 24 h at a configured time, or switch to a new log file whenever a new run is started, i.e. when it received the `start` command. The latter can be especially helpful when many runs are recorded and an easy assignment of logs is required.

Passing messages to Mattermost. Remote monitoring of the system is paramount at all times. Therefore, Constellation comes with a satellite which connects the CMDP logging system with *Mattermost* [35]. The satellite listens to log messages sent by other satellites and forwards them to a configured Mattermost channel. The name of the respective satellite will be used as username to allow distinguishing them easily in the chat history. Log messages with a level of WARNING are marked

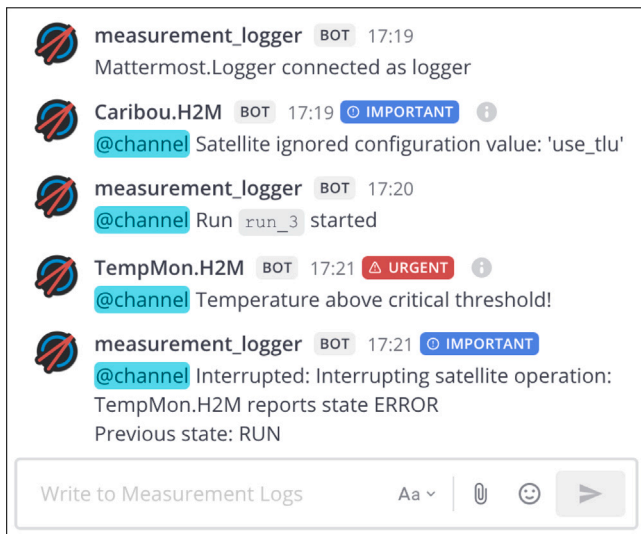


Fig. 16. Screenshot of *Mattermost*, displaying a channel with log messages from *Constellation*. Messages with level WARNING are marked as important, those with level CRITICAL as urgent, and are prefixed with @channel for notifications.

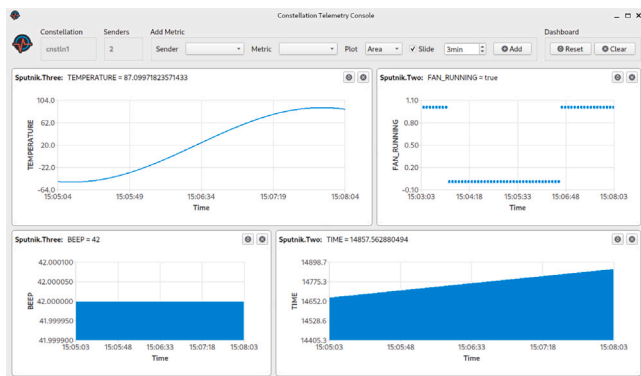


Fig. 17. Main window of the Telemetry Console, displaying a dashboard with different types of graphs for real-time satellite metrics.

as important, messages with level CRITICAL as urgent, and both are prefixed with @channel to notify all users in the Mattermost channel as demonstrated in Fig. 16.

6.3. Monitoring of Telemetry Data

Alongside log messages, also the monitoring of telemetry data is of vital importance to gauge the state of the *Constellation*. Telemetry listeners allow the operator to subscribe to metric topics emitted by the satellites, and to graphically display them in a variety of charts. Currently, there are two interfaces which allow graphical representation of metrics.

Real-time monitoring: Telemetry console. This graphical user interface provides a configurable dashboard to display multiple metric time series in real time. Using the toolbar at the top of the window shown in Fig. 17, operators can add new graphs to the dashboard area below. The interface offers different types of graphs such as spline, scatter plot or area, and the individual graphs can be resized by dragging the dividers with the mouse. Upon closing of the window, the current state of the dashboard – including the displayed graphs, connected nodes and the dashboard layout – is stored and can be restored when reopening the program.



Fig. 18. Example of a Grafana Dashboard displaying telemetry data of a *Constellation*. Shown are trigger rate information emitted by a satellite controlling the AIDA-2020 TLU [38], voltage and current information from a Keithley 2410 power supply satellite, and data rate and disk space information from a *Constellation* receiver satellite.

The metrics are displayed as a time series, starting from the moment the graph is added to the dashboard. No storage or memory of previously recorded metric information is implemented.

Archiving telemetry data: InfluxDB & Grafana. For long-term archiving of telemetry data, the *Influx* satellite, provided as part of *Constellation*, can be used paired with an *InfluxDB* time series database [36] and a *Grafana* instance [37]. The satellite listens to metrics sent by other satellites and writes them to the *InfluxDB*. Fig. 18 shows an example of a user-configured Grafana dashboard with a variety of telemetry information from a *Constellation*, such as trigger information from an AIDA-2020 trigger logic unit (TLU) [38], as well as voltage and current information from a Keithley 2410 source measure unit (SMU) satellite. The satellite classes of *Constellation* already implement several basic metrics such as data rates or available disk space for storage.

7. First Applications in Experimental Setups

In this section, selected examples of early applications of the *Constellation* framework in different deployment scenarios are presented, ranging from tracking of highly energetic particles to characterization of spent nuclear fuel and monitoring of a cryostat.

7.1. Sensor Testing at the DESY II Testbeam

Constellation was used to operate a particle tracking telescope and novel detector prototype at the DESY II Testbeam Facility [39]. The prototype, the so-called *Hybrid-to-Monolithic (H2M)* chip [40], is a monolithic active pixel sensor (MAPS) manufactured in a 65 nm CMOS imaging process. The goal of the measurement campaign was to study its performance in terms of efficiency and resolution as a function of different operation parameters.

The experimental setup comprises a beam telescope, a timing reference detector, a TLU as well as power supplies. Data for reconstructing reference trajectories of the incoming particles is provided by the *ADENIUM* beam telescope [41]. The timing reference consists of a *Telepix2* chip [42], positioned downstream of the telescope, which also serves as a region-of-interest trigger. The H2M prototype, placed at the center of the beam telescope, is operated and read out by the Caribou data acquisition (DAQ) system [43], and receives a reverse bias from a Keithley 2410 SMU. Trigger signals are discriminated and synchronously distributed to all detector systems by the AIDA-2020 TLU [38]. Configuration and readout of all components are coordinated using the *Constellation* framework, replacing the previously used EUDAQ2 framework [7].

A total of nine satellites were connected to the Constellation for data taking and monitoring. The DAQ system satellites for H2M, ADENIUM, Telepix2 and the TLU were ported from EUDAQ2 producers to the Constellation framework, and a receiver satellite was implemented to store data in a EUDAQ2-compatible file format. A *Keithley* satellite controls the bias voltage applied to the H2M sensor. In addition, an *Influx* satellite was used to store telemetry data for display in a Grafana interface, and a *Mattermost* satellite to keep operators informed. A Grafana display of this measurement campaign is shown in Fig. 18, indicating the trigger rate following the DESY II accelerator duty cycle and its correlation with the data rates. The number of generated trigger signals post- and pre-veto stage of the TLU are displayed, along with the input trigger signals from Telepix2. The interface also shows the sensor bias voltage and leakage current, the data transmission rate and the remaining available disk space.

Stable operation of the setup was achieved over several days of the measurement campaign. Data reconstruction and analysis were performed with the Corryvreckan framework [44]. Compared to data previously recorded with EUDAQ2, no changes to the analysis chain were required and consistent results were achieved, while stability, system monitoring and operation were significantly improved.

7.2. Bent Crystals at the CERN SPS North Area

The BIPXL telescope is a precision tracking system developed at the CERN SPS North Area Testbeam for the characterization of bent silicon crystal assemblies [45]. Bent crystals manipulate particle beams for applications such as crystal-assisted collimation for heavy-ion beam [46], shadowing of the SPS electrostatic septum for loss reduction [47], and future fixed-target experiments in HL-LHC [48]. The characterization of crystals in particle beams verifies essential specifications before installation. Key observables are the channeling efficiency relative to particle impact angle with respect to the crystalline plane orientation, track bending angle, and torsion of the crystal assembly.

An accurate selection of tracks inside a critical angle with respect to the crystal channeling planes requires angular precision within a few μrad for both incoming and outgoing particle tracks. To achieve such performance, the telescope consists of two arms, each spanning 10 m with vacuum pipes between its planes. The instrumentation includes up to six Timepix3 pixel detectors [49] and two scintillator-PMT trigger pairs, following a signal- and data-handling approach largely similar to the DESY II tracking setup described above.

Each reference detector plane position is adjustable via paired x/y Zaber motion stages [50], for reference detector alignment and extended angular acceptance of the downstream arm. The crystal under test is centrally positioned on a four-axes goniometer by PI [51] that allows precise alignment of its channeling planes relative to the beam.

System integration and control rely on the Constellation framework, with custom satellites managing the Timepix3 readout and up to 14 motion stages. Crystal alignment routines and operator feedback utilize two Corryvreckan-based satellites [44]. Constellation facilitated rapid deployment at CERN within four weeks, achieving stable, reliable operation throughout a multi-day commissioning campaign.

7.3. Spent Nuclear Fuel Characterization

Constellation has been used as the framework for the data acquisition in a prototype measurement station for spent nuclear fuel characterization [52] at the Swedish Nuclear Fuel and Waste Management Company (SKB) [53]. The data acquisition system consists of 24 Red Pitaya STEMlab 125-14 devices [54], one Red Pitaya SIG-NA Lab [55], and two CAEN Power supplies [56]: a model NDT1470 and a SY5527 crate with two A7435 boards installed.

Dedicated satellites have been implemented in Python for each of the two Red Pitaya models. The satellites are deployed on the Linux OS embedded in each Red Pitaya board and read out data are stored in a

ring buffer by the custom FPGA firmware. The Red Pitaya boards are equipped with Red Pitaya Click Shields to synchronize the clocks and connect additional environmental sensors via the 5 V I/O connectors. For the two different CAEN high-voltage power supplies, one combined satellite implementation was developed based on the PyCaenHV library [57]. The *Influx* satellite is utilized to store monitoring information in an InfluxDB database and to visualize the data using Grafana.

Signals arriving in any of the 98 analog detector channels are collected and processed in real-time on the FPGA of the connected Red Pitaya device. If a pulse is selected by the trigger logic of the channel, the result consisting of timestamp and pulse amplitude is first stored in RAM before being read out by the corresponding satellite. The data are then transferred via a network switch to the host computer connected over a 10G fiber-optic cable. Finally, the *H5DataWriter* satellite stores the data to file in the HDF5 format [58]. A dedicated graphical user interface was implemented in PyQt 6 using the Python controller class described in Section 6.1 to display and control all of the 29 satellites in the Constellation group.

In laboratory tests, the data acquisition system was able to handle continuous rates of 2 million events per second and channel. This corresponds to roughly 800 MB s^{-1} of processed data being written to disk, not counting additional meta information added by the framework and the chosen file format.

7.4. Cryostat Monitoring at MADMAX

The Magnetized Disk and Mirror Axion eXperiment (MADMAX) [59–61] is a dark matter experiment aiming to search for axions in the mass range of $40 \mu\text{eV}$ to $400 \mu\text{eV}$ using the dielectric haloscope approach [62]. The experiment consists of the so-called booster, a focusing mirror, and a microwave horn antenna connected to a sensitive receiver system. In order to achieve the required low noise temperature for the final system, all components will be placed inside a custom cryostat, called the MADMAX Cryostat. It is currently being commissioned in the SHielded Experimental haLL (SHELL) at University of Hamburg, from where it will later move to CERN for data taking in the MORPURGO magnet [63].

For commissioning, and later operating, the MADMAX Cryostat, it is crucial to monitor and log several observables of the required infrastructure such as vacuum and gas pressures, cryogenic temperatures, liquid helium level. Constellation is used to set up a scalable and extensible monitoring system for the studies. The monitored quantities are acquired by dedicated satellites communicating with the different measurement devices, and the resulting observables are recorded in an InfluxDB database by the *Influx* satellite and monitored via Grafana dashboards as described in Section 6.3. Additionally, the *Mattermost* satellite is used to send warnings when parameters leave the nominal operation range or if measurement devices malfunction. The network discovery functionality of Constellation allowed easy and user-friendly addition of satellites on different computers during commissioning.

For the future it is planned to use Constellation for the steering and monitoring of the MADMAX booster, and possibility also for the data acquisition and receiver system, which would allow implementation of the entire experiment control and data acquisition in Constellation.

8. Documentation

Constellation comes with extensive technical documentation which is published on the project website [20] and as PDF documents. It is divided into three separate documents that cater to specific target audiences:

Operator guide. This document targets operators and people who will set up and run Constellation, control satellites and monitor the performance of the system. The guide is structured in four different

components serving different purposes: The *Get Started* section describes the installation procedure and initial setup. *Tutorials* teach how to use Constellation using practical examples, starting from simple situations such as starting and controlling a single satellite, and gradually moving to more complex examples and setups. The *Concepts* section provides detailed explanation of the workings of the framework and the thoughts behind its structure. This is not the technical documentation required by developers, and it simplifies explanations where possible. Finally, the *How-To Guides* provide concise answers on how to achieve a specific goal, such as setting up a telemetry database, or sending log messages to Mattermost using the satellite described in Section 6.2.

Application developer guide. This document is intended for those that would like to integrate new instruments and functionality through satellites, or to implement new user interfaces. It contains detailed instructions for implementing new satellites in C++ as well as in Python, a detailed description of features such as the custom commands introduced in Section 4.2, and guidance on how to integrate them into satellites. The *How-To Guides* section contains instructions such as porting a EUDAQ2 producer [7] to a Constellation satellite, or how to use Constellation as an external dependency in other projects.

Framework development guide. The final document targets developers and potential contributors to the central Constellation framework components. It contains the technical details of the implementation, alongside the protocol RFC documents mentioned in Section 2.3, satellite implementation guidelines, naming conventions, and the reference for the application programming interfaces of the different Constellation components.

In addition to these three documents, the website also hosts a list of satellites available from different sources. For each satellite, a description of the functionality as well as the available commands, metrics and configuration parameters are listed alongside their default values where applicable. The page lists satellites that are part of the main Constellation repository as well as implementations that are hosted externally. The latter is the preferred option for satellites that implement control of specialized hardware, while satellites with broad applicability or those that control commercial off-the-shelf hardware can be submitted for inclusion in the main repository.

9. Conclusions & Outlook

Constellation is a flexible, decentralized, and extensible framework for control and data acquisition in dynamic experimental setups. By putting focus on autonomy, standardized protocols, and ease of integration, it addresses several of the limitations found in existing solutions, particularly for small to mid-sized laboratory experiments.

The design is centered around independent satellite nodes, which enables operation without depending on a central control server. Regular heartbeat messages exchanged between the nodes enable the implementation of advanced features such as autonomous reaction to error conditions or conditional transition sequences.

The framework has already demonstrated its practical applicability in different experimental environments such as charged particle tracking, spent nuclear fuel characterization and cryostat temperature monitoring. It is capable of coordinating multiple heterogeneous components, enables autonomous operation, and provides a stable foundation for control, system monitoring and data acquisition of small to mid-size experimental setups.

Several enhancements are under consideration for future versions of Constellation such as an access control system to avoid state changes by mistake, and hierarchical control domains through intermediary satellites. Future developments will continue to be community-driven, e.g. through EDDA hackathons, and will focus on flexibility and long-term stability of the framework.

CRediT authorship contribution statement

Simon Spannagel: Writing – review & editing, Writing – original draft, Visualization, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Stephan Lachnit:** Writing – review & editing, Visualization, Software, Investigation, Conceptualization. **Hanno Perrey:** Writing – review & editing, Software, Investigation, Funding acquisition, Conceptualization. **Justus Braach:** Writing – review & editing, Software, Investigation. **Lene Kristian Bryngemark:** Writing – review & editing, Writing – original draft, Project administration, Methodology, Funding acquisition, Conceptualization. **Erika Garutti:** Resources, Project administration, Funding acquisition. **Adrian Herkert:** Writing – review & editing, Validation. **Finn King:** Writing – review & editing, Validation, Software. **Christoph Krieger:** Validation, Supervision. **David Leppla-Weber:** Validation, Software. **Linus Ros:** Writing – review & editing, Software, Investigation, Conceptualization. **Sara Ruiz Daza:** Writing – original draft, Investigation. **Murtaza Safdari:** Writing – review & editing, Validation. **Luis G. Sarmiento:** Writing – review & editing, Conceptualization. **Annika Vauth:** Project administration, Funding acquisition, Conceptualization. **Håkan Wennlöf:** Software, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This project has received funding from the Lund University, Sweden and Universität Hamburg, Germany Seed Fund 2023. Part of the presented measurement results have been performed at the Test Beam Facility at DESY Hamburg (Germany), a member of the Helmholtz Association (HGF).

Data availability

No data was used for the research described in the article.

References

- [1] L.R. Dalesio, J.O. Hill, M. Kraimer, S. Lewis, D. Murray, S. Hunt, W. Watson, M. Clausen, J. Dalesio, The experimental physics and industrial control system architecture: past, present, and future, *Nucl. Instrum. Methods A* 352 (1) (1994) 179–184, [http://dx.doi.org/10.1016/0168-9002\(94\)91493-1](http://dx.doi.org/10.1016/0168-9002(94)91493-1).
- [2] G. Grygiel, O. Hensler, K. Rehlich, DOOS: a distributed object oriented control system on PC's and workstations, in: *PCaPAC Conference*, 1996.
- [3] J.M. Chaize, A. Gotz, W.D. Klotz, J. Meyer, M. Perez, E. Taurel, TANGO - an object oriented control system based on CORBA, *Conf. Proc. C* 991004 (1999) 475–479.
- [4] J. Gutleber, L. Orsini, Software architecture for processing clusters based on I2O, *Clust. Comput.* 5 (2002) 55–64, <http://dx.doi.org/10.1023/A:1012744721976>.
- [5] S. Ritt, P. Amaudruz, K. Olchanski, The MIDAS data acquisition system, in: *Proc. IEEE 10th Real Time Conf.*, 1997, pp. 309–312.
- [6] P. Ahlburg, et al., EUDAQ – a data acquisition software framework for common beam telescopes, *J. Instrum.* 15 (01) (2020) <http://dx.doi.org/10.1088/1748-0221/15/01/p01038>, P01038–P01038, [arXiv:1909.13725](https://arxiv.org/abs/1909.13725).
- [7] Y. Liu, M. Amjad, P. Baesso, D. Cussans, J. Dreyling-Eschweiler, R. Ete, I. Gregor, L. Huth, A. Irls, H. Jansen, K. Krueger, J. Kvasnicka, R. Peschke, E. Rossi, A. Rummmler, F. Sefkow, M. Stanitzki, M. Wing, M. Wu, EUDAQ2 — A flexible data acquisition software framework for common test beams, *J. Instrum.* 14 (10) (2019) <http://dx.doi.org/10.1088/1748-0221/14/10/p10033>, P10033–P10033.
- [8] K. Biery, E. Flumerfelt, J. Freeman, W. Ketchum, G. Lukhanin, R. Rechenmacher, Artdaq: DAQ software development made simple, *J. Phys. Conf. Ser.* 898 (3) (2017) 032013, <http://dx.doi.org/10.1088/1742-6596/898/3/032013>.
- [9] M. Boretto, W. Brylinski, G. Lehmann Miotto, E. Gamberini, R. Sipos, V.V. Sonesten, DAQling: an open-source data acquisition framework, *EPJ Web Conf.* 245 (2020) 01026, <http://dx.doi.org/10.1051/epjconf/202024501026>.

- [10] The ZeroMQ authors, ZeroMQ: An open-source universal messaging library, 2024, <https://zeromq.org/>, (Accessed 16 December 2025).
- [11] S. Furuhashi, MessagePack: It's like JSON. but fast and small, 2021, <https://msgpack.org/>, (Accessed 16 December 2025).
- [12] L.K. Bryngemark, E. Garutti, S. Spannagel, A. Vauth, Lund-Hamburg funding: EDDA, 2023, <https://www.uni-hamburg.de/en/internationales/mitarbeitende/mitarbeitende-wissenschaft/foerderprogramme-austauschprogramme-gesamt/lund-hamburg-funding/edda.html>.
- [13] Internet Engineering Task Force, About RFCs, <https://www.ietf.org/process/rfc/>, (Accessed 16 December 2025).
- [14] Clang thread sanitizer, 2025, <https://clang.llvm.org/docs/ThreadSanitizer.html>, (Accessed 16 December 2025).
- [15] The Catch2 authors, Catch2 unit testing framework, 2025, <https://github.com/catchorg/Catch2>, (Accessed 16 December 2025).
- [16] H. Krekel, et al., Pytest: helps you write better programs, 2025, <https://docs.pytest.org/en/stable/>, (Accessed 16 December 2025).
- [17] Python package index - PyPI, 2025, <https://pypi.org/>, (Accessed 16 December 2025).
- [18] Flatpak—the future of application distribution, 2025, <https://flatpak.org/>, (Accessed 16 December 2025).
- [19] The Constellation authors, Constellation software repository, 2025, <https://gitlab.desy.de/constellation/constellation>, (Accessed 16 December 2025).
- [20] The Constellation authors, Constellation: The autonomous control and data acquisition system for dynamic experimental setups, 2024, <https://constellation.pages.desy.de/>, (Accessed 16 December 2025).
- [21] The Constellation authors, Constellation GitHub mirror, 2025, <https://github.com/constellation-daq/Constellation>, (Accessed 16 December 2025).
- [22] The European Commission, The European public licence (EURL), 2017, <https://interoperable-europe.ec.europa.eu/collection/eupl/solution/eupl-freeopen-source-software-licence-european-union>, (Accessed 16 December 2025).
- [23] S. Lachnit, H. Perrey, S. Spannagel, Constellation, 2025, <http://dx.doi.org/10.5281/zenodo.15688357>.
- [24] M. Cotton, L. Eggert, D.J.D. Touch, M. Westerlund, S. Cheshire, Internet assigned numbers authority (IANA) procedures for the management of the service name and transport protocol port number registry, in: RFC 6335, RFC Editor, 2011, <http://dx.doi.org/10.17487/RFC6335>.
- [25] S. Cheshire, M. Krochmal, Multicast DNS, in: RFC 6762, RFC Editor, 2013, <http://dx.doi.org/10.17487/RFC6762>.
- [26] Internet Engineering Task Force, Zero configuration networking, 2004, <https://avahi.org/>, (Accessed 16 December 2025).
- [27] The Avahi developers, Avahi service discovery suite, 2024, <https://avahi.org/>, (Accessed 16 December 2025).
- [28] C. Gaspar, M. Dönselmann, P. Charpentier, DIM, a portable, light weight package for information publishing, data transfer and inter-process communication, *Comp. Phys. Comm.* 140 (1–2) (2001) 102–109.
- [29] S. Lachnit, Developing a network discovery protocol for the constellation control and data acquisition framework, *Nucl. Instrum. Methods A* 1081 (2026) 170838, <http://dx.doi.org/10.1016/j.nima.2025.170838>, arXiv:2503.17077.
- [30] M. Egorushkin, Atomic_queue: C++14 multiple-producer-multiple-consumer lock-free queues based on circular buffers and std::atomic, 2019, https://github.com/max0x7ba/atomic_queue, (Accessed 16 December 2025).
- [31] D. Leijen, B. Zorn, L. de Moura, Mimalloc: Free List Sharding in Action, *Tech. Rep. MSR-TR-2019-18*, Microsoft, 2019, URL <https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/>.
- [32] G. Melman, Spdlog: A fast C++ logging library, 2024, <https://github.com/gabime/spdlog>, (Accessed 16 December 2025).
- [33] The Qt Company, Qt6 documentation, 2025, <https://doc.qt.io/qt-6/>, (Accessed 16 December 2025).
- [34] F. Pérez, B.E. Granger, IPython: a system for interactive scientific computing, *Comput. Sci. Eng.* 9 (3) (2007) 21–29, <http://dx.doi.org/10.1109/MCSE.2007.53>.
- [35] Mattermost, Inc., Mattermost, 2025, <https://mattermost.com>, (Accessed 16 December 2025).
- [36] InfluxData Inc., InfluxDB: Purpose-built database for all types of time series data at any scale, 2025, <https://www.influxdata.com/>, (Accessed 16 December 2025).
- [37] Grafana Labs, Grafana: The open and composable observability platform, 2025, <https://grafana.com/>, (Accessed 16 December 2025).
- [38] P. Baesso, D. Cussans, J. Goldstein, The AIDA-2020 TLU: a flexible trigger logic unit for test beam facilities, *J. Instrum.* 14 (09) (2019) P09019, <http://dx.doi.org/10.1088/1748-0221/14/09/P09019>.
- [39] R. Diener, J. Dreyling-Eschweiler, H. Ehrlichmann, et al., The DESY II test beam facility, *Nucl. Instrum. Methods A* 922 (2019) 265–286, <http://dx.doi.org/10.1016/j.nima.2018.11.133>.
- [40] S. Ruiz Daza, R. Ballabriga, E. Buschmann, M. Campbell, R. Casanova Mohr, D. Dannheim, J. Dilg, A. Dorda, F. King, O. Feyens, P. Gadow, I. Gregor, K. Hansen, Y. He, L. Huth, I. Kremastiotis, C. Lemoine, S. Maffessanti, L. Mendes, Y. Otarid, C. Reckleben, S. Rettie, M. del Rio Viera, J. Schlaadt, A. Simancas, W. Snoeys, S. Spannagel, T. Vanat, A. Velyka, G. Vignola, H. Wennlöf, The H2M Monolithic Active Pixel sensor — characterizing non-uniform in-pixel response in a 65 nm CMOS imaging technology, *J. Instrum.* 20 (06) (2025) C06037, <http://dx.doi.org/10.1088/1748-0221/20/06/C06037>.
- [41] Y. Liu, C. Feng, I.-M. Gregor, A. Herkert, L. Huth, M. Stanitzki, Y. Teng, C. Yang, ADENIUM — A demonstrator for a next-generation beam telescope at DESY, *J. Instrum.* 18 (06) (2023) P06025, <http://dx.doi.org/10.1088/1748-0221/18/06/p06025>.
- [42] L. Huth, H. Augustin, L. Dittmann, S. Dittmeier, J. Hammerich, Y. He, A. Herkert, D. Immig, D. Kim, U. Krämer, R. Kolb, I. Perić, B. Pils, S. Ruiz Daza, T. Senger, A. Schöning, M. Stanitzki, B. Weinläder, A. Wintle, TelePix2: Full scale fast region of interest trigger and timing for the EUDET-style telescopes at the DESY II test beam facility, 2025, <http://dx.doi.org/10.1016/j.nima.2025.170720>, arXiv:2503.08177.
- [43] Y. Otarid, M. Benoit, E. Buschmann, H. Chen, D. Dannheim, T. Koffas, R. St-Jean, S. Spannagel, S. Tang, T. Vanat, Caribou - A versatile data acquisition system for silicon pixel detector prototyping, 2025, arXiv:2502.03903.
- [44] D. Dannheim, K. Dort, L. Huth, et al., Corryvreckan: a modular 4D track reconstruction and analysis software for test beam data, *J. Instrum.* 16 (03) (2021) P03008, <http://dx.doi.org/10.1088/1748-0221/16/03/p03008>.
- [45] Y.A. Chesnokov, V.I. Kotov, V.M. Biryukov, *Crystal Channeling and Its Applications at High-Energy Accelerators*, Springer, Berlin, 1997, <http://dx.doi.org/10.1007/978-3-662-03407-1>.
- [46] S. Redaelli, et al., Crystal collimation of heavy-ion beams, in: Proc. IPAC'24, in: IPAC'24 - 15th International Particle Accelerator Conference, JACoW Publishing, Geneva, Switzerland, 2024, pp. 3927–3932, <http://dx.doi.org/10.18429/JACoW-IPAC2024-FRNX3>, no. 15.
- [47] F.M. Velotti, L.S. Esposito, M.A. Fraser, V. Kain, S. Gilardoni, B. Goddard, M. Pari, J. Prieto, R. Rossi, W. Scandale, L.S. Stoel, F. Galluccio, M. Garattini, Y. Gavrikov, Septum shadowing by means of a bent crystal to reduce slow extraction beam loss, *Phys. Rev. Accel. Beams* 22 (2019) 093502, <http://dx.doi.org/10.1103/PhysRevAccelBeams.22.093502>.
- [48] D. Mirarchi, A.S. Fomin, S. Redaelli, W. Scandale, Layouts for fixed-target experiments and dipole moment measurements of short-lived baryons using bent crystals at the LHC, *Eur. Phys. J. C* 80 (10) (2020) 929, <http://dx.doi.org/10.1140/epjc/s10052-020-08466-x>.
- [49] T. Poikela, J. Plosila, T. Westerlund, M. Campbell, M.D. Gaspari, X. Llopart, V. Gromov, R. Kluit, M.v. Beuzekom, F. Zappone, V. Zivkovic, C. Brezina, K. Desch, Y. Fu, A. Kruth, Timepix3: a 65k channel hybrid pixel readout chip with simultaneous ToA/ToT and sparse readout, *J. Instrum.* 9 (05) (2014) C05013, <http://dx.doi.org/10.1088/1748-0221/9/05/C05013>.
- [50] Zaber Technologies Inc., Simplifying motion control, 2025, <https://www.zaber.com>, (Accessed 19 August 2025).
- [51] Physik Instrumente (PI) SE & Co. KG, Lösungen für präzise bewegung und positionierung, 2025, <https://www.physikinstrumente.de/>, (Accessed 19 August 2025).
- [52] L. Ros, M. Preston, U. Bäckström, E. Huuva, H. Liljenfeldt, A. Mehic, J.-O. Stål, Design of a Prototype Gamma-Ray and Neutron Measurement Station for Spent Fuel Characterisation Prior to Encapsulation, *Tech. Rep. SKB R-25-11*, Svensk Kärnbränslehantering AB, 2025, URL <https://skb.se/publikation/2519889>.
- [53] Svensk Kärnbränslehantering AB, SKB homepage, 2025, <https://skb.se/>, (Accessed 15 August 2025).
- [54] Red Pitaya, STEMLab 125-14, 2025, <https://redpitaya.com/stemlab-125-14/>, (Accessed 15 August 2025).
- [55] Red Pitaya, SIGNALlab 250-12, 2025, <https://redpitaya.com/signallab-250-12/>, (Accessed 15 August 2025).
- [56] CAEN S.p.A, CAEN homepage, 2025, <https://www.caen.it/>, (Accessed 15 August 2025).
- [57] Vassil Verguilov, Pycaenhv - Pure python bindings for CAENHVWrapper, 2025, <https://github.com/vasoto/pycaenhv>, (Accessed 15 August 2025).
- [58] The HDF Group, Hierarchical data format, version 5, 1997–2024, <https://www.hdfgroup.org/HDF5/>, (Accessed 15 August 2025).
- [59] P. Brun, A. Caldwell, L. Chevalier, G. Dvali, P. Freire, E. Garutti, S. Heyminck, J. Jochum, S. Knirck, M. Kramer, C. Krieger, T. Lasserre, C. Lee, X. Li, A. Lindner, B. Majorovits, S. Martens, M. Matysek, A. Millar, G. Raffelt, J. Redondo, O. Reimann, A. Ringwald, K. Saikawa, J. Schaffran, A. Schmidt, J. Schütte-Engel, F. Steffen, C. Strandhagen, G. Wieching, A new experimental approach to probe QCD axion dark matter in the mass range above 40 μeV , *Eur. Phys. J. C* 79 (3) (2019) <http://dx.doi.org/10.1140/epjc/s10052-019-6683-x>, arXiv:1901.07401.
- [60] J. Egge, D. Leppla-Weber, S. Knirck, B. Ary dos Santos Garcia, D. Bergermann, A. Caldwell, V. Dabhi, C. Diaconu, J. Diehl, G. Dvali, M. Ekmedzić, F. Gallo, E. Garutti, S. Heyminck, F. Hubaut, A. Ivanov, J. Jochum, P. Karst, M. Kramer, D. Kreikemeyer-Lorenzo, C. Krieger, C. Lee, A. Lindner, J. Maldonado, B. Majorovits, S. Martens, A. Martini, A. Miyazaki, E. Öz, P. Pralavorio, G. Raffelt, A. Ringwald, J. Redondo, S. Roset, N. Salama, J. Schaffran, A. Schmidt, F. Steffen, C. Strandhagen, I. Usherov, H. Wang, G. Wieching, G. Cancelo, M. Di Federico, G. Hoshino, L. Stefanazzi, First search for dark photon dark matter with a MADMAX prototype, *Phys. Rev. Lett.* 134 (15) (2025) <http://dx.doi.org/10.1103/physrevlett.134.151004>, arXiv:2408.02368.
- [61] B. Ary dos Santos Garcia, D. Bergermann, A. Caldwell, V. Dabhi, C. Diaconu, J. Diehl, G. Dvali, J. Egge, E. Garutti, S. Heyminck, F. Hubaut, A. Ivanov, J. Jochum, S. Knirck, M. Kramer, D. Kreikemeyer-Lorenzo, C. Krieger, C. Lee, D. Leppla-Weber, X. Li, A. Lindner, B. Majorovits, J. Maldonado, A. Martini, A. Miyazaki, E. Öz, P. Pralavorio, G. Raffelt, J. Redondo, A. Ringwald, J. Schaffran,

- A. Schmidt, F. Steffen, C. Strandhagen, I. Usherov, H. Wang, G. Wieching, First search for axion dark matter with a MADMAX prototype, *Phys. Rev. Lett.* 135 (4) (2025) <http://dx.doi.org/10.1103/c749-419q>, [arXiv:2409.11777](https://arxiv.org/abs/2409.11777).
- [62] A. Caldwell, G. Dvali, B. Majorovits, A. Millar, G. Raffelt, J. Redondo, O. Reimann, F. Simon, F. Steffen, Dielectric haloscopes: A new way to detect axion dark matter, *Phys. Rev. Lett.* 118 (9) (2017) <http://dx.doi.org/10.1103/physrevlett.118.091801>, [arXiv:1611.05865](https://arxiv.org/abs/1611.05865).
- [63] M. Morpurgo, A large superconducting dipole cooled by forced circulation of two phase helium, *Cryog.* 19 (7) (1979) 411–414, [http://dx.doi.org/10.1016/0011-2275\(79\)90126-7](http://dx.doi.org/10.1016/0011-2275(79)90126-7).