Constellation

Autonomous Control and Data Acquisition System for Dynamic Experimental Setups

Simon Spannagel, DESY
for the EDDA Collaboration

2nd DRD3 Week, CERN
2024-12-03

Outline

The Process

The Framework

The Project

What do we expect from a “flexible DAQ system”?

- **Useful to control single laboratory setup**
  (e.g. radioactive source measurement)
- **Possibility to integrate multiple setups**
  (Detector DAQ, TCT laser control)
- **Lab supervision mode**
  (multiple setups monitored but control not ceded)
- **Synchronized operations**
  (test beam environment, coordinated start/stop, central control)
- **Scalability for small experiments**
  (many detectors, multiple data endpoints & monitors)

The Process

User Stories

Hackathons

Protocols

EDDA
Exchange on &
Development of
Data **A**cquisitions

The User Stories

Idea: collect stories of how users do or *would like to* use their laboratory setups and derive requirements for the software from this → **user-centered development**

Example: “Slow control and conditions logging”

> I have a detector (or source) mounted on a motorized stage that I want to move for beam scans. I want to set the position as part of run control and log it [...] in a run conditions database. I also want to log environmental data such as temperature, humidity, pressure, ... there. While all this could be done by a shifter, that is error prone and requires manual intervention [...].

Derived Requirements

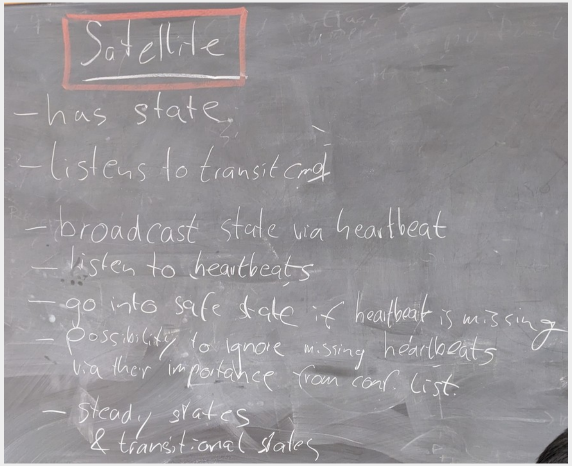
- Possibility to store meta data in database
- Allow satellites to provide metadata for storage (data stream, statistics)

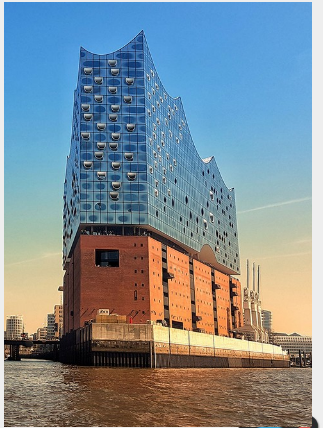
The Hackathons

Idea: sit together and code first drafts, discuss concepts, re-iterate thoughts → **collaboration-driven development**

Organized two hackathons to far:

- 1st in Lund, Nov. 2023 – focus on protocols, first prototypes
- 2nd in Hamburg, May 2024 – focus on user interface, docs





The Protocols

Idea: Instead of coding ahead, write RFC-style protocol documents to precisely define communication → **design-driven development**

This allows parallel, independent implementations in different languages

Constellation Host Identification and Reconnaissance Protocol

- Status: draft
- Editor: The Constellation authors

The Constellation Host Identification and Reconnaissance Protocol (CHIRP) defines how different hosts announce their services and connect to each other on the network.

Preamble

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

The Framework

Satellites

Controllers

Listeners

Introducing Constellation

- Project goals:
  - **Easy** to use, easy & fast to integrate new systems
  - **Stable** operation, reliable error handling
  - **Flexible** and applicable for many use cases
- Participants are called **satellites**
  - Operation is governed by a **finite state machine**
  - Satellites can operate **autonomously** without active user interface
- Independent implementations in Python & C++
- Solid foundation: well-defined communication protocols between components

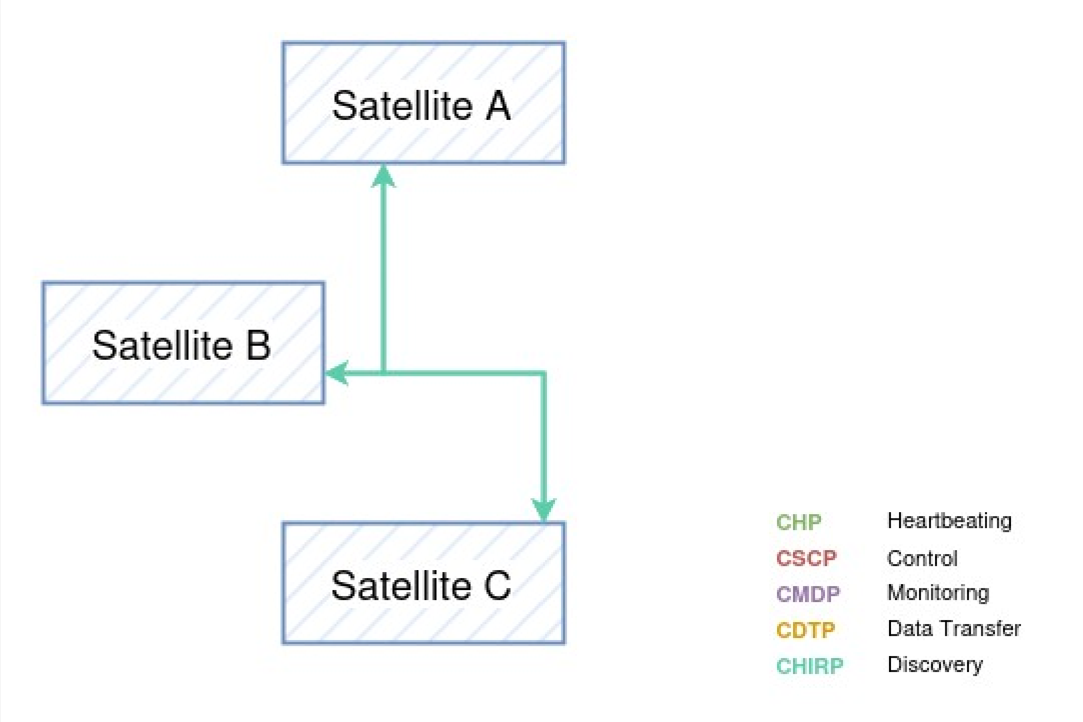
Communication Protocols

Protocol RFC Documents

- All interaction is based on five protocols:
  - **CHIRP** (Constellation Host Identification and Reconnaissance Protocol)
    Automatic network discovery of satellites/services
  - **CHP** (Constellation Heartbeat Protocol)
    Exchange of “I’m alive” signals with FSM state, variable sender-defined intervals
  - **CSCP** (Constellation Satellite Control Protocol)
    Controller protocol which sends transition & other commands
  - **CMDP** (Constellation Monitoring Distribution Protocol)
    Monitoring and logging data broadcasting, only subscribed topics on the wire
  - **CDTP** (Constellation Data Transmission Protocol)
    Data transmission with extra features such as sequences, begin- & end-of-run

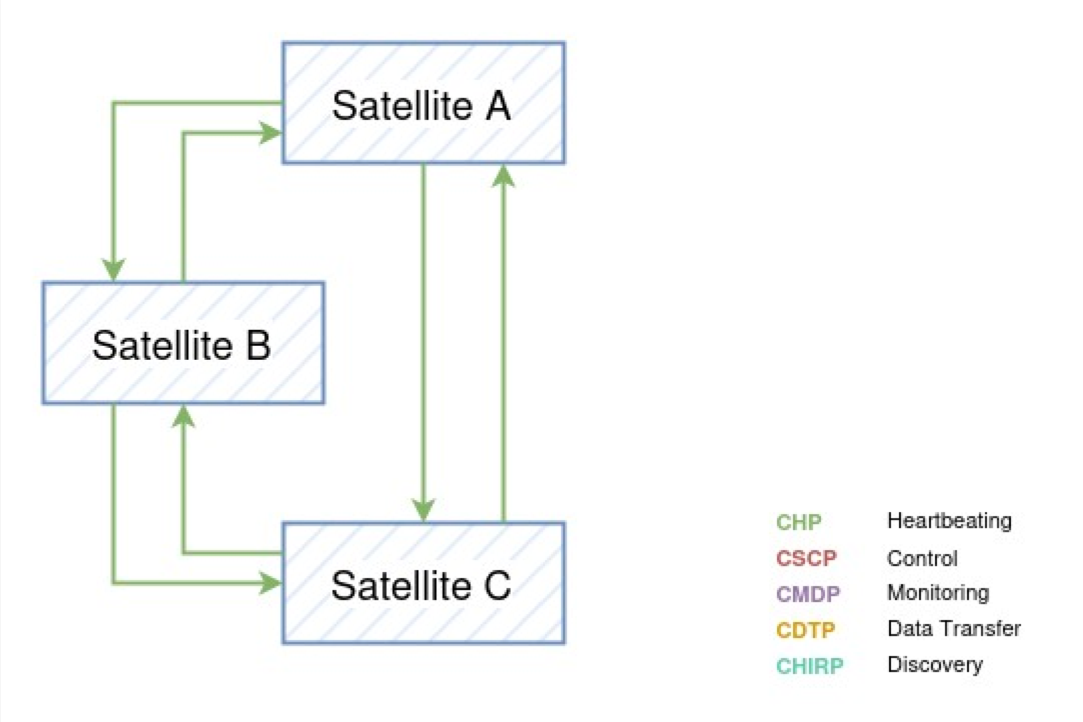
System Architecture – Network Discovery

- At startup, satellites emit CHIRP beacons
  - OFFER for each provided service
  - REQUEST for remote services
- Other satellites within the same group/domain/Constellation are discovered



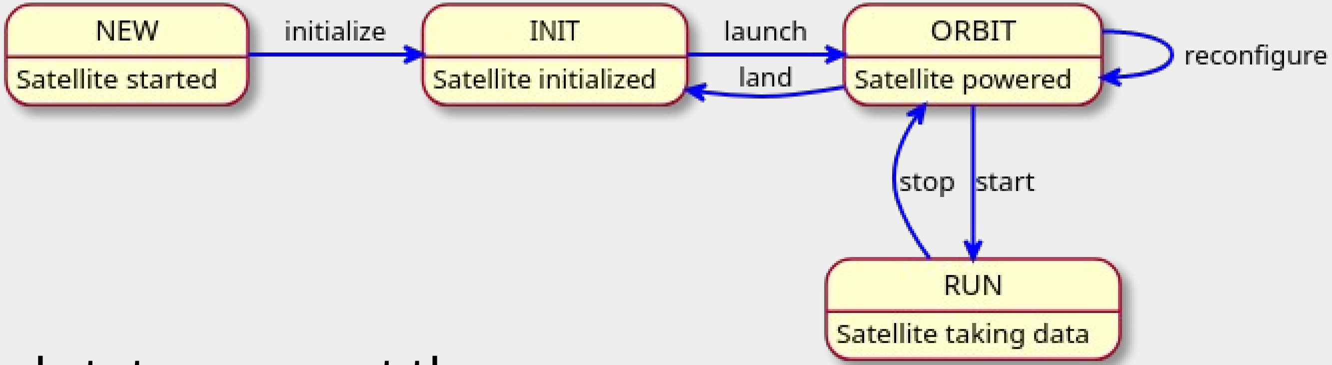
System Architecture – Heartbeating

- Satellites start exchanging heartbeat messages
- Message contains
  - Current FSM state
  - Expected interval to next message
- Heartbeating allows
  - autonomous tracking of Constellation state
  - Reaction even in case of netsplit / connection loss



The Satellite Finite State Machine

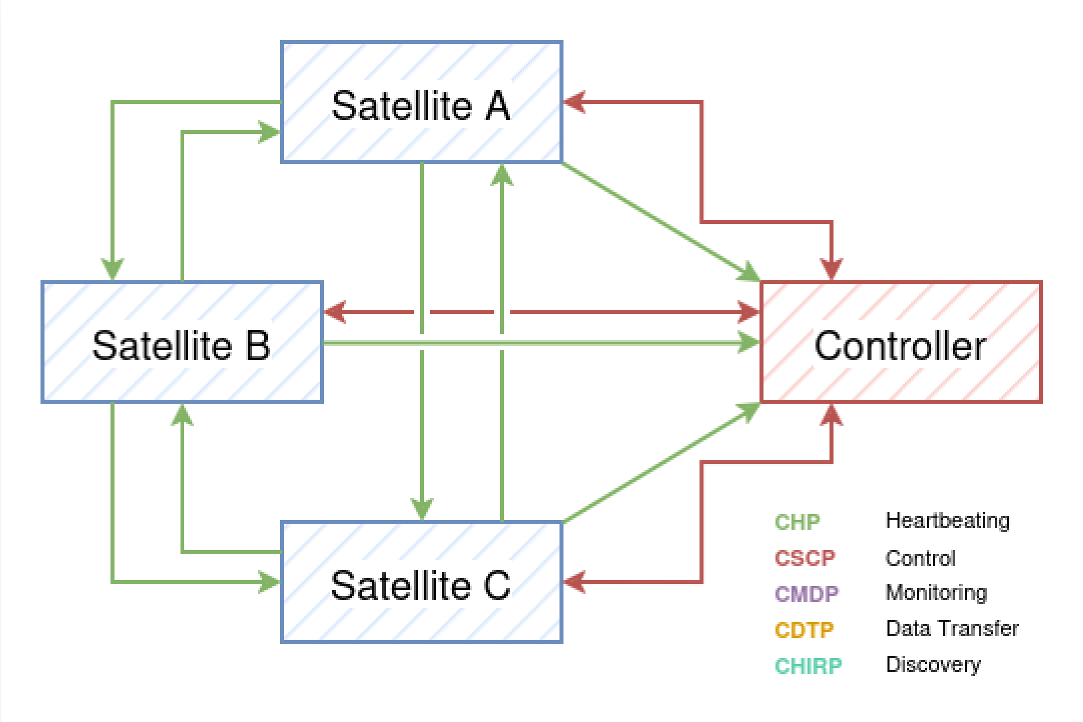
- Central component of the satellite, governing its operation
- Instruments always in a well-defined state
- Satellites know four steady states:

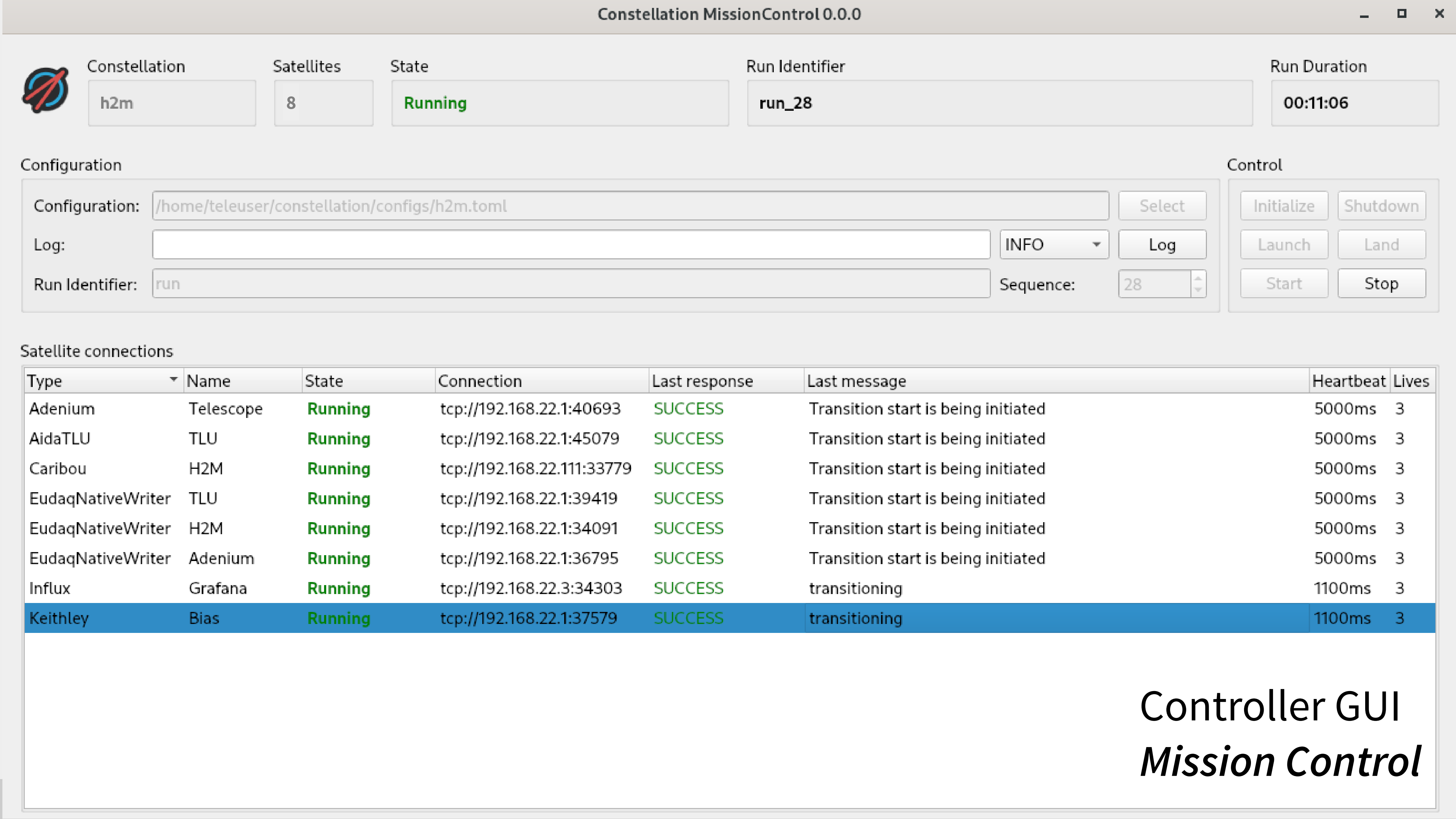


- Transitional states connect them
  commands (via CSCP) initiate transition

System Architecture – Control

- A controller joins the network
  - Uses network discovery to find satellites
  - Uses heartbeats to get state update information
- Commands sent to individual or all satellites via CSCP protocol
  - Classical server-client (request-response) communication
  - Response indicates success / errors
  - Custom per-satellite commands possible





Controller GUI

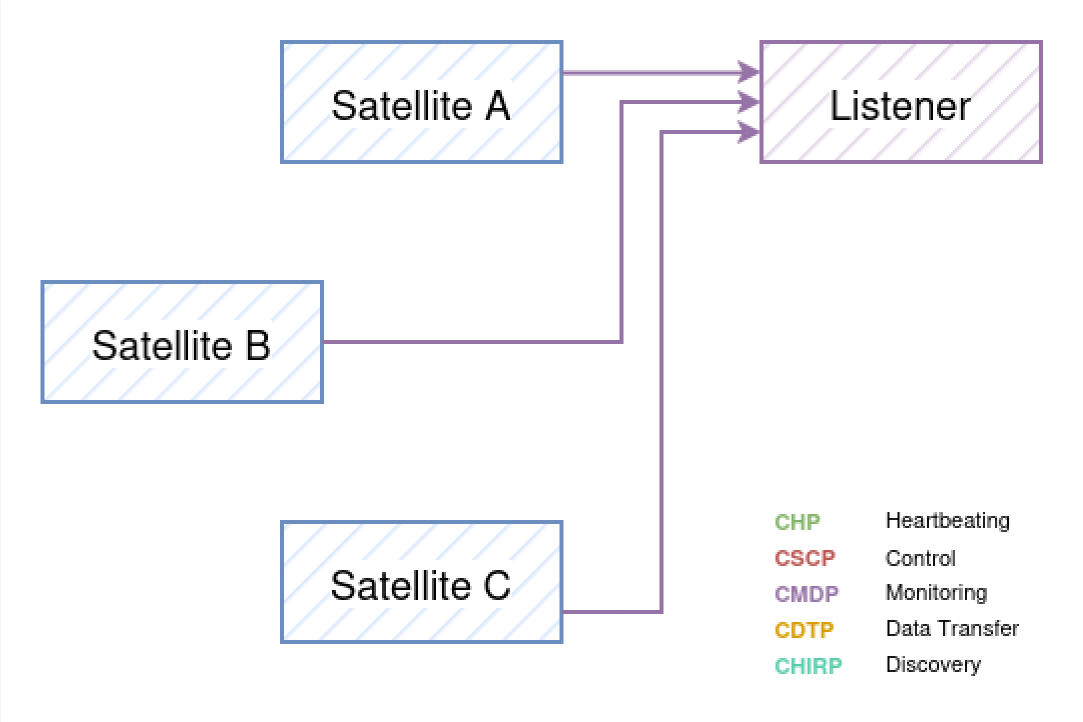
Mission Control

```
{'Sputnik.Agamemnon': SatelliteResponse(
                    msg='transition initialize is being initiated'),
 'Sputnik.Aether': SatelliteResponse(
                    msg='transition initialize is being initiated'),
 'Sputnik.Achlys': SatelliteResponse(
                    msg='transition initialize is being initiated'),
 'Sputnik.Achilleus': SatelliteResponse(
                    msg='transition initialize is being initiated')}

📡 v0.1 (Crux) 🛰4 👷 TRANSITIONING ipython
cnstln1 ❯ constellation.launch()
{'Sputnik.Agamemnon': SatelliteResponse(
                    msg='transition launch is being initiated'),
 'Sputnik.Aether': SatelliteResponse(
                    msg='transition launch is being initiated'),
 'Sputnik.Achlys': SatelliteResponse(
                    msg='transition launch is being initiated'),
 'Sputnik.Achilleus': SatelliteResponse(
                    msg='transition launch is being initiated')}

📡 v0.1 (Crux) 🛰4 👷 TRANSITIONING ipython
cnstln1 ❯ ctrl.states
{'Sputnik.Aether': <SatelliteState.ORBIT: 48>,
 'Sputnik.Agamemnon': <SatelliteState.ORBIT: 48>,
 'Sputnik.Achlys': <SatelliteState.ORBIT: 48>,
 'Sputnik.Achilleus': <SatelliteState.ORBIT: 48>}

📡 v0.1 (Crux) 🛰4 🛰ORBIT ipython
cnstln1 ❯ constellation.launch()
  failure()          get_run_id()       get_type()        interrupt()
  get_commands()     get_satellite()    get_version()     land()
  get_config()       get_state()        group             launch()         >
  get_name()         get_status()       initialize()      reconfigure()
```

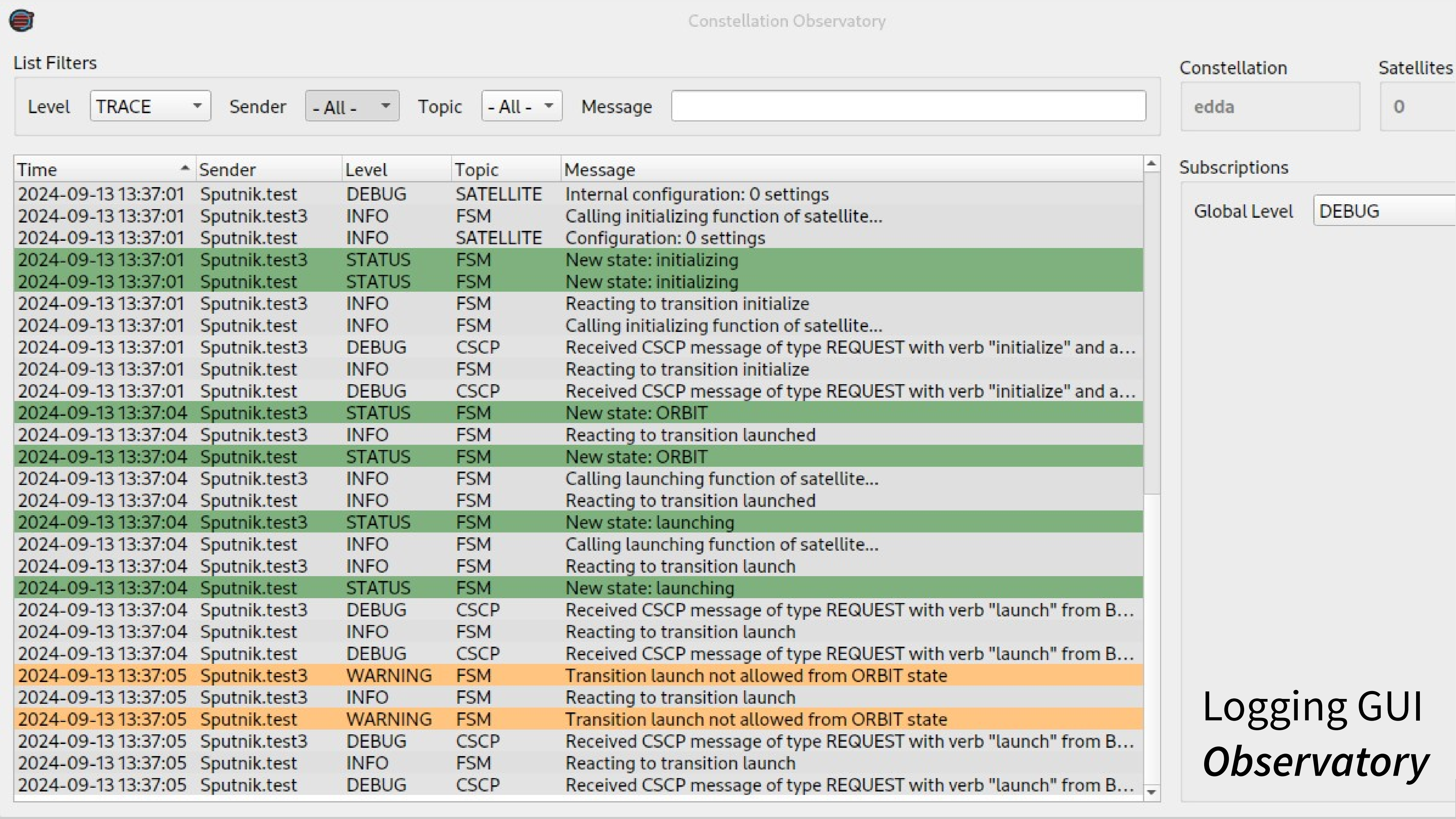
IPython-based
interactive & scriptable
CLI controller
with tab completion

System Architecture – Monitoring

- Satellites can expose metrics and log messages through monitoring protocol
  - Publish-subscribe model
  - Only information with active subscribers is actually transmitted
- Listeners are passive Constellation components which only consume CMDP



Influx satellite feeding monitoring data to InfluxDB & Grafana dashboard



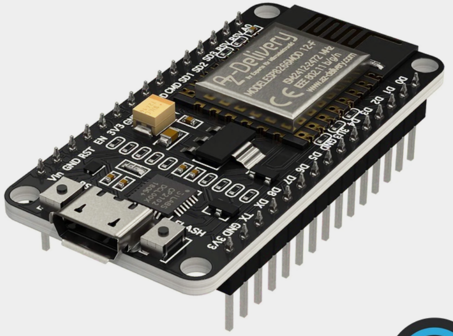
Logging GUI
Observatory

Excursus: MicroSat – A Microcontroller Satellite

- Proof-of-principle:
  - Implement a Constellation Satellite on a microcontroller
  - Had some 5 EUR, low power **ESP8266** available
- Despite limitations of microcontroller (single thread, limited resources) fully functional satellite

"...well-defined protocols means independence from implementation"

- Possible applications:
  - Controlling switches / relays
  - Providing temperature & humidity data
  - …



The Project

Application Example

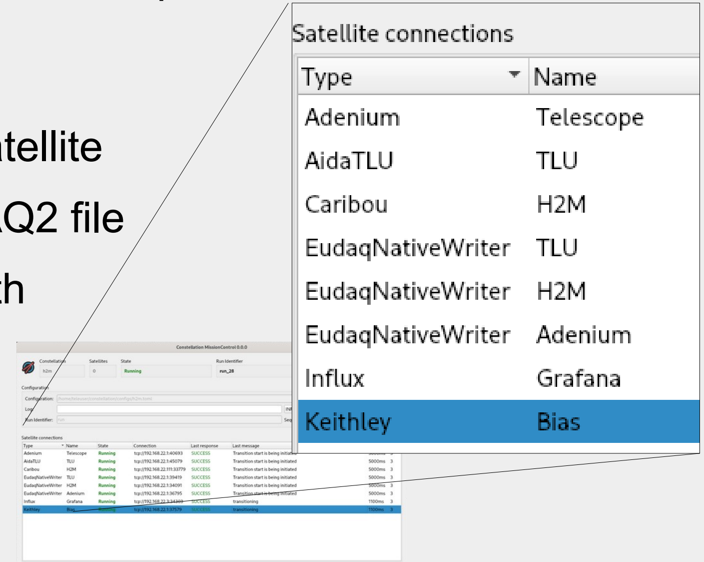
Continuous Integration

Documentation

A Real World Example – H2M Testbeam at DESY-II

Also listen to H2M talk by Finn King, Thu afternoon!

- Synchronously controlling
  - Caribou DAQ with H2M MAPS detector Satellite running on Xilinx UltraScale SoC
  - Keithley 2410 source meter
  - Adenium ALPIDE telescope
  - AIDA2020 TLU
- EudaqNativeWriter satellite to store data in EUDAQ2 file format for analysis with existing tools



Satellite connections

Type	Name
Adenium	Telescope
AidaTLU	TLU
Caribou	H2M
EudaqNativeWriter	TLU
EudaqNativeWriter	H2M
EudaqNativeWriter	Adenium
Influx	Grafana
Keithley	Bias

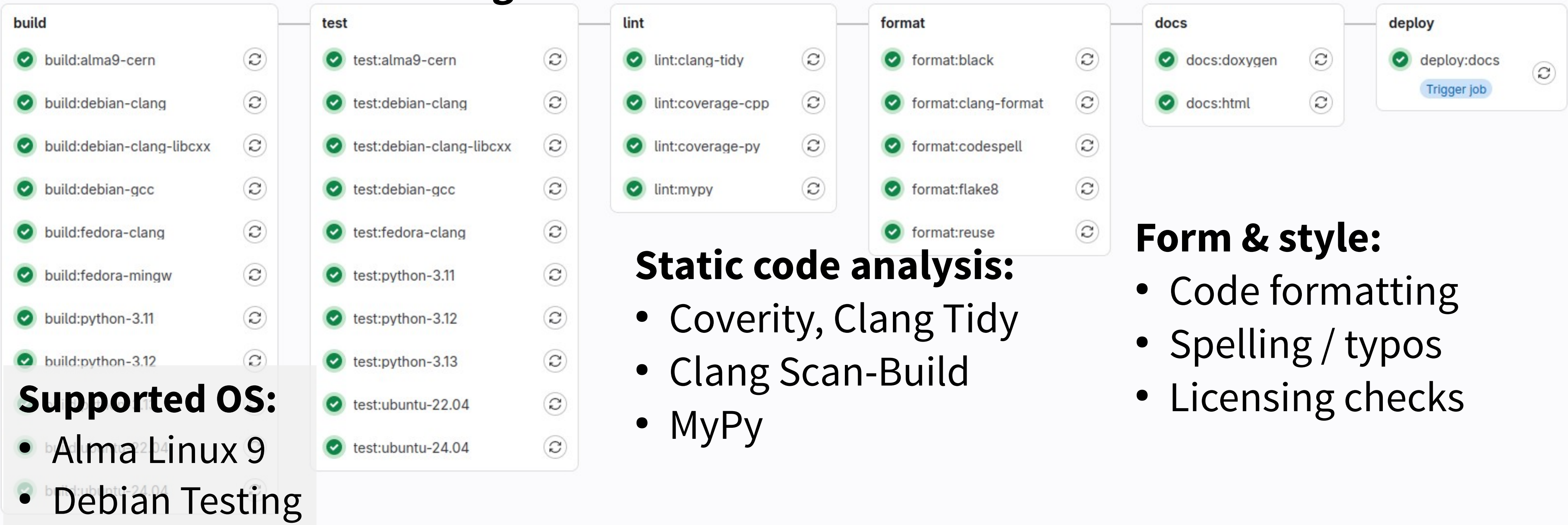
Other users (or soon-to-be):

- electronCT Testbeam @ MAMI / Mainz
- MADMAX Experiment @ Uni Hamburg
- HGTD Alvin Sr90 Setup @ CERN
- ATLAS ITk Module Testing @ Lund Uni
- Collaboration w/ TrackLab (Talk by Petr Manek, now!)
- …

Continuous Integration

Test coverage: ~ 90%

Documentation



Static code analysis:
- Coverity, Clang Tidy
- Clang Scan-Build
- MyPy

Form & style:
- Code formatting
- Spelling / typos
- Licensing checks

Supported OS:
- Alma Linux 9
- Debian Testing
- Fedora 40
- Ubuntu 22.04
- Ubuntu 24.04

Constellation News User Guide Satellites Developer Guide Search Ctrl + K

Constellation

Autonomous Control and Data Acquisition System

Constellation is a control and data acquisition system for small-scale experiments and experimental setup with volatile and dynamic constituents such as testbeam environments or laboratory test stands.

Get Started Concepts

See API Reference →

Autonomous

Constellation operates without a central server, satellites exchange heartbeats to keep in touch.

Flexible

Automatic network discovery of satellites make it easy to add and remove satellites on the fly.

Fast Integration

The finite state machine and satellite interface are designed for fast and easy integration of devices.

Robust

Constellation is based on widely adopted networking libraries such as ZMQ and MsgPack.

Website & Documentation

https://constellation.pages.desy.de

Summary & Outlook

- Implementing a **new control & DAQ system** bottom-up

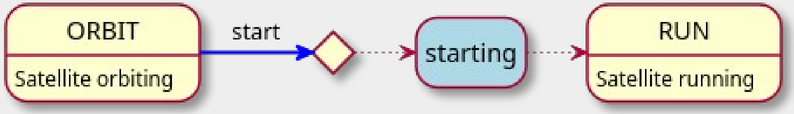
  Centered system design around user stories & protocol definitions

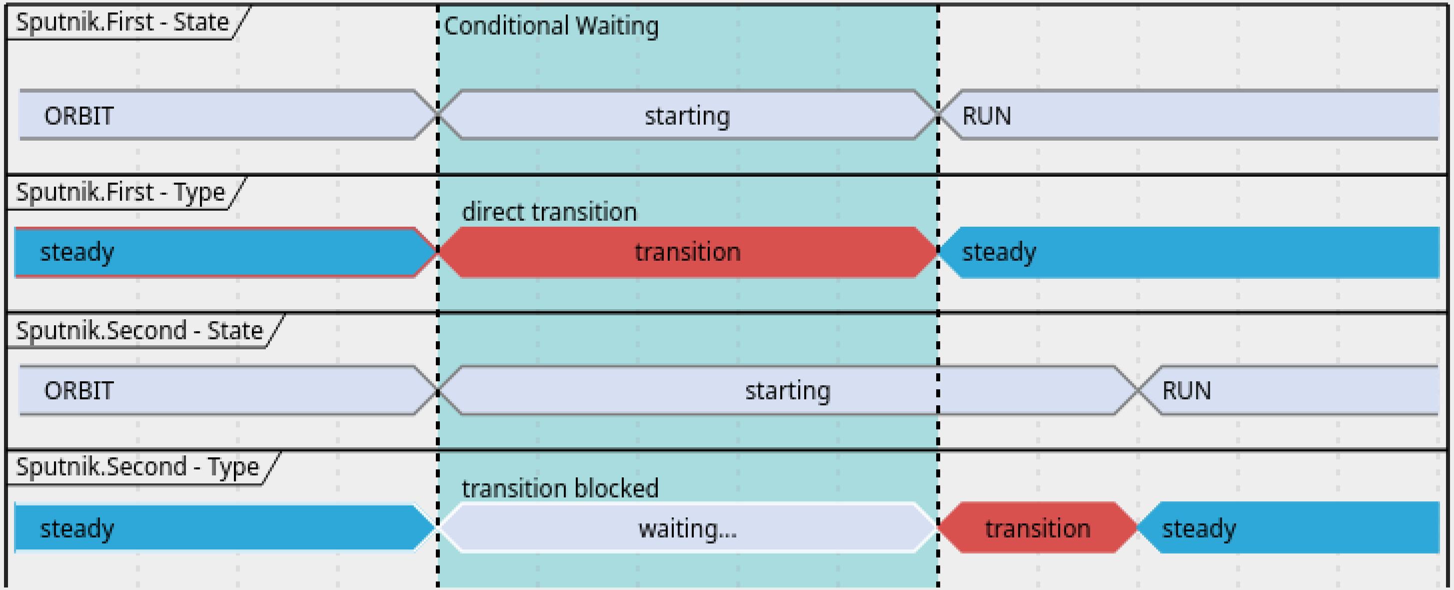
- Most **core concepts and features are implemented**
  - Protocol implementations in C++ & Python
  - Graphical user interfaces for controlling & logging
  - Monitoring via Grafana, data storage in HDF5 files
- First preview ***Constellation 0.1 Crux*** released, second to follow ~ this week
- Get involved as developer, as adopter, as user – join us:
  - EDDA mailing list at lists.desy.de/sympa/info/edda
  - Website with documentation at constellation.pages.desy.de
  - CERN Mattermost team at mattermost.web.cern.ch/constellation
  - Code is open source, EUPL-licensed and available at gitlab.desy.de/constellation

DESY.

Excursus: Autonomous Transitions Orchestration

- Satellites can operate autonomously, no controller has to be active
- Satellites can control certain processes themselves without active intervention
- Example: specific order of start/stop operation:





A Standard Interface for Satellites

- Converged on a standard set of commands any satellite has to understand

  Each satellite must be able to understand and answer to the following commands, and it must accept or provide the corresponding payloads. Verbs and commands are always transmitted as native strings, payloads are always encoded as MsgPack objects.

- Documented as
  Satellite Implementation Guidelines
- Forms the basis for controller classes and UIs

Command	payload	verb reply	payload reply
`get_name`	-	Name of the Satellite	-
`get_version`	-	Constellation version identifier string	-
`get_commands`	-	Acknowledgement	List of commands as MsgPack map/dictionary with command names as keys and descriptions as values
`get_state`	-	Current state (as string)	-
`get_status`	-	Current status	-
`get_config`	-	Acknowledgement	Satellite configuration as flat MsgPack map/dictionary
`get_run_id`	-	Current or last run identifier (as string)	-
`initialize`	Satellite configuration as flat MsgPack map/ dictionary	Acknowledgement	-
`launch`	-	Acknowledgement	-
`land`	-	Acknowledgement	-
`reconfigure`	Partial configuration as flat MsgPack map/ dictionary	Acknowledgement	-
`start`	Run identifier as MsgPack string	Acknowledgement	-
`stop`	-	Acknowledgement	-
`shutdown`	-	Acknowledgement	-