Constellation

A flexible control and data acquisition framework for dynamic small-scale experiments

Stephan Lachnit for the EDDA collaboration
HELIOS Lund Retreat 2025

HELMHOLTZ

About me

- PhD student at DESY / U Bonn
- Master thesis @ DESY / UHH:
  Characterization of a digital SiPM
- Studied Swedish @ LU for half a year
- PhD thesis:
  Characterization of a Monolithic Active Pixel Sensor &
  Development of a flexible DAQ framework

I will talk about this today

Content

1. Brief Introduction of Constellation

2. Installation & Tutorial

3. Protocols & Applications

4. Implementation of Temperature Readout

Introduction

Building a Small Experiment from Scratch

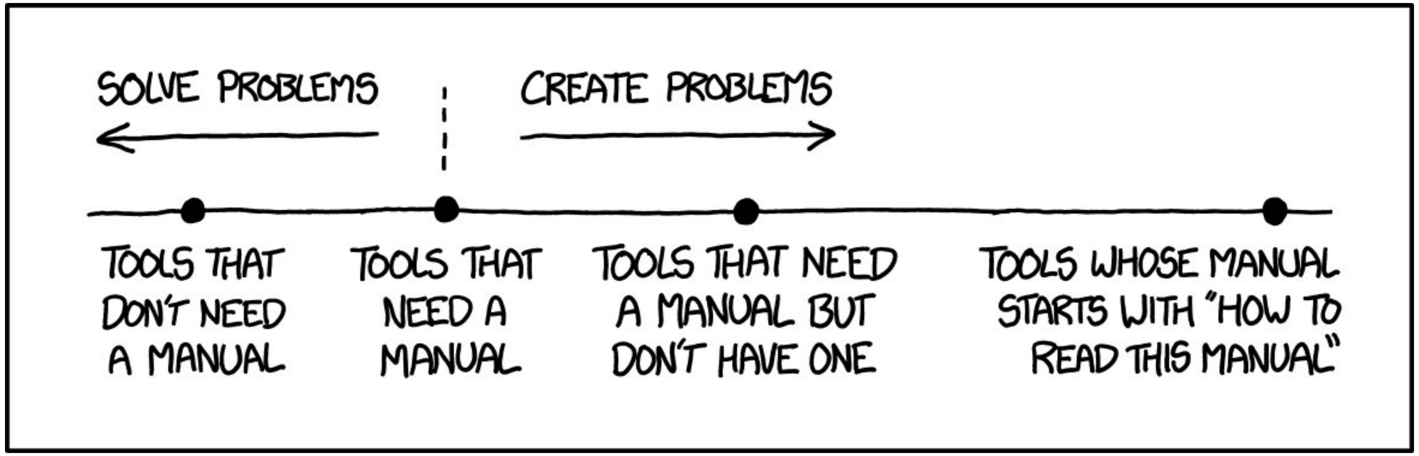
- Example setup:
  - Laser setup with detector
  - Scan over externally applied voltage
  - Temperature monitoring
- Nice-to-have DAQ software features:
  - Comprehensive logging system for debugging
  - Online monitoring of the temperature
  - Notification in case of crashes
  - File format with metadata such as HDF5
  - Easily adjustable in case of setup changes

TODO Sketch

Building a Small Experiment from Scratch

The problem

- **Time is limited**
- This often results in:
  - Lack of good documentation
  - No testing for corner cases
  - Not flexible to setup changes
  - No “fancy” features like a proper monitoring dashboard or failure notification
- Re-implementation of many features feels unnecessary
- Need for a common framework which implements most of the features



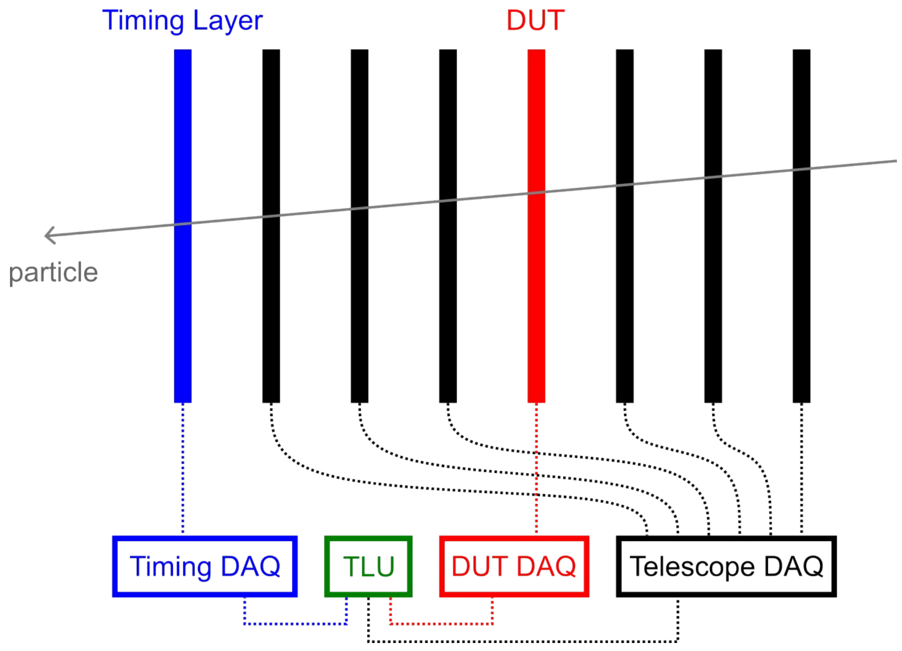
https://xkcd.com/1343/

Test Beam

How to test new particle detectors

- Different devices required:
  - Device Under Test (**DUT**)
  - Beam **telescope** for tracking
  - Trigger Logic Unit (**TLU**) to distribute triggers
  - Timing Layer
- Different DAQ systems for each device
- Different telescopes and timing layer at other facilities

Requires control software for synchronous operation
to configure, start and stop the data acquisition over the network



Setup for the H2M test beam at DESY

Introducing Constellation

In 30 seconds

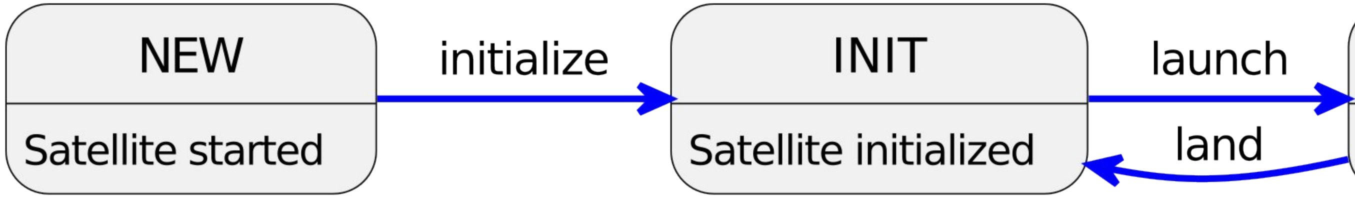
- Autonomous Control and Data Acquisition Framework for dynamic experimental setups
- Participants in the data acquisition are called **satellites**
  - Operation governed by a Finite State Machine (**FSM**)
  - Operate **autonomously** (no central point of failure)
- Network communication defined in RFC-style protocols
- Independent implementations in C++ and Python
- **Provides interfaces for control, configuration, logging, monitoring and data transmission**

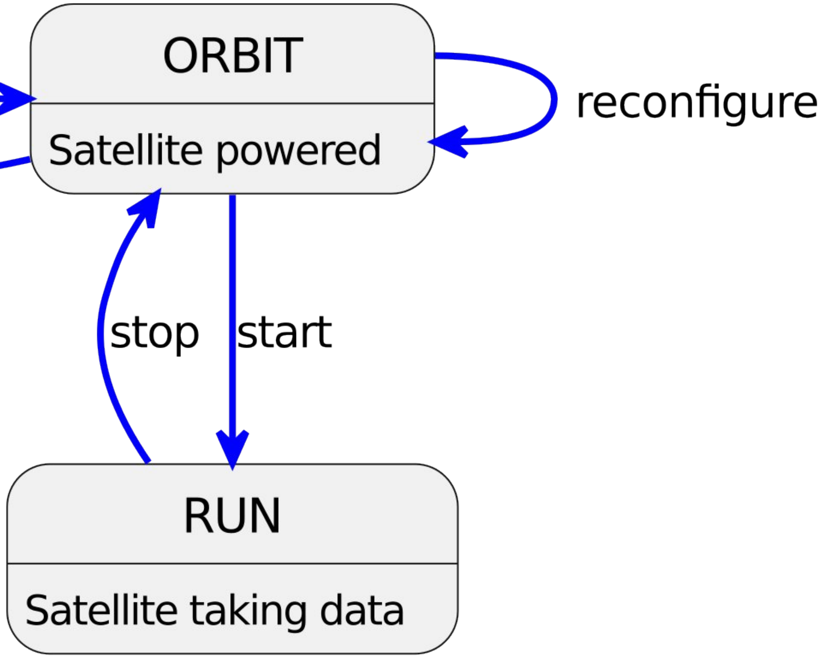
Documentation:

https://constellation.pages.desy.de/

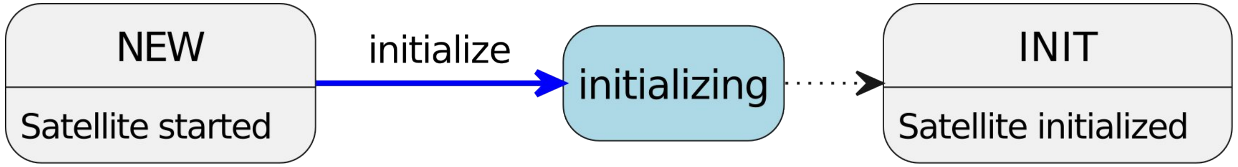
Satellite Finite State Machine

- Device is always in well-defined state





- Each transitions has to be implemented for the device
- Transitions have their own states for feedback on long operations



- Additional SAFE and ERROR states for error handling

Installation & Tutorial

Set up Raspberry Pi (again)

We need to set up the Raspberry Pi again

- Choose `Raspberry Pi OS (64-bit)` this time
- Set up hostname, user, WLAN and SSH as before
- Connect via SSH
- Enable VNC, SPI, I2C and GPIO as before
- Connect via VNC

Update the system (no need to install anything else yet)

- `sudo apt update`
- `sudo apt full-upgrade -y`

Python Installation

Install Python virtual environment

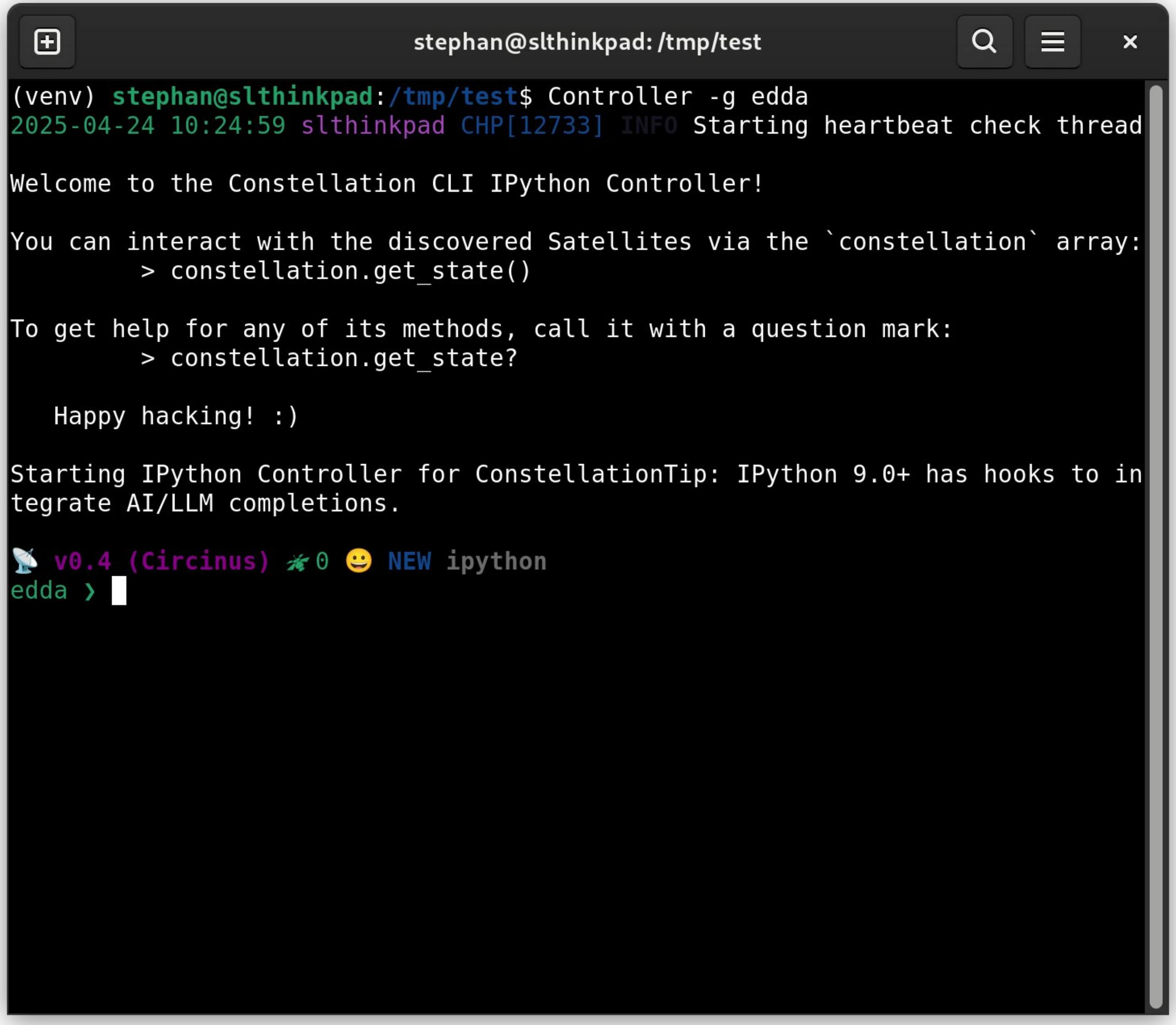
- `sudo apt install -y python3-venv`

Install Constellation in virtual environment

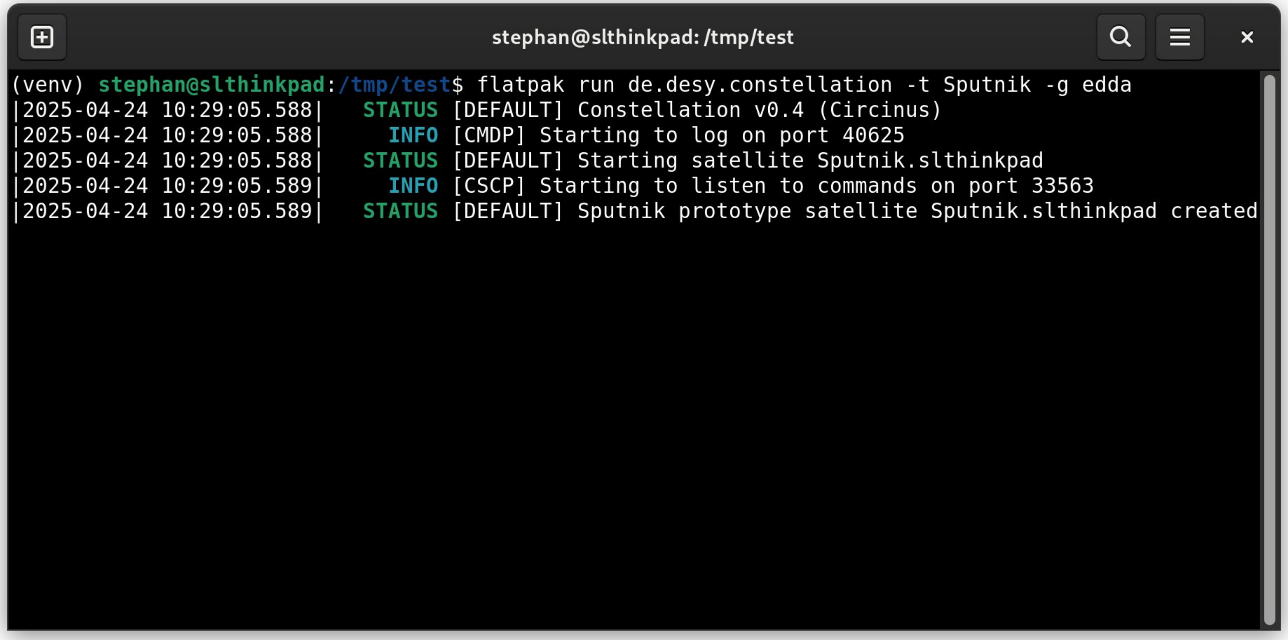
- *Change into a new project folder*
- `python3 -m venv venv`
- `source venv/bin/activate`
- `pip install ConstellationDAQ[cli]`
- `pip install RPi.GPIO smbus2`

Verify Installation

- `Controller -g edda`



C++ Installation



Install flatpak

- `sudo apt update`
- `sudo apt install -y flatpak`

Install Constellation

- `flatpak remote-add --if-not-exists flathub https://dl.flathub.org/repo/flathub.flatpakrepo`
- `flatpak install flathub de.desy.constellation`
- *Reboot*

Verify Installation

- `flatpak run de.desy.constellation -t Sputnik -g edda`

Controlling a Single Satellite

Using the Python controller (command line interface)

- Let's start by starting a controlling a single satellite using an interactive Python shell
- Follow this tutorial:
  https://constellation.pages.desy.de/operator_guide/tutorials/single_satellite.html
- Since the C++ version containing the Sputnik satellite is not installed from source,
  replace `s/SatelliteSpuntik/flatpak run de.desy.constellation -t Sputnik/g`
  or `s/SatelliteSpuntik/SatelliteMariner/g`

TODO group

TODO Screenshot

Controlling Multiple Satellites

Using MissionControl (graphical user interface)

- Let’s continue with a more advanced setup using multiple satellites
- Follow this tutorial:
  https://constellation.pages.desy.de/operator_guide/tutorials/missioncontrol.html
- Since the C++ version is not installed from source, replace
  `s/SatelliteXYZ/flatpak run de.desy.constellation -t XYZ/g`
  `s/MissionControl/flatpak run --command=MissionControl de.desy.constellation/g`

TODO group

TODO Screenshot

Logging of Multiple Satellites

Using Observatory (graphical user interface)

- Let’s continue with a more advanced setup using multiple satellites
- Follow this tutorial: https://constellation.pages.desy.de/operator_guide/tutorials/observatory.html
- Since the C++ version is not installed from source, replace

```
s/SatelliteXYZ/flatpak run de.desy.constellation -t XYZ/g
s/MissionControl/flatpak run --command=MissionControl de.desy.constellation/g
s/Observatory/flatpak run --command=Observatory de.desy.constellation/g
```

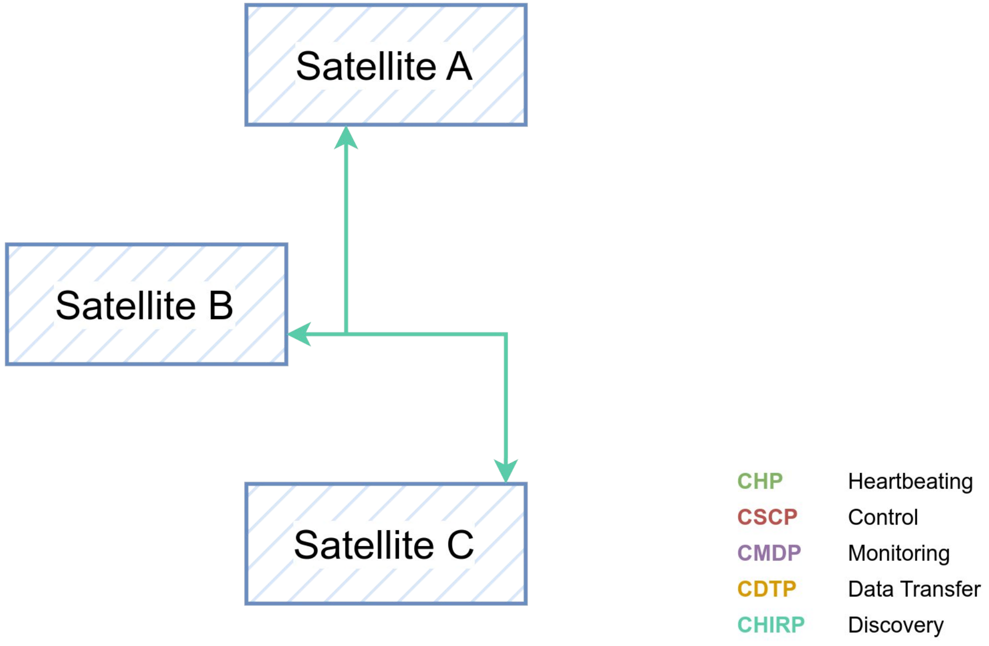
TODO group

TODO Screenshot

Protocols

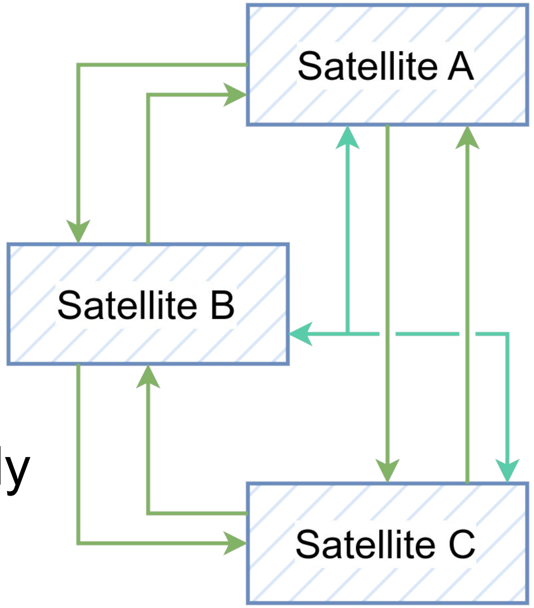
Network Discovery

- Satellites in the local network are discovered automatically
- Advantages:
  - No to assign fixed IP addresses
  - Simpler setup procedure
  - Allows for autonomy in the system
- Uses UDP broadcasts:
  - OFFER for provided services
  - REQUEST to search for remote services



Autonomy

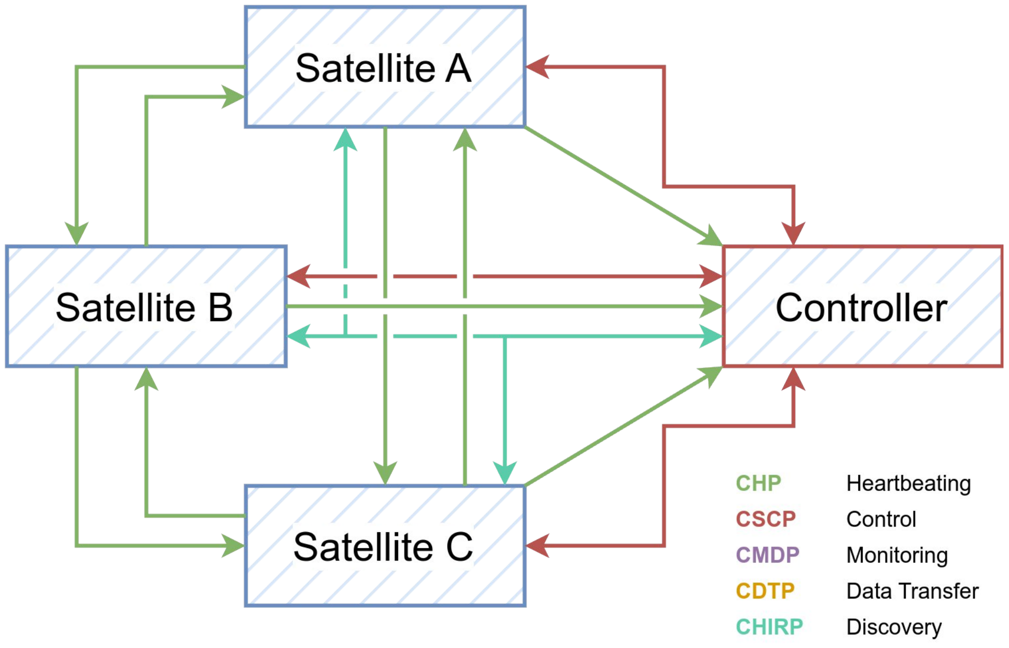
- Satellites emit regular “**heartbeats**”
- Heartbeats contain the current state of the satellite
- Heartbeating can be used for:
  - Stateless monitoring of the network
  - Detecting dead satellites via missing heartbeats
  - Trigger transition to safe operating mode autonomously in case of a failure in other satellites



CHP	Heartbeating
CSCP	Control
CMDP	Monitoring
CDTP	Data Transfer
CHIRP	Discovery

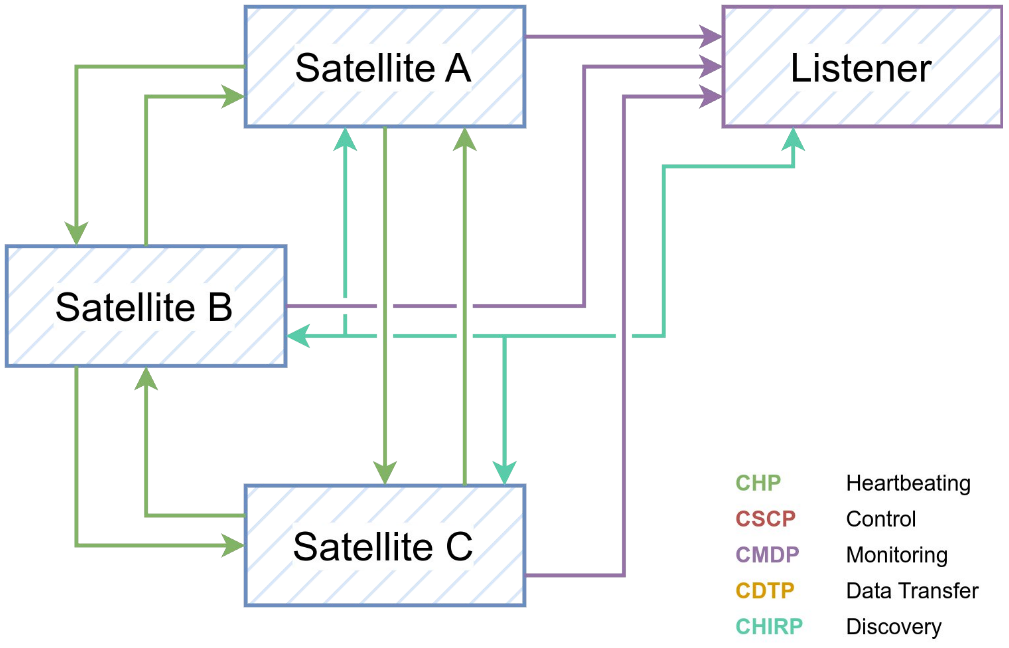
Control

- A controller joins the network:
  - Uses network discovery to find satellites
  - Connects to receive heartbeats
  - Uses heartbeats to monitor the state
  - Sends commands to a single or all satellites
- Client-server pattern for commands
- Controllers are stateless
- Custom commands are possible
  e.g. read register values



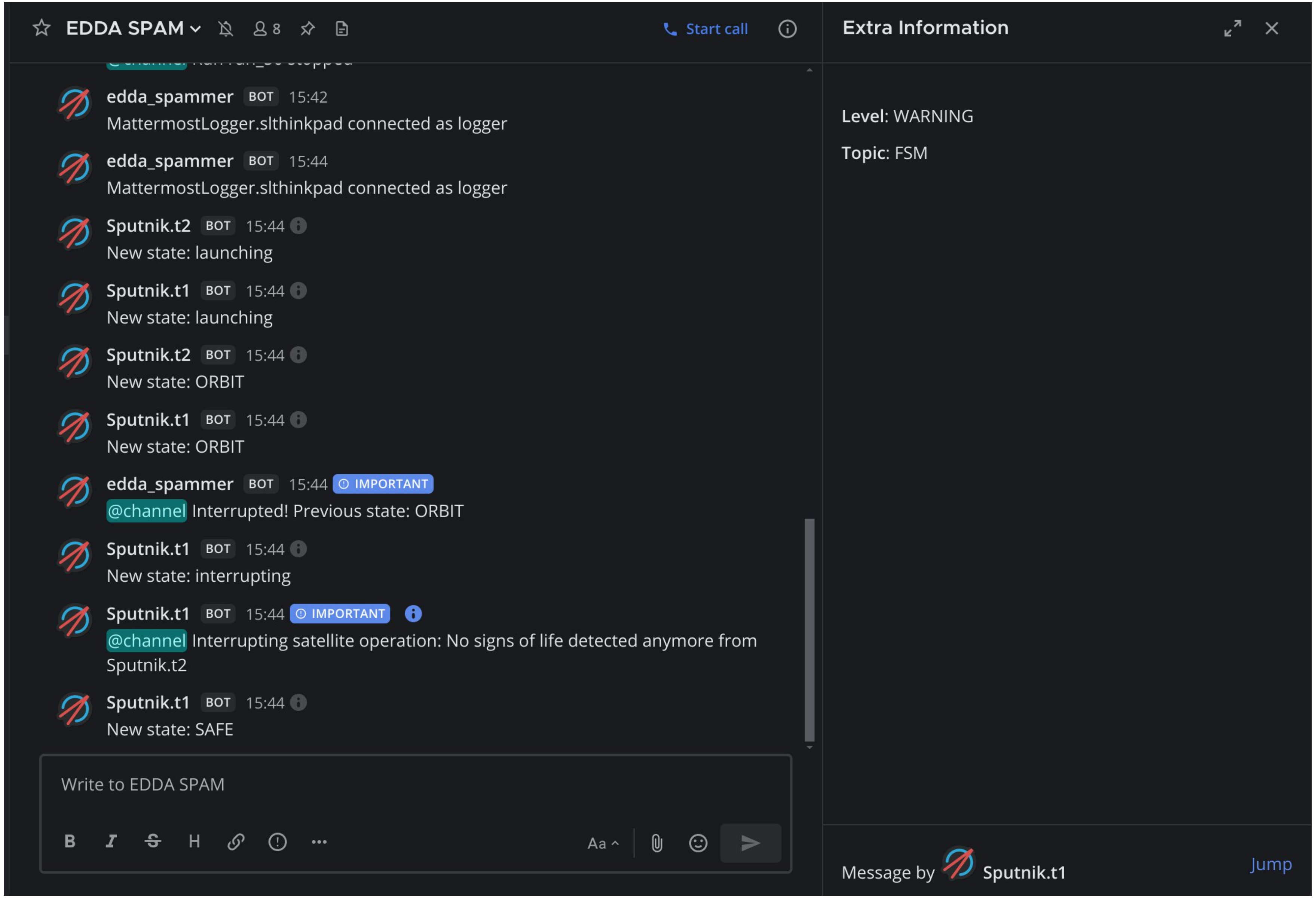
Monitoring

- Satellites can publish telemetry and log messages
- A monitoring listener joins the network:
  - Uses network discovery to find satellites
  - Subscribes to relevant log messages or telemetry
- Publish-subscribe pattern
- Only subscribed monitoring data is actually transmitted
- Simple integration into third-party software like Grafana or Mattermost



Online Monitoring with Grafana

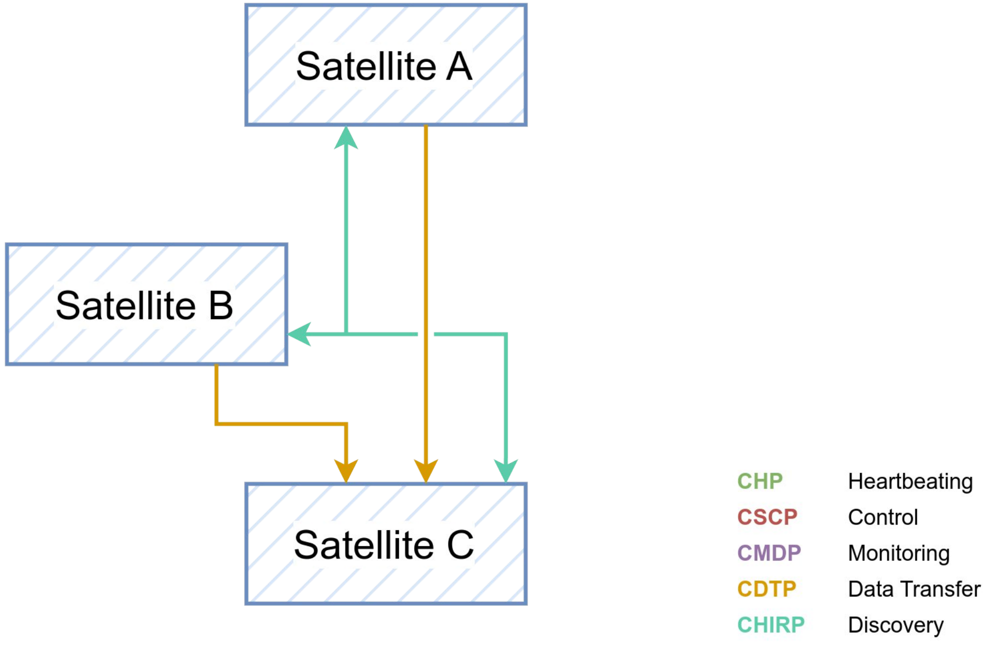
Logging to Mattermost



Data Transmission

- Sending data over the network can be useful:
  - Embedded system (e.g. Caribou)
  - Store all data in one place
  - Flexible Support for file format
    - EUDAQ2
    - HDF5

Caribou DAQ system used for H2M

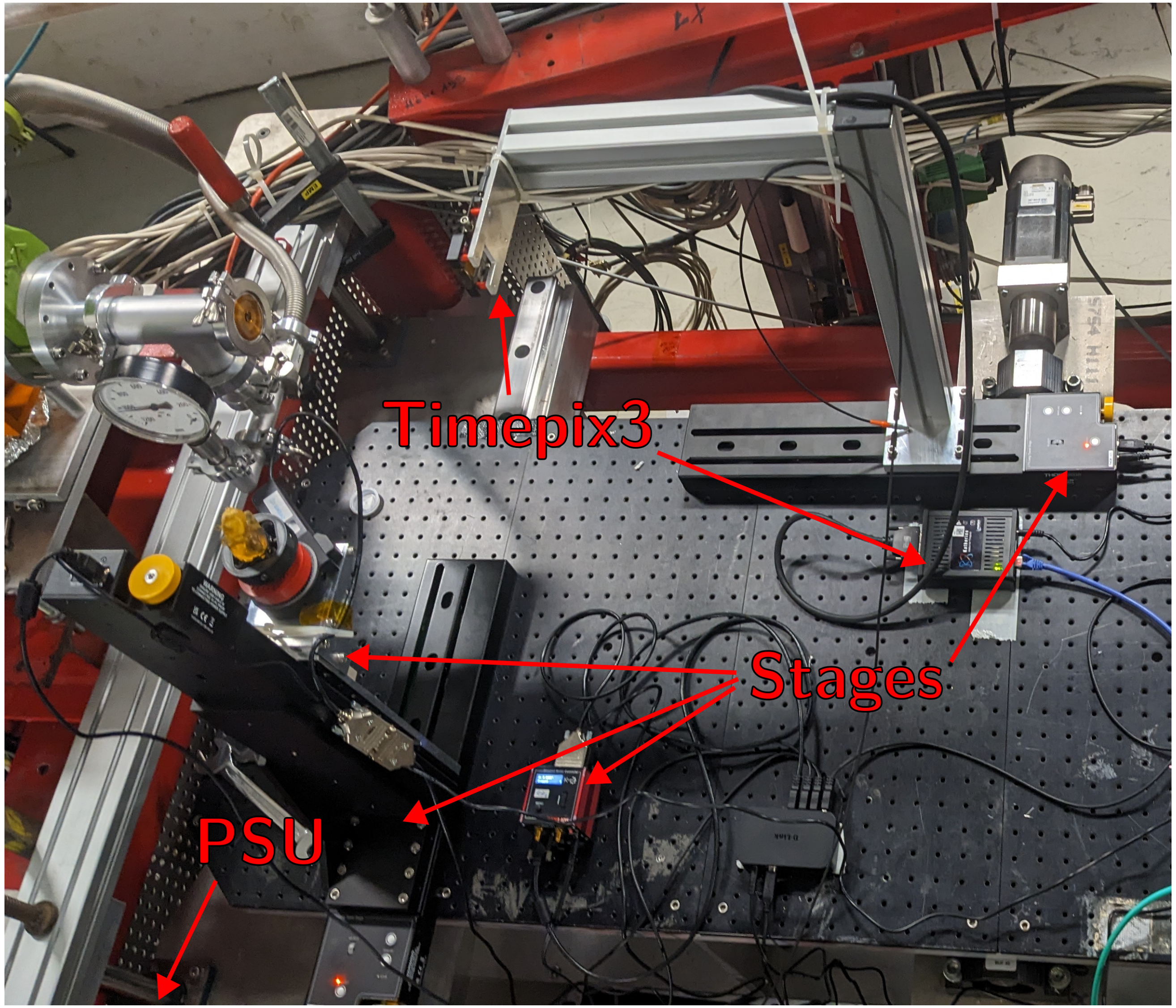


Applications

electronCT Test Beam

First Production Use @ MAMI

- Timepix3
- Keithley 2410 SMU
- Custom 4D stage
- HDF5 for data storage



H2M Test Beam

Evaluation Test @ DESY

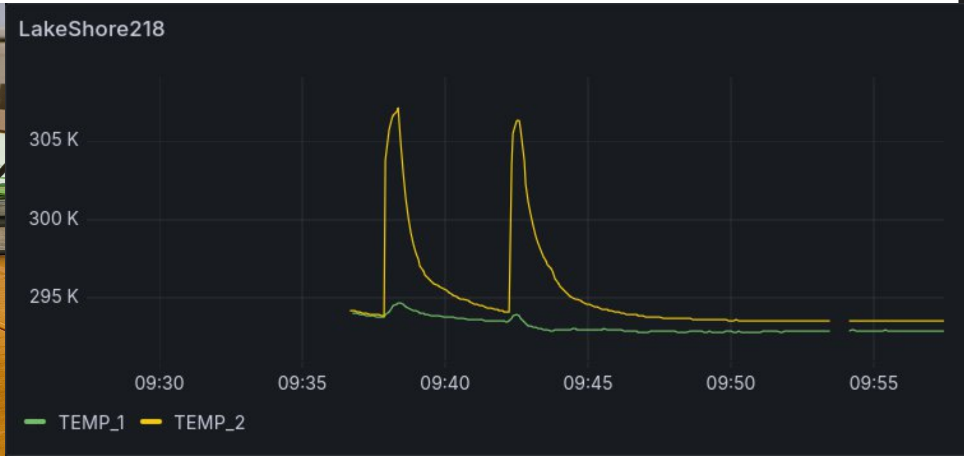
- Caribou DAQ system with H2M running Constellation
- Adenium beam telescope
- AIDA2020 TLU
- Keithley 2410 SMU
- EUDA2 for data storage

MADMAX Temperature Monitoring

Dark Matter Experiment @ UHH

- 8-channel temperature monitor for MADMAX cryostat
- Communication via serial port
- Implementation in <100 LoC

```python
class LakeShore218(Satellite):
    _serial: pyvisa.resources.SerialInstrument

    def __init__(self, *args, **kwargs):
        self._rm = pyvisa.ResourceManager()
        self._lock = Lock()
        super().__init__(*args, **kwargs)

    def do_initializing(self, config: Configuration) -> None:
        port = config["port"]
        visa_address = f"ASRL{port}::INSTR"
        self.log.debug("Opening VISA device %s", visa_address)

        self._serial = self._rm.open_resource(  # type: ignore
            visa_address,
            baud_rate=9600,
            data_bits=7,
            stop_bits=pyvisa.constants.StopBits.one,
            parity=pyvisa.constants.Parity.odd,
            read_termination="\r\n",
            write_termination="\r\n",
        )

    def _get_temp(self, channel: int) -> float | None:
        # Check that _serial exists (required for metrics before INIT)
        if not hasattr(self, "_serial"):
            return None
        # Lock required since serial not thread safe
        with self._lock:
            # Check if input is enabled (return "0" or "1")
            if self._serial.query(f"INPUT? {channel}") == "1":
                return float(self._serial.query(f"KRDG? {channel}"))
        # Return None if not enabled
        return None

    @cscp_requestable
    def get_temp(self, request: CSCPMessage) -> tuple[str, Any, dict]:
        channel = int(request.payload)
        if channel < 1 or channel > 8:
            raise Exception(f"Channel {channel} does not exist")
        temp = self._get_temp(channel)
        verb = f"{temp}K" if temp else "Disabled"
        return verb, temp, {}

    @schedule_metric("K", MetricsType.LAST_VALUE, 5)
    def temp_1(self) -> Any:
        return self._get_temp(1)
```



Implementation

Implementing a Temperature Monitor

Your turn

- Goal: implement a satellite which provides the temperature as metrics
- Additional features:
  - Configurable measurement interval
  - Configurable metric name
  - Temperature available as custom command
  - Configurable critical threshold:
    - Log to critical when threshold is reached
    - Turn on LED if threshold is reached
    - Go to ERROR state
  - (advanced) Implement PID temperature control

Implementing a Temperature Monitor

Getting started

- Ensure you are in the project folder with the Python venv
- Create a new Python file, e.g. `pitemp_satellite.py`
- Import the relevant classes from the constellation module:

```
from constellation.core.configuration import Configuration
from constellation.core.satellite import Satellite
```

- Check that the imports work correctly by running the file, e.g.:

```
python3 pitemp_satellite.py
```

Implementing a Temperature Monitor

Creating a dummy satellite

- Create a new satellite class:

```
class PiTemp(Satellite):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def do_initializing(self, config: Configuration) -> None:
        pass
```

- `__init__` is run when the satellite is created => should not initialize the hardware!
- `do_initializing` is run when the config is received from the controller => initialize hardware

Implementing a Temperature Monitor

Creating the starting script

- Extra code is required to setup the CLI parsing and logging as well as staring the satellite
- Create a new file called `main.py`:

```
from constellation.core.logging import setup_cli_logging
from constellation.core.satellite import SatelliteArgumentParser
from pitemp_satellite import PiTemp

def main(args=None):
    parser = SatelliteArgumentParser()
    args = vars(parser.parse_args(args))
    setup_cli_logging(args.pop("log_level"))
    s = PiTemp(**args)
    s.run_satellite()

if __name__ == "__main__":
    main()
```

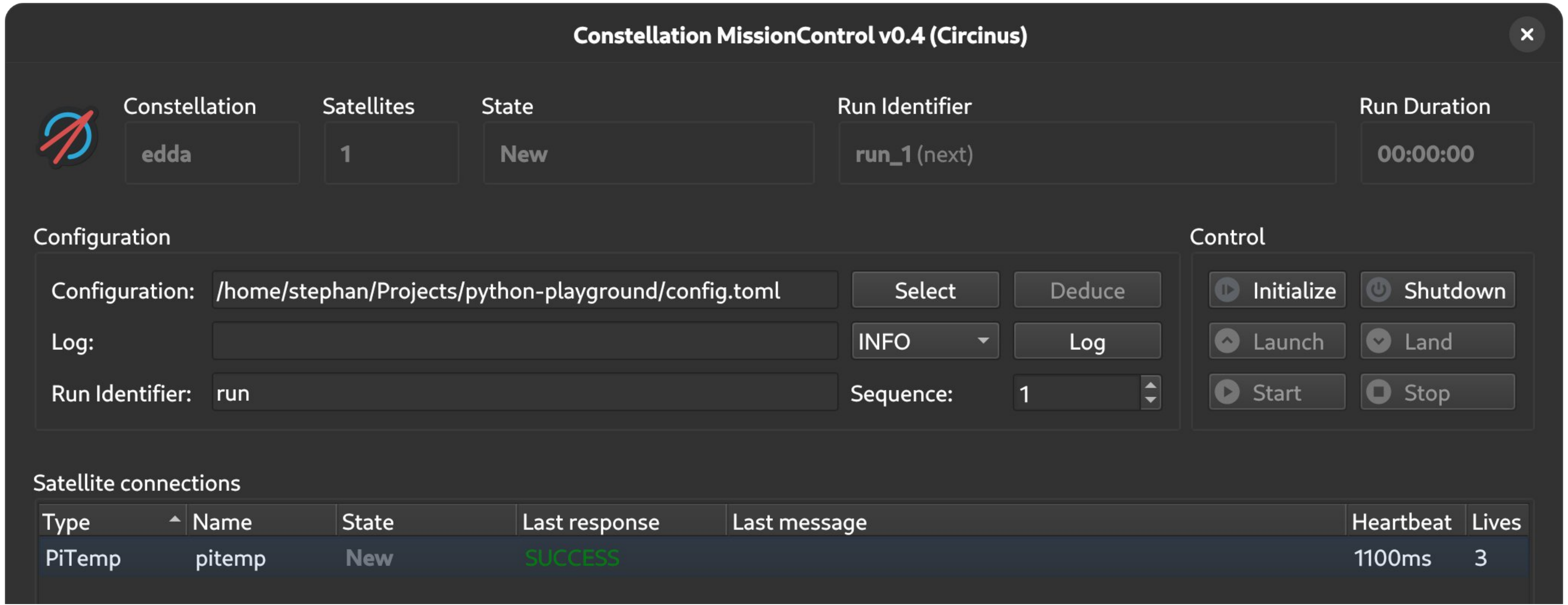
Implementing a Temperature Monitor

Starting the satellite

- Start the satellite:

```
python3 main.py -g group_name
```

- **Reminder**: since everyone is in a common subnet, use a unique group name to avoid controlling satellites from some else
- Open MissionControl and and control the satellite



Implementing a Temperature Monitor

Adding temperature reading code

- Add the `read_temperature()` function from the previous lab course
- You need to copy the `adc_module.py` and `i2c_module.py` files to the project folder
- For now, let's log the temperature in the RUN state by adding a `do_run` function to the satellite:

```
def do_run(self, payload) -> None:
    while not self._state_thread_evt.is_set():
        self.log.info(f'Current temperature: {self.read_temperature()}°C')
        time.sleep(1)do_run()
```

- Open MissionControl and Observatory and initialize and launch the satellite

Time	Sender	Level	Topic	Message
2025-05-02 16:29:23	PiTemp.slthin...	STATUS	FSM	State transition to steady state completed (stopping -> ORBIT).
2025-05-02 16:29:23	PiTemp.slthin...	STATUS	FSM	State transition stop initiated.
2025-05-02 16:29:22	PiTemp.slthin...	INFO	PITEMP	Current temperature: 20°C
2025-05-02 16:29:21	PiTemp.slthin...	INFO	PITEMP	Current temperature: 24°C
2025-05-02 16:29:20	PiTemp.slthin...	INFO	PITEMP	Current temperature: 25°C
2025-05-02 16:29:19	PiTemp.slthin...	INFO	PITEMP	Current temperature: 19°C
2025-05-02 16:29:18	PiTemp.slthin...	INFO	PITEMP	Current temperature: 20°C
2025-05-02 16:29:17	PiTemp.slthin...	INFO	PITEMP	Current temperature: 20°C
2025-05-02 16:29:16	PiTemp.slthin...	INFO	PITEMP	Current temperature: 22°C
2025-05-02 16:29:16	PiTemp.slthin...	STATUS	FSM	State transition start initiated.
2025-05-02 16:29:16	PiTemp.slthin...	STATUS	SATELLITE	Starting run 'run_1'

Implementing a Temperature Monitor

Implementing temperature as custom command

- Let’s read implement a custom command to read the temperature by adding a new function:

```
from constellation.core.commandmanager import cscp_requestable
from constellation.core.cscp import CSCPMessage

    @cscp_requestable
    def get_temp(self, request: CSCPMessage):
        """Get the current temperature"""
        temp = self.read_temperature() # Numeric response
        verb = f"{temp}°C" # Human-readable response
       return verb, temp, {}
```

- Restart and try the custom command with MissionControl
- To use parameters, use `request.payload`, which is a list of parameters, e.g.:

```
channel = int(request.payload[0])
```

Implementing a Temperature Monitor

Implementing metrics

- Let’s schedule a metric for the temperature in `do_initializing`:

```
from constellation.core.cmdp import MetricsType

    def do_initializing(self, config: Configuration) -> None:
        sampling_interval = 1 # in seconds
        self.schedule_metric('NAME', 'UNIT', MetricsType.LAST_VALUE,
                             sampling_interval, self.read_temperature)
```

- Note: any existing metrics with the same name are overwritten
- If the metric name is configurable, the metrics need to be reset before:

```
self.reset_scheduled_metrics()
```

- Verify the satellite initializes (we will not see metrics in action yet)

Implementing a Temperature Monitor

Reading the configuration

- Let's read the configuration in `do_initializing`:

```
def do_initializing(self, config: Configuration) -> None:
    value1 = config['param1'] # Satellite goes to ERROR if not present
    value2 = config.setdefault('param2', 3.14) # Uses default if not present
    self.log.info(f'param1 = {value1} and param2 = {value2}')
```

- If the config is inconsistent or hardware initialization fails, simply raising an exception is sufficient to correctly go to the ERROR state (e.g. `raise Exception('reason')`)
- Goal: implement `channel_name` and `sampling_interval` configuration parameters
- Don't forget to reset the scheduled metrics before registering them again!
- Optional: add parameter to adjust log frequency in `do_run`

Implementing a Temperature Monitor

Monitoring with Grafana

- Let's take a look at the temperature data using Grafana (dashboard viewable in a browser)
- Start your satellite with the `edda` group
- Open MissionControl and Observatory to see what's going on
- I will initialize, launch and start the satellites
- I will start a new satellite called `Influx`, which writes the monitoring data to a database
- We will take a look at the monitoring data and create a dashboard in Grafana

- If you ever want to set up Grafana yourself, check this how-to guide

Implementing a Temperature Monitor

Reading the configuration

- Let’s implement a (configurable) critical temperature threshold:
  - In `do_run`, check if the temperature exceeds the threshold
  - If the temperature threshold is reached, use `self.log.warning` to log a warning
  - Turn on the red LED when the threshold is exceeded
- Run and check with Observatory that the warning is emitted
- Let’s make this more severe by going to ERROR if the threshold is exceeded:
  - Add `raise Exception('reason')` when the threshold is reached
  - Don’t forget to reset the LED in `do_initialize`
- Now try again and start another satellite (e.g. `SatelliteMariner`)
  - When the temperature threshold is reached, the other satellite will transition to SAFE
  - This can be used e.g. to shut off sensitive electronics

Implementing a Temperature Monitor

Possible further improvements

- Add a non-critical threshold which logs a warning and turns on the LED when exceeded
- Implement PID temperature control
- Send data and store it in an HDF5 file
  - Use `DataSender` class from `constellation.core.datasender`
  - Create numpy array in `do_run`:

    ```
    data = np.array([temperature])
    self.data_queue.put((data.tobytes(), {"dtype": f"{data.dtype}"}))
    ```
  - Use `SatelliteH5DataWriter` to store data (requires HDF5 installation via apt)

Summary

Summary

- Constellation is a Control and Data Acquisition Framework for small and dynamic experimental environments
- Provides simple interfaces for control, configuration, logging, monitoring, and data transmission
- Learned how to use these interfaces to implement a satellite which monitors the temperature
- Used experiment-indepdent graphical user interfaces to control and monitor the setup
- Used autonomy to guard against crashing user interfaces and to trigger transitions to SAFE under critical situations

https://constellation.pages.desy.de/