



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT INFORMATYKI

Projekt dyplomowy

*Monitoring software for time distribution in the CMS-PPS
detector at CERN laboratory*

*Oprogramowanie na potrzeby monitorowania dystrybucji sygnału
czasu w projekcie CMS-PPS w CERN*

Autor: Wojciech Koszyła
Kierunek studiów: Informatyka
Opiekun pracy: dr Leszek Grzanka

Kraków, 2022



Contents

1	Project goals and vision	4
1.1	Statement of the Problem	4
1.2	Current situation	5
1.3	Vision of solution	8
1.4	Feasibility study	9
1.5	Risk Analysis	9
1.6	Glossary of terms	10
2	Functional scope	11
2.1	System roles	11
2.2	Functional requirements	11
2.3	Non-functional requirements	12
3	Selected realization aspects	13
3.1	Running a sequence and other terms	13
3.2	Structure of the project	14
3.2.1	Clock System Interface	15
3.2.2	Clock System Interface - How to use	17
3.2.3	Interactive CLI App	18
3.2.4	Interactive CLI App - How to use	18
3.2.5	Flask web server	20
3.3	Technology	22
3.3.1	Programming language	22
3.3.2	Web framework	23
4	Work organization	24
4.1	Project team and roles	24
4.2	Methodology	24
4.2.1	Used tools	25
4.3	Work progress	25
5	Project results	27
5.1	Summary	27
5.2	Testing	27
5.3	Deployment	28
5.4	Future of the project	29

1. Project goals and vision

1.1. Statement of the Problem

Measuring data with high accuracy requires high-grade laboratory equipment set up in the proper configuration. Very often the distance between two pieces of equipment can't differ by even a millimeter, as that could lead to wrong measurements, and in effect, fundamentally flawed interpretation.

Time dilation is a phenomenon that revolves around the relative flow of time for fast-moving objects[3]. It's especially hard to measure, as even the cosmic background radiation could affect the sensitive instruments. When one wants to measure the dilation of billion particles per second going near the speed of light like in the Large Hadron Collider at CERN[6], the influence of environmental factors continues to grow to immense heights.

Scientists working on the CMS experiment at CERN use diamond detectors (fig. 1) for accurate measurements of the position and time of flight of the protons[2]. This instrument, as well as some others, such as variable power supplies and network analyzers, require to be configured specifically for each run of the machine. This can be done either manually or through software installed on the local server. Currently, there exists no optimal tool for this task.

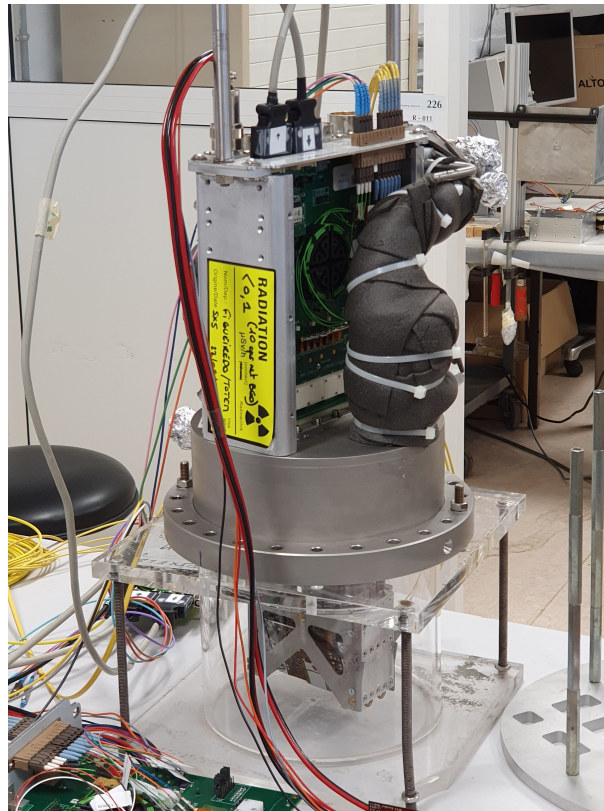


Figure 1: Unassembled detector with diamond plates

The goal of this project is to develop a set of software tools that would allow for remote control and monitoring of the time distribution system in the CMS-PPS detector at the

CERN laboratory. The finished product should use the predefined sequences of commands and be capable of collecting data for future scientific studies to the database by performing periodic runs.

1.2. Current situation

Measuring equipment consists of detectors with diamond plates, laboratory-grade oscilloscopes, variable power supplies (fig. 2), network analyzers, custom-made boards (fig. 3) and many more. They are all reachable, but through different network protocols, such as TCP, UDP and LXI (which allows for communication with devices in remote areas, as if they were on a local network)[10].

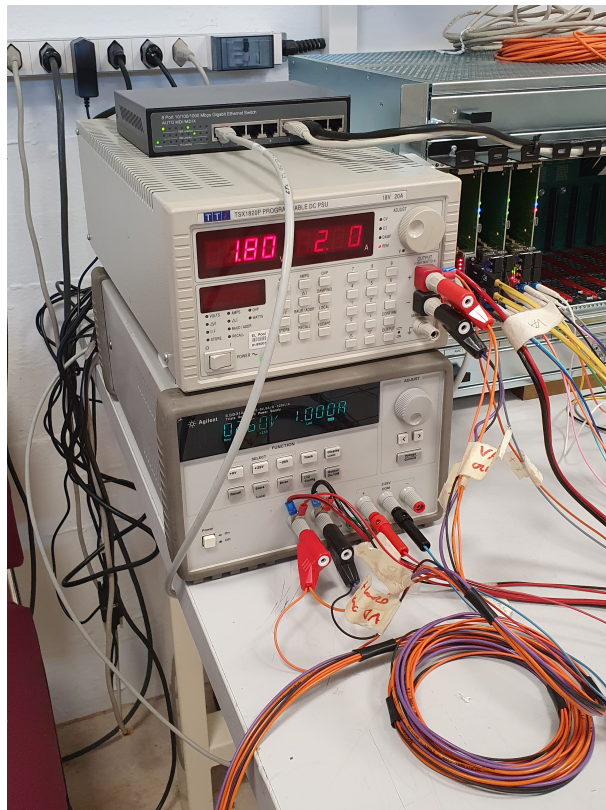


Figure 2: Programmable PSUs (power supply units) at CERN laboratory

To successfully perform "a reading", one has to read data from multiple different devices in a synchronous manner. Sometimes the environment needs to be prepared before reading, e.g. setting up the proper voltage to power up the detector, or changing the minimum and maximum range of the multimeter.

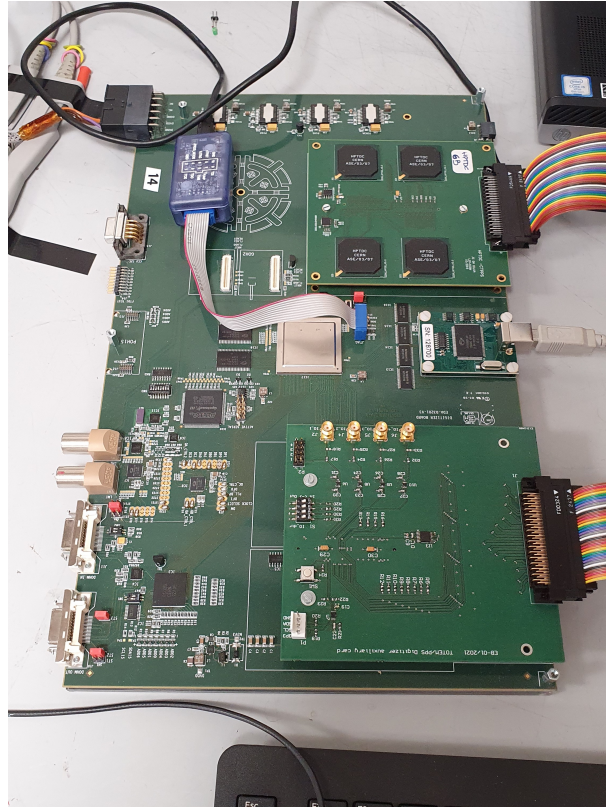


Figure 3: Custom CERN board connected to the network

Scientists working on CMS experiment with Precision Proton Spectrometer(CMS-PPS)[1] decided to write down the proper sequences of "commands" for different workloads in XML format. This file also defines device names and their connectivity details. As the configuration file is being used by other projects it mustn't be modified in any way, shape or form.

A template of the configuration file:

```
<system>
  <name>optical_clock_distr</name>
  <components>
    <component>
      <name>voltmeter_a1</name>
      <type>lxi</type>
      <address>
        <ip>thebest.ip</ip>
        <id>inst2</id>
      </address>
    </component>
    (...)
  </components>
  <sequences>
    <sequence>
      <name>set_example_configuration_of_voltmeter</name>
      <items>
        <item>
```

```

        <component>voltmeter_a1</component>
        <operation>w</operation>
        <command>max_vol</command>
        <value>5.0</value>
    </item>
    <item>
        <component>voltmeter_a1</component>
        <operation>w</operation>
        <command>min_vol</command>
        <value>0.1</value>
    </item>
    <item>
        <component>voltmeter_a1</component>
        <operation>r</operation>
        <command>current_vol</command>
        <value></value>
    </item>
</items>
</sequence>
(...)
</sequences>
</system>

```

Several small scripts to send singular "commands" to the selected device were also created. Those scripts, however, didn't directly make use of the XML configuration file but instead had to be manually modified each time to change the contents of the sent message or the recipient device. The XML configuration file worked just as a kind of "manual"; a paper note, just in a form of a formatted file.

Those scripts had a few flaws. For once, they were written by different people in different languages - C, C++, Python. Moreover, not a single one of those scripts were able to communicate with every hardware component, thus one had to know, which script to use for which instrument. This introduced unnecessary complexity, thus rendering them sub-optimal. They also aren't "user friendly", as they all are console-based applications without GUI.

1.3. Vision of solution

The goal of the project is to create a set of software tools that would allow for controlling and monitoring the whole time distribution system in the CMS-PPS detector at the CERN laboratory.

The project will be fully implemented in the newest (as of the time of writing) version of Python (3.9) and will consist of:

- ***XML configuration file parser***
- ***interface classes*** for communicating with the hardware components
- ***CLI app*** accessible from the console, that can:
 - view the configuration file in a "human" readable format
 - run any sequence of commands defined in the configuration file
 - reload the configuration file without exiting the program
- ***Flask application*** with a user-friendly web interface, that can do everything the CLI app does, but also:
 - program the sequence to run every N seconds
 - save results of the sequence to the specified database
 - is accessible even on mobile phone browser

CERN employees will be able to quickly debug errors when updating the configuration file using the CLI app. If they want to gather data, they will be able to start the repeated sequence on their phone, go back to work on other things and come back to see everything done for them by the application.

Writing the project in Python makes it easier for scientists to understand the flow of the code, as its high level of readability is at the heart of the design of the Python language itself[11].

The command-line interface (CLI) app and Flask application will both be using XML configuration file parser, as well as interface classes. Creating them as independent instances will allow for the future development of new tools.

Both of those tools will handle errors and display them to the user without disrupting the flow of the sequence (if possible). Programs won't crash even if the sequence can't be completed. That will be especially important in the Flask application, as the user won't have direct access to the server running the app.

1.4. Feasibility study

In this project, I could distinguish:

- ***technical feasibility*** - The project will be based on a widely-used programming language and framework. Apart from that, manufacturers of the hardware components provide the necessary libraries to work with.
- ***economic feasibility*** - Time saved by physicists on configuring the instruments outweighs the amount of software engineer work by a major factor.
- ***organizational feasibility*** - The necessity of working remotely imposes a certain way of development.
- ***time feasibility*** - There are about 8 months for the development of this project and it should be sufficient.

1.5. Risk Analysis

The development of every project has a chance to be delayed. I distinguished these problems that I may encounter during the process:

- ***Creating a project by myself*** - one of its biggest advantages is also its biggest disadvantage - lack of discussion. That means there can't be any problems with communication between team members, but there's also no one to point out my mistakes during the process. What's more, if I get sick, the project will be delayed.
- ***Working during Covid-19 pandemic*** - the risk of getting sick is very alarming. Apart from possible permanent damage to the body, the delay could last as long as a whole month.
- ***Lack of professional knowledge*** - while the programming language isn't anything difficult for people with a few years of experience, some of the professional topics about protocols created solely for laboratories, as well as the whole process of obtaining scientific data is a new field of knowledge I have to acquire.
- ***Creating software for closed environment*** - sensitivity of the instruments forbids the use of any communication from the outside. That means everything must work as an offline package. Moreover, the deployment of the project is difficult, as software updates are near impossible.
- ***Working with sensitive information*** - some information about the project can't be disclosed, so working using SSH connection and VPN is required. Getting used to working in that way can take time and dependence on a good internet connection is vastly increased.
- ***Language barrier*** - the clients of the tools will be CERN scientists, so they include members of 23 member states and a few observer organisations. That means everyone will be communicating using the English language, which isn't their native language. This may lead to miscommunication and further delays.
- ***Creating "user-friendly" web interface*** - "user-friendliness" and readability is very subjective and require testing. While evaluation of a proper test group might not be possible, trial with even a few people might become a lengthy process during the pandemic.

1.6. Glossary of terms

- CERN - European Organization for Nuclear Research in Geneva, Switzerland
- LHC - Large Hadron Collider; world largest particle collider
- CMS - Compact Muon Solenoid; one of the large particle physics detectors. The goals of this experiment range from studying the Higgs boson to learning about dark matter[5]
- PPS - Precision Proton Spectrometer; near-beam tracking and timing detectors in the LHC beam-line. It's a continuation of the TOTEM experiment, in which in March of 2021 the existence of a new particle "*Odderon*" was proved.[4]
- TCP - Transmission Control Protocol; one of the most commonly used network protocols
- UDP - User Datagram Protocol; the second most commonly used network protocol
- LXI - Communication protocol for instrumentation and data acquisition using Ethernet [10]
- XML - Extensible Markup Language; a base for HTML
- CLI - Command-line Interface; a process used for communication between the user and the processor
- REST - Representational state transfer; a software architectural style that enforces homogeneous interface, stateless communication and resources
- SSH - Secure Shell; secure network protocol
- IDE - Integrated Development Environment; application with a set of tools for aiding in the process of creating software
- GUI - Graphical User Interface

2. Functional scope

2.1. System roles

The project is targeted towards scientists working on the CMS project at CERN. will be available As the CMS project We can distinguish 3 roles:

- **System developers** - they are the people who continue to develop and extend the time measuring system at CERN. They will mostly use the CLI app to debug the changed configuration file, as well as new communication nodes in the network. They probably won't use the web app, as they don't need scheduled runs and saving to the database.
- **Data scientists** - they are the people who don't need any programming knowledge. They would use the web application to gather continuous data from the instruments.
- **System integrators** - they are the people who create new software solutions that make use of the hardware system in place. They would use the configuration file parser and interface classes in their software.

2.2. Functional requirements

I split the requirements according to their priority, from the highest (0) to the lowest(3).

- As a **System developer** I want to
 - 0 Run sequences of commands (send commands to multiple instruments in a synchronised manner)
 - 0 View the partial responses (from singular commands) while the sequence is running
 - 0 Reload the changed content of the configuration file without exiting from the program
 - 1 See error responses from the hardware on which the command was running on
 - 1 Check the defined sequences of commands in the configuration file
 - 1 Check the defined hardware components in the configuration file
 - 2 Check the available functions of the program
- As a **Data scientist** I want to
 - 0 Run sequences from any device connected to the network without specialised software
 - 0 Save the collected data to the database
 - 0 Schedule a sequence to run periodically on any time interval, which includes:
 - 0 Leave the sequences running while the GUI closed
 - 1 Open the GUI while the sequence is running and view the state of that sequence
 - 0 Check the results of the sequence in real-time while it's being performed
 - 0 See all the actions and the available actions we can perform on them
 - 1 View errors that occurred during the sequence run

3 View the description of the sequence

- As a **System integrator** I want to

0 Use the parser, classes and interface in other projects using the XML configuration file

2 See the description of the classes clearly in the code

2.3. Non-functional requirements

- Both the CLI and web applications are to be user-intuitive and require little to no familiarizing
- The hardware configuration file can't be modified
- The web application has to remain responsive and work correctly on mobile devices
- The program shouldn't require a huge amount of dependencies
- The system mustn't depend on a connection to the World Wide Web, as such won't be available in the production
- The web application should be relatively light, as it will be working on a limited non-scalable server 24/7

3. Selected realization aspects

3.1. Running a sequence and other terms

"*Sequence*" in the sense of a list of "*commands*" run serially, one by one. After the result of the first "*command*" is known, only then the second "*command*" is called.

"*Command*" in the sense of atomic exchange of information between *us* and the recipient hardware instrument. We distinguish 2 types of such commands:

- w - *write* - sending a message to the device
- r - *read* - sending a message and waiting for the answer of the recipient

"*Message*" is any String text that is recognised by a certain hardware instrument. The types of messages that are accepted vary greatly from one component to the other, thus knowledge of their manuals is needed to define them.

Apart from "*commands*" sent to the devices one extra "*action*" is defined in the configuration file - the "*sleep*" command. Our program has to wait for the specified amount of time before continuing to the next step of the "*sequence*".

3.2. Structure of the project

The project was divided into 3 main modules, similar to the 3 client roles:

- **Clock System Interface** or "*Interface tools*", which defines how to load configuration files and how to communicate with hardware components
- **Interactive CLI App** or "*CLI application*", which lets one view and run sequences from the console interface and make quick fast-reloads
- **Flask web server** or "*Web application*", which lets one view the GUI in web browser, view and control sequence runs, as well as save the results to the database and set up periodic runs

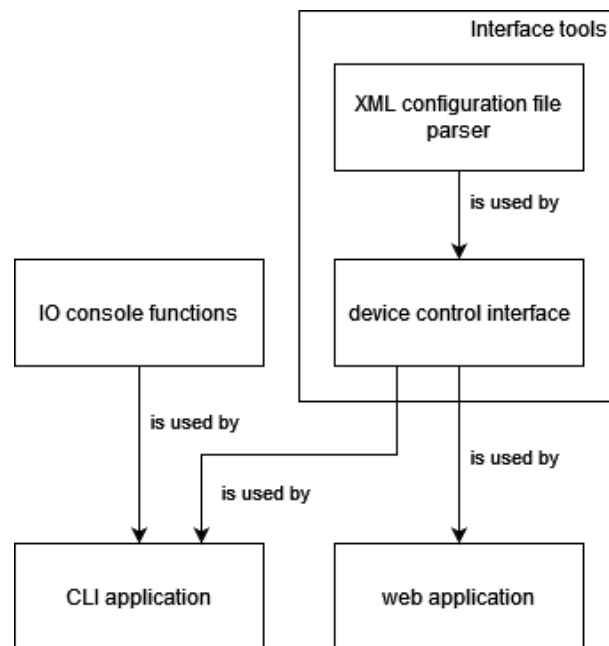


Figure 4: Diagram of project modules

The CLI and web applications are dependent on the same module, which in itself can be used to develop other applications in the future.

3.2.1. Clock System Interface

- It provides a XML hardware configuration file parser.
 - The XML configuration file in its form is currently being used by multiple services and modifying it would require changing the source code of programs on a large scale, thus rendering it nearly impossible. What's more, a unified configuration file makes it easier to update without running any processing on it.
- Allows to send and receive data using predefined network protocols
 - For other modules communication using UDP, TCP and LXI protocols were defined. The code structure was written in a way, such that it will be easy to define more network protocols in the future.
- Acts as a vital part of the other modules (CLI application and Flask server)
- Adds a logic layer for handling the data and communicating with the instruments.
 - Components, sequences and their steps have their own class representations, which in itself helps hide their specific implementation.
 - Most defined classes directly represent the types present in the configuration file:
 - * "Component" class
 - acts as a base for any defined hardware components
 - should be regarded as abstract class and don't contain information about communicating with the hardware
 - has *name*, *type* and *ip*
 - * "ComponentLXI" class
 - represents a component with implemented LXI communication
 - derives from "Component" class
 - apart from the *name*, *type* and *ip* also has an *id* (= LAN device name), which can differentiate between devices on the same IP address. It is represented in this way in the XML file:


```
<component>
  <name>voltmeter_a1</name>
  <type>lxi</type>
  <address>
    <ip>thebest.ip</ip>
    <id>inst2</id>
  </address>
</component>
```
 - * "ComponentUDP" class
 - represents a component with implemented UDP protocol communication
 - derives from "Component" class
 - apart from the *name*, *type* and *ip* also has a *port*, which is exclusive to the device. It is represented in this way in the XML file:

```
<component>
  <name>rasp_a1</name>
  <type>udp</type>
  <address>
    <ip>otherbestip</ip>
    <port>6030</port>
  </address>
</component>
```

* "Sequence" class

- represents a sequence of commands:
 - sending/receiving messages from the hardware components
 - performing an action on the host device, such as "*sleep 5*", which means waiting for 5 seconds before continuing the execution
- has its own *name* and a list of *SequenceItem* elements. It's represented in this way in the XML file:

```
<sequence>
  <name>set_some_value_ch36</name>
  <items>
    (...)
    <item>
      <component>rasp_a1</component>
      <operation>w</operation>
      <command>set_value</command>
      <value>36</value>
    </item>
    (...)
  </items>
</sequence>
```

* "SequenceItem" class

- represents a singular atomic command or action that can be executed
- has *componentName*, *operation* (reading or writing), *command* and *value*. It's represented in this way in the XML file:

```
<item>
  <component>rasp_a1</component>
  <operation>w</operation>
  <command>set_value</command>
  <value>36</value>
</item>
```

- Representation of "actions" on host device are represented like this:

```
<item>
  <component>PC</component>
  <operation>system</operation>
  <command>sleep</command>
  <value>12</value>
```

```
        </item>
    </item>
```

* "ConfigXML" class

- represents the XML configuration file
- loads the data once and holds them in the memory as sets of *components* and *sequences*
- has *pathToXML*, *components* and *sequences*
- contains logic for parsing XML into proper classes

* "DeviceControlInterface" class

- a class responsible for functions regarding connecting to & using hardware components through the network
- acts as an endpoint of this module

3.2.2. Clock System Interface - How to use

This project module can't be used as a standalone application. Instead, it should be imported as a library in future projects that make use of these hardware instruments.

Example of import statements:

```
from interfaceTools.configXML import ConfigXML
from interfaceTools.consoleWriter import *
from interfaceTools.deviceControlInterface import *
```

3.2.3. Interactive CLI App

```

Interactive Console App

https://gitlab.cern.ch/pps-summer-students/clocksystem
https://twiki.cern.ch/twiki/bin/view/CMS/PPSClockSystem

=====
Commands
=====
? / help / commands    - shows the commands
q / exit / quit        - stops the program
sequences              - shows all the sequences from the config file
components             - shows all the components from the config file
sequence <name>        - shows the sequence specified by its name
component <name>       - shows the component specified by its name
sequence <name> run    - runs the sequence specified by its name
reload                - loads the config file again and recreates the config object
=====

```

Figure 5: Console interface of the CLI app

- Allows for
 - running sequences
 - viewing list of components and sequences
 - viewing the details of chosen component/sequence
 - reloading the configuration file
- Provides easy to use commands
- Shows detailed logs about the sequence runs

```

===== Step nr.7 =====
Sending message
Recipient: TCPIP0::tdaqzvl6::inst0::INSTR
Message:   CALC1:STAT:RES? MEAN
Received message
Sender:    TCPIP0::tdaqzvl6::inst0::INSTR
Message:   -75.093517542
Finished the sequence <pdcl loop>
=====

```

Figure 6: Example of the logs returned by the CLI app

- Works continuously - the program doesn't stop unless the user wants it to

3.2.4. Interactive CLI App - How to use

You can start the program by simply calling

```
python interactiveConsoleApp.py
```

however it will run with the default parameters. The recommended way to run the program is to start it from another script, e.g.:

```
def main():
    XML_CONFIG_PATH = '../config/optical_system.xml'
    app = InteractiveConsoleApp(configPath=XML_CONFIG_PATH)
    app.run()

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

While contents of the configuration file can be changed, the path to it cannot once it's set.

Once run, you can use commands:

	Commands
? / help / commands	- shows the commands
q / exit / quit	- stops the program
sequences	- shows all the sequences from the config file
components	- shows all the components from the config file
sequence <name>	- shows the sequence specified by its name
component <name>	- shows the component specified by its name
sequence <name> run	- runs the sequence specified by its name
reload	- loads the config file again and recreates the con

- You can access those commands in the app by typing *? / help / commands* (also if you type the name of the non-existent command)
- App can be stopped with either command *q / exit / quit*, or by sending *CTRL+C* signal.
- Use command *sequences / components* to display the list of the loaded ones.
- Use command *sequence <name> / component <name>* to view the details of a single sequence/component.
- Run sequences using *sequence <name> run*.
- If you made any changes to the configuration file, reload the data with *reload* command.

3.2.5. Flask web server

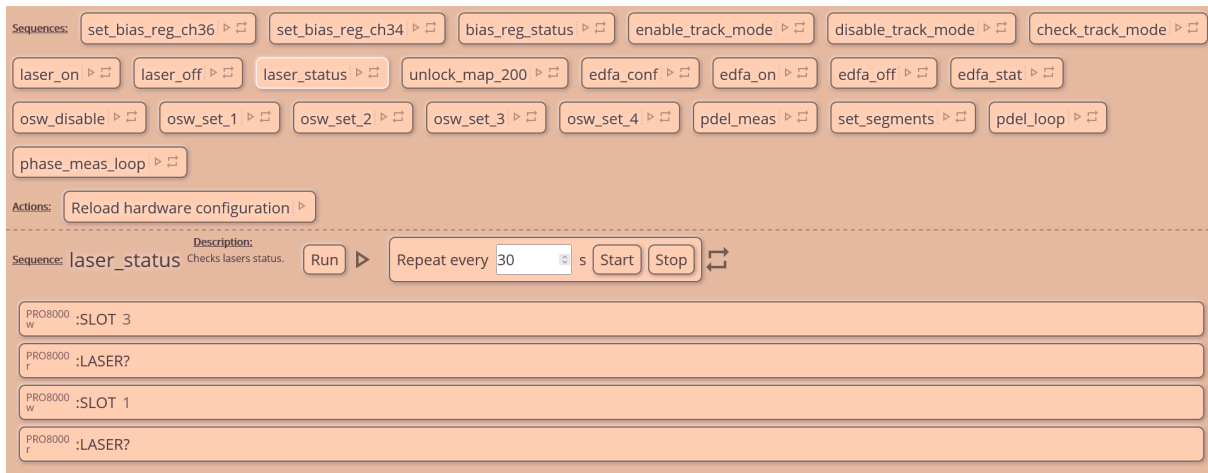


Figure 7: Web interface of the Flask server app

I implemented the web interface as a Single Page Application. The app uses the project module "*Clock System Interface*" and is written using mini-framework *Flask (Python)*.

In terms of capability, it easily outperforms the CLI application.

- Allows for creating "database adapters" for saving the output of the sequence to the database
- Allows for running the sequence periodically on specified time interval
- Implements *request-response* based communication using REST API.
- Provides a web user interface with dynamically loaded sequences
- Sequences can be reloaded "on the fly" using the "Reload hardware configuration" button without restarting the server or accessing the command-line
- Allows for displaying the details of a sequence
- Interface description
 - On the landing page there are 2 main sections
 - * "*Sequences:*" and "*Actions:*"

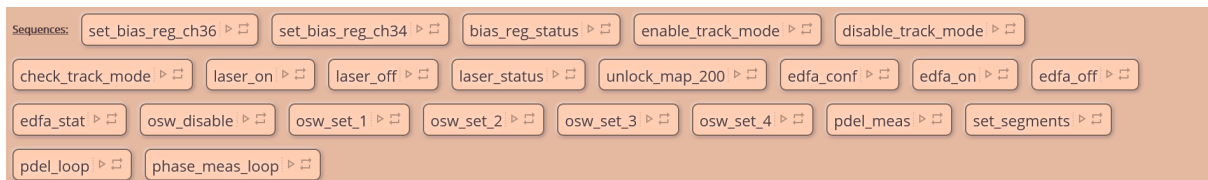


Figure 8: "*Sequences:*" section



Figure 9: "*Actions:*" section

- * empty main section
- The upper "*Sequences:*" and "*Actions:*" section contains a multitude of buttons, each representing one sequence or one action to execute
- Each button consists of:
 - * name, as specified in the configuration file
 - * icons representing their functions.
 Play arrow represents the ability to execute, repeat icon represents the ability to run it periodically and floppy disk icon represents the ability to save the output to the database

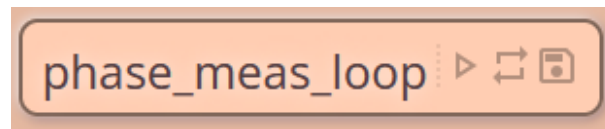


Figure 10: Example of a button from the upper section

- Once any button is pressed, the lower, main section will be populated with the details of that sequence/action

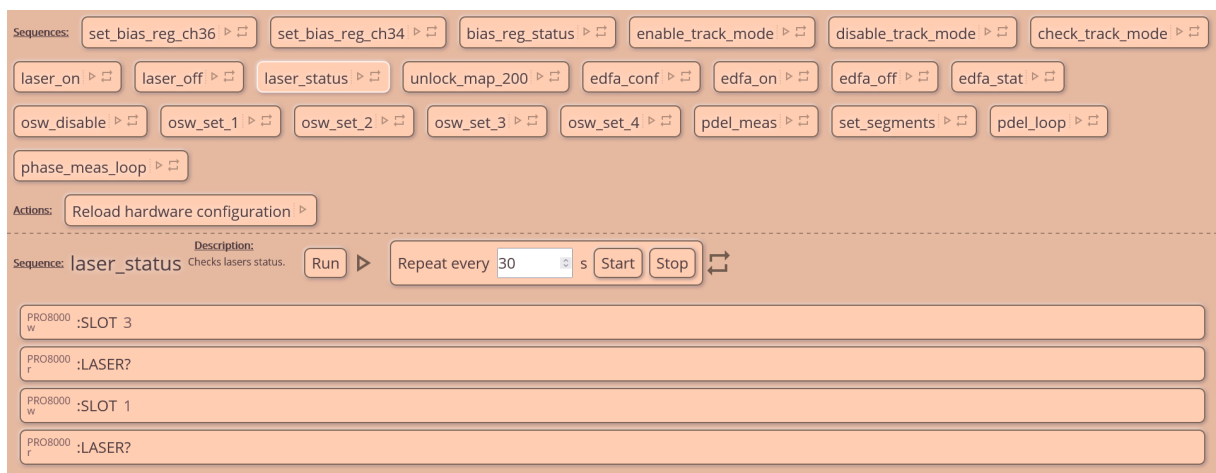


Figure 11: View of the interface after clicking button in the upper section

- In the upper part of the main section we see basic information about the sequence/action:
 - * name
 - * description (if the description was specified in the configuration file)
 - * Run button (if applicable)
 - * section for setting up, starting and stopping periodical sequence runs(if applicable)



Figure 12: Upper part of main section of an action button *Reload*

- If the sequence button is pressed, then the lower part of the main section will be populated with blocks representing the sequence steps.

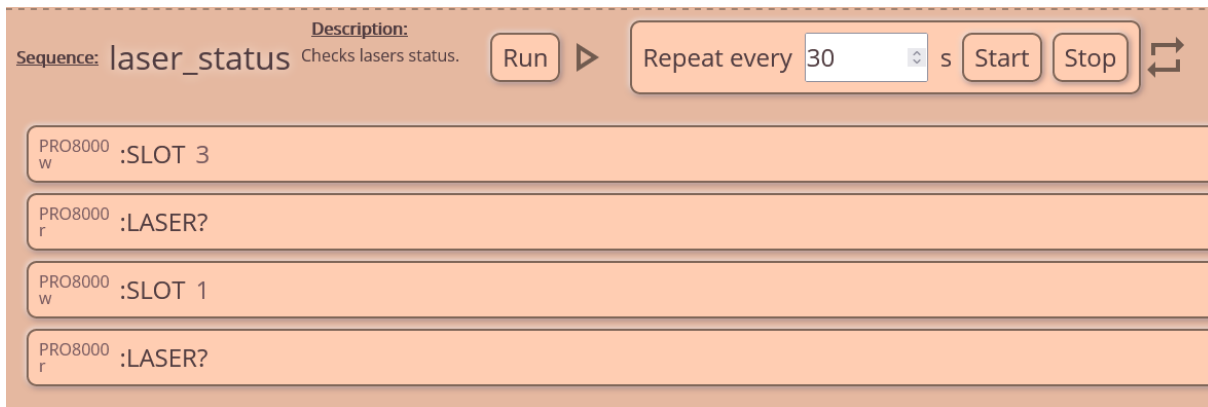


Figure 13: Lower part of main section of an sequence button

- In the top-left corner is the name If the sequence button is pressed, then the lower part of the main section will be populated with blocks representing the sequence steps.



Figure 14: One bar (block) representing a singular sequence step

3.3. Technology

3.3.1. Programming language

I decided to write this project in the newest version (as of the time of writing) of *Python 3.9* language, released in October 2020. The main reason for choosing this language over the others was that it's very popular amongst the scientific community. As such, code written in this language should be comprehensible by CERN staff without education in Computer Science.



Figure 15: Python language logo

3.3.2. Web framework

For the framework, I chose *Flask*[8]. This micro-framework is very fast and lightweight and in comparison to the *Django* framework, it doesn't require the newest version of SQLite to be available on the device.



Figure 16: Flask framework logo

4. Work organization

4.1. Project team and roles

The project is going to be accomplished by one person, assisted by the supervisor from the CERN laboratory.

Executor of the project:

- Wojciech Kosztyła - the person responsible for interviewing potential clients, specifying the program requirements, the vision of the project and full implementation of all the modules and documentation

Supervisor:

- dr Diego Figueiredo - the person overseeing the project and one of the clients who created a draft of the project requirements
- dr Leszek Grzanka - the person overseeing the project and providing knowledge support in the development process

4.2. Methodology

As this development project was supposed to be composed of a set of software tools, instead of a singular solution, the work was planned before any implementation in a specific manner.

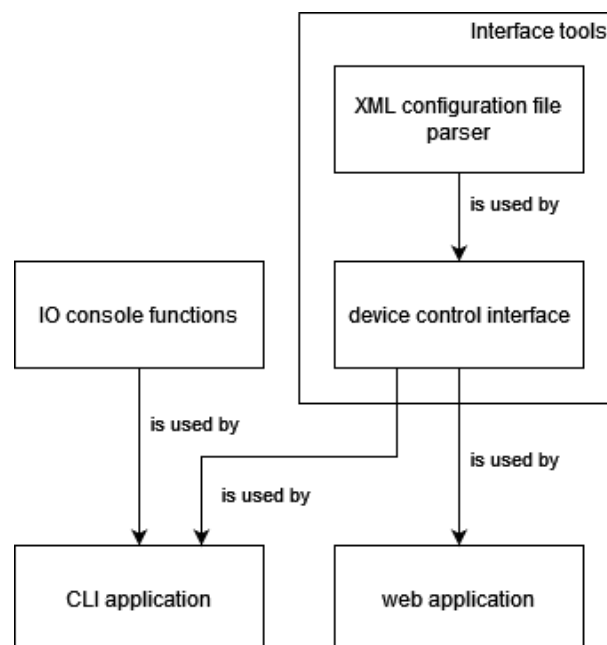


Figure 17: Diagram of project modules

Between the web and CLI applications, the web one required more work. Both of them needed the same components: parser and communication interface. I focused on creating the most basic classes and then incrementally went higher in terms of abstraction.

After creating each of the project components I debugged and tested them. I also revised the project decisions and discussed them with the clients.

During the span of the project, we had arranged remote weekly meetings with my supervisors. During the meetings, we would discuss the progress and possible troubles with implementing the next part of the project. Apart from the remote meetings, for 3 months I was on-site at the CERN laboratory, where I could have live contact with the instruments I was working with, but most importantly my supervisors and other CERN physicists.

4.2.1. Used tools

- Communication tools:
 - *Discord*[7] - communicator for sending messages between me and the supervisors
 - *Indico*[9] - a place for remote weekly meetings with CERN professors
- *GitLab* - code repository
- *Visual Studio Code* - open-source code editor, which I used for creating the website interface
- *Pycharm* - Python IDE
- *Putty* - SSH client for Windows

4.3. Work progress

- *May 2021*:
 - Started interviewing the client about the requirements of the project.
 - Researched about the CERN infrastructure and deepened my understanding of how the "runs" are performed.
- *June 2021*:
 - Set up the environment for remote work at CERN.
 - Checked the previously used tools for controlling the instruments.
 - Made the decision about the technology used for the project.
 - Planned the structure of the project modules.
- *July 2021*:
 - Started working on the "*XML configuration file parser*".
 - * During debugging I created functions, which purpose was to display the data in a more human-readable manner.
 - Fully tested the parser.
 - Started working on the "*device control interface*".
 - Changed the structure of the file system.
 - Installed "*VXI-11 library*", created a small template Python script and tested it.

- Implemented communication with devices with LXI protocol communication.
- Implemented running LXI command sequences.
 - * Testing this required access to the defined hardware components, which proved difficult due to working remote.
- Added "*sleep*" functionality.
- *August 2021:*
 - Implemented running TCP command sequences.
 - Created "*README.md*" file.
 - Created and implemented the "*Interactive Console App*".
 - Started working on the "*Flask web server*".
 - Started working on the GUI design of the web app.
- *September 2021:*
 - Finished the GUI of the app.
 - Started working on "*REST API*".
 - Created server configuration file.
 - Added multi-threading capabilities to the server and created appropriate versions of the "*device control interface's*".
 - Implemented running UDP command sequences.
 - Integrated database connection to the server and created data adapters.
 - Added error handling to the web server.
 - Finished implementing the full functionality of the "*Flask web server*".
- *October 2021:*
 - Added optional "<*description*>" parameter to the hardware configuration file.
 - Revisions according to the client and a lot of debugging.
 - Presentation of the project to the client.

5. Project results

5.1. Summary

All of the functionalities and defined requirements were achieved in the specified time frame, thus the project can be deemed successful. The created set of tools allows for quick debugging(*Interactive CLI App*), data collecting(*Flask web server*) and future software development(*Clock System Interface*).

The project development was an enjoyable and worthwhile ordeal. Creating the project by myself helped me gain experience in all roles of the project, such as:

- *Project manager*
 - for planning, organizing and managing of the project and its development
- *UX designer*
 - for designing the user interface of both *Interactive CLI App* and more importantly *Flask web server* and making sure they're user-friendly
- *Frontend software developer*
 - for creating the client-side of the *Flask web server*
- *Backend software developer*
 - for creating the server-side of the *Flask web server*
 - for creating the *Interactive CLI App*
- *QA Engineer*
 - for performing thorough tests
- *Product owner*
 - for creating the vision of the final project
 - for specifying the project requirements

5.2. Testing

Tests were performed throughout the whole process of development as well at the end of it. *Interactive CLI App* and *Flask web server* were tested against every defined sequence of commands from the file provided by the client on the final hardware. Apart from those, some "fake"(with intentional errors) sequences were added in order to check the handling of exceptions in the script. *Clock System Interface*, as a crucial part of those two, was naturally tested at the same time. The programs behaved as expected:

- If the component in the configuration file has obvious errors (such as missing tags, typos in them or undefined classes), then the component will be omitted with an error message, but the program will continue without it

- If the sequence item (singular command) cannot be executed (due to the hardware component being unreachable or wrong defined) the error will be shown and the execution of the sequence will stop, as to maintain the atomicity of the sequence items
- For *Flask web server*: If one sequence is being executed, requests for other sequences will be omitted and the user interface will be in a "disabled" state till the finish of the execution.
- For *Flask web server*: If the connection with the server will be interrupted, the interface will show the error message and come back to being functional once the connection is reestablished.
- For *Flask web server*: If two people open the interface at the same time and try to run two different sequences at the same time, only the first request will be deemed "proper" and will be executed. The second person will instead be redirected to the currently running sequence and will be able to observe its state in real-time.
- For *Flask web server*: If during the execution or scheduled repetitive execution the user turns off the interface, the program will keep running in the back-end of the server and the user will be able to open the interface yet again and see its proper state.

5.3. Deployment

The project was accepted by the client and **successfully deployed** on the offline underground server at CERN in December 2021. The created tools were thoroughly tested and as a result well received by the scientists working on the CMS-PPS experiment. They are currently being used and will be used in the future development of this experiment and should bring much welcome improvement in the workflow.

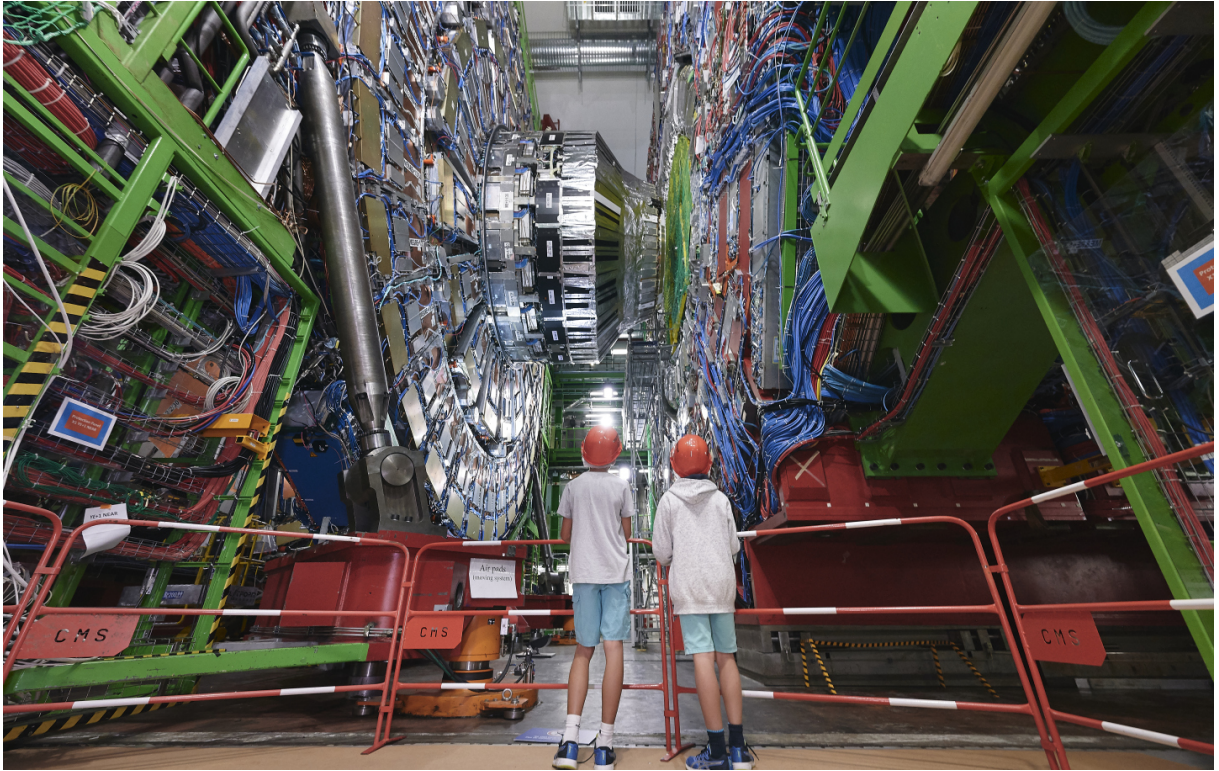


Figure 18: Inside of CMS detector at P5, CERN 2019

5.4. Future of the project

If the project was to be revisited in the future, a few things could be done:

- "server-status" indicator in the corner of the GUI of *Flask web server*, showing whether the connection to the server is maintained or severed
- configuration file editor in *Flask web server*, thanks to which this app alone would be "whole", making the *Interactive CLI App* obsolete
- separate light and dark modes for *Flask web server* GUI

Due to the iterative nature of the development process, few things could profit from the refactorization of the whole project. Rewriting the code from scratch, while having the working project template could help bring out the hidden unnoticed problems of the program.

References

- [1] M. Albrow, M. Arneodo, V. Avati, J. Baechler, N. Cartiglia, M. Deile, M. Gallinaro, J. Hollar, M. L. Vetere, K. Oesterberg, N. Turini, J. Varela, D. Wright, and C. CMS-TOTEM. Cms-totem precision proton spectrometer. `cds.cern.ch/record/1753795`, 2014.
- [2] E. Bossini and N. Minafra. Diamond detectors for timing measurements in high energy physics. *Frontiers in Physics*, 2020. `frontiersin.org/articles/10.3389/fphy.2020.00248/full`.
- [3] J. Freund. *Special Relativity*. World Scientific, 2008.
- [4] Discovery of odderon. `cerncourier.com/a/odderon-discovered/`, 2021.
- [5] About cms. `home.cern/science/experiments/cms`, 2021.
- [6] Facts and figures about the lhc. `home.cern/resources/faqs/facts-and-figures-about-lhc`, 2021.
- [7] The official page of the discord communicator. `discord.com`, 2021.
- [8] The official page of the flask framework. `flask.palletsprojects.com`, 2021.
- [9] The official page of the open-source conference tool indico. `getindico.io`, 2021.
- [10] Lxi standard official website. `lxistandard.org`, 2021.
- [11] The official page of the python programming language. `python.org`, 2021.