UPPSALA
UNIVERSITET

# Open Source Robotized Microplate Automation System

Master thesis - 30 hp

By Carl Emil Thomas Kammarbo

Supervisor: Wesley Schaal

Spring 2020

# 1 Abstract

The purpose of the project was to set up an automation system for use in a biological lab. This system consisted of three devices, a shaker, a dispenser and a washer. The system involved a robot arm with open source scripts and protocols. This arm could then with the information from these protocols, be able to conveniently move around cell plates to designated positions. For example, it could move the plate into any of the devices, as well as put the lid on or off the plate. Having these movements and devices automated could save time and thereby also money for the groups that decide on using the system in their projects.
Automation systems for biological labs have specific constraints and requirements that need to be met. In order to fulfill these needs, careful planning of the overall system was necessary. For example, strict scheduling routines and a controlled environment are necessities.
After the robot arms protocols were successfully set up, a way of controlling these protocols remotely from scripts to the robot was also implemented.
The event driven automation platform Stackstorm was implemented and tested together with the rest of the system. Its purpose was to help with the protocol scheduling. But it was later discovered unnecessary, and manual scripts were constructed to replace it.

After the implementation of the protocols, scripts and the manual scheduling system, the robot arm was able to conveniently decide what cell plate to move, and where to put it. The system would also start the correct procedure on the corresponding device where the cell plate was put.

# 2 POPULAR SCIENCE SUMMARY

When you want to automate something in today's biological labs, the solutions are often sold in big packages to the customers. These packages are usually very expensive, and on top of that acts like a black box, meaning that they cannot be changed or repaired manually by the customer. Instead they require the company selling these systems to send people to the systems that can then adjust whatever needs fixing on the system. This process is often both very costly and also time consuming for the customer, and is often a big problem for places conducting research where it's necessary to test minor changes in order to make it work in the current project.
This is why open source alternatives can be a good fit, especially for these smaller labs that need to make minor tweaks for more precise experiments, or are on a stricter budget. Open source gives the ability to share solutions between organizations and everything is fully customizable. The negative aspect of open source and building everything yourself is that it takes a lot of time from the workers. Another negative aspect of it can also be the lack of expertise in the area, making it difficult to produce the desired product without hiring a third party to help.

This automated robot arm project aims to solve these issues by using a robot arm controlled by custom made protocols. These protocols provide the arm with information on how to move cell plates to different spots. As this project mainly focuses on setting this up for biological labs, it was first tested on a washer, dispenser, shaker, hotel and a liquid handling robot. Each of these equipment pieces does also have different protocols that can be performed in sync with the robot arm during an experiment.
New users of the arm will easily be able to input their own protocols without much training in order to set up the robot arm for their current needs.
The protocols can be set up either by drag-and-drop functions from a GUI that is directly connected to the robot arm or from scripts written in a Python-looking language that are then sent to the robot through a server-client system.

In order to automate the sequence of the scripts and protocols a scheduling system had to be decided on. The possible candidates were Stackstorm, Overlord automation scheduling software or writing a custom system in python. At first it looked like Stackstorm would be the decided platform to use for the scheduling system. This was because of its ability to handle extensions and convenient event-driven architecture. Other good reasons to pick Stackstorm was such as its ease of adding new events, good documentation and the ability to use many different coding languages giving it more room to grow. Big organizations such as Nasa and Netflix also use Stackstorm, further boosting its credibility. With Stackstorm, users can get messages through slack or email depending on different events that are occurring within the system. It was however later determined to go with the more lightweight option of writing a custom system in python, as Stackstorm was moving towards a more commercial business model as well as carrying too much overhead for the project at hand.

The project was left open for future improvements such as implementing a second robot arm or adding more devices. It was also put in a state, where a more advanced AI could be built into the scheduling system, if the experiments were to require an even more flexible solution.

# 3 TABLE OF CONTENTS


1 Abstract ..... 1
2 Popular science summary ..... 2
4 Introduction ..... 5
5 Experimental ..... 8
5.1 Equipment setup ..... 8
5.2 Protocol setup ..... 8
5.3 Deciding on connection method ..... 9
6 Scheduling ..... 10
6.1 Stackstorm ..... 10
6.2 Manual scheduling ..... 10
7 Results and discussion ..... 12
7.1 Atomizing protocols ..... 12
7.2 Connection types shortcomings ..... 12
7.3 Automation flexibility ..... 13
7.4 Future improvements ..... 14
8 Conclusion ..... 15
9 References ..... 16

# 4 INTRODUCTION

This thesis report is structured in such a way that the introduction covers the problems in question, previous research made in the area and it ends with explaining the aim of the thesis. In the next chapter, Experimental, the experimental set up of the project is described. This is followed by a chapter on how the scheduling system was decided on and added into the system. After this, the chapter Results and Discussion follow, where all the project's results are presented as well as a discussion covering the different problems in the project as well as future improvements. The last chapter, Conclusion, consists of a summary which describes how the project ended up.

When implementing automation solutions for biological experiments in research labs, it's important to keep in mind that these solutions should be as flexible as possible. Many automation tools are set up in such a way that they cannot be easily adjusted without the help of the main manufacturer. That is why open source software is important for such a scenario. It can allow for complete control of the system, but at the cost of work and time. It is however often much cheaper than buying an industrial standard closed system. Making it a very desired option for a biological research lab. (Hippel, 2001)
To expand further on this, it's also important to use a multicomponent system so that it can be easier to adjust in the future. This allows for changing one part of the system without causing unnecessary effects on the other parts of the system. (Vasilisa, Kotliar, Kotliar, & Popova, 2018)
More advancements of the multicomponent system can be done by containerizing the system. For example, with Docker or Kubernetes. Which will allow for bundling components and isolating them from the others. Instead, the different containers can communicate through defined channels. (Bernstein, 2014) (Anderson, 2015)

Another problem with biological experiments is reproducibility. As reproducibility is very important in biological labs to properly be able to test different settings on the experiment while still being able to accurately measure the outcome, it needs to be accounted for. Automation can help solve this issue by reducing the human error factor. Although not perfectly as there are a lot of minor things that need to be taken into account that can affect a biological experiment. For example, the moist level in the air or light hitting the biological material etc. The automation system can manage to improve this by as it can do things more precise and in the exact same way as it performed the action last time, unless the system has been manually changed or is malfunctioning. (Sonnenschein & Jessop-Fabre, 2019) (Cooksey, Elliott, & Plant, 2011)

For all of these components to be able to communicate properly, as in send the right action at the right time to perform the correct process, it needs to have a system in place to handle this. A good way of handling all the components in a biological experiment is to set up some kind of scheduling system. The system would need to be open source and flexible in such a way that it is convenient to improve as the biological experiments develop over time. (Stancu, ArkadiyShevrikuko, & Rueda, 2019) A manual system could be set up for this purpose in for example python, but it would be time consuming. One possible software solution for this problem

could be the event driven platform, Stackstorm, that offers an open source, multipurpose scheduling system. It has promising reference uses such as CERN, where it is used to automate and handle the general network configuration for different devices. It is useful in this case because it can implement various policies that will help configure the network flow. (Crooks, et al., 2019) (StackStorm, 2020) It is also used by other big organizations such as Netflix, NASA and Cisco. (itcentralstation, 2020) Stackstorm works in a component-based way. All of these components are built into what is called a package, which can be easily installed and shared through GitHub into the Stackstorm server. They are all open source and allow for contributions between different parties. The components inside a package consists mainly of sensors, triggers, rules and actions. A sensor can be set up in any coding language and polls for new information. When the sensor has collected data that fits a rule, it can issue a trigger which will perform an action. All components are set up with yaml files to inform Stackstorm of which information should be sent where. The action component can be built with any coding language and issue any command to be executed. This flow can all be monitored and controlled either from a command line interface or from a web gui that is provided with the basic Stackstorm setup. (Stackstorm, 2020)

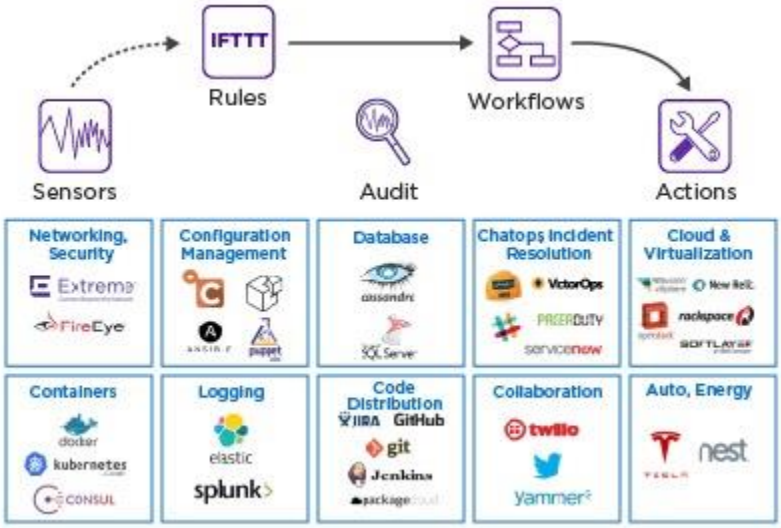


*Figure 1: Stackstorm component workflow. (extremenetworks, 2020)*

There is also various other alternative software to Stackstorm. One example of these other software is the automation tool Overlord Automation scheduling software. This software allows for an easy drag and drop system after being set up. But it is however not open source, which Stackstorm is, and that could be problematic as previously mentioned. (paa-automation, 2020) It is also important that these systems can be adapted into being more adaptive and eventually maybe even controlled by an ai. (Kaber & Endsley, 2004) (Liberatore, 1997)

The aim of this project is to solve these previously mentioned problems while setting up a system for automating cell proliferation experiments. To do so it is important to structure the project's main pieces into multiple components that have got proper connections to each other. A good scheduling system for automating these components would then have to be decided on and implemented. While also keeping in mind of leaving the project flexible for future expansions, like more devices or other control systems.

# 5 EXPERIMENTAL

## 5.1 EQUIPMENT SETUP

For this experiment, a robot arm UR10 was set up on a table in a box. A washer, dispenser and a shaker were also placed on this table. These devices are all part of the cell proliferation experiments and are some of the main targets for the robot arm. Outside of these, a hotel with 18 spots was also set up on the table in purpose of storing cell plates in between runs.

## 5.2 PROTOCOL SETUP

The project started by manually setting up protocols for the robot arm. A protocol is a file containing a series of actions of which the arm will perform when the selected protocol is being executed. These protocols were saved as URP-files on the computer that was directly connected to the robot. The protocols were created with the help of the stock GUI-control panel that was installed to the robot-computer. It allowed for moving the robot arm in the different desired positions and then saving them in sequences gradually building up the protocols. This could be done by either putting the robot into free-drive mode which allowed for moving the robot arm manually or by using the GUIs controls to move each joint of the robot. From the beginning these protocols could only be started from the GUI on the robot-computers control panel.
Planning could then begin after understanding how to set up these protocols on how the robot arm should maneuver around in the box to be able to perform its task as efficiently as possible. One big restriction was that the arm cannot rotate more than 360 degrees, meaning that it wouldn't be able to rotate any further at a certain point and would have to go back around instead. Another thing that had to be planned for was how the cell plates should be lidded and delidded. It was decided on to use a certain spot on top of a shelf where the robot conveniently could reach. On this spot, the robot could land plates that were going between the hotel and the devices. And after putting down a plate here the arm could either take off the cell plates lid and put it on the side if it was coming from the hotel, or take an already disconnected lid and put it back on the cell plate before returning it to the hotel.
Another problem that occurred was that the robot arm could only reach to put down or grab cell plates by holding them horizontally into the washer while having to grip them vertically to put them down into the dispenser. As the arm can't change orientation while already holding the cell plate, in order to solve this, the previously assigned landing spot on the shelf was used again. The arm will always grab the cell plates horizontally from the hotel. Therefore, no change was needed for the path into the washer. But to solve it for going between the hotel and the dispenser, the arm had to switch into a vertical state after taking the lid of the plate on the shelf. It could then proceed to put it into the dispenser. And then then do the same movement in reverse to return into a horizontal state and put the lid back on the cell plate.

## 5.3 Deciding on connection method

Eventually, a program acting as a client was created as a python script that could connect to the robots “dashboard-server” from another computer with a simple telnet connection. Through this connection, URP-files could now be loaded into the robot's memory and then run. Other basic commands such as stop, pause and get currently loaded file also worked with this program.
One big limitation of doing it this way was that the robot has a build in “start safety limit”. This made it so if the robot was currently positioned too far away from where the next protocol being run should start it would just prompt an error message. In order to work around this, multiple small protocols containing only single check points were created. These checkpoint-protocols were then run between the main protocols to be able to maneuver without errors.
In the start of the project, rather large protocols were set up, but it was later discovered that multiple small protocols would work much better. The reason for this is because it gave a lot more flexibility when calling the protocols with the python program remotely.
Another python client script was also set up. This script could connect to another interface on the robot then what the previous telnet-script had connected to, and it was done by using a TCP connection. Instead of being able to load and run programs, this connection could directly issue move commands to specific commands. The problem with this way of connecting was that there was no way to retrieve any useful data from the robot. In other words, there was no way of knowing when one move command was done running and it should issue the next. Experiments were made by mixing this new way of connecting and the telnet connection. But it was eventually decided to only go with the telnet connection that only loaded and ran URP-files.
The first version of the script was set up in such a way that all the possible URP-files the script could load were saved in separate variables. The user could then add these variables into a list in the order that they want them to be executed by the robot.
API’s were set up by another person for the shaker, washer and dispenser. These API’s allowed for checking the devices current running status as well as execute them with a specific protocol. From the robot script, the shaker, washer and dispensers’ commands could now also be added into the list for execution. When the program is executing the list, it will simply check what device the command belongs to. If the command was determined to be for the robot it will send it through the telnet connection to the robot. And if it's to any of the device’s washer, dispenser or shaker it will perform a get request to communicate with the API. At this stage, the script could also handle various waiting phases for the different devices. It will do so by checking the status received from the API and then using a sleep method, pausing the process temporarily before trying again to see if the status has changed. When the status condition is eventually fulfilled the program will proceed to execute the next step. For example, the robot arm could have been waiting for the washer to finish its current running protocol, and when done, the robot arm would pick up the plate from the washer.

# 6 SCHEDULING

## 6.1 STACKSTORM

After the basic system was set up, the implementation of the event driven automatization platform Stackstorm got started. A Stackstorm server was first set up. There was trouble using the official auto installer, so all components of the Stackstorm server were instead installed manually. This allowed for installing Stackstorm packages as well as accessing a premade web-GUI included with the Stackstorm server. A Stackstorm package consists of different components. The package itself, as well as the different components consist of a bunch of yaml files and various scripts that are connected to each other. These yaml files communicate between the packages and explain to the Stackstorm server how and when they should be used. The main components of the packages are Sensors, Rules and actions.

Sensors are a way to poll information from devices or scripts to check current conditions. These can be built by various different coding languages, but python was selected for this project to keep uniform. The information gathered from the sensors can then be checked with the rules. If something is true, then proceed with performing an action. An action component executes or performs something. For example, start or stop the robot. Stackstorm also supports three different workflow architectures. These are Orquesta, ActionChain and Mistral. They can then set up a chain of events to handle the different components in different desired orders. The most modern and well put together workflow architecture of these are the Orquesta architecture and it was used in this project.

A Stackstorm package was set up where the sensor was used to check for the different main devices running status. It was doing this basically the same way that the python script was doing it earlier. Which means that for the washer, dispenser and shaker it was performing get requests to communicate with the device API. While for the robot it used a telnet connection to get the running status. The rules were then set up to handle these requests according to the situation and in response, an action was fired off. The action in this case, consisting of starting parts of the robot's python script that runs the robot as before, or building protocols for it.

The web-GUI was set up in such a way that all the actions and sensors could easily be monitored and executed directly from any web browser that had access to logging into the Stackstorm server.

## 6.2 MANUAL SCHEDULING

It was later decided that using Stackstorm for the scheduling system of the project would not be worth it. Instead of using Stackstorm, development of a scheduling system in python was started. The new program was set up with eight different classes. These were the following: a main class, an event server class, an event client class, a plate class, a protocol builder class, a system runner class, a robot connection class and a prioritization class.

The main class was used for controlling the flow and starting the other classes. While the event server was built in a way so that it allowed for accepting incoming protocols. It also checks for

inputs at the same time with the help of running the different processes with threading. From the event client, protocols could be sent to the event server. When the event server receives a new protocol, it creates a new plate object and stores it into a plate list. A plate object holds the plate's current position, the next position and a path list containing all the positions the plate is going to traverse. It was all set up in such a way that new positions to already created plates could also be added by sending a command from the event client to the event server. The prioritization class was made to decide which plate to move to what position at the current step. A snapshot from this can be seen below in figure 2. It was set up to priorities plates in the order that they were added to the system. If a plate with a higher priority was occupied in a plate it would check if it could move the next plate etc. The scheduling system also included a time factor to check if it was worth waiting for a higher prioritized target or move on to the next target. From the protocol builder, a path could then be constructed. The path was ranging from the current global position of the system to the decided upon destination with the corresponding plate. When the paths are being built, required checkpoints are being added between the current position and the destination. These are stored into a list and sent back to the calling function in the main system. The runner is used from the main system, where the previously built protocol with the checkpoints is used as an input parameter. This part of the system will go through each step of the protocol and execute it accordingly. It will do so by checking for the type of command on each spot in the list. If it's a command meant for the robot, it will use the previously set up telnet connection to communicate with the robot and issue the command. And if it is a command for any of the devices, washer, shaker or dispenser it will make a get request to the given address server. It will also check with ready requests to respective devices to see if it is ready. This is also done with get requests. The runner will then communicate with the prioritization object to prepare for the next step in the sequence.

```
def priority_system(self): # Get the plate to move
    for plate in self.plate_list: # Prioritises plates in added order
        cur_step = plate.path[plate.cur_step] # Get the current plats destination
        if self.build_checkpoints.w_get in cur_step: # Is current destination washer?
            if self.robot_run.is_washer_ready(): # Check if the washer is ready
                return plate # Return this plate for running
            else:
                continue # Continue and check if the next plate can be moved
        # ...
```

*Figure 2: Segment from the priority system for the scheduler*

# 7 RESULTS AND DISCUSSION

## 7.1 ATOMIZING PROTOCOLS

From the start of the project, it was very common to build large protocols for the robot to run with. A large protocol meaning that it contained many different positions in one run file. But as time progressed, these protocols were being made to be more and more atomic. As it was discovered that running the protocols remotely would be much more flexible by having smaller protocols, in other words, adding a lot of more possibilities when later running them through scripts.

## 7.2 CONNECTION TYPES SHORTCOMINGS

Different connection methods between a remote host machine and the robot were tested during the project. They all came with their own problems, and as running from the robots control panel GUI was too clumsy, this has to be solved. The first method of connection was to connect to the robot's dashboard interface, this only allowed for loading, starting and stopping different protocols that were already set up on the robots directly connected computer as urp files. And the biggest problem here was that to be able to start a new protocol, the current global. robot position needs to already be close in 3d-space to the starting point of the new protocol. Therefor new protocols, acting as checkpoints had to be added. These checkpoints then had to be played in a sequence between the current position and the destination for the robot to be able to move smoothly. One problem with this method is that it takes some time for the computer to load each protocol file. Meaning that the robot would have to stop for a few seconds between each checkpoint to load the new file. These files could also not be loaded in advance as the robot could only run the currently loaded file. This was however determined to not be a practical problem, as it would not be a time limiting factor in the cell proliferation experiments that would later be performed with this system set up.

Another alternative of connecting to the robot was by connecting to another interface on the robot's TCP/IP socket connection. This allowed for manually sending coordinates for the robot to move to. The biggest problem here was that there was no way to get any feedback returned from the robot's server. Meaning that there was no way of knowing if the current action was running or finished. Another smaller problem was that getting the exact coordinates for the system to move accordingly would be quite cumbersome to set up, as they would first have to be set up manually with URP files and then later collected from these files.

There was also the possibility of manually setting up a server on the robot's computer. This was however determined to be redundant at time consuming. So, the method of choice was decided to be the one that allowed for connecting directly to the robot's dashboard server. As this one had no critical drawbacks. And its only real problem, the time constraint between loading new protocol files, would not be a practical difficulty as mentioned earlier. It would also be much more convenient to develop then the full manual server client system.

## 7.3 AUTOMATION FLEXIBILITY

To establish the connection to the interface decided on. A telnet connection was used for simplicity. The system would only be allowed to connect from one computer, already located in a secure network, to the robot's main computer. Because of this, there would have to be no security concerns about the data transfer with using the telnet connection. And using SSH would only add an extra, unnecessary layer of work.

This connection was set up with a simple python script in the beginning. Where a list would be inputted with all the protocols stated in order. The list was then looped through by the script and each spot was executed one by one while polling for information from the robot to see if it was ready in position to accept a new command. One problem here was that each checkpoint had to be added one by one to the inputted list between the actual spots that the robot should move between. This was not only time consuming but would also be very confusing for new users, as they would have to learn exactly what check points would be needed between each practical spot for the robot. To solve this a protocol builder system was developed that automatically added the needed checkpoints between two spots, and eventually compiled it into a new protocol list that could be executed by the systems runner. The next problem with this set up was that it was set up per project and not per cell plate. Meaning that it was only tracking an overview of the whole project with all the plates, instead of focusing individual plates. Making it less flexible than it had the potential to be. As of now there was no real way of keeping track of individual plates. This made it so that making changes or adding new commands to the system while running multiple plates would be a huge problem. Therefore, the system had to have some kind of scheduling system.

A promising candidate for the scheduling system seems at first to be Stackstorm. Which is an event driven open source automation platform. But after further development with Stackstorm, it was noticed that it would not be a good fit for this use case. As Stackstorm requires the setup of multiple yaml files for all of its packages and package components. It was deemed to be too clumsy as the whole Stackstorm server had to be run, which might as well have been built manually in for example python. Which would allow for much more flexibility in the project. The advantage of Stackstorm is that it provided a lot of premade packages from various companies. These packages are however not needed in this project and unlikely to be needed in the future. So even though Stackstorm is open source, it would be more convenient to create a scheduling system from scratch then to adjust the Stackstorm code. Another thing that spoke against using Stackstorm was that it was later noticed that they seem to be moving towards a more commercial business model. (Stackstorm, 2020) Adding paid features that are not open source. Which would not be good for a biological lab experiment as previously mentioned. This would decrease the flexibility that is needed for such an experiment.

So instead of using Stackstorm, a manual scheduling system was built with python. And with this, a specific plate class was added that could keep track of its current position, next position and full path.

The system now had the ability to keep track of every plate position that was in the system. It was also able to properly select which plate had the priority to be moved next depending on the current situation. For example, the priority order depended on what order the plates were added

to the system. If one plate is being proceeded in a device like the shaker. It would drop in priority and the robot arm would move to the next available plate and process it and so on.

## 7.4 Future improvements

Further advancements in the project could include implementing another smaller robot arm. This robot arm could, for example, be used to put the lid on the plates or take the lid of plates at a specific station. If this was implemented, it would save a lot of time, as the main robot arm could be performing other actions simultaneously. The new arm could also be put in positions where the main arm would have trouble reaching. For example, moving the plates between the device's washer and dispenser. This would also save a lot of time, not only because the arms could perform actions simultaneously. But also, because the main arm has to switch between holding the plate from horizontal to vertical to be able to reach and plant or pick up the plate in the dispenser. The main arm also had the rotation restriction of 360 degrees meaning it had to reverse a whole lap if it wanted to pass the border of 359 to 0 degrees. If the mini robot would handle the moving between these stations, it would solve most of these problems.

Other possible improvements could be things like improving the scheduling system. It could, for example, have an advanced AI controlling it and making proper decisions that adapt to the situation. This wouldn't really be needed if the experiments are done at the current scale. But if the experiments or the lab were to be scaled up in the future, it would be a huge advantage to not have to manually set up new procedures every time by the operator.

Another thing that would expand the scalability and overall dynamic of the project would be to containerize the different components. This could be done with, for example, docker or Kubernetes. This would allow splitting the different components of the system into different containers. Allowing for expansion or modification of the desired component without affecting the rest of the system. It would also allow for a more secure backup system.

Smaller improvements that could be added to the project involve adding a slack or Gmail notification system. Allowing the user to get notifications when certain conditions are fulfilled. For example, the robot finishing its current task or an error is occurring. The system could also have been connected to a Grafana GUI, allowing for a graphical representation of what is currently going on in the experiment.

# 8 CONCLUSION

The default dashboard connection on the robot's server turned out to be the best interface to connect to and the one that was decided to be used in the project. This was as it had no critical faults preventing it from being used compared to some of the others. For example, one connection type could not check the running status of the robot, which made it unusable as it would be necessary to check its status to be able to determine the next cause of action. The chosen interface only had some minor flaws. For example, taking a few seconds to progress between each URP file being loaded into the system. This was however not a limiting factor as the robot arm will most likely be waiting for other stations to finish up anyway.

When it came to implementing the scheduling system, it was determined that Stackstorm was not worth using as its infrastructure was too inconvenient for its needs. It also seems like the business model of Stackstorm is moving towards being more commercial and less open source. Which is as stated earlier, is not convenient for biological research experiments because of its lacking flexibility and cost. Instead of this, a manual scheduling system was developed with python and merged with the rest of the client system. Giving more control at a lower level to the user, while still being easy to use.

If the project were to be repeated, a lot of time could have been saved by directly skipping the use of a third-party software for the scheduling system. Instead a custom system could have been built directly. It would also have been more efficient if the custom python program were directly made to be used as a per plate object, instead of handling the experiments with a higher overview as it was first designed to do.
Due to circumstances of the time when the project was performed, certain elements were difficult to properly test. And therefore, had to be created without testing, and then tested in bulk. Resulting in overall time and quality loss. This could have been avoided if the project was carried out at another time, where these circumstances are not a factor.

The project has been with flexibility and scalability in mind. Meaning that it will be easy to expand upon in the future to make it fit new needs. Not only for stacking new functions into it, but also for modifying the core scheduling function to be even more dynamic. And as previously mentioned, changing the scheduling could include implementation of an advanced AI to handle its different tasks.

To conclude this, the planned project worked as the robot arm could handle the plates according to a scheduling system. And this could be done in sync with the other devices.

# 9 REFERENCES


Anderson, C. (2015). https://ieeexplore.ieee.org/abstract/document/7093032. *IEEE software*, 102-c3.

Bernstein, D. (Sept 2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE*, ss. 81-84.

Cooksey, G. A., Elliott, J. T., & Plant, A. L. (2011). https://pubs.acs.org/doi/abs/10.1021/ac200273f. *Analytical Chemistry*, 3890-3896.

Crooks, D., Vâlsan, L., Mohammad, K., McKee, S., Clark, P., Boutcher, A., . . . HenrykGiemza. (2019). Operational security, threat intelligence & distributed com-puting: the WLCG Security Operations Center WorkingGroup. *EPJ Web of Conferences* (s. 214). EPJ Web of Conferences.

extremenetworks. (2020). *extremenetworks.com*. Hämtat från extremenetworks orchestration: https://www.extremenetworks.com/extreme-networks-blog/leveraging-automation-and-orchestration-the-precursor-to-ml-and-ai/

Hippel, E. v. (den 15 06 2001). Learning from Open-Source Software. *Forthcoming Sloan Management Review*.

itcentralstation. (2020). *itcentralstation*. Hämtat från itcentralstation stackstorm: https://www.itcentralstation.com/products/stackstorm-reviews

Kaber, D., & Endsley, M. (2004). The effects of level of automation and adaptive automation on human performance, situation awareness and workload in a dynamic control task. *Theoretical Issues in Ergonomics Science*, 113-153.

Liberatore, M. J. (1997). Automation, AI and OR: In search of the synergy and publication priorities. *European Journal of Operational Research*, 248-255.

paa-automation. (2020). *paa-automation.com*. Hämtat från paa-automation scheudling software: https://paa-automation.com/products/overlord-scheduling-software/

Sonnenschein, N., & Jessop-Fabre, M. M. (2019). Improving Reproducibility in Synthetic Biolog. *Front. Bioeng. Biotechnol.*

Stackstorm. (2020). *Stackstorm*. Hämtat från Stackstorm overview: https://docs.stackstorm.com/overview.html

Stackstorm. (2020). *Stackstorm ewc*. Hämtat från Stackstorm ewc: https://docs.stackstorm.com/install/ewc.html

StackStorm. (05 2020). *StackStorm Features*. Hämtat från Stackstorm.com: https://stackstorm.com/features/

Stancu, S. N., ArkadiyShevrikuko, & Rueda, D. G. (2019). Evolving CERN's Network Configuration Management Sys-tem. *EPJ Web of Conferences* (s. 214). EPJ Web of Conferences .

Vasilisa, E., Kotliar, A., Kotliar, V., & Popova, E. (2018). MULTICOMPONENT CLUSTER MANAGEMENT SYSTEM FOR THE COMPUTING CENTER AT IHEP. *Proceedings of the VIII International Conference "Distributed Computing and Grid-technologies in Science and Education"* . Moscow Region: Russia .