UiT

THE ARCTIC UNIVERSITY OF NORWAY

The Faculty of Science and Technology, Department of Computer Science

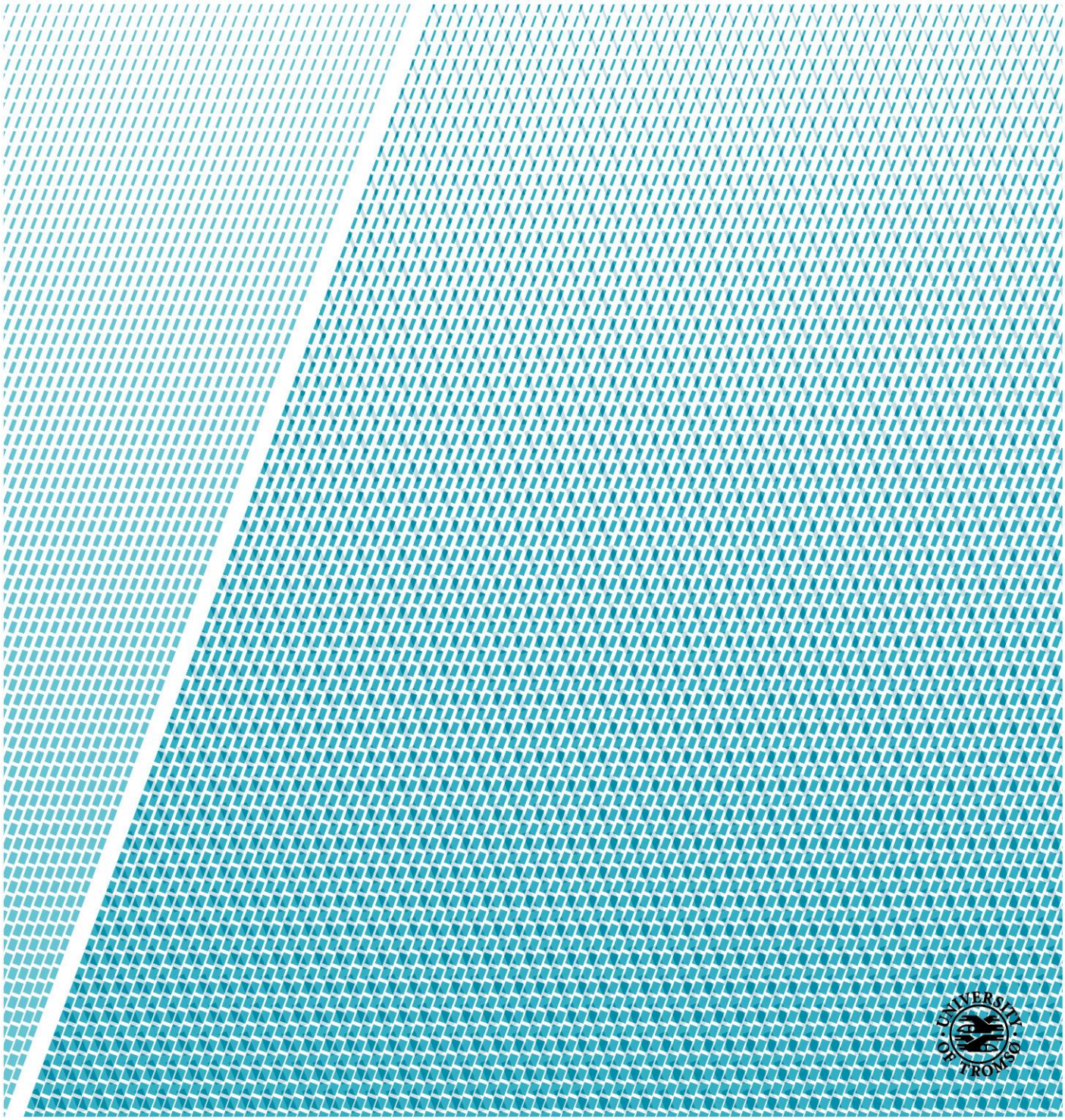
DaoCron

Job Scheduler for Autonomous Observation Units in the Arctic Tundra

—

Martin Sommerseth Moe Carstens

INF-3981 - Master's thesis in Computer Science … June 2019



Abstract

The Distributed Arctic Observatory (DAO) aims to improve the data collection from the arctic tundra using Observation Units (OUs). These OUs are given a set of tasks which they are expected to schedule at certain intervals. Such tasks could be reading from a sensor module, sending data to a remote server or performing maintenance work on the OU just to mention a few. In order to schedule these tasks at certain time-intervals one can make use of Cron, which is a tool for time-based scheduling of jobs in a Unix-like system. However, the harsh conditions of the arctic tundra may result in unavailable networks and low levels of energy which may affect the results of the scheduled tasks. This thesis presents DaoCron, which is a task scheduling service designed for the OUs at the arctic tundra.

DaoCron can be set to monitor the dependent network connections and energy level of the OU. Scheduling tasks which are restricted by these variables only during time-periods were the requirements are met. DaoCron will keep tasks dependent on network connections to succeed on hold during periods of no network connectivity and reschedule them once the connection comes back online. The user might specify a minimum energy level required for a task to be scheduled, which DaoCron will consider when performing scheduling of the given task.

DaoCron provides more control and customizability for the user than what is achievable through Cron. Allowing the user to provide a larger set of configurations for the tasks, custom written pre-schedule evaluation scripts and performing periodical reporting. Ultimately, the user can have the final saying in whether a task is to be scheduled or not. Most of these configurations and controls are entirely optional and DaoCron will adapt to the user’s needs.

As the network and energy levels of the arctic tundra is rather unstable, DaoCron tries to include historical models in the scheduling process in an effort to avoid scheduling tasks at periods of high historical network unavailability and low average energy levels. This is achieved my offering different scheduling methods and the option to use and perform analytics. DaoCron can be configured to periodically build models for the network, energy and task error rates from its collected data which is used to improve the scheduling of tasks.

Acknowledgements

I would like to express my deepest gratitude to my main advisor Professor Otto Anshus, and co-advisors Professor John Markus Bjørndalen and Post Doc Issam Raïs for your valuable ideas and feedback.

A big thanks to my family for being amazingly supportive. Especially my father Ken who's been helping me since the first day of school. I would like to dedicate my efforts to my mother Stine, and my sisters Nina and Marie.

I would like to thank the boys for keeping my spirits up during the whole 5 years.

Thank you Ye.

Table of Contents

1 Introduction 1
1.1 Cron 2
1.2 The shortcomings of Cron 2
1.3 Defining Observation Units (OUs) 3
2 Related work 3
3 Idea and requirements 5
4 Architecture and Design 7
4.1 The DaoCron data store 9
4.2 Task-files, the DaoCron variant of Crontabs 9
4.2.1 The basic task 9
4.2.2 The more advanced task 10
4.3 Scheduling tasks 10
4.4 Cron-like time-based scheduling 11
4.5 Dynamic N-every-M time-based scheduling 12
4.6 Task-based scheduling 14
4.6.1 Evaluating the output 14
4.6.2 Chaining tasks / Creating Automated Workloads 15
4.7 Handling network-dependent tasks 15
4.7.1 Specifying network-dependent tasks 16
4.7.2 How DaoCron handles network dependent tasks 16
4.7.3 Network queuing 18
4.8 Tasks and energy levels 18
4.9 More control to the user with DaoCronLib 19
4.10 Pre-evaluating tasks before scheduling 19
4.11 Reporting 20
4.11.1 Scheduling reports 20

4.11.2 Custom reports using DaoCronLib 20
4.12 Performing analytics and building models 21
5 Implementation 23
5.1 Technologies 23
5.2 DaoCron configuration and task-files 23
5.2.1 Configuration 23
5.2.2 Task-file 25
5.3 The database 28
5.4 Task-output parser 28
5.4.1 Parsing task results with DaoCron's built in parser 29
5.4.2 Parsing task results using custom Python scripts 29
5.5 Pre-evaluation and DaoCronLib 31
5.6 The analytics model 32
6 Evaluation 35
6.1 Scheduling 35
6.2 Evaluating how DaoCron handles unstable networks 38
6.3 Time spent calculating schedule times 41
6.4 Analytics model generation 43
7 Discussion 47
7.1 Customizability and user-control 47
7.2 The Cron-like time-based scheduling method versus the N-every-M time-based method 48
7.3 The power of chaining tasks 49
7.4 "Live" model building versus semi-cold analytics tool generated models 50
7.5 The problem with collecting analytics data and performing predictions 51
7.6 Running tasks in the cloud 53
8 Conclusion 55

9 Future Work 56
References 57

List of Tables

Table 1 - The specified tasks to be scheduled with their given scheduling method, the scheduling interval, if it should schedule a child task based on its output and if it should perform pre-evaluation before running. 36
Table 2 - The simulated networks and their base time-values. 39
Table 3 - The specified tasks with their respective network dependencies, detection method and schedule time 39

List of Figures

Figure 1 – An overview of the DaoCron architecture with relation mapping between the different components. 8
Figure 2 - Visualization of the schedule times for each task over the course of 12 hours. The figure shows the times at which tasks were scheduled (red dots) along with the expected schedule times (blue dots). For Task 2 the expected schedule times is 4 times every 4 hour, and it could be scheduled at any time. This is represented as a blue line, as the task could be scheduled at any point. 37
Figure 3 – Visualization of the schedule times, as red dots, along with the network disconnection period for the dependent network as a cyan line 40
Figure 4 - The same data from Figure 3 but only between 12:00 and 18:00 40
Figure 5 - The average execution time in milliseconds with error bars for the different types of time-based scheduling methods 42
Figure 6 - The average execution time in milliseconds with standard deviations for each time-based scheduling method compared to the execution time of the analytics tool 43
Figure 7 - Visualization of the energy samples generated by the energy simulator. It represents the energy levels of an OU over the course of 24 hours. A value of 1.0 indicated a fully charged battery, while 0.0 indicates a depleted battery 44
Figure 8 - A visualization of the energy model generated by the analytics tool. It shows each hour of the day with the average energy level for each hour. 45

List of Listings

Listing 1 - An example DaoCron configuration defined in JSON using all possible fields 24
Listing 2 - An example task-file specifying two tasks 26
Listing 3 - An example task output parser written in Python. The script implements the function with correct signature required by DaoCron. It will return a true or false value indicating whether the child-task should be scheduled or not 30
Listing 4 - An example implementation of a pre-evaluation script written in Python. It makes use of the DaoCronLib Python library when producing results for DaoCron 31
Listing 5 - An example snippet of parts of a network model. This listing shows the entries 15-17 of an example network model. Each hour has a sub JSON-object containing the four different quarters of the given hour along with an hour total value. 33

1 Introduction

Over the recent years we have seen how the Internet of Things and its evolving technologies has affected the industry and moved us towards more distributed autonomous systems. It has allowed us to build cheap yet quite complicated sensors at large quantities. This allows us to build large networks of sensors for monitoring data for large scale areas. It allows for viable solutions when it comes to monitoring of desolate areas like the arctic tundra, which can be both expensive and dangerous to visit. There are lots of interesting data to be collected from the arctic tundra. For biologists and climate researchers the data collected is very useful when monitoring the effects of climate change, as the arctic tundra is especially sensitive [1]. Currently, the researchers must manually deploy cameras, sensors and other observation tools to the field, configure them in the field and collect them again when the observation period ends. The Distributed Arctic Observatory (DAO) project is a cyber-physical system for ubiquitous data and services covering the arctic tundra, which aims to improve the data collection and analytics from sensors deployed at the arctic tundra. The research done by the DAO-project is based around an observation system which consists of numerous instruments monitoring the environment, called Observation Units (OUs). These instruments are intended to collect data, perform maintenance tasks and report their system status at certain intervals. The different functionalities are provided by programs, or tasks, which are responsible for performing the actual work. These tasks can be scheduled at different intervals using Cron, which is a tool for time-based scheduling of jobs in a Unix-like system [2]. However, Cron was never designed for the unstable and highly variating environments across the arctic tundra at which the OUs may be deployed at. Such as environments where energy supply may be scarce and network connectivity is only available for periods of time. For DAO-project use-cases the scheduler should be aware of these restraints. Cron only provides us with one scheduling method and is totally ignorant to the network and energy. This paper presents DaoCron, which is a task scheduling tool for OUs deployed at the arctic tundra. It aims to be highly configurable in order to adapt to the different conditions an OU may be deployed into, while also supporting the underlying scheduling method of Cron. It aims to bring network connectivity, energy and user control into the equation when scheduling tasks. It presents three methods of scheduling. The “Cron-like”, “N-every-M” and “task-based” methods. DaoCron can also be configured to monitor the network connectivity and energy levels and put tasks on hold during periods of network downtimes and low levels of energy, rescheduling them once the resource demands are met.

1.1 Cron

Cron is a tool for time-based scheduling of jobs in a Unix-like system [2]. Cron allows users to set up tasks, commands or shell scripts, which are to be scheduled periodically at fixed times. The tasks are provided to Cron through “crontab”-files. These are files containing the configuration of shell commands and their scheduling. Each crontab represent a single task and contains five fields for the time-values at which the task should be scheduled along with the command/shell script to execute. In all its simplicity the task is scheduled to run when the current time and date match the five “time-vector” fields provided in the crontab [3].

1.2 The shortcomings of Cron

Cron allows us to set-up tasks which are to be scheduled at certain time intervals through “crontabs”. These crontabs allows the user to specify certain time-vectors (minutes, days, months and years) at which the task should be scheduled. For instance, the user can set a task to be scheduled at the minute values 0 and 30, every hour, every day, every month and every year. This would result in the task to be scheduled every 30 minutes. The requirements would hold each time the minute value strikes 0 or 30, as the other time-vectors hold for all instances. This allows us to schedule tasks at every possible combination. However, Cron is completely indifferent to the current network connectivity, the energy level, task priority and similar factors which becomes highly important in an OU deployed in areas of unstable networks while relying on power from a battery with limited capacity. The scheduler should be able to adapt to the different conditions of the OU and its resources. Also, it should be able to be configured for different deployment conditions as well. An OU could be deployed far out in the arctic tundra where the network is only available at certain time-frames a day with a battery with energy levels varying. At the same time another OU might be deployed on the outside of a house, with constant access to WIFI and a stable energy flow from a wall socket. If the energy levels are low, DaoCron should only schedule the highest prioritized tasks specified by the user. A task which is dependent on network connectivity should be set on hold if a network is unavailable, picking it up again once the network becomes available. Cron will only schedule the tasks at the specific times the user provides. Which likely will result in failed tasks and energy wastage. DaoCron should provide the user with more control over the scheduling times rather than just providing time-values for the scheduling of tasks. Ultimately, the user should have total control when a task is scheduled if desired.

1.3 Defining Observation Units (OUs)

The DAO-project base their observation service around Observation Units (OUs), which is defined as an instrument monitoring the environment. Though OUs are heavily referred to as small computers with sensors deployed around the arctic tundra throughout this thesis the OU definition expands beyond that. As DaoCron is mainly designed for this use case, it should also be available for use cases beyond this. An OU could therefore be any type of computer/device, as long as it runs a UNIX-based system. However, for the rest of this paper an OU will mainly be referred to as a Raspberry PI running the Raspbian Stretch Lite operative system, as it has been used as the development platform throughout the development of DaoCron. Executable tasks are assumed to be anything which could be initiated from the UNIX command-line. The DaoCron system does not perform energy readings directly, we assume that there is some other service which dumps the energy levels to a file at certain intervals which DaoCron can then read.

2 Related work

DaoCron is inspired by Cron and it aims to achieve more customizability and user control than what Cron provides, as the functionalities of Cron does not take factors like network and energy into account when scheduling tasks. DaoCron is designed with the arctic tundra in mind, where energy may be limited, and network connections are uncertain. DaoCron is still designed to imitate the base functionality of Cron, like how tasks are defined through “crontabs” and how the time-based scheduling algorithm of Cron works. DaoCron could in some way be seen as an extension of Cron.

There are multiple other implementations of Cron, among these are “Anacron”, “Fcron” and “Vixie-cron” [4]. Each tackling different problems they had with Cron. For instance, Anacron is implemented to run tasks periodically on systems which are not meant to run continuously. Meaning that the system can be shut off for periods of time. Fcron is the variant that resembles DaoCron the most, as it expands the scheduling possibilities beyond Cron. However, none of these solutions are close to solve the problems of network and energy as none of these take these into consideration when performing scheduling. The amount of customizability in most, if not all, is too scarce for all the different environments a OU may be deployed at.

3 Idea and requirements

As mentioned, Cron does not provide us with functionalities which brings network and energy into play when performing task scheduling. It only provides a single strict time-based method of scheduling.

DaoCron should provide more customizability as a scheduler compared to Cron. In order to do so it should allow for different scheduling methods in addition to the time-based scheduling method of Cron. In order to stay true to the scheduling functionalities of Cron, DaoCron allows its users to opt for a "Cron-like" time-based scheduling method which reflects the scheduling functionalities of Cron while also bringing network and energy into the equation. The "Cron-like" scheduling method is considered strict as it will run at the exact time values provided by the user. For some use-cases the tasks might not be required to run at such strict intervals. Therefore, DaoCron should be able to schedule tasks using a less strict method which allows for manoeuvring of schedule times to avoid network and energy constraints. This can be done using what is called the "N-every-M" scheduling method. The user should be able to tell DaoCron to run a task *n* times every *m* time-vector using this scheduling method. Exactly when DaoCron schedules these tasks are not important as long as they are scheduled the correct number of times during the given period. The result of one scheduled task should be able to trigger the scheduling of another task based on the output. This can be done using DaoCron's "task-based" scheduling method. The user should be able to provide some requirements for the output of one task which will allow for DaoCron to schedule another task if these requirements are met.

A task may be dependent on reaching a network endpoint during its execution in order to succeed. Using Cron the task will be scheduled at the exact time values the user provides, resulting in a failed task if the network is disconnected. Therefore, DaoCron should be able to monitor different network dependencies and schedule task which are dependent on them only when the network is available. In order to avoid schedule times being missed due to the network connectivity DaoCron should reschedule the task once it detects the network becoming available again. The same goes for energy levels. Some tasks may be dependent on a minimum energy level before they are executed which DaoCron should keep track of.

In order to provide more user control than Cron, DaoCron should give the user a bigger role in the scheduling process if desired. The users should be able to provide their own scripts which DaoCron will evaluate before scheduling a task. Letting the users have the final saying

through their custom scripts. This should also be the case when evaluating the results of a task when performing “task-based” scheduling. The users should be able to provide their own custom result parsers, which decides on whether a task should be scheduled or not based on the result of another.

4 Architecture and Design

Figure 1 shows an overview of the DaoCron architecture. DaoCron is based around the DaoCron process (daemon), which is the main component responsible for orchestrating the system. The configuration file is loaded by the daemon upon start-up. It contains information about where the task-files, defining the tasks to scheduled, is located. It also includes information about whether the daemon should perform monitoring of networks and energy levels, perform reporting and/or perform analytics. The daemon does not read the energy levels directly itself, and for the sake of this thesis it is assumed that some third-party tool is writing the energy levels to a file which the daemon will check continuously (if configured to do so). The analytics tool is an executable which will be scheduled by the daemon. DaoCron provides an analytics tool process. However, users can opt out of using this implementation and rather use their own. The only requirement is that DaoCron can find and parse the generated models and that the executable is provided in the configuration file.

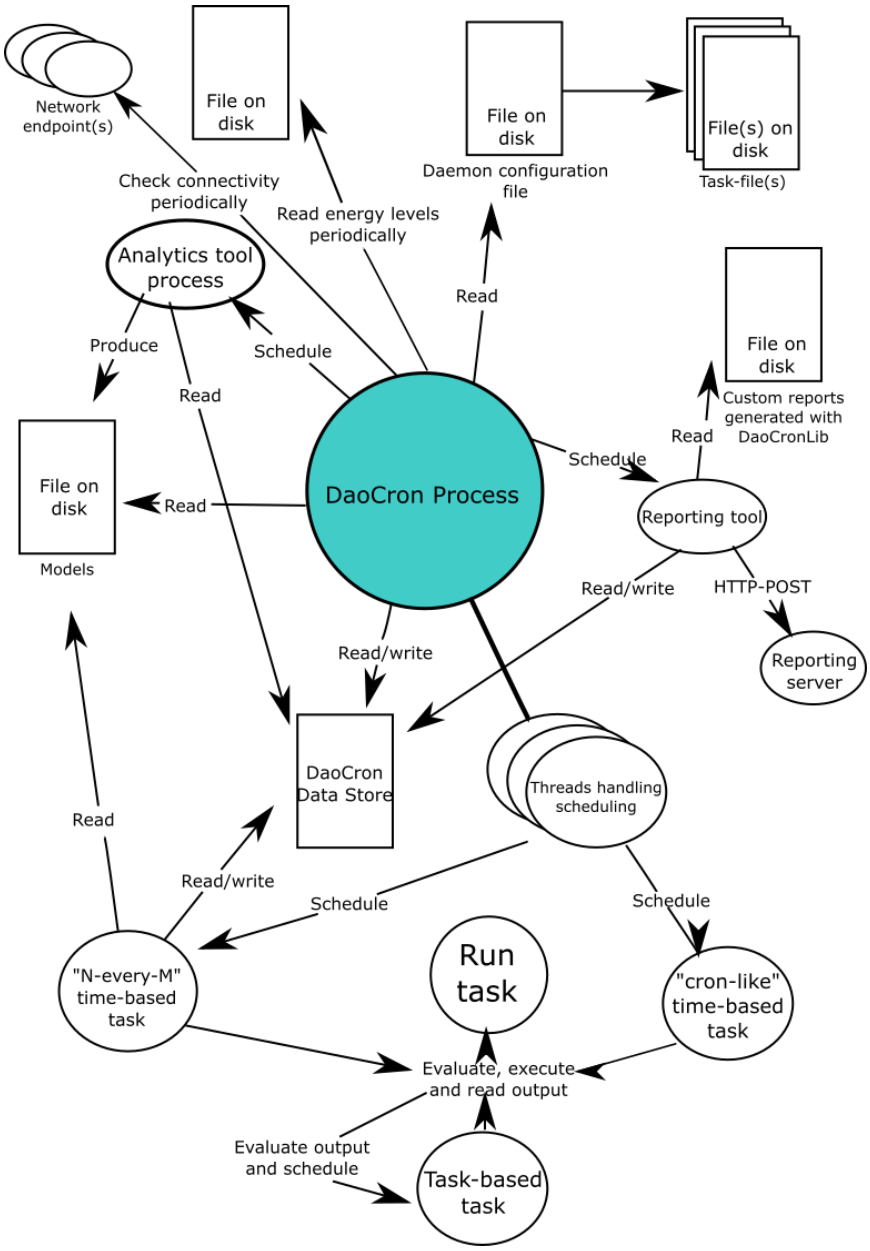


Figure 1 – An overview of the DaoCron architecture with relation mapping between the different components.

DaoCron will initialize and handle the threads which performs the scheduling of the tasks. There are three types of scheduling methods that can be given to a task. A task can be scheduled using a “Cron-like” time-based scheduling method, which is implemented to imitate the scheduling functionality of Cron. The “N-every-M” scheduling method is similar to the “Cron-like” method as it also schedules based on time-intervals. The main difference is that the “N-every-M” method does not enforce “strict” time-intervals, but rather a *n*-times every *m*. For example, a task may be scheduled to run twice within every three days. The task-based scheduling method schedules tasks based on the output of the parent task. The “Cron-like” tasks will be assigned their own thread which handles the scheduling. The “N-

every-M" tasks share one thread which performs the scheduling of these tasks. While the task-based tasks are not handled until another task produces an output which meets the requirements for the task-based task to be scheduled.

4.1 The DaoCron data store

A shared database is used by all the different components. The main use-case for the database is to log task executions, task statistics, errors, network-downtimes and similar. The database becomes a central part of the DaoCron architecture and serves as a useful resource for a variety of problems. The DaoCron daemon mainly uses the database for logging task executions and statistics. It's also a useful tool when scheduling tasks, though not used heavily directly but rather through the analytics tool. The analytics tool builds its models for tasks, networks and energy based on the entries found in the database. It also serves a purpose for the end-users as they can make use of the database when writing their tasks, custom pre-evaluation scripts or custom reports.

4.2 Task-files, the DaoCron variant of Crontabs

Cron makes use of their so-called "crontabs" when defining time-periods for which tasks are scheduled [3]. A crontab is a simple yet powerful way of specifying tasks and schedules, but for our use case they quickly become a bit too primitive. Since DaoCron tries to provide a task scheduler like Cron with more configurability and options, the DaoCron task-files becomes a central part in archiving exactly that. The task-files is DaoCron's equivalent of Cron's crontabs and it should be noted that these are used solely to define the tasks to run and that the configuration of DaoCron is specified in the configuration-file, which contains a list of file-paths to the task-files which should be loaded. A task-file consists of a list of task specifications. The variables required in defining a simple task are few while it gives a variety of optional fields which can be added to each task. As an example, the user may provide a task with a field indicating that the task requires access to a certain network-endpoint when executed. And since an OU may be deployed in an environment where the network connectivity is scarce and unstable, DaoCron will take this into account when scheduling the tasks.

4.2.1 The basic task

Setting up the simplest task only requires two fields in the task-file task specification. The required fields are an identifier and a string-value representing the executable, or in other words how the task should be initiated. Providing just these two field will not make DaoCron schedule the task, but it is aware of its existence. This task could be scheduled as a result of

another task's output, by referencing it by its task id. In order to make DaoCron schedule this task the user needs to specify a scheduling method to be used. The two options are the "Cron-like" time-based method, which schedules the task at specific times provided by the user. This requires the user to provide values for the different time-vectors (e.g. minute, hour, day, month, year) for which the task should be scheduled. The second option is to use the "N-every-M" scheduling method, which is similar but not as strict. The user can specify the task to run *n*-times every *m* time-vector. For example, five times within every 7 days.

4.2.2 The more advanced task

While the information required to set-up and start scheduling a basic task is relatively low, the user can provide additional information for each task which DaoCron should consider when scheduling the task. The user can provide each task with a priority level indicating the importance of the task relative to the other tasks. This is used by DaoCron when deciding which tasks to schedule in what order. For example, when multiple tasks are waiting for a network to become available, DaoCron will let the tasks with highest priority run first in order to avoid network contention. The user may also specify a task to be network-dependent. This tells DaoCron that the task requires a certain network-endpoint to be available in order to succeed. DaoCron will avoid scheduling the task when the network is not available (given that DaoCron itself is set to monitor the networks), and if unavailable it will keep it on hold until the network becomes available again. If the user wants to perform their own custom pre-schedule evaluation on whether the task is allowed to run or not, they can provide a field specifying so. This field requires information on how to run a evaluation script containing a specific function which returns the final decision on whether the task should be scheduled or not. Another option is to tell DaoCron to evaluate the output of the scheduled task and to schedule another task based on the given tasks output. DaoCron provides a few simple evaluation methods, but the user can also provide their own evaluation scripts in the same manner as with the pre-evaluation method.

4.3 Scheduling tasks

In its attempts to provide the user with more options and flexibility when it comes to handling and scheduling tasks, DaoCron provides three different methods for scheduling tasks. As DaoCron aims to improve upon Cron while staying true to its main concept, one of these solutions is implemented to mimic the scheduling method performed by Cron. Referred to as the "Cron-like" scheduling method, this method allows the user to specify time-intervals in the same fashion as when using Cron and crontabs. DaoCron also provides a more dynamic

and less strict method of scheduling referred to as “N-every-M” scheduling. Which in general aims to schedule a task *n*-times every *m* time-vector. This thesis uses the term time-vector quite frequently, which represent a vector of time like hour, day, week, month and so on. An example of “N-every-M” scheduling could then be to schedule a task to run twice within every five hours. Exactly when the task is scheduled is not important as long as it is scheduled the correct number of times within the given time-frame. This gives DaoCron more freedom which makes dealing with both network and energy related issues easier. The last scheduling method is the task-based scheduling method. Which basically tells DaoCron that if the result of a given task holds certain requirements, schedule this other task. Ultimately allowing for task workloads.

4.4 Cron-like time-based scheduling

Cron allows the user to specify so called “crontab” files. These files contain information on Cron-entries and when they should be scheduled to run. In all its simplicity, tasks can be scheduled to run at certain minute, hour, day and month values. For instance, by setting the minute values to 0 and 30, the hour value to 15 and 16, and the rest to a value indicating every occurrence, a task would be scheduled to run at 15:00, 15:30, 16:00 and 16:30 every day of the year. DaoCron allows for tasks to be scheduled in the same fashion.

For each task the user can specify an entry with list of integer values for each time-vector (e.g. minute, hour, day) at which the task should be scheduled at. As an example, the minute values can contain every integer between 0 to 59, representing all possible minute values. All time-vectors can also be given a value which indicates that the task should be scheduled for all instances of this time-vector. Specifying the hour value with this indicator means that the task will be scheduled every hour. However, this is affected by how the rest of the values are set. If a task is set to be scheduled for every instance of the hour value and for the minute values 15 and 45, the task will be scheduled every half hour. Likewise, if the minute values are set to 15 and 45, but the hour values to 0, 6, 12 and 18, the task will be scheduled twice every sixth hour.

Each task which is specified to be scheduled using this scheduling method will be assigned its own thread which is responsible for calculating the next scheduled time. When finding the next time at which the task should be scheduled, the scheduler starts off by creating a new datetime which is just one minute from the current. It will then validate if each time-vector of this new datetime is within the values provided in the integer-lists for that given time-vector.

It will start of by validating the highest time-vector, which is the year value, and move down to the lowest time-vector, the minute value. For the datetime 15:00:00 – 03/30/2019, it will first validate if the year 2019 exists in the task's valid year values, moving on to check if March (03) is within the valid month values and so on. It will use this new datetime as the next datetime the task should be scheduled if each time-vector in the new datetime is within the valid time-vector values. If any of the time-vectors is invalid, it will calculate a new datetime based on a time-vector delta value, which represents the closest value where the given time-vector value would be valid. Let's imagine a situation where a task just finished execution and needs to calculate the next time it should be scheduled. The current time is 15:00 03/30/2019 and the valid year values are 2020 and 2021, with valid months 02 and 03, while the rest of the time-vectors are set to be valid for every instance. In the process of finding the next time the task should be scheduled, the scheduler would first start off by creating the new datetime at 15:01 03/30/2019. In the validation process of this new datetime it would invalidate the year 2019 as it starts with the biggest time-vector, the year, and 2019 is not within the list of valid datetimes. The scheduler creates a new datetime at 00:00 01/01/2020, which is the lowest datetime that can be created from the closest valid value of that time-vector. It would then redo the datetime validation process using this new datetime, starting with the highest time-vector value, the year, moving down to the lowest, the minute. The year value, 2020, of the new datetime is considered valid and it will move on to validate the month value. The month value, 01, is invalid as the only valid month values are 02 and 03. It will again create a new datetime, which represents the lowest possible datetime using the year and month values, giving the datetime 00:00 02/01/2020. Again, it will start-over the validation process. This time all time-vectors would be considered valid, as the day, hour and minute specifications were set to be any. Therefore, the next time the task would be scheduled is at 00:00 02/01/2020.

This scheduling method is considered a strict time-based scheduling method, as the task will be scheduled exactly at the values specified by the user in the task's configuration.

4.5 Dynamic N-every-M time-based scheduling

Similar to the "Cron-like" scheduling method, the user can define a task to be scheduled using the "N-every-M" method. This method aims to be a less strict time-based alternative. Allowing the user to specify the number of times a task should be scheduled within a time-frame, not necessarily needing the task to be scheduled at an exact time. The user can specify the task to run *n*-times every *m* time-vector (minute, hour, day, month, year). A multiplier can

be added to the time-vector to allow for any time-frame to be specified. For instance, a task could be scheduled to run 5 times every 10 days. Where 5 is referred to as the frequency, days as the time-vector and 10 as the time-vector multiplier.

The N-every-M scheduling method is intended for tasks which are scheduled at a less frequent rate, as it is designed to be combined with the analytics tool in an effort to minimize the chances of scheduling the task at a time where the network is disconnected, the energy levels are not sufficient or at time periods where the task historically has had a higher rate of failure. The analytics tool creates simple models for the network, energy and each unique task which DaoCron will use when scheduling N-every-M based tasks.

The algorithm for scheduling a N-every-M based task starts of by calculating the max datetime ahead using the time-vector and time-vector multiplier provided by the user. It then calculates the total amount of minutes between the current datetime and the max-datetime, dividing this by the user-given frequency. If DaoCron is not configured to use analytics, there is not much more to the N-every-M scheduling algorithm. It might add a random noise in terms of minutes based on a percentage of the average number of minutes between each scheduled time, in order to avoid tasks with similar scheduling specifications to be scheduled at the exact same time. However, if DaoCron is configured include analytics in the scheduling process it will load the network model (if task is network-dependent), the energy model and the respective task model for the given task (if any built yet). The different models are built up by a map of each hour of the day with each quarter within the given hour included. The analytics tool has built the models so that each entry represents a number of times the network has been disconnected, the average energy or the number of times the task has failed before for each hour-quarter combination. The scheduler will find all hour-quarter combos within the time-window the task is to be scheduled. For instance, if the task is to be scheduled sometime between 18:00 and 19:00, the scheduler will load the entries for 18:00, 18:15, 18:30 and 18:45 from the different models. It will then build a score for each of the possible quarters based on the values stored for each quarter in the different models. For the network model the scheduler will sort the entries by the lesser amount of disconnections historically. Given 4 possible entries, the entry with less historical disconnections will be given a score of 4, the next best a score of 3 and so on. Similarly, the quarters in the energy model with the highest average energy level will be given the highest score and the quarter in the task-model with the fewest historical errors will also be granted the highest score. The scheduler will then combine the scores for each quarter from every model and choose the hour-quarter

combination with the highest score as the quarter the task should be scheduled within. After an hour-quarter combination is assigned, a random number of minutes between 0 to 15 will be added in order to create noise and avoid overlapping tasks.

4.6 Task-based scheduling

In certain situations, the user might want to schedule tasks based on the result of other tasks rather than a set of time-coordinates. DaoCron allows us to specify tasks which will trigger the scheduling of other sub-tasks based on the output the specified task produces. An example of such a task could be one which reads the disk usage of the system, and if the result is higher than a certain threshold it will run another task which performs some memory clean-up to allocate more memory. The task which is initially executed, and performs the evaluation, is referred to as the "parent-task". The task initiated after the evaluation by the parent-task is referred to as the "child-task". A parent-task is defined as any other DaoCron task. It can be scheduled using time-coordinates, or it might be initiated by some external event. The child-task can be defined as a single command which is to be executed by the shell, just like the executable of a DaoCron task. One can also insert the unique identifier of another DaoCron task into the child-task, allowing for the more configurable approach of scheduling tasks using DaoCron and the DaoCron task definitions.

4.6.1 Evaluating the output

Before scheduling the child-task to run, DaoCron must evaluate whether the parent-task output meets the required conditions given. The user can configure how the output should be evaluated by DaoCron. One option is to provide a datatype and evaluation string in the task-specification. The datatype-field indicates what type of output the task produces, DaoCron supports basic datatypes like integers and strings. The evaluation string is a user-provided string which specifies how DaoCron should evaluate the output. The string consists of one operator and a value separated by a space. An example evaluation specification could be as follows:

$$\text{“}ge\ 100.0\text{”}$$

When the datatype is set to be a floating point this evaluation string will schedule the child-task if the output of the parent-task is greater or equal to 100. Where the "*ge*" indicates a greater or equal operator and "100.0" is the floating-point value to evaluate against. Numeric datatypes support similar operators like "*lt*" (less than) or "*eq*" (equal), while string datatypes implements other evaluation operators like "contains" which will schedule the child-task if

the output string of the parent-task contains a given value. All different operators are described in the documentation of the code.

While this allows the users to perform simple evaluations on a handful of predefined datatypes, one might have more complicated needs of evaluating the output of the parent-task. Therefore, DaoCron allows the users to write custom Python scripts which will evaluate the task output precisely like the user requires. The Python scripts are required to contain a function with a specific name which will return either “true” or “false”, based on whether the task should be scheduled or not. DaoCron will dynamically load the Python module containing the evaluation function at runtime and call the function. DaoCron will provide the task output as a string as the only parameter to the function. Which the user then can evaluate in any way desired. In order to use custom evaluation scripts, the user will only have to provide a file-path to the script in the task-specification which will automatically override DaoCron’s built in evaluation.

4.6.2 Chaining tasks / Creating Automated Workloads

Since a DaoCron task identifier can be provided as the child-task, the child-task can again be specified to perform output evaluation and schedule new tasks. Meaning that the tasks can in theory be chained to infinity. Thus, creating an automated workload based on tasks. This allows the user to build automated workloads through a series of tasks, where each task can easily be exchanged for another by referencing another task identifier. Tasks can be references by several other tasks creating a layer of abstraction for the user, as combining tasks into more complicated workloads is as simple as referencing the task identifiers of the tasks which should be included. A task can even reference itself, allowing the task to be scheduled continuously until a condition holds.

As an example, one could set a parent-task to be scheduled to run every 10 minutes. This parent-task is specified to schedule another task, by its task identifier, if the output fulfils some conditions. This child task might again be specified to schedule yet another task, by another task identifier, based on its output.

4.7 Handling network-dependent tasks

When deployed to remote locations, like the arctic tundra, the availability and quality of networks may vary. The OU might find itself lacking required networks during periods of time. The availability can be dependent on many factors. Like having to wait for a satellite to fly above the area or waiting for Unmanned Aerial Vehicles (UAVs) used by the DAO-

project [5]. Such UAVs are intended to provide backhaul network access and energy to OUs. They will manoeuvre close to the deployed OUs providing network connectivity for a limited time. It is therefore important that the tasks which has been waiting for the network is able to run once this network is detected. Using Cron this window could easily be missed as it is oblivious to the presence of the newly available network.

At the same time the OU may be deployed at locations where the network is not a problem at all, and DaoCron should therefore be able to handle both possibilities. Tasks given to a scheduler may have to be able to reach a certain network-endpoint in order to successfully complete. Cron does not provide us with any functionality for handling such cases. Scheduling such tasks when the network is not available would just result in a failed task. DaoCron allows the user to specify tasks as network-dependant and will use this information when scheduling tasks.

4.7.1 Specifying network-dependent tasks

In order to inform DaoCron that a task is dependent on a network the user must provide this information under the given task in the task-file. The entry describing the given task must be given a sub-object named *"network-dependent"*, which contains further information on how this network can be reached. As long as this entry exists in the task-entry, DaoCron will handle this task as network-dependent. In the entry describing the tasks network dependency, the user can provide fields for the address the task is dependent on along with a method of communication. The method of communication specifies how DaoCron will confirm its availability. It can be done in two ways, pinging the IP-address of the endpoint or performing an HTTP-request directly to the HTTP-endpoint provided in the address field. The user can give the address value of *"www"* to specify that the task is just required to reach the world-wide web, and not some specific endpoint in general. By using this for any general-Internet dependent tasks, and specific endpoints for special networks like LANs, the number of networks DaoCron needs to monitor is reduced. Resulting in fewer network calls and less energy usage.

4.7.2 How DaoCron handles network dependent tasks

There are different ways DaoCron handles network-dependent tasks. This section describes how DaoCron monitors the network availability, adds and removes tasks from the scheduler and handles tasks which missed their schedule-time because of network disconnection. How

DaoCron schedules tasks based on network-availability models and prediction is described in the analytics and "N-Every-M scheduling" sections.

When DaoCron starts up and reads all the tasks from the task-file(s), it will create a map of all the different network-addresses and methods the tasks are dependent on. The map is represented as a dictionary where the address and method combined represents the key, and a network-state object value which holds information about the endpoint, its availability and a queue of tasks waiting for it to become available. Any tasks which has been set as network dependant but not provided an address will by default use the *"www"* endpoint, telling DaoCron that it only requires to be able to connect to the Internet in general.

DaoCron will not actually monitor the networks unless it is specified to do so in the configuration. In the general DaoCron configuration-file the user must provide a *"monitor-network"* entry. This entry tells DaoCron that it will monitor the networks that the tasks are dependent upon and keep a map of their availability. The user must provide an interval-field to this entry, representing how often DaoCron should check the connectivity of each network. The user can also provide a default address, which will be used for the *"www"* endpoint.

While DaoCron is running it will monitor these network endpoints at the interval given in the DaoCron configuration file. Based on the method provided in the network dependency entry in the task specification, it will either perform a ping of the IP-address or a HTTP-GET request directly to the endpoint. If the ping responds or the HTTP-request status code is 200 the network will be considered as available. If not, it will be considered unavailable. DaoCron will perform such a request for each unique address-method combination provided by the tasks.

If the monitoring reveals that the network state is different from the currently set state, as if it went from available to unavailable, DaoCron will retrieve the network-state object from the network-map based on its key and change its availability state. Based on the new availability state, the network-state object will also keep track of the disconnection period. If the network becomes unavailable it will log the time it went down. Then if the network becomes available again, it will log the full disconnection period into the DaoCron database which is later used by the analytics tool when building the models. As DaoCron keeps an internal map of tasks allowed to run, it will filter these tasks based on the new network state. If a network becomes unavailable every task which is dependent on this network and is currently allowed to be

scheduled will be filtered out of the scheduling-map. Refusing it to be scheduled until the network becomes available again. Likewise, if a network becomes available the tasks which was previously not allowed to be executed will be added to the scheduling-map.

4.7.3 Network queuing

As a network becomes unavailable DaoCron will remove any task dependent on that network from the scheduling-map, refusing it to be scheduled. If a task was supposed to be scheduled during the down-period of the given network, it will be considered as a missed schedule. The network-state object containing the info about the given network-endpoint also contains a task queue which will hold all tasks which missed a schedule due to the network being unavailable. So once a task misses an execution due to the network it will be added to this queue. Each network-state object contains its own queue representing that specific network. The correct queue for the task is found by using its network dependent address and method as a key into the scheduler-wide network-map. This key will return the correct network-state object and the task will be added to its queue.

Once DaoCron notices that a network becomes available again it will fetch the network-state object representing the given network and get all awaiting tasks. The network-state object will provide the scheduler with the task queue sorted by the task's priority levels. The scheduler will then schedule the tasks one at the time based on the priority levels, running the highest prioritized tasks first and lowest last. Scheduling all tasks at once may result in network-contention, where all the tasks would eat up all the bandwidth. It should be noticed that a task which is scheduled by the network queue will still have to uphold all its other requirements for running. As an example, a task may be scheduled after being queued for a network to become available, but it turns out that the minimum energy required for the given task was not met. The task would not be executed, but rather just transferred from the network-queue over to the energy-queue.

4.8 Tasks and energy levels

The functionality behind handling energy restricted tasks is quite similar to the one for handling network dependent tasks. The user may specify a task to have an energy restriction, a minimum required energy level in order to be scheduled, in the task-file. The user will also have to configure DaoCron to perform energy monitoring through the DaoCron configuration file. If DaoCron does not perform energy monitoring it will not be aware of the current energy level and the tasks with energy restrictions will not be removed from the scheduler if the

energy level drops below the minimum requirement. Every time DaoCron detects a change in the current energy level it will filter out the tasks which are not within the new energy level or add tasks to the scheduler which previously was evicted based on the energy level. If a task is scheduled to run during a period of unmet energy requirements it will be added to an energy queue. Tasks in the queue will be removed from the queue and scheduled once their energy requirements are met. It should be noted that a task which is scheduled from the queue may be added to the network queue if the tasks network dependencies are not met. This could result in a task being thrown between the network and energy queue if the both requirements are conflicting. If multiple tasks are pulled from the queue at the same time, DaoCron will schedule the tasks one at a time based on their priority levels. The task with the highest priority level will be scheduled to run first, while the task with the lowest priority level will be scheduled last.

4.9 More control to the user with DaoCronLib

DaoCron aims to provide more customizability and control for the user when scheduling tasks. One effort in achieving this is the implementation of DaoCronLib. DaoCronLib is a Python library which the user can import when writing task-output parsers, pre-evaluation scripts and tasks in general. The goal of DaoCronLib is to give the user even more control of the scheduling of tasks. For instance, as the user can write pre-evaluation script which are loaded and executed by DaoCron right before scheduling a task, the user can make use of DaoCronLib to provide DaoCron with more information than just the basic run or not (true/false return type). DaoCronLib may tell DaoCron to schedule the task right away, while also schedule it for a number of datetimes provided by the user through the script.

4.10 Pre-evaluating tasks before scheduling

Though DaoCron tries to schedule tasks in the best possible way by evaluating different factors before scheduling, the user should ultimately be able to have the final word in this process. Therefore, the user can specify in the task-file that a task is to perform pre-evaluation right before running the task. This pre-evaluation is done in a user provided Python script which performs the final decision whether the task should be executed or not. The user must provide a Python script containing a specific function which returns either a Boolean value or a DaoCronLib data-structure informing DaoCron on its final decision. What the pre-evaluation script does between the function call and the returned value is up to the user. DaoCronLib provides functionality which gives the user more control that just the basic true or false option. The basic evaluation result produced needs to contain the decision on whether

the task should run or not. By using DaoCronLib the user can include other information as well. The user may tell DaoCron to run the task, but to also schedule it at certain times in addition to this original execution.

4.11 Reporting

As DaoCron is designed to run on OUs, the task of monitoring the health of DaoCron and its scheduled tasks becomes increasingly tedious for each new OU deployed. To minimize manual labour DaoCron provides a functionality for automatically building reports and sending them to any specified RESTful server over HTTP. The reports will contain fields representing the time-period the report covers, the number of tasks scheduled, the number of errors and a user specified OU identifier. If any errors have occurred during the given report period, the report will also contain an error-field which lists all error instances. In an effort to send as little data over the network as possible the report will count similar errors and send one instance along with the count of similar errors, rather than sending each recurring error. The user can specify three different levels of detail for the reports. These are low, medium and high detail levels. Using the low-detail option, the report will only contain the OU identifier, the given time-period of the report, the number of tasks scheduled and the number of failed tasks. The medium-detail level will add the error-field which contains the list of different errors. When using the medium-detail option the error-instances will include information about the task identifier, the error message and the number of similar errors. Opting for the highest detail level will include the datetime the errors were scheduled along with the full stack trace.

4.11.1 Scheduling reports

Once the user specifies that DaoCron will be performing reporting it will schedule the reporting using a variant of the N-Every-M method. It will schedule the reporting at a user provided frequency, using the network and energy models in order to predict the best time-frame the task should be scheduled within. This is given that DaoCron is set to actually perform network or energy monitoring.

4.11.2 Custom reports using DaoCronLib

The basic report data generated by DaoCron's reporting functionality might not be enough for some users. They may wish to include their own data directly from the executing tasks as well. DaoCronLib provides users with a set of functions to use in their tasks. By using the provided interface one can append their own data onto the existing DaoCron generated report. DaoCronLib will generate a report file on disk which holds all the user generated data. When

DaoCron performs a report, it will read the user-provided data and append it into its existing data. The user becomes responsible in controlling the amount of data that is sent over the network, as DaoCron will not combine similar entries as it does for the error logs. However, DaoCronLib provides functionality for determining if older instances of data should be overwritten or not when new data is added. The user can therefore determine if data-fields should be overwritten or added with a tag representing the iteration number. Once DaoCron has successfully sent the report data it will wipe the built-up user defined report data.

4.12 Performing analytics and building models

DaoCron can be configured to use and schedule the analytics tool. The analytics tool creates a simple model for the network connectivity, the energy level and the error-rates for each task. The model is outputted to a file which DaoCron will load into memory. The different type of models all shares the same basic structure. Each model consists of a map of each hour within a day, from 0 to 23, where each entry has four sub-entries each representing one of the four quarters within that given hour along with a field holding the total value of the four quarters. For the network model each entry holds the number of times the network has been disconnected within the given hour-quarter combination. The energy model entries hold the average energy level for each combination, while the task model holds the number of times the given task has failed within that time-frame.

The version of DaoCron that is provided with this thesis comes with an implementation of a simple analytics tool, but the user can also write their own analytics tool which DaoCron can schedule instead. The given implementation will start off by reading the logs generated by DaoCron in the DaoCron database. If DaoCron is configured to monitor the network connections it will write the start -and end-datetime pairs of each disconnection-period into a table in the database. The analytics tool will load all these entries and calculate all the hour-quarter combinations this period covers and sum up the number of times the network was disconnected for all combinations. Similarly, it will calculate the average energy levels for all hour-quarter combinations based on the logged entries in the database. It will create a map of all tasks logged in the database and sum up the error-logs for the given task for all hour-quarter combinations.

5 Implementation

5.1 Technologies

DaoCron is implemented using Python and requires version 3.6.2 or higher. Python is an open-source high-level programming language which supports multiple programming paradigms, including object-oriented and functional programming [6]. The DaoCron implementation heavily relies on the functional programming paradigm, though still far from a fully functional programmed implementation. Most of the code is still based on the object-oriented paradigm. Functional programming techniques can reduce the complexity of applications, which often leads to fewer malfunctions [7]. This is especially useful for Internet of Thins applications like DaoCron running on the OUs, as updating such applications deployed at the arctic tundra may be difficult considering network and accessibility limitations. DaoCronLib is implemented as a Python module as has the same requirements as the DaoCron implementation. The user written scripts for task result parsing and pre-evaluations is required by DaoCron to be written using Python as DaoCron will dynamically load the Python scripts and access its code. The tasks which are scheduled to run by DaoCron can be any executable which is callable from the UNIX command line.

The deployment target for this thesis is an out of the box Raspberry PI 3 B+ which will represent an OU. The Raspberry PI runs the April 2019 version of Raspbian Stretch Lite. The task which are to be scheduled can be any executable which can be initiated from the Unix command line. As DaoCron will start tasks as command-line subprocesses, a task can be specified as any command-line input. All stored data is represented using either JSON or through the SQLite3 database.

5.2 DaoCron configuration and task-files

5.2.1 Configuration

As stated, DaoCron uses JSON-formatted files in order to specify the configuration of DaoCron and the specification of tasks. Listing 1 shows an example DaoCron configuration file, which includes all possible configurable fields.

```
{
    "taskfiles" : [
        "/path/to/taskfile.json"
    ],
    "monitor-network-connection" : {
        "interval" : 600,
        "address" : "www.google.com"
    },
    "monitor-energy-levels" : {
        "interval" : 600,
        "input-file" : "/path/to/file/containing/energy/levels"
    },
    "perform-analytics" : {
        "min-scheduled-since-last" : 50,
        "exec" : "python3 /path/to/analytics.py -some arguments"
    },
    "reporting" : {
        "detail-level" : "medium",
        "address" : "http://somereportingserver.com/endpoint",
        "ou-identifier" : "Some unique identifier for the OU",
        "min-samples" : 30
    }

}
```

Listing 1 - An example DaoCron configuration defined in JSON using all possible fields

Task-files

The configuration file has only one required field which is the *"taskfiles"* field specifying the file-path(s) of the task-files DaoCron should load. It is represented as a JSON-list of strings, each entry representing a file-path to a JSON-file which contains task specifications.

Monitoring networks

If the field *"monitor-network-connection"* is provided in the configuration file, DaoCron will continuously monitor the different network dependencies of the tasks. The user can specify the interval at which DaoCron should check the different networks, using the *"interval"* field, which is represented in seconds. The *"address"* field specifies the address which DaoCron should check for tasks which have specified only that they require Internet access but no specific endpoint.

Monitoring energy

Similarly. the user can provide a field indicating that DaoCron should monitor the energy levels of the OU. The user can also here specify a *"interval"* value indicating the interval in

seconds DaoCron should check the energy levels. Note that as DaoCron does not directly monitor the energy levels itself, it requires a file-path to a file containing the energy levels.

Performing and using analytics

Providing the field *"perform-analytics"* tells DaoCron to schedule the analytics tool and make use of the generated models when scheduling N-every-M tasks, reporting and the analytics tool itself. The user can provide a minimum number of task-schedules which has to be executed before rebuilding the models. The user can also specify the command-line executable to use when executing the analytics tool.

Reporting

By adding the field *"reporting"* to the DaoCron configuration file DaoCron will build and send reports at frequent intervals. The user can specify the values *"low"*, *"medium"* and *"high"* for the *"detail-level"* field, telling DaoCron how much detail to include in the generated reports. DaoCron needs to know where to send the generated reports, which the user can specify in the *"address"* field. Assuming the user deploys several OUs, the report can be labelled with an OU identifier, making it easier to separate the reports. The user can also provide a minimum number of samples required before the report should be sent, specified in the *"min-samples"* field.

5.2.2 Task-file

Listing 2 shows an example task-file which specifies two tasks, one with all possible fields specified and one with just three fields provided. The bare minimum of any task is that it has an *"id"* field and an *"exec"* field. The first specifying the unique identity of the task and the latter the command-line input for running the task.

```
[
    {
        "id" : "absc14t1",
        "exec" : "python3 sometask.py -some arguments",
        "priority" : 2,
        "network-dependent" : {
            "endpoint" : "http://127.0.0.1:51024/some/endpoint",
            "method" : "http"
        },
        "min-energy" : 0.5,
        "pre-evaluation" : "/path/to/pre-evaluation-script.py",
        "time-coordinates" : {
            "minute" : [ 0, 10, 20, 30, 40, 50],
            "hour" : [ 0, 3, 6, 9, 12, 15, 18, 21 ],
            "day" : [ -1 ],
            "month" : [ -1 ],
            "year" : [ -1 ]
        },
        "task-coordinates" : {
            "child-task" : "51ca156a",
            "output-type" : "integer",
            "conditions" : "ge 100",
            "python-script" : "/path/to/parser-script.py"
        },
        "n-every-m-coordinates" : {
            "frequency" : 4,
            "time-vector" : "hour",
            "multiplier" : 7
        }
    },
    {
        "id" : "51ca156a",
        "exec" : "echo testtest",
        "task-coordinates" : {
            "child-task" : "cd /some/folder && rm -rf subfolder",
            "output-type" : "string",
            "conditions" : "contains test"
        }
    }
]
```

Listing 2 - An example task-file specifying two tasks

Network dependency and energy level

Through the field *"network-dependent"* the user can provide a JSON-object describing the network dependency of the task. If no sub-fields of this object are given, the default values

will be to ping the default Internet address (referenced as the *"www"* value). Providing the *"method"* sub-field informs DaoCron which method it shall use when checking the availability of the network. The accepted values are *"ping"* and *"http"*. The *"address"* field provides DaoCron with a network address the task is dependent upon. When choosing a minimum energy level required for a task to run the user can provide a floating-point value to the "min-energy" field between the values 0 and 1. Representing the minimum energy percentage required.

Cron-like scheduling

The *"time-coordinates"* JSON-object specified the time-vectors and values the task should be scheduled at using the "Cron-like" scheduling method. Including this object will tell DaoCron that this method is to be used to schedule the task. The different sub-fields represent the different time-vectors, each containing a list of integer values. The integer values indicate the values for each time-vector the task should be scheduled. The user can provide the value -1 in order to specify that the task should be scheduled for all instances of this time-vector.

Task-based scheduling

The field *"task-coordinates"* indicates that the given task will evaluate its output and schedule a child task based on the evaluation. The *"child-task"* field can be set to a command-line executable or the *"id"* field of a task. This tells DaoCron what to run if the evaluation holds. The *"output-type"* and *"conditions"* fields provides information on how DaoCron should parse the task output. If the user provides the *"python-script"* field, these other fields are ignored. This field tells DaoCron where it can find the parser-script that should be called in order to evaluate the output.

N-every-M scheduling

Like the Cron-like method, the user informs DaoCron to use the N-every-M scheduling method by providing the JSON-object with the field *"n-every-m-coordinates"*. This requires a string field specifying the time-vector and one for the frequency. The time-vector can be multiplied with the *"multiplier"* field.

5.3 The database

The database which DaoCron, DaoCronLib and the analytics tool utilizes is based on SQLite3. SQLite3 is an in-process library that implements a self-contained, zero-configuration, serverless, transactional SQL database engine [8]. The main use-case for the database is storing historical data such as task schedule times, task failures, energy levels and network downtimes. The database consists of 7 tables. DaoCron uses one table for logging all the scheduled tasks. This includes the executable, task identifier, scheduled datetime and exit code. DaoCron also measures the time elapsed from the task was scheduled until it completed successfully. The elapsed time is stored in milliseconds along with the task id and scheduling datetime in a table dedicated to task statistics. Failed tasks will be logged to the "error log" table. DaoCron stored the task identifier, the error message, full stack trace and scheduled time in the table. This data is used by both the analytics tool and the reporting tool. When DaoCron is configured to monitor the network dependencies it will measure the down-periods of networks. This is stored as a entry into a table dedicated for network downtimes, represented as a column for the start-datetime and one column for the end-datetime, along with the address of the network endpoint. The analytics tool will use this information to build the network models. DaoCron also keeps track of analytics tool -and reporting executions in their own respective table. Here DaoCron logs information such as the schedule time, exit-code and error messages. The last table DaoCron utilizes is the energy level table. When DaoCron is configured to perform energy monitoring it will log each read into the database, represented as the energy level and the read datetime. This information is pulled from the database by the analytics tool when building energy models.

5.4 Task-output parser

As mentioned, a task can be configured to schedule another task based on its output. In order for DaoCron to decide whether the task should be parsed or not it needs to know how it should be parsed. DaoCron implements two options for this problem. The first option is to use DaoCron's built in parser functionality. This requires the user to specify the data-type the output should be parsed as, along with information on how DaoCron should evaluate the output. The second option is for the user to write a custom Python script which contains a specific function which will return the result to DaoCron, optionally with the help of DaoCronLib

5.4.1 Parsing task results with DaoCron's built in parser

When the user specifies a task to perform scheduling based on its output, which is done through the "task-coordinates" object in Listing 2, the user also has to specify what data-type the output produces ("output-type" in Listing 2) and how it should be evaluated ("conditions" in Listing 2). DaoCron allows for three different data-types to be provided, which is integer, floating-point and string. When specifying the required conditions, the user must provide a string with a format containing first the operator and next the value, as demonstrated in "conditions" in Listing 2. In Listing 2 the "conditions" parameter has been set to "ge 100". Here "ge" is considered the operator of the condition, where "ge" represents the operator "greater than", and the value "100" is considered the value related to the operator. For the example in Listing 2, the task will parse its output as an integer and schedule the child-task if the outputted value is greater than 100.

5.4.2 Parsing task results using custom Python scripts

If the "task-coordinates" object in the task specification contains the field "python-script", as it does in Listing 2, it will override the DaoCron built in parser functionality and use the user provided Python script instead. The field needs to represent the file-path of the script which DaoCron will load dynamically at runtime. DaoCron has a few requirements when using custom Python scripts. DaoCron requires the script to contain a function named "parse_output" which takes exactly one argument. The function argument is the task output represented as a string value, which DaoCron will call the function with.

```
from datetime import datetime

#function with required signature
def parse_output( task_output ) -> bool :

    try:

        #parse task output as datetime
        output_as_datetime = datetime.strptime(
            task_output,
            '%Y-%m-%dT%H:%M:%SZ' )

        #if parsed date is greater than current datetime
        #return True indicating the child task to be scheduled
        if output_as_datetime > datetime.now() :
            return True
        #if not return False
        #indicating that the child-task should not be scheduled
        else:
            return False

    # if datetime parsing fails return False
    #indicating that child-task should not be scheduled
    except:
        return False
```

Listing 3 - An example task output parser written in Python. The script implements the function with correct signature required by DaoCron. It will return a true or false value indicating whether the child-task should be scheduled or not

Listing 3 shows an example task output parser script. The script implements the required function “parse_output” which takes the required string argument containing the output of the task provided by DaoCron. DaoCron will dynamically load this script in runtime, along with the required libraries and other functions specified in the Python file. This allows the user to create more complicated parsers than the one defined in Listing 3. The function in Listing 3 tries to parse the task output as a datetime with a specific signature. If the parsed datetime is greater than the current datetime it will return a “true” value back to DaoCron, indicating that the requirements are met and DaoCron should schedule the task. If the parsed datetime is less than the current datetime or the parsing fails it will return a “false” value back to DaoCron, indicating that the requirements are not met and DaoCron will not schedule the child-task.

5.5 Pre-evaluation and DaoCronLib

Similar to how the user can write custom task output parsers, the user can also provide a pre-evaluation script to DaoCron. The user will have to specify the file-path to the Python script in the task specification under the field “pre-evaluation” as shown in Listing 2. If a task has been assigned this field in the task-file, DaoCron will dynamically load the script right before the task is scheduled and schedule the task based on the results produced by the function in the script. Just like the parser scripts, the pre-evaluation scripts have a specific function requirement which DaoCron will look for when loading the script. The function is required to have the name “pre_evaluate” and takes no arguments. Similar to the parser scripts, this function should return either a true/false value or a data-structure provided by DaoCronLib.

```
from daocronlib import daocronlib
from datetime import datetime, timedelta

#the function which DaoCron looks for in a pre-evaluation script
def pre_evaluate() :

    #tell DaoCron to schedule the task, using the
    #EvaluationResult data-type provided by DaoCronLib
    #EvaluationResult is a class which takes a boolean value
    #indicating whether DaoCron should run the task immediately or not
    res = daocronlib.EvaluationResult( True )

    # set up datetimes where the task should also be scheduled
    sched_times = [
        now + timedelta( minutes=10 ),
        now + timedelta( minutes=20 ) ]

    #set EvaluationResult to tell DaoCron to also
    #schedule the tasks at the given datetimes in sched_times
    res.schedule_multi( sched_times )

    #return the EvaluationResult data-structure to DaoCron
    return res
```

Listing 4 - An example implementation of a pre-evaluation script written in Python. It makes use of the DaoCronLib Python library when producing results for DaoCron

Listing 4 displays an example implementation of a pre-evaluation script written in Python. The script implements the required function “pre_evaluate” which has to produce either a Boolean true/false value or return the DaoCronLib data-type “EvaluationResult”. In this example the script makes use of DaoCronLib and its data-type when producing the result. The script does not really perform any evaluation in this scenario, it only generated the

DaoCronLib data-type and assigns information to it. First it initializes the “EvaluationResult” Python class, which takes a true or false value indicating if DaoCron should run the task immediately or not. The script could already return the result here, but it would make no difference from using the Boolean data-type. Instead the script creates a list of datetime values. The list contains two entries, one which is assigned to the current datetime plus 10 minutes and another which is assigned to the current datetime plus 20 minutes. The function “schedule_multi” is then called on the “EvaluationResult” object along with the list of datetimes. This function tells the “EvaluationResult” class that the task should be scheduled multiple times at the datetimes provided in the list. DaoCron is able to understand this and will schedule the task immediately (as the “EvaluationResult” class was initiated with “True”) and also at 10 and 20 minutes later. This is a simple example, but it shows the control the user can achieve in the scheduling process. The user could almost override the entire scheduling process and perform the scheduling through pre-evaluation scripts and DaoCronLib.

5.6 The analytics model

If DaoCron is configured to schedule and utilize analytics it will schedule the analytics tool at certain intervals. The analytics tool is responsible for building the analytics models. DaoCron comes with a sample implementation of an analytics tool, but the users can also provide their own implementation. However, the produced models need to follow a certain format in order for DaoCron to be able to parse and utilize it. When talking about an analytics model for DaoCron it means a JSON-formatted representation of the different data over the course of 24 hours. Both the task, energy and network models share the same base structure. Each model consists of a map of each hour in the day (0 to 23) with a map of the four quarters of each hour along with the hour total as an object of the given hour-entry. What the different values represents varies slightly between the models. For the energy model each hour-quarter combination represents the average energy for all historical records which were logged sometime within that quarter of the given hour. The entries in the network model represents the number of times the given network has been disconnected during that quarter for the given hour. Likewise, for the task models the hour-quarter entry represents the number of times the task has failed historically during that quarter of the given hour. Listing 5 illustrates parts of a network model for the hour entries 15-17.

```
{
    "15" : {
        "hour-total" : 11,
        "0" : 3,
        "15" : 2,
        "30" : 3,
        "45" : 3
    },
    "16" : {
        "hour-total" : 3,
        "0" : 1,
        "15" : 0,
        "30" : 0,
        "45" : 2
    },
    "17" : {
        "hour-total" : 9,
        "0" : 3,
        "15" : 1,
        "30" : 4,
        "45" : 1
    }
}
```

Listing 5 - An example snippet of parts of a network model. This listing shows the entries 15-17 of an example network model. Each hour has a sub JSON-object containing the four different quarters of the given hour along with an hour total value.

Listing 5 is just 3 out of the 24 hour-entries that make up a single model. The full models include a bit more information, for example the number of database entries the model was built upon and the network endpoint address. However, the main information of any model is represented through these hour objects. Each hour object contains values for the four quarters within that given hour along with the total value of all its quarters combined. If DaoCron were to schedule a task using the "N-ever-M" scheduling method combined with analytics, and it were to be scheduled within these three hours given in Listing 5. It would first look at the hour total value of each entry. Since the network model shows the number of times the network has been down during the given hour-quarter combinations it should aim for the hour with the fewest historical network disconnections. Therefore, the scheduler chooses hour 16. Now it aims for the quarter within this hour that has seen the fewest network disconnections historically. Of course the scheduler will also bring the task and energy models into consideration, but if it were to only use the network model the scheduling would have been similar.

6 Evaluation

6.1 Scheduling

DaoCron was set up with three different tasks, one relying on the Cron-like time-based scheduling method, one using the N-every-M method and one task which will be scheduled based on the result of the Cron-like time-based task (task-based). The Cron-like time-based task, referred to as task 1, is set to be scheduled 10 minutes past every second hour, for instance 00:10, 02:10, 04:10 and so on. Task 1 is also appointed to schedule task 3, the task without any provided schedule method, based on a parser script. The parser script will just check the current datetime and if the hour-value is no equal to 06 it will schedule task 3 to run instantly. Task 2, the N-every-M based task, is set to be scheduled 4 times within every 4 hours. Task 3 which is scheduled as a child-task of task 1 will perform pre-evaluation by running a Python script and returning a result with the help of DaoCronLib. The pre-evaluation script will tell DaoCron that the task should be scheduled instantly if the current hour is dividable by 3. Since task 1 only is scheduled to run every second hour task 3 will be scheduled every sixth hour. If the current hour is equal to 02, it will schedule task 3 instantly while also informing DaoCron to schedule the task at the next 10, 20, 30 and 40 minutes (02:10, 02:20, 02:30, 02:40 and 02:50). The system was set to run for 12 hours on an out of the box Raspberry PI 3 B+. Table 1 holds the information about the different tasks.

	TASK 1	TASK 2	TASK 3
SCHEDULING METHOD	Cron-like time-based scheduling	N-every-M scheduling	None, called based on the result of task 1
SCHEDULE INTERVAL	At 10 minutes past every second hour (00:10, 02:10, 04:10 …)	4 times within every 4 hours	
SCHEDULE CHILD TASK	Will schedule task 3 after every run, unless the current hour is equal to 06	None	None
PRE-EVALUATION	None	None	Task allowed to run instantly if current hour is dividable by 3. If current hour is 02 run instantly and schedule again for the next 10, 20, 30 and 40 minutes

Table 1 - The specified tasks to be scheduled with their given scheduling method, the scheduling interval, if it should schedule a child task based on its output and if it should perform pre-evaluation before running.

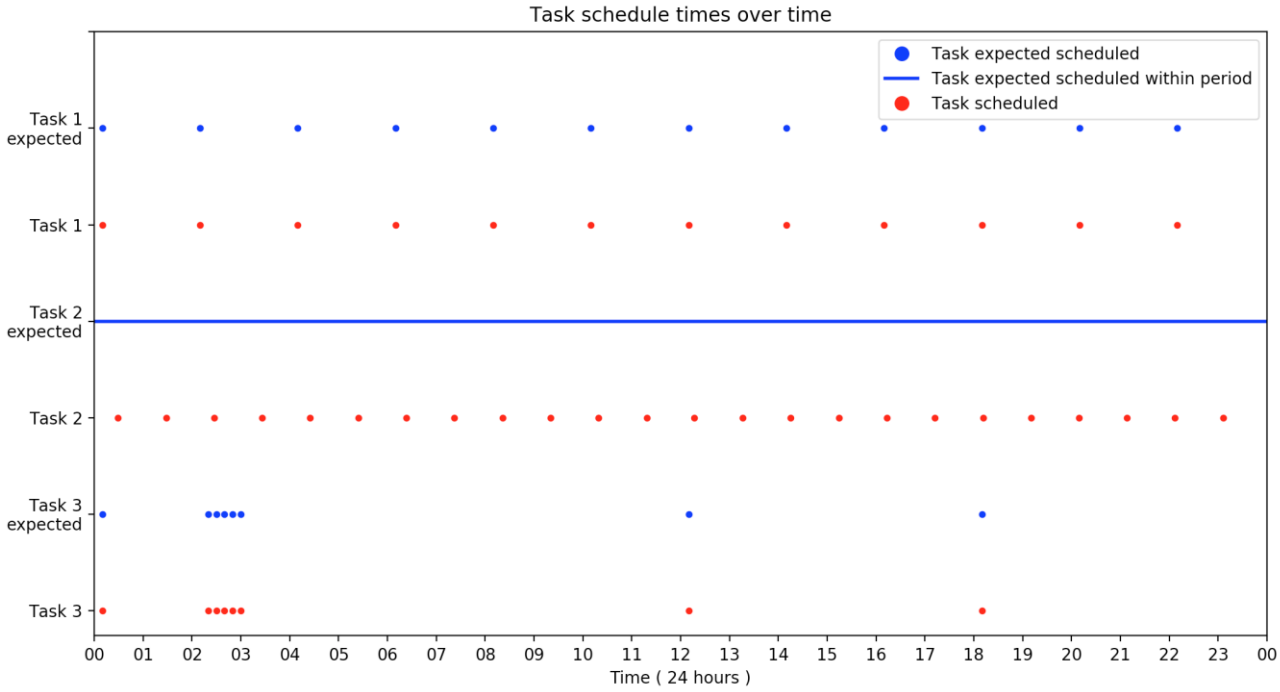


Figure 2 - Visualization of the schedule times for each task over the course of 12 hours. The figure shows the times at which tasks were scheduled (red dots) along with the expected schedule times (blue dots). For Task 2 the expected schedule times is 4 times every 4 hour, and it could be scheduled at any time. This is represented as a blue line, as the task could be scheduled at any point.

Figure 2 displays a visualization of the scheduled times for each task respectively. The graph shows the times at which the tasks are scheduled, marked as a red dot, over the course of 12 hours (from 20:30 until 20:30 the next day). The figure also includes the expected times at when the tasks should be scheduled, represented as a blue dot. Task 2 uses the "N-every-M" scheduling method which means that it can be scheduled at any time as long as it upholds the number of schedules during the given time-period. Therefore, the expected schedule time for the task can be any given time, which is represented as a solid blue line across the figure. Task 1, using the Cron-like scheduling method, is scheduled at 10 minutes past every second hour as intended. Task 2 does schedule 4 times within every 4 hours, as there are no energy and network disconnection samples in the model used the fluctuations in the scheduling times are small. Task 1 is configured with task 3 as a child-task, which makes task 3 a task-based task, and after each execution of task 1 DaoCron will evaluate if task 3 is to be scheduled. When specifying the child task of task 1 it was given a parser script which is to be evaluated before scheduling task 3. The parser script will allow task 3 to be scheduled as long as the current hour is not 6. Task 3 was also given a pre-evaluation script, which tells DaoCron that it can only be scheduled if the current hour is dividable by 3, or if the current hour is 2 it will allow it to schedule while also scheduling it again the next 10, 20, 30 and 40 minutes. Since task 1 only schedules every second hour, and including the results of the parser script and pre-

evaluation script, the only possible hours task 3 can be scheduled at is 0, 2, 3, 12 and 18. According to Figure 2 this was in fact when task 3 was scheduled. It also shows that the task is scheduled at 02:20, 02:30, 02:40 and 02:50 as a result of the pre-evaluation function. The figure shows that the tasks does indeed schedule at the expected tasks, as the red dots representing the schedule time matches the blue dots representing the expected schedule times.

6.2 Evaluating how DaoCron handles unstable networks

In order to validate correct handling of network dependent tasks and unstable networks an experiment was constructed. The goal of the experiment is to let DaoCron run 3 different tasks which are dependent on each of their own network endpoint, and let DaoCron handle the monitoring of the networks, the scheduling of the tasks and the queueing of missed executions due to network disconnection. In a successful experiment DaoCron would let the tasks run when their respective network is available and queue them if it's not. It would run the queued tasks once it detects the network again. DaoCron would also keep correct track of the availability times of the different networks.

The experiment sets DaoCron to schedule 3 different tasks, all using the simple Cron-like time-based scheduling. Each task is dependent on a specific endpoint which is to represent a LAN connection, connection to the Internet, connection to a roaming drone or similar. The experiment was executed by deploying DaoCron to an out of the box Raspberry PI 3B+.

Each network connection is simulated as a locally running server which is brought up and shut down periodically by a script. This script is given a base uptime and downtime to simulate, along with a max-differential variable where a random value between the negative max value and positive max value is added to the uptime and downtime for each state-change. Simulating a random noise in the uptime and downtimes. When the Raspberry PI starts up it will start three instances of this script each representing a server emulating a network connection the tasks are dependent upon. Table 2 shows the different uptime, downtime and differential values for each simulated network.

DaoCron is configured to monitor these networks every 30 seconds. The tasks run a simple script which will perform an HTTP-request towards their given service, returning an exit-code of 0 representing a successfully scheduled task. It will return an exit code with a value other than 0 if it fails, and DaoCron will consider the task as failed.

	BASE UPTIME	BASE DOWNTIME	MAX DIFF
NETWORK 1	20 minutes	10 minutes	6 minutes
NETWORK 2	60 minutes	20 minutes	5 minutes
NETWORK 3	120 minutes	40 minutes	15 minutes

Table 2 - The simulated networks and their base time-values.

	TASK 1	TASK 2	TASK 3
DEPENDENT ON NETWORK	Network 1	Network 2	Network 3
METHOD	HTTP	HTTP	HTTP
SCHEDULES	Minutes: 0, 15, 30, 45 Every hour, every day, every month, every year	Minutes: 10, 40 Every hour, every day, every month, every year	Minutes: 12, 32, 52 Every hour, every day, every month, every year

Table 3 - The specified tasks with their respective network dependencies, detection method and schedule time

Table 3 shows the 3 different tasks and what network they are dependent upon. They all used HTTP-to perform the network request, which is what DaoCron will use to monitor the connectivity. The scheduling row shows at what times the task is set to be schedules. As all tasks uses the Cron-like scheduling method they execute at the exact given times. All three tasks are specified to run every hour, every day, every month and every year. They differ however at what times of the hour they will run. For example, task 2 will run when the minute value is 10 and 40, so at 15:10, 15:40, 16:10, 16:40 and so on.

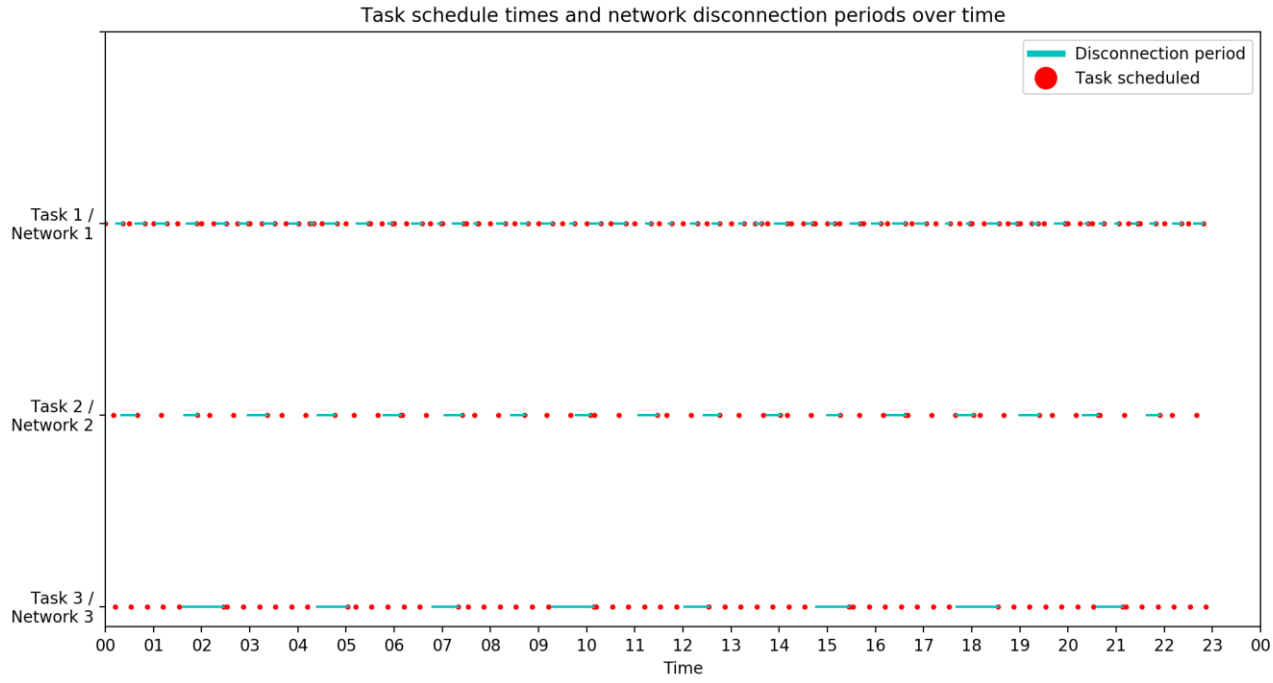


Figure 3 – Visualization of the schedule times, as red dots, along with the network disconnection period for the dependent network as a cyan line

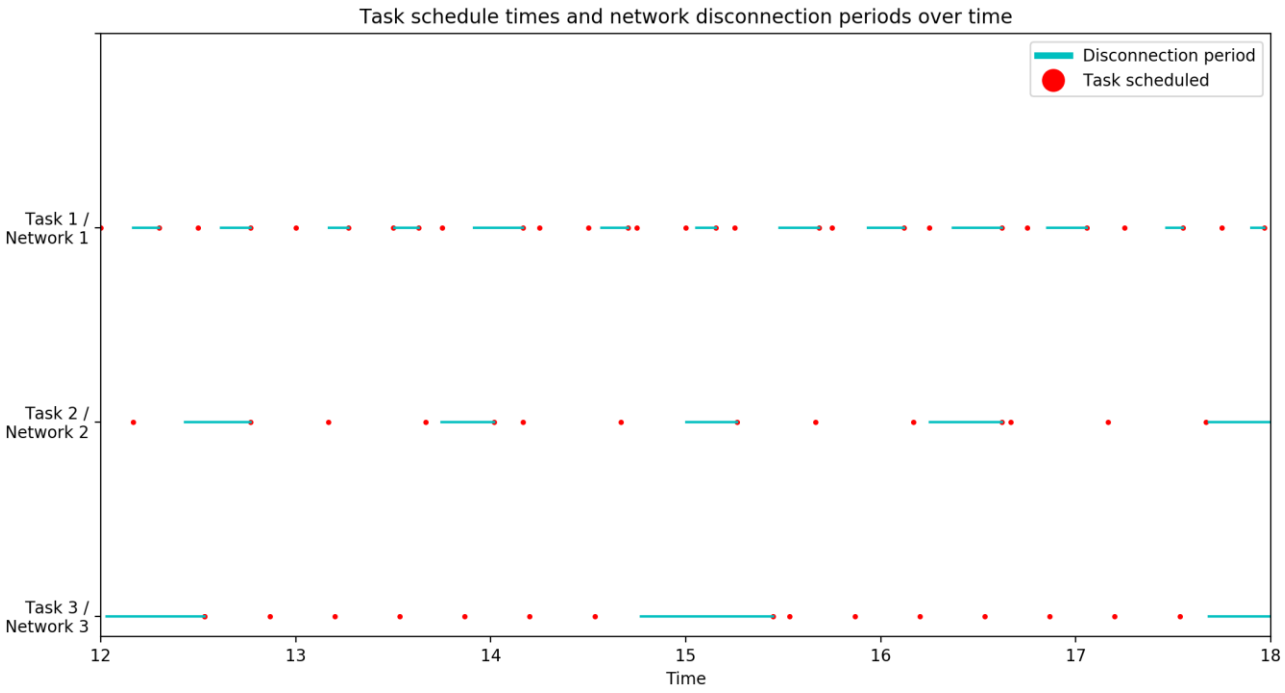


Figure 4 - The same data from Figure 3 but only between 12:00 and 18:00

Figure 3 and 4 shows the visualized output of the experiment. The figures show the schedule times for each task and the disconnection period for the network the task is dependent upon. As the graph shows, the tasks are not scheduled during the downtime of the related networks, which is represented as the cyan lines, but are scheduled when the networks are up. The graph

also shows us the effects of the networking queue, which will keep a task which is meant to be scheduled during a downtime on hold, as one can see that tasks have been scheduled at the exact moment the networks come back up. The cyan lines indicating a network downtime has a red dot, indicating that the task is scheduled, at the very end. There are no tasks scheduled while their respective network is down, which would be represented as the cyan line crossing over a red dot. Tasks are scheduled when networks they don't depend on are down, which is visible at around 15:00 in Figure 4 where task 1 is scheduled during the downtime of network 3. Which is the intended result.

6.3 Time spent calculating schedule times

Three tasks were set up to be scheduled, one task using the Cron-like method, one task using the N-every-M method without including analytics and one task using the N-every-M method along with analytics. Each time DaoCron had to calculate the next datetime for the given task the elapsed time in terms of milliseconds were measured and logged. The experiment was set up to run on the same out of the box Raspberry PI 3 B+ as used in the other evaluation experiments. This lets us compare the expected execution time when scheduling a single task, which for less frequent tasks might just be a drop in the ocean, but for more rapidly scheduled tasks it may be an interesting comparison. Each task was scheduled 100 hundred times each.

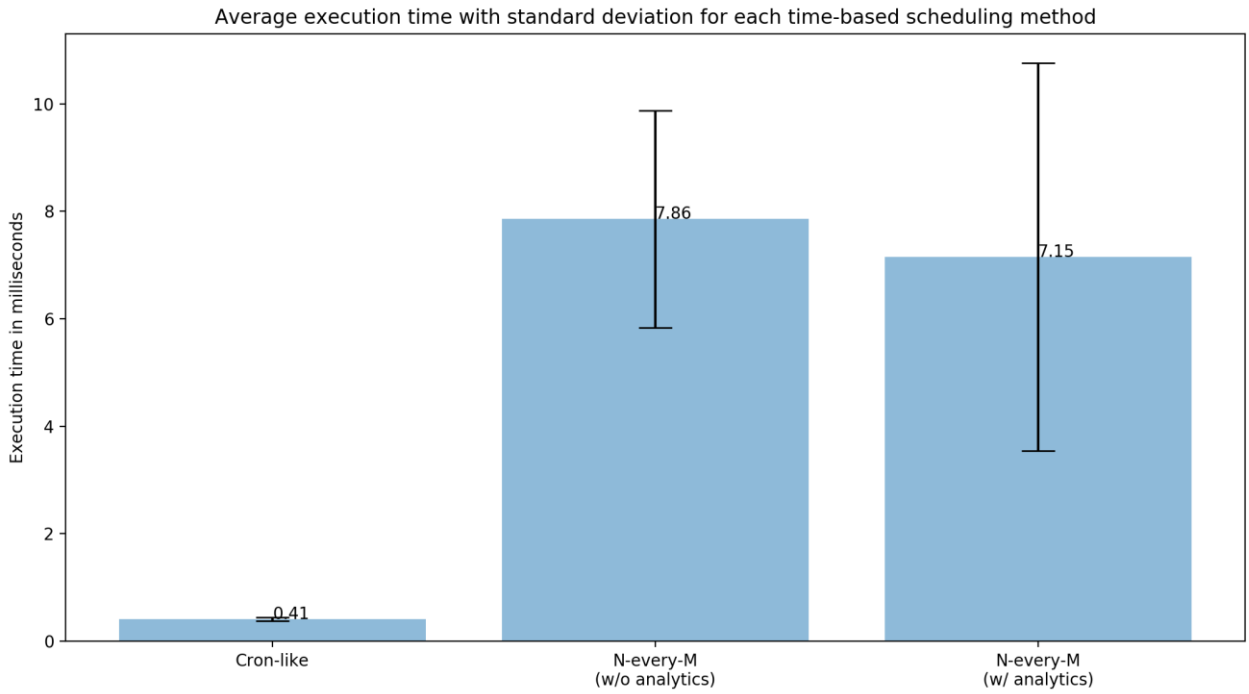


Figure 5 - The average execution time in milliseconds with error bars for the different types of time-based scheduling methods

As figure 5 shows, the time spent scheduling the different tasks are quite insignificant. All three scheduling methods just needs a few milliseconds in order to find the next datetime at which the task should be scheduled. What is interesting is the fact that the N-every-M method which makes use of the analytics created models here has a lower average execution time than its N-every-M equivalent which does not make use of the models. Figure 5 shows that the standard deviation for the N-every-M method using the analytics models is broader than its non-analytics counterpart. This might be explained by the fact that the scheduling period for the tasks were set to smaller time-periods, which leaves the algorithm with fewer decisions to evaluate. The models that are created by the analytics tool which comes with this thesis are created at hour and quarter intervals. Meaning that scheduling an N-every-M based task 30 minutes ahead leaves little room for manoeuvring as there are only 2 different model instances to compare against (one for the first and one for the second quarter of the 30 minutes). Which again may explain the high standard deviation as some time-intervals may overlap more model entries than others. For instance, scheduling a task 20 minutes ahead, starting at 12:10 will include the quarter models for 12:00, 12:15 and 12:30. While scheduling 20 minutes ahead at 12:00 will only include the 12:00 and 12:50 quarter models. Scheduling the same task at 5 times over the period of 10 hours could therefore include way more model evaluations than 5 times over 1 hours.

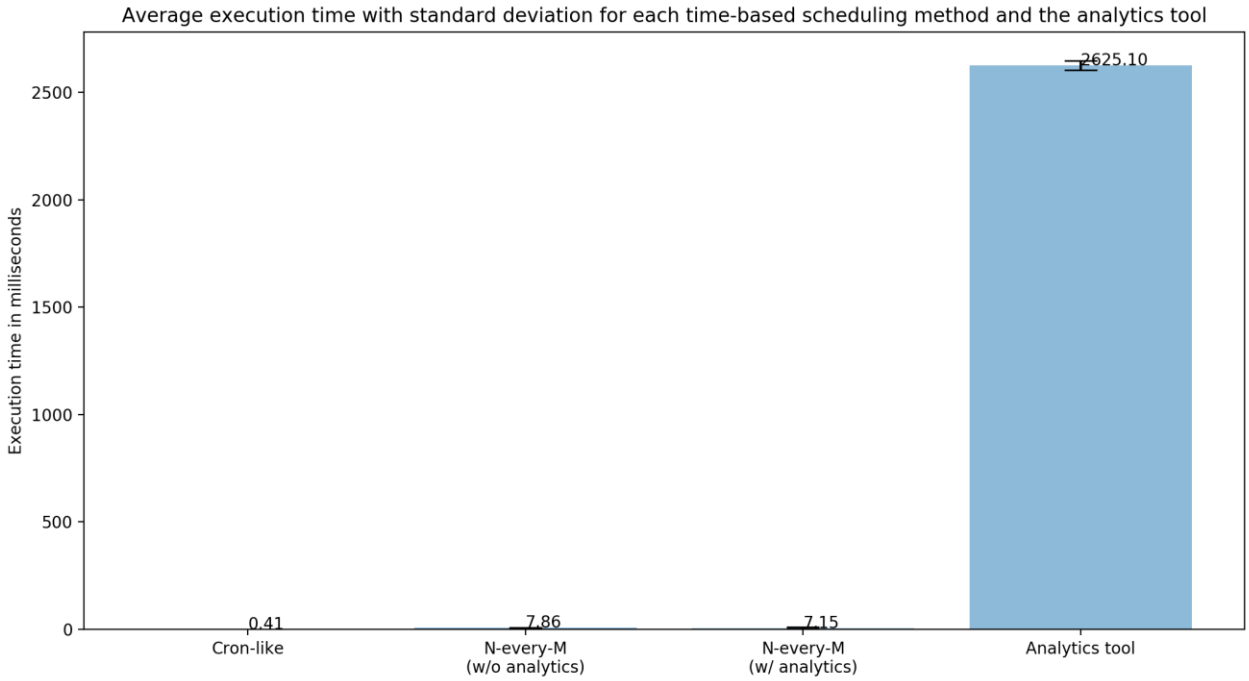


Figure 6 - The average execution time in milliseconds with standard deviations for each time-based scheduling method compared to the execution time of the analytics tool

One reason why it's interesting to evaluate the average execution times is because of the impact the analytics tool would have if DaoCron were to perform "live" analytics. Meaning that it would build a model each time it was scheduled. Figure 6 shows the entries from Figure 5 compared to the average execution time and standard deviation of the analytics tool. The analytics tool was given a database with 100 entries of network disconnection periods, 100 entries of energy level readings and 100 entries of task errors. The average execution time was calculated over the course of 100 runs. As the graph shows, the average time spent building the models is at just over 2600 milliseconds. This is significantly higher than the time-based scheduling methods. Whether to build fresh models each time DaoCron schedules a task, in order to get the most accurate models, compared to letting the analytics tool build the models at far less frequent periods becomes a question of trade-offs between computational power spent and model accuracy. Which may have a significant impact on an OU deployed at the arctic tundra with a limited amount of power.

6.4 Analytics model generation

In order to gain some insight and be able to evaluate the accuracy of the analytics model generation an experiment was set up. This experiment simulates energy levels over a period of time and generates an energy model using the analytics tool. Since the generated models for network, energy and tasks are quite similar, evaluating the energy model only is sufficient

for this experiment. The experiment used a script to simulate energy levels over 24 hours and feed it to the DaoCron database. After the sample energy entries were created the analytics tool was scheduled. The generated model is visualized in a bar-chart.

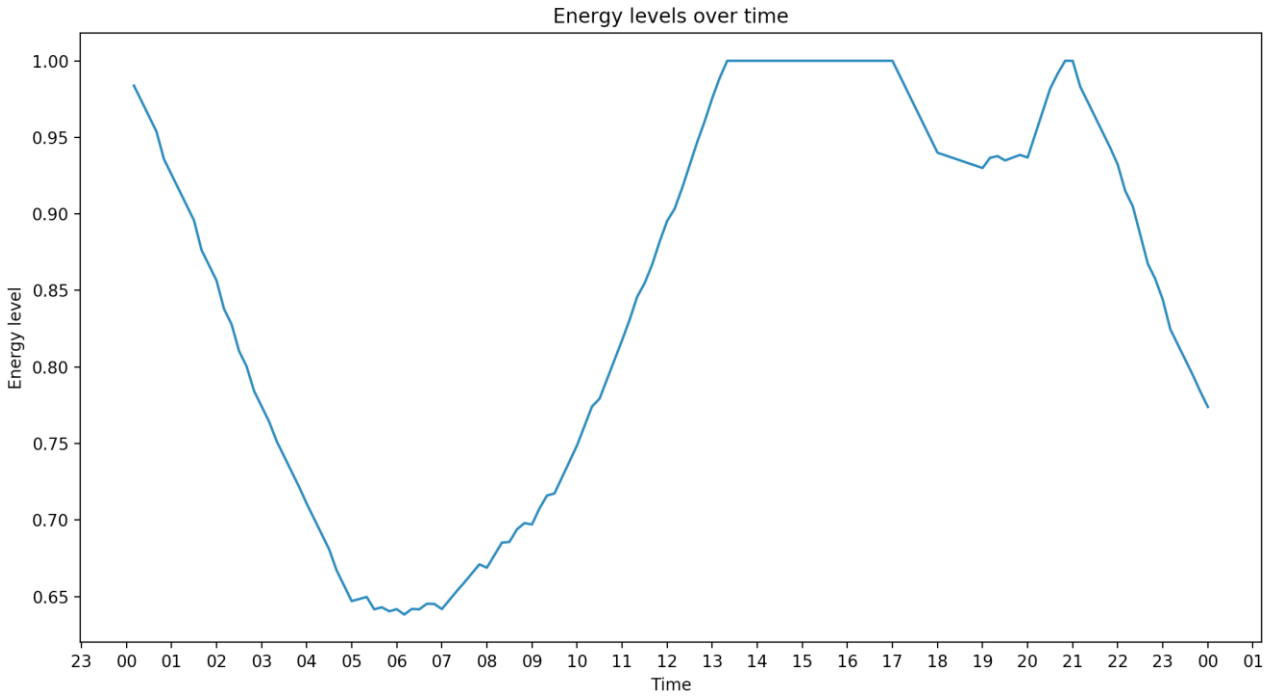


Figure 7 - Visualization of the energy samples generated by the energy simulator. It represents the energy levels of an OU over the course of 24 hours. A value of 1.0 indicated a fully charged battery, while 0.0 indicates a depleted battery

Figure 7 shows a visualization of the energy samples generated by the energy simulator. This is the raw data that was fed into the DaoCron database. The values range from 0.0 to 1.0, indicating a depleted and fully charged battery respectively. The analytics tool uses the energy levels logged in the DaoCron database to build models where the entries represents the hourly average and the quarterly average.

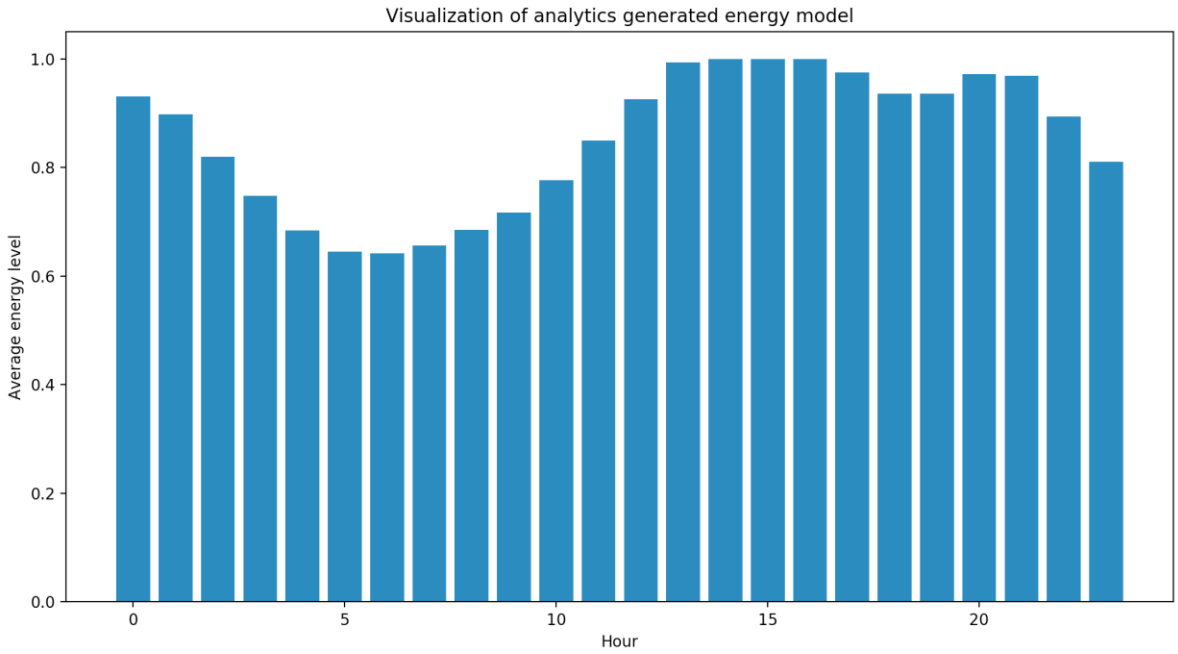


Figure 8 - A visualization of the energy model generated by the analytics tool. It shows each hour of the day with the average energy level for each hour.

Figure 8 is a visualization of the energy model generated by the analytics tool. Each hour entry shows the average energy level for each entry in the database which was logged during the respective hour. As the figure shows, the generated model resembles the raw energy data displayed in Figure 7. In this case the outputted model is expected to heavily resemble the input data, as the input data only covers 24 hours. If the input data were to cover longer periods of time, then the generated model should display the average for the complete input period. By just including 24 hours of sample data its easier to see the accuracy and resemblance in the visualizations. When DaoCron schedules tasks based on the “N-every-M” scheduling method, this would be the model it would base the scheduling times on (along with network and task models). DaoCron would aim for the hours and quarters where the average energy level is the highest. Note that Figure 8 only shows the hourly average energy level. The actual energy model that DaoCron would use also includes the average energy level of every quarter within every hour.

7 Discussion

7.1 Customizability and user-control

One of the main goals of DaoCron is to provide more user control and task customizability than what is provided by Cron. Especially with the deployment environment of the arctic tundra in mind. Which may bring in factors like scarce network connectivity and energy levels. With an emphasis on the “may”, as an OU could be deployed far out on the tundra where the network connectivity is based on whether a satellite is currently orbiting above the unit or at periods where a drone fly’s by. The energy powering the unit may come from a battery, rechargeable or not. At the same time the OU could be deployed near buildings in more populated areas, having access to WIFI at any time of the day and “unlimited” energy from a wall socket. Therefore, the design of DaoCron is heavily based around the “may”-factor. As described in the previous sections of this thesis, most of the different configurations which can be given to DaoCron and its tasks are optional and DaoCron will try to adapt to the environment that the user specifies.

From the scheduling evaluation the result shows that we can control the scheduling of tasks with multiple different configurations and get the expected results. It displays how the use of pre-evaluation scripts, provided by the user, gives the absolute final decision before a task is scheduled to run. It also displays how the user is able to evaluate the output of one task, by providing a custom parser or using DaoCron’s built in parser functionality, to control the scheduling of other tasks. Which allows the user to ultimately build complex task workloads based on the task outcome and state of the system. As a suggested improvement upon the pre-evaluation functionality is perhaps the ability to interfere with task scheduling before the last second before the task is cancelled. This could be solved by performing “maintenance” routine calls to the pre-evaluation script at certain intervals up until the scheduled time is reached. Allowing the user to evaluate the scheduling period continuously. By using the same methods and technology used through DaoCronLib and pre-evaluation script we could even allow the users to provide their own custom scheduling methods or even a complete scheduler.

There are of course certain variables which is just not possible to control. If the OU that runs DaoCron is deployed in such a location where there never is any network connectivity, there is no way for the user to be able to collect information, like through the reporting functionality, at all. Which in turn leaves the user with less control than initially desired. For

an OU which may be deployed for extensive periods of time the user has no idea of whether the tasks were scheduled as planned, if they succeeded or if the unit itself died after just a few hours. The user will first be able to read the reports when the OU is collected, if the OU is even planned to be collected at all. What if DaoCron itself fails? Imagine if an OU was deployed out in the arctic tundra and is intended to perform measurements at certain time intervals through tasks scheduled by DaoCron. The OU is without network connectivity for the whole period, which refrains the user from being able to communicate with the unit until it is finally collected at the end of the period. Currently there is no mechanism which monitors the health of DaoCron itself. Meaning that if DaoCron crashes it will most likely stay down until the unit reboots. The user could of course schedule the unit to reboot at certain time intervals in case DaoCron crashes, but this would at the same time clutter up the planned schedules which resides in DaoCron's virtual memory. DaoCron tries to include this factor to some extent, like checking the task scheduling log from the database when performing N-every-M scheduling for a task. This should reduce the number of missed task due to crashes, reboots or downtime caused by battery depletion. However, this certainly is one of the main problems that should be addressed in any future work.

It therefore becomes important to ensure that DaoCron is a robust system which can handle errors without causing downtime. This requires us to test all the different possibilities thoroughly. Combining these with the many possible configurations which can be given to DaoCron quickly becomes a time-consuming task, which also has been reflected in the DaoCron design as it aims for high user customizability but still try to limit it to a sane level.

7.2 The Cron-like time-based scheduling method versus the N-every-M time-based method

The Cron-like time-based scheduling method is supposed to imitate the scheduling functionality of Cron. Allowing the users to stick to regular Cron functionality if they desire. As DaoCron aims to extend the Cron functionality rather than replace it, it seemed natural to include the base functionality of Cron. Though it should be noted that network and energy dependent tasks using this method will still be affected by low energy levels and missing network connectivity. Which is not really a problem as a task dependent on a network would indeed fail anyways if it was scheduled to run without network connectivity.

The N-every-M based scheduling method is designed for longer periods of time, as is reflected by how it uses the analytics generated models to move around the time at which a

task should be scheduled. For tasks which is intended to be scheduled at faster intervals, say down at minute or hour levels, there is not much to gain in using this scheduling method over the Cron-like method as the time-frame at which the scheduling time could be moved around is far tighter than for less frequent scheduling intervals. But as we see from the evaluation of the average execution time for the different scheduling method the extra computational time is insignificant. Therefore, the Cron-like method is far from superior at higher intervals, but hopefully the N-every-M method is superior at less frequent intervals. As it allows for more dynamic movement of the scheduling time. As the results displayed in Figure 2 shows, the N-every-M based tasks are scheduled without a lot of differences in terms of minutes for each scheduling. Which supports the point of the limited gains in using the N-every-M based method at such short intervals. DaoCron would want to evaluate these intervals at way higher time intervals, like weeks to months (or even years), in order to get a better understanding of how this method behaves. But the limited time-frame of this thesis makes it hard to perform any real organic evaluations in this field. Since its also heavily reliant on the analytics generated models, the analytics tool needs a sufficient number of historically collected data to build accurate models, for then to evaluate the process over a longer time span.

From the evaluations performed the results do indeed show that the different scheduling methods produces the expected result. As Figure 2 shows, the Cron-like based task does indeed schedule the tasks to run at 10 minutes past every second hour as intended. The N-every-M based task does also schedule the task at the expected times, which is 4 times within every 4 hours. One can also see the small fluctuations in the scheduling time for the N-every-M based task, though a bit hard to spot in the figure. It was however not expected any large fluctuations in this experiment as the time-frame was rather narrow.

7.3 The power of chaining tasks

DaoCron allows the user to create tasks which will evaluate their output and schedule other tasks accordingly. These tasks are referred to as “task-based” tasks. The user can provide a command-line command to such task which are to be executed by DaoCron if the evaluation holds. The user can also provide the identifier of another task which is specified in one of the task-files provided to DaoCron. This allows DaoCron to schedule a new task with the same amount of configuration as any other task from the result of the “parent”-task. Meaning that a task specified to evaluate its output and schedule another task could be set to reference a task with its own result evaluation and child-task scheduling. This allows the user to build a chain of task based on the output of the different steps in the task-chain. Ultimately this allows the

user to build advanced workloads with high customizability between the different steps in the task-chain. Combining such tasks with pre-evaluations and the use of DaoCronLib further enables the user to control the workload. The motivation behind this functionality is to provide a new level of abstraction for the user. By combining smaller tasks through such a chain hopefully allows the user to build more complex pipelines at a higher abstraction level for less effort. These tasks could also be re-used by other tasks again and the different task-chains could easily be altered by changing the different task references. However, the user still has to define and configure these tasks. Though the task-files are designed to be easy to configure, it quickly becomes tedious to specify all the different configurations through JSON. A proposed solution to this is to create a simple tool, like a command-line tool, which helps define, parse and structure tasks. The tool becomes responsible for keeping references and helps visualize the task-chains and their dependencies. This would hopefully allow for simpler task creation through a more reader friendly interface. It could also allow for users with less experience to simply create tasks. This could again be built upon by using some functionality similar to the one used through the reporting tool. DaoCron could periodically look for new updates to its tasks and configuration by querying a RESTful-API for changes between its current configuration and new ones. A server with a web-interface for designing tasks could be set-up and serve updates to the different OUs.

7.4 “Live” model building versus semi-cold analytics tool generated models

As described earlier on in the thesis, DaoCron can combine the “N-every-M” based scheduling method with the models generated by the analytics in order to try to avoid scheduling a task during periods of historically higher chances of network disconnections, low energy or task failures. The analytics models used by DaoCron is generated by the analytics tool at certain intervals of time by DaoCron itself. It will pull the historical data from the database and built network, energy and task failure models upon it.

Since the models used by DaoCron is only generated by the analytic tool at certain intervals, the models may not be completely up to date with the current situation. For large amounts of collected data its not really any big deal if the models do no include the entries from the last 24 hours. Especially considering that networks probably does not disconnect that often over the course of 24 hours, along with the fact that most tasks are not scheduled too often either. Tasks which are scheduled frequently does not have much to gain from using the analytics models, as the time-period the planned schedule time can manoeuvre within is relatively

short. However, if DaoCron nonetheless were to aim for the freshest possible models every time it was to plan the scheduling of a task it would have to build the models when scheduling the task.

Building the models each time DaoCron is to schedule a task would of course add some overhead. Instead of only running the analytics tool every now and then it must build the models during each task scheduling. The overhead of the analytics tool would be dependent on the amount of historical data collected and how well implemented the analytics tool itself is. As Figure 6 shows, the execution time of the analytics tool is significantly higher than the different scheduling methods. The execution time in Figure 6 is based on about 100 entries of task errors, network disconnections and energy levels. Resulting in a total of 300 entries to build the model upon. It quickly becomes relatively expensive to build the analytics models for each time DaoCron should plan the scheduling of a task. On the other hand, even though the analytics tool uses longer time to complete than the scheduling methods it does not really use that much of a time. If a task only is to be scheduled one every two hours, then DaoCron would only have to run the analytics tool for that given task one every two hours. Then suddenly two seconds becomes insignificant. Its even more insignificant for task that schedule at even less frequent periods. Again, the penalty of building the analytics models every time DaoCron plans a task schedule depends on the number of tasks, the amount of historical data and the frequency of the task scheduling. For large amounts of historical data, it the effects of the models not being completely up to date may be insignificant or even non-existing.

7.5 The problem with collecting analytics data and performing predictions

As DaoCron uses analytics in order to improve its scheduling and reduce the failure rates it requires good and enough historical data to base its models on. DaoCron builds models for the individual tasks, different networks and the energy usage. Since DaoCron is designed to run on OUs which may be deployed in a large variety of locations and environments there is not really any pre-built models that can provide sufficient information for all these specific targets. It is therefore dependent on collecting data and building its models during deployment. The problem is however that for specific tasks and targets the time to build up enough data for it to be representable can be painfully long. Consider a task which is set to run once every month, after a year it will only have collected 12 data points to base the models and predictions on. Likewise, a network which is only disconnected a few hours every month will have the same problem. Even if it were to collect a decent amount of data after a

few years, the technology and the surrounding environment may have changed drastically making the older data less representative of the current day situations.

DaoCron was initially designed to perform predictions based on its collected data which would help it schedule tasks and predict network downtimes and energy levels. However, it quickly became obvious that for most use-cases this did not provide a positive contribution but rather just noise. For most use cases a simple model representing the quarterly rates at which events occurred provided a relatively better solution, in terms of avoiding network downtimes, low energy levels and task error, compared to the problems caused by more advanced solutions. As DaoCron's focus is on customizability and handling of resources when they disappear rather than predicting when they disappear, the time-window for this thesis set a limit on what was possible to achieve and what had to be prioritized. The current models do however provide a fairly relevant representation of the current deployment environment of an OU.

A proposed solution to this is to build distributed models to help OUs collectively build and improve their local models and predictions. This would of course mean that OUs reliant on this solution would require network connectivity at periods. Making the solution obsolete for OUs with no network access at all. The distributed models may be fed to the OUs before deployment, but once deployed it will have no room for improving these models through the distributed solution. The vast number of different environments an observation unit could be deployed at would require us to do some form of classification of the different data. Does the OU have an unlimited supply of energy? How is the network connectivity for the given OU? What hardware is it running? There are a lot of different factors which needs to be included into the equation in order to properly match deployments with data models. Such a solution quickly becomes complicated and the data models should match the characteristics of the OUs closely in order to properly benefit from them.

Decision trees are one of the most commonly used models when performing supervised-learning. This method has been studied for a long time now and is often used to identify entities and classify them [9, 10]. For our problem the system could make use of a decision tree in order to classify the OU and map it to relevant training data. By allocating variables like network dependency, battery usage, task identifiers, deployment area and similar to the distributed models the system could use the decision tree when trying to find what data is

most relevant for the deployed OU. Hopefully, this could help us gather and assign historical data collected by the OUs and distribute them between themselves.

7.6 Running tasks in the cloud

There are problems with running computational heavy tasks, tasks which are CPU and memory intensive, over longer time-periods. Especially for OUs dependent on power from a limited power source like a battery. The battery could deplete during the task execution wasting the progress and energy in terms of using energy and time on a task that resulted in nothing. The hardware of the OU may be insufficient or obsolete for the task that the OU wants to run and the task output may eat up all the storage space on the OU. There are several factors which could make the execution of such a task on an OU insufficient. Therefore, it could be interesting to investigate outsourcing the execution of the task to the cloud.

As a proposed solution to this problem, a task execution service could be set-up to run tasks passed to it by the OUs. The system could make use of a container service, like Docker [11], to run the desired tasks on virtual machines representing an OU. There are several ways this could be accomplished. By settings up a basic UNIX-based image with Python pre-installed on the Docker service the OU could pass its tasks along with the required resources to the execution server. The OU would also send a requirements file for the different Python libraries the container should install in order to be able to run the Python scripts. The tasks that were passed from the OU to the Docker service would be limited to Python based tasks with this solution. Similarly, the OUs could be allowed to send a Dockerfile, which Docker uses to build new containers, to the service. Letting Docker build the container from scratch with the exact configuration required. Alternatively, if multiple OUs are expected to schedule the same task and the only variating factor is the input fed to this task, the container could be pre-built with the requirements and then the OU only need to send its input to the service.

There are several factors which comes into play in order to make this solution sufficient. The OU must of course have access to network during periods of time in order to be able to send the data required. Sending data over the network is an energy costly task, and the trade-off in terms of energy saved should be significant enough. DaoCron should therefore be able to predict the energy cost of sending the data to the task execution service versus running the task locally and outsource the task to the cloud if it believes that the energy savings is worth the hassle. This is not a new problem, and it exists studies and solutions with proposed algorithms which have proven this to be doable [12, 13].

The OU might also need to collect the data if it is to be used by the OU afterwards. If the data is of no interest to the OU itself then this is not really a problem. However, if the OU does indeed need the data it will need to know when and how it can collect it. When the OU sends its data to the task execution service, the service could respond with an address at where the results will be available. Letting the OU ping this address at certain intervals until it is completed. However, if the task is dependent on retrieving the results as soon as it is available there may be several other factors which should be considered before outsourcing the task. Imagine that the OU sends its task to the task execution service which spends one hour completing the task. During this hour the OU lost its network connection and did not become available again until hours later. For the OU this could turn out to be a far less superior solution that just running the task itself locally.

8 Conclusion

This thesis presented DaoCron, a service for scheduling tasks on Observation Units deployed at locations with unstable network and energy. The goal of DaoCron is to provide the scheduling functionalities of Cron while allowing for more user customizability and control. DaoCron tries to reduce the number of failed tasks due to network disconnection by putting tasks dependent on network connectivity on hold and rescheduling them as soon as the networks become available again. The same goes for tasks which has minimum energy requirements. DaoCron will put these on hold and reschedule them at later times when the requirements are met. By building analytics models from the historical data, it collects it tries to improve the scheduling of tasks in order to avoid network disconnection, low energy levels and periods of high rates of task failure.

The evaluations performed shows that DaoCron is able to schedule the tasks as desired, while allowing the user to highly configure the scheduling process. We see that DaoCron is able to monitor and adapt to networks disconnecting and reconnecting by evicting the scheduling of network dependent tasks during periods of network disconnection and rescheduling them when the networks become available again. The user is able to highly customize and control the scheduling process through various different scheduling methods, providing custom written pre-evaluation scripts and through reporting functionalities.

Through the process of trying to improve the scheduling method by using analytics and predictions it quickly became obvious that the amount of data required to be able to generate meaningful predictions is way more than what is usually collected trough a single OU. The analytics models ended up only representing simple rates at hour and quarter levels for a single 24-hour period. The thesis covers, discusses and propose solutions for problems which may occur when DaoCron is deployed at different locations. Like scarce sample data for analytic models and energy related problems for computational heavy tasks. These are topic that would be interesting to look further into in any future work.

9 Future Work

Through DaoCron does achieve both increased customizability and user control there are many paths one could follow for future work. Of course, one could continue to increase the customizability of DaoCron and its task into the infinite. However, one has to ask oneself how much DaoCron and its tasks should be able to be configured, and when it just becomes excessive.

This thesis mentioned how performing predictions along with building the different models could help increase the failure rates of scheduled tasks during the discussion section. As brought up in the discussion section, performing predictions with valuable outputs can be hard to achieve using only the historical data collected by DaoCron as it may take a long time for DaoCron to collect enough data to build proper predictions upon. Therefore, it would be interesting to look further into this issue and possible solutions.

As DaoCron is mainly designed for OUs located at the arctic tundra where the energy levels may be limited, DaoCron should be optimized to be as energy efficient as possible. This is something which has not been focused heavily on in this thesis and could be an interesting path to follow from here. DaoCron currently only cuts certain tasks from being scheduled if the user has configured the tasks to have minimum energy level requirements in order to run. The implementation of DaoCron itself and its associated components could possibly be optimized for energy consumption by a fair share.

As discussed in the thesis it would be interesting to try and outsource the executions of tasks to the cloud. If DaoCron were able to estimate the energy cost of outsourcing the task to the cloud versus running it locally, it could perhaps save a decent amount of energy.

References

[1] R. A. Ims, J. U. Jepsen, A. Stien, and N. G. Yoccoz, “Science plan for coat: climate-ecological observatory for arctic tundra”, Fram Centre report series, vol. 1, pp. 1–177, 2013

[2] M. S. Keller, “Take Command: cron: Job scheduler”, Linux Journal, vol. 1999, issue 65es, Sept. 1999

[3] https://help.ubuntu.com/community/CronHowto , visited May 2nd, 2019

[4] https://wiki.gentoo.org/wiki/Cron#vixie-cron , visited May 24th, 2019

[5] I. Raïs, J. M Bjørndalen, P. H. Ha, K. A. Jensen, L. S. Michalik, H. Mjøen, Ø. Tveito and O. Anshus, «UAVs as a leverage to provide energy and network for Cyber-Physical Observation Units on the Arctic Tundra”, 2019

[6] https://www.python.org/ , visited May 2nd, 2019

[7] T. Hänisch, “A Case Study on Using Functional Programming for Internet of Things Applications”, Athens Journal of Technology & Engineering, 3., pages 29-38, 2016

[8] https://www.sqlite.org , visited May 2nd , 2019

[9] V. T Chakaravarthy, V. Pandit, S. Roy, P. Awasthi, M Mohania, “Decision Trees for Entity Identification: Approximation Algorithms and Hardness Results”, PODS’07 Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 53-62, 2007

[10] O. Irsoy, E. Alpaydin, “Distributed Decision Trees”, 2014

[11] https://www.docker.com , visited May 15th, 2019

[12] D. Yao, C. Yu, H. Jin, J. Zhou, “Energy efficient Task scheduling in Mobile Cloud Computing”, 10th International Conference on Network and Parallel Computing (NPC), Springer, Lecture Nots in Computer Science, pp. 344-355, 2013

[13] T. Mathew, K. C. Sekaran, J. Jose, «Study and Analysis of Various Task Scheduling Algorithms in the Cloud Computing Environment», 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2014