BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ASYNCHRONOUS TASK PROCESSING IN THE PCS PROJECT

ASYNCHRONNÍ ZPRACOVÁNÍ ÚLOH V PROJEKTU PCS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAL POSPÍŠIL**
AUTOR PRÁCE

SUPERVISOR **doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Pospíšil Michal**
Programme: Information Technology
Title: **Asynchronous Task Processing in PCS Project**
Category: Parallel and Distributed Computing

Assignment:

1. Familiarize yourself with Python and its libraries for asynchronous processing (e.g.: asyncio, multiprocessing, tornado).
2. Familiarize yourself with the PCS project and requirements for the tasks that it facilitates in the cluster.
3. Design an interface for creating and managing tasks in an asynchronous environment. This interface should enable running more PCS commands at the same time and provide information about their status and results.
4. Implement the proposed interface in a form of a REST API (tornado library can be used).
5. Create a set of unit and integration tests for the implemented interface.
6. Demonstrate on at least two simultaneously running PCS commands that the asynchronous processing works.

Recommended literature:

- Project PCS: https://github.com/ClusterLabs/pcs
- Tornado framework: https://www.tornadoweb.org/
- Python Asyncio library: https://docs.python.org/3.6/library/asyncio.html
- Python Multiprocessing library: https://docs.python.org/3.6/library/multiprocessing.html

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

The PCS project is a distributed application; therefore, many actions need a way to launch actions in remote application instances. The goal of this thesis is to implement a minimum viable solution for executing actions through a REST API that uses the asynchronous programming model. However, actions themselves are not implemented asynchronously and cannot be invoked directly from asynchronous code. The REST API is connected to an asynchronous scheduler that circumvents this limitation by launching actions in a process pool. The scheduler hides actions behind an abstraction layer of tasks that store information about their status and results. All the actions need to send real-time updates to the clients. This is made possible via a one-way communication channel from the actions to the scheduler that updates the tasks. The REST API provides methods for creating, getting results, and killing tasks. Clients can periodically check the task status and show these updates to the user. Clients can also choose to kill tasks that take too long to finish.

Abstrakt

Projekt PCS je distribuovaná aplikácia. Z toho vyplýva, že potrebuje spôsob ako spúšťať akcie vo vzdialených inštanciách PCS. Cieľom tejto práce je vyvinúť minimálne životaschopné riešenie pre spúšťanie akcií cez REST API, ktoré je implementované metódami asynchrónneho programovania. Tieto akcie však nie sú implementované asynchrónne, takže nemôžu byť spustené priamo z asynchrónneho kódu. REST API je preto napojené na asynchrónny plánovač, ktorý obchádza toto obmedzenie spúšťaním akcií v sade procesov (process pool). Plánovač skrýva akcie za abstrakčnú vrstvu úloh, ktoré uchovávajú informácie o stave a výsledkoch akcií. Všetky akcie potrebujú posielať aktualizácie svojho stavu klientom v reálnom čase. Toto je dosiahnuté jednosmerným komunikačným kanálom medzi akciami a plánovačom, ktorý správy od akcií ukladá do úloh. REST API umožňuje vytváranie, kontrolu stavu a rušenie spracovania úloh. Klient teda môže opakovane žiadať o stav úlohy a takto zobrazovať aktualizácie stavu z akcií. Klient tiež môže zrušiť spracovanie úloh, ktoré bežia príliš dlho.

Keywords

cluster, high-availability cluster, PCS, Pacemaker/Corosync Configuration System, asynchronous programming, AsyncIO, REST, REST API, Tornado

Kľúčové slová

klaster, klaster s vysokou dostupnosťou, PCS, Pacemaker/Corosync Configuration System, asynchrónne programovanie, AsyncIO, REST, REST API, Tornado

Reference

POSPÍŠIL, Michal. *Asynchronous Task Processing in the PCS Project.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšírený abstrakt

PCS je distribuovaná aplikácia na správu klastrov s vysokou dostupnosťou. Z tohto dôvodu potrebuje spúšťať akcie vo svojich vzdialených inštanciách. Pre tieto potreby využíva démona v úlohe servera poskytujúceho REST API na spúšťanie týchto akcií.

Táto práca sa zameriava na nahradenie existujúceho REST API, ktoré umožňuje len synchrónne spúšťanie akcií. Keďže nie je vopred známe ako dlho akcie trvajú, toto API často prekračuje časový limit na doručenie odpovede. Nové REST API sa sústreďuje na asynchrónne spracovanie úloh. Úlohy sú abstrakčnou vrstvou pre akcie zapúzdrujúce informácie o priebehu a výsledkoch akcií. Cieľom tejto práce je navrhnúť a implementovať minimálne životaschopný systém na spúšťanie akcií, ktoré nepoužívajú metódy asynchrónneho programovania, z prostredia asynchrónne implementovaného REST API.

Nové REST API poskytuje rozhranie pre vytváranie, zisťovanie stavu a rušenie prebiehajúcich úloh. Tým eliminuje problémy s vypršaním času na doručenie výsledku akcie, keďže stav úlohy môže byť poskytnutý okamžite. Tiež zaručuje, že strata výsledku pri prenose sieťou sa nebude prejavovať rovnako ako nedosiahnuteľnosť servera. Vedľajším efektom je zjednotenie rozhrania pre klientov implementujúcich užívateľské rozhrania a samotného PCS, ktoré potrebuje spúšťať akcie vo svojich vzdialených inštanciách.

Na pozadí REST API je implementovaný asynchrónny plánovač úloh, ktorý sa stará o spúšťanie akcií v oddelených procesoch a správu úloh. Kombinuje tak časti softvéru používajúce asynchrónny model programovania s časťami, ktoré ho nepodporujú. Plánovač navyše obsahuje podporu pre spracovanie správ generovaných počas behu akcií z oddelených procesov. Na posielanie správ z oddeleného procesu je použitý zdieľaný front bezpečný pre použitie viacerými procesmi naraz. Správy generované akciami môžu takto byť doručené klientovi ešte počas behu akcie. Toto je odlišné od existujúcich riešení, ktoré umožňujú len prenos samotného výsledku.

Sada jednotkových a integračných automatizovaných testov dokazuje funkčnosť implementovaných rozhraní. Minimálna implementácia tenkého klienta je tiež súčasťou riešenia a testovanie s jej pomocou ukázalo, že riešenie je funkčné v reálnom svete. Je nutné poznamenať, že ide o *minimálne* riešenie, ktoré musí byť rozšírené hlavne v oblasti bezpečnosti.

Asynchronous Task Processing in the PCS Project

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Adam Rogalewicz. The supplementary information was provided by Mr. Ondrej Mulár. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Michal Pospíšil
May 17, 2022

Acknowledgements

My biggest thanks goes to Mr. Ondrej Mulár from Red Hat Czech, s.r.o. who suggested this project to me. I am sure that I would *not* be able to complete the project without his supervision and guidance. And of course, this extends to the whole PCS development team that I could always turn to with any question. I have learned so much from our interactions over the course of writing this thesis.

A close second on this list is my thesis supervisor at Brno University of Technology, Mr. Adam Rogalewicz. He patiently answered every question that I had and always dispensed some good advice when I needed it the most.

Of course, without the support of my family, I would not have even made it to this stage. I want to thank my mom, dad and sister for their love, care and continuous encouragement.

Another big thanks goes to my closest friends for their moral support. A special thanks goes to Denton Wood for trying to keep my English in check and teaching me a thing or two about writing research papers. I will do better next time, hopefully!

Contents

Chapter 1

Introduction

The PCS[1] project is a configuration tool for Pacemaker[2] high availability cluster stack. It consists of a command-line utility and a daemon that provides a web interface for cluster management. Originally, PCS was only meant to be a command-line utility. A common theme in software engineering is that software requirements cannot be precisely defined at the beginning of the development process. During development and as users get their hands on the software, new demands start to flood in. Any architectural decisions made based on the old requirements may very well become limitations to further improvements.

As time passed, the original architecture of the PCS project started to cause more and more headaches both for users and developers. To counteract this trend, the PCS development team had to come up with a new architecture and a plan of how to transition a fairly large project to it. This effort has been ongoing for a few years. This thesis aims to fit into this plan with a fundamental change to the core of the PCS project.

In very simple terms, any action done by PCS is going to be processed in a very distant future by the PCS daemon – a process running in the background. This process needs to be able to handle many requests at once, and it uses one of the asynchronous programming models to achieve this. The challenge is that this model is incompatible with the non-asynchronous actions already implemented in PCS.

This work is structured into three main chapters. The first chapter further explains the concepts of high-availability clustering, introduces the PCS project and its problem areas. The second chapter explains the concepts that this thesis deals with and are crucial for making the right design decisions backed by the analysis of problem areas. In the third chapter, the implementation and testing of the previously designed solution is explained.

[1] PCS – Pacemaker/Corosync Configuration System, more at https://github.com/ClusterLabs/pcs
[2] Pacemaker, more at https://github.com/ClusterLabs/pacemaker

Chapter 2

About the PCS Project

The PCS project[1] is the main area of interest in this thesis. Understanding what it does and how it falls into the bigger picture is going to be crucial in understanding the architectural choices described in the next chapters. But first, it is important to learn what the bigger picture is – it is high availability clusters.

2.1 About High Availability Clusters

If someone runs a website on a bare metal server, it would generally involve maintaining at least a web server and a database. Even though this is a very simple scenario, a failure will eventually occur, and that website will become unavailable. All software has flaws and hardware is bound by the laws of physics, so it cannot run without failure. To fix this failure, manual intervention from an administrator is needed. But first, the administrator needs to detect that there is a problem. Then more time passes as the fix is being applied. In addition, any process that involves human factors adds uncertainty as to whether the fix is going to address the issue. All while the website is unavailable. Although this might not be a big problem for a personal website, there are certain applications that require high availability. This could include air traffic control systems [15] that could cause loss of life if they are not available. Another example would be enterprise applications where their unavailability prevents the business from operating and causes a monetary loss.

High availability is about minimizing the time when a service is unavailable – this period of time is called **downtime**. In this example, the web server and the database are obvious **points of failure**. To find others, the system needs to be looked at as a whole unit. The server (as a machine) runs because of electricity, and a power outage or power surge can bring it down. Going one level lower, the server is made up of components that can each fail, and the list goes on. High availability cluster is a solution to minimize downtime because it automates problem detection and corrective actions.

Since the high availability part is now clear, it needs to be established what a cluster is. Cluster is a group of computers and other resources connected through a network that acts as a single system from an outside perspective. Computers that make up the cluster are

[1] PCS – Pacemaker/Corosync Configuration System, more at https://github.com/ClusterLabs/pcs

called **nodes**. All nodes typically perform the same task, but this depends on the type of cluster. Depending on the purpose, a cluster solution can be one or a blend of these:

- **Storage clusters** operate on a single shared file system, eliminating data redundancy, increasing read and write speeds, simplifying backup and administration of multiple machines.
- **Load-balancing clusters** distribute network service requests across cluster nodes to balance resource use and make scaling effective.
- **High-performance clusters** combine the computational power of the nodes to perform a highly demanding task in parallel to speed it up.
- **High-availability clusters** (sometimes called failover clusters) provide services with minimal downtime by eliminating points of failure. [13]

How is high availability achieved in reality? It was already mentioned what is classified as a point of failure. High-availability clusters eliminate them by redundancy. If something happens to one node, the cluster moves the services to a different node – this is called **failover**. It is essential to note here that a cluster is still just software and *will not fix the root cause of the problem.* Administrators still need to examine why the original node was deemed unsafe and fix the underlying problem. Failover is just a corrective action taken to maintain availability and give administrators more time to find a solution. [24]

2.2 What is PCS

The name PCS is an initialism for **Pacemaker/Corosync Configuration System**. Pacemaker and Corosync are the two main components of the Pacemaker cluster stack developed by the ClusterLabs community. The Pacemaker cluster stack is a collection of open-source projects that provide HA[2] cluster functionality on many Linux distributions. Despite its name, PCS is a high-level configuration utility for Pacemaker[3], Corosync[4], QDevice[5], SBD[6] and Booth[7]. In the context of this thesis, it is not that important to understand the inner workings of the Pacemaker cluster stack. However, if the readers are interested, Clusters from Scratch[8] is a very good introduction. [2]

The PCS project was created to streamline the setup and management of high-availability clusters based on the Pacemaker cluster stack. Using PCS, administrators can manage their HA cluster from a command-line interface or a web interface with a smaller subset of features. PCS hides the minute details and simplifies the most common operations, such as cluster setup, adding resources, constraints, etc. Since each of the cluster's components is a separate project, it is configured and used differently. PCS is one place to view, edit,

[2] HA – high availability
[3] Pacemaker, more at https://github.com/ClusterLabs/pacemaker
[4] Corosync Cluster Engine, more at https://github.com/corosync/corosync
[5] Corosync QDevice Daemon, more at https://github.com/corosync/corosync-qdevice
[6] Shared-storage based death, more at https://github.com/ClusterLabs/sbd
[7] The Booth cluster ticket manager, more at https://github.com/ClusterLabs/booth
[8] Clusters From Scratch – available online at https://www.clusterlabs.org/pacemaker/doc/2.1/Clusters_from_Scratch/

and validate the configuration of the whole Pacemaker cluster stack. For some cluster components, configuration utilities already exist and PCS makes use of them. For others, PCS edits the configuration files and synchronizes them across all nodes if necessary.

History

Before Pacemaker and Corosync entered the world of open source clustering software, there was a cluster manager called CMAN. That is where PCS development actually started, with a utility called CCS. CCS stands for **cluster configuration system**. This tool was used to generate and distribute the only configuration file `cluster.conf`. CCS was a very simple project consisting of just a couple of Python[9] scripts. [11]

The development of PCS started in 2012, shortly after the introduction of the Pacemaker cluster stack to Fedora[10] and Red Hat Enterprise Linux[11]. The project was started by Chris Feist from Red Hat by transforming his CCS utility to PCS[12]. At the time, The CRM Shell[13] was the only high-level configuration tool for Pacemaker. The differentiating factor of PCS was that it was designed to be the all-in-one cluster configuration tool for the entire cluster stack. Its command syntax was designed to be more action-centric, providing abstraction over complicated Pacemaker configuration syntax and multitude of other administration tools. [10]

Pacemaker uses an XML[14] file to store its configuration and cluster state, called the **cluster information base (CIB)**. This file could be edited with the Cibadmin tool, but the configuration changes needed to be written in XML. Pacemaker also contains configuration tools like `crm_mon`, `crm_resource` and others that provide some level of abstraction over the CIB. PCS provides even higher level of abstraction than these tools.

Corosync has a configuration file `corosync.conf` that is stored on every cluster node. This file needed to be edited and then manually copied to each node. PCS introduced support not only for editing the file, but also for distributing it to other nodes. This was the original purpose of **the PCS daemon (PCSD)**.

Red Hat has always provided a web interface for cluster management. The first web interface was called Conga and was running atop of Luci, Ricci and Modcluster. With Red Hat's discontinuation of CMAN in Red Hat Enterprise Linux 7, a new web interface was needed. PCS was the natural choice, the new web interface with a REST-like back-end was integrated into PCSD. [12, 10]

[9] Python programming language, more at: https://www.python.org/

[10] The Fedora Project, more at: https://getfedora.org/en/

[11] Red Hat Enterprise Linux operating system, more at: https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[12] The rename was finished by this commit: https://github.com/ClusterLabs/pcs/commit/84f506ea6865b681db7e3032cb3a2ea1fc4240db

[13] The CRM Shell, more at: https://crmsh.github.io/

[14] XML – Extensible Markup Language, more at: https://www.w3.org/XML/

2.3 The Old Architecture

Since PCS originated from CCS, its architecture remained similar. PCS is built around commands. Command is a logical unit of actions that retrieve or change the state of the cluster. These commands interact with the cluster stack configuration utilities or change the configuration files directly. Commands can be run from the command line or launched by the web interface back-end. Due to the distributed nature of PCS, components from other cluster nodes can launch commands on remote nodes through the PCS daemon (PCSD).

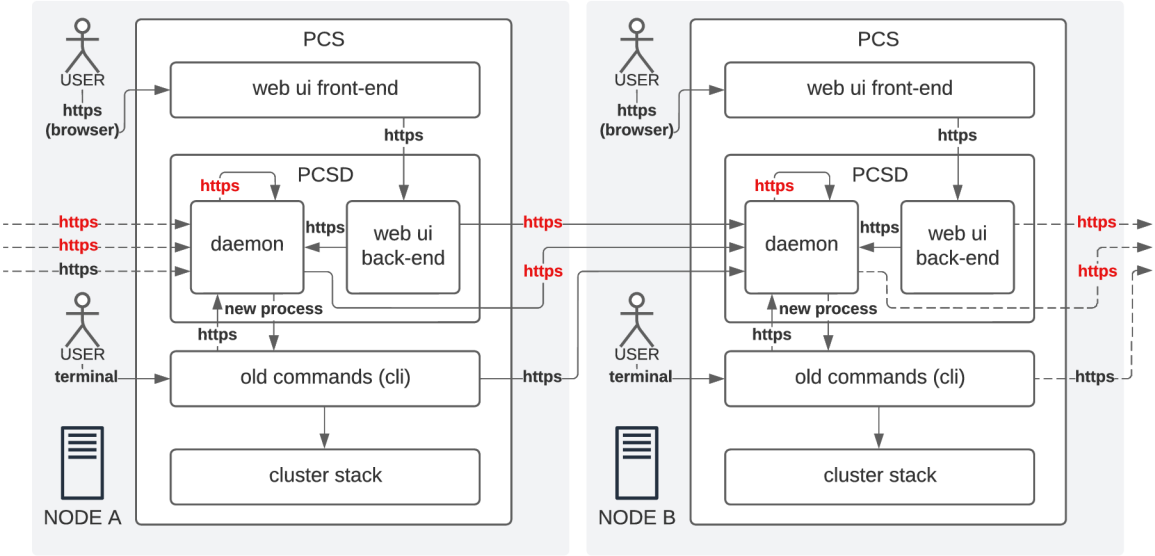

Figure 2.1: **The main components of the old PCS architecture** Arrows are annotated with the request type and show the flow of requests. Particularly interesting are many request paths through the daemon. Components may call the daemon which will then call remote nodes or the component may choose to call all remote nodes by itself. Unrecoverable request timeouts are marked with red. These timeouts can propagate to higher layers.

HTTPS requests are used in PCS to access a REST-like API for running commands. HTTPS requests are also much easier to use than low-level network communication methods, such as sockets. First, there were problems with PCS hanging if the target nodes were offline.[15] The requests were sent to PCSD, but sometimes due to network issues or the target being offline, PCS would hang. This was solved by adding timeouts to the requests and a new parameter, `--request-timeout`, that could override them. Timeouts usually cannot be adjusted globally because the time it takes commands to finish cannot be predicted. Shutting down a database server takes a lot of time and it is entirely dependent on its size and the database used.

The little excursion through history of PCS showed that there were many changes to the non-functional requirements throughout its development. Using the term old architecture is almost not fair since the functionality of the tool had to grow rapidly. There was not enough

[15] Bug 1292858 - pcs should timeout during network requests https://bugzilla.redhat.com/show_bug.cgi?id=1292858

information or time to design an architecture. The term old architecture is used to refer to an evaluation of the resulting implementation after it was finished.

Essentially, the web interface was placed on top of the command-line interface. While this allowed for reusing the already implemented functionality, the requirements of command line users and the web interface back-end are vastly different, which added more complexity. Error messages, warnings and results from command-line programs are all printed to the terminal. This meant that the web interface had to parse this output to provide results to the user. In the very distant past, many layers implemented this output parsing, not just the web interface back-end but sometimes even front-end. In some cases, the entire standard output was just dumped into the web interface. This was obviously not an ideal solution.

Commands

Commands in the old architecture are implemented as standalone functions that handle everything. Option parsing and validation are rarely handled by a function that is reused through multiple commands. They do not have a common interface and were often expanded with new functionality without significant architectural changes. Many of the old commands are also too complex to maintain at this point in time.

As shown in the architecture diagram 2.1, commands can call local PCSD to communicate with other cluster nodes. Some of them also talk directly to PCSD on other cluster nodes. This creates multiple execution paths that each have different problems.

The Ruby PCS Daemon

The PCS daemon is a server application that fulfills multiple roles in the PCS package:

- server for performing actions issued by remote nodes;
- configuration synchronization tool;
- web interface back-end.

PCSD is a daemon process that contains a server that handles HTTPS requests. It has become overly complicated over time due to the variety of its uses. Unlike PCS, which is implemented in Python, PCSD is implemented in Ruby[16]. This adds dependencies to the PCS package in the form of the Ruby interpreter and Ruby gems that expose PCS to more security threats.

Focusing on PCSD as a command running server, it is not just a simple communication layer that executes PCS commands implemented in Python. Some functionality of PCS is reimplemented directly in PCSD since it could not be reused from Python. This is not ideal from a maintenance standpoint; the Ruby and Python implementations of the same functionality can easily diverge.

[16]Ruby programming language, more at: https://www.ruby-lang.org

Timeout problems also affect PCSD itself. There is a bug[17] in which the web interface fails to add a new node to the cluster. The `pcs node add` command only needs to be performed on one node. The web interface back-end makes requests to all nodes in the cluster until the command is successfully executed. If the PCS command for adding a node runs longer than the request timeout, remote PCSD[18] returns an error. But the command continues to run on the remote node, and the new node is added to the cluster. The web interface back-end tries all remaining nodes, but all return errors. The `pcs node add` command now fails because the new node is already in the cluster. Since the web interface back-end got an error response on all requests, the action will be reported to the user as unsuccessful. For a visual representation of this bug, see the sequence diagram in Figure A.2.

2.4 The New Architecture

The new architecture brings a clear separation of responsibilities between the layers of the architecture shown in Figure 2.2. The new clients are communicating with a REST API that provides a unified way of calling the new commands from the lib package.

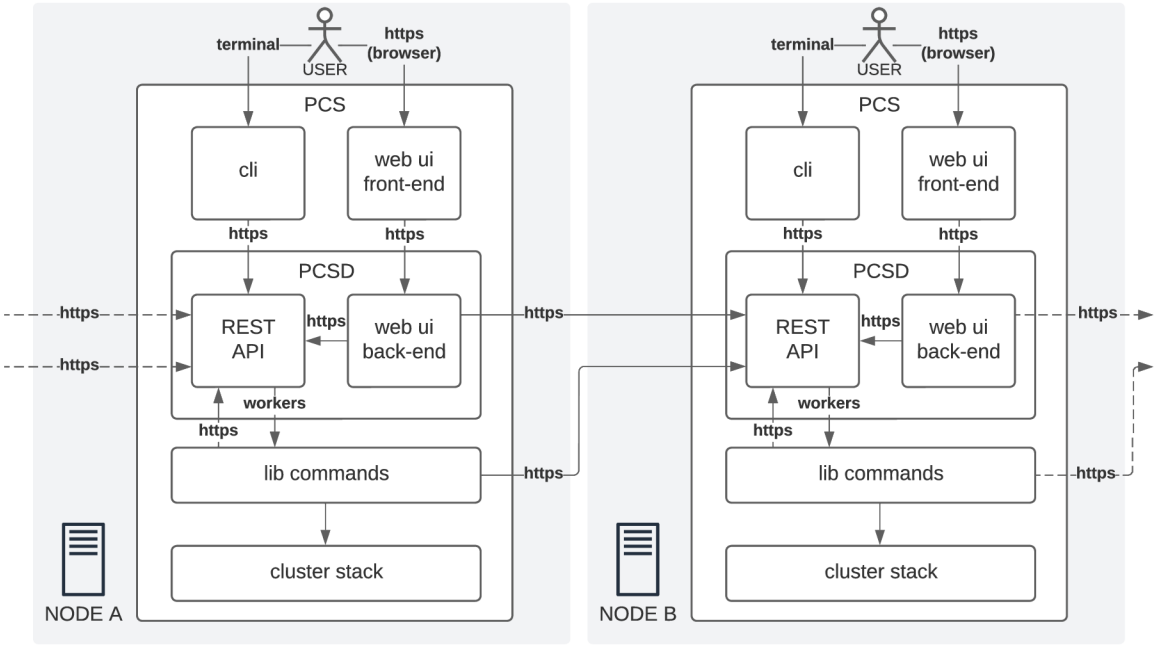

Figure 2.2: **The new PCS architecture** is a solution to problems outlined in the previous section. Arrows are annotated with the request type and show the flow of requests.

[17] Bug 1539694 - [GUI] Adding a node may take longer than 30s and produce a timeout error https://bugzilla.redhat.com/show_bug.cgi?id=1539694

[18] Even the local node is considered a remote node in this case

Library

Lib(rary) commands are more modular and use a more layered approach to implement functionality that allows better code reuse. In the old architecture, the commands were a 1:1 mapping to the command-line interface. In the lib package, commands represent actions, and not all of them are necessarily command-line interface commands.

The library aims to be a Python API for actions that can be used more easily by clients and other projects. This API standardizes the interface of all lib commands. The first argument is the `LibraryEnvironment` object, and the subsequent arguments are the command parameters that depend on the command. Lastly, there is an optional keyword argument for force flags, which can override configuration and input validations.

The `LibraryEnvironment` object is an abstraction layer for all resources that commands might require. It provides facilities for manipulating configuration files, running other cluster utilities, intra-cluster communication, user permissions, the report processor, and so on. The significance of report processors is explained later in the subsection called Reports.

Use of Type Hints and Data Classes

Python, being a language with dynamic typing, allows greater flexibility when using variables. On the other side, this greater flexibility can also cause run-time errors. To prevent programming errors and improve code readability, the new architecture uses type hints along with a third-party static type checker Mypy[19]. Type hints also open doors to many quality of life improvements for the developers like code completion and autocompletion in IDEs[20]. [5]

A long-standing problem with the old architecture is the passing of data to functions in dictionaries. This approach was used to minimize the argument list size since many commands work with a lot of data. The problem is that there is no guarantee on which keys are available in a dictionary. The first solution to the unpredictable structure of dictionaries was to use named tuples. This solution was quickly superseded by the use of data classes. Unlike named tuples, which are based on tuples, data classes are based on standard Python classes. This adds all the benefits of object-oriented programming, such as inheritance and class methods. Data classes are by default mutable but can optionally be made immutable. This makes data classes more versatile than named tuples. For PCS releases targeting Python 3.6+, the support is taken from the Dataclasses package[21], newer PCS releases target Python 3.9+ and use the Dataclasses module from the Python standard library introduced in Python 3.7. [25]

Another advantage of data classes comes from combination with the Dacite project [7]. Dacite is based around methods for converting the data classes to dictionaries and back. The dictionaries can be easily serialized to JSON[22] and back with the help of the JSON

[19]The Mypy project, more at: http://mypy-lang.org/
[20]IDE – Integrated Development Environment
[21]Dataclasses package in Python Package Inventory, more at https://pypi.org/project/dataclasses/
[22]JSON – JavaScript Object Notation, more at: https://tools.ietf.org/html/rfc8259

package[23] from the Python standard library. Since PCS is a distributed application, it needs to communicate over the network. The data is exchanged using data transfer objects (DTO) which are implemented as immutable data classes. As Fowler explains, DTOs reduce the number of network requests and allow more data to be returned in each response. Furthermore, the Dacite project separates the logic for object serialization which is also an advantage of using DTOs. [3]

Reports

In order to provide feedback to users, PCS uses special objects called reports. Reports are designed to store errors, warnings, debugging data or just general information in a presentation independent fashion. This solves the problem of error messages in the old architecture that were written for the command-line interface. This made them unsuitable for use in the web interface.

The report class contains a report code that makes automated processing of reports easier. In addition to that, it contains all the raw data used to construct the original message. The receiver can choose to append or make its own message. One of the use cases is a report for the command line. It appends the original report with a hint about the command that can be used to resolve the particular error.

PCS reports are the prime example of architectural benefits of data transfer objects. Initialization is handled by the data class which reduces the amount of code needed to define a report. Type hinting support allows for static analysis with Mypy. Serialization logic is handled by Dacite which further minimizes the amount of code for every report. Reports are made immutable so that they cannot be manipulated during their lifetime.

When reports are created, they are put into a report processor. `ReportProcessor` is an abstract class that defines an interface for processing reports. Its main purpose is to allow for report distribution in a separate channel from results. This simplifies automated output processing for other components. Thanks to `ReportProcessorToConsole` class, the output for a terminal user looks like before. There are more implementations for different purposes in PCS.

Another big advantage is that reports are distributed at the time of their creation while the command is running. This was also possible in the old architecture for command-line users. But for the web interface, the output that is now considered reports was only distributed to the web interface after the command finished.

The report processor also helps with error handling. Commands in the old architecture exit after the first error. Any reports with severity of error change internal state of the report processor. This state can be checked by calling `has_errors` method which is then used by lib(rary) commands to raise a `LibraryError` exception. This makes PCS print all the errors that can be detected in a single run, which improves user experience.

[23] The JSON package in Python standard library, more at: https://docs.python.org/3.6/library/json.html

The Python PCS Daemon

The new daemon in PCS is implemented in Python as the rest of PCS. Its minimal implementation was shipped in PCS 0.10.1 and replaced the old Ruby daemon. The Ruby daemon had to be added back in PCS 0.10.5 due to the utilization spikes caused by spawning a new process for Ruby code from Python[24,25]. The new daemon receives all requests and serves as a proxy for the old Ruby daemon via a socket.

The new daemon should provide a REST API that no longer suffers from timeout issues. The REST API is just a simple interface for running lib(rary) commands remotely. This removes the added layer of complexity in the Ruby daemon that deals with calls to the daemons running on other nodes.

The remaining functionality of the old Ruby daemon, such as synchronization of configuration files, will be ported to the new daemon. There is no set plan on how to do this at the time of writing, so this functionality is not shown in Figure 2.2.

Current State of the Architecture

As all software projects, the requirements for the PCS project have changed rapidly since its beginning. As the project grew, the old architecture began to prevent the new goals from being reached. However, as with any complex application, it cannot be rewritten from scratch in a short period of time.

At the time of writing this thesis, the current version 0.11 of PCS is transitioning to the new architecture described in the previous section. The new lib(rary) commands are slowly replacing the old commands. Reports are already being used in the lib(rary) commands. As discussed earlier, the new Python daemon is already part of PCS. The remaining task is to implement the REST API and its underlying layer for task management.

[24] Bug 1783106 – GUI: sinatra_cmdline_wrapper.rb causes CPU utilization spikes: https://bugzilla.redhat.com/show_bug.cgi?id=1783106

[25] PCS 0.10 release notes: https://github.com/ClusterLabs/pcs/blob/pcs-0.10/CHANGELOG.md

Chapter 3

Analysis and Design

In the previous chapter, the problems of the current PCS architecture were outlined along with plans to address them. In this chapter, the goals of this thesis are formulated. With the goals set in stone, a description of the concepts that are crucial for understanding the requirements and design decisions follows. Based on the analysis of requirements, the first design decisions are made on a theoretical level. These decisions are cemented by implementing a proof of concept. The findings discovered up to this point of the chapter culminate in the creation of a design document for the final implementation.

3.1 Goals

Starting with the title of this thesis, Asynchronous Task Processing in the PCS project implies that there are some tasks that need to run in an asynchronous environment. Since the new PCS daemon should implement a REST API for the lib(rary) commands and it is written using the Tornado library, this is the asynchronous environment. Tasks are the new lib(rary) commands, and the problem is that they are not implemented asynchronously. There is a good reason for this difference. Without diving into the concept too much, it is important to understand that asynchronous programming is a form of concurrency. The REST API is designed to handle many concurrent requests from many users (or clients, in this case). Although multiple commands can be run at once, this does not make them a good candidate for the asynchronous programming model. More about asynchronous programming is given in Section 3.2. Therefore, the main goal of this thesis is to resolve how tasks can be executed from the asynchronous environment with the requirements of lib(rary) commands.

Problem Analysis

In the previous chapter, it was demonstrated that commands can time out in many parts of their execution path. If the commands are represented as tasks, the tasks hold all of the command's current state and results. The new REST API should only look at task metadata and provide responses very quickly. There is no waiting until the command is finished, eliminating timeouts. The client (lib command or some user interface) should poll the REST API for the status of the task until it finishes.

The REST API should be a fairly lightweight interface that serves only as a communication layer. It needs to be connected to some kind of task manager. In this thesis, it will be called **asynchronous scheduler**. The asynchronous scheduler handles the creation of tasks, managing where and how they run, communication with tasks, task lifecycle management, and storage of all important data associated with these responsibilities.

The thesis assignment suggests using the Python standard library in the form of Multiprocessing package and AsyncIO library. If a Linux package contains a dependency that is not included in the target distribution, it needs to be included in the package. This adds overhead in the form of checking new versions of dependencies in every PCS release. Every time the dependency is updated, there is the risk that it will break the application. There is no such problem with the Python standard library. Its functionality only changes with a new Python version. PCS only changes the minimum Python version with a major version bump when necessary. Therefore, it is advantageous to use the Python standard library over third-party libraries if possible.

3.2 Concurrency in Python

It is clear from the goals of this thesis that concurrency in Python will be heavily used in the implementation. Before going any further, concurrent programming flavors in Python and their caveats need to be explained to understand the following sections.

Python is an interpreted language, and there are many Python interpreters. The original and reference interpreter is called CPython[1]. PCS was developed with CPython in mind since it is the most widely used Python interpreter. One of its implementation limits is **global interpreter lock (GIL)**. When accessing Python objects or calling Python C API functions, the GIL protects access to the shared resources used by the Python interpreter. This effectively means that the code can run only in one thread at a time – the thread where the GIL is acquired. The execution of other threads is blocked until the lock is released. Since Python is an object-oriented language and the Python standard library takes advantage of it, working with Python objects is very common. Therefore, using thread-based concurrency in Python usually does not result in performance benefits associated with concurrency. [20, 8]

Asynchronous Input and Output Processing

Most of the information in this section was found in the book Using Asyncio in Python by Caleb Hattingh. [8] One of the ways to allow concurrency that sidesteps the GIL is asynchronous input and output processing. Input and output operations, like reading from a disk, are really slow in comparison to the speed of today's processors. Another and more relevant example is network communication, where servers spend most of their time waiting for clients or other code that provides the content of the responses. Servers also need to serve many clients at once, so concurrency is also necessary.

In web servers such as Apache[2], each connection is handled by a separate thread. [26] Network communication is handled by blocking functions that stop the execution until

[1] CPython interpreter, more at: https://github.com/python/cpython
[2] The Apache HTTP server project, more at https://httpd.apache.org/

the result of the function is available. The operating system then handles switching between threads to make the best use of the processor. Depending on the operating system and resources available, this might be a true parallel approach when multiple threads run on different cores of the processor. This approach has some drawbacks:

- Since the operating system switches between threads, a switch can occur at any time. Applications must be properly designed to prevent race conditions. Race conditions are difficult to diagnose and find since the execution path is non-deterministic.
- There is a limited number of threads that can be created. Each thread allocates some space on the stack. This is less of a problem in modern systems with gigabytes of operating memory. Operating systems also impose some limits on the maximum number of threads.
- There is no way for the operating system to know when is the appropriate time to switch between threads. There may be times when the thread is waiting for data on a socket, and the operating system switches to it. This leads to inefficient use of processor time.

The asynchronous input and output model does not rely on the operating system to provide concurrency. The code is executed in a single process with a single thread. This is why GIL in CPython is not a concern. To give a similar example, the same web server could be implemented with Tornado in Python. It eliminates all of the drawbacks of multithreading from the previous list:

- The functions decide when a switch to another function can occur. The execution path is deterministic, so access to shared resources can be more easily controlled.
- There is no limit to how many coroutines can be created at once, except for the available resources.
- Functions are only switched to if they can continue to execute.

To achieve concurrency, all code is executed in an **event loop**. Only special asynchronous functions called **coroutines** can be executed by the event loop. The event loop is a special mechanism that switches between coroutines and handles asynchronous events. An example of an asynchronous event is an incoming request to a server. In this model, the coroutines need to be specially designed to return control to the event loop so that other coroutines share the resources fairly.

Concurrency is achieved by quickly switching between coroutines. Because of that, it is important that there are no coroutines that block the event loop for too long. To do that, non-blocking functions can be utilized. Another way is to use asynchronous sleep, which forces the event loop to switch to another coroutine for some period of time. These are the high-level concepts, more details are implementation-specific. [9]

Multiple Processes

Another form of concurrency in Python uses multiple processes. The GIL does not prevent true parallel execution because every process has its own Python interpreter. This is a very

different approach from asynchronous input and output suited for different kind of tasks. In particular, tasks that require heavy computation can be finished sooner when multiple processor cores can work on them at once. Similarly, tasks that do not need to share data can be executed in parallel to speed up their processing.

However, there are some drawbacks to using multiple processes. Workloads that use shared memory need to employ an interprocess communication method that consumes resources. All of the dangers associated with switching threads also apply to switching processes.

3.3 The AsyncIO Library

In this section, a closer look will be taken at the AsyncIO library[3] provided in the Python standard library since it was chosen for the implementation of the asynchronous scheduler. There are alternatives to the AsyncIO library which will be mentioned along with the history of AsyncIO library and why it was chosen. At last, the more implementation-specific concepts of asynchronous input and output processing model will be further explained based on the implementation of AsyncIO.

Brief History and Alternatives

Long before the AsyncIO library was introduced, Twisted[4] was the go-to library for asynchronous programming in Python. It is a networking framework for callback-based asynchronous programming. It uses `Deferreds` – objects representing a return value that execute callbacks when the return value becomes available. Futures in AsyncIO are very similar and many other architectural decisions were influenced by Twisted. Twisted cannot be used because its integration was deprecated in Tornado 6.0 which is the version used by PCSD.[5]

The AsyncIO library was added to Python 3.4 as a provisional API. This status means that backward-incompatible changes may occur in the next Python releases[6]. Python 3.5 added some important syntax improvements in the form of async/await syntax. The release notes for Python 3.6[7] remove the provisional status of AsyncIO, making it a stable API. With the level of support for the Python standard library and its mature status, this is the module that was chosen to supplement Tornado. [8]

Curio[8] and Trio[9] are other well-known libraries that provide an implementation of the asynchronous input and output model. Curio was started as a project that would be smaller and faster than AsyncIO. Curio is not designed to be compatible with other asynchronous libraries like Tornado making it not usable in PCSD.[10] Trio was mainly inspired by Curio, but

[3] AsyncIO library, more at: https://docs.python.org/3.6/library/asyncio.html
[4] Twisted – event-driven networking engine, more at: https://twistedmatrix.com/trac/
[5] See Tornado documentation: https://www.tornadoweb.org/en/latest/twisted.html
[6] Provisional API definition in Python documentation: https://docs.python.org/3.6/glossary.html#term-provisional-api
[7] Python 3.6 release notes: https://docs.python.org/3.6/whatsnew/3.6.html#asyncio
[8] Curio on GitHub: https://github.com/dabeaz/curio
[9] Trio on GitHub: https://github.com/python-trio/trio
[10] From the Curio readme: https://github.com/dabeaz/curio/blob/master/README.rst

was further refined by focusing on usability and correctness.[11] AsyncIO already provides all of the features that will be needed in this thesis. Furthermore, Python core developers constantly look at these libraries as an inspiration for future development of AsyncIO.[12] In the future, the novel concepts from these libraries could land in AsyncIO, further confirming it as a good choice.

Coroutines

Coroutines are Python functions defined with the `async def` keywords. Before Python 3.5, coroutines were defined with decorators `@asyncio.coroutine` (when using AsyncIO) or `@tornado.gen.coroutine` (in Tornado before AsyncIO). They share a lot of code and properties with generator functions. Generator functions are used to create iterators in Python. They can remember their internal state and point of execution to return multiple values to be iterated over. That is very similar to coroutines, which also remember their internal state which allows for switching between them.

The switch between coroutines is achieved with the keyword `await`. The `await` keyword suspends the currently executed coroutine and causes the event loop to execute and wait for the result of the coroutine defined as its parameter. When that coroutine finishes, the event loop resumes execution after the await in the calling coroutine. [8]

Event Loop

It was already briefly explained what the event loop is. The event loop not only facilitates switching between coroutines, but also handles incoming asynchronous events. This is the reason why Tornado is a very performant web framework. Coroutines for handling connections are only switched to if they need to handle a connection, not when they are waiting for the network. Speaking about Tornado, since AsyncIO was introduced, Tornado uses its event loop implementation, making integration of other libraries easier. That is also why AsyncIO (or Python native) coroutines are preferred over Tornado's generator approach.

In the quick overview of asynchronous programming in section 3.2, it was mentioned that coroutines should not block the event loop. Unfortunately, some workloads cannot be adapted to coroutines. The solution is to launch these tasks in a new thread or process. A standard approach in AsyncIO is to use the process or thread pool from the `concurrent.futures` module. [8]

Futures and Tasks

A Future represents the state and optionally a return value of a non-blocking function. Its API allows for checking the status of the task and cancellation. Status is only binary, the function can be completed or not completed. It is also possible to set callbacks that are executed on the event loop when Future is completed.

[11] From the Trio readme: https://github.com/python-trio/trio/blob/master/README.rst
[12] Talk of the AsyncIO maintainer: https://youtu.be/m28fiN9y_r8

In AsyncIO, Task is a subclass of Future. A Task is created when a coroutine is scheduled to run on the event loop with the `create_task` function. For most uses, Tasks are recommended over Futures as a more high-level interface. [8]

Communication

AsyncIO provides queues similar to the ones from the Queue module of the Python standard library. These queues are meant to be used for distribution of work between coroutines. They are designed to work with asynchronous code but are not thread-safe. [16]

3.4 Asynchronous Task Processing

Going back to the name of this thesis, Asynchronous Task Processing in the PCS Project mentions tasks. These tasks are not tasks from the AsyncIO library. Tasks are an abstraction layer for status and eventual result of remote actions implemented as functions in the `lib` package of PCS. These functions can also be referred to as lib(rary) commands. It was stated in the beginning of this chapter that lib(rary) commands are not implemented asynchronously. Now that the concept of the event loop was explained, it can be said that the lib(rary) functions would block the event loop. The worst kind of blocking could even be tens of minutes long. Some lib(rary) commands accept a `wait` parameter that makes the command wait until the changes are applied with a default timeout of 60 minutes. During this time, the server would not respond to any requests, which is unacceptable. It is clear that the tasks need to run outside the event loop. The alternative is to rewrite lib(rary) commands to use async/await syntax. This would be a huge undertaking. Every PCS component would need to be rewritten, and the benefits do not outweigh the costs of this solution.

Task Execution

When describing the event loop, the `concurrent.futures` module was mentioned to allow the execution of blocking functions from coroutines. From the description of concurrency in Python, the thread pool executor can easily be disqualified due to the GIL[13]. A good candidate seems to be the process pool executor from the same module.

One of the requirements of the PCS development team was that the processes that launch the tasks should be daemonic. Daemonic workers allow for faster task processing since incoming tasks do not need to wait until a worker process is created. The only concern with daemonic processes is that if a library function manipulates the global state, subsequent use of this worker may affect the later tasks. This is the reason for another requirement – daemonic processes should be restarted after some number of processed tasks.

[13]GIL – global interpreter lock, see Section 3.2

When looking at the documentation of `ProcessPoolExecutor`, these requirements cannot be met. However, dissecting its CPython implementation reveals possible candidates at a lower abstraction level:

- Level 4 – `concurrent.futures.ProcessPoolExecutor` – this is the highest level of abstraction and the interface is designed to be used in AsyncIO.
- Level 3 – `multiprocessing.Pool` – this is the process pool that `ProcessPoolExecutor` uses. Its setup provides options for aforementioned requirements. It is the best candidate at this moment.
- Level 2 – `multiprocessing.Process, multiprocessing.Queue` – these are Python high-level objects that represent processes, thread-safe data structures and synchronization primitives. If `multiprocessing.Pool` cannot be used, this is the lowest level of abstraction that should be used in such a high-level language. The problem is that, at this level of abstraction, a reimplementation of the process pool functionality would be needed. That would cause an enormous increase in scope of this project. Especially when the requirement of testing is considered. Even from a maintenance standpoint, there is no reason to have such complex functionality unrelated to PCS to maintain.
- Level 1 – `os.fork...` – Python wrappers for system calls – mentioned here just for completion
- Level 0 – the C implementation – all Python wrappers translate to the C calls made by CPython, and this is mentioned here just for completion again.

The process pool of the Multiprocessing package has many methods to submit tasks. These are representative of these widely recognized types of parallelism in programming languages:

- Data parallelism – the same operation is applied to independent elements of a large data structure. Because the operations are independent, parallelism can help to obtain results in a much shorter amount of time. These are the `Pool` methods that contain the word map.
- Functional parallelism – also called task parallelism – allows one to run independent program parts in parallel. These are the `apply` and `apply_async` methods.

Even by its name, tasks in PCS are a representation of task parallelism. Even in the old architecture, multiple commands can be launched at once by running the utility in different terminals. There is no need to manage which commands can be run at the same time. They can be considered independent blocks of code since PCS or the utilities it uses safely handle access to shared resources.

From the description of tasks, the `apply_async` method is the most suitable for sending tasks to be processed in a worker process. This is an implementation of a task pool – tasks are being put into a shared task pool where a predetermined number of processes retrieve tasks for execution. [18, 21]

Obtaining Results of Tasks

The `apply_async` method returns an `AsyncResult` object. This object can be used to obtain the result of the task. The `get` method waits for the result and returns it. This method cannot be used in the asynchronous scheduler since it would block the event loop. It is possible to implement an asynchronous waiting mechanism using the `ready` method. However, an even better approach is to have the task notify the scheduler about its completion. This can be achieved by sending a message to the scheduler.

Task Killing

The mechanism for killing tasks is an optional extension of the asynchronous scheduler. The decision to implement it was made early on since it affects the architecture very significantly. Lib(rary) commands still need to be terminated from time to time. Unavailability of cluster nodes or network instability may cause commands to never finish. Another entirely different use case is for users who want to stop a PCS command they launched. This is convenient for a command that is running for too long or when a user notices a mistake in the command.

The chosen process pool does not provide any interface for controlling worker processes. In the same way that PCS can be terminated on the console by pressing `Ctrl+C` and the terminal issuing a `SIGINT` signal, the scheduler can send a signal to the worker process. The only problem is that the PID[14] of the worker must be known to the scheduler. Before the command is executed by a worker process, it can send a message to the scheduler with its PID.

Interprocess Communication

In the previous subsections, sending messages to the scheduler was proposed. Another requirement of lib(rary) commands is the ability to send reports back to the asynchronous scheduler so that clients can retrieve them while the command is running. These messages require a form of interprocess communication, as the messages are passed between the worker processes and the PCSD process.

The PCSD process requires an asynchronous messaging interface. The AsyncIO library provides queues for use in coroutines, as was explained in Section 3.3 (Queues). The problem is that these queues are not thread-safe. The queue is supposed to be a bridge into worker processes and race conditions could occur since switching between processes is handled similarly as switching between threads.

A very quick search on the Internet returns the Aioprocessing[15] library. This library uses thread pool executors to provide asynchronous interfaces to the objects that only provide blocking methods from the Multiprocessing package. Due to the use of threading which adds extra overhead due to GIL, it was eliminated from the list of candidates.

Since the Multiprocessing package is already used, it is logical to look at its implementation of interprocess communication. Documentation of the Multiprocessing package suggests

[14] PID – process identifier

[15] Aioprocessing library, more at: https://github.com/dano/aioprocessing

using pipes or queues for message passing. Pipes allow only a single producer and consumer, so they would need to be set up for each worker separately. Queues were chosen because the process pool is a multi-producer environment, and the asynchronous scheduler is going to be the sole consumer of these messages. In addition to blocking methods, the `Queue` class also provides non-blocking methods `put_nowait` and `get_nowait` that can be used in the asynchronous scheduler.

Sockets could also be used to transfer messages. Although AsyncIO can handle sockets asynchronously, a message-passing protocol would need to be designed to exchange messages. Queues provide a higher level of abstraction while providing all of the features that are needed.

3.5 Proof of Concept

So far, many theoretical solutions have been presented. To test if they work, a proof of concept was implemented. The proof of concept only uses the AsyncIO library to test the asynchronous task processing, task killing, and interprocess communication methods proposed up to this point. Details of its implementation and test results that demonstrate that the chosen approach works can be found in Section 4.1. Some key details will be mentioned here to show how they influenced the design decisions described later.

Structure

The `scheduler.py` module contains the asynchronous scheduler object. This object handles process pool management, interprocess communication, creating tasks, and storing tasks status. Internally, every task is represented by a `Task` object implemented in the `task.py` module. Data types used for interprocess communication are defined in the `messaging.py` module. Instead of a REST API, the tasks are generated and fed into the scheduler using the `test_scheduler.py` script. Tasks that mimic the possible scenarios of real lib(rary) commands are implemented in the module `workloads.py`. Lastly, the `common.py` module contains common functions and structures used by other modules.

The Scheduler

The asynchronous scheduler is enclosed in the `Scheduler` class. The `Scheduler` class has the `update` method that periodically runs in the event loop and handles all responsibilities of the scheduler which are:

1. Scheduling tasks for execution that were submitted to the scheduler.
2. Scheduling retrieval of messages from the queue. (`worker_communicator`) from workers

Interprocess Communication

For simplicity of access to data fields, messages are ordinary Python classes. Ordinary classes were also chosen because they can be easily serialized by the Pickle module used by `multiprocessing.Queue`. [18, 19]

All messages used for interprocess communication are instances of the `Message` class. `Message` contains only a task identifier of the receiving task and a payload. There is no need to include a payload type since the object type is preserved by the Pickle module. The payload is `StatusUpdate` or `Report`. `StatusUpdate` carries a new task state and a PID of the worker. `Report` just a string in its message field and simulates a PCS report. Messages are received and handled by the scheduler. The scheduler then updates task information through their public interface.

The original intention was to use a `Queue` from the Multiprocessing package. Attempts to use it in the proof of concept generated a run-time error: „Queue objects should only be shared between processes through inheritance“. The error is generated because the queue was sent to the worker process as a parameter of `apply_async` call. The Pickle module used by the `multiprocessing.Queue` was not able to serialize this object.

The solution is to use the `multiprocessing.Manager.Queue` object. This object wraps the `multiprocessing.Queue` in a proxy object that can be pickled.[16] [18]

Task Killing

The task killing was tested by issuing the `kill` command from the terminal. The only parameter this command takes is the number of the signal to be sent. From the list of signals, three were chosen to test how the worker processes of the process pool react to them:

- `SIGKILL` – killing a task is considered a last resort in the same way as the kill signal is for terminating a process.
- `SIGINT` – usually mapped to `Ctrl+C` in terminal emulators, is a pretty common way for users to terminate programs running in the terminal.
- `SIGTERM` – terminates a process gracefully. Default signal used by the `kill` command. After issuing the `SIGTERM` signal during testing, the output showed that a new worker was started.

The testing has shown that the `SIGTERM` signal is safe to use for killing a process. Right after the signal was issued, a new worker process was spawned by the process pool. Full findings are presented in the test report of the proof of concept in Section 4.1.

[16] Answer by michael (https://stackoverflow.com/users/4804903) on Stack Overflow: https://stackoverflow.com/a/45236748

3.6 Design of the Asynchronous Scheduler

With knowledge of the concepts used by the asynchronous scheduler, its components can be designed. First, the basic principle of work was illustrated by a sequence diagram that shows how the scheduler operates with the process pool. The design of classes was inspired by the proof of concept and was further extended to a class diagram. When the classes were designed, a package diagram was created to distribute them into modules and illustrate imports. All of these diagrams were created using the Unified Modelling Language (UML).[1] Because of their size, they were inserted into Appendix A. These diagrams will be referred to throughout this section.

The Scheduler Core

The entire scheduler logic is included in the `Scheduler` class. Its public interface follows the user expectations that are the same as for the REST API:

- Run a lib(rary) command that is represented by a task identifier.
- Check the status of a running task.
- Optionally, provide a method for cancelling a running task.

From these actions, the public interface of the scheduler that the REST API is going to use was created:

- `new_task(command_dto:CommandDto) -> str` – creates a task and returns its identifier.
- `get_task(task_ident:str) -> TaskResultDto` – returns the status of a task.
- `kill_task(task_ident:str) -> None` – marks a task to be killed.

The `task_ident` name for a task identifier was chosen as a replacement for `task_id`. While `task_id` is an acceptable variable name, its concatenation to `id` is not. There is a name collision with a built-in function `id()` in Python. Python interprets `id` as a pointer to the `id` function instead of an identifier.

The following properties and methods are responsible for the process pool management, task management, and interprocess communication:

- `_proc_pool` – contains reference to the pool of workers.
- `_proc_pool_manager` – contains the manager of shared resources for the process pool.
- `_created_tasks_index` – a list of task identifiers in the created state. Traversing and querying state of all the running tasks may be quite an expensive operation. Task identifiers that need to be sent to the worker pool can be quickly obtained here.
- `_task_register` – a dictionary of task objects addressed by the task identifier. Holds all of the tasks that are known to the scheduler.

- `_worker_message_q` – queue used for sending messages from workers to the scheduler.
- `_logging_q` – queue used for sending log records from workers to the scheduler.
- `_logger` – a reference to the PCSD logger.
- `perform_actions()` – coroutine that performs all responsibilities of the scheduler. This method should run periodically on the event loop.
- `_receive_messages()` – coroutine responsible for emptying `worker_message_q` that contains messages from workers.
- `_schedule_tasks()` – coroutine responsible for sending tasks to the worker pool.
- `_garbage_collection()` – coroutine responsible for killing tasks and deleting tasks that were either running too long or did not have their result retrieved by the client.
- `terminate_nowait()` – cleanly terminates the scheduler, used for resource cleanup on daemon exit.

One crucial difference between the class diagram and this description of class attributes can be spotted. The class diagram illustrates access restrictions to a class attribute with a plus for public attributes and a minus for private attributes. In Python, there are no truly private attributes with access restriction enforced during runtime. Attributes that should be considered non-public by the callers and inheriting classes should have a leading underscore. That is why leading underscores are used here but not in the class diagram in Figure A.4. [6]

The `perform_actions` is the equivalent of the `update` method in the proof of concept and it schedules the `schedule_tasks`, `receive_messages` and `garbage_collection` methods. The asynchronous scheduler adds a garbage collector to remove task objects past their lifetime. A more detailed explanation of how these methods work can be found in the sequence diagram in Figure A.3.

Task Scheduling

When the scheduler receives a task description, the task representation is created and the task identifier is placed in a queue for scheduling. The scheduler then runs the `schedule_tasks` method to send the tasks to the worker pool. This is a familiar concept taken from the proof of concept.

Interprocess Communication

The original idea used for the proof of concept was that the scheduler as the receiver of messages should also process them. The scheduler would only manipulate tasks through their interface. This separation of responsibilities added unnecessary interfaces to the task class. If messages can be passed into tasks, the task can update its private parameters without the need for a public interface.

Unchanged form the proof of concept, the `Message` object contains only the task identifier and a payload. Adding a payload type was considered because treating an object on the basis of its type is not considered good practice in Python. Duck typing is preferred, when the object type should be treated by its content. This would be however quite impractical – message handlers would have to be tried one by one until one of them accepts the message. So, the decision was made to keep `Message` same as in the proof of concept except that now it is a frozen data class.

Message can contain three payload types that are better suited to their purpose than those used in the proof of concept:

- `TaskExecuted` – sent by the worker right before the lib(rary) command is executed. Its only field is `worker_pid` to tell the scheduler in which process the task is going to run.
- `TaskFinished` – sent by the worker when the lib(rary) command finishes or raises any exception. Contains fields:
  - `task_finish_type` – contains more information about how the lib(rary) command finished.
  - `result` – contains a return value of the lib(rary) command, None if the command did not finish.
- `ReportItemDto` – sent by the report processor in the worker. This type is imported from PCS.

The new `TaskExecuted` and `TaskFinished` message types remedy the limited usability of the `StatusUpdate` message from the proof of concept. The task state change is represented by the type of the message now and every message only contains relevant information. `StatusUpdate` would need to be modified to also contain the return value if it was used in the asynchronous scheduler. The downside would be that every time this message was sent, it would contain unnecessary information.

Task Killing and Garbage Collection

The concept of task killing by sending signals was taken from the proof of concept. Task killing can be requested by garbage collection or the REST API by calling `request_kill` method on the `Task` object. Garbage collection in a second step requests the `Task` object to kill the worker with its `kill` method. A better description is provided based on the implementation in Section 4.3.

There are two timeouts that determine when the task is killed by the scheduler:

- Unresponsive timeout – the only way to detect a task malfunction is to calculate the time since the last message was delivered. Tasks that do not produce reports and need to wait for long periods of time can be modified to send more messages. Garbage collection is responsible for detecting this timeout and killing the task.
- Abandoned timeout – useful when the connection to a client is lost – the task becomes abandoned. In the finished state, the timestamp of the last message always belongs

to the `TaskFinished` message, so it is indicative of the time when the task has finished. To prevent abandoned tasks from piling up, the task result is available only for this period of time before the task is deleted by the garbage collector. The task does not need to be killed in this case.

Garbage collection is not responsible for deleting tasks that have been completed. These tasks are deleted by the `get_task` method of the `Scheduler` class when the task state is `FINISHED`. There is no reason for the client to need the result again and it makes the garbage collector use less resources.

Task Representation

Any lib(rary) command launched through the REST API is represented in the asynchronous scheduler by the `Task` class. Its main purpose is to store information about the progress of the task that clients can query. Its second purpose is to store information relevant to task management. The tasks are stored in `Scheduler._task_register` which is a dictionary where the key is the task identifier and the value is the `Task` object.

The `Task` class has the following instance variables:

- `_task_ident` – stores the identifier used by the client and scheduler to deliver messages to the right task.
- `_command` – stores information about the command and its parameters.
- `_reports` – a list of reports generated during the command runtime.
- `_result` – contains the return value of the command when or if it finishes.
- `_state` – represents the state of the task.
- `_task_finish_type` – carries information about how the command finished.
- `_kill_reason` – signals that a kill was requested and why.
- `_last_message_at` – timestamp of the when the latest message was delivered from the worker process.
- `_worker_pid` – contains the PID of the worker where the command is running.

Supported methods for tasks are:

- `state(new_state)` – sets the task state to `new_state`.
- `state()` – returns the task state.
- `task_ident()` – returns the task identifier.
- `_is_abandoned()` – returns whether the timeout was reached for tasks that were finished but never picked up by a client and removed from the scheduler.
- `_is_defunct()` – returns whether the timeout for a task that is running for too long was reached.

- `_is_kill_requested()` – return whether the task was marked for killing
- `request_kill(reason)` – marks task for killing with `reason`.
- `kill()` – kills the worker process if needed.
- `receive_message(message_payload)` – distributes `message_payload` to following message handlers:
  - `message_executed` – handles `TaskExecuted` message.
  - `message_finished` – handles `TaskFinished` message.
  - `store_reports` – saves reports generated by the lib(rary) command in the task.
- `to_dto()` – converts this `Task` instance into a `TaskResultDto`.

Task States and Lifecycle

The task state is represented by one of these states during its lifecycle:

- Created – the task has been created and put in the incoming task queue of the asynchronous scheduler.
- Queued – the task has been applied to the process pool.
- Executed – a worker reported that task processing have started.
- Finished – a worker has completed the task.

Lifecycle of tasks and how their state changes is pretty complicated and cannot be linearly described in text without being confusing. A better representation is a state machine diagram that can be found in Figure 3.1.

External Facing Data Structures

External facing data structures are used to pass data from and to the REST API. PCS uses data transfer objects[17] for this purpose, so this was also adopted for the scheduler.

The first data structure that is encountered when dealing with the scheduler is used to send data about the task. Its name is `CommandDto` and it contains these members:

- `command_name:str` – contains the name of the lib(rary) command
- `params:Dict[str, Any]` – contains the argument list and values that will be inserted into the function. The intention is to unpack this dictionary into the function call. The side effect of this decision is that positional arguments must be named. If a default value should be used for any argument, it should be completely absent from the dictionary.

[17] Significance of data transfer objects in PCS was explained in Section 2.4 (Use of Type Hints and Data Transfer Objects)

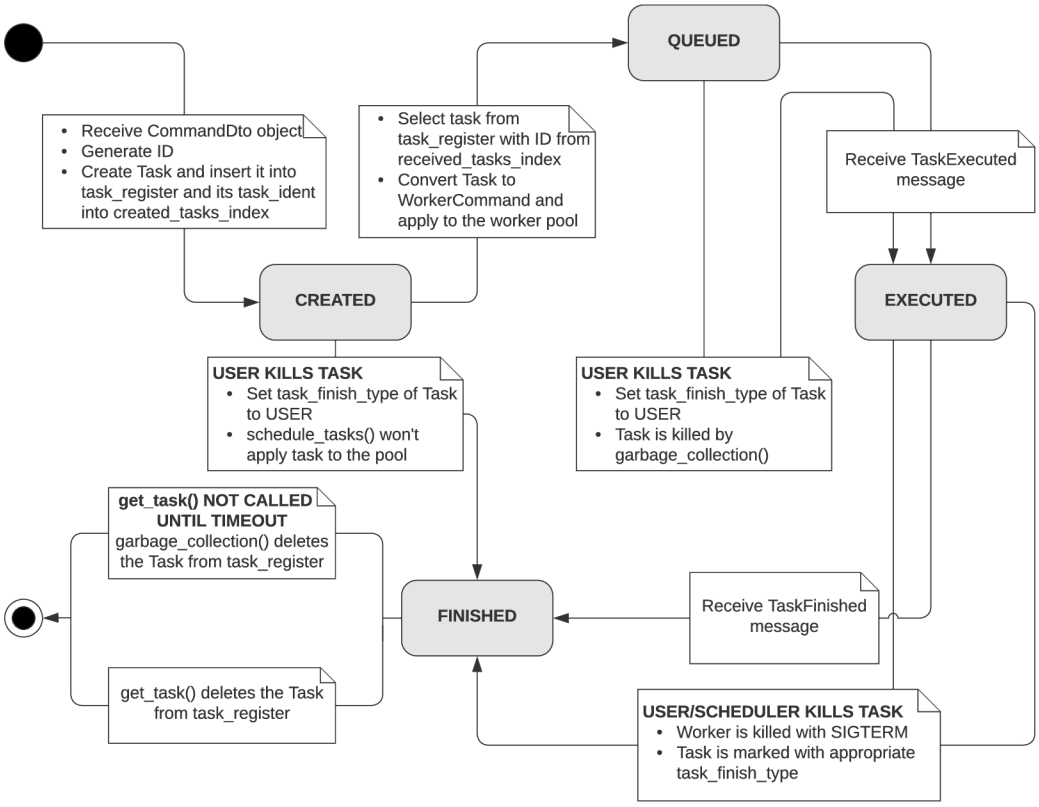

Figure 3.1: **Task lifecycle diagram** This sequence diagram shows the lifecycle of `Task` objects are created and what causes them to change their state.

The `TaskResultDto` structure is used to return the result of the task to the client. It is created by serializing the `Task` object with data fields only relevant to the scheduler left out. These are the fields that are missing:

- `last_message_at` – used by the scheduler to calculate timeouts, it has no relevance to the client
- `worker_pid` – used by the scheduler to send kill signals, this is useless for the client since the PID of the worker is recorded in logs

The scheduler uses enumeration types for constant values used throughout data fields of the `Task` and `TaskResultDto` objects:

- `TaskState` – represents the state of the task. Its values were already explained in Section 3.6. Used for task lifecycle management and informing clients about processing status.
- `TaskFinishType` – further specifies how the lib(rary) command ended.
- `TaskKillReason` – used to mark the task to be killed. At the same time, it contains information about the origin of the kill request. Clients may find this information useful.

`TaskFinishType` is `UNFINISHED` until the task enters the `FINISHED` state. If the command finishes successfully, the finish type is `SUCCESS`. Lib(rary) commands fail by raising a `LibraryError` exception, so in that case, the task has a `FAIL` finish type. In case that a different – unhandled – exception occurs, task has an `UNHANDLED_EXCEPTION` finsh type. If the task was killed, this is indicated by the `KILL` finish type.

The `TaskKillReason` is set to `None` initially. Only if the task is marked to be killed, the kill reason is set. Kill reason is `USER` if the killing was requested through the REST API, `COMPLETION_TIMEOUT` if the task ran for too long and `INTERNAL_MESSAGING_ERROR` if an unknown message was received from a worker. There is no kill reason for an abandoned task since the task does not need to be killed. In that case, the task can be just deleted from the task register.

Internal Data Structures

Internal data structures used to communicate between the scheduler and its worker pool were already described in Section 3.6.

Another data structure is needed to pass information about the task on to the worker. This structure is called `WorkerCommand` and includes the `CommandDto` structure through composition in its command field. The main reason for why this structure exists is that `CommandDto` cannot be modified and the worker needs the task identifier to send messages back to the scheduler. So, another field in `WorkerCommand` is `task_ident`. This data structure is only a data class, not a data transfer object, since it is serialized by the Pickle module.

So far, many classes were presented. To better understand their relationships, the complete visualization of the classes can be found in a class diagram in Figure A.4.

Logging

Initially, scheduler and workers each had their own log files. This is quite impractical for use in production. The PCS daemon already has a log file, so it was desirable to merge worker and scheduler logs into it. The problem is that logging into a file in Python is not supported from multiple processes. Python does not have a cross-platform implementation of locks for multiple processes, so logging into a file from multiple processes can cause data corruption.[23, 22]

The solution presented in the Python documentation suggests using a queue to send logs to a logging process. Workers get a `QueueHandler` object that processes normal logger calls and sends them into a queue. The PCS daemon gets a `QueueListener` that reads logs from the other side of the queue. The logs are written by the logging process, so the data race is eliminated.

Structure and PCSD Integration

The layout of the modules of the asynchronous scheduler was inspired by the proof of concept. The core scheduler files are integrated into PCSD as the `pcs.daemon.async_tasks`

module. Some type definitions were moved into deeper layers of PCS for better reusability and import relationships. These are the changes compared to the proof of concept:

- Code that runs in the worker processes got separated into its own module `worker.py`.
- Mapping definitions to lib(rary) commands is implemented in `command_mapping.py` module. This serves the same purpose and fully replaces the `workloads.py` file.
- External facing data structures were moved into the `pcs.common.types` module where they can better import the data transfer object types and are more easily accessible by external modules.
- Report processor used by the worker reimplementing the abstract `ReportProcessor` class was implemented in the `report_proc.py` module.
- Logging functionality was implemented in the `logging.py` module.

The complete view of what is defined in which module and the import relationships can be found in the package diagram in Figure A.1.

3.7 The REST API Design

The REST API is the entry point for the implementation of this thesis and its design is analogous to the public interface of the asynchronous scheduler described in Section 3.6.

It is assumed that the concepts of REST APIs are well known, so in this section, only the design aspects will be discussed. The set of rules used throughout this section comes from the REST API Design Rulebook by Mark Massé[14].

Resource Naming

With the client expectations described, the API can be designed. For a quick refresher on terminology; resource is any entity placed on the world wide web that may be addressed by a unique identifier, such as a URI[18]. RFC 3986 defines the URI syntax as:
`URI = scheme „://" authority „/" path [ „?" query ] [ „#" fragment ]`.
These names will be used throughout this subsection.

The first set of rules is related to the URI design. The scheme and the authority are already determined by the implementation of the PCS daemon since this API integrates there. For historical reasons, all PCSD services run on port 2224, making the scheme and authority part of the URI: `https://<node-hostname-or-ip>:2224/`.

The next part of the URI is the path. The existing paths in PCSD are:

1. `/ui/*` – the web interface front-end – Tornado redirects these requests to Ruby for the old web interface (PCS 0.10) or serves the TypeScript[19] files of the new web interface (PCS 0.11).

[18] URI – Uniform Resource Identifier, more at: https://www.ietf.org/rfc/rfc3986.txt
[19] TypeScript – strongly typed language built on JavaScript, more at: https://www.typescriptlang.org/

2. `/run_pcs` – running PCS commands remotely.
3. `/remote/*` – special remote actions that have to be handled separately.
4. `/api/*` – Ruby web interface back-end (since PCS 0.11).

Since this is the new API for asynchronous task management, `async` was chosen as the first part of the path to separate it from other uses of PCSD. The next step in the hierarchy is the `task` resource that groups all the task commands. Thanks to this approach, support for other async actions or task actions can be added in the future. The last step in the hierarchy are the controller resources. „A controller resource models a procedural concept. Controller resources are like executable functions, with parameters and return values; inputs and outputs.“[14] Therefore, the last part of the path is always defined as an action or a function call. The expectations that were defined earlier can be translated pretty easily into three API endpoints:

Method	Endpoint	Parameters	Return value
`POST`	**/task/create**	command, parameters	task identifier
`GET`	**/task/result**	task identifier	task state structure
`POST`	**/task/kill**	task identifier	none

Table 3.1: Summary of the proposed API

Methods in this API mostly follow the CRUD[20] equivalent HTTP methods. The methods were chosen based on the type of action:

- `create` – `POST` was used to execute a controller resource.
- `result` – while the whole API was intended to be controllers that should use the `POST` method for execution, it is better to use `GET` here. The intended effect is to look at the state of a controller *resource* which is exactly what the `GET` method should be used for.
- `kill` – again, `POST` was used here to execute a controller. The semantically closest HTTP method is `DELETE`, but it does not fully capture what is happening in the background. Calling this endpoint only sends a request to kill a task. This is an asynchronous request that might not be performed immediately. Also, the task object is not deleted with this request, so the `DELETE` method does not make a lot of sense here.

For all endpoints, with the exception of `result`, the request body is expected in JSON[21] format. These requests must contain the `Content-Type` header set to `application/json`. For the `result` endpoint, the task identifier is a query parameter, since the `GET` request body must be empty. [14]

[20] CRUD – create, retrieve, update, delete
[21] JSON – JavaScript Object Notation, more at: https://tools.ietf.org/html/rfc8259

Error Responses

This API places a strong emphasis on the verbosity of the responses when errors occur. The common response body of an error is as follows:

```
{
  "http_code": 415,
  "http_error": "Unsupported Media Type",
  "error_message": "The 'Content-Type' request header must be set to '
     application/json'.",
}
```

Listing 3.1: Example of an error response

HTTP response code and the corresponding message are included in the response to provide a complete picture just from the response body. The `error_message` gives users of this API a clear idea of what went wrong. Even though the only people who see this are developers, it is much nicer to know why the code is not working. The complete list of common errors for all endpoints is here:

`http_code`	`error_message`
400	Required key '<missing-key>' is missing in request body.
400	Request body contains unexpected keys: '<key-1>', '<key-N>'.
400	Malformed request body.
400	Malformed JSON data.
415	The 'Content-Type' request header must be set to 'application/json'.

Table 3.2: Summary of common errors for all API endpoints

The REST API Design Rulebook [14] recommends using the most suitable HTTP error code for all types of API errors. There were no codes corresponding to the malformed body of the request, so a general client-side error code 400 was used in these cases.

To complete the list of possible errors returned by the REST API, these are endpoint-specific errors:

Endpoints	`http_code`	`error_message`
`status`	400	URL argument '<url-argument>' is missing.
`create` `kill`	404	Task with this identifier does not exist.

Table 3.3: Summary of endpoint-specific errors

Return Types

The rationale behind the `TaskResultDto` object was already described in subsection about external facing data structures of the scheduler in Section 3.6. To complete the design of the API, its JSON representation is provided here:

```
{
  "task_ident": "aafef8c15992413c8d27c156671068bb",
  "command": {
    "command_name": "cluster status",
    "params": {}
  },
  "reports": [
    {
      "severity": {
        "level": "ERROR",
        "force_code": null
      },
      "message": {
        "code": "CRM_MON_ERROR",
        "message": "error running crm_mon, is pacemaker running?\n Could not
            connect to pacemakerd: Connection refused\n crm_mon: Connection
            to cluster failed: Connection refused",
        "payload": {
          "reason": "Could not connect to pacemakerd: Connection refused\
              ncrm_mon: Connection to cluster failed: Connection refused"
        }
      },
      "context": null
    },
  ],
  "state": "FINISHED",
  "task_finish_type": "FAIL",
  "kill_reason": null,
  "result": null
}
```

Listing 3.2: Example of the task status returned by the API. A serialized PCS report is also visible in this example.

`TaskIdentDto` is a very simple structure that only contains the task identifier. The only reason for its existence is that it unifies the deserialization process of request bodies across all request handlers. Here is an example of the JSON representation:

```
{
  "task_ident": "aafef8c15992413c8d27c156671068bb",
}
```

Listing 3.3: Example of the task identifier used in API requests and responses

Chapter 4

Implementation and Testing

The first part of this chapter will focus on the implementation and testing of the proof of concept. In the following part, implementation of the asynchronous scheduler will be explained and often compared to the proof of concept, showcasing the improvements. Implementation of the REST API will be explored next. The last component that was developed was simple client used for testing the REST API along with the asynchronous scheduler. This leads into the testing section where the automated tests are described and some manual testing of the REST API is performed.

4.1 Proof of Concept

The proof of concept was the testing ground for many concepts used for the asynchronous scheduler. It is a much simpler version of the scheduler that allows for easier debugging. This section gives a closer look at its most interesting parts.

Terminology

In the design phase for the asynchronous scheduler, a lot of names were changed to be less confusing. To understand the proof of concept, this is the table that maps these newer names to the old ones:

New name	Old name
`WorkerCommand`	`TaskAssignment`
`TaskExecuted` `TaskFinished`	`StatusUpdate`
`Scheduler.new_task(...)`	`Scheduler.new_assignment(...)`
`Task.message_executed(...)` `Task.message_finished(...)`	`Scheduler.update_handler(...)`
`Task.store_reports(...)`	`Scheduler.report_handler(...)`

Table 4.1: Mapping of names from the implementation to the proof of concept

The Scheduler Module

The main method that periodically executes all scheduler coroutines on the event loop is the `update` method. The name was chosen because it updates the state of tasks, which in turn *update* the status of the scheduler.

```
async def update(self) -> None:
    while True:
        if len(self._assignment_q) > 0:
            asyncio.ensure_future(self._schedule_task())
        if self._receive_messages:
            # Unreliable number of pending messages
            messages_pending = self._worker_communicator.qsize()
            retrieve_limit = messages_pending if messages_pending < 5 else 5
            for i in range(retrieve_limit):
                asyncio.ensure_future(self._get_worker_message())
            await asyncio.sleep(0.1 * retrieve_limit)
            if messages_pending - retrieve_limit > 5:
                asyncio.ensure_future(self.update())
                return
        await asyncio.sleep(0.2)
```

First optimization is using the `_assignment_q`, if it is empty, the task scheduling never runs. There was also an optimization attempt for interprocess communication that would lighten the load on the event loop. The `_receive_messages` and `workers_idle` variables of the `Scheduler` class were used to stop and start checking the messages in the worker message queue. With every `apply_async` call, `receive_messages` was set to `True`. When the task was executed, the `workers_idle` counter would decrease. When the scheduler got a message that the task was finished, the counter was increased and if it reached the original worker count, checking for messages was halted.

The scheduling of message retrieval is quite complicated because it tries to minimize the number of waiting messages without taking too much time on the event loop. First, an estimate of waiting messages is taken by using `qsize()`. The maximum number of messages for one run is capped to five. Then the `sleep` with duration multiplied by number of messages is used to switch to other coroutines – mainly to the recently scheduled message retrievals. If there are more messages in the queue, the `update` method immediately reschedules itself on the loop to retrieve them, removing the pressure from the queue. If not, the infinite loop starts the `update` method again after the control resumes after the `sleep`.

Worker Implementation

For testing purposes, three functions were created, representing situations that can happen in a lib(rary) command:

- „normal“ – sends a `StatusUpdate` message and simulates work by calling the `sleep` function, then exits. This is indicative of a command that finishes successfully.

- „exception“ – immediately raises an exception. The exception is caught by the worker and a report is sent. Tests the error callback function of the process pool.
- „hang“ – after a while, the function enters an infinite loop with the `sleep` command. Can be used to test signals for killing tasks.

The worker process is initialized with the `worker_init` function. This function only prints that the worker started.

The functions are executed by the `task_executor` function that looks up the correct function to execute based on the `TaskMeta` structure that defines command, its parameters and task identifier. Since lib(rary) commands can raise a `LibraryException`, this is simulated by handling `CustomException` that can be raised by the testing functions.

The notion of task numbers from the `make_tasks` function is not carried into a worker, only the task identifier is available here. Any printing that needs to be done here cannot use the

In case of an unknown exception, `apply_async` call has a parameter `error_callback` which is set to a simple function that just prints the exception message for testing purposes.

4.2 Testing the Proof of Concept

Since the proof of concept was the testing ground for all concepts, the most important tests that were performed are presented here. All logs provided here have this line format: time since the script started | PID | message.

Tunable parameters of the proof of concept and their default values are:

- `TASK_SPEC = 6 * („normal", )` – an array of task types which will be applied to the process pool. The task types were described in Section 4.1.
- `SECS_BETWEEN_TASKS = 0.5` – delay between generating tasks.
- `SECS_TASK_RUNNING = 2` – duration of simulated non-asynchronous activity by using `time.sleep()`.
- `WORKER_COUNT = 2` – number of processes in the process pool.
- `REGENERATE_WORKER_AFTER = 2` – how many tasks are processed before the worker process is refreshed.
- `UNHANDLED_EXCEPTION = False` – produces an unhandled exception in „exception“ task type.
- `OPT_ON = True` – toggle for messaging optimizations, `False` turns the optimizations off.

Test 1 – Parallel Execution

This test uses all of the default parameters and demonstrates that the proof of concept works. It can be seen in the output that two tasks are always running. In fact, the workers

are restarting after processing two tasks. Processing of six tasks that sleep for 2 seconds must take at least 12 seconds if they are processed sequentially. Perfect linear scaling would suggest that two workers should be able to do it in 6 seconds. Of course, the perfect linear scaling cannot be achieved because of context switching and multitasking, the testing system is not executing just this program. The last task was finished after 7.3 seconds which is pretty close, even closer if the half-second until the first task was applied to the process pool is subtracted.

```
0.000003 | 230824: Parent starting.
0.001597 | 230826: Worker starting.
0.002059 | 230827: Worker starting.
0.008438 | 230824: Created a process pool.
0.008765 | 230824: Created the event loop.
0.008817 | 230824: Spinning up the event loop.
0.509567 | 230824: Applied 'normal' task 1.
0.627240 | 230826: Task 1 started.
0.813134 | 230824: Task 1 reported status 'EXECUTED'
1.010914 | 230824: Applied 'normal' task 2.
1.131172 | 230827: Task 2 started.
1.318309 | 230824: Task 2 reported status 'EXECUTED'
1.512310 | 230824: Applied 'normal' task 3.
2.013323 | 230824: Applied 'normal' task 4.
2.514157 | 230824: Applied 'normal' task 5.
2.632621 | 230826: Task 3 started.
2.825695 | 230824: Task 1 reported status 'FINISHED'
2.826369 | 230824: Task 3 reported status 'EXECUTED'
3.015207 | 230824: Applied 'normal' task 6.
3.140288 | 230827: Task 4 started.
3.226653 | 230824: Task 2 reported status 'FINISHED'
3.226860 | 230824: Task 4 reported status 'EXECUTED'
4.721836 | 230861: Worker starting.
4.724630 | 230861: Task 5 started.
4.833383 | 230824: Task 3 reported status 'FINISHED'
4.833604 | 230824: Task 5 reported status 'EXECUTED'
5.221804 | 230873: Worker starting.
5.223671 | 230873: Task 6 started.
5.234646 | 230824: Task 4 reported status 'FINISHED'
5.234791 | 230824: Task 6 reported status 'EXECUTED'
6.841831 | 230824: Task 5 reported status 'FINISHED'
7.345442 | 230824: Task 6 reported status 'FINISHED'
```

Test 2 – Single Worker

This test really proves that tasks are running in parallel because the overall execution time is much higher, closer to the theoretical time of tasks running sequentially.

```
0.000002 | 230668: Parent starting.
0.001372 | 230670: Worker starting.
```

```
 0.007896 | 230668: Created a process pool.
 0.008216 | 230668: Created the event loop.
 0.008268 | 230668: Spinning up the event loop.
 0.509378 | 230668: Applied 'normal' task 1.
 0.623358 | 230670: Task 1 started.
 0.815754 | 230668: Task 1 reported status 'EXECUTED'
 1.010885 | 230668: Applied 'normal' task 2.
 1.511925 | 230668: Applied 'normal' task 3.
 2.013692 | 230668: Applied 'normal' task 4.
 2.514392 | 230668: Applied 'normal' task 5.
 2.631205 | 230670: Task 2 started.
 2.728607 | 230668: Task 1 reported status 'FINISHED'
 2.729386 | 230668: Task 2 reported status 'EXECUTED'
 3.015365 | 230668: Applied 'normal' task 6.
 4.720918 | 230716: Worker starting.
 4.723719 | 230716: Task 3 started.
 4.739835 | 230668: Task 2 reported status 'FINISHED'
 4.740734 | 230668: Task 3 reported status 'EXECUTED'
 6.733500 | 230716: Task 4 started.
 8.832632 | 230733: Worker starting.
 8.835424 | 230733: Task 5 started.
10.843512 | 230733: Task 6 started.
12.945223 | 230754: Worker starting.
```

The keen-eyed readers might have noticed that the execution time is, in fact, correct, but why are there no messages from tasks after task 3? There is a simple explanation that also shows why these optimizations are not part of the final implementation.

After the message that the first task finished, the only worker is idle, so scheduler stops listening for messages. This is expected. The next call to `apply_async` is made and scheduler starts to listen for messages. Then the `make_tasks` coroutine outpaces the processing time of tasks in the pool and applies tasks faster than they can finish.

Yes, setting the scheduler to receive messages with every `apply_async` operation might look like it guarantees that every time a new task arrives, the scheduler starts listening for messages. When task 6 is applied to the process pool, this is the last time that `_receive_messages` is set to `True`. Since the message retrieval coroutine is scheduled ahead, message that task 3 was executed makes it through but then the scheduler starts to think that all workers are idle. That stops checking the queue for new messages. There is no way to wake the scheduler, so this optimization is proven to be a bug.

Test 3 – Single Worker without Optimizations

After turning off the message retrieval optimization, this is what was expected in the second test:

```
 0.000003 | 230471: Parent starting.
 0.002307 | 230473: Worker starting.
 0.012417 | 230471: Created a process pool.
```

```
  0.012985 | 230471: Created the event loop.
  0.013072 | 230471: Spinning up the event loop.
  0.514278 | 230471: Applied 'normal' task 1.
  0.636021 | 230473: Task 1 started.
  0.822322 | 230471: Task 1 reported status 'EXECUTED'
  1.015296 | 230471: Applied 'normal' task 2.
  1.516323 | 230471: Applied 'normal' task 3.
  2.018714 | 230471: Applied 'normal' task 4.
  2.520401 | 230471: Applied 'normal' task 5.
  2.640981 | 230473: Task 2 started.
  2.738175 | 230471: Task 1 reported status 'FINISHED'
  2.738954 | 230471: Task 2 reported status 'EXECUTED'
  3.021118 | 230471: Applied 'normal' task 6.
  4.715465 | 230503: Worker starting.
  4.716129 | 230503: Task 3 started.
  4.749840 | 230471: Task 2 reported status 'FINISHED'
  4.750264 | 230471: Task 3 reported status 'EXECUTED'
  6.723414 | 230503: Task 4 started.
  6.759004 | 230471: Task 3 reported status 'FINISHED'
  6.759233 | 230471: Task 4 reported status 'EXECUTED'
  8.770949 | 230471: Task 4 reported status 'FINISHED'
  8.833142 | 230527: Worker starting.
  8.835397 | 230527: Task 5 started.
  9.073350 | 230471: Task 5 reported status 'EXECUTED'
 10.843285 | 230527: Task 6 started.
 10.985839 | 230471: Task 5 reported status 'FINISHED'
 10.986472 | 230471: Task 6 reported status 'EXECUTED'
 12.945179 | 230545: Worker starting.
 12.998439 | 230471: Task 6 reported status 'FINISHED'
```

Test 4 – Testing Signals

Testing with the `SIGKILL` signal has shown that the worker immediately produces an exception because the internal queue used by the `apply_async` function is corrupted, rendering the entire process pool unusable. This is not surprising since `SIGKILL` does not terminate a program gracefully.

```
 11.197599 | 13307: Task 5 reported status 'FINISHED'
Process ForkPoolWorker-1:
Traceback (most recent call last):
  File "/usr/lib64/python3.6/multiprocessing/pool.py", line 125,
  in worker
    put((job, i, result))
  File "/usr/lib64/python3.6/multiprocessing/queues.py", line 347,
  in put
    self._writer.send_bytes(obj)
  File "/usr/lib64/python3.6/multiprocessing/connection.py", line 200,
  in send_bytes
```

```
    self._send_bytes(m[offset:offset + size])
  File "/usr/lib64/python3.6/multiprocessing/connection.py", line 404,
  in _send_bytes
    self._send(header + buf)
  File "/usr/lib64/python3.6/multiprocessing/connection.py", line 368,
  in _send
    n = write(self._handle, buf)
BrokenPipeError: [Errno 32] Broken pipe
```

The results for the `SIGINT` signal were inconsistent. Sometimes, the `KeyboardInterrupt` exception was raised and new worker was not started and sometimes the test script would just freeze.

```
Process ForkPoolWorker-4:
Traceback (most recent call last):
  File "/usr/lib64/python3.6/multiprocessing/process.py", line 258,
  in _bootstrap
    self.run()
  File "/usr/lib64/python3.6/multiprocessing/process.py", line 93,
  in run
    self._target(*self._args, **self._kwargs)
  File "/usr/lib64/python3.6/multiprocessing/pool.py", line 119,
  in worker
    result = (True, func(*args, **kwds))
  File "server.py", line 38, in workload
    sleep(SECS_TASK_RUNNING)
KeyboardInterrupt
  9.7667208 | 22841: Task 5 reported status 'EXECUTED'
 11.7668343 | 22841: Task 5 reported status 'FINISHED'
```

Testing the `SIGTERM` signal:

```
[mpospisi@mpospisi eloop_prototype]$ python3 test_scheduler.py 4
  ...
  0.510530 | 51491: Applied 'hang' task 1.
  0.618578 | 51493: Task 1 started.
  1.011504 | 51491: Applied 'hang' task 2.
  1.019648 | 51494: Task 2 started.
  1.513065 | 51491: Applied 'exception' task 3.
  2.014499 | 51491: Applied 'exception' task 4.
  2.515834 | 51491: Applied 'normal' task 5.
  3.017406 | 51491: Applied 'normal' task 6.
  3.518455 | 51491: Applied 'normal' task 7.
```

At this point, `kill 951493` command was issued from another terminal. A new worker was started immediately:

```
 16.948591 | 51580: Worker starting.
 16.951204 | 51580: Task 3 started.
```

The most common problem during the testing was indicated in the traceback as „[Errno 32] Broken pipe“. The implementation of `multiprocessing.Queue` uses pipes to send data. The abrupt termination of a process probably caused this error.

Test 5 – Testing Error Callback

This test demonstrates that the error callback function is called if tasks raise unhandled exceptions. These exceptions are returned to the parent process where the error callback function is executed.

```
  0.000002 | 55949: Parent starting.
...
 24.060806 | 56024: Worker starting.
 24.061691 | 56024: Task 3 started.
 24.062464 | 55949: Unhandled exception: 'Exception('this is unexpected',)'
```

In all tests, closing the daemonic test script with `Ctrl+C` propagates the resulting `SIGINT` signal to the worker processes of the process pool. This is not desirable because it produces exceptions in the workers and prints undesired traceback to the console.

4.3 The Asynchronous Scheduler

This section focuses on the implementation of the most vital parts of the asynchronous scheduler. Not all implementation details will be covered since the design section has already covered a lot of detail. Parallels will be drawn to the proof of concept throughout this section to show where improvements were made.

Running Scheduler on the Event Loop

The asynchronous scheduler uses its `perform_actions` method to perform all its duties. This is similar to the scheduler in the proof of concept and its `update` method. It was shown that optimizing the `update` method was not that easy and resulted in a bug. The `perform_actions` method is much simpler. It just calls the coroutines for each scheduler responsibility and waits for each one to complete.

The scheduling of the `perform_actions` method on the event loop should be handled by the Tornado application, not by the scheduler itself. This is a more flexible approach compared to the proof of concept. The scheduler has a single method that takes care of all its responsibilities, and application is responsible for scheduling it according to its own needs.

The scheduling is implemented in the `run_scheduler` coroutine in the `daemon/run.py` file. It contains an infinite loop that awaits the `perform_actions` method and then returns control to the event loop by calling the asynchronous sleep function. Until the event loop is stopped, the scheduler executes this infinite loop. Asynchronous sleep is used to ensure that the `perform_actions` method leaves enough time for other coroutines on the event loop even when `perform_actions` runs longer than its scheduling interval. More information on how this integrates into PCSD will be mentioned in Section 4.4 (REST API Integration).

Worker Implementation

The worker functions used in the scheduler are not that different from the proof of concept at first glance. One of the differences is that all worker-related code was moved to its own module `worker.py`. Another difference is the method of passing queues that was presented earlier. But the biggest difference is that the `task_executor` function now deals with real lib(rary) commands.

Commands are launched from a dictionary of command names and functions imported from the library. This dictionary is located in its own file `comand_mapping.py`. This was done in preparation for integration of a large number of functions in the future. The commands are then called by the reference from this command map. Thanks to the unified interface of library functions, after the first argument, the `LibraryEnvironment`, all command parameters from the `CommandDto` structure can be simply unpacked into the argument list.

```python
def task_executor(task: WorkerCommand) -> None:
    ...
    # TaskExecuted message
    env = LibraryEnvironment(
        logger,
        WorkerReportProcessor(worker_com, task.task_ident),
    )

    task_retval = None
    try:
        # Call the command from command map and unpack the parameters
        task_retval = command_map[task.command.command_name](
            env, **task.command.params
        )
    except LibraryError as e:
        for report in e.args:
                # PCS report message
        # TaskFinished message, TaskFinishType.FAIL
        pause_worker()
        return
    except Exception:
        # TaskFinished message, TaskFinishType.UNHANDLED_EXCEPTION
        pause_worker()
        return
    worker_com.put(
        # TaskFinished message, TaskFinishType.SUCCESS
    )
    pause_worker()
```

Listing 1: Simplified implementation of the `task_executor` function that shows launching the lib(rary) commands, exception handling and report distribution. Initialization and logging calls were omitted.

The `task_executor` function now handles all exceptions raised by the lib(rary) commands. The most frequent exceptions is the `LibraryError` exceptions raised primarily by validations in the lib(rary) commands. An earlier concept of report distribution in PCS was that the reports were stored in the `args` variable of this exception. This case is handled by inspecting the exception and sending these reports to the worker report processor.

Speaking of the worker report processor, it handles the transfer of reports through the queue to the asynchronous scheduler, where the client can retrieve them throiught the REST API. The worker report processor is implemented in the `report_proc.py` module. Its report distribution method uses the blocking `put` function to send messages to the scheduler since the worker is not an asynchronous environment.

All other exceptions are also handled by the `task_executor` function. Using the error callback function, as the proof of concept did, had no observable benefits. On the other hand, handling all exceptions in the worker had – all of the possible scenarios can be reported with the `TaskFinished` message. This simplifies the scheduler since it does not have to provide an interface that an error callback can use to report unhandled exceptions.

Interprocess Communication

The only significant change from the proof of concept is how the communication queue is passed to the worker. It was ineffective to pass the same reference with every task. A much better solution is to pass the queue into the `worker_init` function. The problem is that the `task_executor` function where the queue is predominantly used has its own scope. Although global variables are almost never a good solution to any problem, an argument can be made for this use case. This is a worker process, so even its global namespace is isolated from the application.

Unfortunately, the integration tests showed that these assumptions were partially incorrect. More can be found in Section 4.6 (Integration Testing).

Task Killing and Garbage Collection

Task killing must be implemented in two stages. The reason is that tasks in the `QUEUED` state cannot be killed. When the task is submitted for processing to the process pool, there is no way to cancel this operation.

The scheduler marks the task for killing by setting the kill reason in the first stage. This can be done from any place, such as the method for killing tasks when clients asks to kill the task. For tasks that are running too long, the garbage collector sets a kill reason. A special case is the timeout when the client does not pick up the task result, garbage collector just deletes that task from the task register.

The `SIGTERM` signal is sent in the second stage when the garbage collector calls the `kill` method of the task. It only sends the signal if the task is in the `EXECUTED` state. The method

is responsible for setting the finish type and changing the status to `FINISHED` on tasks in any state. This is done to ensure that all tasks destroyed by the garbage collector end up in the `FINISHED` state. Tasks react to being killed differently in each state:

- `CREATED` – the `schedule_tasks` method checks for `kill_reason` and does not apply the task to the process pool. The garbage collector then changes state to `FINISHED` by calling the kill method of the task.
- `QUEUED` – the task is only marked to be killed later by the garbage collector.
- `EXECUTED` – the `SIGTERM` signal is sent to the process where the task is running and the task state is changed to `FINISHED`.
- `FINISHED` – nothing changes.

Data Races and Race Conditions

Due to the asynchronous input and output processing model, data races cannot occur in the scheduler. Data races can only happen when two threads are working with the same data and at least one of them is writing. [4] Only one coroutine runs on the event loop at any given time. The scheduler therefore internally does not need any locks to function.

However, race conditions can still occur between the asynchronous code and the process pool. There is a delay between marking the task for killing and sending the `SIGTERM` signal. This can cause a race condition if the task finishes after it was marked to be killed and before the signal is sent. In that case, the worker would receive a new task and then it would be killed shortly. A completely different task would be killed. This is fixed by pausing the worker process with the `SIGSTOP` signal when it sends the `TaskFinished` message. The worker resumes with the help of a `SIGCONT` signal only when the `TaskFinished` message handler sets the task state to `FINISHED`. At this point, the kill signal would not be sent, so the race condition is eliminated.

Logging

The starting point of the merged logs is the `log.py` module of the Python daemon. The PCSD logger already integrates logs from the Ruby daemon. The merged log from both daemons is written into `/var/log/pcsd/pcsd.log`.

The PCSD log format is:
`<level[0]>, [<date-time> #<pcsd-group-id>] <level> - :  <message>`

Every line starts with the first letter of the severity level. The next part is date and time and PCSD group ID which is out of the scope of this text and will not be needed. The level is repeated in full and the actual log message follows.

All of this translates to:
`I, [2022-05-14T18:43:19.762 #00000] INFO - :  Binding socket for address '*' and port '2224'`

The log records from the scheduler can be placed into this log without any problem. Even though the modules are different, the Logger module returns references to the same logger

if the same name is used in one process. The Python daemon is running only in one process. Scheduler logging is solved by importing the logger object from the `log.py` module.

Worker processes use the concept of passing log records through a queue described in the logging subsection of Section 3.6. The function that sets up logging for the worker is implemented in the `logging.py` module of the asynchronous scheduler. The queue for report distribution is not passed between worker initializer and task executor with a global variable. Instead, a logger is set up in the worker initializer and is later retrieved by requesting the logger of the same name through the Logging module.

The receiving end of the log records have to somehow integrate into the existing PCSD logger. This is solved during scheduler initialization by this function:

```
1 def _init_worker_logging(self) -> handlers.QueueListener:
2     q_listener = handlers.QueueListener(
3         self._logging_q, *self._logger.handlers
4     )
5     q_listener.start()
6     return q_listener
```

The `QueueListener` class requires handlers which are responsible for writing the logs. While this is not documented, the `Logger` class contains a `handlers` instance variable. Since there are no leading underscores, this is interface can be considered stable. The function above steals the handlers from the PCSD logger and uses it to write to the PCSD log. This will not cause data corruption since the logger module is thread-safe. [17]

An important addition to worker logs is the worker process identifier. Without it, identifying the source of the messages would be impossible in a merged log. For this reason, all worker logs start with: „`Worker#<process-ident>:` “ where `<process-ident>` is the process identifier.

Termination

The proof of concept produced unwanted tracebacks at exit. This was caused by the propagation of signals from parent process to the process pool. The tracebacks were caused by an unhandled `KeyboardException` in the worker. This was solved by implementing an empty signal handler in the worker registered to handle the `SIGINT` signal.

The `Scheduler` also implements a method for a clean exit called `terminate_nowait`. It is used to free the resources used by the scheduler. Its first action is to empty the logging queue used by the worker so that all logs are written and it can be destroyed. Then the process pool is terminated which kills all of the processes and destroys all of its other resources. After that, the scheduler is correctly terminated.

Settings

Just as the scheduler in the proof of concept can be configured, the same applies to the final implementation of the asynchronous scheduler in PCS. These settings are stored in a configuration file `settings.py.in` together with the default settings for PCS:

- `async_api_scheduler_interval_ms = 100` – this number specifies how often is the `Scheduler.perform_actions` method scheduled on the event loop. The default value is 100 ms to allow the event loop to process other coroutines and to be fairly lightweight on resources.
- `worker_count = multiprocessing.cpu_count()` – number of processes in the process pool. Defaults to number of available cores to prevent lib(rary) commands competing for resources.
- `worker_task_limit = 5` – maximum number of tasks for one worker process.
- `task_unresponsive_timeout_seconds = 60 * 60` – timeout for killing an unresponsive task, the task will be killed if no message is delivered from the worker for 60 minutes. Commands that need longer time to finish can be changed to send more messages.
- `task_abandoned_timeout_seconds = 1 * 60` – timeout for task deletion. Task is deleted when client does not query its status for one minute after it has been completed.

4.4 The REST API

The REST API has been implemented within the existing Tornado application which is the new PCS daemon. The code that handles the creation of the HTTPS server was already part of PCSD. Apart from server initialization, Tornado applications are just pairs of URIs[1] and `RequestHandlers` that communicate with clients.

The Request Handlers

The API consists of three request handlers that inherit some common functionality from `BaseAPIHandler`. The `BaseAPIHandler` class implements the request body preprocessing that deserializes the JSON[2] request body of the requests that contain it. Deserialization is handled by the JSON package[3] from the Python standard library. The resulting dictionary is saved to a new instance variable `json` in the request. It also implements error handling by converting any exceptions from lower layers to the `APIError` exception. How that exception is processed will be described later in this section.

[1] URI – Uniform Resource Identifier, more at: https://www.ietf.org/rfc/rfc3986.txt
[2] JSON – JavaScript Object Notation, more at: https://tools.ietf.org/html/rfc8259
[3] The JSON package in Python standard library, more at: https://docs.python.org/3.6/library/json.html

The three API endpoints defined in the design chapter, Section 3.7 are represented by these three request handlers that implement a single method named after the HTTP request method:

- `NewTaskHandler` – handles the `POST /task/create` REST API endpoint by calling the `new_task` method of the asynchronous scheduler. The request body is a JSON representation of the `CommandDto` data transfer object. This object specifies the lib(rary) command to execute and its arguments. A task identifier is returned to the client as a `TaskIdentDto` object serialized to JSON. The HTTP response code 201 signifies that a task resource was created. No errors can occur during handling of this request.
- `TaskInfoHandler` – handles the `GET /task/result` REST API endpoint by calling the `get_task` method of the scheduler. Its only parameter is the `task_ident` string, which is read from the query part of the URI. The `TaskResultDto` data transfer object is returned to the client. The following exceptions can be raised in this handler:
  - If the query argument `task_ident` is not present in the query section of the URI, Tornado raises the `MissingArgumentError` exception.
  - If the task identifier does not exist in the task register, the scheduler raises the `TaskNotFoundError` exception.
- `KillTaskHandler` – handles the `POST /task/kill` REST API endpoint by calling the `kill_task` method of the asynchronous scheduler. The request body is a JSON representation of the `TaskIdentDto` data transfer object. Empty HTTP response with the code 202 indicating a pending asynchronous action is returned to the client. If the task identifier does not exist in the task register, the scheduler raises the `TaskNotFoundError` exception.

Error Handling

Generally, any error in a `RequestHandler` class from the Tornado library is handled by raising the `HTTPError` exception. This exception is then handled by the `write_error` method of the `RequestHandler` class. This method creates and sends the response with the error description and closes the connection.

To implement the error responses of this API, a custom `APIError` exception that inherits from `HTTPError` was defined. This exception renames the `reason` field to `http_error` and adds the `error_msg` field. These fields are used to construct the error responses in the `write_error` method of the `BaseAPIHandler` class.

The `BaseAPIHandler` is where the bulk of the error handling is implemented. It handles common errors described in Table 3.2. Various errors are being handled at various stages of request processing. The first stage of error handling is the `prepare` method. This method is only used for the `POST` requests since `GET` requests do not have a body. First, the Content-Type HTTP header is checked if it contains the `application/json` application type. In the next step, the JSON body of the request is parsed and converted to a dictionary by the JSON package. In case of an error, the exception from the JSON package is converted to an `APIError` exception.

The `BaseAPIHandler` also implements a method for JSON deserialization with Dacite which is an important feature of the new architecture introduced in Section 2.4 (Use of Type Hints and Data Transfer Objects). This method is called `_from_dict_exc_handled` and it converts exceptions raised by Dacite to `APIError` exceptions. This method accepts a dictionary from the `json` instance variable of the request handler and returns a data transfer object of a requested type if there were no conversion errors.

Security

The REST API and the scheduler were designed with some security in mind. The task identifier was chosen to be UUID[4] version 4 which is completely random. This provides some layer of passive security against a malicious actor attempting to guess the task identifier.

Notably missing is also any authentication mechanism for the API. It is likely that an authentication mechanism used by some other service of the Python daemon will be reused. For example, one of these methods is token-based authentication. To simplify debugging and testing, no authorization mechanism was implemented for now.

PCS also has an authorization mechanism for commands. Not all users are authorized to run all commands. This is missing from the asynchronous scheduler because a fully-featured implementation was never a goal. Before the scheduler is released as part of PCS, this functionality must be added.

PCSD Integration

PCSD initialization code is located in the `pcs/daemon/run.py` file. The most important function `configure_app` creates tuples of URIs and Tornado handlers. These are used to create a Tornado `Application` object that captures all the server and application configuration. The Scheduler instance is passed as an argument to `configure_app` so that it can pair the REST API handlers with the public interface of the `Scheduler` class. These pairings are determined by the `async_api.get_routes` function that further propagates the scheduler instance to the handlers.

A Tornado application is started by creating an event loop and calling its `start` method. Before the start method is called, all coroutines needed for the daemon are scheduled to be run on the event loop. One of these coroutines is the `run_scheduler` coroutine that schedules periodic execution of the scheduler responsibilities. All of these coroutines are executed one-by-one once the event loop goes live.

4.5 Client

For debugging and testing purposes, a simple client was developed. This client was never designed to become a user-facing component. Some thought process went into its architecture, but not to the same degree as with the scheduler or the REST API.

[4] UUID – Universally Unique Identifier, defined by RFC4122: https://datatracker.ietf.org/doc/html/rfc4122

The client is implemented in the `pcs/async_client.py` file. The client is built mainly with the help of the Pycurl[5] library, which is the Python interface for the Libcurl[6] client-side library for network communication. This library was chosen because PCS already uses it.

Usage

The client does not have the same argument parsing capabilities as PCS. The command names are the same as for PCS but arguments and switches are handled differently. There is no validation, apart from checking that the arguments match any supported command. The client accepts the same arguments as the corresponding lib(rary) command function. The argument values have to be inputted in JSON since the values may not be simple data types. More about usage is described in the manual in Section B.2.

Running Commands

The client was developed to connect to the local REST API. When the command is parsed, a `POST` request is created and sent to the `task/new` endpoint. When the task identifier is received, the client periodically queries the `task/status` endpoint. Reports are saved after every query, and new ones are printed on the standard output. When the task reaches the `FINISHED` state, periodic queries are halted. A short status of the task is printed, informing about the task identifier, the finish type, and kill reason. Task identifier is particularly useful for inspecting logs.

Killing tasks

If a task is not completed, the client has the ability to kill it. This can be achieved by pressing `Ctrl+C` when the command is running. The resulting `SIGINT` signal is handled by a signal handler. For this reason, the reports mentioned above and the task identifier are saved in global variables. The task identifier is saved because a request to kill the running task needs to be formed inside the signal handler. Additionally, the task identifier is truthy only when the task is not finished, so it is used to produce an error when there is no task to kill. Reports are saved because the client needs to remember which reports were already printed. The reports delivered during the time it takes the scheduler to kill the task are printed from the signal handler.

Error Handling

All responses from the REST API are filtered through the `_handle_api_error` function. If the HTTP response code does not indicate success, `error_message` in the API common error format is printed on the output. If this message is not found in the response, the response body is printed in full.

[5]Pycurl library, more at http://pycurl.io/
[6]Libcurl URL transfer library, more at https://curl.se/libcurl/

4.6 Testing the Asynchronous Scheduler and REST API

The first half of this chapter is dedicated to the implementation of automated testing. First, the PCS test suite and its limitations are introduced. Next, after a short aside on coroutine testing, the facilities used for automated testing of the scheduler are explored. Automated tests also presented some challenges that needed to be overcome, so, the solutions are presented. The remainder of the chapter is dedicated to manual testing of the scheduler that confirmed its usability in the real world.

Automated Testing

The thesis specification requires the creation of a set of unit and integration tests. These tests were built into the PCS test suite. The PCS test suite consists of two big groups:

- Tier 0 tests – unit and integration tests built around the Unittest module[7] from the Python standard library. These tests cannot create any processes or make any external calls. In other words, they should be environment independent. These tests make extensive use of mocking, often using special mock classes like `EnvAssistant` that replaces the `LibraryEnvironment` object in tests.
- Tier 1 tests – integration tests that launch PCS and evaluate its behavior. A large number of tests cannot be implemented here since they might require a live cluster to work. Mocking is not possible, since PCS is executed in a separate process.

All unit tests are placed in the `pcs_test/tier0/daemon/async_tasks` folder. The exception is the `test_integration.py` file that contains integration tests. More than 70 unit and integration tests were implemented. No tier 1 tests were implemented, since these tests would require a running cluster to use the library functions. Even if the daemon was running, timeouts, for example, could not be tested. These tests require mocking of the function calls that provide the current time.

Returning to tier 0 tests, the most interesting part is that they mostly test coroutines. This led to the creation of the modified `TestCase` class from the Unittest module and `AsyncTestCase` class from Tornado. Neither Asyncio nor Unittest provide built-in support for coroutine testing. Tornado decorator `tornado.testing.gen_test` had to be used to test coroutines.

The `TestCase` and `AsyncTestCase` classes were extended by the scheduler mock object which is integral to the tier 0 tests. These classes became `SchedulerBaseTestCase` and `SchedulerAsyncTestCase`. These base testing classes can be used in conjunction with classes that provide mock objects for other calls or provide other extra functionality. The mixin design pattern was utilized to reduce the boilerplate code in the form of mock initialization in every test case. Python allows multiple inheritance, so even more than one mixin can be used. The implemented mixin classes are:

- `MockDateTimeNowMixin` – mocks the calls to query the current date and time. Useful for testing timeouts.

[7] Unittest module from the Python standard library, more at: https://docs.python.org/3.6/library/unittest.html

- `MockOsKillMixin` – mocks the call of the `os.kill` function. It is unacceptable to let the tests kill a random running processes.
- `AssertTaskStatesMixin` – adds the function that asserts the number of tasks in each possible state. Good for evaluating task state changes.

Scheduler Modifications for Testing

The aforementioned scheduler mock is in the `SchedulerTestWrapper` class. This is a wrapper for all scheduler mock objects used throughout tier 0 tests. It replaces these facilities of the `Scheduler` class:

- `_proc_pool_manager` – the manager spawns a child process that holds the shared resources. [18] This is not acceptable in tier 0 tests. It is replaced by a mock object configured to return patched queues.
- `_worker_q` – this queue has to be patched, since the process manager cannot be used. The queue is patched with a queue from the Multiprocessing package.
- `_logging_q` – same as above.
- `_logger` – the original logger instance has to be patched by a logger that does not write logs to the disk.

Patching the queues is not only necessary; it also solves a unique problem that arises from tests running in parallel. The PCS test suite supports parallelization by dividing tests into buckets and passing each bucket to a different process. The queue for communication between the workers and the scheduler is a global variable of the `worker.py` module. A global variable is the only way to pass the queue from the task initializer to the task executor. This is not a problem in a worker process, but when the module is imported for testing, it becomes a global variable for all tests. Therefore, every test has to patch the global queue with a unique instance. This is the most important function of the test wrapper.

Integration Tests

The asynchronous scheduler had to be modified to be usable in the test suite. The biggest problem was that non-blocking methods of the queue were not reliable enough to guarantee delivery of all messages. Queues are also much slower than tests which call `put` and `get_nowait` in rapid succession. Therefore, the tests would not be able to get the message since it was still in transit. This resulted in missed messages and tests that did not work reliably.

Message passing was not repeatable, sometimes the messages were not properly delivered. This had to be fixed by a change to the `perform_actions` scheduler method that now returns the number of messages received. The `SchedulerForIntegrationTests` class adds an active waiting mechanism that runs the message retrieval coroutine until the expected number of messages was received.

```python
async def perform_actions(self, message_count):
    received = await self.scheduler.perform_actions()
    while received < message_count:
        received += await self.scheduler._receive_messages()
```

Other modifications of the scheduler in the aforementioned class are related to simulating actions of the worker processes. For convenience, a function for generating multiple tasks was also added.

Manual Testing

Manual testing was an important part of the development process. These tests confirm that everything works in the real world. These tests were performed with the help of the asynchronous client or by calling `curl`[8] from the console. The results of these tests are provided in the following subsections.

Test 1 – Cluster Status

This test shows that the `cluster status` command works the same as its PCS counterpart, which does not use the new daemon API. The only addition to the output is diagnostic information about the task created by the scheduler.

```
[root@fedora36 pcs]# pcs_async cluster status
Cluster name: cluster1
Cluster Summary:
  * Stack: corosync
  * Current DC: fedora36 (version 2.1.3-0.1.rc1.fc36-a988afd4e7) -
partition with quorum
  * Last updated: Sat May 14 19:12:25 2022
  * Last change: Sat May 14 19:12:07 2022 by root via cibadmin on fedora36
  * 1 node configured
  * 1 resource instance configured

Node List:
  * Online: [ fedora36 ]

Full List of Resources:
  * dummy1 (ocf::heartbeat:Dummy): Started fedora36

Daemon Status:
  corosync: active/disabled
  pacemaker: active/disabled
  pcsd: active/enabled
```

[8] Curl – command line tool for transferring data with URLs, more at: https://curl.se/

```
--------------------------
Task ident: 466147ac523d448cbd02e4e83ad11015
Task finish type: SUCCESS
Task kill reason: None
```

This is the same output as printed by PCS:

```
[root@fedora36 pcs]# pcs cluster status
Cluster name: cluster1
Cluster Summary:
  * Stack: corosync
  * Current DC: fedora36 (version 2.1.3-0.1.rc1.fc36-a988afd4e7) -
partition with quorum
  * Last updated: Sat May 14 19:12:28 2022
  * Last change: Sat May 14 19:12:07 2022 by root via cibadmin on fedora36
  * 1 node configured
  * 1 resource instance configured

Node List:
  * Online: [ fedora36 ]

Full List of Resources:
  * dummy1 (ocf::heartbeat:Dummy): Started fedora36

Daemon Status:
  corosync: active/disabled
  pacemaker: active/disabled
  pcsd: active/enabled
```

Test 2 – Enable a Resource and Kill the Task

This test demonstrates that the `resource enable` command works and that it can be killed from the console. The task output shows that the task was killed by the scheduler on the basis of the request from the user.

```
[root@fedora36 pcs]# pcs resource disable dummy1
[root@fedora36 pcs]# pcs_async resource enable \
--resource_or_tag_ids='["dummy1"]' --wait=null
Waiting for the cluster to apply configuration changes...
^CTask kill request sent...

--------------------------
Task ident: 316210010b8e44bbb6eaefd08b472479
Task finish type: KILL
Task kill reason: USER
```

Just before the task information, the asynchronous client printed that the task kill request was sent following the `^C` character sequence which indicates that `Ctrl+C` was pressed.

Before that, a report can be seen that was sent before the command was finished. This indicates that the reports are being delivered to the client while the command is running.

Test 3 – Parallel Execution of Two Commands

In this test, the two previously demonstrated commands were executed in parallel. Disabling a resource was much faster than retrieving the status of the cluster, so the `fg` command had no process to switch to. The cluster status shows that the `dummy1` resource is disabled, which is correct since these commands were launched in parallel. There was no chance that the resource enable action would finish in time to affect the cluster status.

```
[root@fedora36 pcs]# pcs resource disable dummy1
[root@fedora36 pcs]# pcs_async resource enable \
--resource_or_tag_ids='["dummy1"]' \
& pcs_async cluster status && fg
[1] 55092

--------------------------
Task ident: 783fa41304654680b671d31d678ce89c
Task finish type: SUCCESS
Task kill reason: None

Cluster name: cluster1
Cluster Summary:
  * Stack: corosync
  * Current DC: fedora36 (version 2.1.3-0.1.rc1.fc36-a988afd4e7) -
partition with quorum
  * Last updated: Sat May 14 19:37:46 2022
  * Last change: Sat May 14 19:36:45 2022 by root via cibadmin on fedora36
  * 1 node configured
  * 1 resource instance configured (1 DISABLED)

Node List:
  * Online: [ fedora36 ]

Full List of Resources:
  * dummy1 (ocf::heartbeat:Dummy): Stopped (disabled)

Daemon Status:
  corosync: active/disabled
  pacemaker: active/disabled
  pcsd: active/enabled

--------------------------
Task ident: b72769cf8a3f4ad791031282f93d783b
Task finish type: SUCCESS
Task kill reason: None
```

```
[1]+ Done pcs_async resource enable --resource_or_tag_ids='["dummy1"]'
-bash: fg: current: no such job
```

To see if the tasks were really executed in parallel, the PCSD log can be examined. Much of the debug output generated by the library functions was omitted for clarity. The log clearly shows two tasks being executed at the same time. It can be said that scheduler is performing very well since creating the tasks and generating the UUID took less than a millisecond.

```
I, [2022-05-14T19:37:46.800 #00000] INFO -- : 201 POST /async/task/create
(127.0.0.1) 0.73ms
I, [2022-05-14T19:37:46.801 #00000] INFO -- : 201 POST /async/task/create
(127.0.0.1) 0.58ms
I, [2022-05-14T19:37:46.809 #00000] INFO -- : 200 GET /async/task/result?
task_ident=783fa41304654680b671d31d678ce89c (127.0.0.1) 0.47ms
I, [2022-05-14T19:37:46.811 #00000] INFO -- : 200 GET /async/task/result?
task_ident=b72769cf8a3f4ad791031282f93d783b (127.0.0.1) 0.51ms
I, [2022-05-14T19:37:46.873 #00000] INFO -- : Worker#54207:
Task b72769cf8a3f4ad791031282f93d783b executed.
I, [2022-05-14T19:37:46.873 #00000] INFO -- : Worker#54797:
Task 783fa41304654680b671d31d678ce89c executed.
I, [2022-05-14T19:37:46.975 #00000] INFO -- : Worker#54797:
Task 783fa41304654680b671d31d678ce89c finished.
D, [2022-05-14T19:37:46.976 #00000] DEBUG -- : Worker#54797:
Pausing until the scheduler updates status of this task.
I, [2022-05-14T19:37:47.065 #00000] INFO -- : Worker#54207:
Task b72769cf8a3f4ad791031282f93d783b finished.
D, [2022-05-14T19:37:47.065 #00000] DEBUG -- : Worker#54207:
Pausing until the scheduler acknowledges the TaskFinished message.
I, [2022-05-14T19:37:47.095 #00000] INFO -- : Worker#55122: Initialized.
I, [2022-05-14T19:37:47.125 #00000] INFO -- : 200 GET /async/task/result?
task_ident=b72769cf8a3f4ad791031282f93d783b (127.0.0.1) 3.81ms
I, [2022-05-14T19:37:47.133 #00000] INFO -- : 200 GET /async/task/result?
task_ident=783fa41304654680b671d31d678ce89c (127.0.0.1) 11.84ms
D, [2022-05-14T19:37:47.188 #00000] DEBUG -- : Acknowledge TaskFinished
message from worker#54207. The worker can continue.
```

It is worth mentioning that the timestamps for log records from workers are not indicative of when the log record was generated. The timestamps are generated by the logger in PCSD, so the timestamp shows when the log records were delivered through the logging queue. The tasks did not finish at the same time, but were close enough to have the messages picked up by the same run of the `_receive_messages` coroutine.

Test 4 – Error Handling of the REST API

These tests demonstrate that the API error handling is working as expected. The Curl command-line utility was used to perform these improper requests. The `error_message` captures the purpose of every test.

```
[root@fedora36 pcs]# curl --insecure --request GET \
https://localhost:2224/async/task/result?task_ident=not-an-identifier
{
    "http_code": 404,
    "http_error": "Not Found",
    "error_message": "Task with this identifier does not exist."
}

[root@fedora36 pcs]# curl --insecure --request GET \
https://localhost:2224/async/task/result?id=id
{
    "http_code": 400,
    "http_error": "Bad Request",
    "error_message": "URL argument 'task_ident' is missing."
}

[root@fedora36 pcs]# curl --insecure --request POST \
--header "Content-Type: application/json" \
--data '"command_name":"status" "params":' \
https://localhost:2224/async/task/create
{
    "http_code": 400,
    "http_error": "Bad Request",
    "error_message": "Malformed JSON data."
}

[root@fedora36 pcs]# curl --insecure --request POST \
--data '{"command_name":"status", "params":{}}' \
https://localhost:2224/async/task/create
{
    "http_code": 415,
    "http_error": "Unsupported Media Type",
    "error_message": "The 'Content-Type' request header
       must be set to 'application/json'."
}

[root@fedora36 pcs]# curl --insecure --request POST \
--header "Content-Type: application/json" \
--data '{"command_name":"cluster status", "params": {}, "unexpected": ""}' \
https://localhost:2224/async/task/create
{
    "http_code": 400,
    "http_error": "Bad Request",
    "error_message": "Request body contains unexpected keys: 'unexpected'."
}
```

```
[root@fedora36 pcs]# curl --insecure --request POST \
--header "Content-Type: application/json" --data '{}' \
https://localhost:2224/async/task/create
{
    "http_code": 400,
    "http_error": "Bad Request",
    "error_message": "Required key 'command_name' is missing
        in request body."
}
```

Chapter 5

Conclusion

The main goal of this thesis was to design and implement a minimal viable solution that allowed launching non-asynchronous functions from the asynchronous environment of the new REST API for task management. This goal has been achieved.

Before the design phase could start, the general concept of the asynchronous input and output programming model were explained. These concepts were expanded on by explaining concepts specific to the AsyncIO library[1] from the Python standard library.

With these concepts in mind, the asynchronous scheduler with an interface for creating and managing tasks was designed. Thanks to the inclusion of a processing pool, it is possible to execute more commands at once.

The public interface of the asynchronous scheduler was then used to model the REST API. The REST API was implemented as a part of the existing Tornado application which is the new Python PCS daemon.

A set of more than 70 unit and integration tests was created. These tests are part of the PCS testing suite used in continuous integration pipelines. Manual testing was also performed with a minimal client implementation, which is part of this thesis. One of these tests successfully demonstrated that two PCS commands were running simultaneously.

The incremental results of this thesis were shared with the PCS development team. The feedback from the team ultimately shaped the final result. Thanks to this feedback, the results were up to expectations, and the PCS development team is already adding authentication and authorization capabilities to the REST API and task executor at this time.

In the near future, the author plans to look into using Futures for handling timeouts in the scheduler. Looking into a more distant future, performance testing needs to be conducted that will almost certainly result in further optimization of scheduling coroutines. An interesting project could also be an attempt to implement multiprocess safe queues in Python with both asynchronous and blocking interfaces. This could possibly be achieved with sockets. Sockets have native asynchronous handling methods in AsyncIO and also support traditional blocking functions.

[1] AsyncIO library, more at: https://docs.python.org/3.6/library/asyncio.html

Bibliography

[1] Arlow, J. and Neustadt, I. *UML2 a unifikovaný proces vývoje aplikací*. 1st ed. Brno, CZ: Computer Press, a.s., 2007. ISBN 978-80-251-1503-9.

[2] ClusterLabs maintainers. *ClusterLabs > Home* [online]. 2011. 2022-01-31 [cit. 2022-04-26]. Available at: https://clusterlabs.org/.

[3] Fowler, M. *Data Transfer Object* [online]. 2003-01 [cit. 2022-04-27]. Available at: https://martinfowler.com/eaaCatalog/dataTransferObject.html.

[4] Grimm, R. *Race Conditions versus Data Races* [online]. 2017. 2017-05-21 [cit. 2022-05-09]. Available at: https://www.modernescpp.com/index.php/race-condition-versus-data-race.

[5] Guido van Rossum, Lehtosalo, J. and Langa, L. *PEP 484 – Type Hints* [online]. 2014. 2022-04-20 [cit. 2022-04-27]. Available at: https://peps.python.org/pep-0484/.

[6] Guido van Rossum, Warsaw, B. and Coghlan, N. *PEP 8 – Style Guide for Python Code* [online]. 2001. 2022-01-25 [cit. 2022-05-09]. Available at: https://peps.python.org/pep-0008/.

[7] Halas, K. et al. *The Dacite project on GitHub* [online]. 2018. 2022-04-27 [cit. 2022-04-27]. Available at: https://github.com/konradhalas/dacite.

[8] Hattingh, C. *Using Asyncio in Python*. 1st ed. Sebastopol, CA: O'Reilly Media, march 2020. ISBN 978-1-492-07533-2.

[9] Hunt, J. Concurrency with AsyncIO. In: Mackie, I., ed. *Advanced Guide to Python 3 Programming*. Cham: Springer International Publishing, 2019, p. 407–417. Undergraduate Topics in Computer Science. ISBN 3030259420.

[10] Kelly, M. and Caulfield, C. *High-Availability Clustering in the Open Source Ecosystem* [online]. 2014. 2016-05-28 [cit. 2022-04-29]. Available at: https://www.alteeve.com/w/High-Availability_Clustering_in_the_Open_Source_Ecosystem.

[11] Levine, S. *Configuring Red Hat High Availability Add-On With the ccs Command* [online]. 2010. 2017-10-18 [cit. 2022-04-27]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/cluster_administration/ch-config-ccs-ca.

[12] Levine, S. *Starting luci* [online]. 2010. 2017-10-18 [cit. 2022-04-29]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/cluster_administration/s1-start-luci-ricci-conga-ca.

[13] Levine, S. *High Availability Add-On Overview* [online]. 2013. 2019-08-07 [cit. 2019-11-08]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/high_availability_add-on_overview/index.

[14] Massé, M. *REST API Design Rulebook*. 1st ed. Sebastopol, CA: O'Reilly Media, october 2011. ISBN 978-1-449-31050-9.

[15] Micro Focus. *DFS – Success | Micro Focus* [online]. 2014. n.d. [cit. 2022-04-26]. Available at: https://web.archive.org/web/20200807153641/http://www.novell.com/success/dfs.html.

[16] Python Software Foundation. *Asyncio — Asynchronous I/O, event loop, coroutines and tasks* [online]. 2016, 2021-12-29 [cit. 2022-04-25]. Available at: https://docs.python.org/3.6/library/asyncio.html.

[17] Python Software Foundation. *Logging — Logging facility for Python* [online]. 2016, 2021-12-29 [cit. 2022-05-09]. Available at: https://docs.python.org/3.6/library/logging.html.

[18] Python Software Foundation. *Multiprocessing — Process-based parallelism* [online]. 2016, 2021-12-29 [cit. 2022-05-06]. Available at: https://docs.python.org/3.6/library/multiprocessing.html.

[19] Python Software Foundation. *Pickle — Python object serialization* [online]. 2016, 2021-12-29 [cit. 2022-05-09]. Available at: https://docs.python.org/3.6/library/pickle.html.

[20] Python Software Foundation. *Thread State and the Global Interpreter Lock* [online]. 2016, 2021-12-29 [cit. 2022-04-25]. Available at: https://docs.python.org/3.6/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[21] Rauber, T. and Rünger, G. Parallel Programming Models. In: *Parallel programming*. 2nd ed. Berlin, Germany: Springer, June 2013, chap. 5. ISBN 978-3-642-37801-0.

[22] Sajip, V. *Using logging with multiprocessing* [online]. 2010. 2010-09-07 [cit. 2022-05-09]. Available at: http://plumberjack.blogspot.com/2010/09/using-logging-with-multiprocessing.html.

[23] Sajip, V. *Logging Cookbook* [online]. 2016. 2021-12-29 [cit. 2022-05-09]. Available at: https://docs.python.org/3.6/howto/logging-cookbook.html.

[24] Schmidt, K. *High Availability and Disaster Recovery Concepts, Design, Implementation*. 1st ed. Berlin: Springer, Berlin, Heidelberg, 2006. 418 p. ISBN 978-3-540-34582-4.

[25] Smith, E. V. *PEP 557 – Data Classes* [online]. 2017. 2022-01-21 [cit. 2022-04-27]. Available at: https://peps.python.org/pep-0557/.

[26] The Apache Software Foundation. *Apache MPM worker* [online]. 2022 [cit. 2022-05-01]. Available at: https://httpd.apache.org/docs/2.4/mod/worker.html#comments_section.

Appendix A

Diagrams

A.1 Package Diagram of the Asynchronous Scheduler

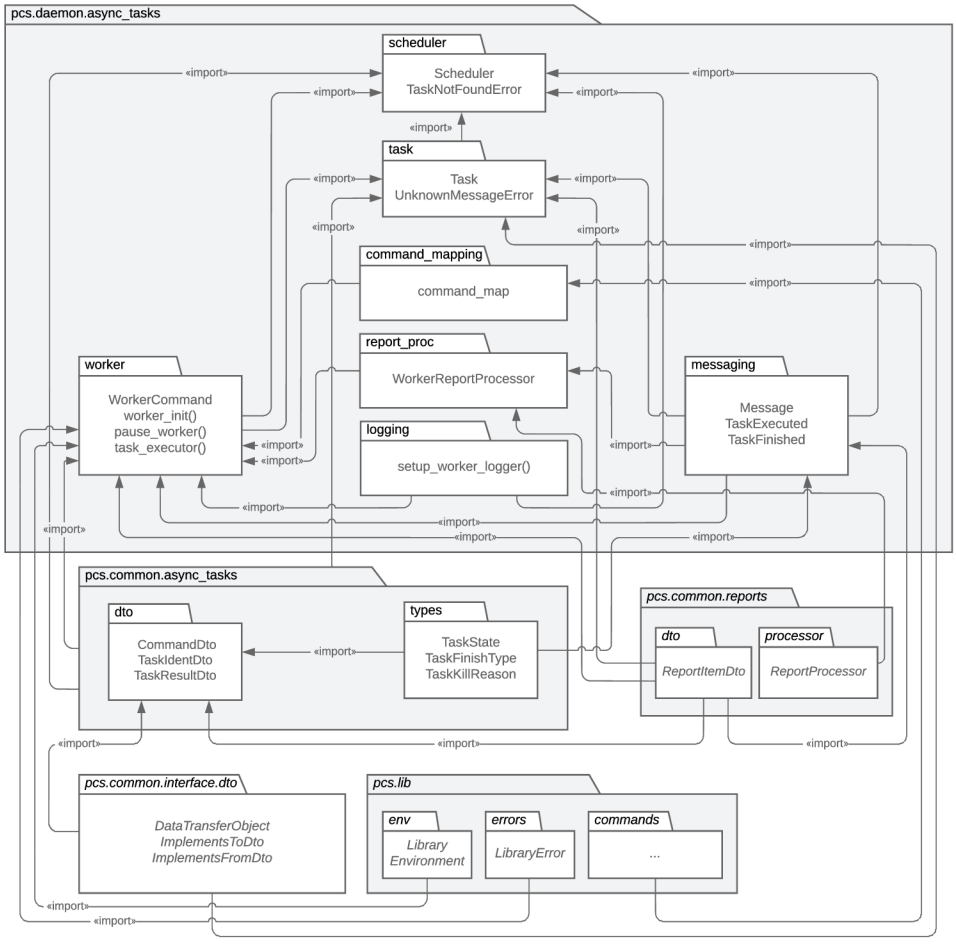

Figure A.1: **Package diagram** This diagram shows the modules and which classes or functions they implement. In italics are the modules or functions that were a part of PCS and were used. All other modules were implemented by the author.

A.2 Example of a Request Timeout in the Old Architecture

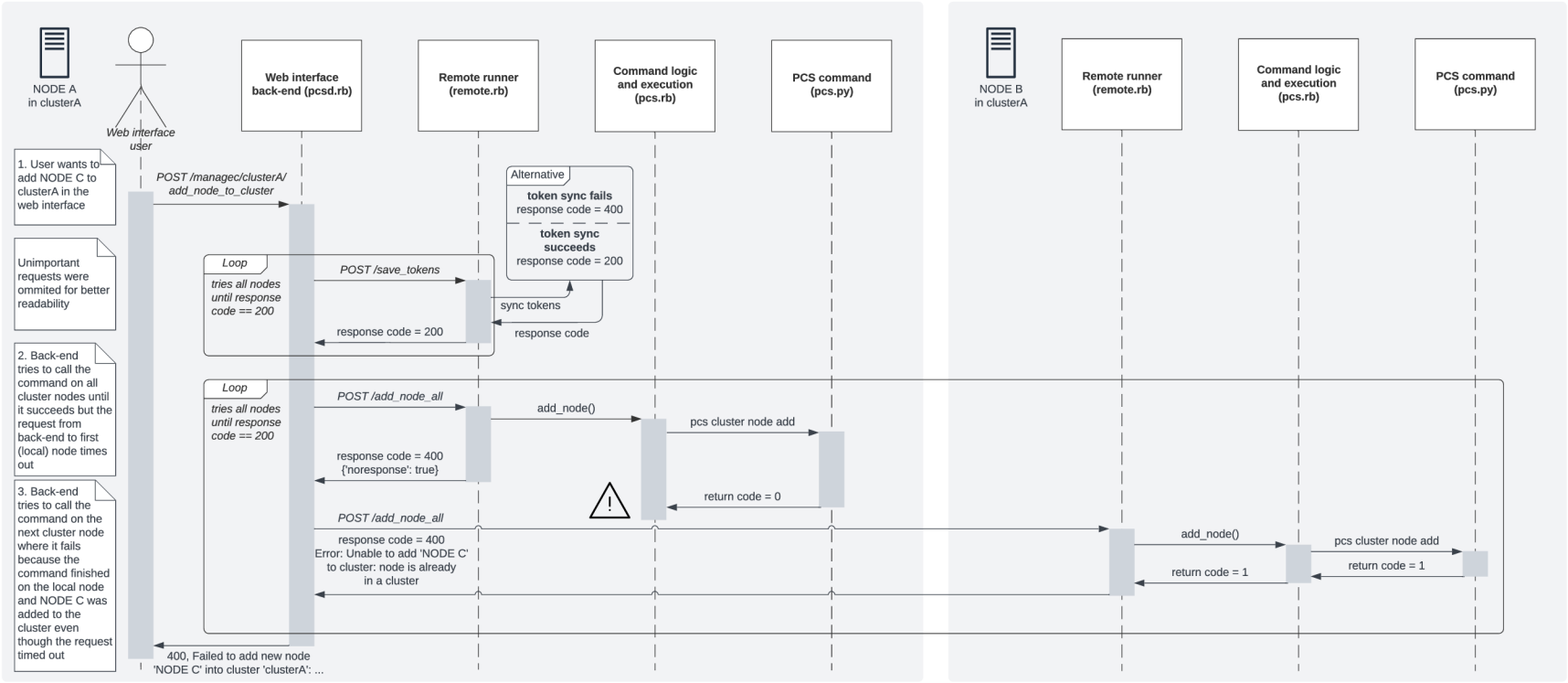

Figure A.2: **Adding a new cluster node from web interface** Cluster „clusterA“ contains nodes „NODE A“ and „NODE B“. There is a new node, „NODE C“, that is not a member of any cluster (this is a prerequisite to add a node to a cluster). In this case, the timeout occurs where the alert icon is located. The first call – addressed to NODE A – to attempt to add NODE C succeeds, but only after the timeout is exceeded. The back-end interprets this as a failure. Therefore, the remaining nodes are tried with the same request (NODE B in this figure). All these calls fail, too, because the prerequisite of no cluster membership is not met for NODE C. The result is that NODE C is added but the web interface reports that the action did not succeed.

A.3 Sequence Diagram of the Asynchronous Scheduler

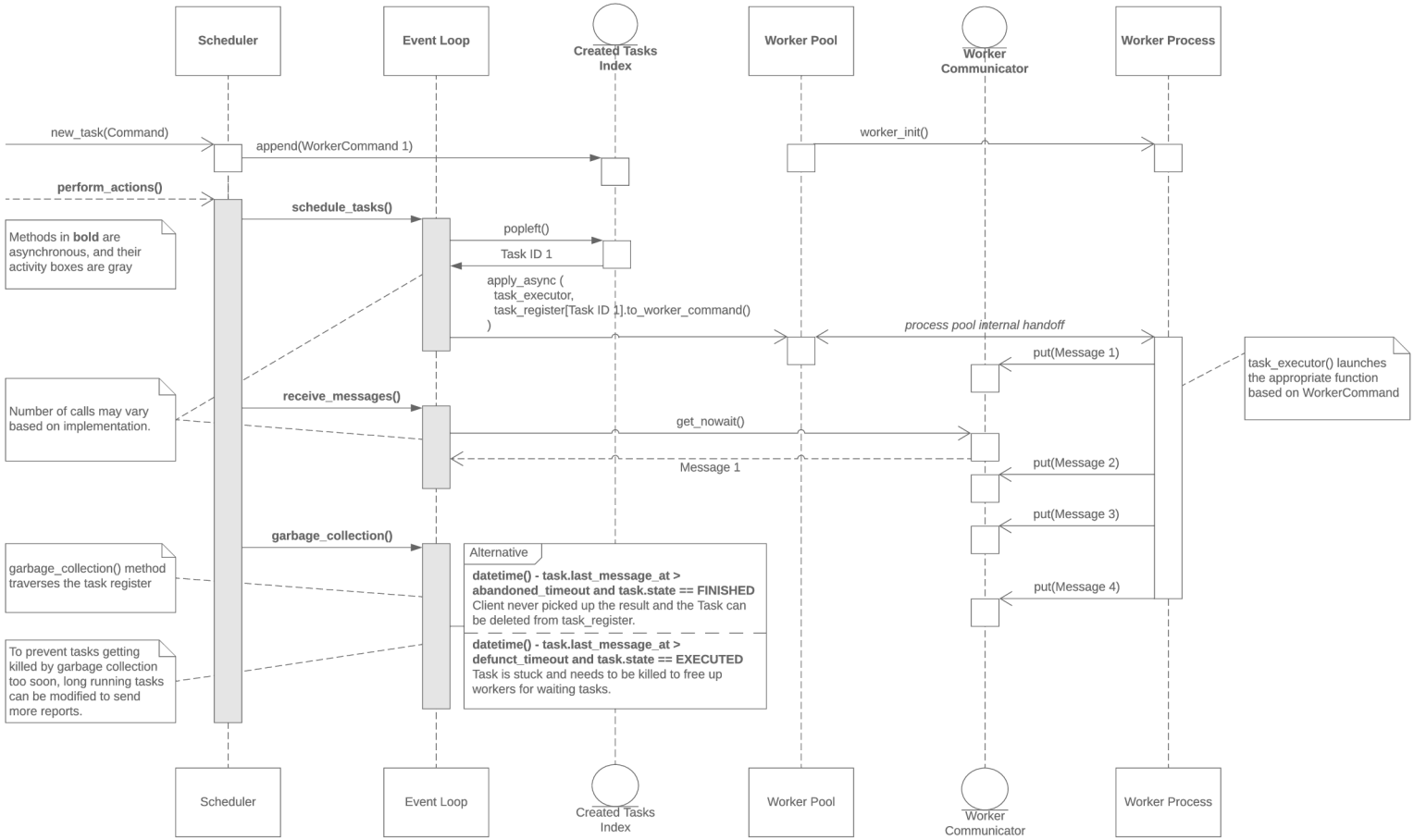

Figure A.3: **Sequence diagram** This diagram shows the basic principle of operation of the scheduler. The `perform_actions` method is periodically scheduled to run on the event loop. The messages from the worker shown as not picked up by the scheduler will be picked up in subsequent runs of `receive_messages`.

A.4 Class Diagram of the Asynchronous Scheduler

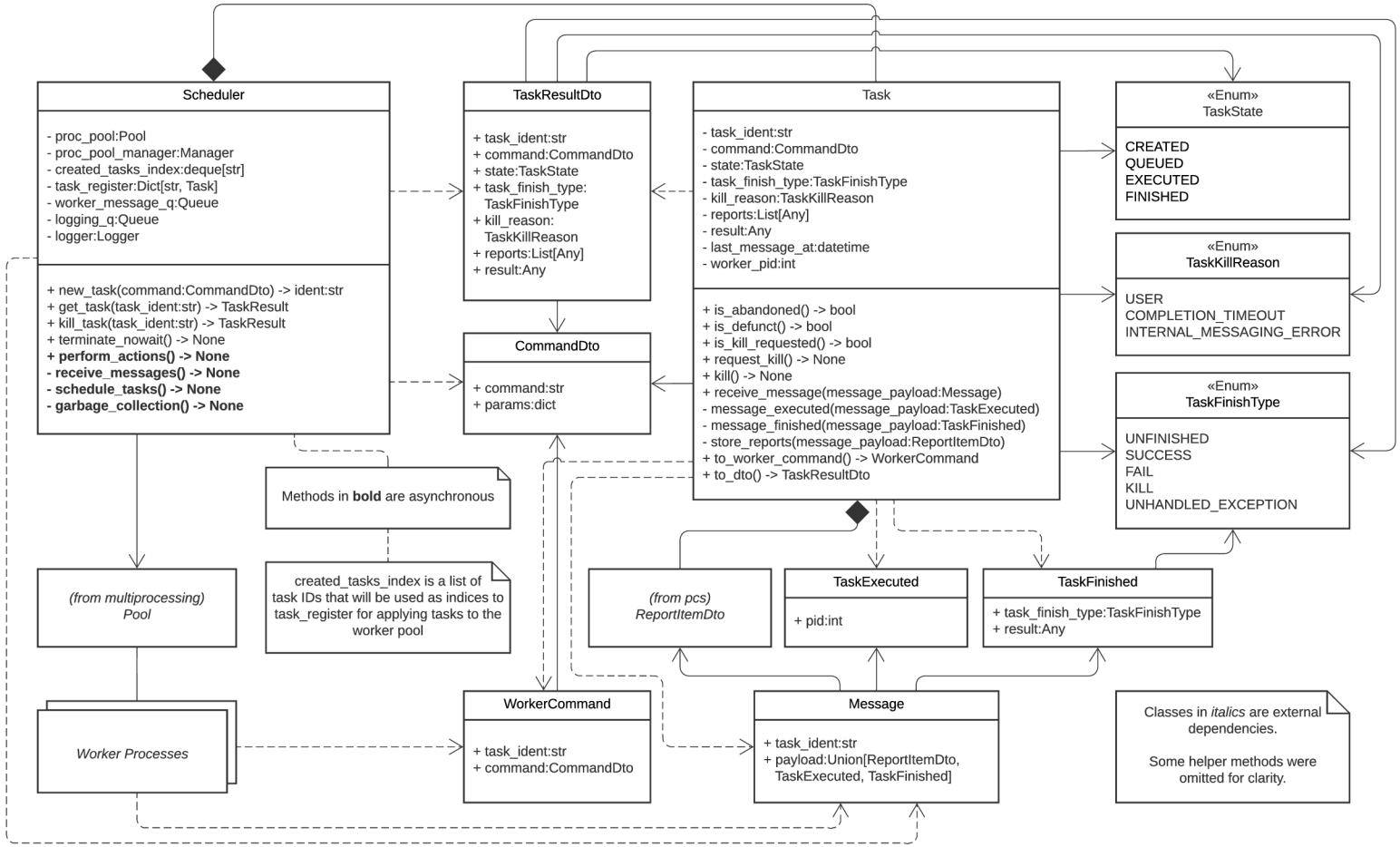

Figure A.4: **Class diagram** This diagram shows the classes of the asynchronous scheduler and their relationships. Properties are annotated with their data type. The method signatures have arguments and return values annotated with their data type as well. Typing was an integral part of the design process and it better shows relationships between objects.

Appendix B

Manual

B.1 Proof of Concept

The proof of concept requires only the Python 3.6+ interpreter to run. The only supported argument is a number between 1 and 6 that corresponds to the test cases in Section 4.1. To use custom options, the constants at the top of the `common.py` file can be modified and launched as the first test case.

Usage:
`python3 test_scheduler.py <test-number>`

Options in the `common.py` file:

- `TASK_SPEC` – a list of tasks to be processed by the scheduler, use the task types;
- `SECS_BETWEEN_TASKS` – seconds elapsed between generating every task;
- `SECS_TASK_RUNNING` – seconds spent simulating work in a task;
- `WORKER_COUNT` – number of processes executing tasks;
- `REGENERATE_WORKER_AFTER` – number of tasks after which worker is killed and replaced;
- `UNHANDLED_EXCEPTION` – produces an unhandled exception in the „exception“ task instead of an expected exception;
- `OPT_ON` – enable message listening optimization (contains bugs, scheduler may stop receiving messages).

There are 3 types of tasks that can be scheduled in `TASK_SPEC`:

- `„normal"` – task is created, sleeps for `SECS_TASK_RUNNING` and ends successfully;
- `„exception"` – task is created and raises an exception;
- `„hang"` – task is created and enters an infinite sleep loop. To test task killing, issue signals with the `kill` command to the worker process identifier found in the output.

B.2 PCS with the New Asynchronous Scheduler and REST API

Since the final implementation is an essential part of PCS, this manual focuses on installation of the modified version of PCS. For demonstration of the asynchronous scheduler and the new REST API, a simple client that implements PCS commands `pcs cluster status` and `pcs resource enable` was created. Its behavior is similar to the client provided in the stable PCS package.

On the attached medium, an RPM[1] package with the modified version of PCS is included in addition to the source codes. It is much easier to set up for demonstration purposes.

Caution! This package is NOT SUITABLE for anything other than testing. It opens a port accessible from the Internet with no security barriers in place.

Environment Preparation

A Pacemaker cluster is needed for testing. The easiest cluster to install is a single-node cluster on a virtual machine. The chosen operating system for the guest is Fedora Server Linux[2] 36 because it is the version targeted by the base PCS version where the scheduler is integrated.

These procedures describe the fastest way to test PCS. By no means is this a guide for setting up clusters that guarantee high availability and are secure.

1. Download the ISO for Fedora Server 36: https://download.fedoraproject.org/pub/fedora/linux/releases/36/Server/x86_64/iso/
2. Create a virtual machine. Minimum VM requirements: 2 processors, 8GB storage, 2GB RAM, Ethernet connection to any network.
3. Install Fedora on the virtual machine.
   - In the Network & Host Name section of the installer, the host name must match the name of the virtual machine. Configure a static IPv4 address on the Ethernet interface. Configure the DNS servers.
   - In the Root Account section of the installer, select the „Enable root account“ radio box. Set the root password and check the „Allow root SSH login with password“ check box.
   - Remaining options can use the default values.
4. Install PCS, start and enable PCSD on startup:
   ```
   dnf install pcs
   systemctl start pcsd
   systemctl enable pcsd
   ```

[1] The RPM Package Manager – more at: https://rpm.org/
[2] Fedora Server, more at: https://getfedora.org/en/server/

Installation from the Provided Package

The provided RPM package installs a modified version of PCS that contains the implementation of this thesis. It installs the asynchronous client (`pcs_async`) alongside the stable PCS client (`pcs`). The package also contains the modified PCS daemon that is used by default when PCSD is enabled on the system.

- Copy the PCS package (`pcs-async.rpm`) from the attached medium to the virtual machine.
- Install (or update installed PCS) the modified PCS package with dnf[3]:
  `dnf install ./pcs-async.rpm`

With these steps completed, modified PCS is ready to be used:

- PCSD contains the new REST API for the asynchronous client. Start the daemon and enable it on startup with:
  `systemctl start pcsd`
  `systemctl enable pcsd`
- The fully featured PCS is still available as `pcs`.
- The asynchronous client is available as `pcs_async`.

Launch PCS from Sources

To debug PCS, the asynchronous scheduler, or its REST API, the source code may need modifications. To test them, PCS and PCSD need to be launched from the modified sources.

- Install development dependencies:
  `dnf install git make automake autoconf wget openssh-clients gcc \`
  `gcc-c++ libcurl-devel libcurl fence-agents-all fence-virtd \`
  `fence-virt ruby-devel rubygem-bundler rubygem-minitest libffi-devel \`
  `booth booth-site python3-setuptools_scm python3-wheel python3-devel \`
  `python3-pip python3-virtualenv`
- Copy the provided Git[4] repository to the virtual machine.
- Create a virtual environment (outside the Git repository):
  `python3 -m venv --system-site-packages <venv-folder-path>`
- Activate the virtual environment:
  `. <venv-folder-path>/bin/activate`
- Change directory to the PCS Git repository.
- Start the autogen script:
  `./autogen.sh`.

[3] DNF software package manager, more at: https://docs.fedoraproject.org/en-US/quick-docs/dnf/
[4] Git version control system, more at: https://git-scm.com/

- Launch Autoconf[5]:
  ```
  ./configure --enable-local-build --enable-dev-tests \
  --enable-concise-tests --enable-parallel-tests \
  --enable-parallel-pylint
  ```
- Launch make:
  ```
  make
  ```

With these steps completed, PCS is ready to be used from the repository:

- To start the modified PCSD, use the test script that stops PCSD from the system PCS installation and launches PCSD from the repository: `./scripts/pcsd.sh`.
- To use the modified PCS, replace `pcs` with: `./pcs/pcs`.
- To use the modified asynchronous client, replace `pcs_async` with: `./pcs/async_client`.
- To launch the automated scheduler tests, launch: `make tests_tier0`.

Setup a Single-Node Cluster

A single-node cluster allows to use the `pcs cluster status` and `pcs resource enable` commands.

1. Enable the ports used by the Pacemaker cluster stack in the firewall:
   ```
   firewall-cmd --permanent -add-service=high-availability
   firewall-cmd --reload
   ```
2. Set a password for the `hacluster` user.
   ```
   passwd hacluster
   ```
3. Authenticate PCS to PCSD, use the `hostname` configured during the system installation and credentials for the `hacluster` user when prompted:
   ```
   pcs host auth localhost
   ```
4. Create a single-node cluster named `cluster-name` consisting of this virtual machine only and start it:
   ```
   pcs cluster setup <cluster-name> --start --wait localhost
   ```
5. Disable fencing (fencing is a vital part of ensuring high availability, never use an HA cluster without fencing configured):
   ```
   pcs property set stonith-enabled=false
   ```
6. Add at least one dummy resource with the name `resource-name`: `pcs resource create <resource-name> ocf:heartbeat:Dummy`

[5] Autoconf, more at: https://www.gnu.org/software/automake/

Some Useful PCS Commands

This subsection provides some basic PCS commands that are needed to start and stop the cluster and disable resources to use the `resource enable` command.

- `pcs cluster start [--wait[=n]]`
  Starts the cluster on the local node. For clusters with more than one node, the `--all` switch can be added to start the whole cluster. The `--wait` option can be used to let the command end only after the cluster has started, n can be specified for a maximum wait time in minutes. The command does not wait for the cluster to start by default.

- `pcs cluster stop --all`
  Stops the cluster. Using this command without the `--all` switch produces a quorum loss warning on a single node cluster. The warning can also be overridden by using `--force` but this is a more dangerous option.

- `pcs resource disable <resource id | tag id>...`
  Stops the resource and tells the cluster to keep it stopped. This command takes more parameters but these are not that important for the purposes of this manual, use the `--help` switch to learn about them.

Using the Asynchronous Client

The asynchronous client supports these PCS commands:

- `pcs_async cluster status`
  Prints the cluster status on the standard output. If the cluster is not running, it generates an error report that is printed on the standard error output.

- `pcs_async resource enable --resource_or_tag_ids=<JSON-list> \`
  `[--wait=<JSON-value>]`
  Tells the cluster to start the specified resources if the configuration allows it. The `--wait` option can be used to let the command end only after the resource has been enabled. The `n` controls the maximum wait time in seconds. The command does not wait for the resource to be enabled by default.

The PCS help and manual pages specify the `resource enable` options differently. The reason for this change is the minimal argument parser used by the asynchronous client. The values must be encoded in JSON[6] to allow for lists and other composite data types without explicit support from the argument parser. This is a translation reference for the argument names and values:

- `<resource id | tag id>...` => `resource_or_tag_ids`
  List of tag or resource identifiers.

- `wait` => `wait`
  Wait is disabled if it is not specified or is set to `false` (without quotes). The default timeout of 60 minutes is enabled with `null`. A positive number is interpreted as the wait timeout in *seconds*.

[6] JSON – JavaScript Object Notation, more at: https://tools.ietf.org/html/rfc8259

NOTE: Using JSON in the terminal can be tricky. Single quotes should be used around the whole argument value. These single quotes will be consumed by the shell and protect the double quotes. Double quotes must be used to delimit strings inside JSON encoded arguments. Boolean values are `true` or `false` without the double quotes. Conversion to the Python `None` type can be achieved by specifying `null` without double quotes. These are some example commands demonstrating these rules:

- `pcs_async resource enable --resource_or_tag_ids='[„dummy1"]' \`
  `--wait=null`
  Enables the `dummy1` resource and waits for the resource to start for a maximum of 60 minutes before returning.
- `pcs_async resource enable \`
  `--resource_or_tag_ids='[„dummy1", „tag1"]' --wait=30`
  Enables the `dummy1` resource and resources grouped under `tag1` and waits for the resources to start for a maximum of 30 *seconds* before returning.
- `pcs_async resource enable --resource_or_tag_ids='[„dummy1"]'`
  Tells the cluster to enable the `dummy1` resource and immediately returns.