FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Analysis of Data from a Network of Pixel Detectors
Student:	Petr Fiedler
Supervisor:	doc. Dr. André Sopczak
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2019/20

Instructions

A network of 16 pixel detectors (Timepix devices) were installed in the ATLAS experiment at CERN (atlas.ch) to measure particle fluxes and the intensity of the proton beam collisions at the Large Hadron Collider. The analysis of these data is performed by the IEAP at the Czech Technical University in Prague. The current system stores data produced by the Timepix devices to the local storage server at IEAP. The project is carried out in an international collaboration and offers the possibility to travel to CERN.

Design, implement and document a program that transfers and stores periodically the data from the IEAP server to world-wide grid wlcg.web.cern.ch. Installation of software needs to access the grid on a local server based on the "rucio" package and transfer of the current data to the grid.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague March 14, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Analysis of Data from a Network of Pixel Detectors

Petr Fiedler

Department of software engineering
Supervisor: Doc. Dr. André Sopczak

May 14, 2018

Acknowledgements

I would like to thank my supervisor, Doc. Dr. André Sopczak, for all advises and active involvement during the work. I also would like to thank my family and my girlfriend for the encouragement during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 14, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Fiedler, Petr. *Analysis of Data from a Network of Pixel Detectors.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá návrhem a implementací balíčku skriptů, sloužících k automatizaci nahrávání dat pomocí middleware Rucio ze serveru `atlastpx2.utef.cvut.cz`, patřící Ústavu technické a experimentální fyziky na Českém vysokém učení technickém, na Worldwide LHC Computing Grid koordinovný Evropskou organizací pro jaderný výzkum. Díky tomuto nahrávání jsou tato data dostupná pro všechny, kdo je potřebují a mají přístup na zmíněný grid. Navíc software poskytující nástroje potřebné pro připojení a používání zmíněného gridu byl nainstalován a nakonfigurován v rámci této práce.

Klíčová slova přenos dat, Rucio, WLCG, instalace, CernVM-FS, ÚTEF

Abstract

This thesis deals with design and implementation of a bundle of scripts automating the upload of specific files using Rucio middleware from server `atlastpx2.utef.cvut.cz` at Institute of Experimental and Applied Physics at Czech Technical University to Worldwide LHC Computing Grid coordinated by European Organization for Nuclear Research. Thanks to the upload, the data are available to everyone who needs them and has access to that Grid. Furthermore, installation and configuration of software providing access to tools necessary to connect and use the Grid are part of this thesis.

Keywords data transfer, Rucio, WLCG, installation, CernVM-FS, IEAP

Contents

List of Figures

Introduction

Particle physics attracts more and more attention, new discoveries are made, which revolutionary change our view of the world. Although research in this field cannot be done without computer technologies. Computer systems process an enormous amount of data during a research. In the Large Hadron Collider (LHC) in European Organization for Nuclear Research (CERN) bunches of protons collide and thousands of particles are produced. About every particle very precise information is collected. Every second, petabytes of raw data are collected. These data are comprised, filtered, stored and processed so they can be used for further analysis. Processed data have to be shared.

At Institute of Experimental and Applied Physics (IEAP) at Czech Technical University in Prague (CTU), the data are transferred from CERN servers to IEAP servers, where they are stored. That also applies to server `atlastpx2.utef.cvut.cz`, where data from Timepix detectors from A Toroidal LHC Apparatus (ATLAS) are stored and analyzed. Timepix installed in the ATLAS cavern is a pixel detector for particle imaging and detection. An analysis of these data takes a lot of time because there are terabytes of the data per year and the server does not have enough computing power to analyze them fast. Moreover, if it is necessary to some external scientists work with these data, they need access to the server and that means a lot of paperwork, which takes some time.

The output of this work will speed up the research at IEAP using the technology of the Worldwide LHC Computing Grid (WLCG). It is global computing infrastructure coordinated by CERN and providing resources to store, distribute and analyze the data from LHC.

This thesis has three individual goals. The first goal is to get an overview of what the WLCG and Rucio are and how to use them to accomplish following goals. The second goal is to find a way how to connect from IEAP server to the WLCG and manage data on it using Rucio middleware. The third and the ultimate goal is to analyze processes, that need to be done, so the data from IEAP server can be uploaded to WLCG, and to design and implement

program that transfers and store the data from the IEAP server to WLCG.

In the first chapter, I explain what the WLCG and Rucio are. In the next chapter, I analyze possible way of getting access to Rucio and other tools necessary to connect to the WLCG. Once there is access to the necessary tools, a program transferring data to IEAP server can be used. In chapter three I deal with analysis and design of this program. The IEAP server is accessible via domain `atlastpx2.utef.cvut.cz`.

Chapter 1

Technologies

In this chapter, the technologies I have to work with are described. The terms are explained and basics of the technologies are described.

1.1 Worldwide LHC Computing Grid

The WLCG project is coordinated by CERN. The WLCG is global computing infrastructure providing computing resources. As the project is a collaboration of more than 170 computing centers in 42 countries all around the world, it is the largest computing grid on the world and serves to store, distribute and analyze the data from LHC. The data on the WLCG are equally available to all partners no matter their physical location. [1]

In order to use the WLCG, a user must be registered in an LHC-recognized Virtual Organization (VO) [2]. The Virtual Organization Membership Service (VOMS) enables VO to authorize and is at the core of the WLCG authorization [3].

The VOs are sets of identifiers. These identifiers are organized in groups and these groups have roles assigned to individuals. These are groups of people at different locations working to achieve a common goal. [4]

1.2 Rucio

Rucio is a project developed by the ATLAS experiment. It provides services and associated libraries for management of large volumes of data spread across facilities, institutions and organizations, and countries. Rucio is highly scalable and modular. [5]

Because data are physically stored in files, Rucio's smallest operational unit is the file. Files can be grouped into named sets of files, datasets, these can be grouped into named sets of datasets, containers, and even these can be grouped into other containers. All three types use the same naming scheme

composed of a scope and a name. The combination of a scope and a name is called data identifier (DID). So the Logical File Name (LFN), a term used in data grid terminology is equivalent to the DID in Rucio. The scope divides the namespace into several sub-namespaces. By default, users can read from all scopes but write only into their own. The DIDs are unique over all time, so once the DID is used, it can never be reused to refer to anything else. [6]

The smallest unit of storage space addressable in Rucio is a Rucio Storage Element (RSE). It is a logical representation of storage system for physical files. The RSEs are grouped into sites. The RSEs in a site have different purposes. The most common RSEs are scratch disks, local group disks, and data disks. [7]

Every user has a quota of 30% of each scratch disk RSE. A user which is registered in relevant VO has 95% quota on all local group disk RSEs in the corresponding country. Data disk RSEs serve for special purposes and only special services or admin accounts can write to these RSEs. [8]

File replicas are managed according to replication rules. A replication rule defines the minimum number of file replicas on a set of RSEs. It is owned by an account and every account can set multiple replication rules. Rules can have a limited lifetime and can be added, modified or removed at any time. The file replicas can be deleted in two different modes: greedy and non-greedy. The greedy mode deletes immediately all replicas not protected by replication rule. The non-greedy mode deletes the replicas when storage policy dictates that space must be freed. [9]

Accounting is the measure of how much resource an account has used. A quota is a policy limit applied to an account by the system. Rucio accounts are accounted only replicas requested by replication rules. If two users set a replication rule on the same file and request replica on the same RSE, they are both accounted for the file, even though there is only one physical copy of the file. [10]

Chapter 2

Getting access to the Rucio

In this chapter, the technologies used to get an access to the Rucio and the WLCG are briefly described and why I decided to use them is explained. Also, the process of an installation and configuration of the technologies is described.

An easy way how to control data on the WLCG using Rucio middleware is to use Rucio client. It is an application written in python and it removes the complexity of raw HTTP requests sent to Rucio's REST API [11].

To use the WLCG and therefore the Rucio, the VOMS must be used to authorize. A simple way to do so is to use the VOMS proxy. It is used to request an Attribute Certificate (AC) from VOMS server and to create a proxy that is used to control authentication and authorization when accessing third party services. The AC carries the attributes that a person holds in a certain VO. [3, 12]

I found two ways how to make these tools available. The first one is to install the software directly on the local machine. The problem is that on the server, where the software had to be installed, the operating system Scientific Linux 7.3 is running. It is relatively new system and every guide how to install the VOMS proxy is only for the Scientific Linux 5 and 6. So I choose the second way, to use the technology of CernVM File System (CernVM-FS).

2.1 CernVM File System

The CernVM-FS is a read-only file system delivering software onto virtual machines and physical computers in a fast, scalable, and reliable way. Files and metadata are downloaded on demand and cached. The CernVM-FS ensures data authenticity and integrity of the data. In contrast to general-purpose network file systems such as NFS or AFS, the CernVM-FS is optimized for running and compiling software. Software installed in CernVM-FS does not need to be further packaged as it does in virtual machine images or Docker images. The software is versioned and distributed file-by-file. [13]

The CernVM-FS automatically mounts repositories when they are accessed and automatically unmounts them after some system-defined idle time [14].

2.2 Installation and configuration

To install the CernVM-FS, I used the package manager `yum`, which is built in Scientific Linux. The CernVM-FS is configured to mount repositories `atlas.cern.ch`, `atlas-condb.cern.ch` and `grid.cern.ch`. The repository `atlas.cern.ch` contains various software and some of it uses data in the repository `atlas-condb.cern.ch`. The repository `grid.cern.ch` contains various grid software which is used to work with the WLCG [15]. The last part of the configuration is to prepare the environment for use of so-called `setupATLAS`, which prepares the environment for other software setups such as Rucio clients setup.

```
1 # CernVM-FS
2 yum install 'https://ecsft.cern.ch/dist/cvmfs/cvmfs-
    release/cvmfs-release-latest.noarch.rpm'
3 yum install cvmfs cvmfs-config-default cvmfs-auto-
    setup
4
5 cvmfs_config setup
6
7 CVMFS_CONFIG_FILE='/etc/cvmfs/default.local'
8 echo 'CVMFS_REPOSITORIES=atlas.cern.ch,atlas-condb.
    cern.ch,grid.cern.ch' > $CVMFS_CONFIG_FILE
9 echo 'CVMFS_HTTP_PROXY=DIRECT' >> $CVMFS_CONFIG_FILE
10
11 cvmfs_config reload
12 cvmfs_config probe
13
14 # setupATLAS
15 echo '' >> /etc/bashrc
16 echo 'export ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.
    ch/repo/ATLASLocalRootBase' >> /etc/bashrc
17 echo 'alias setupATLAS="source ${
    ATLAS_LOCAL_ROOT_BASE}/user/atlasLocalSetup.sh"'
    >> /etc/bashrc
```

Figure 2.1: CernVM-FS installation

As seen in figure 2.1 the configuration of the CernVM-FS is stored in the file `/etc/cvmfs/default.local` and the tool `cvmfs_config` to work with

the configuration of the CernVM-FS. The environment is set up for use of the `setupATLAS` using the `/etc/bashrc`.

The `cvmfs_config` is utility providing commands used to set up the system in order to use the CernVM-FS. The command `cvmfs_config setup` does the base setup, such as creating a user for CernVM-FS and allowing all users to access the CernVM-FS mount points. The command `cvmfs_config reload` hotpaches the CernVM-FS instance in a way, that active code is unloaded and the new code is loaded from the currently installed binaries. The CernVM-FS is re-initialized and therefore the configuration is re-read. The command `cvmfs_config probe` accesses the mount point of every repository. [16]

2.3 User credentials for VOMS

To use the VOMS proxy, user credentials must be passed to it during initialization. User credentials can be put anywhere and their location passed via appropriate options, but there is also the default setup. The credentials should be put in the user's home directory. The certificates can be encoded in PKCS #12 or PEM formats. If the PKCS #12 credentials are used, the default path is `~/.globus/usercred.p12` and the file permissions must be `600`. If the PEM is used, the default path of the certificate is `~/.globus/usercert.pem` and the default path of the private key is `~/.globus/usercert.pem`. The permissions of the certificate file must be `644` and the permissions of the private key file must be `400`. If both formats are present, the PEM takes priority. [12]

Chapter 3

Uploading to Worldwide LHC Computing Grid

In this chapter, the analysis of processes necessary to upload the Timepix data in a specific form is described. Also, analysis, design, and implementation of each process are described in this chapter.

The data should be uploaded in a form of measured data per day. To achieve this, more steps are needed to be done. I analyzed the processes and found few common ones. The first one is to locate where the data are stored. There are more data disks on the server and the data can be stored on whichever of them. The second one is to merge the data. It is good to have data stored in per day files but some are in per hour files. The third and also the last one is the upload itself. The files should be uploaded only when there is a reason for it, such reason can be that the data are not yet uploaded or that the uploaded data are not up-to-date.

All these processes are mostly a work with files, so I decided to implement them as a bundle of Bash scripts. The bundle works with the data from Timepix devices, so I called it simply `tpx`. It provides commands for getting the path of the data, merging files and uploading to the WLCG. As seen in figure 3.1, some commands use the bundle itself. Both the upload and the merge need to locate the data path and the upload sometimes needs to merge the files first.

The bundle contains more than just three scripts, one for each use case, `tpx_path`, `tpx_merge`, and `tpx_upload`. It contains three more scripts. One of them, `rucio_connect`, serves for setting up the ATLAS environment, the Rucio clients, and the VOMS proxy. Another one, `tpx_claim`, serves for a transition of ownership. Every file operations by this script are made purely on the WLCG, but this script is not part of this thesis, it is there only for a future possible use. Next one, `tpx`, covers up the bundle as a whole. It provides help with a short summary of available commands and invokes them when the command is provided as the first argument. It is done the way the symbolic

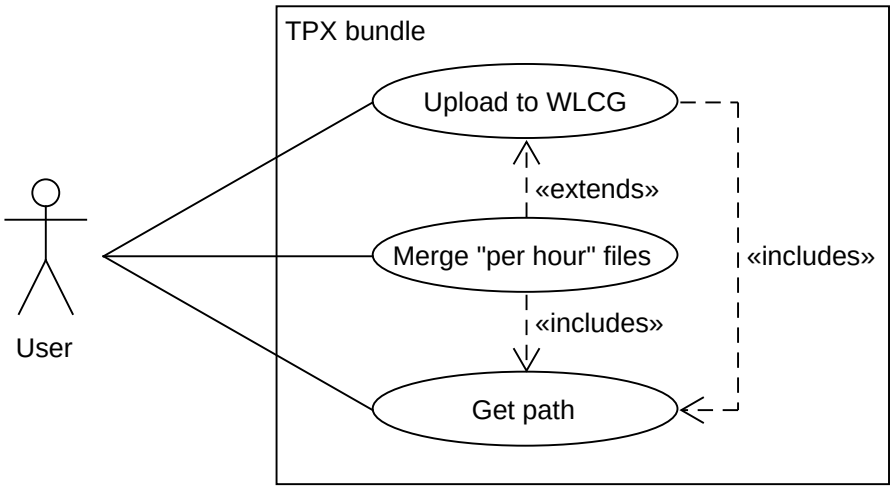

Figure 3.1: Use cases of the TPX bundle

links pointing to this script can be used. As seen in figure 3.2, the script, `tpx_config`, serves as a base for any of those scripts. It contains functions for setting up help message, providing temporary storage and managing exits.

The scripts in the bundle use four exit codes, these are number 0 for a successful exit, number 40 for invalid input, number 44 when data to work with were not found and number 50 when some other error occurs. Various scripts can have various reasons why to fail, but all of them print info into the error output. The error codes are inspired by HTTP status codes. The first and last digits of the HTTP status codes are used.

The temporary files are not stored in `/tmp` as it usually does. When the files had to be merged there, it failed because of not enough space, so I made another temporary storage on one of the data disks. The path to the storage is `/data7/tmp` and the storage directory has set the same permissions as the original `/tmp` directory. Every instance of the scripts in the `tpx` bundle creates its own temporary directory. It does so by creating a new temporary file in `/tmp` using the command `mktemp`. It uses the file name and creates a directory with the same name in `/data7/tmp`. The temporary file is removed afterward.

3.1 Getting data path

There are more data disks on the server. They are called `/data<N>` where `<N>` is a number, for now between 1 and 8. The Timepix data are stored on disks `/data6` and higher. On these disks is a directory `ATLASTPX_root` containing directory structure with the Timepix files. The directory `ATLASTPX_root` con-

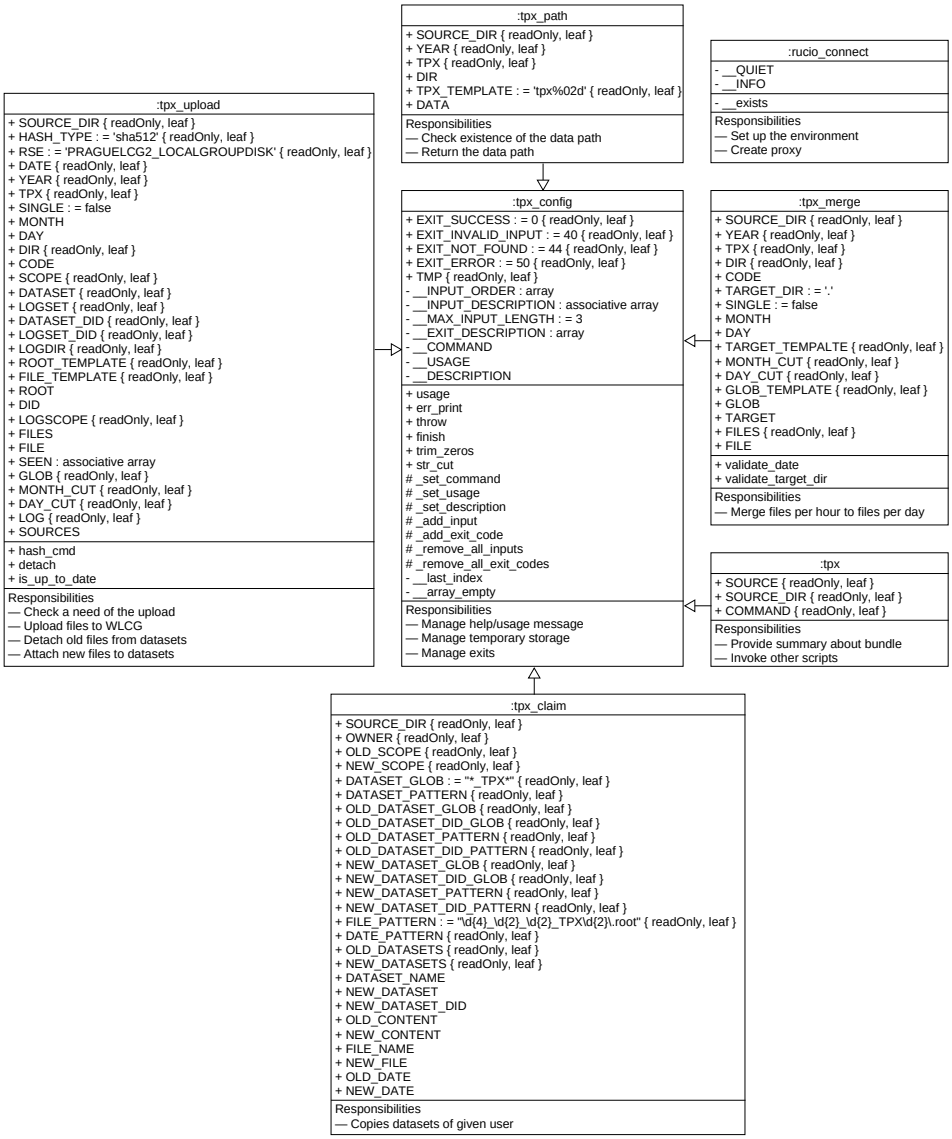

Figure 3.2: Structure of the bundle

tains directories of data per year. Their name corresponds to the year number. In these directories, the directories representing each individual detector are stored. They are called `tpx<M>` where `<M>` is a number formatted to two digits, so the `<M>` has to be between 01 and 16. For the purpose of locating the data, I created the `tpx_path` script.

The script first checks if the first argument is `--help`, if it does then the help message is printed to error output and the script exits with zero code, otherwise, the script continues. Then it is checked if the arguments are valid,

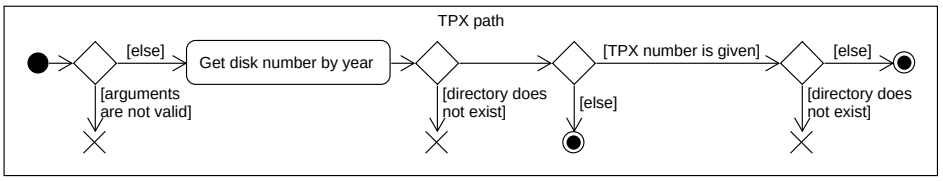

Figure 3.3: Process of getting the data path

that means that the number of arguments equals one or two, the year argument consists of four digits and if the detector number argument is passed it is a number. If the arguments are not valid, the script exits with code 40, and the help message is printed to the error output. If the arguments are valid the data path can be processed.

The data per one year should be all on one disk, so the script decides on which disk to look for by the year. The 2015 data are stored on `/data6`, the 2016 data on `/data7` and the 2017 and 2018 data on `/data8`. Then the script checks if the data are really there. It does so by checking the path existence, e.g. for the 2016 data the path `/data7/ATLASTPX_root/2016` should exist, otherwise, it is considered that no data for the year exists and script exits with the code 44. If a path of the detector directory is requested it is checked if the year directory contains it, e.g. for the detector 3 for the 2016 data the path `/data7/ATLASTPX_root/2016/tpx03` should exist, otherwise it is considered that no data for the detector in given year exists and script exits with the code 44. If everything goes well the path is printed to standard output and the script exits with the code 0.

3.2 Merging the files

It is good to have the data stored in per day files but some data are stored in per hour files. The forms file per day and file per hour are even mixed together. They are mixed together so much that for example in the directory for 2016 data from detector 2, there are data from some days in per day files and some in per hour files mixed without any rule or pattern. It makes harder to process these data. For this purpose, I created script `tpx_merge`.

The basic idea of the script is to filter out the per hour files and merge them together by days. Because the files are ROOT files, the merge must be done accordingly, using the tool called `hadd`. The ROOT is a data analysis framework and it stores data in binary files called as ROOT files. These files contain histograms and trees. Good custom is to add the suffix `.root` to the file name, so it is easily recognizable. The ROOT also provides the `hadd` to merge these files. Its usage is simple, it is used like `hadd target.root file1 .root file2.root ... fileN.root`.

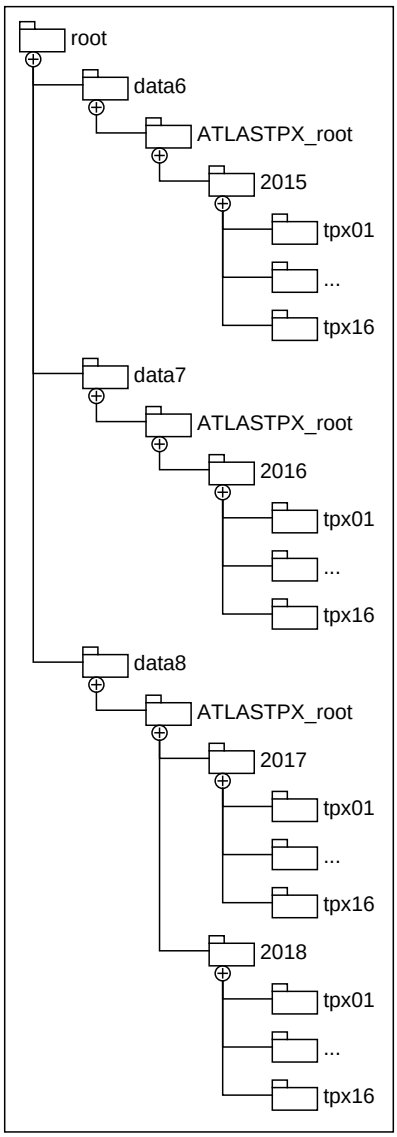

Figure 3.4: Directory structure of the data

The idea of usage of the script is to provide more freedom in use. The script accepts all information as arguments. It can accept up to five arguments but only two are mandatory. The first argument has to be the year of origin of the data and the second one the detector number. With this configuration, the script merges all per hour files stored in directory determined by these two arguments to per day files stored in a current working directory. Then another two arguments can be provided, a month and a day in the month. These make

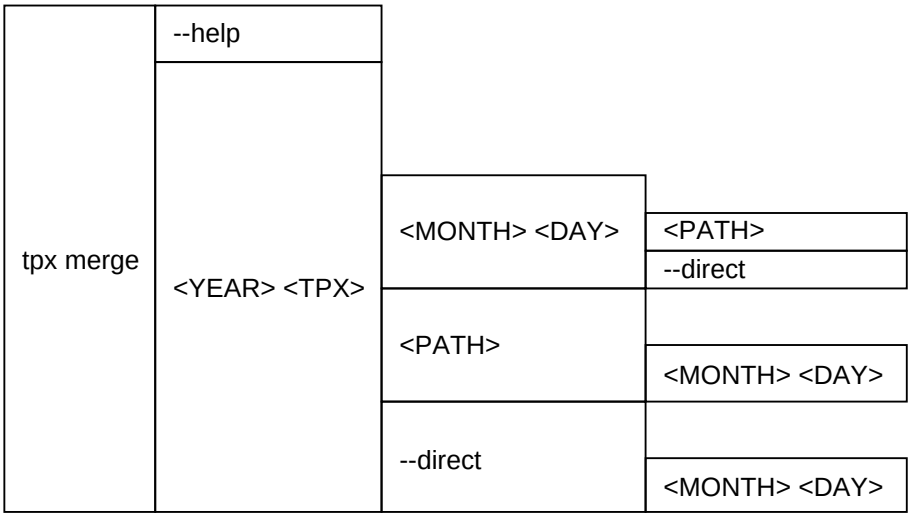

Figure 3.5: Usage of tpx merge

the script to merge only per hour files from the given day to a single file. To manipulate where the data are stored another argument can be provided. The argument can be a path or the option `--direct`. If the path is provided, it is used as the place to store the merger files. If the argument is `--direct` the merger files will be stored in the same directory where the source files are stored, in other words, the per day files will be stored in the same directory as the per hour files. The target directory argument can be used even without the single day specifying arguments and even does not matter in which order they are provided.

The script first needs to ensure that the user does not ask for help. It does so by checking if the first argument is equal to the `--help`. If it does, the script exits with zero code and prints the help message to the error output. If the user does not ask for help, then the script needs to ensure that the mandatory arguments are valid. The validation process corresponds to the validation process in script `tpx_path` mentioned earlier in section 3.1.

Then using the command `tpx path` the data path of given year and given detector number is retrieved. If the `tpx path` failed, its exit code is processed. It the code equals to 40, which means that the arguments for the `tpx path` are not valid, the script exits also with the code 40 and the help message is printed to the error output. However, this situation should never happen, because the validation processes of both scripts are the same. It the exit code equals to 44, which means no data for given detector number and given year exits, the script exits also with the code 44 and the content of error output of the `tpx path` is printed to the error output of the script.

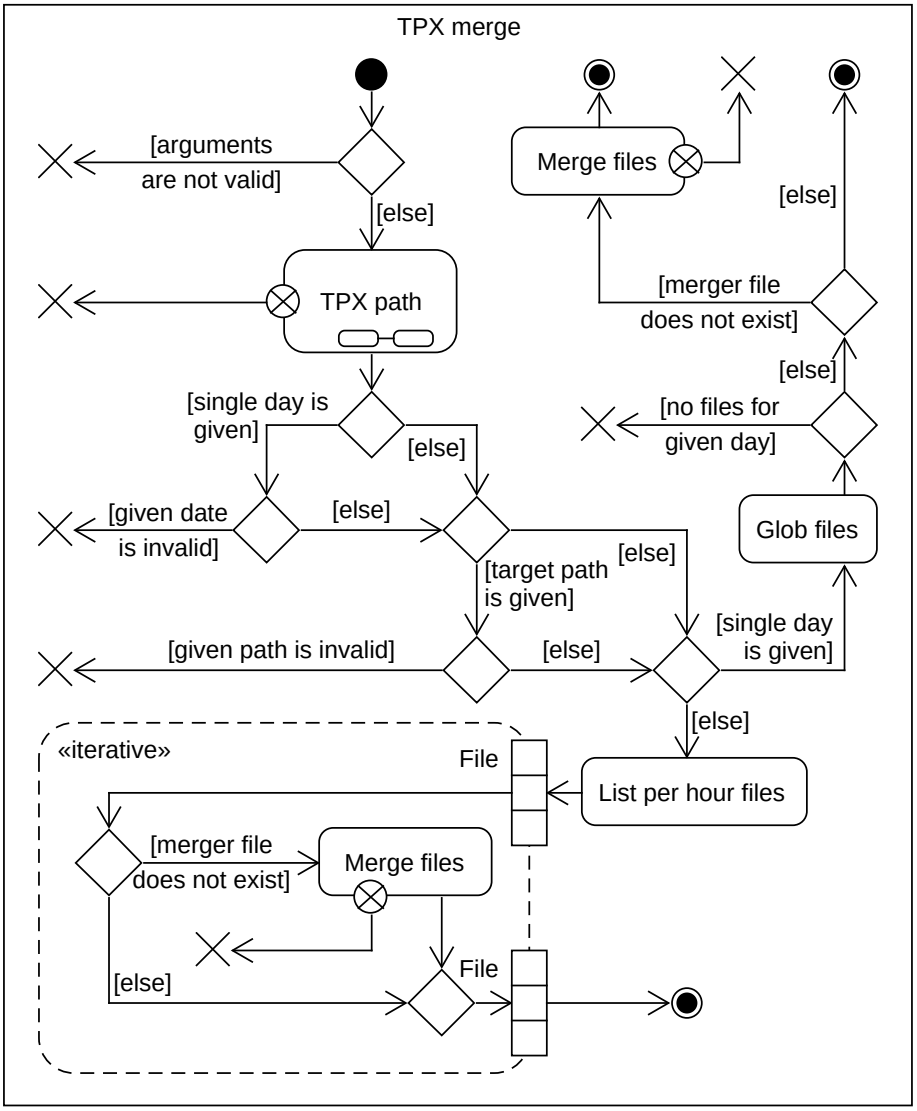

Figure 3.6: Process of merging files

Then a validation of the arbitrary arguments happens if some are passed. If at least four arguments are provided and the third one is a double-digit number it is considered that the third and fourth arguments are month and day specifying files from a single day to merge. If the fifth argument is provided it is considered as the target path argument. If there are only three arguments or the third one is not a number, the third argument is considered as target path argument. If there are five arguments and the third one is

considered as the target path argument, the fourth and fifth arguments are considered as the month and day arguments. The month and day arguments are validated using the command `date`. If the `date` command refuses the arguments, they are considered as invalid because it means that the provided date is not valid. The target path argument is invalid if it is not equal to `--direct` and it is not an existing directory. Again if any of the validation fails, the script exits with code 40, and a message describing the problem is printed to the error output together with the message "`Try 'tpx merge --help'`". When the validation is done the control flow of the script continues in one of two branches, the branch for merging per hour files from a single day or the branch for merging all per hour files.

If the month and day are provided, the branch for merging per hour files from a single day is chosen. The branch starts with building a globing string and a merger file name. Then it is checked if the string matches some files or not. If it does not match any files, the script exits with the code 44 and prints a message, that there are no requested per hour files, to the error output. If the string matches some files, it is checked that the merger file does not exist, otherwise the script exits with the zero code and prints a message, that the merger file already exists, to the error output. If the merger file does not exist, the globed per hour files are merged. If the merging fails, the uncompleted merger file is deleted and the script exits with the code 50 and prints a message, that the merger file could not be created a thus has been deleted, to the error output.

If the month and day are not provided, the branch for merging all per hour files is chosen. The branch starts with listing all per hour files. The files are then iterated one by one. For each file, the month and day are cut off the file name and are used to build a globing string and a merger file name. If the merger file exists and was not tested before, the script prints a message, that the merger file already exists, to the error output. If the merger file does not yet exist, the files are merged. In this case, the testing that the globing string matches some files is useless because the globing string is built from file names of those files which it has to match. If the merging fails, the uncompleted merger file is deleted and the script exits with code 50 and prints a message, that the merger file could not be created a thus has been deleted, to the error output. The script exits though there are still files that could be merged because I use the policy "if it should crash, do it fast". I use the policy because when the script is integrated into other scripts, the ignorance of an error could lead to undefined behavior.

The detection of merging failure is little misleading. The `hadd` always exits with the zero code and prints all information including error messages to standard output. If the standard output and the error output points to the same file, nothing is printed to error output and only to standard input is printed. But if the standard output and the error output points to different files, all information is still printed to the standard input and only error messages are

printed to the error output. So I decided to check if the error output contains the word error, which is on the beginning of all `hadd` error messages.

3.3 Uploading

The ultimate goal of this thesis is to make a program that automatically uploads the Timepix data to the WLCG. The files should not be uploaded unless there is a reason for it otherwise it is just waste of time and resources. Such reasons are that file is not yet uploaded or that the content of the file or content of one of the files has changed. The detection that the file is not yet uploaded is easy, it suffices to list the files on WLCG and check that the file there. The detection that the content of the file or content of one of the files has changed is not so easy. The file could be downloaded but that is inefficient, moreover, if the file is a merger there is no easy way how to check if it is up-to-date.

To check if a data file is up-to-date, I decided to create checksum files. To use checksum files one must choose a hash function creating sums. There are various hash functions to choose from, the standard ones such as the MD4 and MD5 or various types of the SHA, or the less know hashes provided by the tool `rhash`. I chose the SHA-512 because it is known standard with widely spread support and also because of the length of the output. It is the 512-bit long digest, which provides a lot of possibilities and thus small chance of collision. A collision of checksums is critical because then there is no way how to distinguish if a content of the file or content of one of the files it is merged from has been changed, except downloading the file. Another bonus of using the SHA-512 is that there is the tool `sha512sum` which enables to easily create and check the checksum files.

When a list of files is passed to the `sha512sum`, it prints pairs of a checksum and a file name. If the output is stored in a file, it can be used later to check that the files have not been modified. This can be done using options `-c` or `--check`. It returns the code 0 when no file from the list has been modified and the code 1 when some of them do. I decided for one checksum file per one uploaded file so it can be easily checked if the file has been modified. The checksum files for per day files contain a single checksum of that file, but the checksum files for the mergers of the per hour files contains checksums for every per hour file it is merged from.

Once the file is uploaded to the WLCG it cannot be modified. At the beginning of the upload, the record about the file is added to a Rucio catalog. The record contains the DID, the GUID, file size, and Adler-32 checksum. None of these can change, so once the record is created only the file with the same content can be uploaded, otherwise, the file size or the checksum do not match. But when a data file is modified there is a need to re-upload it or its merger file, so I decided to use a specific naming convention. The date of the

upload is appended at the of the DID.

How I mentioned in section 1.2, the DID is composed of a scope and a file name delimited by a colon. A user without any special privileges can write only to his own scope. The user scope is a string `user.<USER>`, where the `<USER>` is a username of the user. When the scope is on the beginning of a filename and the scope is not provided, the Rucio guess it by the filename and it makes better compatibility with similar older software, Don Quijote 2. So the final DID of a file looks like `user.<USER>:user.<USER>.<YEAR>_<MONTH>_<DAY>_TPX<TPX>.root.<DATE>`, where the `<USER>` is the username, the sequence `<YEAR>_<MONTH>_<DAY>` is a date of the origin of the data, the `<TPX>` is a number of a detector where the data were measured and the `<DATE>` is the date of the upload.

Because there is a lot of files in the scope and it is not trivial to filter from them the ones with the highest number (the latest date) at the end, I decided to create a dataset for every combination of a year and a detector number. The datasets, unlike the files, can be modified, so the files can be attached and detached as you wish. In these datasets are attached only the latest files. I chose a similar naming convention for the dataset as for the files. The dataset DIDs look like `user.<USER>:user.<USER>.<YEAR>_TPX<TPX>`, where again the `<USER>` is the username, the `<YEAR>` is a year of the origin of the data and the `<TPX>` is a number of a detector where the data were measured.

The checksum files are stored also on the WLCG and they are also attached to datasets to make easy to identify and manipulate with the latest files. The checksum files have their own datasets. These are similar to those ones to which data files are attached. There is one checksum file per one uploaded file, so the content of the datasets of checksum files is almost the same as the content of the datasets of the data files, but instead of data files, the corresponding checksum files are attached to them. I also decided to use a similar naming convention. The DIDs of the datasets of the checksum files are equal to the DIDs of the corresponding datasets with data files with the suffix equivalent to the hash type so in this case the `.sha512`. So the DIDs of the datasets of checksum files looks like `user.<USER>:user.<USER>.<YEAR>_TPX<TPX>.sha512`, where again the `<USER>` is the username, the `<YEAR>` is the year of the data origin and the `<TPX>` is a number of a detector. I decided to use the same naming principle to checksum files, so the DID of a checksum file is equal to the DID of corresponding data file with the suffix `.sha512` appended.

The script has two mandatory arguments and one pair of arbitrary arguments. The mandatory arguments are the same as at the script `tpx_merge` explained in section 3.2 and their validation process is also the same. That means that they are a year and a detector number specifying the data to work with and that the process is also the same as in script `tpx_path` which is described and explained in section 3.1. The arbitrary pair of arguments is a month and

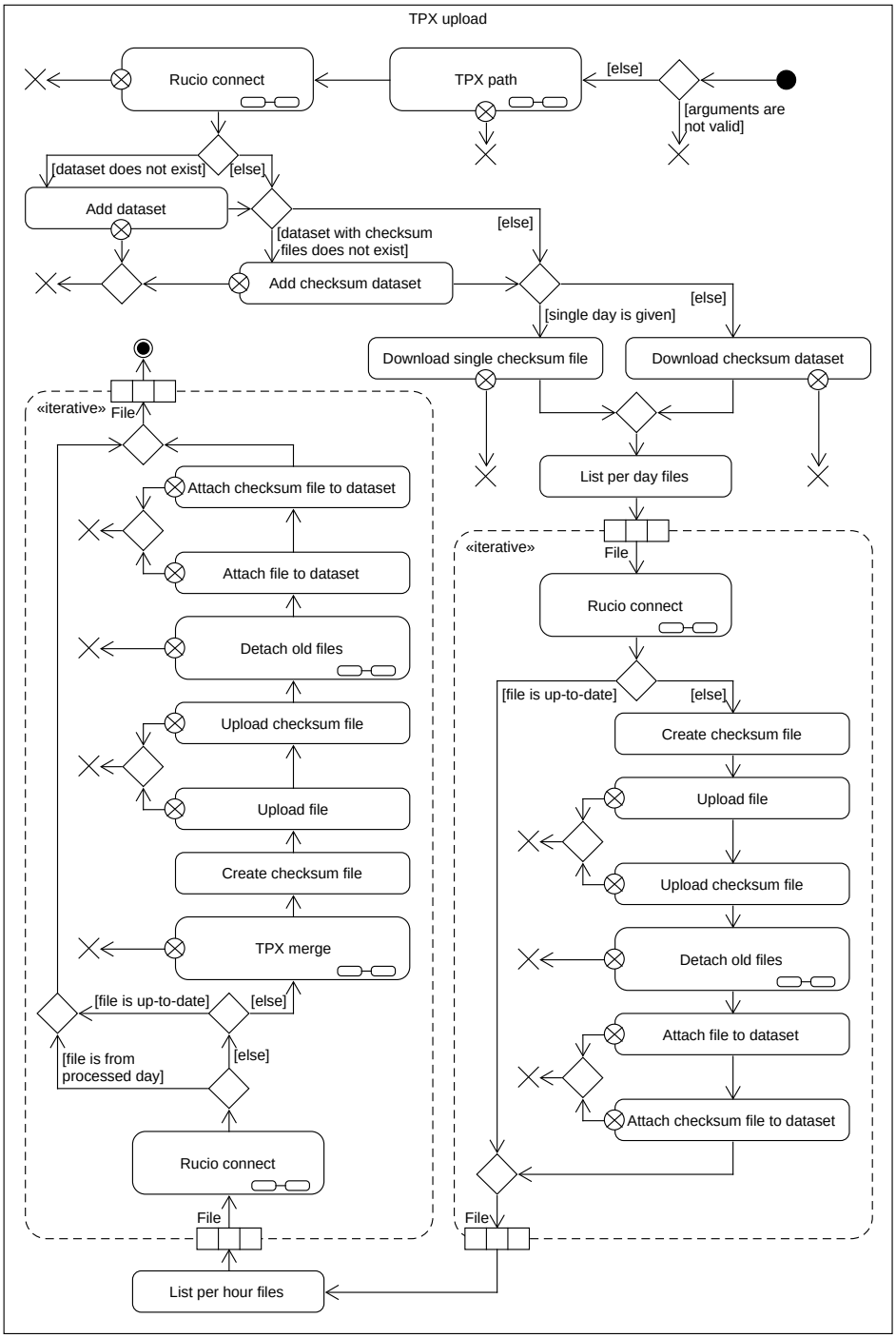

Figure 3.7: Process of uploading to the WLCG

a day specifying data from the single day. These have to be one or two digit numbers, otherwise, the arguments are not valid. If the arguments are not valid, the script exits with the code 40 and prints the help message to the error output. Instead of mandatory and arbitrary arguments, only one argument can be provided. This argument is the `--help` option. If this argument is provided, it means that a user is asking for help. In such case, the script exits with the zero code and prints the help message to the error output.

Then the data path is retrieved using the script `tpx_path`. If the script exits with the code 40, the current script exits with the code 40, too, and prints the help message to the error output. However, this should not ever happen, because both scripts use the same validation techniques. If the script exits with the code 44, the error message from the script is printed to the error output together with the message "`Try 'tpx upload --help'`" and the upload script exits with the code 44 as the script `tpx_path` does.

Because the script needs to use the Rucio and using it communicate with the WLCG. The Rucio clients and the VOMS proxy must be set up. This is done using the script `rucio_connect` which is described and explained in section 3.4. If it fails, that is if the user enters an invalid passphrase, the error message from the script `rucio_connect` is printed and the script exits with the code 50.

Then the existence of the dataset of data files and the dataset of checksum files is checked. If one of them does not exist, it is created. If the creation fails, the script exits with the code 50 and prints error message, which tells that the dataset and which dataset could not be created, to the error output. Since after the check it is sure that both datasets exist, the script can start to use them.

The next step depends on the argument passed to the script. If a single day is specified it checked if a corresponding checksum file is attached to the dataset of checksum files. If it does it is downloaded and the scope is parsed out. Is it so because the script is prepared for a future possibility of ownership transition made by the script `tpx_claim`. If the download fails, the script exits with the code 50 and prints an error message, which tells that the checksum file could not be downloaded. If it is downloaded successfully, it is downloaded into the temporary directory into a directory with the name of the scope of the checksum file. The directory is then renamed to the name of the dataset, so it can be used the same way as if the whole dataset is downloaded.

If a single day is not specified the dataset of checksum files is simply downloaded as a whole. If the download fails, the script exits with the code 50 and prints error message, which tells that the dataset could not be downloaded. The files are downloaded into the temporary directory into the directory with the name of the dataset, that is the DID without the scope part.

When the checksum files are downloaded, everything is prepared to start iterating over the files and evaluating if there is a need to upload them and eventually upload them. This is done in two phases, the first phase goes

through the per day files and the second one through per hour files. If data from some days are stored in both, the per day files and the per hour files, the per day files have higher priority and the per hour files are not merged and neither uploaded.

In the first phase, the per day files are listed. If the single day is specified, the list contains only the specified file, but if data from that day are stored only in per hour files, that means no per day file exists for the specified day, the list is empty. If the single day is not specified the content of the directory retrieved using the script `tpx_path` is filtered and per day files are stored in the list.

Then the script starts iterate over the files in the list. Because there can be up to hundreds of files containing terabytes of data, each iteration is ensured that the VOMS proxy is not yet expired and will not happen during the upload. This is again done using the script `rucio_connect` described in section 3.4. Because the environment is already set up by the first invocation, it skips the setting up part and only works with the VOMS proxy. If the proxy for the user does not exist or it is expired or less then hour has left until expiration, it is re-initialized.

Each file is checked if it is up-to-date or not. It is important to re-upload the file if the uploaded one is obsolete. So I decided to use strict criteria to consider it up-to-date. There are five of them. They can be seen if the figure 3.8. To consider a file up-to-date, the flow must go through one of the "check the checksum" activity and have a positive output from it.

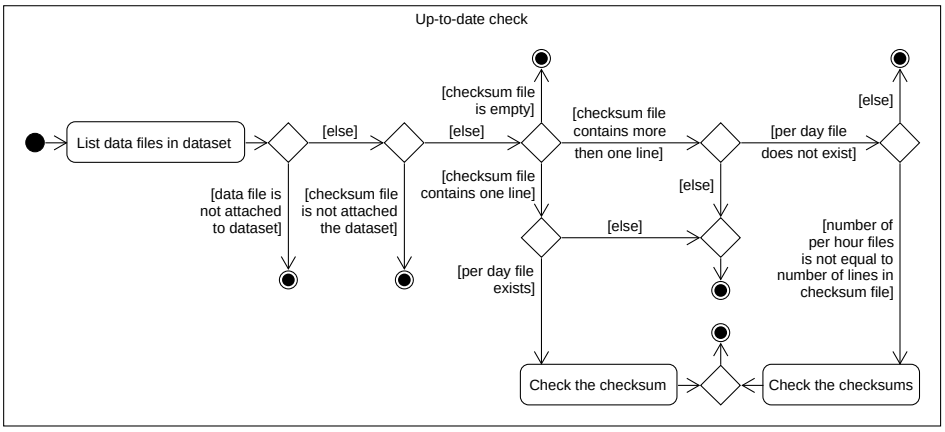

Figure 3.8: Process of checking if a file is up-to-date

The first one is of course that the file is attached to the dataset, otherwise, it is considered it is not even uploaded. The second one is that the corresponding checksum file is attached to the corresponding dataset, otherwise, there is an inconsistency between the dataset of data files and the dataset of checksum

files and it has to be fixed. A certain way how to fix it is to re-upload the file.

The third one is that the checksum file is not empty, otherwise, there is something wrong about the file and it has to be fixed and again a certain way how to fix it is to re-upload the file. In the cases, when the checksum file is missing or it is empty, is highly probable that there is a problem with the WLCG and it might be also possible that there is a problem with the data file, too, and it is unnecessary to download it and it could be not even possible to download it, for example when the file replicas are lost. Therefore the certain and easy way is to re-upload the files, the data file, and the checksum file.

The fourth criteria is that if the checksum file contains only one line, the per day file exists and its checksum matches, because if the checksum file contains only one line, it means that when the file was uploaded the per day file was present, and if it does not exist anymore, the file structure of data from that day has changed and it cannot be reliably checked if the data are up-to-date.

The fifth one is that if the checksum file contains more than one line, the per day file does not exist, the number of lines in checksum file is equal to a number of per hour files and their checksums matches. The reason for the non-existence of the per day file is analogical to the reason of the previous criteria, but with the difference not only that it cannot be reliably checked if the data are up-to-date, but without the download, it is not even possible. If the number of lines in the checksum file does not equal to the number of per hour files, it means that the merger file cannot be the same, because there are more, or fewer files than there were earlier and no matter the type of the change the result cannot be the same. If in the fourth or in the fifth criteria the checksums do not match, it means that some of the files were modified and so it has to be re-uploaded.

If the iterated file is up-to-date, the day it is from is just marked as processed and a next file goes on a turn. The more interesting is if the file is not up-to-date. Then the checksum file is created and upload starts. Because from the user perspective it is more important to have uploaded the data file than the checksum file, which is there only for checking purposes, as the first the data file is uploaded and the checksum file is uploaded afterward. If one of the uploads fails, the script immediately exits with the code 50 and prints an error message, which tells that the file and which file could not be uploaded, to the error output. If the upload of the data file fails, the upload of the checksum file does not even start, because it is gratuitous.

When both files are uploaded, there are just a few operations left to do. They are to detach old files from the datasets and attach the new ones. These file manipulations happen purely on the WLCG, but first must be determined on exactly which data the operations have to happen or if they even have to happen. The situation when the operations should not happen occurs during the detachment of old files. As seen in figure 3.9, a situation when no file is detached from any dataset can occur.

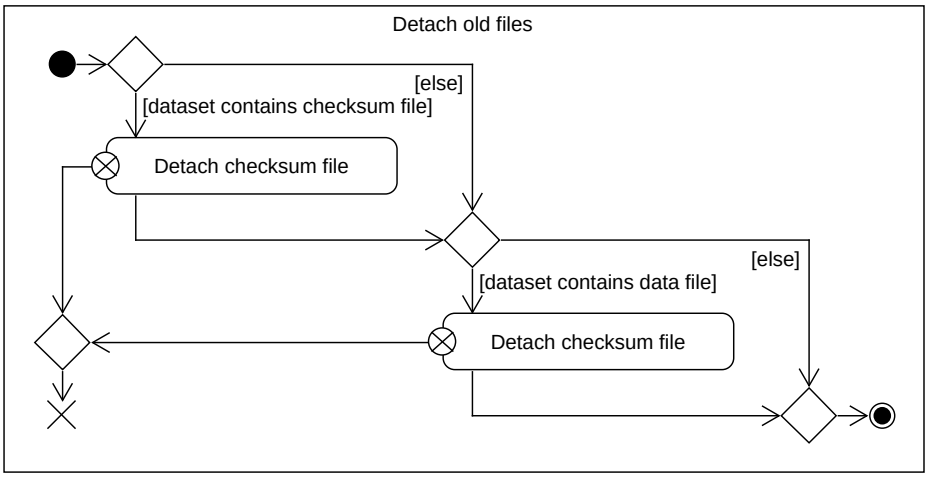

Figure 3.9: Process of detaching old files

During the detachment, data files attached to the dataset are listed. If the list contains file from the same day as was just uploaded. It is detached from the dataset. The same happens even for the checksum files. The important thing is that both steps are done independently of each other, so if there were some inconsistencies between the data files and the checksum files, in this step they are fixed. The checksum is detached as the first for similar reasons as it is uploaded as the second. From the user perspective, it is better to have at least old data file rather than no data file. However, if one of the detachments fails, the process of the detaching of old files is immediately interrupted. If it happens so, the script exits with the code 50 and prints an error message to the error output, which tells that the files and which files could not be detached from which dataset but not exactly which file, if the data one or the checksum one.

After the detachment, the newly uploaded files can be attached to the datasets and so they are. Again, the data file has priority to be attached. If one of the attachments fail, the script exits with the code 50 and prints an error message to the error output, which tells that the file and which file could not be attached to which dataset. At the end of each iteration, the checksum file is deleted so it does not waste a space.

Almost the same iteration process is done for per hour files. There are just a few differences, these can be seen in figure 3.7. First one is that before any operation as the first thing each iteration it is checked whether the file is from some already processed day. If it does, nothing happens and a next file goes on the turn. If it is not from an already processed day, the day it is from is marked as processed. The second difference is when the uploaded file is not up-to-date, the per hour files from the same day are merged together using

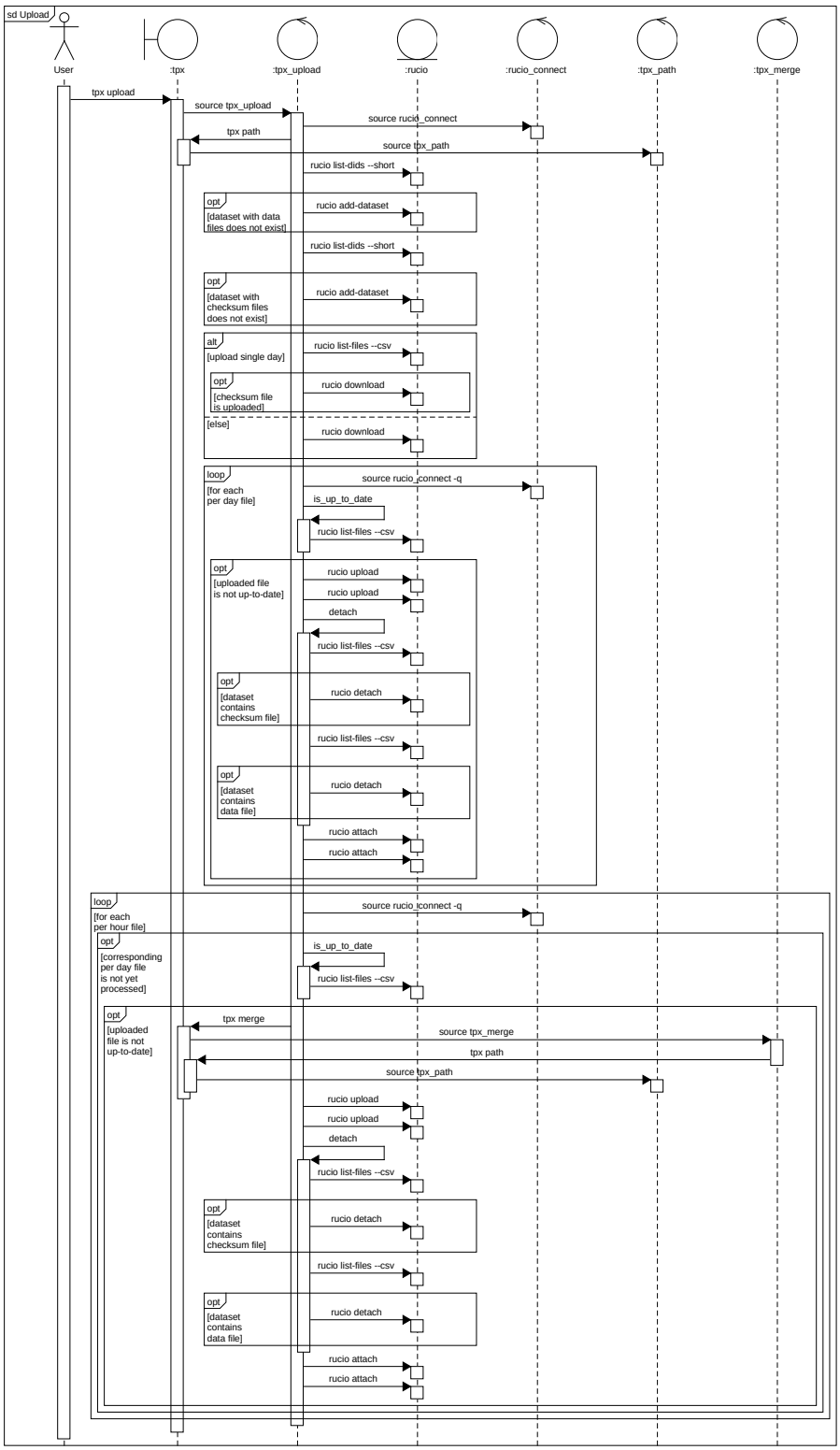

Figure 3.10: Invocations during the upload

the script `tpx_merge` described in section 3.2, and the resultant merger file is stored in the temporary directory. The third difference is that the checksum file is not created from the merged per day file, but from the per hour files, which it is merged from. The fourth and the last difference is that at the end of each iteration not only the checksum file is deleted but the merger file is deleted, too.

The uploading script is the biggest one and the most complex one of the scripts in the bundle. As seen in figure 3.10, it also uses all of them, except the `tpx_claim` (that is why it is not part of this thesis). There is also seen how much it communicates with the Rucio. It seems like the uploading script uses the other one only rarely, but the bodies of the loops can be executed hundreds of times. The uploading loops can have even thousands of iterations, but because there is one iteration per every per hour file and only the per day files are uploaded, most of the iterations are just skipped.

The numerous communication cannot limit the functionality. However, every communication with the Rucio is a network communication, so it has enormous overhead, not mentioning that during every request the Rucio has to work with enormous data, where terabytes of data are stored in form of records. For example, a listing of files takes up to few seconds and uploading of a small simple file such as the checksum one can take up to few minutes. The same applies for the download. That is one of the reasons why I tried to use so few uploads and downloads as far as it is possible. The problem is not in the volume of the data but in the overhead of the communication and Rucio.

There is also another problem associated with using the Rucio. I do not know how it works elsewhere but the Prague site is uncomfortably unstable. The Prague site is part of the WLCG managed by Institute of Physics of the Czech Academy of Sciences [17, 18]. Every moment some service necessary to upload or download the files is not available. The problem is that if some necessary service is not available, the upload fails and thus the uploading script exits. So during the initial upload, until the all currently available data are uploaded, it has to be re-invoked many times, so it could finish. Each invocation part of the directory is uploaded. I recommend to re-invoke it till it returns the zero code.

3.4 Setting up Rucio clients

The guide [8] recommends using three commands in specific order to set up Rucio clients and initialize VOMS proxy. These three commands are `setupATLAS`, `localSetupRucioClients` and at the end `voms-proxy-init -voms atlas`.

The command `setupATLAS` is just an alias for command `source /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/user/atlasLocalSetup.sh` wh-

ich sets up an environment for cluster user. It does so by using setting aliases and variables. One such alias is the `localSetupRucioClients`, which is an alias for `source /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase /utilities/oldAliasSetup rucio`. The command is used to set up anything with correct dependencies, in this case, the Rucio clients. It also modifies the `PATH` variable and so makes the VOMS proxy accessible. After that, the command `voms-proxy-init -voms atlas` can be executed. It creates a proxy and includes an AC containing user attributes in the proxy certificate. The proxy has a default lifetime of 12 hours, which can be extended to 24 hours.

Because the upload should start by a single command, the setups and initialization of the VOMS proxy have to be automatic, but re-invoking of the setups might cause problems and re-initializing of the VOMS proxy is needless, so I created a Bash script which automatically decides which steps need to be done and which do not.

The script at first exports the variable `ATLAS_LOCAL_ROOT_BASE`, which represents the root base of the repository "atlas.cern.ch", that is located at path `/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase`. The variable is required by following setups. After the export, it checks if the necessary tools are available. These tools are `rucio`, `voms-proxy-init` and `voms-proxy-info`. If one of them is not available, the Rucio clients must be set up. But before this setup, the ATLAS environment must be set up. The good identifier whether the ATLAS environment is set up is to check whether the variable `ALRB_availableTools` is set. This variable is set during the ATLAS setup, so if it is not set the command `source $ATLAS_LOCAL_ROOT_BASE /user/atlasLocalSetup.sh` is invoked.

The Rucio clients are not set up by `source $ATLAS_LOCAL_ROOT_BASE /utilities/oldAliasSetup.sh` as they are in the guide, because the script just removes arguments equal to two dashes and the rest passes as arguments to another command. So I decided to skip the `oldAliasSetup.sh` step and invoke directly the command `source $ATLAS_LOCAL_ROOT_BASE /packageSetups/localSetup.sh rucio`.

If the Rucio clients are set up the VOMS proxy comes into play. The VOMS proxy is initialized only when there is no VOMS proxy present for the user or when the proxy expires in less than hour or it is already expired. These checks are done using the `voms-proxy-info`. It fails if there is no VOMS proxy present, otherwise, various information is printed. One such information is time left to expiration. The VOMS proxy initialization needs a passphrase for the user certificate. If a file `~/.globus/passphrase` exists and its permissions are `400` or `600`, its content is used as the passphrase, otherwise, the user is asked to enter the passphrase. If the user enters an incorrect passphrase the script exits with a code 1. If the user wants to upload a lot of data, I recommend to save the passphrase in the file, so the VOMS proxy can be automatically re-initialized when there is the need.

In the end, if the script is not launched in quiet mode, it prints time left to the VOMS proxy expiration and path to the proxy. These are information provided by `voms-proxy-info`.

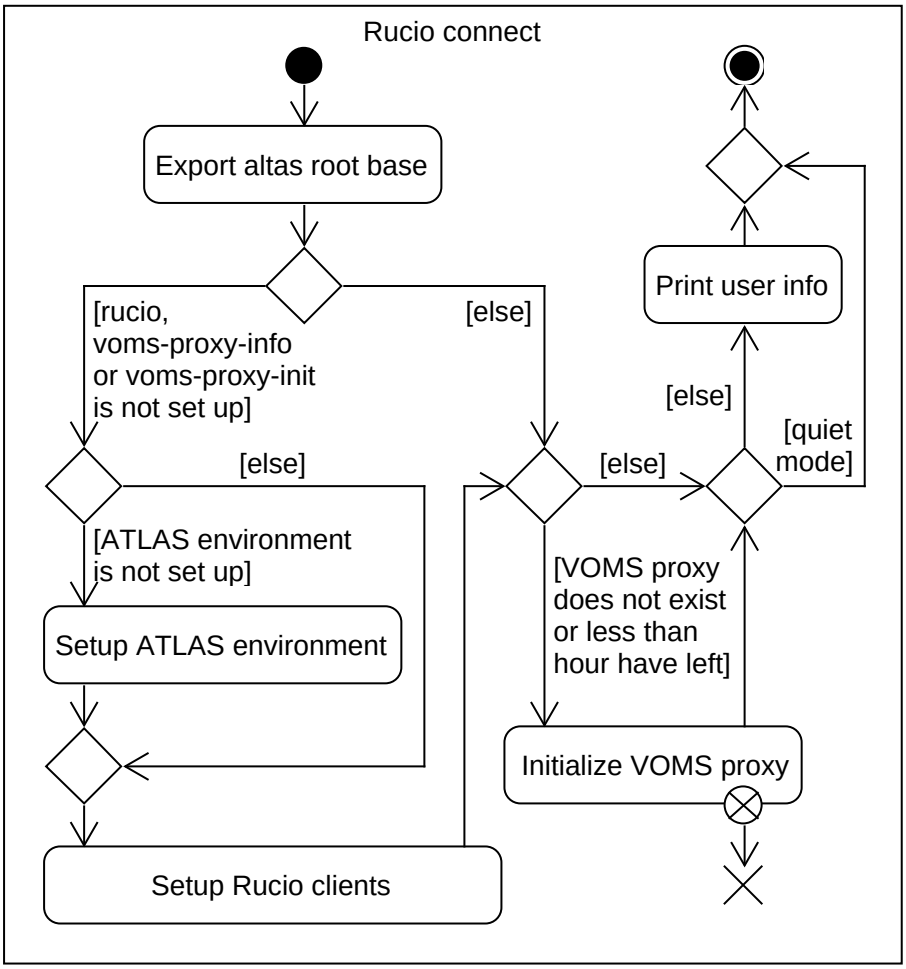

Figure 3.11: Decisions made during setting up the Rucio clients

Conclusion

This thesis had three individual goals. The first goal was to get an overview of what WLCG and Rucio are and how to use them to accomplish following goals. The second goal was to find a way how to connect from IEAP server to the WLCG and manage data on it using Rucio middleware. The third and the ultimate goal was to analyze processes, that need to be done, so the data from IEAP server can be uploaded to WLCG, and to design and implement a program that transfers and store the data from the IEAP server to WLCG.

The fulfillment of the first goal is proved by the chapter 1 and by the fact that I successfully accomplished the next two goals. The second goal is accomplished using the technology of the CernVM-FS, which provides software I use to connect to the WLCG using Rucio. The software is Rucio clients and the VOMS proxy used by the Rucio to authorize. I accomplished the third goal by design and implementation of a bundle of Bash scripts, which locates the data files, merges the per hour files into the per day merger files and uploads the per day files onto the WLCG.

The use of the bundle during the upload of 2015, 2016 and 2017 data, which have together about 13.7 terabytes, prove that the bundle works. However, there are always things to improve. The bundle could be optimized, so it does fewer operations and fewer iterations during the upload. The reasons of fails of the upload could be distinguished and decisions, if to try again or quit, could be made.

Bibliography

[1] Welcome | Worldwide LHC Computing Grid. [online], [Cited 2018/04/17]. Available from: http://wlcg-public.web.cern.ch

[2] Getting Started | WLCG. [online], [Cited 2018/04/18]. Available from: http://wlcg.web.cern.ch/getting-started

[3] VOMS home. [online], [Cited 2018/04/18]. Available from: http://italiangrid.github.io/voms/index.html

[4] De Stefano, J. S. Joining a VO — RACF. [online], Apr. 2013, [Cited 2018/04/24]. Available from: https://www.racf.bnl.gov/docs/howto/grid/joinvo

[5] Welcome to Rucio's documentation! — Rucio 1.2 documentation. [online], [Cited 2018/04/22]. Available from: https://rucio.readthedocs.io/en/latest/

[6] Files, Datasets and Containers — Rucio 1.2 documentation. [online], [Cited 2018/04/22]. Available from: https://rucio.readthedocs.io/en/latest/overview_File_Dataset_Container.html

[7] Rucio Storage Element — Rucio 1.2 documentation. [online], [Cited 2018/04/22]. Available from: https://rucio.readthedocs.io/en/latest/overview_Rucio_Storage_Element.html

[8] Chudoba, J. RucioClientsHowTo < AtlasComputing < TWiki. [online], Aug. 2017, [Cited 2018/04/22]. Available from: https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/RucioClientsHowTo

[9] Replica management with replication rules — Rucio 1.2 documentation. [online], [Cited 2018/04/22]. Available from: https://rucio.readthedocs.io/en/latest/overview_Replica_management.html

[10] Accounting and quota — Rucio 1.2 documentation. [online], [Cited 2018/04/22]. Available from: https://rucio.readthedocs.io/en/latest/overview_Accounting_and_quota.html

[11] Rucio Clients — Rucio 1.2 documentation. [online], [Cited 2018/04/18]. Available from: https://rucio.readthedocs.io/en/latest/clients.html

[12] VOMS Client Guide. [online], [Cited 2018/04/18]. Available from: http://italiangrid.github.io/voms/documentation/voms-clients-guide/3.0.3/

[13] Overview — CernVM-FS 2.4.3 documentation. [online], [Cited 2018/04/18]. Available from: http://cvmfs.readthedocs.io/en/stable/cpt-overview.html

[14] CernVM-FS Client Quick Start | cernvm.web.cern.ch. [online], [Cited 2018/04/18]. Available from: http://cernvm.cern.ch/portal/filesystem/quickstart

[15] Manzi, A. WLCGCVMFSGridArea < LCG < TWiki. [online], Mar. 2016, [Cited 2018/04/25].

[16] Client Configuration — CernVM-FS 2.4.3 documentation. [online], [Cited 2018/04/25]. Available from: http://cvmfs.readthedocs.io/en/stable/cpt-configure.html

[17] ATLAS Grid Information System. [online], Feb. 2018, [Cited 2018/05/11]. Available from: http://atlas-agis.cern.ch/agis/site/detail/praguelcg2/

[18] Welcome to our website | Fyzikální ústav Akademie věd ČR. [online], [Cited 2018/05/11]. Available from: https://www.fzu.cz/en

Appendix A

Acronyms

ÚTEF Ústav technické a experimentální fyziky.

AC Attribute Certificate.

AFS Andrew File System.

API Application Programming Interface.

ATLAS A Toroidal LHC Apparatus.

Bash Bourn Again Shell.

CERN European Organization for Nuclear Research.

CernVM CERN Virtual Machine.

CernVM-FS CernVM File System.

CTU Czech Technical University in Prague.

DID data identifier.

GUID globally unique identifier.

HTTP Hypertext Transfer Protocol.

IEAP Institute of Experimental and Applied Physics.

LFN Logical File Name.

LHC the Large Hadron Collider.

MD Message-Digest algorithm.

NFS Network File System.

PEM Privacy-enhanced Electronic Mail.

PKCS Public-Key Cryptography Standards.

REST Representational State Transfer.

RSE Rucio Storage Element.

SHA Secure Hash Algorithm.

VO Virtual Organization.

VOMS Virtual Organization Membership Service.

WLCG Worldwide LHC Computing Grid.

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `tpx_bundle` ..... the directory with the bundle of scripts
  - `tpx` ..... the script covering up the whole bundle
  - `tpx_path` ..... the script locating data files
  - `tpx_merge` ..... the script merging per hour files
  - `tpx_upload` ..... the script uploading data to the WLCG
  - `tpx_claim` ..... the script for ownership transition of datasets
  - `tpx_config` ..... the parent script for all scripts above
  - `rucio_connect` ..... the script setting up environment and proxy
- `thesis_src` ..... the directory of LaTeX source files of the thesis
  - `*.pdf` ..... the thesis figures
  - `*.tex` ..... the LaTeX source code files of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format
  - `assignment.pdf` ..... the thesis assignment