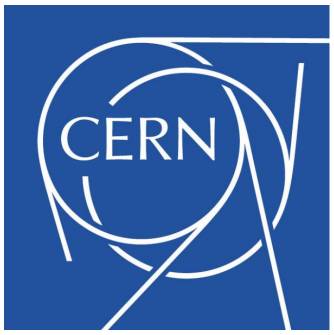

European Organization for Nuclear Research

Summer student report

EOS quota node visualization and permissions analysis

Author:
Pieter Robyns

Supervisor:
Luca Mascetti

2013

Contents

1 Introduction

In this report, I will give a brief description of the concepts and motivation for the assignments I had to complete during my summer student internship at CERN. The idea is to focus more on the *what* and *why* of the assigments instead of the *how*[1].

The main goal of my project was to create manageability tools for EOS, a multi-petabyte disk storage system based on the Scalla / Xrootd framework which is being used by all 4 of the LHC experiments for physics data.

One of the first tasks was to create a web visualization for EOS quota nodes. This subject will be covered in Section 2. The second part – analyzing permission configurations on EOS – is described in Sections 3 and 4. In the third part I implemented a file history view using Hadoop, which is explained in Section 5.

Finally, in the Appendix (Section 7), there will be some screenshots of the developed tools in action.

2 Quota visualization

Operators of the EOS system can assign a quota to certain users or groups in order to limit their space usage on the system. This section explains the first part of my work: the visualization of these quotas.

2.1 Using eos quota ls

EOS provides a command line tool `/usr/bin/eos` which allows users or operators to execute commands on the storage system. To view the assigned quota nodes per namespace for example, operators can execute the command `eos quota ls`. This presents a human readable representation of each quota node like in Listing 1.

Though this data is complete, it is hard to have a clear overview of all quota nodes. Furthermore, it is not immediately obvious how much available space a user has left. For example, in Listing 1 the user `user1` actually has `500.00 GB` available space instead of the `0.00 B` indicated in the row. Finally, the data presented is static and cannot be sorted or manipulated easily.

For the above reasons, the operators of the system needed a tool to convert this data into a more graphical representation. To accomplish this I created a Python script that parses the quota data, generates `html` files containing Google Charts API charts [4] and uploads them to a webserver.

[1] A detailed description on how to use the tools and visualizations I made during my stay can be found on the CERN TWiki[1].

2.2 Parsing the quota data

In order to easily parse the quota data, we need an appropriate format to do so. The functionality for generating this kind of format is conveniently already implemented in EOS: we can execute the command `eos quota ls -m`. The output is shown in Listing 2.

In this format, we have a collection of key value pairs per quota entry and each entry is seperated by a newline. We can thus easily parse all values for the namespaces and store them in memory to perform more complex operations such as lookups, building a hierarchical tree structure, sort, select a top 5, etc.

2.3 Types of visualizations

Several quota data visualization types were created using the Google Chart API: tree map, table view, pie chart, bar chart, per user quota visualization and tree view. Each of them has a different use case or appearance and will be briefly explained in the following subsections.

2.3.1 Tree map

The tree map (Figure 5) builds a tree of the quota data and displays this tree in a heatmap like fashion. Each node size is calculated by recursively summing the available quota space of the node's children. The color of the node is calculated based on the percentage used quota space.

An operator can go down or up in the tree by left or right clicking respectively. Upon clicking, a bar and pie chart will be drawn (see Section 2.3.3 and Section 2.3.4).

This view can be useful for getting an idea of the general allocated quota space under a certain part of the namespace.

2.3.2 Table view

Contrary to the tree map, the table view (Figure 6) does not recursively sum the sizes of all children in the tree, but displays the allocated quota for each namespace individually. Therefore, this view is more useful for inspecting single quota node configurations.

Analogous to browsing a directory tree, the user can browse through the namespace. Upon clicking a node, a bar and pie chart will also be drawn.

2.3.3 Pie chart

The pie chart (Figure 4) is displayed when a user clicks on a tree map or table node. This view shows the top 5 quota consumers in the namespace, together with the

share of all other consumers and the total available free space left in the quota node.

2.3.4 Bar chart

The bar chart (Figure 3) is also displayed when a user clicks on a tree map or table node. It displays the top 5 quota consumers, where each user quota is divided in *used space*, *available space* and *exceeded space*. The latter is only shown when the user exceeds a user or group quota.

2.3.5 Per user quota visualization

In *per user quota visualization* (Figure 10), one can enter a username and get all namespaces in which this user has a quota. Aside from the *used space*, *available space* and *exceeded space* the *used number of files*, *available number of files* and *exceeded number of files* are also displayed.

2.3.6 Tree view

The tree view (Figure 7) shows the whole namespace tree at once. Though this view is not always handy because the tree could become too large for instances such as ATLAS, it can be useful to quickly find a child node quota without having to traverse the tree in smaller instances. In this case we can also easily spot anomalies in the quota node hierarchy.

2.4 Results

The resulting charts and scripts are located in the EOS operations folder in AFS. The chart output can be viewed by accessing EOS Cockpit [2] and clicking on "Quotas". In order to generate new charts from a set of quota key value pairs, the following command can be issued:

`python parseeosquota.py <md_path> <website_path>`

Where `<md_path>` is the path to the metadata file and `website_path` is the root document directory of the web server.

3 Eos-log-print tool

The second assignment during my stay at CERN was to correct the various permission misconfigurations made by users in EOS. These misconfigurations are potential security vulnerabilities as data may get accidentally or maliciously deleted (see Section 4).

However, checking the permissions on a live system by actively scanning directories may unnecessarily put extra load on the system as the whole directory tree needs to be searched and analyzed. For this purpose, the eos-log-print tool had to be developed.

The tool analyzes a binary metadata file (∼5 GB in size) made by the system and generates output in human readable text. Aside from solving the above mentioned problem, this is also useful for operators when a quick manual check has to be done on a file or directory: the operator can use standard Unix tools such as `grep` or `sed` to extract data from the logs.

Usage of the tool is straightforward: `eos-log-print <logfile> <destination>`. Optional parameters can be provided such as `-o {<key> <key2> . . . }` to limit the output to a set of keys and `-f {<key1>=<value1> <key2>=<value2> . . . }` to filter on records where $key_n = value_n$ is true for all n provided key value pairs. Finally, one can add the `-s` flag to prevent the full path to a directory or file from being constructed, reducing execution time.

4 Permissions analysis

Now that the `eos-log-print` generated a parsable file containing the full path, flags, mode and ACL values for each directory, we can use this file to check for permission errors. In particular we check the mode and ACL values of all directories. Typically the number of directories is in the order of millions, depending on the experiment.

4.1 EOS permission system

The EOS permission system is based on a combination of ACLs and Unix permissions [3]. Users can configure a Unix-like mode by executing `eos chmod <mode> <path>`

Listing 1: `eos quota ls`

```
# ------------------------------------------------------------------------------------------------------
# ==> Quota Node: /eos/atlas/atlascerngroupdisk/data-prep/
# ------------------------------------------------------------------------------------------------------
user  used bytes logi bytes used files aval bytes aval logib aval files filled[%] vol-status ino-status
user1 62.61 GB   31.31 GB   47.47 k-   0.00 B     0.00 B     0.00 -     100.00    ignored    ignored
# ......................................................................................
group used bytes logi bytes used files aval bytes aval logib aval files filled[%] vol-status ino-status
zp    62.61 GB   31.31 GB   47.47 k-   1.00 TB    500.00 GB  2.00 M-    6.26      ok         ok
# ------------------------------------------------------------------------------------------------------
```

Listing 2: `eos quota ls -m`

```
quota=node uid=user1 space=/eos/atlas/atlascerngroupdisk/data-prep/
usedbytes=62613498375 usedlogicalbytes=31306735092 usedfiles=47465 maxbytes=0
maxlogicalbytes=0 maxfiles=0 percentageusedbytes=100.00 statusbytes=ignored
statusfiles=ignored

quota=node gid=zp space=/eos/atlas/atlascerngroupdisk/data-prep/
usedbytes=62613498375 usedlogicalbytes=31306735092 usedfiles=47465
maxbytes=1000000000000 maxlogicalbytes=500000000000 maxfiles=2000000
percentageusedbytes=6.26 statusbytes=ok statusfiles=ok
```

on a directory. The `<mode>` field consists of three digits representing *owner*, *group* and *other* permissions respectively. The digit itself is a decimal representation of the *read*, *write* and *execute* (browse) boolean flags. For example:

```
7 = 111 -> read, write execute
5 = 101 -> read, execute
4 = 100 -> read
1 = 001 -> execute
```

Secondly, a user can also configure permissions through setting ACLs on the directory level. To do so, the user can set the `user.acl` attribute of the directory like so: `user.acl=<acllist>`, where `<acllist>` should be a comma seperated list of rules `<acllist> = <rule1>,<rule2>...<ruleN>` and where each rule is defined as `<rule> = u:<uid|username>|g:<gid|groupname>|egroup:<name>:<options>`.

4.2 Potential security issues through misconfiguration

Giving the user the freedom to autonomously configure permissions for a directory means that there is a potential window for human error; in particular if the user does not understand how EOS permissions are handled. Furthermore, since new directories inherit permission settings from their "parent" directories, misconfigurations can quickly affect thousands of directories.

4.2.1 Group write permissions

One of the most frequent permission misconfigurations seen in the `ATLAS` and `CMS` instances is the one where the user sets group *write* permissions in the mode or ACL, without specifying the `!d` *forbid deletion* flag in the ACL rule set. Note that depending on the group the user is in, this may imply write permissions for hundreds of users whereas the user only intended to give write permissions to a single colleague belonging to the same group.

Consequently, it takes only one user within the entire group executing an erroneous deletion command (e.g. `rm -r /*`) to delete all data within this folder.

4.2.2 Other write permissions

Even worse than the above, but less frequently occurring, is the case where the user has set the mode of a directory to allow `write` permissions for others. This means that *anyone* can delete or write data to said directory.

Analogous to group write permissions, this increases the chances of an accidental or malicious deletion.

4.2.3 Invalid ACLs

During the inspection of the converted log files, we noticed that there are some users who use an incorrect ACL syntax. This will result in EOS incorrectly or not applying the permission settings. In future releases, it will no longer be possible to set incorrectly formatted ACLs (see Section 4.3), but the previously configured ACLs can still pose a security risk.

Suppose the user wants to forbid deletion for the group `apple` and sets the following ACL rule: `g:<8000|apple>:<w!d>`. This rule would be accepted by the system, but it would not behave as the user expected: neither the `gid` of 8000 nor the `apple` identifier would be matched to the group "apple", so essentially the rule would be dropped! Such misconfiguration may happen if the user literally copy-pastes the ACL syntax specification as displayed on the EOS website [3] or on the manpage of the command.

In a more subtle, but interesting case suppose the user wants to only allow browsing for a group named "spies", while allowing the e-group "friends" to read, write and execute. The user sets the following ACL: `user.acl=g:spies:x;egroup:friends:rw!dx`.

While at the first sight this may seem a correct configuration, note that EOS expects a *comma* seperated list instead of a semicolon seperated list. The incorrect rule will be interpreted by EOS as follows:

```
user.acl = g:spies:x;egroup:friends:rw!dx
-> g:spies:x;egroup:friends:rw!dx (split on ",")
-> g : spies : x;egroup : friends : rw!dx (split on ":")
-> g : spies : x;egroup (take first 3 fields)
-> g : spies : xr (strip invalid characters)
```

The result is that the group "spies" can execute and read, while "friends" can't do anything. This is not what the user intended.

4.3 ACL syntax check

From the previous section we can conclude that there is a need for preventing the insertion of malformatted ACLs into the system and notifying the user when an incorrect ACL was provided. Therefore I added a sanity check to the EOS `C++` code that also returns the appropriate feedback to the user.

I chose to create a regular expression to check the ACL format, as this approach is flexible and compact. Before inserting the ACL, the regex will discard formats that don't match the regex in Listing 3.

Only if the input passes the regular expression check, the attribute will be set for the directory.

4.4 Performing the check

To check for the permission misconfigurations mentioned in Section 4.2, I developed a Python script that reads the directory metadata generated with `eos-log-print`. The check can be performed by issuing the command `python2.6 checkpermissions.py [-h] [--destination DESTINATION] [--prefix PREFIX] [--mode MODE] [--eosserver EOSSERVER] [--eosroot EOSROOT] [--cc CC] logfile` with `mode` equal to `print`.

In summary, the script will generate error or warning reports, e-mails for users and groups and a script containing all commands needed to fix the errors. All these files will be stored in the `DESTINATION` directory.

4.4.1 Notifying users

When the `mode` parameter of the script is set to `mail`, the script will gather all errors for a certain user and notify them about the issues via e-mail. This way the user can solve the issue before any automated fix is applied. The user will consequently be able to understand why a certain permission setting is erroneous.

4.4.2 Automated fixing

The last mode, `fix`, will execute the `FIXES.sh` script generated in the `print` phase. This script automatically fixes all errors found by the permission checker. After this, the operator can convert the metadata again and check if all errors are gone. The operators plan on using this kind of automated fixing if the user does not react to the mails sent in `mail` mode.

4.4.3 Exceptions, customization and policies

Since every instance has its own special directories, ways to address users and policies, we had to think of a way to configure this dynamically.

For special directories which require exceptions to certain permission checks, the operator can provide a newline seperated list of directories in the `permission_exceptions` config file. This can be particularly useful for test directories, which typically have special permissions.

The e-mail sent in `mail` mode can be customized by changing the `emailprefix` and `emailsuffix` config files. These files contain special variables which will be replaced by script generated content:

- `$LOGIN:` Login name of the user
- `$UID:` ID of the user
- `$INSTANCE:` Instance name (ATLAS, CMS, ...)
- `$GID: ID of the group the user is in`

Aside from customizing the format of the e-mail, it's also possible to exclude certain users from being mailed by configuring the `mail_exceptions` file.

Finally, a custom policy can be defined for each instance. If we would use configuration files, this would require a complex query-like language to check for values in the mode and ACL. For this reason, I chose to use a Python module for each policy, because this allows both flexibility and complex checking procedures. For example, to use the ATLAS policy, the operator has to add the line:

```
from atlaspolicy import InstancePolicy
```

Then, a `check(...)` function will be called from the module, which performs the permission analysis as defined by the instance policy.

4.5 Generation of test data

Before we perform corrective actions for the permission settings in production, it is good practice to first perform these actions in a test environment to see whether the issue is fixed correctly.

To do this, I created a script `generator.py` that takes a key value log file as input and generates the entire directory tree from the log file to a specified destination: `generate.py [-h] [--destination DESTINATION] [--eosserver EOSSERVER] path`.

It's now possible to run the permission checker in fix mode (see Section 4.4.2) on the test destination instead of a production environment. This way, we can make sure the automated fixes work as intended.

5 Hadoop file history

In the EOS head nodes and disk servers, every transaction of a file (transfer, login, open, ...) is kept in log files. When operators face a certain issue with the system, it is common practice to use tools like `grep` on these log files to help diagnose the problem. However,

Listing 3: `ACL format regular expression`

```
!^(((((u|g):(([0-9]+)|([\\.[:alnum:]-]+)))|(egroup:([\\.[:alnum:]-]+))):(r|w|wo|x|m|!d|[+]d|!u|[+]u|q|c)+)[,]?)*$
```

when dealing with log file sizes of mulitple gigabytes, this approach becomes infeasible because of the sheer amount of time one has to wait before getting results. For example, on a machine with an Intel® Core™ 2 Duo CPU at 2.40GHz, a `grep` of a 2.6 GB file took 13 minutes and 8 seconds.

To solve this problem, the operators wanted to have a system with the following characteristics:

- Find a specific file's history in a collection of very large log files (multiple GBs of data).
- Complete the find operation in a reasonable amount of time.
- Use only the existing Hadoop architecture.
- Combine log searching with long term archival.

The above concept was implemented according to the following design:

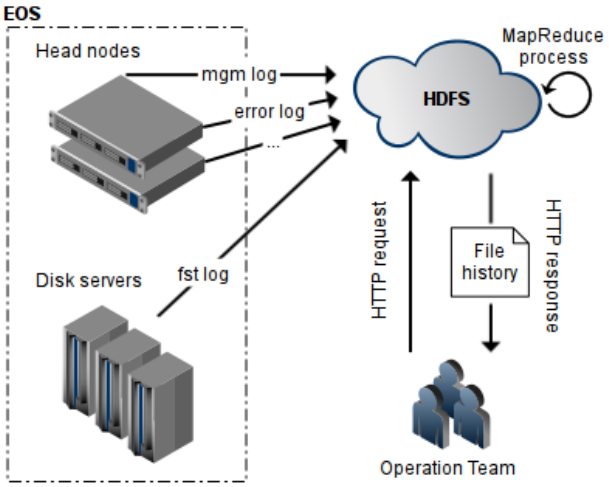

Figure 1: File history view using Hadoop

First of all, the disk servers and head nodes copy all relevant log files to Hadoop periodically using the `hadooplog.sh` script, which is run every day in a cron job.

After that, the operators can execute the `run.sh` tool with as arguments a file name and time range in order to find this file's history. A parallel MapReduce operation will then be performed on HDFS to filter out all relevant log entries for the query. Aside from the `run.sh` tool, operators can also use the web interface on EOS Cockpit to perform queries.

Using the above method, we can indeed combine the long term archival of log files with searching a file's history in a reasonable amount of time. Figure 2 shows a plot of this amount of time in function of the total size of log files being searched. The duration of a query is significantly less compared to the traditional approach. Additionally, the operators can now perform a parallel search over different machines.

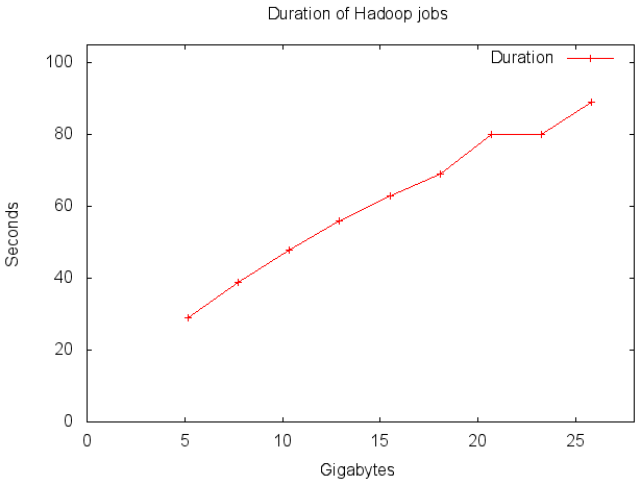

Figure 2: Query duration in function of collective log file size

6 Conclusion

We have seen the importance of data security and examples where things can go wrong. I'm therefore glad to have helped improving this important aspect for the users working in the experiments of the LHC.

Aside from helping the users, the operators now also have a new set of tools to work with; for example the eos quota visualization tool. This will help resolving errors or spotting anomalies more quickly. Another example is the HDFS file history tool, which allows operators to track bugs faster than with the traditional approach.

All in all, I'm happy that I had the opportunity to work with the IT-DSS-FDO team on these projects and I would like to thank my supervisor Luca Mascetti and the EOS team for this experience, their support and their involvement.

7 Appendix

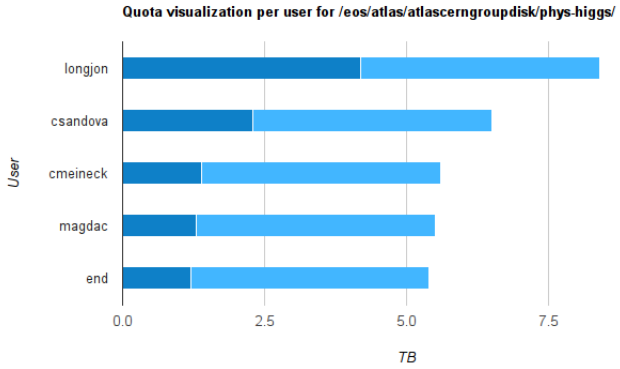

Figure 3: Bar chart

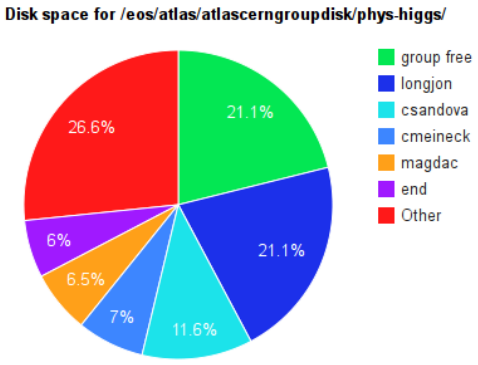

Figure 4: Pie chart

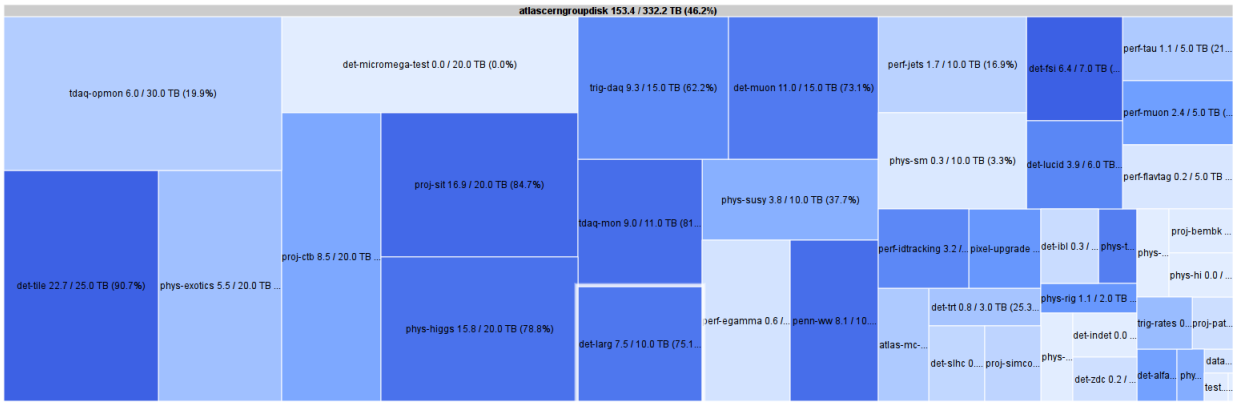

Figure 5: Tree map

Namespace	Used node space (TB)	Total node space (TB)	Percentage
../	0	0	
/eos/cms/store/caf	0	2.5	0.0%
/eos/cms/store/cmst3	0	0	100.00%
/eos/cms/store/eos	0	0	100.00%
/eos/cms/store/express	84.9	100	84.9%
/eos/cms/store/group	770	600	128.3%
/eos/cms/store/lhe	45.2	50	90.4%
/eos/cms/store/relval	278.3	300	92.8%
/eos/cms/store/t0streamer	0	5	0.0%
/eos/cms/store/t0temp	0	5	0.0%
/eos/cms/store/temp	0	0	100.00%
/eos/cms/store/unmerged	0.6	100	0.6%
/eos/cms/store/user	100	400	25.0%

Figure 6: Table view

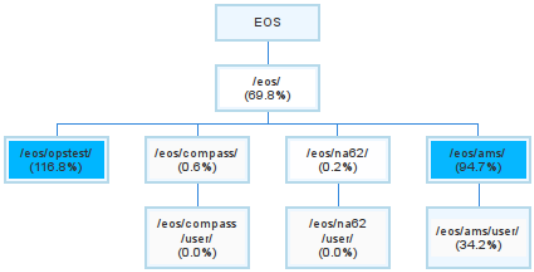

Figure 7: Tree view

```
has following problems: readable by others, writable by other
s!, group writable (mode=42777,sys.acl=u:22014:rw,u:31275:rw,u:5410:rw,usr.acl=)
has following problems: readable by others, writable by others!, group writable (mode=42777,sys.acl=u:22014:
rw,u:31275:rw,u:5410:rw,usr.acl=)
has following problems: readable by others, writable by others!, group writable (mode=42777,sys.ac
l=u:22014:rw,u:31275:rw,u:5410:rw,g:1399:!d,usr.acl=)
```

Figure 8: Permission checker in action

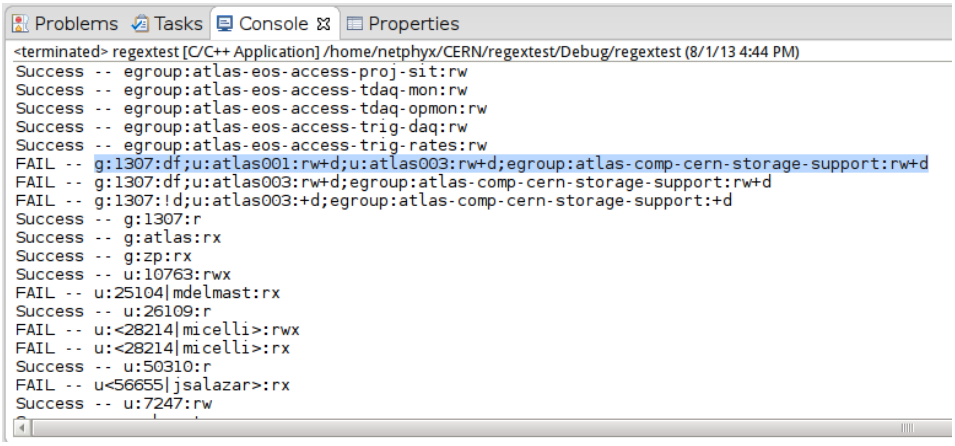

Figure 9: ACL syntax regex in action

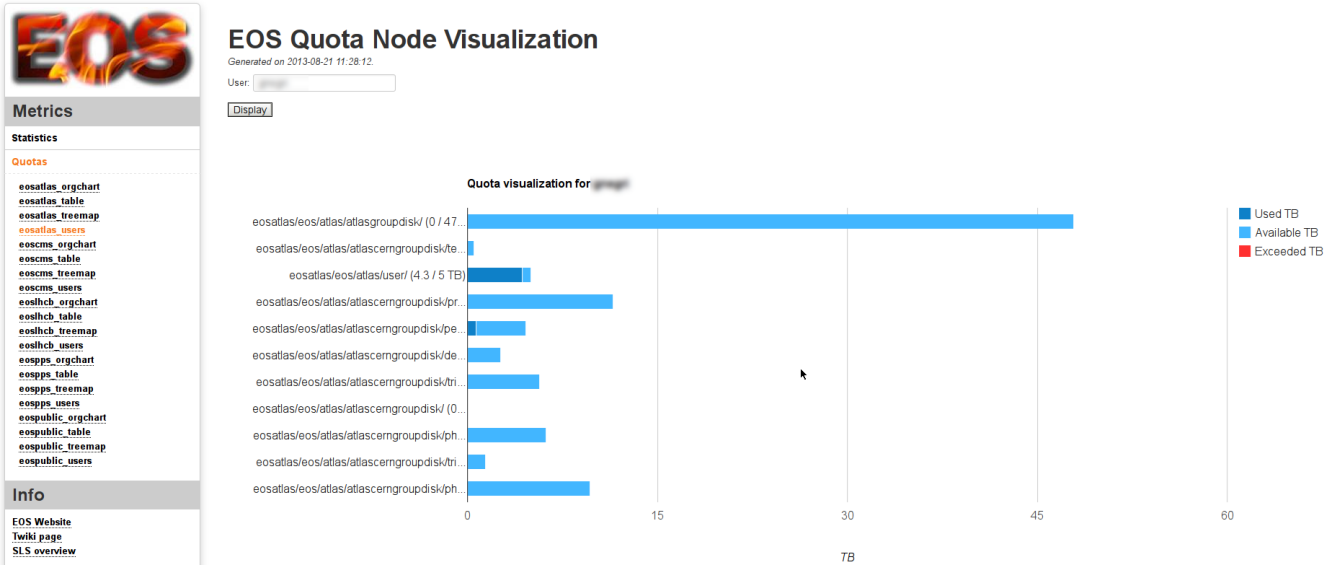

Figure 10: EOS Cockpit

References

[1] CERN TWiki. `https://twiki.cern.ch/twiki/bin/view/EOS`.

[2] EOS Cockpit. `http://eoscockpit.cern.ch/`.

[3] EOS Permission System. `https://eos.cern.ch/index.php?option=com_content&view=article&id=75&Itemid=79`.

[4] Google Charts API. `https://developers.google.com/chart/`.