B2Safe module administrator guide

Table of Contents

Updated to the version 2.2.1 of the module.

Architecture

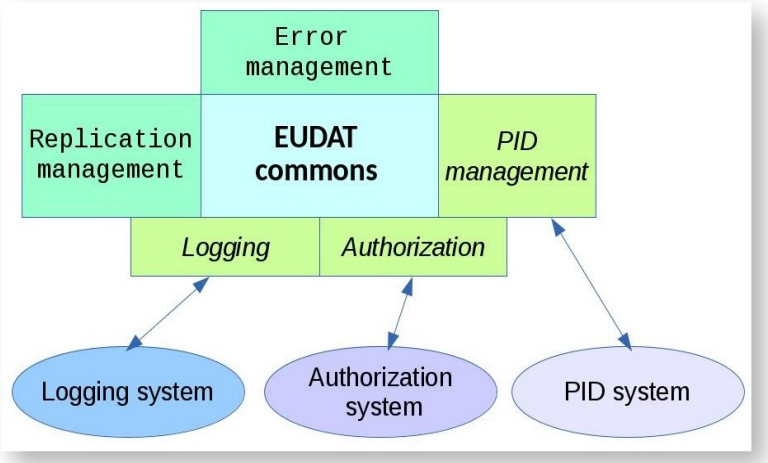

Figure 1: module architecture

The architecture of the module is organized into small sub-modules. It includes the following major components:

1. EUDAT commons

It is a rule set containing the basic functions for processing replication and control-files of each single data object.

2. Logging

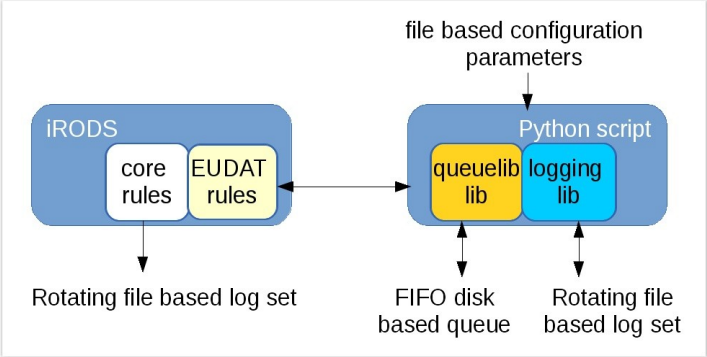

Figure 2: logging

It is built on two different data structures: “queue” and “log”, and based on the python-script `/<irods_home>/modules/B2SAFE/cmd/log.manager.py` to do different operations (push, pop, queuesize, log) on the aforementioned data structures. This logging mechanism is used for sorting out only the important information, supporting the monitoring of the data transfers and of the pid management operations. It works as a replication tracking system, which allows to manage the failed transfers in an asynchronous way.

3. Authorization

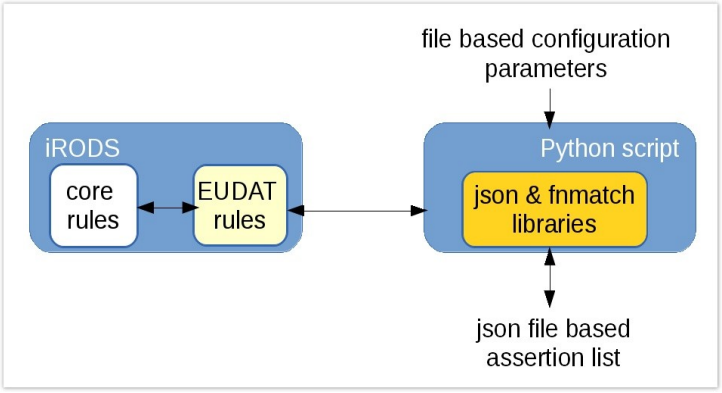

Figure 3: authorization

The authorization decision is based on the file "*authz.map.json*", which contains triplets (subject, action, target) called assertions: they represent who are allowed to do what.

4. PID management:

This rule set contains the rules to perform the main operations related to the PID service. It is based on the script `<irods_home>/modules/B2SAFE/cmd/epicclient.py`; which relies on the configuration file `<irods_home>/modules/B2SAFE/conf/credentials`.

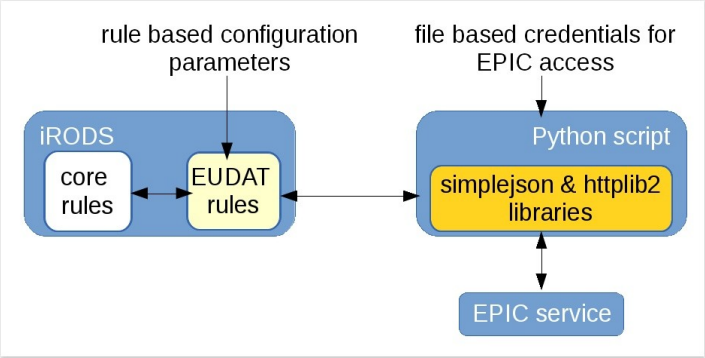

Figure 4: PID management

5. Replication Management

This rule set includes functions used to transfer a single data object or a whole collection, together with integrity check mechanisms.

6. Error Management

The last rule set includes functions able to process errors during data transfer. It can detect errors about checksum and size inconsistencies, PID processing and data object ownernership.

Replica integrity verification

By the integrity verification we mean computing and verifying checksums of replicas of the data objects. (Note: term digest would be more appropriate than checksum, but we follow term used in iRODS). The checksum is computed initially while data ingest (e.g. just after iput) and replication (e.g. in B2SAFE replication is done automatically using iRODS rules) and stored in the iCAT database. The first computed checksum is the referential one and it is also stored in PID service.

The checksums may be also re-computed and verified afterwards at least in two cases:

1. periodically (e.g. every 2 years in order to detect any storage media corruption)
2. on demand (e.g. when the data is migrated to another storage)

In both cases the suitable policy will depend on physical storage (type, configuration, etc.) and availability of computational resources required for re-computing potentially large amount of data, thus it must be decided by data center administrators. This policy is called "local verification policy" later on. The "global" part of replica integrity verification workflow includes exchange of checksum related metadata between PID service and iCATs.

The mechanism of the local verification policy is implemented as shown on the below figure.

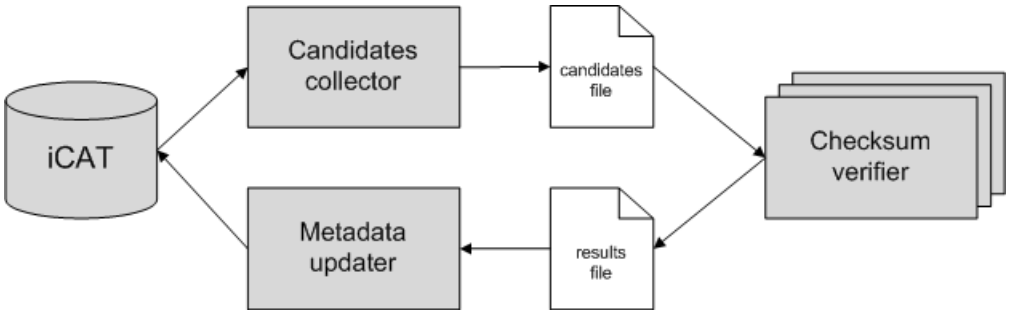

Figure 5: the local verification flow

The mechanism consists of the following functional blocks:

- **iCAT** database contains metadata describing location of the physical replicas, checksums and time of the last checksum verification (format of checksum related metadata is described in a following paragraph).
- **Candidates collector** selects candidate data replicas from the iCAT. The selection is based on resource and time of the last verification. This module is implemented as `eudat-get-checksum-verify-candidates` script. The output of the module is **candidates file**.
- **Checksum verifier** performs the verification of replicas taken from candidates file and outputs to the **results file**. The candidates file is removed. This is a storage specific module and thus will have many implementations. It may be run on iRods server or on underlying storage server or partially on both. Two reference implementations will be provided: `eudat-verify-checksums-disc` (to be run on iRods server and local discs) and `eudat-verify-checksums-hsm` (to be run on HSM server on a NFS share provided for iRods server).
- **Metadata updater** reads results file and updates the iCAT metadata (time of verification) of verified objects or reports the problem if the verification failed. The results file is removed. This module is implemented as `eudat-update-checksums` script. The script also calls the rule updatePidChecksum.r (see below), which updates the checksum in the PID record using iRODS rules implemented in the rulebase of the B2SAFE module.

Authentication (experimental)

The module in the directory `<irods_home>/modules/B2SAFE/scripts/` includes a set of python scripts which allow to get user information from a remote source and import them in a cache, local to the EUDAT node executing the scripts and eventually into B2Safe.
At present just the first step depicted in the following pictures is actually available.
It is possible to test it, using the script `remote.users.sync.py` and the configuration file inside the directory "conf". In the directory "test" there is an example of json file, implementing the local cache. The remote source considered so far is Unity (http://www.unity-idm.eu/), since it is part of the EUDAT AA infrastructure.

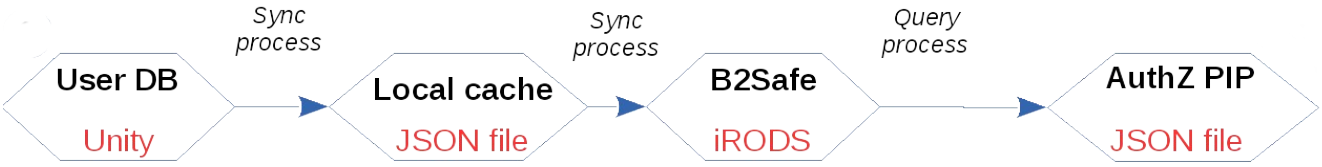

Figure 6: user account synchronization process

Install

See <module_home>/install.txt .

Configuration

Authorization

- iRODS 3.3.x is required.
- in case you are using the version 3.3.0 please apply the patch in "rsExecCmd.patch" placed the dir "patches", using the unix “patch” command:
  ```
  $ patch <  rsExecCmd.patch
  ```
- in the rule file "*eudat.re*": there are two rules called "*EUDATAuthZ*" and "*getAuthZParameters*". The "*EUDATAuthZ*" calls an external python script placed in `<irods_home>/server/bin/cmd` and called "*authZ.manager.py*". Which requires a configuration file placed in `<irods_home>/modules/B2SAFE/conf` and called "*authz.map.json*". The script provides just a couple of methods: "test" and "check", which returns a boolean value of True if the authorization is granted, False otherwise. The authorization decision is based on the file "*authz.map.json*", which contains triplets (subject, action, target) called assertions. So, for example, passing to the script in input a request like:
  ```
  testuser#testzone,
  read,
  /<irods_home>/modules/B2SAFE/conf/credentials
  ```
  It will be accepted if the json file contains:
  ```
  "assertion 1":
    { "subject":
       [ "testuser#testzone" ],
      "action":
       [ "read" ],
      "target":
       [ "/<irods_home>/modules/B2SAFE/conf/credentials" ]
    }
  ```
  Or even:
  ```
  "assertion 1":
     { "subject":
        [ "*#testzone" ],
       "action":
        [ "read" ],
       "target":
        [ "/<irods_home>/modules/B2SAFE/conf/*" ]
     }
  ```
  Because it supports the wild characters in the same way a shell do.
- in the rulebase file "*core.re*" the hook should be configured using the patch "*corere.patch*" placed in the folder "patches" of the module, using the unix “patch” command:
  ```
  $ patch <  corere.patch
  ```
- The entry point for rules specific for certain external executables should be called inside the "*EUDATAuthZ*" as fall back.

- The assertions are parsed in sequence till the conditions of the request are matched, hence those remainng after that one who matches are skipped. In other words, the first assertions (sorted top-down) have higher priority than the following ones. So for example, in this case:

```
    "assertion 1":
       { "subject":
          [ "red#testzone" ],
         "action":
          [ "/<irods_home>/server/bin/cmd/analyze" ],
         "target":
          [ "/testzone/projectA/collection1" ]
       }
    "assertion 2":
       { "subject":
          [ "blue#testzone" ],
         "action":
          [ "/<irods_home>/server/bin/cmd/analyze" ],
         "target":
          [ "/testzone/projectA/collection2" ]
       }
```

  to the user “blue” will be forbidden to analyze the collection1, but he/she will be allowed to do it with collection2.

Logging

We are considering here `/<irods_home>/modules/B2SAFE/conf/log.manager.conf`. Just configure the logging level (INFO, DEBUG, ERROR) and the path to the logging directory:

```
{
"log_level": "DEBUG",
"log_dir": "/<irods_home>/modules/B2SAFE/log",
}
```

Public profile

In order to publish the basic local B2SAFE configuration we need to know for a site:

- whether this one provides a iRODS service.
- which is its local B2SAFE configuration:
  - its iRODS zone
  - the logical iRODS resource to use
  - and a kind of absolute iRODS path, where EUDAT data is stored (*/iRODSZone/home/*...) ..."

The EUDAT registry creg.eudat.eu can serve this purpose. In creg.eudat.eu, there is the "Services" view, where it is possible to select a specific B2SAFE instance page and update its "Service Extension Properties" section.

At the bottom of the page there are the following "Service Extension Properties":

Name	Value
irods_path	e.g. /vzRZGE/eudat/

irods_zone	e.g. vzRZGE
irods_resource	e.g. cacheResc

The "+" sign "Add Property" allows to add properties to the service.

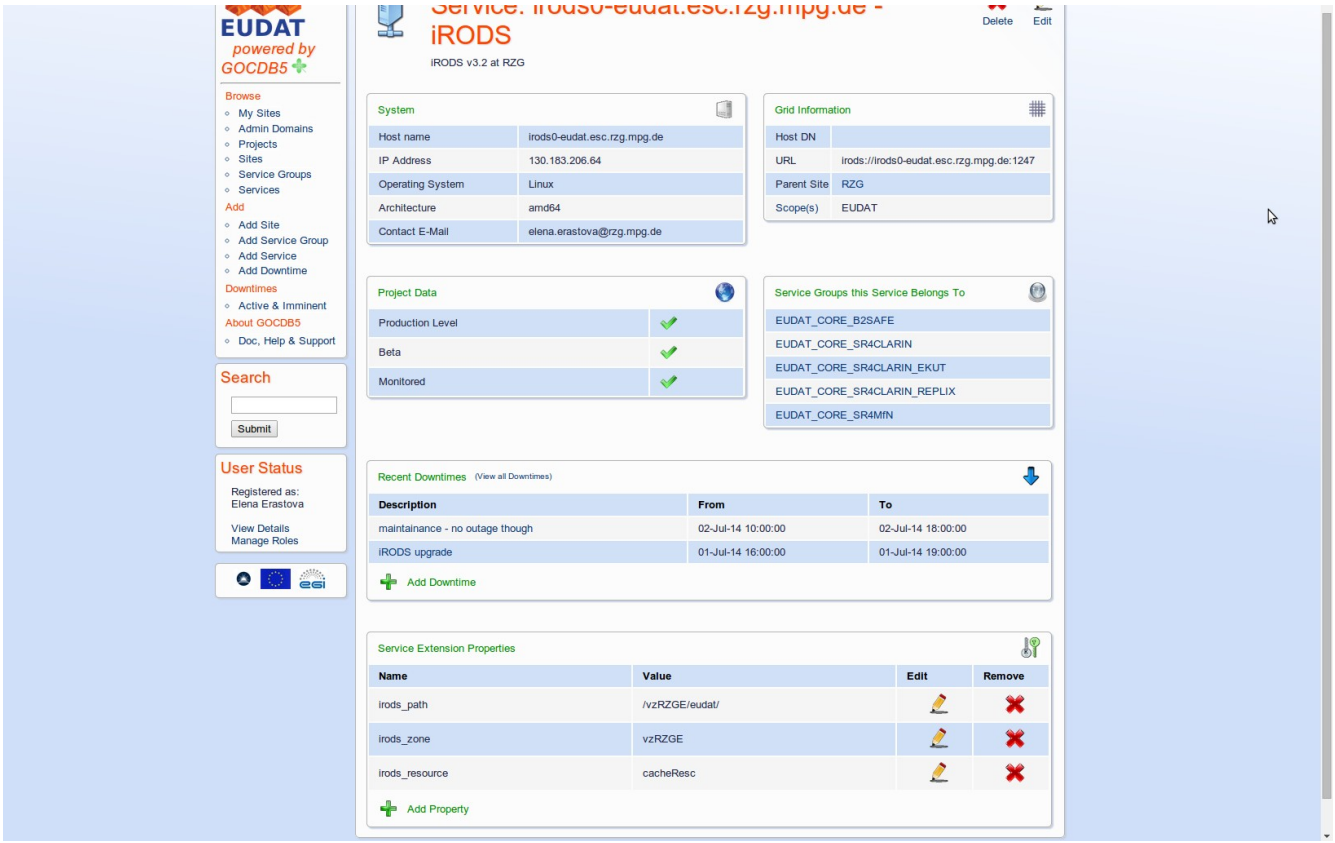

Figure 7: EUDAT service registry (GOCDB)

Replica integrity verification

The scripts described below are place in `/<irods_home>/modules/B2SAFE/scripts/` .

Periodical check

All modules are run from cron daily or several times a day. Option `-d` of `eudat-get-checksum-verify-candidates` script is set to number of days of the period (e.g. 365 if the check is to be performed every year). The mechanism will verify replicas with time of last verification between 365 and 366 days ago.

On demand check

If check of all replicas from a resource is needed, `eudat-get-checksum-verify-candidates` script must be run by the administrator with option `-d` set to 0. Then the rest of the modules must be run either by the administrator or by cron mechanism.

Format of metadata

The mechanism requires information about the time of initial checksum computation or time of its last

verification. While the reference checksum is an attribute of the data object (logical file), the real checksum (and time of its last computation/verification) is an attribute of each data replica (physical file on the resource). The metadata in iCAT is normally connected to data object, not to the replica, thus the name of the resource is part of the metadata attribute. The format is:

```
attribute: eudat_dpm_checksum_date:<Resource>
value: <date of computation>
units:
```

Where <Resource> is name of the resource where the replica is located and <date of computation> is date of its last computation/verification in seconds since Epoch. E.g.:

```
$ imeta ls -d /tempZone/home/rods/test2/testfile.z26952
AVUs defined for dataObj /tempZone/home/rods/test2/testfile.z26952:
attribute: eudat_dpm_checksum_date:demoResc
value: 1402231096
units:
----
attribute: eudat_dpm_checksum_date:hsmResc
value: 1402231333
units
```

Format of checksum verification candidates file

Each line of the file is a single record describing physical file. Each record is a triplet separated with spaces: <path relative to the root of resource> <reference checksum> <date of previous computation in sec since Epoch>. E.g:

```
home/rods/test2/testfile.z26460
sha2:Dw6dY0tp2lvuqMvh0OpfYLMicgGZfGmdzKzCK3G9Kxg= 1402744383
home/rods/test2/testfile.z26500
sha2:VHaQ8qqQEhPOmhbprBjOF/4cA2VSBcj2NScBWlSq3Ms= 1402744383
home/rods/test2/testfile.z26526
sha2:A5td6CeZkhVkBFjSHtifLm9PY7K0176vkkvQNVuyiuU= 1402744383
home/rods/test2/testfile.z26896
sha2:KdjKGXem0GQ8pPKEEvPxrePrc+tYTSBU5SQ0meAoYp8= 1402744383
home/rods/test2/testfile.z26952
sha2:jlVGfGkiPcFMNi5CnMnAjw6+IcPaQ+nxKIHD30vpmiU= 1402744383
```

Format of checksum verification results file

Each line of the file is a single record describing physical file. Each record is a triplet separated with spaces: <path relative to the root of resource> <result of verification> <date of current computation in sec since Epoch>. The result is OK if the actual checksum matches the referential one or ERR else. E.g:

```
home/rods/test2/testfile.z25660 OK 1403004246
home/rods/test2/testfile.z26500 OK 1403004246
home/rods/test2/testfile.z26526 OK 1403004246
home/rods/test2/testfile.z26896 ERR 1403004246
home/rods/test2/testfile.z26952 OK 1403004246
```

In the above example checksum of "home/rods/test2/testfile.z26896" doesn't match, the replica might be spoiled and the problem should be comunicated to the administrator.

Usage of scripts

Usage of eudat-get-checksum-verify-candidates

```
Usage: eudat-get-checksum-verify-candidates [-r <iRods_resource>] [-d
<valid_days>] [-l <log_level>] <candidates_file>
       eudat-get-checksum-verify-candidates -h | --help

The script queries iCAT and gets list of physisical data replicas
that are candidates to verify checksum.
The result is written to <candidates_file>
Only replicas located on <iRods_resource> and those having checksum
verified before <valid_days>.
Amount of logging info is controlled by <log_level>. Allowed values
are 0-6.

Defaults:
  <iRods_resource> = demoResc
  <valid_days> = 365
  <log_level> = 3
```

Usage of eudat-update-checksums

```
Usage: eudat-update-checksums [-r <iRods_resource>] [-i
<input_results_dir>] [-l <log_level>] <root_path_of_the resource>
       eudat-update-checksums -h | --help

The script reads files from <input_results_dir> in order of their
creation.
The files contain results of the checksum verification, including
verification time.
The script sets the new time in iCAT.
Amount of logging info is controlled by <log_level>. Allowed values
are 0-6.

Defaults:
  <iRods_resource> = demoResc
  <input_results_dir> = <root_path_of_the resource>/../checksum-
verify-results
  <log_level> = 3
```

Usage of eudat-verify-checksums-disc

```
Usage: eudat-verify-checksums-disc [-i <input_candidates_file>] [-o
<output_results_dir>] [-l <log_level>] <root_path_of_the resource>
       eudat-verify-checksums-disc -h | --help

The script reads input file with candidates to verify checksum
<input_candidates_file>,
recalculates the checksums and writes verification results to a file
in <output_results_dir> directory.
The paths of the candidate files are relative to <root_path_of_the
resource>.
Amount of logging info is controlled by <log_level>. Allowed values
are 0-6.

Defaults:
  <input_candidates_file> = <root_path_of_the resource>/../checksum-
verify-candidates
  <output_results_dir> = <root_path_of_the resource>/../checksum-
verify-results
```

Usage of eudat-verify-checksums-hsm

```
TO BE DONE
```

updatePidChecksum.r

updatePidChecksum.r

```
updatePidChecksum{
    EUDATSearchPID(*path, *pid);
    if ( str(*pid) == "empty" ) {
        EUDATePIDcreate(*path, *pid);
    }
    logInfo("DPM CHECKSUM update related to PID *pid");
    getEpicApiParameters(*credStoreType, *credStorePath, *epicApi,
                         *serverID, *epicDebug);
    msiExecCmd("epicclient.py","*credStoreType *credStorePath modify
               *pid CHECKSUM 'none'", "null", "null", "null", *out);
    EUDATiCHECKSUMget(*path, *checksum);
    msiExecCmd("epicclient.py","*credStoreType *credStorePath modify
               *pid CHECKSUM *checksum", "null", "null", "null",
               *out);
    msiGetStdoutInExecCmdOut(*out, *response);
    logInfo("EUDATeCHECKSUMupdate -> modify handle response =
            *response");
}
```

```
INPUT *path=""
OUTPUT ruleExecOut
```

Changelog

See <module_home>/changelog.txt

API (EUDAT rules)

Commons

EUDATiCHECKSUMretrieve(*path, *checksum)

Get an existent checksum from iCAT

EUDATiCHECKSUMget(*path, *checksum)

Get, if exist or create if not, a checksum from iCAT

EUDATgetObjectTimeDiff(*filePath, *age)

Calculate the difference between the creation time and the modification time of an object (in seconds).

EUDATfileInPath(*path,*subColl)

Check if a file is in a given path

EUDATCreateAVU(*Key,*Value,*Path)

Create a metadata triplet on iCAT

EUDATiDSSfileWrite(*DSSfile)

Create an object containing a list of PIDs related to a specific collection

Logging

EUDATLog(*message, *level)

Log an event

EUDATQueue(*action, *message, *number)

Log a failure to a FIFO queue

Authorization

EUDATAuthZ(*user, *action, *target, *response)

Authorization policy decision point

PID management

EUDATCreatePID(*parent_pid, *path, *ror, *iCATCache, *newPID)

Create PID

EUDATSearchPID(*path, *existing_pid)

Search PID

EUDATSearchPIDchecksum(*path, *existing_pid)

Search PID by checksum

EUDATUpdatePIDWithNewChild(*parentPID, *childPID)

Update PID record field 10320/LOC

EUDATGetRorPid(*pid, *ror)

Get PID record field RoR's value

EUDATeiPIDeiChecksumMgmt(*path, *PID, *ePIDcheck, *iCATuse, *minTime)

Replication management

EUDATUpdateLogging(*status_transfer_success, *path_of_transfered_file, *target_transfered_file, *cause)

Log a transfer event to the log file and, if it is a failure, to the FIFO queue

EUDATCheckError(*path_of_transfered_file,*target_of_transfered_file)

Perform error checks about the transfer

EUDATTransferSingleFile(*path_of_transfered_file,*target_of_transfered_file)

Transfer a single file

EUDATTransferUsingFailLog(*buffer_length)

Retry to perform a certain number of failed transfers queued in the FIFO queue

EUDATCheckReplicas(*source, *destination)

Check whether two files are available and identical and trigger replication if they are not

EUDATTransferCollection(*path_of_transfered_coll,*target_of_transfered_coll,*incremental,*recursive)

Error management

EUDATCatchErrorChecksum(*source,*destination)

Catch error with Checksum

EUDATCatchErrorSize(*source,*destination)

Catch error Size of file

EUDATProcessErrorUpdatePID(*updfile)

Process error update PID at Parent_PID. It will be processed during replication_workflow, called by updateMonitor.

EUDATCatchErrorDataOwner(*path,*status)

Catch error Data Owner if user is not owner of Data from *path

Create or update a PID, including checksum

EUDATiPIDcreate(*path, *PID)

Create a PID as iCAT metadata

EUDATiFieldVALUEretrieve(*path, *FNAME, *FVALUE)

Get a metadata value from iCAT

EUDATePIDcreate(*path, *PID)

Create a PID as EPIC service record

EUDATePIDsearch(*field, *value, *PID)

Search a PID on the EPIC service

EUDATeCHECKSUMupdate(*PID)

Update the PID record field checksum

EUDATeURLupdate(*PID, *newURL)

Update the PID record field URL

EUDATePIDremove(*path)

Delete a PID

EUDATeiPIDeiChecksumMgmtColl(*sourceColl)

Walk through the collection. For each object, it creates a PID and stores its value and the object checksum in the iCAT.

EUDATiRORupdate(*source, *pid)

Add the ROR field of the PID of the object to iCAT

EUDATeParentUpdate(*PID, *PFName, *PFValue)

Update the EUDAT ROR or PPID field in the PID record

Transfer a whole collection

Best Practices

Authorization

If you want to implement an ACL for the execution of an external command, such as a python script, a C code executable or a shell command, you can use the iRODS hook:

```
acPreProcForExecCmd(*cmd, *args, *addr, *hint) {
    if (*cmd != "authZ.manager.py") {
        EUDATAuthZ("$userNameClient#$rodsZoneClient",
                   *cmd, *args, *response);
    }
}
```

This hook can be put in the ruleset `<irods_home>/server/config/reConfig/core.re` .

Then in the file `<irods_home>/modules/B2SAFE/conf/authz.map.json` can be added

the suitable assertions. So for example if the objective is to implement:

```
Only user guybrush#MIslandZone can execute the python script
<irods_home>/server/bin/cmd/drink_grog.py
```

Then just add the following assertion in the authorization map:

```
{ "subject": [ "guybrush#MIslandZone" ],
  "action":  [ "<irods_home>/server/bin/cmd/drink_grog.py" ],
  "target":  [ "*" ]
}
```

But if you want a more fine-grained ACL, you can also specify the allowed input arguments:

```
Only user guybrush#MIslandZone can execute the python script
<irods_home>/server/bin/cmd/drink_grog.py -in acid_battery
```

```
{ "subject": [ "guybrush#MIslandZone" ],
  "action":  [ "<irods_home>/server/bin/cmd/drink_grog.py" ],
  "target":  [ "-in acid_battery" ]
}
```

In principle, the same mechanism can be applied directly to filter the execution of every rule. For example, adding a line before the rule invocation in this way:

```
acPostProcForPut {
    EUDATAuthZ("$userNameClient#$rodsZoneClient",
               "EUDATTransferSingleFile", “*”, *response);
    EUDATTransferSingleFile(*path,*replicaPath);
}
```

And the related assertion in the map:

```
{ "subject": [ "user#CompanyZone" ],
  "action":  [ "EUDATTransferSingleFile" ],
  "target":  [ "*" ]
}
```

However the authorization mechanism implies a certain overhead so it should be used carefully.

Replication

In order to replicate a single file the best option is the rule:

```
EUDATTransferSingleFile(*path_of_transfered_file,
                        *target_of_transfered_file)
```

contained in the ruleset `<B2SAFE_module>/rulebase/replication.re` .
This rule already includes integrity checks (based on object's size and checksum), proper registration and update of PIDs associated to the source and to the target, and a logging mechanism to track replication errors.

In case the replication source is a collection, then it is more suitable the following rule:

```
EUDATTransferCollection(*path_of_transfered_coll,
                        *target_of_transfered_coll,
                        *incremental,*recursive)
```

Which is included in the aforementioned ruleset.
It relies on `EUDATTransferSingleFile`, hence it has its same features.

However this rule is quite demanding in terms of memory consumption, therefore in case of collection containing more than 1000 objects, it is advised to execute it in delayed mode (see <B2SAFE_module>/rules/testtransfercollection.r), for example:

```
delay("<EF>1s REPEAT UNTIL SUCCESS OR 10 TIMES</EF>") {
        EUDATTransferCollection(*Path,*replicaPath,
                                bool("true"),bool("true"));
}
```

Note the two boolean flags set to true which mean respectively “perform it in a incremental way” and “perform it in a recursive way”. The different options available for the delay mode are listed here: https://irods.sdsc.edu/index.php/Rule_Execution_modes .
In this way it will be possible to recover the replication, even after a failure due to out of memory errors.

Performance

In order to improve the performance in case of multiple processes requesting iRODS resources is advisable to change the following line in the rule set *core.re*:

```
# 17) acSetReServerNumProc - This rule set the policy for the number of processes
# to use when running jobs in the irodsReServer. The irodsReServer can now
# muli-task such that one or two long running jobs cannot block the execution
# of other jobs. One function can be called:
#    msiSetReServerNumProc(numProc) - numProc can be "default" or a number
#    in the range 1-4. numProc will be set to 1 if "default" is the input.
#
#acSetReServerNumProc {msiSetReServerNumProc("default"); }
```

With the next one:

```
acSetReServerNumProc {msiSetReServerNumProc("4"); }
```

But be aware that on irods 3.3 you need to apply a patch to the file <irods_home>/server/core/src/reServerLib.c. Otherwise it will not work properly. For the patches, see irods chat: https://groups.google.com/forum/#!msg/irod-chat/IGYRQdNk9QE/FqcMct5OY-UJ ("intermittend stat errors since the enabling of 4 ruleservers").